redis

RedisTimeSeries

RedisTimeSeries is a Redis module developed by Redis to enhance your experience managing time series data with Redis.

Features

- Query by start time and end-time
- Query by labels sets
- Aggregated queries (Min, Max, Avg, Sum, Range, Count, First, Last) for any time bucket
- Configurable max retention period
- Compaction/Roll-ups - automatically updated aggregated timeseries
- labels index - each key has labels which allows query by labels

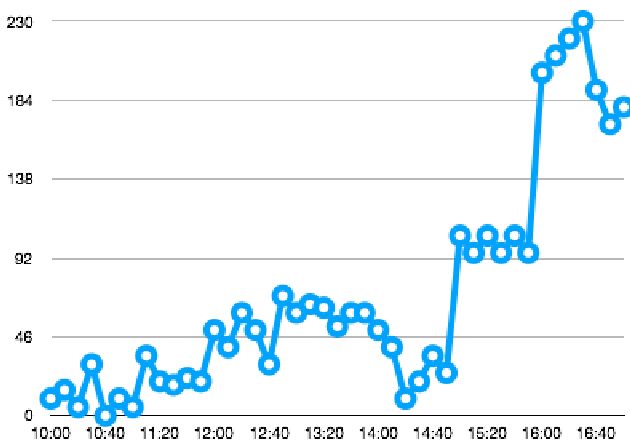
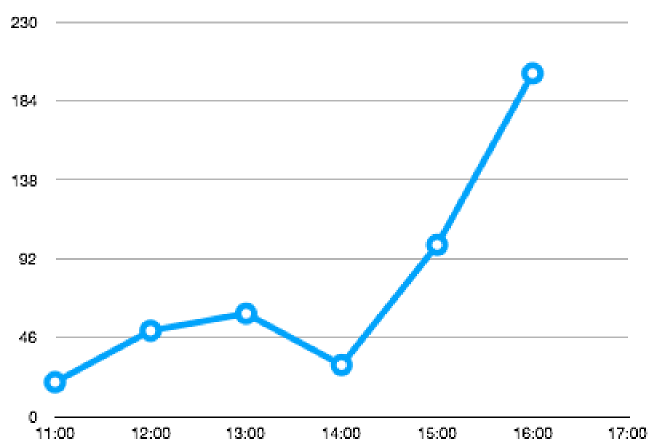
Memory model

A time series is a linked list of memory chunks. Each chunk has a predefined size of samples. Each sample is a tuple of the time and the value of 128 bits, 64 bits for the timestamp and 64 bits for the value.

RedisTimeSeries capabilities

In RedisTimeSeries, we introduce 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

