Distributed ETL

A lightweight, pluggable, and scalable ingestion service for real-time data

Joe Wang
joewang@cs.stanford.edu

ABSTRACT

This paper provides the motivation, implementation details, and evaluation of a lightweight distributed extract-transform-load (ETL) service designed for large scale ingestion of real-time data.

1 INTRODUCTION

Today's world of internet connected devices has shown to generate larger and larger volumes of data. Buried in these massive streams of data are insights many companies would find valuable. For example, the telemetry components in Windows 10 emit data on the health of the system, so that Microsoft can detect and resolve widespread issues users might face. As services grow in size, however, a challenge many companies face is the need to scale their data pipelines to accommodate the ever increasing amounts of data to process.

One part of the pipeline that requires scale is the ingestion component, which is responsible for receiving and processing raw data from varying sources. Traditionally, extract-transform-load (ETL) is the process in many data warehouses, where a front-door receives data in some semi-structured state, it applies some transformations to produce a standardized structured result, and stores the result in a database. Due to the enterprise nature of existing ETL software, most such systems are not designed to scale to the internet.

We propose a service that performs ETL, but is designed to easily scale up in accordance with data volumes.

CS 244B, Distributed Systems
2017.

2 COMPONENTS

To understand the ingestion system as a whole, we first define the components that make up the system.

2.1 Source

Sources are the origination point for data to consume. The most common source might be a subscriber to a publish/subscribe system, such as Apache Kafka or Microsoft Azure EventHub. These systems increase in scale by partitioning their data streams, so we model our service to do the same. In particular, we define an `EventSource` to be the data stream for a single partition from a given `EventSourceFactory`, which is the collection of partitions that make up a logical data stream.

We require these sources to produce data in an ordered and reliable manner, that is, by specifying some "offset" to a partition, the source always returns the same stream of data.

2.2 Processor

We define an `EventProcessor` as any process that takes some input data set of some input type, applies filter, join, and projection operations, and produces some output data set of some output type.

We require the processors to be idempotent, so that the same input data will always produce the same output data.

2.3 Sink

`EventSinks` are the destination to which transformed data is written. This could be a database, some shared filesystem, or a cloud storage provider like Amazon S3 or Azure Storage. A common format used might be Apache Avro, which is a schematized, row-oriented, compact data serialization format typically used in Hadoop oriented big data systems.

3 GOALS

We'd like to achieve the following capabilities with the Distributed ETL system:

Scalability. The system should be able to increase capacity simply by adding additional nodes for processing.

Reliability. The system should continue processing data even while a minority of the nodes fail. The system should not unintentionally drop data during processing, that is, we need *at least once* semantics. Ideally, we would like *exactly once* semantics, where duplicate data should not be written.

Extensibility. It should be easy for add new implementations for sources, processors, sinks, and other components (described later).

Low Maintenance Overhead. The system should be comprised of as few components as required to function.

4 IMPLEMENTATION

The next several sub-sections describe the implementation used in the Distributed ETL project.

4.1 Data Flow

A typical data consists of a single `EventSource`, wired through a single `EventProcessor`, to a single `EventSink`. The `EventSource` makes requests for data to be retrieved at the last processed offset; the `EventProcessor` transforms the retrieved data; the `EventSink` writes the transformed data. Finally, a new offset associated with the processed data is persisted for the partition.

4.2 Workload Management

Running an ingestion service at scale across many nodes requires coordination and persistence of two pieces of data: the assignment of which node should process which partition, and the last offset that has been processed for each partition. Our implementation uses a modified Raft [1] consensus protocol for electing a leader to distribute the workload and keeping offset checkpoints.

We found that the original Raft specification, while is general for any given state machine, was overly complex for our requirements. Instead, we propose the following changes.

4.2.1 State Log. Because the state we maintain is a snapshot of the cluster at a given point in time, we found there was no need to keep a complete log: we always want each node to be at the latest state. Hence, we simply keep the last versions of entries in the state, associated with a monotonically increasing global version number for each mutation. For nodes that fall behind in state, we simply sync the entire state (which is small) via `AppendEntries`.

Because the original `AppendEntries` RPC specification requires rejecting requests whose versions do not match, we need to make a modification to allow the state sync. To the RPC, we add an additional field called `force`. When set to true, log version and term checks are ignored, as we assume the data received will always be the latest.

