Hurricane: Distributed real-time data-processing

Jeffrey Warren, Vedha Sayyaparaju, Vikas Velagapudi, Zack Drach
{jtwarren, vedha, vvelaga, zdrach} @mit.edu

Demo link: https://www.youtube.com/watch?v=FmS21saPdkY
Code: https://github.com/vedharaju/hurricane

Overview/Abstract

Hurricane is a distributed real time data processing system. It allows for distributed in-memory computation while retaining the fault tolerance of data flow models like MapReduce. At a high level, Hurricane performs chained MapReduce jobs in small batches at a high frequency (about once per second). Intermediate results are stored in read-only Resilient Distributed Datasets (RDDs), sharded across multiple worker nodes. Although the implementation is unoptimized, Hurricane achieves throughput of 50k records/second/node.

Hurricane runs with a single master node and multiple worker nodes. The system reads data from a persistent, streaming, log-file storage system such as Kafka. The master reads the log data in batches and then launches tasks on workers to process each batch and generate relevant RDDs. The master tracks and persists all RDDs and their dependencies.

Hurricane is able to survive faults on master and worker nodes. Fault-tolerance on the master is achieved by persisting state to disk so that it may resume after coming back online. Fault-tolerance on worker nodes is achieved by reconstructing RDDs from their sources. RDDs may be replicated across several workers to speed up the recovery.

Workflows are defined on the master using a custom text-based syntax. Individual jobs are user-defined-functions (UDFs) that can be implemented in any language. Hurricane's workflow semantics are more powerful than MapReduce, allowing for constructions such as efficient windowed aggregation.

Design & Implementation

Hurricane's design was motivated by the design of Storm and Spark Streaming. Hurricane is implemented in the Go programming language. There is a single unreplicated master node and several replicated worker nodes. The master node is responsible for maintaining the configuration and current state of the system. The worker nodes are responsible for executing tasks assigned to them by the master and storing data.

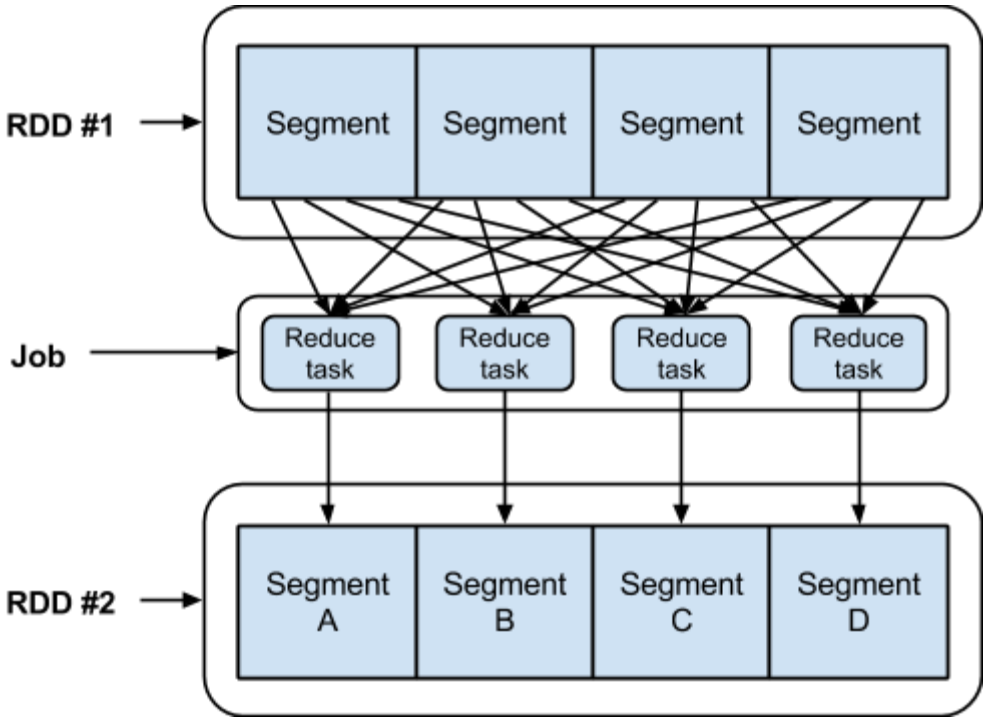

Figure 1: An overview of the system showing two RDDs with 4 segments each. Each segment may be stored on a separate worker. A reduce job is being run over the data and creating a new RDD.

