FireHose Documentation

http://www.sandia.gov/~sjplimp/firehose.html

Table of Contents

These web pages describe a collection of open-source stream processing benchmarks we have defined, called FireHose. We are inviting users and authors of various stream-processing frameworks to implement and experiment with the benchmarks in order to refine/expand their definitions and collect performance results. After this initial phase, we plan to create a new FireHose web site, which will include the definition of the benchmarks, sample software implementations, and performance results. We hope the benchmarks and the WWW site will stimulate further discussion and interactions within the streaming community.

The sponsors of the FireHose benchmarks are Karl Anderson (DoD) and Steve Plimpton (Sandia National Labs).

PDF file of the entire documentation, generated by htmldoc

Motivation

Streaming data arrives continuously and in volumes and rates that are ever increasing. Timely processing of streaming data is computationally challenging due to limited resources. These include limited CPU operations that can be performed on a datum before the next one arrives, limited memory for storing state information about the stream, limited disk storage for the data itself so that the datums may only be seen once, and limited budgets for energy consumption or CPU/memory/disk resources.

In some settings, it's not known in advance what interesting features are embedded in the stream of data. In an algorithmic sense, this makes the above limitations even more challenging. In addition, the hardware options for collecting streaming data are increasing in variety and capability from multicore CPUs, to GPUs, to special-purpose processors like Tilera chips, to solid-state (Flash) disks. While choices are a good thing, they also challenge developers of stream-processing software to support a growing range of hardware.

Computing on streaming data has been referred to as "drinking from a firehose". Hence the name FireHose for this suite of stream processing benchmarks.

The main purpose of FireHose is to enable comparison of streaming software and hardware, both *quantitatively* vis-a-vis the rate at which they can process data, and *qualitatively* by judging the effort involved to implement and run the benchmarks. Because we are both users and designers of software for stream processing, we have wanted for some time to define useful metrics for the speed and flexibility various software/hardware solutions can offer. FireHose represents our initial effort in this regard.

We plan to release the benchmark definitions and sample implementation code openly on the FireHose web site, so that anyone can implement them with their own stream-processing software and/or run them on their own hardware. Our hope is that this effort will generate interest and feedback from the academic and commercial streaming communities. With their help in refining and exanding the suite of benchmarks, and their contributions of performance results, we hope the FireHose benchmarks can become a valuable resource for those interested in stream processing.

The initial benchmarks in the FireHose suite reflect common tasks typical of real-time processing of network packets arriving at high rates as a network is monitored, e.g. for cybersecurity purposes. We plan to augment the suite with benchmarks that process streaming graph edges, and are open to additional suggestions as well.

Benchmark design

Each of the benchmarks in the FireHose suite has two parts: a *front-end generator* that writes a stream of events (datums) to one or more UDP sockets in a text-based format, and a *back-end analytic* that reads the stream and processes the datums.

The generators create a stream of datums with the following attributes:

- reproducible
- scalable rate, up to tens of millions of datums/sec
- serial or parallel generation

The computations performed by the analytics have the following attributes:

- well-defined and relatively easy to implement
- allow for use of more sophisticated data structures and algorithms
- scoring metrics for the processing rate and accuracy of the result
- allow for serial or parallel implementation (shared or distributed memory)

The FireHose tarball provides code for each generator. They are meant to be used as-is, without modification. The tarball also provides sample implementations of the analytics, but users are free to re-implement them in a manner optimized for their streaming framework or their machine, or with better algorithms, so long as they follow the rules of the benchmark and measure the relevant scoring metrics.

UDP sockets are used as the link between the generator and analytic, so that the generator is not throttled by the analytic, just as in stream processing scenarios where data must be processed as it appears in real-time. The UDP protocol means the analytic will drop datums if it cannot keep up with the generated stream, which is one of the effects we wish to measure.

Generators

The current version of FireHose has three stream generators:

1. biased power-law generator
2. active set generator
3. two-level generator

Source code for these is in the tarball directory "generators". Each sub-directory has code for one generator. Instructions for how to build and run the generators is given in the Running the benchmarks section.

Timing results listed below for datum generation rates were run on a desktop Dell Linux box with dual hex-core 3.47 GHz Intel Xeon (X5690) CPUs.

Each generator writes a stream of datums to one or more specified UDP ports on one or more specified hosts. The host/port pairs are specified as arguments to the generator. If multiple arguments are specified, the same stream of datums is written to multiple hosts/ports. Hosts can be specified in numeric IPV4 or IPV6 format, or as alphabetic hostnames. A port ID is optional and is appended to the host with a "@" separator, as in the following examples. If not specified, a default port ID of 55555 is used. These are example host/port arguments:

127.0.0.1	localhost (this box)
127.0.0.1@55555	localhost + port ID
134.251.2.121	another box
134.251.2.121@5678	another box and port ID
fe80::a213:1234:1234:1234	IPV6 host
fe80::a213:1234:1234:1234@12345	IPV6 host and port ID
otherbox.mynet	another box
otherbox.mynet@5432	another box and port ID

All the generators produce a stream of "packets" in ASCII text format. Each packet contain one or more "datums" bundled together to increase the effective stream generation rate. The format of a packet with N datums is as follows; each line is terminated by a newline:

```
packet ID
datum1
datum2
...
datumN
```

The ID (following the word "packet") is an integer, which can be stored as a 64-bit unsigned value. When a single generator is running, it is a number from 0 to N-1, where N = the number of packets. Packet IDs are altered when multiple generators are running in parallel, as discussed below. Analytics which read the packets typically ignore the packet ID, but it can be monitored if desired when developing or debugging an analytic. Each datum is a string of text, followed by a newline. The format of a datum is generator-specific, as discussed below.

In addition to the host/port argument(s), each generator takes optional command-line switches, each of which has a default setting. The following switches are common to all generators. Generator-specific switches are discussed below.

These command-line switches affect packet generation:

- -n Npackets = # of packets to generate, default = 1000000 = 1 million
- -b Nbundle = # of datums per packet, default = 50 (40 for generator 3)
- -r rate = rate at which datums are produced in datums/sec, default = 0

The total # of datums generated is Npacket*Nbundle. The default rate of 0 means packets are written as fast as the generator can create them. For finite rates, a timer is called after every packet is written and the generator delays sending the next packet by sleeping for an appropriate time.

Multiple copies of a generator can be launched simultaneously, so that a different stream is produced by each. When this is done, the following two switches should be used:

- -p Nprocs = # of instances of generator being run, default = 1
- -x Iproc = which instance this generator is (0 to Nprocs-1), default = 0

All the generators launched should specify the same value of Nprocs. Each should use a different value of Iproc from 0 to Nprocs-1. The generators use these values to change the random # seeds used internally for datum generation, as explained below. They also use them to encode a unique ID for each packet across all the generators. For example, if 3 generators are launched, and each is generating 5 packets (-n switch), then generator #1 will generate packet IDs 0,3,6,9,12, generator #2 will generate IDs 1,4,7,10,13, and generator #3 will generate IDs 2,5,8,11,14, for a total of 15 packets. Of course if all generators write to the same host/port, there is no guarantee the packets will arrive in numeric order.