Before Downsampling	After Downsampling
	

If you want to keep all of your raw data points indefinitely, your data set grows 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
```

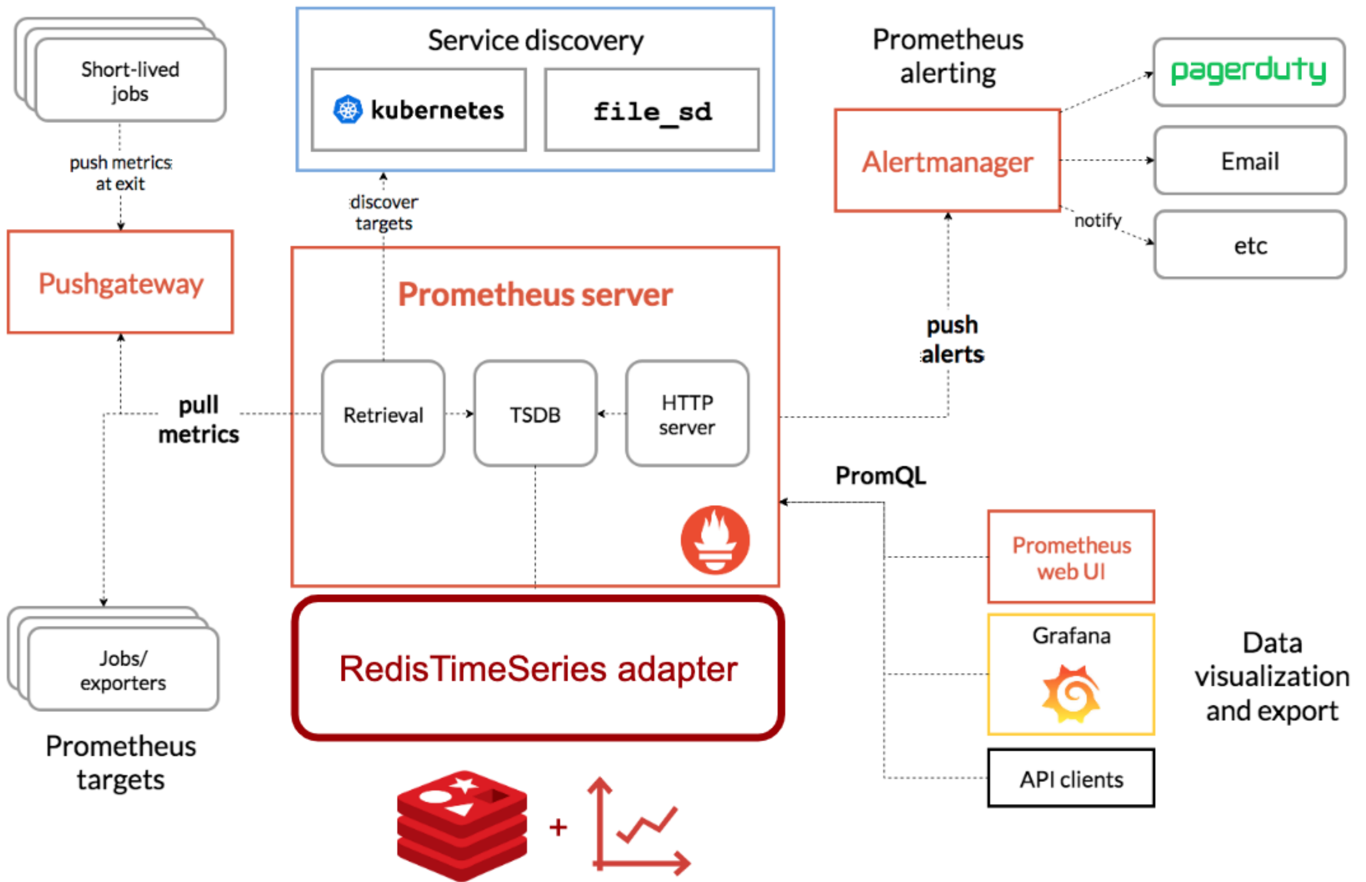
Aggregation at read time

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:

```
    127.0.0.1:12543> TS.RANGE temperature:3:32 1548149180000 1548149210000 AGGREGATION avg 
5000
    1) 1) (integer) 1548149180000
       2) "26.199999999999999"
    2) 1) (integer) 1548149185000
       2) "27.399999999999999"
    3) 1) (integer) 1548149190000
       2) "24.800000000000001"
    4) 1) (integer) 1548149195000
       2) "23.199999999999999"
    5) 1) (integer) 1548149200000
       2) "25.199999999999999"
    6) 1) (integer) 1548149205000
       2) "28"
    7) 1) (integer) 1548149210000
       2) "20"
```

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 entirePrometheus ecosystem.