Master

The master is implemented with a single-threaded event loop that sends asynchronous requests to worker nodes. The master knows the location of each worker, what each worker is working on, and what each worker has stored in its memory. For the sake of fault-tolerance, the master stores its state on disk transactionally via PostgreSQL.

Data Structures

The master keeps track of several types of data: RDDs, Jobs, Batches, and Workflows (which consist of ProtoJobs).

RDD

An RDD is an unbounded list of tuples and is typically created as the output of a job. The tuples are stored in memory and sharded across worker nodes into segments.

The master is responsible for storing the metadata associated with the RDDs including the ID, the workers that contain the RDD, references to the job that created the RDD, etc.

Segment

A segment is internally broken up into multiple partitions based on the output partitioning of the job that created it.

The master is responsible for storing the metadata associated with each segment including the ID, a reference to the RDD that the segment belongs to, a reference to the task that created the segment, a reference to the worker(s) where the segment resides, etc.

Task

A task is the smallest unit of work that can be done on a single worker. Each task takes in one or more segment partitions as input and outputs a single segment.

The master is responsible for storing the metadata associated with each task including the ID, status of the task (pending, complete), etc.

Job

A Job is an instantiation of a Map-Reduce job with one or more input RDDs. The output of a job consists of multiple segments.

The master is responsible for storing the metadata associated with each job including the ID, a list of the task IDs, a reference to the RDD where the output is stored, a reference to the workflow batch, a reference to the ProtoJob definition, etc.

Protojob

A ProtoJob contains the UDF that gets executed to run a job, as well as metadata related to partitioning and replication.

The master is responsible for storing the metadata associated with each protojob including the ID, references to the commands, a reference to the workflow, indexes of the partition fields for inputs and outputs, replication factor, etc.

Workflow

A workflow is a directed graph of ProtoJobs. The jobs of a workflow are defined by a custom syntax which is defined below.

The master is responsible for storing the metadata associated with each workflow including the ID, a reference to the source ProtoJob, etc.

Workflow batch

Each workflow gets executed once for each batch of data that goes through the system.

The master is responsible for storing the metadata associated with each workflow batch including the ID, the start time, the duration, a reference to the protojob, etc.

Worker nodes

The master keeps track of the status of each worker node.

The master is responsible for storing the metadata associated with each worker batch including the ID, the URL, a reference to the RDDs on the worker, etc.

Worker

Workers may be started and added to the system at any point in time. They register to the master node by sending an RPC to the master node. Workers receive RPC commands from the master node to execute tasks. The command contains the locations of all of the source partitions, the ID of the output segment, and all other job-related information. When the job is complete, the worker responds to the master with a success notification, and the master records the location of the new destination segment.

LRU Cache

The worker maintains an LRU cache for storing segments. The size of the LRU cache for each worker is specified on startup. When the capacity of the cache has been exceeded and new segments need to be stored, the least recently used segment is evicted from memory and written to a file on disk. We used the golang gob library to handle serialization and deserialization as segments move between memory and disk. A more efficient implementation would only write the segments that are checkpointed, for now we write everything to disk.

UDFs

User Defined Functions (UDFs) are external programs which read serialized tuples from stdin and output them to stdout. UDFs are distributed to workers via an external program, such as NFS. In our implementation we compile all UDFs directly on the worker nodes.