When a generator finishes writing its stream of packets, it prints its generation rate in datums/sec, which should be very close to the rate setting (-r switch).

After this message is printed, the generator does not exit, but goes into an infinite loop, writing STOP packets at a rate of 100/sec to its output host/ports. Each stop packet contains a single integer (Iproc) followed by a newline.

Iproc is the argument to the -x switch discussed above (default = 0), which is effectively a unique ID for the generator. The message signifies that the generator is finished. When the analytics receive this packet (or multiple such packets if multiple generators are running), they know the stream has ended and can terminate. Note that the generator(s) themselves never terminate; you must kill each explicitly, e.g. with a Ctrl-C in the window each was launched from.

The Running the benchmarks has instructions about how to launch and kill multiple generators using scripts provided in the tools directory.

---

(1) Biased power-law generator

See the generators/powerlaw directory for source code for this generator. This generator creates datums in a 3-field, comma-separated format:

```
1000582694,0,1
```

The first field is a "key", represented internally as a 64-bit unsigned integer. The second field is a "value" of 0 or 1. The third field is a "truth" flag, also 0 or 1.

Keys are generated randomly from a static range of M keys. M is hardwired to 100,000 to represent a moderately large key space, which means all keys will be in the range 0 to M-1 (see the -o setting below). The selection is power-law distributed so that keys at the low end of the range are generated much more frequently than keys at the high end. The slope of the power-law distribution is hardwired internally so that more than 90% of the keys in

the range will typically be generated at least once during even short benchmark runs.

The keys are divided into 2 categories, "unbiased" and "biased", with one biased key for every 255 unbiased keys. This is done randomly, but consistently each time the same key is generated. Each time a key is generated, the generator also assigns it a random "value" of 0 or 1. For unbiased keys, 0 versus 1 is chosen with equal probability. For biased keys, 0 is selected with probability 15/16, and 1 is selected with 1/16 probability. A truth flag is appended to each datum; it is 0 for unbiased keys and 1 for biased keys.

This generator accepts 3 command-line settings, in addition to those common to all generators as discussed above:

- -o offset = offset for all generated keys, default = 1000000000 = 1 billion
- -s seed = random # seed for key generation, default = 678912345
- -m mask = random mask used to flag keys as anomalous, default = 12345

The offset is added to all generated keys.

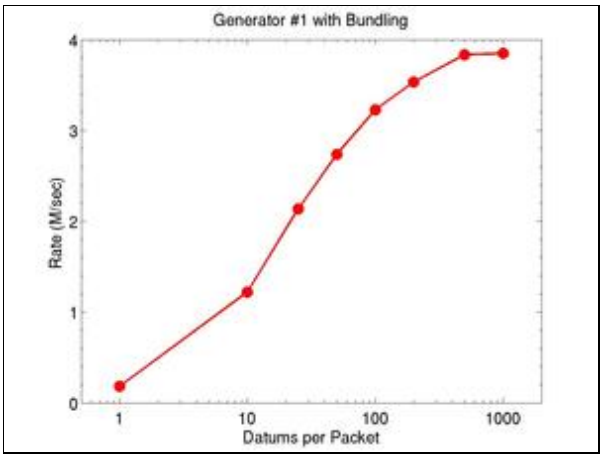
The seed is used for the random # generator that samples keys and values. If seed is set to 0, then the current time-of-day is used to generate a seed, which means the stream will not be the same the next time you run the generator.

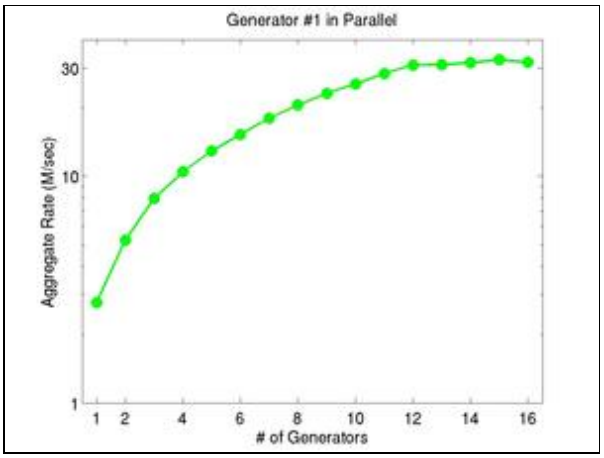
The mask is used when hashing keys to flag them as unbiased or biased.

When running multiple generators in parallel, the value of Iproc (-x switch) is added to the seed and to the offset (multiplied by 1 billion) used by each generator. This means that each instance of the generator will create its own set of keys that do not overlap with the other generators.

These two plots show the single-core and parallel performance of the generator, running on the Dell box described above. The generators are writing to a UDP port on the same box.

The first plot shows generation rate in datums/sec on a single core as a function of bundle size. The second plot shows aggregate generation rate when running up to 16 generators on the same box (which has only 12 cores), all writing to the same UDP port, with a bundle size of 50 datums/packet. An aggregate rate of 30M datums/sec can be achieved.





Click on the plots for a larger version.

(2) Active-set generator

See the generators/active directory for source code for this generator. This generator creates datums in the same 3-field, comma-separated format as the previous generator:

```
1000582694,0,1
```

The first field is a "key", represented internally as a 64-bit unsigned integer. The second field is a "value" of 0 or 1. The third field is a "truth" flag, also 0 or 1.

The second generator performs the same operations for labeling keys as unbiased versus biased and for generating values based on the two flavors of keys. However, rather than use a static key range, it selects keys from a continuously evolving "active set" of keys. Thus the number of unique keys generated is effectively infinite if the generator runs for a long time. The size of the active set is 128K keys. Each key is generated a limited number of times, then removed from the active set and replaced by a new key.

The probability with which an individual active key is sampled also varies in time according to a roughly bell-shaped curve. This creates a "trending" effect where a key appears occasionally, then appears more frequently, then tapers off.

This generator accepts 2 command-line settings, in addition to those common to all generators as discussed above:

- -s seed = random # seed for key generation, default = 678912345
- -m mask = random mask used to flag keys as anomalous, default = 12345

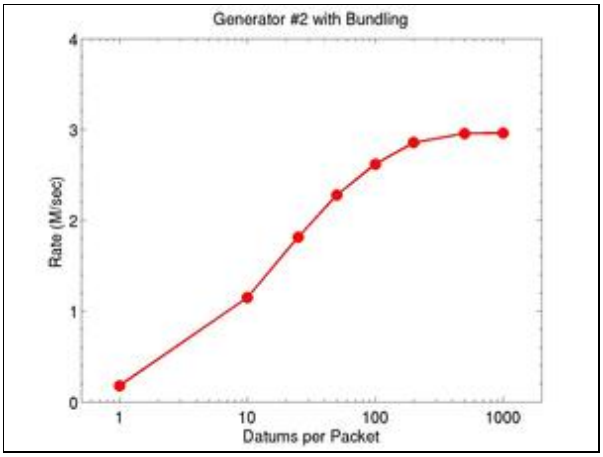
The seed is used for the random # generator that samples keys and values. If seed is set to 0, then the current time-of-day is used to generate a seed, which means the stream will not be the same the next time you run the generator.

