Improving go graphite stroage stack

% Xiaofan Hu

What is Graphite

Graphite is a time-series database. It was originally written in python (mainly). There are lots of components including:

- relay
- storage
- api/webapp

Figure 1: architecture

(copy from https://github.com/graphite-project/whisper)

Graphite at Booking

Various components in python are being rewritten, for example:

- carbonapi, rewritten by Damian Gryski, Vladimir Smirnov and many others.
- relay is now nanotube, it was carbon-c-relay
- go-carbon for storage

My story today is mainly about the storage program: go-carbon.

A typical graphite metric

sys.app.host-01.cpu.loadavg

What is Whisper

In graphite, each metric is saved in a file, using the a round-robin database format, named whisper.

- This means that every data point is positionable in a whisper file.
- In whisper file, each data point is saved in 12 bytes (4 bytes for timestamp, 8 bytes for value)

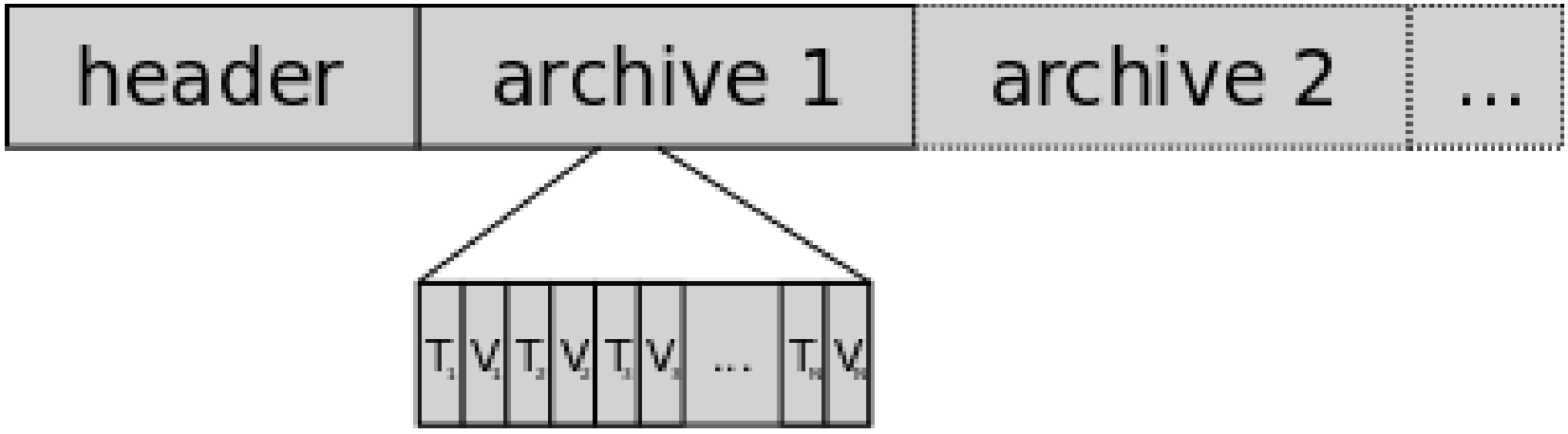

Figure 2: whisper

What is Gorilla compression

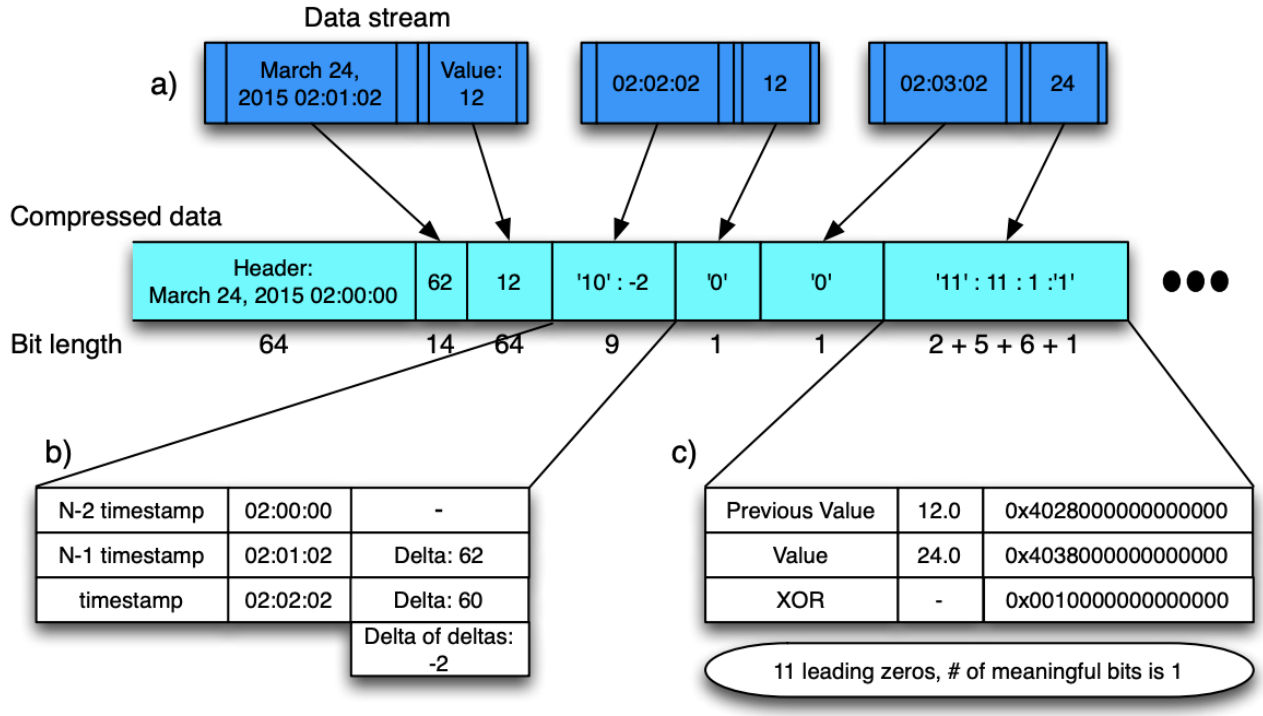

Figure 2: Visualizing the entire compression algorithm. For this example, 48 bytes of values and time stamps are compressed to just under 21 bytes/167 bits.

Figure 3: gorilla

With the compression algorithms introduced in the gorilla paper, a best case of saving a data point could be done in a 2 bits

How to comibine Gorilla and Whisper

