CS 744: SPARK STREAMING

Shivaram Venkataraman
Fall 2019
ADMINISTRIVIA

- Midterm grades this week
- Course Projects sign up for meetings $\rightarrow N_0, 2, 1, 25$
Scalable Storage Systems

Datacenter Architecture

Resource Management

Computational Engines

Applications

Machine Learning  SQL  Streaming  Graph

Dataflow model
- Queries
Properties of Data, Queries
CONTINUOUS OPERATOR MODEL

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

Diagram:
- Driver -> Control Message
- Task -> Network Transfer

Naiad
Flink
CONTINUOUS OPERATORS

Resource / Memory overhead

Upstream backup

Strict ordering

Synchronization might not mitigate stragglers

Recovery is very fast

Mutable state differences across replicas

Order of data

Mutable state

Node 1

Node 2

Node 1

Node 2

Synchronization

Count by (country) Pageview

Count by (country) Pageview
SPARK STREAMING: GOALS

1. Scalability to hundreds of nodes
2. Minimal cost beyond base processing (no replication)
3. Second-scale latency
4. Second-scale recovery from faults and stragglers
DISCRETIZED STREAMS (DSTREAMS)

- Short, deterministic tasks
- Stateless operation
- Replicate/recovery operation
- Run ops in parallel
- Each batch can be run in parallel
- Recovery or straggler mitigation
- Low latency recovery

Dependencies within across timestamp

Stages:
1. **Batch operation**
2. **Immutable dataset**
3. **D-Stream**
```scala
pageViews = readStream(http://..., "1s")
ones = pageViews.map(event => (event.url, 1))
counts = ones.runningReduce((a, b) => a + b)
```
ARCHITECTURE

Spark Streaming
- divide data stream into batches
- stream input data stream
- batches of input data as RDDs
- generate RDD transformations

Spark
- Task Scheduler
- Memory Manager
- Spark batch jobs to execute RDD transformations

FS
- read FS every 1s before next force
- count() for each
- stream input batches
- write FS every 1s and check for new files?

Spark Streaming
- socket, filesystem
- DStream
- convergence
- recursive directory
DSTREAM API

Transformations ➞ on DStream

Stateless: map, reduce, groupBy, join

Stateful:

- window("5s") ➞ RDDs with data in [0,5), [1,6), [2,7)
- reduceByWindow("5s", (a, b) => a + b)

map ➞ Stream

1:1 mapping

sliding window

operate across windows
SLIDING WINDOW

Add previous 5 each 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")
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
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 \textit{in timestep} and \textit{across timesteps}
pageViews = readStream(http://..., "1s")

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

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

Straggler Mitigation

Use speculative execution

Task runs more than 1.4x longer than median task → 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).
DISCUSSION

https://forms.gle/xUvzC1bdV7H48mTM8
Double time step \( \Rightarrow \) double throughput

Larger window \( \Rightarrow \) more throughput

Cluster Throughput (GB/s)

- **Grep**
  - Linear scaling

- **WordCount**
  - 1 sec vs. 2 sec
  - 3.93x more throughput

- **TopKCount**
  - 1 sec vs. 2 sec
If the latency bound was made to 100ms, how do you think the above figure would change? What could be the reasons for it?

- Throughput might decrease
- Overhead of creating RDDs
- Take more time to process
- Linear scaling might not work

驱动器非常短 | 100ms
---|---
Memory management | 100ms
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: low latency
- Incremental output
- Applications that might have fewer stragglers

- Spark Streaming: lower latency
- Recovery
- API simple
Next class: Graph processing
Sign up for project check-ins!
SHORTCOMINGS?

Expressiveness
- Current API requires users to “think” in micro-batches

Setting batch interval
- Manual tuning. Higher batch → better throughput but worse latency

Memory usage
- LRU cache stores state RDDs in memory
COMPUTATION MODEL: MICRO-BATCHES

Micro-Batch:

Driver → Control Message

Task

Network Transfer
SUMMARY

Micro-batches: New approach to stream processing

Higher latency for fault tolerance, straggler mitigation

Unifying batch, streaming analytics