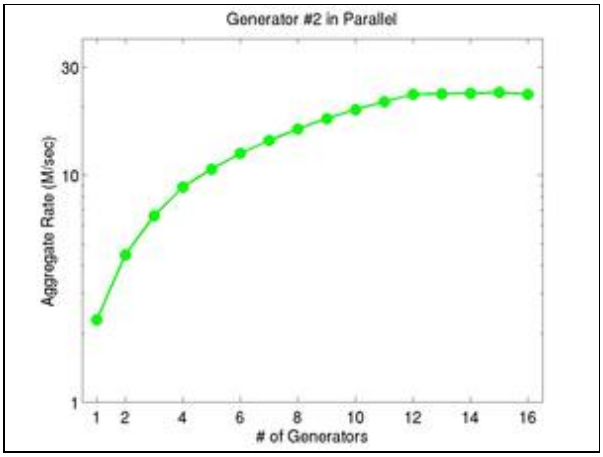
The mask is used when hashing keys to flag them as unbiased or biased.

When running multiple generators in parallel, the value of Iproc (-x switch) is added to the seed used by each generator. This means that each instance of the generator will create its own set of keys that do not overlap with the other generators. More precisely, the set of active keys created by each instance of the generator will be randomly selected as 64-bit unsigned integers (roughly 1.8 x 10^19). Depending on how lo

These two plots show the single-core and parallel performance of the generator, running on the Dell box described above. The generators are writing to a UDP port on the same box.

The first plot shows generation rate in datums/sec on a single core as a function of bundle size. The second plot shows aggregate generation rate when running up to 16 generators on the same box (which has only 12 cores), all writing to the same UDP port, with a bundle size of 50 datums/packet. An aggregate rate of 22M datums/sec can be achieved.





Click on the plots for a larger version.

---

(3) Two-level generator

See the generators/twolevel directory for source code for this generator. This generator creates datums in a 3-field, comma-separated format:

```
3929309884956,57364,1
```

The first field is a "key", represented internally as a 64-bit unsigned integer. Within the generator this is an "outer" key. It is generated the same way as keys in the active-set generator described above, using a continuously evolving "active set" of keys (128K in size) and a "trending" effect which varies the frequency at which the key is emitted.

Each outer key is generated at most 5 times (though it may be emitted less), before it is replaced in the active set. The first 4 times it is emitted, a random 16-bit unsigned integer is chosen as the "value". The fifth time it is emitted, one of the following 4 quantities is the "value":

- 00
- 01
- 10
- 11

The first four 16-bit values are meant to be concatenated together to form a new 64-bit unsigned integer which is an "inner" key. The value associated with the inner key is the rightmost digit of the two-digit value for the fifth occurrence of the outer key, i.e. 0 or 1.

The generator creates a stream of datums such that multiple copies of the same inner key will be produced, each associated with 5 occurrences of a different outer key. The 0/1 values assigned to a specific inner key will be biased or unbiased, using the same procedure described above for the powerlaw and active set generators.

The "truth" flag for the inner key is the leftmost digit of the two-digit value for the fifth occurrence of the outer key, it is 0 for unbiased inner keys and 1 for biased inner keys.

The third field of the outer key datum is a counter from 1 to 5, which encodes the Nth time this outer key has been emitted. It can be used by the analytic to insure it has seen all occurrences of a particular outer key, and can thus compute a valid inner key/value pair.

This generator accepts 1 command-line setting, in addition to those common to all generators as discussed above:

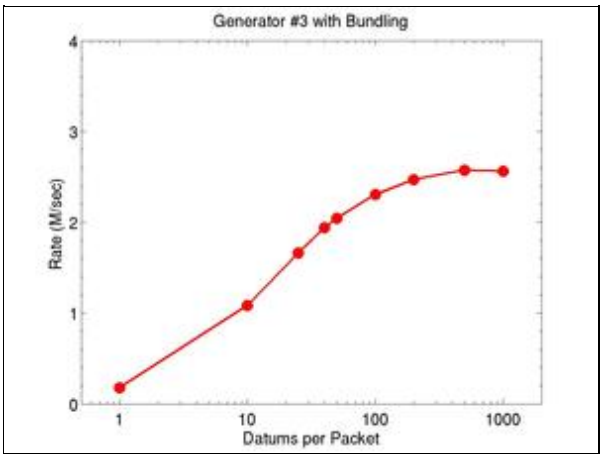
- -s seed = random # seed for key generation, default = 678912345

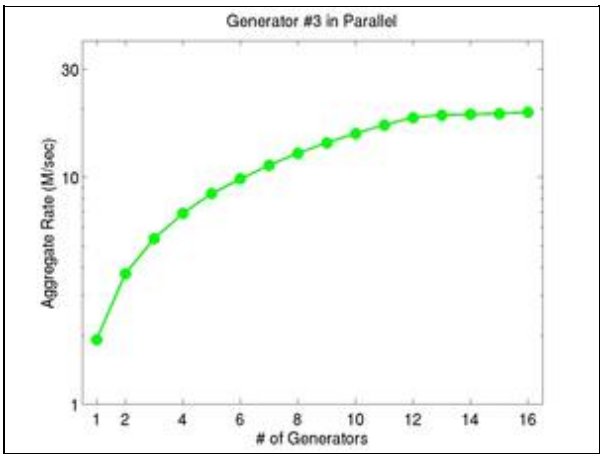
The seed is used for the random # generator that samples outer and inner keys and values. If seed is set to 0, then the current time-of-day is used to generate a seed, which means the stream will not be the same the next time you run the generator.

When running multiple generators in parallel, the value of Iproc (-x switch) is added to the seed used by each generator. This means that each instance of the generator will create its own set of outer/inner keys that do not overlap with the other generators. More precisely, the set of active keys created by each instance of the generator will be randomly selected as 64-bit unsigned integers (roughly 1.8 x 10^19). Depending on how long the generators run, there is a small possibility that two or more generators will emit the same outer/inner keys.

These two plots show the single-core and parallel performance of the generator, running on the Dell box described above. The generators are writing to a UDP port on the same box.

The first plot shows generation rate in datums/sec on a single core as a function of bundle size. The second plot shows aggregate generation rate when running up to 16 generators on the same box (which has only 12 cores), all writing to the same UDP port, with a bundle size of 40 datums/packet. An aggregate rate of 20M datums/sec can be achieved.





Click on the plots for a larger version.

Analytics

The current version of FireHose has three analytics:

1. anomaly1 = anomaly detection in a fixed key range
2. anomaly2 = anomaly detection in an unbounded key range
3. anomaly3 = anomaly detection in a two-level key space

Source code for these is in the tarball directory "analytics". Each sub-directory has code for multiple versions of the analytic, developed in different languages and for different streaming frameworks. Instructions for how to build and run the analytics is given in the Running the benchmarks section.

These are the implementations we currently provide. Over time, as users contribute them, we hope to add more implementations of the analytics, written for various streaming frameworks.