A new file format needs to be designed in order to compress data points in gorilla.

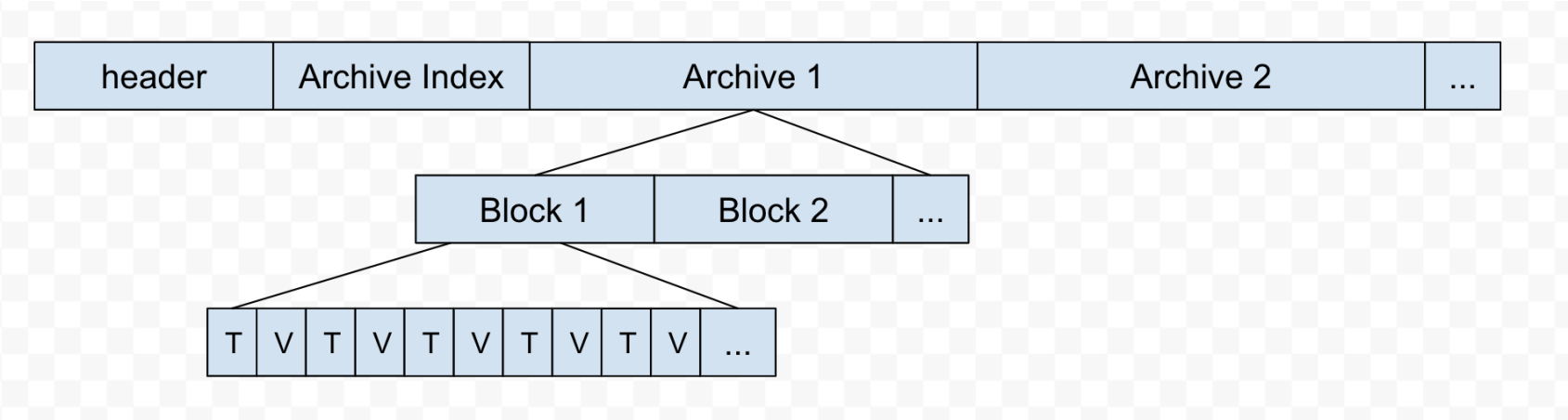

Figure 4: cwhisper

Result

Metrics	Whisper (standard)	CWhisper (compressed)
Total Metrics	50.6 Millions	53.1 Millions
Num of Servers	32	9
Disk Usage (**45.75% less**)	32.28 TB	14.77 TB
Total Disk Space (2.9TB Per Server)	92.8 TB	26.1 TB
Theoretical Capacity Per Server (Metrics)	~4.5 Millions	~10.43 Millions

Figure 5: compression performance

Globbing graphite metrics

Using Standard library

Pro: Simple
Con: High performance cost in a large file tree (millions of files)
Glob is a userspace implementation, so it first needs to ask the kernel returning all the files and then glob over it.

Using Trigram

Originally implemented by Damian Gryski.
TLDR: it breaks downs all the metrics as trigrams, and maps the trigram to the metrics (an inverted index). A glob query is also convert as a trigrams, then intersects the metric trigrams and query trigrams, then it would use the glob to make sure the files match the query.
Pro: faster than standard library (no syscalls after index, and file list are cached in memory)
Con: index is expensive to build when dealing higher number of metrics (above 5 millions or more).
Minor: corner cases like if part of the metric name has one or two chars, need to use filepath.Match to double check if files are really matched, etc.

