redis

t-digest: A New Probabilistic Data Structure in Redis Stack

Lior Kogan and Pieter Cailliau
March 14, 2023

With the introduction of the latest Redis Stack, we celebrate a new probabilistic data structure: t-digest.

You can use t-digest to efficiently estimate percentiles (e.g., the 50th, 90th, and 99th percentile) when your input is a sequence of measurements (such as temperature or server latency). And that's just one example; we explain more of them below.

The support for t-digest extends the existing functionality of Redis Stack and its support for data models and processing engines, which includes RedisInsight, search and query, JSON, time series, and probabilistic data structures.

Here we explain what t-digest is when it's a good option, and how to use it.

Probabilistic data structures in Redis

Redis already has plenty of probabilistic data structures: HyperLogLog, Bloom filter, Cuckoo filter, Count-min sketch, and Top-k. You can use them to answer common questions concerning data streams and large datasets:

- How many unique values appeared in the data stream? (HyperLogLog)
- Did the value v appear in the data stream? (Bloom filter and Cuckoo filter)
- How many times did the value v appear in the data stream? (Count-min sketch)
- What are the *k* most frequent values in the data stream? (Top-k)

Getting accurate answers can require a huge amount of memory. However, you can reduce the memory requirements drastically if you are willing to sacrifice accuracy. Each of these data structures allows you to set a controllable tradeoff between accuracy and memory consumption. In addition to requiring a smaller memory footprint, probabilistic data structures are generally much faster than accurate algorithms.

As you can see, probabilistic data structures are really fun! (Well, at least for software developers.)

What is t-digest?

t-digest is a probabilistic data structure that can help you answer questions like:

- Which fraction of the values in the data stream are smaller than a given value?
- How many values in the data stream are smaller than a given value?
- Which value is smaller than *p* percent of the values in the data stream? What is the *p*-percentile value?
- What is the mean value between the p_1-percentile value and the p_2-percentile value?
- What is the value of the n^{th} smallest (or largest) value in the data stream? What is the value with [reverse] rank *n*?

Ted Dunning first introduced t-digest in 2013 and described it in several publications:

- Computing Extremely Accurate Quantiles Using t-Digests (2019)
- Conservation of the t-digest Scale Invariant (2019)
- The Size of a t-Digest (2019)
- The *t*-digest: Efficient estimates of distributions (2021)

Practically speaking, t-digest may help in several ways. Here are a few scenarios

Hardware/software monitoring

You are measuring your online server response latency and want to know:

- What are the 50th, 90th, and 99th percentiles of the measured latencies?
- What percentage of the measured latencies are under 5 milliseconds?
- What is the mean latency, ignoring outliers? Or, more accurately: What is the mean latency between the 10th and the 90th percentile?

Online gaming

Millions of people are playing a game on your online platform. You want to provide each player with the following information:

- Your score is better than *x* percent of the game sessions played

Network traffic monitoring

You are measuring the number of IP packets per second transferred over your network. You want to quickly detect potential denial-of-service attacks. You might want to query:

- Does the number of packets in the last second exceed 99.9% of previously observed values?
- How many packets do I expect to see under normal network conditions – say, between the 0.5 and the 99.5 percentile?

Predictive maintenance

You are measuring a machine's readings, such as noise level or current consumption. To detect suspicious behavior, you can query:

- Was the measured parameter irregular? That is, it was not within the [1st percentile ... 99th percentile] range? Or, to add your logic: Was the measured parameter irregular for at least 10 seconds during the last hour?
- To which values should I set my alerts?

Can't I do all this without t-digest?

Of course, you can. But t-digest makes it easier. Let's compare a before-and-after scenario.

Suppose you want to measure HTTP request latency on a website you run. The HTTP latency is the amount of time it takes from when a request is made by the user to the time it takes for the response to get back to that user).

Latency varies greatly depending on many factors, so it is common to measure the 50th, 90th, and 99th percentile of the latency. Trivially, half of the requests are served in less than the 50th percentile, 90% of the requests are served in less than the 90th percentile, and so on.

How would you determine these statistics without t-digest? The trivial way would be to store all measured latencies (which can be millions or billions per day) in a sorted array. To retrieve the 90th percentile, for example, you would read from the sorted array the value in an index equal to 90% of its size. More complex data structures and algorithms could be used, but usually under a given set of assumptions, such as the latency range and resolution, its distribution, and the set of constant percentiles that would be queried.

With t-digest, no such assumptions are needed. In addition, the memory footprint is small, and adding and querying data is very fast.

So what is the catch? Be ready to tolerate a very small (usually negligible) relative error in the estimations. In the vast majority of cases where statistics are concerned, a small error in estimators is acceptable.

How can I use t-digest?

Let's get practical and see how it works.

Let's continue the HTTP request latency example. One option is to create a t-digest with `TDIGEST.CREATE` and add observations – or measurements, if you prefer – with `TDIGEST.ADD`.

```
TDIGEST.CREATE key [COMPRESSION compression]
```

This initializes a new t-digest data structure (and emits an error if such a key already exists). The `COMPRESSION` argument specifies the tradeoff between accuracy and memory consumption. The default is 100. Higher values mean more accuracy.