- C++: This is a reference implementation which runs on a single core. It can be examined to learn details of how the analytic is defined and can be implemented efficiently. Typically it is a single file.
- Python: This is also a reference implementation which runs on a single core. It is typically more concise than its C++ counterpart, but will not run as efficiently.
- PHISH: These are implementions in C++ and/or Python which run on top of the PHISH library. PHISH is a lightweight, open-source framework for processing streaming data with distributed-memory parallelism. It passes datums between processes running on one or more cores via message-passing, using either the MPI (message-passing) or ZMQ (sockets) library as a back-end. The processes are stand-alone programs (C, C++, Python, etc) which, in this case, contain code similar to what is found in the serial implementations of the analytic.

As explained below, each analytic is required to perform a specific computation on the streaming data it receives via a UDP port from one of the stream generators described in the Generators section. The format of the stream is defined by the generator.

An implementation of the analytic should follow the rules discussed below. Most importantly, it should be robust to the possibility it will miss packets in the stream which arrive before the analytic is ready to read them. This is because the generators can be run at high rates, which may outpace the analytic.

If you wish to implement an analytic yourself, or to run a benchmark on your hardware, you are free to implement it in a different manner from the provided code. But you need to perform the computations outlined below and adhere to the following guidelines, at least if you wish to contribute your implementation to the code archive and your performance results to the Results section.

- The analytic should not archive (e.g. to disk) the entire stream or large portions thereof and perform its computation when the stream ends. This violates the spirit of a streaming benchmark and would not work in practice when the stream is infinite.
- As described in the Generators section, when running in parallel to achieve higher stream rates, each instance of a generator will emit keys that do not overlap with the keys of the other generators. This is to make it easier to check for correct answers and score the results. However, the analytic cannot take advantage of this fact. It must assume that a datum (and its key) could have come from any of the generators. Thus it is not permissible, for example, to process a stream coming from N generators as N independent streams being read by N copies of an analytic program, each processing its own unique set of keys.

- Each analytic takes a single argument which is the UDP port ID to read the stream from. Each analytic may also define optional command-line switches, useful either to pass in stream parameters, or for debugging. The options for the serial implementations are listed below. Versions written for different frameworks may define different optional arguments. Check the README files in the implementation sub-directories for details.
- While an analytic is running, it should print results to the screen as it finds them, i.e. in real-time as it identifies anomalies. This needs to be in the format described below so that the results can be checked against the correct answer, for scoring purposes. See the Results section for details.
- The analytic must implement logic for detecting when the stream ends. As described in the Generators section, this is done by checking for specially-formatted STOP packets. If a single generator is sending to an analytic, this is a simple check. The logic is more convoluted if multiple generators are sending data to the analytic. See the reference implementations for an example of how this can be handled.
- When the analytic terminates, it should print final statistics on the number of packets it received, as well as summaries of its computations, as discussed below. These are useful for debugging and are used in the scoring metrics. Some of the summary values also appear in the tables of the Results section.

Any additional anayltic-specific rules are discussed below.

---

(1) Anomaly detection in a fixed key range

This analytic is designed as a simple test of anomaly detection; keys are flagged as unbiased or biased (anomalous) depending on the values associated with an individual key as it appears multiple times. Only a small fraction of the keys are biased.

It is designed for use with the biased-powerlaw generator, which produces packets of datums, each of which is an ASCII text string in the following format:

```
1000582694,0,1
```

The first field is a "key", which can be represented internally as a 64-bit unsigned integer. The second field is a "value" of 0 or 1. The third field is a "truth" flag, also 0 or 1.

The computational task for this analytic is to identify biased keys. An unbiased key will have 0 versus 1 values appear with equal probability. A biased key will have 0 versus 1 values appear in a skewed 15:1 probability ratio.

Detection can be performed by keeping track of the values seen for each key, e.g. in a hash table of fixed size (see below). When an individual key has been seen 24 times, the key is labeled "biased" if a value of 1 appeared 4 or less times; otherwise the key is labeled "unbiased". When the decision is made, the answer is checked against the "truth" flag appended to each datum. The large majority of the results will be keys correctly flagged as unbiased (true negative). The other 3 cases are more rare; the count for each of the 4 categories is tallied by the analytic:

- true anomalies = keys correctly flagged as biased
- false positives = keys incorrectly flagged as biased
- false negatives = keys incorrectly flagged as unbiased
- true negatives = keys correctly flagged as unbiased

The analytic also emits a line of output when a detection event in one of the first 3 categories occurs, in this format, where the last field is the key:

```
true anomaly = 12345
false positive = 34567
false negative = 67890
```

Note that due to statistical variation in the way the generator produces random values, for the 3 rare outcomes, true anomalies should mostly be detected, but about 20% of the rare cases will be false positives, with an occasional false negative as well.

The analytic also keeps track of the following statistics which are printed in the following format at the end of the benchmark:

```
packets received = 999986
datums received = 49999300
unique keys = 85578
max occurrence of any key = 16709489
true anomalies = 34
false positives = 6
false negatives = 0
true negatives = 3220
```

A reasonable way to implement this analytic is with a hash table that stores a running tally of the values seen for each unique key. Note that the analytic does not know how many times an individual key will appear in the stream, though it need not retain a key after it has been seen 24 times.

Optional command-line switches in the serial reference implementations (C++ and Python):

- -b = bundling factor = # of datums in one packet, default = 50
- -t = check key for bias when appears this many times, default = 24
- -m = key is biased if value=1 <= this many times, default = 4
- -p = # of generators sending to me, only used to count STOP packets, default = 1
- -c = 1 to print stats on packets received from each sender, default = 0

Additional implementation rules:

- The biased-powerlaw generator is hard-wired to produce at most 100,000 unique keys per instance of the generator. It is OK for the analytic to know and use this value, e.g. to pre-allocate memory for a hash table of fixed size. Note that for good performance this size should typically be a multiple of the number of generators contributing to the stream (see the -p switch), since the keys each generator emits do not overlap.
- The truth flag cannot be used, except to check answers.

---

(2) Anomaly detection in an unbounded key range

This analytic is also designed as a simple test of anomaly detection; keys are flagged as unbiased or biased depending on the values associated with an individual key as it appears multiple times. However the number of unique keys in the stream is effectively infinite for long benchmark runs, so the analytic must decide which keys to store state for.

It is designed for use with the active set generator, which samples from an unbounded range of keys.

In benchmark runs using this analytic, huge numbers of unique keys will be generated, which means it will be inefficient to use a hash table to store all of them, unless some provision for expiring keys is implemented (see below).

Aside from this change in the distribution of keys in the data stream, the computational task of detecting biased keys is identical to the decscription for analytic #1, as is the format of output the analytic should generate. The only exception is that a final output for "max occurrence of any key" is no longer a relevant value to tally.

A reasonable way to implement this analytic is with an expiring hash table that attempts to keep track of values for keys in the active set of the generator, e.g. via a least-recently used (LRU) metric, where the key selected for expiration (removal from the hash table) is the one seen longest ago in the stream. As discussed below, the size of the set of active keys is limited by the generator which can be used by the analytic to age keys efficiently. Note however that the analytic does not knowhow many times an individual key will remain in the generator's active set or appear in the stream, though it need not retain a key after it has been seen 24 times.