What is trie, NFA/DFA

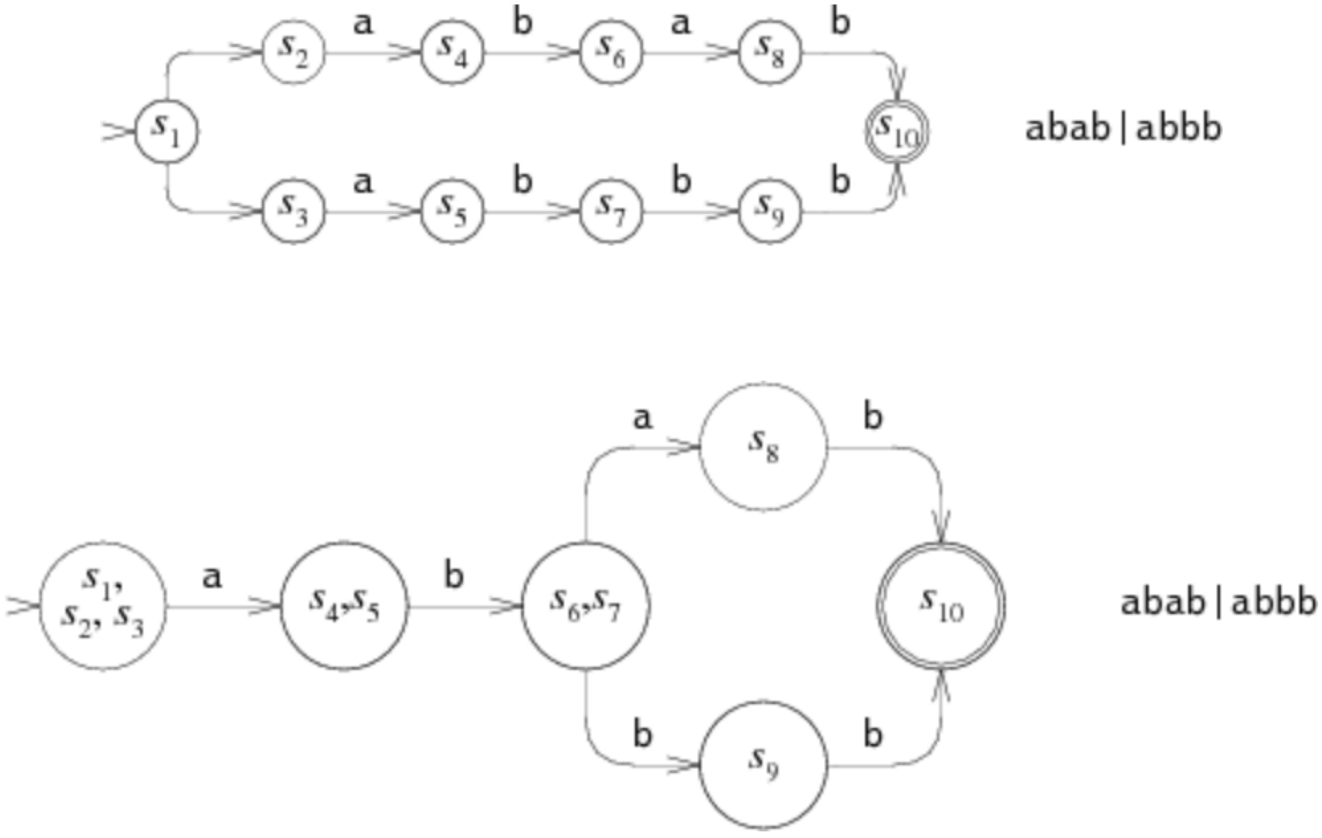

Figure 6: nfa_dfa

The new index solution

TLDR: index all the metrics in go-carbon instance with trie, compile the glob queries first as nfa (then dfa during walking). And walking over the trie and nfa/dfa at the same time.
Pro:

- faster index time
- less memory usage
- no standard library fallback
- better/predictable performance

Con:

- it's my baby, it's perfect.
- some queries are faster with trigram (like foo.*bar.zoo, because of the leading star, the new index algorithm needs to travel the whole namespace, however, arguably, you can design your metric namespace properly to avoid this issue)

Result

Time Range	Trigram	Trie+DFA
1µs-10µs	1621	0
10µs-100µs	104911	85662
100µs-1ms	20617	74514
1ms-10ms	18214	19454
10ms-100ms	34601	4164
1m40s-16m40s	11	418
100ms-1s	3851	6
1s-10s	219	0
10s+	21	0
Total	184066	184218
Queries Finished in 10ms	78.97%	97.51%

Figure 7: trie/dfa-vs-trigram

Production and Community usage status

Restropection