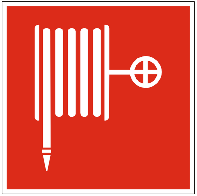
Chesapeake Large Scale Analytics Conference



FIREHOSE: Benchmarking Streaming Architectures

Karl Anderson
Laboratory for Telecommunication Sciences

October 2016

What is stream processing for data?

- Mostly small compute, bit I/O (memory, network)
- Event driven processing
- Large branchy processes
  - Conditional processing
  - Reduce processing by short-circuiting processing stages
  - hard to map to GPUs
- State Tracking / Correlation of data over time
  - Random access memory lookups
  - I/O bound processing
- Data level parallelism
  - Data streams can be divided and processed independently
  - Data shuffling to move data for correlation
- Pipeline parallelism
  - Divide up processing stages, challenging to balance

Performance of Streaming Architectures

- Memory access is critical
  - Random access lookups for state
  - Cache for local event processing
  - Data copies can be expensive
- Shared memory is key for thread scaling
  - Thread to thread communication: around 1 million pointers per second across a lock-free queue
- Message passing scales to cluster
  - Distributed processing
  - ZeroMQ, TCP, UDP
  - Serialization can be a significant overhead

Why Benchmark Streaming Architectures?

- To understand the overhead of processing using a particular architecture/system
  - measure actual intended use vs marketing claims
  - measure data models
  - measure hardware, framework and communication overhead
- To diagnose scaling problems
  - using data generation that can go beyond current data rates
- To measure various processing algorithms and approaches

Attributes of a good streaming benchmark

- Scalable stream rate
- Well-defined analytics that are easy to implement and measure
  - Impacts core capabilities of processing frameworks
  - Ground truth is known
- Data quantities that overflow memory and force real-time processes
- Allow for serial and parallel implementations
- Open and accessible

Measuring Streaming Architectures

- Ideally we want to measure
  - Energy-use per data (Joules/data) for a given processing system and data rate
  - Power, space and cooling are key design features
  - We have not done this yet

Issues in measuring streaming performance

- Problems in streaming architectures can often only be found when running *continuously* for hours or days
  - Resource limitations are not seen initially
  - Memory fragmentation can reach catastrophic conditions
  - Example: STL Hashtable resizing
- Not all processing is equivalent
  - Exact vs. probabilistic
  - Windowed vs. continuous

FIREHOSE Benchmark Package

- 3 Data Generators
  - C code
  - UDP packet output
  - Multiple events per packet, millions of events per second
- Reference implementations of streaming analytics
  - C++ and Python
- Testing Documentation, Ground Rules
- Available at
  `https://github.com/stream-benchmarking/firehose`

Reference Analytic

- Generate useful <key, value>pairs
- Examine data over time for each key (state)
- Trigger condition by accumulating values for each key

GOAL: measure the ability to perform data correlation

Generator One

The Story: Find anomalous keys that are producing biased values. Values for each key are either 0 or 1 with a probability of 0.5 for generating a value of 1 for most key. Some keys are chosen to be biased and generate more zeros than ones.

The Reference Analytic

- Accumulate the first 24 values for each key
- Generate alert if observed sums less than 5
- Compare answer with a ground truth value in order to report
  - true positives, false positives
  - true negatives, false negatives

Generator One

Goal: measure basic processing and state tracking

- Continuous generator
- Fixed key space (100,000 total keys per generator)
- Skewed key emission
- UDP packet containing
  - KEY(64bit), VALUE(0,1), Bias Truth(0,1)

Example:

322342123234, 0, 0
993248345234, 1, 0
323423422322, 0, 1
...

Generator Two

Goal: measure performance of state tracking and expiration

- Continuous generator
- Unbounded key space
  - Active Set Size: 131,072 keys
  - Number of events per key is chosen from a skewed distribution
  - Space/time between reoccurring key events is chosen from a trend curve
  - Many keys only generated once
  - No notification of key expiration inside generator
- Forces processing analytics to expire state
- Similar output to Generator One
  - could use exact same processing analytic
  - except keys are infinite, so expiration matters

Events per key

In order to simulate common datastreams, keys are generated from a skewed distribution

- Most keys will not generate enough events to trigger analytic reporting
- Only when a key has generated 24 events will an analytic be required to report results

Intensity: Trend Curve

In order to simulate common datastreams, keys are spaced out following a trend curve

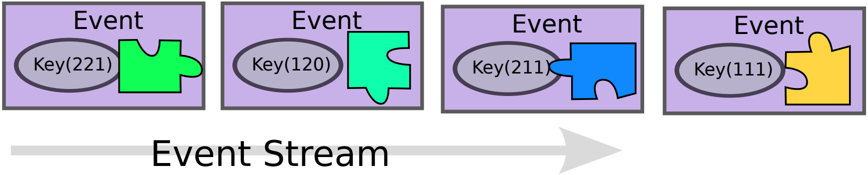
- key spacing is implemented using a priority queue that allows for control of when keys get generated
- When a key first starts out, its generation is spaced out sparsely in the event stream
- Over the generation-life of a key, a key will increase and then decrease in intensity
- Currently all keys-events are mapped from the same trend curve