Optional command-line switches in the serial reference implementations (C++ and Python):

- -b = bundling factor = # of datums in one packet, default = 50
- -t = check key for bias when appears this many times, default = 24
- -m = key is biased if value=1 <= this many times, default = 4
- -p = # of generators sending to me, only used to count STOP packets, default = 1
- -c = 1 to print stats on packets received from each sender, default = 0
- -a = size of active key set, default = 131072 (128K)

Additional implementation rules:

- The active-set generator is hard-wired to use an active set size of 131072 (128K) keys. It is OK for the analytic to know and use this value (see the -a switch above), e.g. to setup an expiring hash table that keeps only the most recently seen N keys. Since the analytic does not know which keys the generator keeps in its active set, N should be larger than the active set size, so that it is unlikely to expire a key that the generator still treats as active. A good value for N is 2x the active set size. Note that the size of the active set seen by the analytic is effectively a multiple of the number of generators contributing to the stream (see the -p switch), since the keys each generator emits do not overlap.
- The truth flag cannot be used, except to check answers.

---

(3) Anomaly detection in a two-level key space

This analytic also performs anomaly detection; a set of keys are flagged as unbiased or biased depending on the values associated with an individual key as it appears multiple times. However, the key/value pairs for which the bias detection is done, are derived from state information that must be stored for a different set of keys, namely those which appear in the data stream. This requires two iterations of storage and lookup in different hash tables.

It is designed for use with the two-level generator, which produces packets of datums, each of which is an ASCII text string in the following format:

```
3929309884956,57364,1
```

As explained on the doc page for the two-level generator, the first field is an "outer" key (64-bit unsigned integer), the second field (16-bit unsigned integer) is a portion of an "inner" key, and the the third field is a "counter" for how many times the outer key has been emitted.

The first 4 times the outer key is seen, the analytic should store the 4 values. The fifth time the outer key is seen, the 4 previous values are concatenated (right to left, i.e. low-order to high-order) into a new "inner" key, which also becomes a 64-bit unsigned integer. The value for the inner key is the rightmost digit of the value for the fifth occurrence of the outer key, which is a 0 or 1.

The inner key/value pairs are treated the same as the key/value pairs in the first analytic and checked for bias. I.e. when an individual inner key has been seen 24 times, the inner key is labeled "biased" if a value of 1 appeared 4 or less times; otherwise the inner key is labeled "unbiased". When the decision is made, the answer is checked

against the "truth" flag, which was the leftmost digit in the value of the fifth occurrence of any of the outer keys which generated this inner key.

The four categories (true anomalies, false positives, false negatives, true negatives) of inner keys, and the format of output the analytic should generate, is the same as described above for the anomaly1 analytic. As in the second analytic, the only exception is the final output of "max occurrence of any key" is no longer a relevant value to tally.

Note that if packets are dropped, the analytic should not try to construct inner keys from incomplete information. The third counter field in the outer key datum can be monitored to insure a specific outer key is seen all 5 times it is needed.

A reasonable way to implement this analytic is with two expiring hash tables, one for outer keys and another for inner keys, and to expire keys from each via a least-recently used (LRU) strategy, as explained for the anomaly2 analytic. Outer keys will be emitted at most 5 times (but may be emitted less); individual inner keys will be encoded in the stream an unknown number of times.

Optional command-line switches in the serial reference implementations (C++ and Python):

- -b = bundling factor = # of datums in one packet, default = 50
- -t = check key for bias when appears this many times, default = 24
- -m = key is biased if value=1 <= this many times, default = 4
- -p = # of generators sending to me, only used to count STOP packets, default = 1
- -c = 1 to print stats on packets received from each sender, default = 0
- -a = size of active key set, default = 131072 (128K)

Additional implementation rules:

- The two-level generator is hard-wired to use active set sizes of 131072 (128K) keys for both the outer keys it generates explicitly and the inner keys it generates implicitly. It is OK for the analytic to know and use this value (see the -a switch above), e.g. to setup expiring hash tables that keep only the most recently seen N keys. Since the analytic has no way of knowing which keys the generator keeps in its active sets, N should be larger than the active set size, so that it is unlikely to expire a key that the generator still treats as active. For both hash tables, a good value for N is 2x the active set size. Note that the size of the active set seen by the analytic is effectively a multiple of the number of generators contributing to the stream (see the -p switch), since the keys each generator emits do not overlap.
- The truth flag cannot be used, except to check answers.

Running the benchmarks

The current version of FireHose defines three benchmarks:

1. anomaly1 = anomaly detection in a fixed key range
2. anomaly2 = anomaly detection in an unbounded key range
3. anomaly3 = anomaly detection in a two-level key space

Each benchmark uses a specific stream generator and a specific analytic, as discussed below. Multiple copies of a generator may be run to create higher-bandwidth streams, writing either to a single or multiple UDP ports. An implementation of the analytic can be run in serial or parallel (shared or distributed memory), so it may also choose to read from multiple UDP ports.

Currently, we advise running both the generator(s) and the analytic on a single (multicore) box. This will test performance of the analytic without bottlenecks associated with network bandwidths or latencies. However, the generators and analytics can be run on different boxes (across a network) which has the advantage that the box running the analytic will devote no CPU cycles to generating the stream.

IMPORTANT NOTE: A single FireHose generator running on a single core, may generate as much as 50 Mbytes of data per second. Since the UDP protocol means no receiver is throttling the sender, injecting the stream as UDP packets into a network can quickly saturate available bandwidth and have a negative impact on network performance,

This page has the following sub-sections:

Building the generators and analytics

Each of the generators can be built from the appropriate directory as follows:

```
% cd generators/powerlaw
% make
```

The generator makefiles use gcc as the compiler. You can edit the makefiles to use another compiler or if some setting needs to be changed for your machine.

The serial C++ versions of the analytics can be built from the appropriate directory as follows:

```
% cd analytics/anomaly1/c++
% make
```

The serial Python versions of the analytics do not need to be built.

Other implementations of the analytics may need to be built in different manners. See the README in the sub-directory for those implementations for details.

For example the PHISH implementation requires you have the PHISH Library installed on your box as well as either an MPI or ZMQ library that PHISH uses. Once PHISH is installed, e.g. with the MPI back-end, you can type

```
% cd analytics/anomaly1/phish
% make linux.mpi
```

to build the PHISH "minnows" that are invoked when a PHISH input script is launched.

The analytic makefiles (for c++ and PHISH) use g++ as the compiler. You can edit the makefiles to use another compiler or if some setting needs to be changed for your machine.

Testing the generators

You can test any of the generators, without running an analytic, as follows. In one window, type

```
% nc -l -u 55555
```

Then in another window, type the following (for the powerlaw generator):

```
% cd generators/powerlaw
% powerlaw 127.0.0.1
```

This should trigger a rapid output of packets and their datums in the first window. If you let the generator run long enough, it will print a summary message like the following, when it is finished generating all its packets:

```
packets emitted = 1000000
datums emitted = 50000000
elapsed time (secs) = 15.6978
generation rate (datums/sec) = 3.18516e+06
```

Neither the generator or the "nc" command will exit. Kill both of them with a Ctrl-C.

Testing the analytics

You can only test one of the anayltics by running it in tandem with the corresponding generator. The steps for doing this are as follows:

1. launch the analytic in one window
2. launch the generator in another window
3. kill the generator after the stream is complete and the analytic exits

IMPORTANT NOTE: The order of these steps is important. You must launch the analytic before launching the generator, else the analytic will miss initial packets.