Making this optimization also removes the need for processes like log compaction.

4.2.2 SynchronizedFollower. Raft uses three states, *Leader*, *Candidate*, and *Follower* to provide state machine replication. Because we also require each node to acquire the latest state before it begins processing, we add a fourth state called *SynchronizedFollower*. The state is equivalent to *Follower* for the Raft protocol, but only transitions on the first accepted `AppendEntries` RPC that was not forced. Upon transition to *SynchronizedFollower*, the node is notified to begin processing.

4.2.3 Losing Quorum. If at any point the cluster loses quorum, Raft is unable to update the partition information. In this scenario, each node will continually retry the update until the cluster is back in quorum, impeding progress. To reduce the amount of unnecessary RPCs, we make an optimization to stop node processing until the cluster has quorum again.

To do so, we maintain a "keep-alive" timer in the program, similar to the election timer, and we also add one more field to `AppendEntries` called `hasQuorum`. When a follower receives a heartbeat for which `hasQuorum=true`, it resets the keep-alive timer. When the keep-alive timer elapses, we assume quorum was lost, and we stop processing.

4.3 RPC

gRPC libraries were used for communication among nodes.

4.4 Sources and Sinks

For evaluating correctness of the system, we supply an event source and event sink.

4.4.1 SampleEventSource. The `SampleEventSource` generates data with the time, a monotonically increasing offset value, the name of the node processing the source, and whether the node was a leader. The values contained in this data stream enables us to verify that all events were in fact correctly processed, and no data was dropped.

4.4.2 PositionedFileEventSink. This writes each event as a single line out to a text file at a configurable path. The sink also maintains a set of checkpoints mapping offsets to file positions, which enables *exactly once* semantics for data processing.

4.5 Scheduler

We implement a very simple scheduler based on consistent hashing; the set of active nodes are placed on a circle, and partitions are assigned based on their hashed locations on the circle.

4.6 Object Pools

Processing large amounts of data in a managed language has the potential for undesirable garbage collection overhead. To mitigate this performance issue, we use a simple object pooling implementation to recycle objects being used.

4.7 Plugin Architecture

To support extensibility, we provide a configuration driven mechanism by which all data flows are setup. All components must implement their respective interfaces and have the attribute `IngressComponent` with a unique name. The configuration thus provides the association between a source, a processor, and a sink.

At runtime, reflection is used to catalog and register with Autofac (an IoC container library) each valid component in the currently assembly, as well as assemblies specified in configurations. For each configuration, Autofac is then used to resolve the appropriate implementations for instantiation.

5 EVALUATION

In evaluating Distributed ETL, the primary characteristic we wish to ensure is correctness. We need to check that work is properly assigned, that when nodes are removed and rejoined, work is properly reassigned, and that the data being produced is valid.

While performance may also be interesting, that is much more dependent on factors not directly related to the partition management system (e.g. network I/O speeds). All in all, the total throughput capability of the system matters much more so than latency.

5.1 Setup

We configure a 5 node cluster with the following parameters:

- Node names:
  - BN4SCH102032018
  - BN4SCH102031924
  - BN4SCH102032017
  - BN4SCH102031919
  - BN4SCH102031619
- Partitions: 10
- RPC timeout: 1000 ms
- Heartbeat interval: 300 ms
- Election timeout: 3000 ms
- Keep-alive timeout: 3000 ms

The `SampleEventSource` emits 10 events every second, and the `PositionedFileEventSink` writes to a network share available to all nodes.

Once the cluster becomes operational in a steady state, we invoke the steps in Table 1. We note that at 31:32, the cluster no longer has quorum, so all progress should pause until 32:16.

5.2 Results

We use the data written out for each partition to visualize the behavior of the cluster. By writing

Table 1

Time	Action
29:15	BN4SCH102032018 (leader) killed
30:15	BN4SCH102031924 (leader) killed
30:46	BN4SCH102032018 restored
31:32	BN4SCH102031919 killed
	BN4SCH102032017 killed
32:16	BN4SCH102031924 restored

out both the partition as well as the node, we can produce graphs of time plotted against offset (representing progress) by the two pivots respectively.