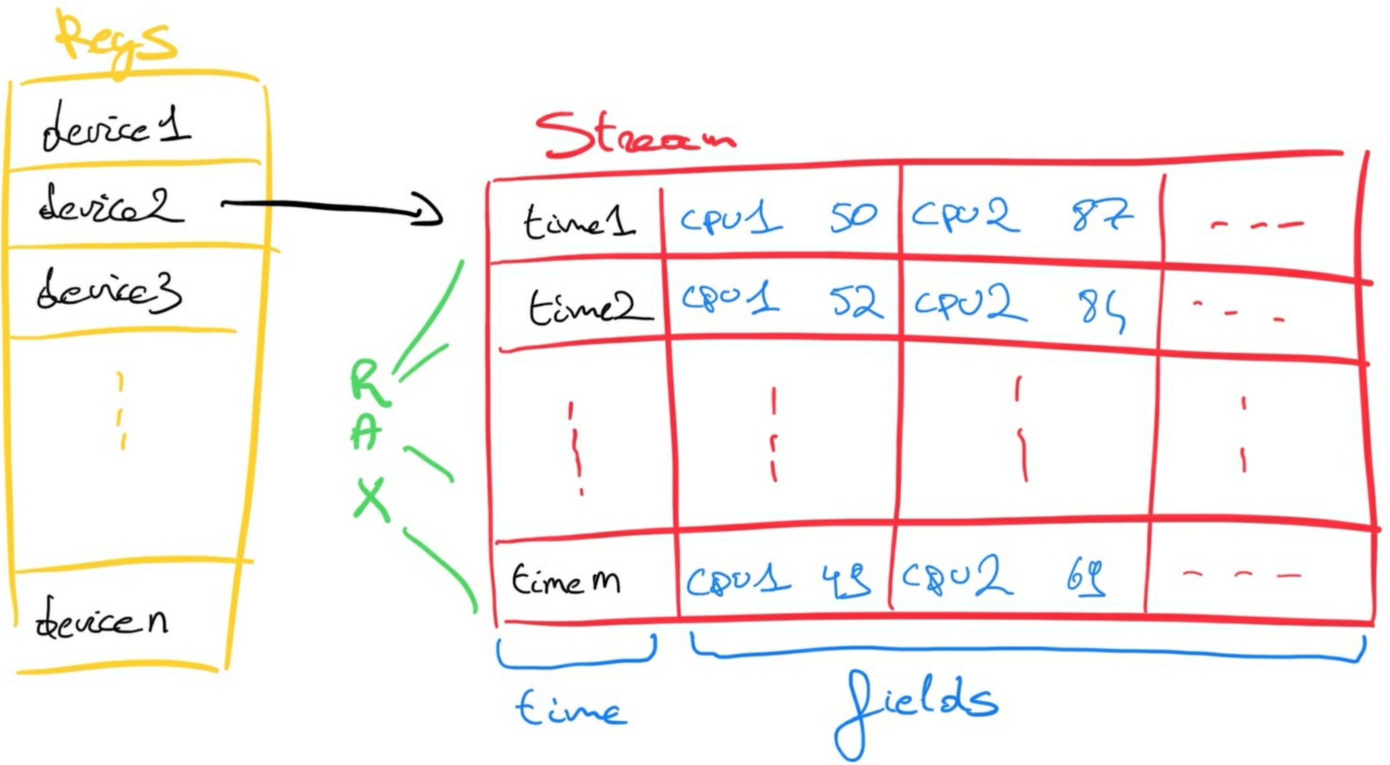


Furthermore, we also created direct integrations for Grafana and Telegraph. This repository contains a docker-compose setup of RedisTimeSeries, its remote write adaptor, Prometheus and Grafana. It also comes with a set of data generators and pre-built Grafana dashboards.