You can launch the C++ version of an analytic by typing the following (for the anomaly1 analytic):

```
% cd analytics/anomaly1/c++
% anomaly1 55555
```

You can launch the Python version by typing the following (for the anomaly1 analytic):

```
% cd analytics/anomaly1/python
% python anomaly1.py 55555
```

Instructions on how to launch other versions are given in the README in the sub-directory for those implementations.

The powerlaw generator is the one to use with the anomaly1 analytic. It can be launched in another window as described above:

```
% cd generators/powerlaw
% powerlaw 127.0.0.1
```

As soon as you do this, the anomaly1 analytic should begin to find anomalies and print information to the screen. This should continue until the generator has emitted all its packets, which is 1 million (50 million datums) by default. (You can change this with a "-p N" switch on the powerlaw command if you wish.) This should take 15 to 30 seconds on a typical machine.

When the generator is finished it will print a summary of its generation statistics to the screen. The analytic should do likewise and should exit. The generator will not exit, so you need to kill it with a CTRL-C.

Note that not all of the analytics produce output as immediately as anomaly1. For example, there is a delay of a few seconds for anomaly3 to begin finding anomalies, due to its processing two levels of keys.

If your run does not drop packets, any version (C++, Python, etc) of each of the analytics should produce results identical to those in the results directory of the FireHose tarball. Those were run using a single generator for varying numbers of generated packets from 10,000 to 10 million. As discussed in the Generators section, you can set the number of packets via a "-p N" switch for any of the generators.

You can check if the analytic proocessed all the packets in the stream by comparing the "packets emitted" value printed by the generator to the "packets received" value printed by the analytic. If your run drops packets, you will likely get different output when comparing to the files included in the results directory. You can dial down the generation rate using a "-r N" swtich on any of the generators, which should eventually reduce the drop rate to 0.0. See the Generators section for details.

You can increase the stream rate by running multiple generators. The easiest way to do this is by running one from each of multiple windows. E.g. to run with two generators, launch the analytic as follows:

```
% cd analytics/anomaly1/c++
% anomaly1 -p 2 55555
```

And then launch the generators in each of two windows:

```
% cd generators/powerlaw
% powerlaw -p 2 -x 0 127.0.0.1

% cd generators/powerlaw
% powerlaw -p 2 -x 1 127.0.0.1
```

The meaning of the -p and -x switches is discussed in the Generators and Analytics sections.

If you launch both generators at (nearly) the same time, they will also finish at the same time, and the analytic will see a data stream arriving at (nearly) 2x the rate of a single generator. Again the analytic should exit when both

generators have finished, and you will then need to kill both the generators with CTRL-C.

You will quickly find that launching and killing generators and analytics in different windows is tedious work. See the sub-section below about the launch.py script in the tools directory, which automates this procedure.

Tuning your machine

The default settings within the kernel of most Linux installations are not ideal for sending and receiving UDP packets at high streaming rates with minimal packet loss. The default buffer sizes are too small to prevent loss of data when there are "hiccups" in the system due to OS operations, e.g. swapping and switching between processes.

You can query the current settings for both send ("s") and receive ("r") buffers by the following commands:

% cat /proc/sys/net/core/rmem_max % cat /proc/sys/net/core/rmem_default % cat /proc/sys/net/core/smem_max % cat /proc/sys/net/core/smem_default

If they are only a few 100 Kbytes, you should set them to values more like 16 Mbytes, via these commands:

% sysctl -w net.core.rmem_max='16777216' % sysctl -w net.core.rmem_default='16777216' % sysctl -w net.core.smem_max='16777216' % sysctl -w net.core.smem_default='16777216'

You will need sudo or root privileges to make these changes. You should write down the original values if you wish to restore them after running FireHose benchmarks.

These buffer sizes assume you have adequate memory on your box that reserving 32 Mbytes for UDP send and receive buffers is not a problem.

Benchmark #1 - anomaly detection in a fixed key range

This benchmark uses the biased-powerlaw generator and the anomaly1 analytic.

While it is running, this analytic should print information about anomalies it finds to the screen, one line at a time, flagging the key value as either a "true anomaly", "false positive", or "false negative". The meaning of these various flavors of anomalies is discussed on the anomaly1 analytic doc page.

At the end of the run, the analytic prints a summary like this:

```
packets received = 1000000
datums received = 50000000
unique keys = 85567
max occurrence of any key = 16713641
true anomalies = 34
false positives = 9
false negatives = 0
true negatives = 9645
```

If no packets were dropped, these are exactly the results you should see if a single copy of the generator produces 1 million packets (the default), i.e. 50M datums. See other files in the results directory for output from runs with fewer or more generated packets.

The number of unique keys (85K in this case) is out of the 100,000 keys the powerlaw generator samples from. See its doc page for details.

To produce an entry for the Results section, this benchmark needs to be run for 5 minutes, ideally with no dropped packets. The generation rate and number of packets necessary to do this will depend on which implementation of the analytic you use, and what machine you run it on. See the discussion below about the launch.py script which can help automate such a run.

Benchmark #2 - anomaly detection in an unbounded key range

This benchmark uses the active set generator and the anomaly2 analytic.

While it is running, this analytic should print information about anomalies it finds to the screen, one line at a time, flagging the key value as either a "true anomaly", "false positive", or "false negative". The meaning of these various flavors of anomalies is the same as for the anomaly1 benchmark, and is discussed on the anomaly2 analytic doc page.

At the end of the run, the analytic prints a summary like this:

```
packets received = 1000000
datums received = 50000000
unique keys = 34885815
true anomalies = 395
false positives = 87
false negatives = 6
true negatives = 102338
```

If no packets were dropped, these are exactly the results you should see if a single copy of the generator produces 1 million packets (the default), i.e. 50M datums. See other files in the results directory for output from runs with fewer or more generated packets.

Note that the number of unique keys, as well as the number of anomalies found is much higher than for the anomaly1 benchmark. This is because the active set generator samples from an unbounded range of keys. See its doc page for details.

To produce an entry for the Results section, this benchmark needs to be run for 30 minutes, ideally with no dropped packets. This is so that the expiration strategy for the hash table of the analytic is stress-tested. The generation rate and number of packets necessary to do this will depend on which implementation of the analytic you use, and what machine you run it on. See the discussion below about the launch.py script which can help automate such a run.

Benchmark #3 - anomaly detection in an two-level key space

This benchmark uses the two-level generator and the anomaly3 analytic.

While it is running, this analytic should print information about anomalies it finds to the screen, one line at a time, flagging the key value as either a "true anomaly", "false positive", or "false negative". The meaning of these various flavors of anomalies is the same as for the anomaly1 benchmark except that they apply not to keys in the data stream, but to "inner" keys derived from their values, as discussed on the anomaly3 analytic doc page.

At the end of the run, the analytic prints a summary like this:

```
packets received = 1000000
datums received = 40000000
unique outer keys = 24005720
unique inner keys = 475860
true anomalies = 15
false positives = 2
false negatives = 0
true negatives = 5204
```

If no packets were dropped, these are exactly the results you should see if a single copy of the generator produces 1 million packets (the default), i.e. 40M datums. See other files in the results directory for output from runs with fewer or more generated packets.