For correctness, we expect:

(1) A linear association between the time and offset for each partition with no missing intermediary values.
(2) At any given point when progress can be made, progress is eventually made.

5.2.1 Partition View. Figure 1 displays the progress made on partition 5 by the cluster. The disconnected line segments corresponding to the 3 series associated with the nodes processing this partition clearly show points in time when cluster configurations change. The work is assigned by default to BN4SCH102031924. When BN4SCH102031924 is killed, we see that the work shifts to BN4SCH102031919. When BN4SCH102032018 is restored, we see that the work is reassigned to BN4SCH102032018. When the cluster loses quorum, work stops. When quorum is restored by adding BN4SCH102031924, work is given back to BN4SCH102031924.

Where the work for any given partition is assigned is simply a function of the scheduler, with which we do not concern ourselves for the purposes of this paper. What's important to see is that the work being performed is correct.

5.2.2 Node View. Figure 2 displays the work performed by node BN4SCH102031919 across 6 different partitions. Like Figure 1, we observe the changing set of partitions being processed, with work stopping when the cluster loses quorum. Curiously, we see that this node does not resume work when the cluster returns to quorum, but this is most likely due to a delay in the scheduler while reassigning partitions. Had the run continued, this node would have been assigned new work at some point in the future.

6 CONCLUSION

In this paper, we've motivated the need for Distributed ETL, described a working implementation, and characterized its behavior. Before using it in production, however, we should evaluate Distributed ETL along side existing stream analytics software that could be repurposed to perform the same function, as well as implement additional functionality necessary for real life workloads.

6.1 Related Work

Many general purpose stream analytics platforms like Apache Flink, Apache Spark, and Apache Storm could be repurposed to perform the work of ETL, but they require a much more complex setup of differing roles (master, worker), and do not provide fault-tolerance on their own. Instead, they rely on the service owner maintaining a ZooKeeper instance for high availability. Distributed ETL provides all the coordination required out of the box.

As this project was underway, LinkedIn announced their own implementation of a very similar service called Brooklin [2], which performs data transformations using SQL like semantics. However, their coordination also relies on a ZooKeeper instance.

6.2 Future Work

6.2.1 Additional Sources and Sinks. For the system to be useful, additional sources (Kafka, EventHub) and sinks (Avro, S3, Azure Storage) should be supposed. These extensions should be a straight forward implementation of the given interfaces, and time taken will mostly depend on the SDKs being used.

6.2.2 Scheduler. Throughout the paper, we've assumed that all sources being processed require similar amounts of resources from the node, which may not always be the case. A scheduler that reallocates work based on statistics collected from all nodes (CPU/memory/network utilization, etc) has the potential to improve overall performance.