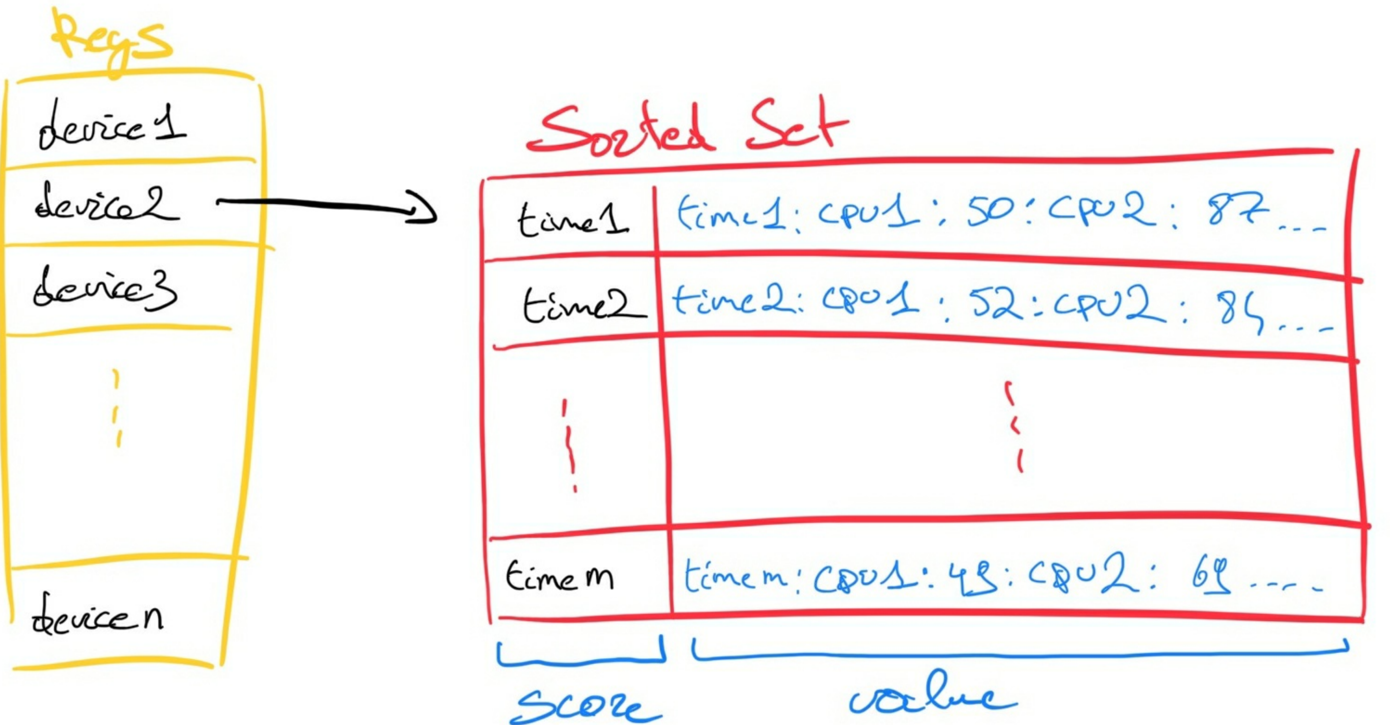
Time series modelling approaches with Redis

Data modelling approaches

Redis Streams allows you to add several field value pairs in a message for a given timestamp. For each device, we collected 10 metrics that were modelled as 10 separate fields in a single stream message.



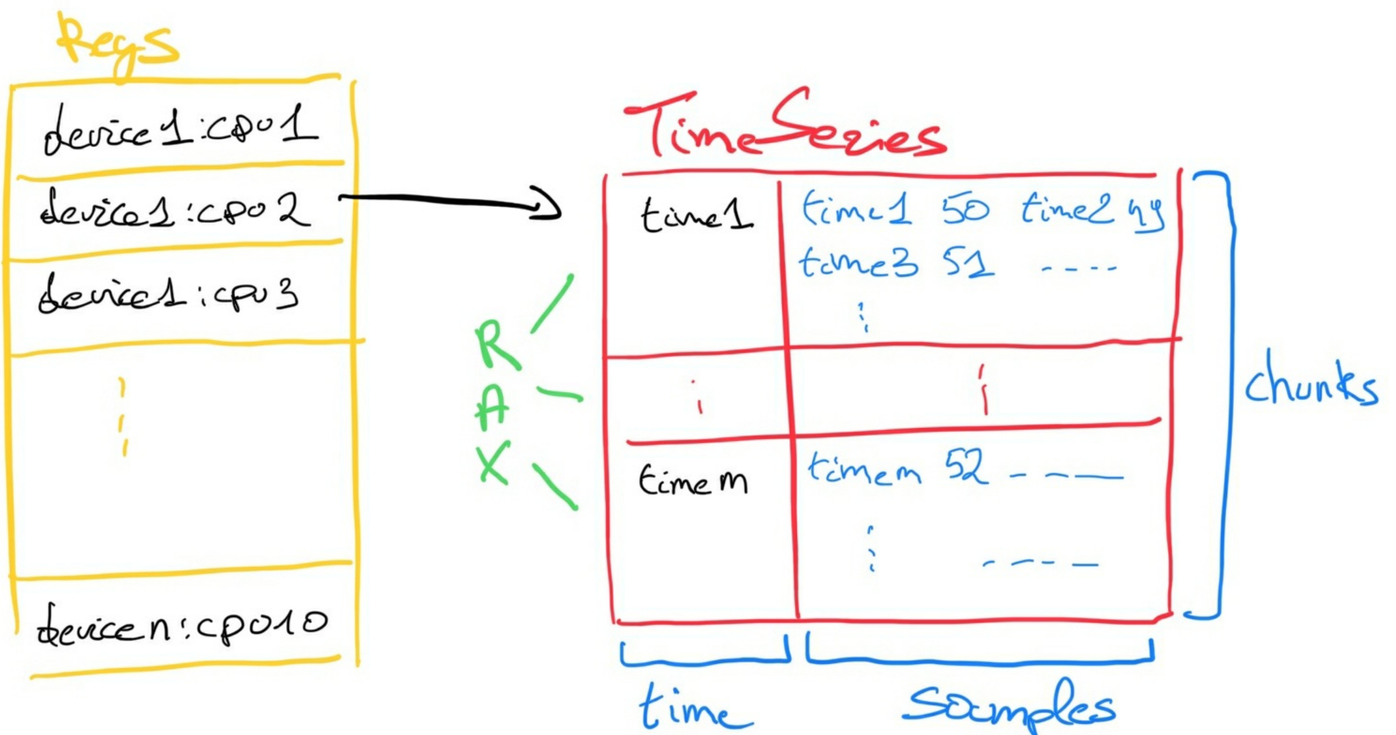
For Sorted Sets, we modeled the data in two different ways. For “Sorted Set per Device”, we concatenated the metrics and separated them out by colons, e.g. “<timestamp>:<metric1>:<metric2>: … :<metric10>”.



Of course, this consumes less memory but needs more CPU cycles to get the correct metric at read time. It also implies that changing the number of metrics per device isn’t straightforward, which is why we also benchmarked a second Sorted Set approach. In “Sorted Set per Metric,” we kept each metric in its own Sorted Set and had 10 sorted sets per device. We logged values in the format “<timestamp>:<metric>”.

Another alternative approach would be to normalize the data by creating a hash with a unique key to track all measurements for a given device for a given timestamp. This key would then be the value in the sorted set. However, having to access many hashes to read a time series would come at a huge cost during read time, so we abandoned this path.

In RedisTimeSeries, each time series holds a single metric. We chose this design to maintain the Redis principle that a larger number of small keys is better than a fewer number of large keys.