Generator 3 - two level active set

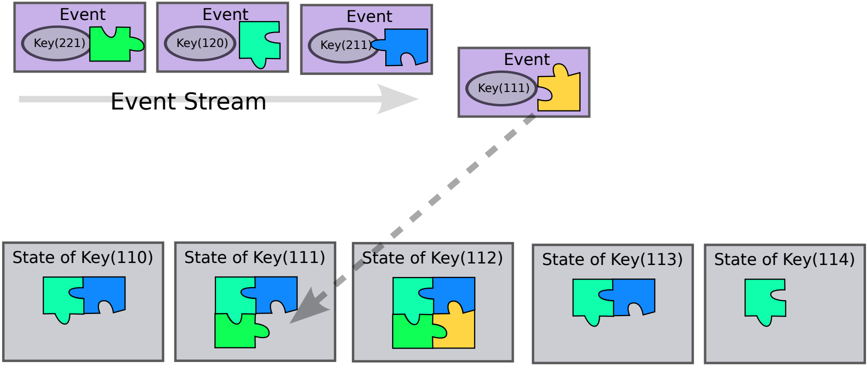
Goal: to measure multi-state data shuffles and simulate complex event streams

- Continuous generator, unbounded keyspace
- Two levels of events
  - Outer events used to build inner events
  - Two skewed active-set generators are maintained for two generators
- The outer generator emits $< key, value >$ pairs
  - values from the same outer key are pieced together to build inner $< key, value >$ events
  - an inner $< key, value >$ is made from 5 outer events
- Inner generator emits values that are 0 or 1 with potential bias
- An analytic that shuffles data for outer state tracking will need to reshuffle for inner keys

Two level event stream



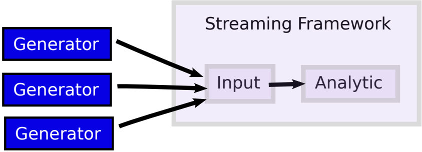
Two level event stream



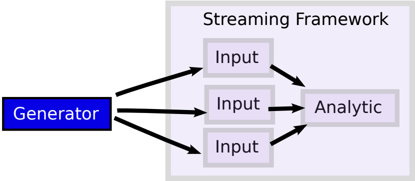
Generator Tuning

- Each generator can be configured to emit events at a prescribed rate (events/sec)
- You can set the random seed for events
  - deterministic testing
- Configure the number of receivers and UDP ports
  - emits packets in round robin to each receiver
- Supports parallel generation
  - each generator has independent key-spaces
  - must start at same time via shell scripts

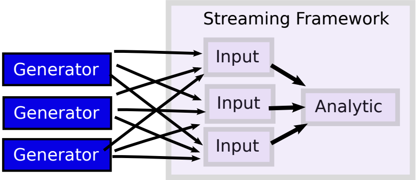
Parallel Generation



Many To One



One to Many



Many to Many

Measuring Results

- If the streaming analytic is performing correctly
  - Measure packet/event receive rate (drop rate)
  - Count total number of keys observed
  - Count accuracy of anomaly detection
- Testing
  - Ramp up rate until dropping less than 1% of packets
  - Compare against reference implementations on same system conditions

Benchmarking Results:

- Dell dual hex-core 3.47 GHz Intel Xeons (X5690)
- Maximum rates reported when rate reached no packet drops

Implementation	Benchmark	# Generators	Rate (events/sec)
C++	#1	2	5.6M
Python	#1	1	450K
Waterslide(serial)	#1	5	12M
Phish(serial)	#1	5	5.5M
Phish(parallel)	#1	5	10M
C++	#2	1	1.9M
Python	#2	1	140K
Waterslide(serial)	#2	2	3.4
Phish(serial)	#2	1	1.9
Phish(parallel)	#2	2	3.4
C++	#3	1	1.5M
Waterslide(serial)	#3	2	2.9M

Table: Reference Analytic Results

Waterslide Stream Framework

- An High-Speed framework for stream multi-threading
- Available as Open Source at `https://github.com/watersideLTS/waterslide`
- A C based modular system
- An agile BASH-like script-able language for specifying workflows
- Scaling: A Multi-threaded pass-by-reference system
  - Based on Sandia's Q-Threads
  - Stream-optimized garbage collector
- Multi-way expiring state tables
  - Large scale key-value cache with LRU expiration

Future Streaming Benchmarks

- Graph Generation
  - Find triangles
  - Find triangles that have the same number of events
  - Find small connected components
  - Find large temporally correlated groups (time/space clustering)
- missing: a graph generator than creates realistic streaming graphs
- current experimentation: evolving E-R graphs

Special Thanks

- Steve Plimpton at Sandia - Firehose Benchmarking
- Waterslide Development team