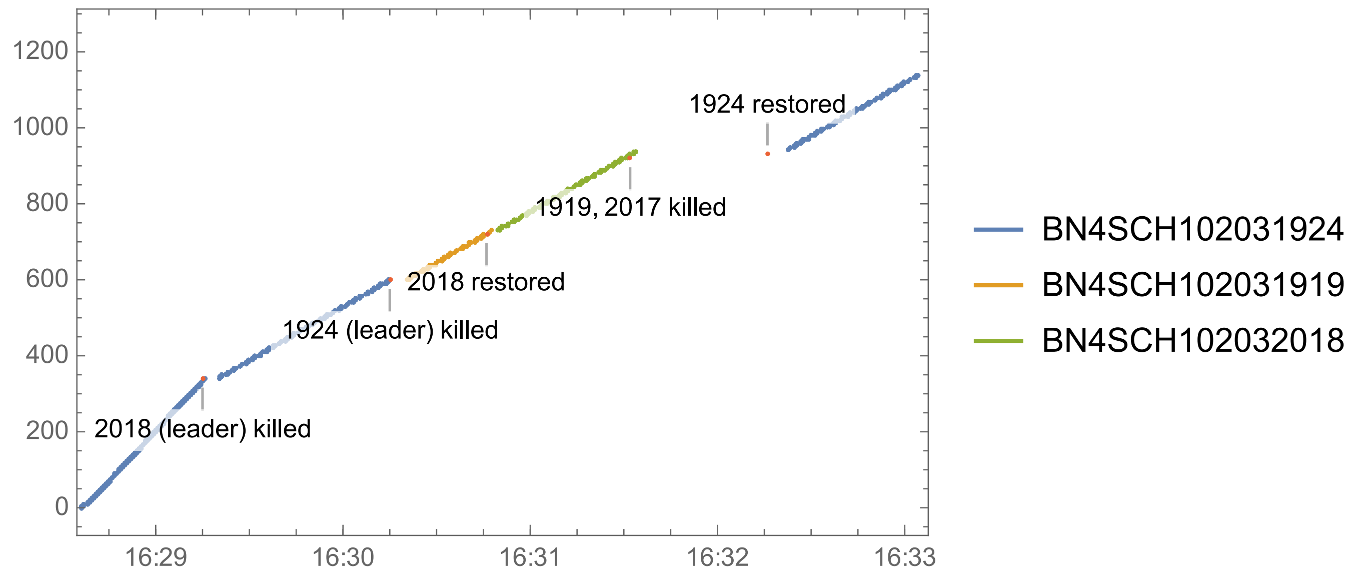

Figure 1: Partition 5 Progress

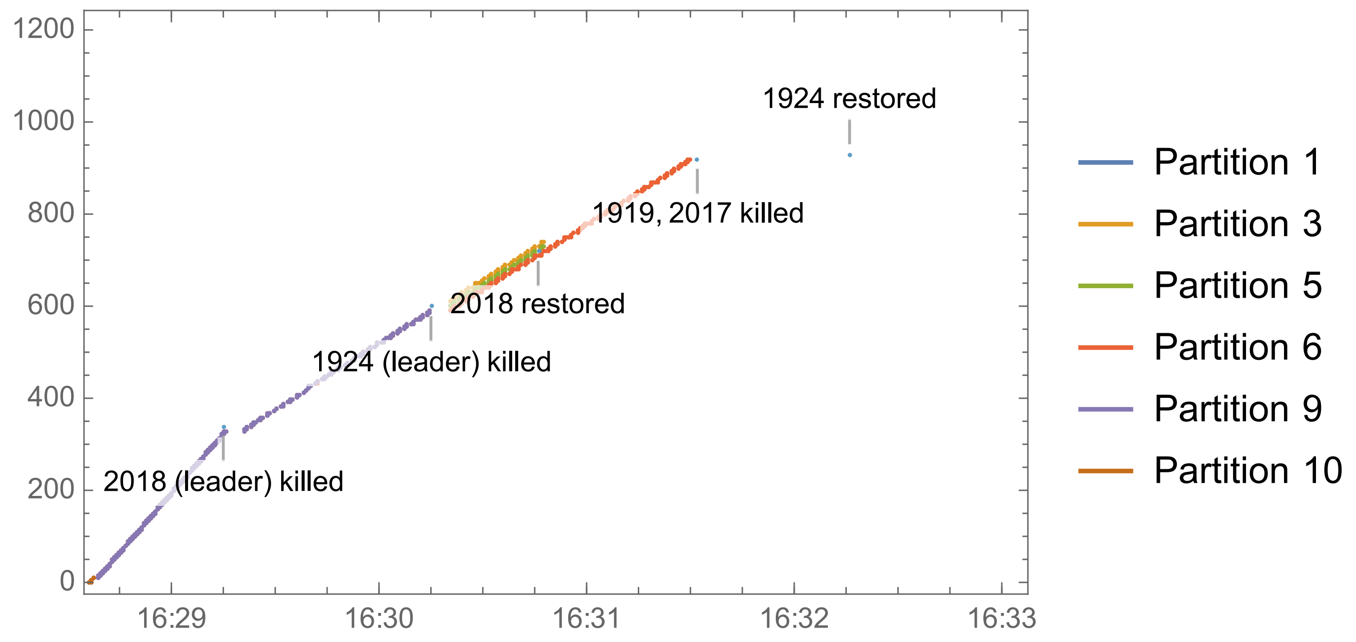

Figure 2: Node BN4SCH102031919 Work

REFERENCES

[1] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 305–320. http://dl.acm.org/citation.cfm?id=2643634.2643666

[2] Samarth Shetty. 2017. Streaming Data Pipelines with Brooklin. (2017). https://engineering.linkedin.com/blog/2017/10/streaming-data-pipelines-with-brooklin