It is important to note that our benchmark did not utilize RedisTimeSeries' out-of-the-box secondary indexing capabilities. The

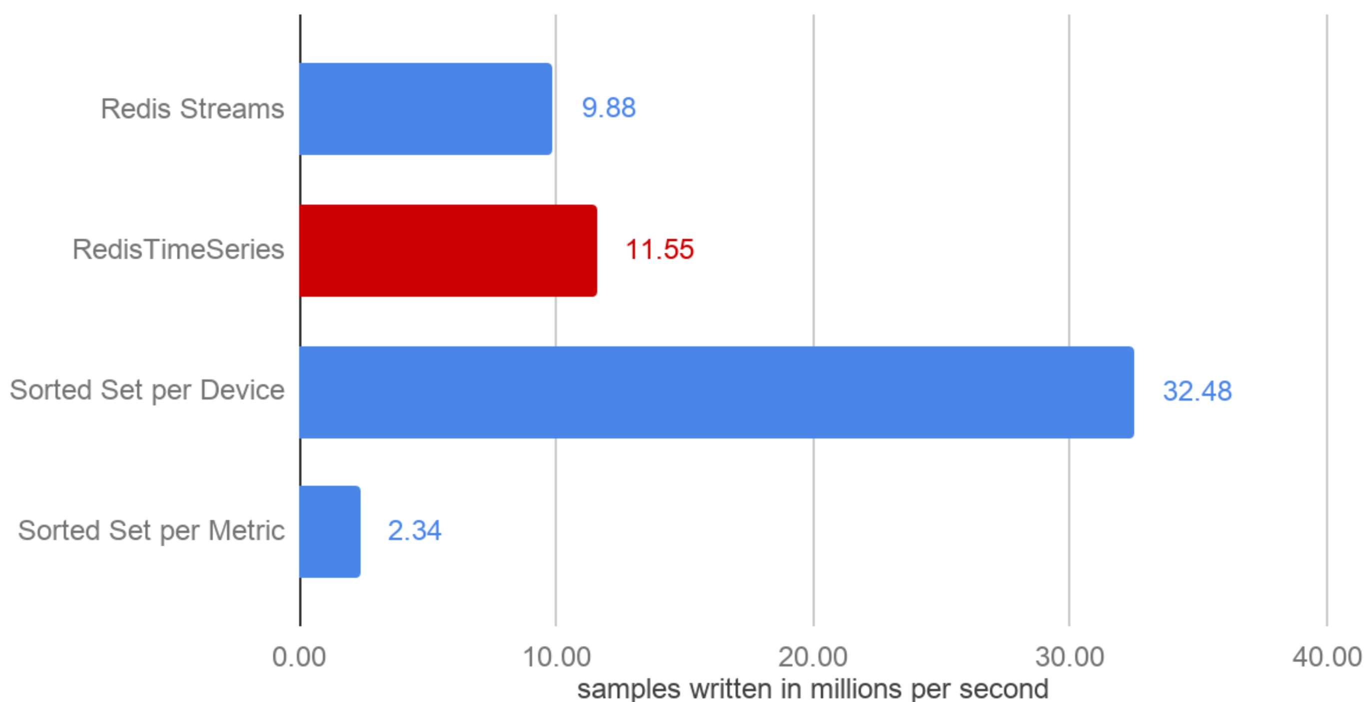
module keeps a partial secondary index in each shard, and since the index inherits the same hash-slot of the key it indices, it is always hosted on the same shard. This approach would make the setup for native data structures even more complex to model, so for the sake of simplicity, we decided not to include it in our benchmarks. Additionally, while Redis Enterprise can use the proxy to fan out requests for commands like TS.MGET and TS.MRANGE to all the shards and aggregate the results, we chose not to exploit this advantage in the benchmark either.

Data ingestion

For the data ingestion part of our benchmark, we compared the four approaches by measuring how many devices' data we could ingest per second. Our client side had 8 worker threads with 50 connections each, and a pipeline of 50 commands per request.

Ingestion details of each approach:

	Redis Streams	RedisTimeSeries	Sorted Set per Device	Sorted Set per Metric
Command	XADD	TS.MADD	ZADD	ZADD
Pipeline	50	50	50	50
Metrics per request	5000	5000	5000	500
# keys	4000	40000	4000	40000



All our ingestion operations were executed at sub-millisecond latency and, although both used the same Rax data structure, the RedisTimeSeries approach has slightly higher throughput than Redis Streams.

Each approach yields different results, which shows the value of prototyping against specific use cases. As we see on query performance, the Sorted Set per Device comes with improved write throughput but at the expense of query performance. It's a trade off between ingestion, query performance, and flexibility (remember the earlier data modeling remark).

