RedisTimeSeries 1.0 release notes

Requirements

RedisTimeSeries v1.0.3 requires:

- Minimum Redis compatibility version (database): 5.0.0
- Minimum Redis Enterprise Software version (cluster): 5.4.0

v1.0.3 (September 2019)

Update urgency: Medium

This is a maintenance release for version 1.0.

This release improves overall stability and provides fixes for issues found after the previous release.

Main features:

- #143 Standard Deviation for Aggregations
- #163 TS.RANGE and TS.MRANGE can limit results via optional COUNT flag
- #161 Support for ARM architectures
- #160 Optional TIMESTAMP in TS.INCRBY and TS.DECRBY

Main Fixes:

- #199 RETENTION is now 64bit
- #211 write commands to return OOM error when redis reaches max memory

Main Performance improvements:

- #3651 Do not use _union if there's only 1 leaf in the index
- #0a68 Make _difference faster by iterating over the left dict (which is always smaller)

v1.0.1 (July 2019)

Update urgency: Minor

This is a maintenance release for version 1.0.

Secondary index should work faster when a filter consists of a list of k=v predicates.

v1.0.0 (June 2019)

This is the General Availability release of RedisTimeSeries! Read the full story here

Features
In RedisTimeSeries, we are introducing a new data type that uses chunks of memory of fixed size for time series samples, indexed by the same Radix Tree implementation as Redis Streams. With Streams, you can create a capped stream, effectively limiting the number of messages by count. In RedisTimeSeries, you can apply a retention policy in milliseconds. This is better for time series use cases, because they are typically interested in the data during a given time window, rather than a fixed number of samples.

**Downsampling/compaction**

If you want to keep all of your raw data points indefinitely, your data set will grow linearly over time. However, if your use case allows you to have less fine-grained data further back in time, downsampling can be applied. This allows you to keep fewer historical data points by aggregating raw data for a given time window using a given aggregation function. RedisTimeSeries supports downsampling with the following aggregations: avg, sum, min, max, range, count, first and last.

**Secondary indexing**

When using Redis' core data structures, you can only retrieve a time series by knowing the exact key holding the time series. Unfortunately, for many time series use cases (such as root cause analysis or monitoring), your application won't know the exact key it's looking for. These use cases typically want to query a set of time series that relate to each other in a couple of dimensions to extract the insight you need. You could create your own secondary index with core Redis data structures to help with this, but it would come with a high development cost and require you to manage edge cases to make sure the index is correct.

RedisTimeSeries does this indexing for you based on field value pairs (a.k.a. labels) you can add to each time series, and use to filter at query time (a full list of these filters is available in our documentation). Here's an example of creating a time series with two labels (sensor_id and area_id are the fields with values 2 and 32 respectively) and a retention window of 60,000 milliseconds:

```
TS.CREATE temperature RETENTION 60000 LABELS sensor_id 2 area_id 32
```

**Aggregation at read time**

When you need to query a time series, it's cumbersome to stream all raw data points if you're only interested in, say, an average over a given time interval. RedisTimeSeries follows the Redis philosophy to only transfer the minimum required data to ensure lowest latency. Below is an example of aggregation query over time buckets of 5,000 milliseconds with an aggregation function:

```
127.0.0.1:6379> TS.RANGE temperature:3:32 1548149180000 1548149210000 AGGREGATION avg 5000
1) (integer) 1548149180000
2) "26.199999999999999"
3) (integer) 1548149185000
2) "27.399999999999999"
4) (integer) 1548149190000
2) "24.800000000000001"
5) (integer) 1548149195000
2) "23.199999999999999"
6) (integer) 1548149200000
2) "25.199999999999999"
7) (integer) 1548149205000
2) "28.0"
8) (integer) 1548149210000
2) "20.0"
```

**Integrations**

RedisTimeSeries comes with several integrations into existing time series tools. One such integration is our RedisTimeSeries adapter for Prometheus, which keeps all your monitoring metrics inside RedisTimeSeries while leveraging the entire Prometheus ecosystem. Furthermore, we also created direct integrations for Grafana and Telegraph. This repository contains a docker-compose setup of RedisTimeSeries, its remote write adaptor, Prometheus and Grafana. It also comes with a set of data generators and pre-built Grafana dashboards.