Hi

CS 744: SPARK STREAMING

Shivaram Venkataraman
Fall 2021
ADMINISTRIVIA

- Midterm grades this week ➔ Friday
- Course Projects feedback ➔ Today
- Google Cloud Credits ➔ $50 / student email address
  
  Private Piazza / e-mail
CONTINUOUS OPERATOR MODEL

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

Driver -> Control Message
Task -> Network Transfer

Map to extract user Id
Sum of events/userId

Avoid stragglers with good engineering

Flink
Naiad
CONTINUOUS OPERATORS

- Replicate every operator to another machine
- Minimizes recovery time
- 2x the resources
- Replicas remain in-sync
- Replication protocol which also adds overhead

Diagram:
- Input flows into node 1 and node 1', which are in sync.
- Node 1 and node 1' send to node 2 and node 2' respectively.
- Node 2 and node 2' have mutable state that needs synchronization.
- Primaries and replicas ensure state consistency across machines.
SPARK STREAMING: GOALS

1. Scalability to hundreds of nodes $\rightarrow$ high throughput

2. Minimal cost beyond base processing (no replication) $\rightarrow$ resource efficiency

3. Second-scale latency = time between input arriving to when it is part of the output

4. Second-scale recovery from faults and stragglers
DISCRETIZED STREAMS (DSTREAMS)

- Divide time into a number of micro-batches
  - 18 as micro-batch
  - \( t = 18 \) to \( 2s \) \( \rightarrow \) all events

- Run the batch operation on the input events, and save output (operator state) also as an RDD

- Next micro-batch, use the output from previous and compute

\[ \text{input events} \rightarrow \text{batch operation} \rightarrow \text{operator state} \rightarrow \text{output from previous and compute} \]
will this be the same

Processing time stamps / not event timestamps
pageViews = readStream(http://..., "1s")

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

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

running sum of url events
DSTREAM API

Transformations

Stateless: map, reduce, groupBy, join → very similar to RDD API

Stateful:

window(“5s”) → RDDs with data in [0,5), [1,6), [2,7)

reduceByWindow(“5s”, (a, b) => a + b)

Sliding

form a window of 5s and reduce using sum
SLIDING WINDOW

(a) Associative only

(b) Associative & invertible

Add previous 5 each time

Computation is cheaper
STATE MANAGEMENT

Tracking State: streams of (Key, Event) \(\rightarrow\) (Key, State)

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

- User defined state object for every key
- User defined state object for every key
- User defined state object for every key

Timeout to delete old states

old state new event

update my state based on event that arrived
SYSTEM IMPLEMENTATION

Master
- D-Stream lineage
- Input tracker

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

Client

Tracks D-Streams relate to each other

Exist in Spark

Input tracker

RDD lineage

Task scheduler

Block tracker

Input receiver

Task execution

Block manager

Comm. Manager

Replication of input & checkpointed RDDs

RDD in memory of 2 machines

New

Modified

Poll http server read from FS/storage system
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

stores in HDFS

§ ¥ £ ☐

£ ☐ p,

stored in HDFS
**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*

Note
- Only recompute state / tasks which were lost
pageViews = readStream(http://...,
        "1s")

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

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

EXAMPLE

[Diagram showing data processing]

\textit{can be run in parallel}
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).

Similar in spirit to HFS recovery
SUMMARY

Micro-batches: New approach to stream processing

Simplifies fault tolerance, straggler mitigation

Unifying batch, streaming analytics
DISCUSSION

https://forms.gle/4Xbu9y9KTW5qph8H8
If the latency bound was made to 100ms, how do you think the above figure would change? What could be the reasons for it?

Overhead increases with cluster size.

Every microbatch:

\[ \text{Fixed overhead} + \text{Time to do Grep/Word Count} \]

\[ \sim 2s \]  

\[ 100 \text{ms} \]

\[ \text{Bigger factor} \]

Small microbatch \( \Rightarrow \) lots of RDDs  
\( \Rightarrow \) more metadata  
\( \Rightarrow \) tracking/scheduling tasks
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 / Flink
- low latency / quick processing

Spark Streaming
- fault recovery
NEXT STEPS

Next class: Graph processing!
Midterm grades ASAP!