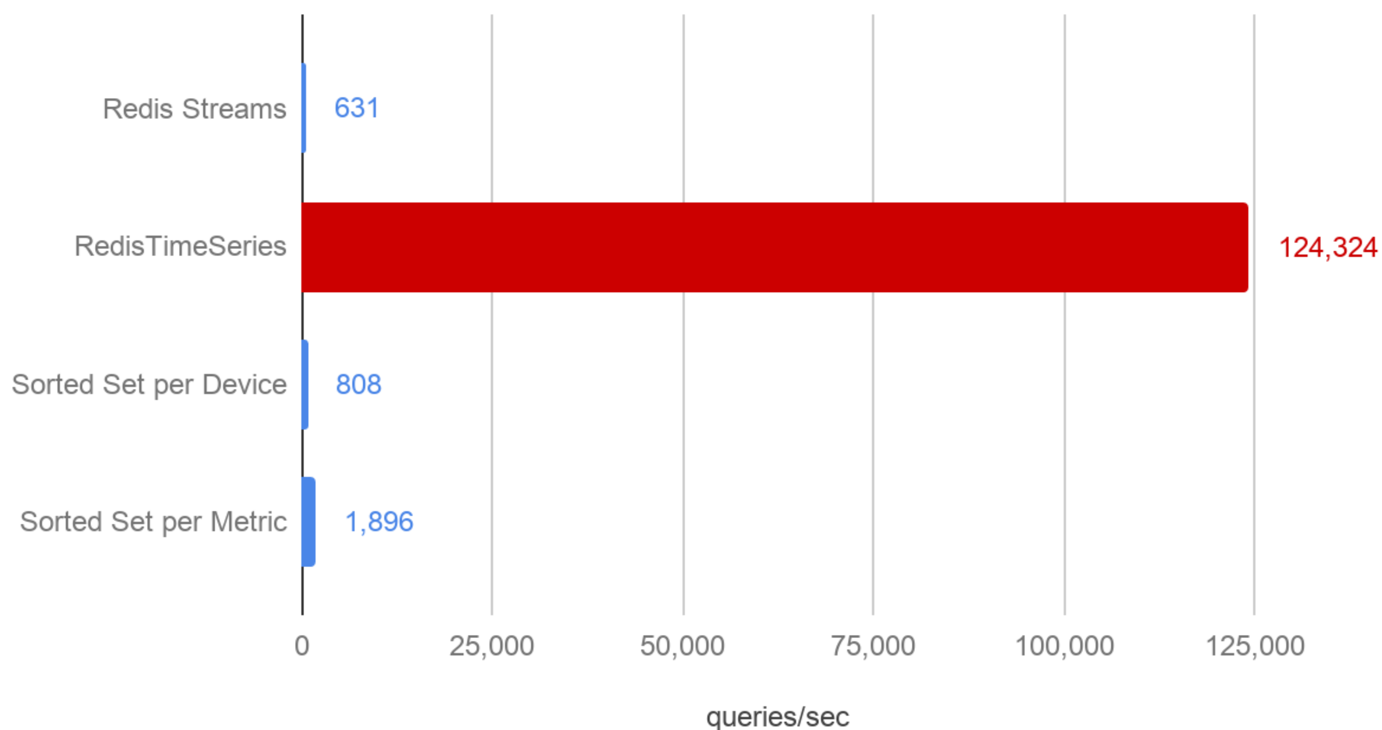
Read performance

The read query we used in this benchmark queried a single time series and aggregated it in one-hour time buckets by keeping the

maximum observed CPU percentage in each bucket. The time range we considered in the query was exactly one hour, so a single maximum value was returned. For RedisTimeSeries, this is out of the box functionality (as discussed earlier).

```
    127.0.0.1:12543> TS.RANGE cpu_usage_user{1340993056} 1451606390000 1451609990000
AGGREGATION max 3600000
```

For the Redis Streams and Sorted Sets approaches, we created the following LUA scripts. The client once again had 8 threads and 50 connections each. Since we executed the same query, only a single shard was hit, and in all four cases this shard maxed out at 100% CPU.