Note that statistics are given for both "outer" keys (in the data stream) and "inner" keys (derived from the values of outer keys). As with the anomaly2 benchmark, both of these kinds of keys are sampled from unbounded ranges.

To produce an entry for the Results section, this benchmark needs to be run for 30 minutes, ideally with no dropped packets. This is so that the expiration strategy for the hash tables of the analytic are stress-tested. The generation rate and number of packets necessary to do this will depend on which implementation of the analytic you use, and what machine you run it on. See the discussion below about the launch.py script which can help automate such a run.

Using the launch.py script to run benchmarks

The tools directory of the FireHose distribution has Python and shell scripts which are useful when running benchmarks or when running the generators and analytics on their own.

The launch.py script can be used for the following tasks:

- Run a benchmark, with one or more generators and an analytic together, for a specified time or at a desired rate.
- Run a benchmark interatively at slower rates until the drop rate is acceptable, then perform a benchmark run.
- Run one or more instances of a generator with various command-line options.
- Run an analytic with various command-line options, including analytics like PHISH versions that require specialized launch commands or post-processing of results to format them for scoring.
- Perform a benchmark run (possibly with dropped packets), then perform a second benchmark run (at a slower rate, or with the C++ reference implementaion) with no drops.
- Compute a "score" between two runs: one with dropped packets and one without.

The most convenient attribute of launch.py is that it manages all the "processes" that are part of a benchmark run, namely one or more generator proceses and an analytic process. It launches them in the correct sequence, monitors and collects their output, and cleans up after they have finished, killing them as needed. A summary of the run is printed to the screen and a log file for later review.

As a simple example, the following command could be used to run the C++ version of the first benchmark at a stream rate of 5.6M datums per second (which requires 2 generators) for 5 minutes, to generate the results needed for an entry in the table of the Results section.

```
python launch.py bench -bname 1 -gnum 2 -grate 5600000 -gtime 5 -adir c++ -id bench1.c++
```

The output from the various processes are written to these files:

- generator.bench1.c++.0 = output from 1st generator
- generator.bench1.c++.1 = output from 2nd generator
- analytic.bench1.c++ = output from analytic (list of found anomalies)
- log.bench1.c++ = summary of run, including drop rate

A single launch.py command can also be used to perform a benchmark run followed by a 2nd no-drop run and "score" the results of the 1st against the 2nd, to compare the accuracy. The scoring metrics are explained in the Results section.

Examples of using the launch.py script to perform multiple, successive benchmark runs for various tasks, are illustrated in the bench.py script, also included in the tools directory. For example, if you know Python, it is easy to write a loop in bench.py to loop over a series of short runs, each invoked with launch.py, that generate datums at different stream rates, to quickly estimate a maximum stream rate that can be processed without dropped packets.

The full documentation for launch.py and its options is listed at the top of the launch.py file, and included here. In brief, there is one required argument which is the "action" to perform, which is one of the following:

```
# gen = run the generator only
# analytic = run the analytic only
# bench = run a benchmark with generator and analytic together
# score = score a pair of already created output files
# bench/score = perform 2 benchmark runs and score them
#   1st run with specified settings
#   2nd run with -adrop 0.0
```

Then there are optional switches for the generator (all of which have defaults):

```
# -gname powerlaw/active/... = which generator to run (def = powerlaw)
# -gcount N = total # of packets for all generators to emit (def = 1000000)
# -grate max/N = aggregate rate (datums/sec) for all generators (def = max)
# -gnum N = # of generators to run (def = 1)
# -gtime N = minutes to run all generators (def = 0)
#   if non-zero, overrides -gcount
#   iterates over short test runs to reset -gcount
# -gtimelimit N = time limit in short iterative -gtime runs (def = 10.0)
# -gswitch N string = command-line switches for Nth generator launch
#     (def = "" for all N)
#   N = 1 to gnum
#   enclose string in quotes if needed so is a single arg
#   note: these switches are added to ones controlled by other options
#   note: -r, -p, -x are controlled by other options
#   note: can be specified multiple times for different N
# -gargs N string = arguments for Nth generator launch (def = "" for all N)
#   N = 1 to gnum
#   enclose string in quotes if needed so is a single arg
#   note: default of "" means 127.0.0.1 (localhost) will be used
#   note: can be specified multiple times for different N
```

and likewise for the analytic:

```
# -aname anomaly1/anomaly2/... = which analytic to run (def = anomaly1)
# -adir c++/python/phish/... = which flavor of analytic to run (def = c++)
# -adrop any/N = target drop rate percentage for analytic (def = any)
#   if not any, overrides -grate setting
#   iterates over short test runs to reset -grate
# -adropcount N = use for -gcount in short iterative -adrop runs (def = 1000000)
# -afrac fraction = drop -grate computed by -adrop by this fraction
#                   to insure no drops, only used if -adrop 0.0 (def = 1.0)
```

```
# -acmd string = string for launching analytic (def = "")
#   enclose string in quotes if needed so is a single arg
#   not needed for -adir c++/python
#     unless you need to change command line settings
#   required for other -adir settings
#     e.g. this script doesn't know how to launch a PHISH job
#   note: script will cd to -adir setting before cmd is invoked
# -apost string = command to invoke to post-process analytic output into
#                 expected format, e.g. to merge multiple output files,
#                 launch.py treats stdout of invoked string as file content
#                 note: will invoke command from within -adir directory
```

Finally, there are a few generic optional switches that can be specified:

```
# -id string = string to append to all output file names (def = "launch")
# -bname N = which benchmark to run (def = 0)
#   if non-zero, overrides -gname and -aname
# -table no/yes = if scoring, also output table format (def = no)
# -fdir path = FireHose directory (def = ~/firehose)
```

If you don't want to use the "launch.py" tool, the simpler "launch.sh" script can be edited to launch multiple versions of a generator (nearly) simultaneously. It is written for use with the "tcsh" shell, but similar scripts could be created for "bash" or other shells. E.g. you can run it by typing

```
% tcsh launch.sh
```

Note that the script launches the generators in the background. You should see output from all instances of the generator in the window you run the script from. After they complete, you MUST be sure to explicitly kill each of the generator processes, else they may continue to run, unnoticed in the background. See the next section for further discussion.

Killing instances of running generators

IMPORTANT NOTE: Because a generator process does not terminate until explictly killed, it is possible for one or more instances of running generator to persist after a benchmark run completes. Because a running generator continues to send its STOP packets in an infinite loop, it will continue to consume system resources and may confuse an analytic the next time the analytic is launched (e.g. it immediately terminates without processing new packets).

An unkilled generator can results from several scenarios:

- a benchmark run crashes
- you kill the benchmark run before it ends
- the launch.sh script is used to launch generator(s) in the background
- the launch.py or bench.py scripts encounter some error condition

The best way to insure no generators are currently running are to type commands like this:

% ps -ef | grep powerlaw % ps -ef | grep active % ps -ef | grep twolevel

to identify their process IDs.

Then a kill command such as this:

```
% kill 93985
```

can be invoked on the ID to explicitly kill a running generator.