Workflow Syntax

The master parses a special syntax to generate a workflow. Following is a simple aggregation example:

```
d=1000

JOBS

A: @/src/demo/huge/input.udf \I \D 5 ;; r=false & p=(0,1) & w=10 & b=10
B: @/src/demo/huge/reduce.udf ;; r=true & p=() & w=5 & b=1 & c=1
C: @/src/demo/huge/output.udf ;; r=false & p=() & w=1 & b=1

WORKFLOW

A -> B
B, C 1, B 2 -> C
```

Following is a description of the various job parameters:

- The duration “d” of each batch is specified in milliseconds.
- The reduce flag “r” determines how partitions are distributed among tasks.
- The partition string “p” determines which tuple index is used to partition the job output.
- The worker number “w” determines the parallelism of each job.
- The bucket number “b” determines the number of partition buckets in each segment.
- The copies number “c” determines the number of additional workers should replicate the output segment
- The command line flags “\S” and “\D” specify the start and end times for reading log data from a Kafta broker

The workflow is a directed graph. Multiple inputs for a single job are separated by commas. Since a job might depend on an RDD generated in a previous batch, the number immediately after the input name determines the delay of that input edge. This allows Hurricane to express constructions such as windowed aggregation.

Fault tolerance

Fault tolerance is implemented for both the master and worker nodes.

If the master node fails and reboots, it reads its persistent state and re-schedules any outstanding tasks which might not have completed as well as any tasks that should have been scheduled while the master was dead.

If a worker node fails, the master marks that node as dead and lazily reconstructs any segments when they are required by future jobs. This lazy strategy was adopted to prevent excessive work during temporary network partitions. If a worker is partitioned and reconnects before any of its segments are needed, then the recovery procedure is never invoked.

Performance

Hurricane was performance tested by running an aggregated sum of log data on Amazon Web Services (AWS). There was a single master node running on a c3.2xlarge[1] machine. There were 3 worker nodes running on m3.2xlarge[2] machines. All servers were running the Ubuntu[3] operating system.

The performance workflow was composed of 3 jobs. One job created log output (acted as a mock kafka broker) and stored the result in an RDD. The next job computed the running sum of errors and successful log entries. The final job wrote the result to disk.

---

[1] https://aws.amazon.com/ec2/instance-types/#Compute_Optimized
[2] https://aws.amazon.com/ec2/instance-types/#General_Purpose
[3] Ubuntu Server 14.04 LTS

The number of log entries created and processed per batch as well as the batch frequency were tested. The key metrics we were looking at was the latency between start and end time of the jobs as well as the size and growth of the event queue (used for queuing jobs). Additionally, we monitored memory, CPU, and disk usage. Jobs began to backup and the CPU usage increased as the number of log entries increased. In the end, Hurricane was able to process a steady 50k records/node/second with about 20s of latency with no difficulty. This test was run for 10 minutes, long enough to prove the system was stable. We found that the optimal batch frequency was about one batch per second.

During performance testing we also killed a worker node and a master node, subsequently bringing them back up. This resulted an immediate performance hit but the system eventually became stable again.

Spark Streaming is able to process 200k - 400k records/node/second while performing similar computations on similar machines. Our implementation is slower than Spark Streaming for several reasons:

- Hurricane must serialize and deserialize tuples twice when sending them to UDFs. Spark eliminates this overhead by sharing memory with its UDFs.
- Hurricane's network operations uses go's built-in RPC calls, which incur significant serialization and buffering overhead. Spark allows tuples to stream through workers without this overhead.
- For the sake of convenience, Hurricane workers downloads input partitions sequentially. A better implementation would stream them in parallel.

Conclusion

We have designed and implemented a fault-tolerant system capable of processing large amounts of data on a distributed cluster. It is suitable for handling aggregation tasks like counting the number of errors in a log in the last 10 minutes. Although it is highly unoptimized, its performance is within an order of magnitude of the current state of the art.

In the future, we would like to implement some of the optimizations described in the performance section. Additionally, we would like to support stronger semantics during dynamic topology changes. Currently, Hurricane only allows new jobs to be appended to the workflow if they use the output of older jobs in the workflow. Supporting a wider variety of topology changes would increase the flexibility of Hurricane in a production environment.