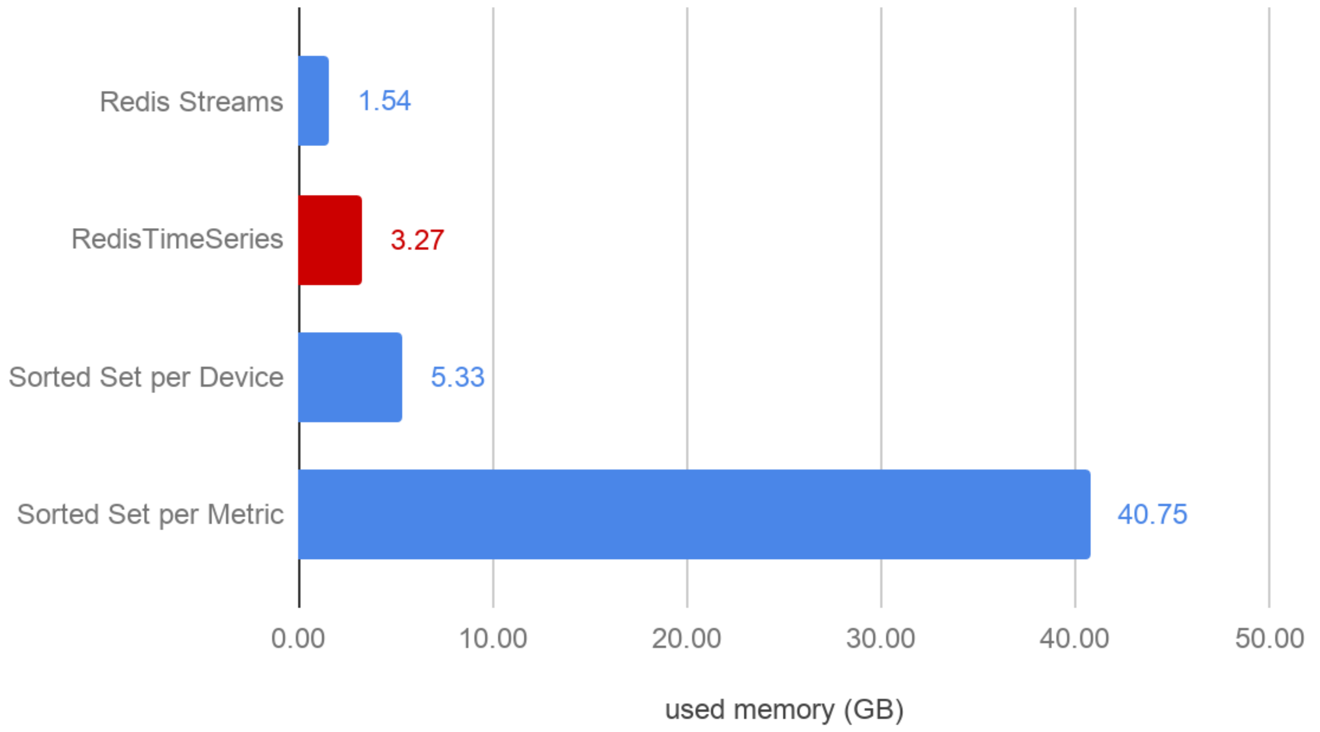


This is where you can see the real power of having dedicated data structure for a given use case with a toolbox that runs alongside it. RedisTimeSeries just blows all other approaches out of the water, and is the only one to achieve sub-millisecond response times.

Memory utilization

In both the Redis Streams and Sorted Set approaches, the samples were kept as a string, while in RedisTimeSeries it was a double. In this specific data set, we chose a CPU measurement with rounded integer values between 0-100, which thus consumes two bytes of memory as a string. In RedisTimeSeries, however, each metric had 64-bit precision.

RedisTimeSeries can be seen to dramatically reduce the memory consumption when compared against both Sorted Set approaches. Given the unbounded nature of time series data, this is typically a critical criteria to evaluate - the overall data set size that needs to be retained in memory. Redis Streams reduces the memory consumption further but would be equal or higher than RedisTimeSeries when more digits for a higher precision would be required.


Used Memory (GB)
Redis Streams
1.54
RedisTimeSeries
3.27
Sorted Set per Device
5.33
Sorted Set per Metric
40.75
0.00
10.00
20.00
30.00
40.00
50.00
used memory (GB)