Results

There are FireHose benchmark results for the currently defined benchmarks:

1. anomaly #1 = anomaly detection in a fixed key range
2. anomaly #2 = anomaly detection in an unbounded key range
3. anomaly #3 = anomaly detection in a two-level key space

They were run as described in the Running the benchmarks section, on these Machines.

For each benchmark a table is shown below with the following columns of information:

- Version = which version of the analytic was run
- Machine = what machine the benchmark was run on
- Ngen = # of generators used
- Ndata = # of datums generated
- Grate = aggregate stream rate for all generators, in datums/sec
- Cores = # of cores the analytic was run on
- Packets = percentage of packets or datums processed, 100.0 = no drops
- Events = total events for the analytic to detect
- Score = score for the analytic calculation, 0 = perfect (see discussion below)
- Instant = no/yes for whether the analytic produces its answers instantly when a datum triggers a result, or whether there is some delay in the algorithm it uses
- LOC = lines-of-code required to implement the analytic
- Comments = footnotes to additional comments

Some implementations of the analytic will have multiple entries, e.g. if they were run on different machines, at different stream rates, or in parallel on different numbers of processors.

The "Packets" entry is the percentage of emitted packets that were received and processed by the analytic. Thus a value of 99.99% means that 1 in 10,000 packets was dropped. Ditto for datums since each packet contains a fixed number of datums.

The "Events" entry is a count of possible "events" within the stream for the analytic to find. Note that this is a function of how much data is generated during the benchmark run. Since each run is for the same length of time, runs at a faster stream rate will generate more datums and have a higher event count. Each benchmark defines what it counts as an "event".

For the "Score" entry, the value is meant to measure deviations from the correct result, either due to dropped packets, or invalid computations by the analytic. A low score is good and a high score is bad.

The score is computed by comparing the results from the benchmark run in the table to results from a run by the C++ reference implementation which is run so that it sees the identical stream from the generator(s) at a rate slow enough to insure no packets are dropped. The latter is considered the correct result.

Each benchmark defines how it computes a score, as discussed below. Typically one point is assigned for missed or mistaken events, so the maximum score is roughly that of the event count. But it's possible for the score to exceed the event count, e.g. finding no actual events and many incorrect events.

If the benchmark run completed with no dropped packets, it should receive a score of 0 if the analytic performed its computations correctly. The score will typically increase as more and more packets are dropped. How sensitive the score is to dropped packets depends on the benchmark.

There are at least 2 reasons why a no-drop run could result in a small non-zero score. First, when multiple generators are used, packets arrive at the analytic in an indeterminate order. This can alter the events detected by the analytic. Second, for benchmarks which use expiring hash tables to track keys, there can be small differences between the keys the generator keeps in its active set (and thus may emit) versus keys the analytic deems to be active. If the generator emits a key the analytic has already discarded, this can lead to a difference in a event counts (versus what the generator counts as events). Ditto if comparing two runs where the order of packets may vary between them due to using multiple generators. If the guidelines discussed in the Analytics section are followed for expiring hash tables, these differences should be minimal.

The "LOC" is a lines-of-code count for the files in the analytics sub-directories, including comments and blank lines. For frameworks like PHISH, only lines in scripts or code added to the framework to run the benchmark are counted; lines in the framework code itself are not included.

In all the cases currently listed in the tables below, we attempted to run the benchmarks at the maximum stream rate a particular version of the analytic could process the stream and still maintain a (nearly) zero drop rate.

---

(1) Benchmark #1 - anomaly detection in a fixed key range

This benchmark is discussed here and uses the biased-powerlaw generator and anomaly1 analytic.

The table lists results for runs of 5 minutes. A "K" in the table means thousand, an "M" means million, a "B" means billion.

Version	Machine	NGen	Ndata	Grate	Cores	Packets	Events	Score	Instant	LOC	Comments
C++	Dell	2	1.54 B	5.6 M/sec	1	100%	119 K	0	yes	275	none
Python	Dell	1	135 M	450 K/sec	1	100%	18.7 K	0	yes	190	none
PHISH/C++ 1/0/1	Dell	2	1.55 B	5.5 M/sec	2	100%	120 K	0	yes	520	(a)
PHISH/C++ 1/4/2	Dell	4	3.00 B	10.0 M/sec	7	100%	230 K	0	yes	525	(b)

The event entry is the sum of true anomalies, false positives, false negatives, and true negatives found in the correct result.

One score point is tallied for each key in each true anomaly, false positive, or false negative output of the correct result but which is not in the benchmark result. Likewise one point is tallied for keys in those 3 categories in the benchmark result but which are not in the correct result. Individual keys which true negatives are not included in the results (there are a lot of them), but the total key counts are. The absolute value of the difference between the true negative totals in the 2 runs is also added to the score.

Comments:

(a) A 2-process PHISH run on top of MPI, using the in.anomaly script. One process reads packets, the other performs the anomaly detection. PHISH runs as a distributed-memory parallel program.

(b) A 7-process PHISH run on top of MPI, using the in.parallel script. One minnow (process) reads packets, 4 minnows re-bundle them by hashing the keys, and 2 minnows perform the analytic computation, each on a subset of the key space. PHISH runs as a distributed-memory parallel program.

---

(2) Benchmark #2 - anomaly detection in an unbounded key range

This benchmark is discussed here and uses the active set generator and anomaly2 analytic.

The table lists results for runs of 30 minutes. A "K" in the table means thousand, an "M" means million, a "B" means billion.

Version	Machine	NGen	Ndata	Grate	Cores	Packets	Events	Score	Instant	LOC	Comments
C++	Dell	1	3.42 B	1.9 M/sec	1	100%	3.73 M	0	yes	415	none
Python	Dell	1	252 M	140 K/sec	1	100%	432 K	0	yes	290	none
PHISH/C++ 1/0/1	Dell	1	3.42 B	1.9 M/sec	2	100%	3.74 M	0	yes	660	(a)
PHISH/C++ 1/4/2	Dell	2	6.12 B	3.4 M/sec	7	100%	6.80 M	10	yes	665	(b)

The event and score entries are computed in the same manner as for benchmark #1.

(a) See the (a) comment for benchmark #1.

(b) See the (b) comment for benchmark #1.

---

(3) Benchmark #3 - anomaly detection in a two-level key space

This benchmark is discussed here and uses the two-level generator and anomaly3 analytic.

The table lists results for runs of 30 minutes. A "K" in the table means thousand, an "M" means million, a "B" means billion.

Version	Machine	NGen	Ndata	Grate	Cores	Packets	Events	Score	Instant	LOC	Comments
C++	Dell	1	2.70 B	1.5 M/sec	1	100%	1.12 M	0	yes	495	none

The event and score entries are computed in the same manner as for benchmark #1.

The events detected by this benchmark, and thus the score, are quite sensitive to dropped packets, since a missed packet means an instance of an "inner" key cannot be constructed, even if other packets contributing to that key are received. Thus a low drop rate can still induce a poor score.

Machines

These are the machines listed in the tables above:

Dell = desktop machine running RedHat Linux, with dual hex-core 3.47 GHz Intel Xeon (X5690) CPUs. The generator(s) and the analytic were both run together on the same box.