To add a new floating-point value (observation) to the t-digest, use:

```
TDIGEST.ADD key value [value ...]
```

For example, to create a digest named t with a compression setting of 1000 (very accurate) and add 15 observations, we'd type in:

```
TDIGEST.CREATE t COMPRESSION 1000
TDIGEST.ADD t 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5
```

You can repeat calling `TDIGEST.ADD` whenever new observations are available. To estimate values by fractions or ranks, use `TDIGEST.QUANTILE`:

TDIGEST.QUANTILE key quantile [quantile ...]

It returns, for each input *fraction*, an estimation of the *value* (floating point) that is smaller than the given fraction of observations. In other words, fraction 0.9 is equivalent to the 90th percentile.

The following query retrieves the latency that is smaller than 0%, 10%, 20%, ..., 100% of the observed latencies:

```
TDIGEST.QUANTILE t 0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
1) "1"
2) "2"
3) "3"
4) "3"
5) "4"
6) "4"
7) "4"
8) "5"
9) "5"
10) "5"
11) "5"
```

The query contains 11 fractions; hence the reply is an array of 11 latency values.

You can see that 10% of the latencies are smaller or equal to 2; 20% and 30% of the latencies are smaller or equal to 3; 40%, 50%, and 60% of the latencies are smaller or equal to 4; and lastly, 70%, 80%, 90%, and 100% of the latencies are smaller or equal to 5.

You can also query for the *n* minimal measured latencies. Use `TDIGEST.BYRANK` for this.

TDIGEST.BYRANK key rank [rank ...]

It returns, for each input *rank*, an estimation of the *value* (floating point) with that rank.

The following query retrieves the first, second, ..., and ninth smallest observed latencies:

```
TDIGEST.BYRANK t 0 1 2 3 4 5 6 7 8 9
1) "1"
2) "2"
3) "2"
4) "3"
5) "3"
6) "3"
7) "4"
8) "4"
9) "4"
10) "4"
```

The query contains 10 ranks; hence the reply is an array of 10 values.

When the rank is 0, the result is the smallest observation (1 in this example). Similarly, when the rank is equal to the number of observations minus one, the result is the largest observation. For these two ranks, the results are always accurate; the results for any other rank are estimates. When the rank is equal to or larger than the number of observations, the result is inf.

You can see that the second and third smallest latencies (the observations with ranks 1 and 2, respectively) are 2. Similarly, the third, fourth, and fifth smallest latencies (the observations with rank 3, 4, and 5, respectively) are 3, and the sixth, seventh, eighth, and ninth smallest latencies (the observations with rank 6, 7, 8, and 9 respectively) are 4.

And, of course, you can query for the *n* largest measured latencies. Do so using `TDIGEST.BYREVRANK`.

TDIGEST.BYREVRANK key reverse_rank [reverse_rank ...]

It returns, for each input *reverse rank*, an estimation of the *value* (floating point) with that reverse rank.

The following query retrieves the first, second, ..., and ninth-largest observed latencies:

```
TDIGEST.BYREVRANK t 0 1 2 3 4 5 6 7 8 9
1) "5"
2) "5"
3) "5"
4) "5"
5) "5"
6) "4"
7) "4"
8) "4"
9) "4"
10) "3"
```

The query contains 10 reverse ranks; hence the reply is an array of 10 values.

When the reverse rank is 0, the result is the largest observation (5 in the example). Similarly, when the reverse rank is equal to the number of observations minus one, the result is the smallest observation. For these two reverse ranks, the results are always accurate. The results for any other reverse rank are estimates. When the reverse rank is equal to or larger than the number of observations – the result is -inf.

You can see that the first, second, third, fourth, and fifth largest latencies (the observations with reverse ranks 0, 1, 2, 3, and 4, respectively) are 5. Similarly, the sixth, seventh, eighth, and ninth-largest latencies (the observations with reverse ranks 5, 6, 7, and 8, respectively) are 4.

To estimate fractions by values, use TDIGEST.CDF:

TDIGEST.CDF key value [value ...]

It retrieves, for each input *value*, an estimation of the fraction of observations smaller than the given value and half the observations that are equal to the given value.

The following query retrieves the *fraction* of latencies that are *smaller* than 0, 1, 2, 3, 4, 5, and 6 milliseconds respectively:

```
TDIGEST.CDF t 0 1 2 3 4 5 6
1) "0"
2) "0.033333333333333333"
3) "0.13333333333333333"
4) "0.29999999999999999"
5) "0.53333333333333333"
6) "0.83333333333333337"
7) "1"
```

The query contains seven latency values; hence the reply is an array of seven fractions.

As you can see, all the estimations in this simple example are accurate: the fraction of latencies smaller than 0 plus half the latencies that are equal to 0 is 0. Similarly, the fraction of latencies smaller than 1 plus half the latencies that are equal to 1 is 3.33%, etc

The following query retrieves the number of latencies that are smaller than 0, 1, 2, 3, 4, 5, and 6 milliseconds respectively:

TDIGEST.RANK key value [value ...]

This is similar to TDIGEST.CDF, but it returns, for each input *value*, an estimation of the *number* of observations *smaller* than a given value added to half the observations equal to the given value.

```
TDIGEST.RANK t 0 1 2 3 4 5 6
1) "-1"
2) "1"
3) "2"
4) "5"
5) "8"
6) "13"
7) "15"
```

The query contains seven latency values; hence the reply is an array of seven ranks.

Again, all estimations in this example are accurate: there are no latencies smaller than 0; hence, the resulting rank is -1. The number of latencies smaller than 1 plus half the latencies that are equal to 1 is 1. Similarly, the number of latencies smaller than 2 plus half the latencies that are equal to 1 is 2, etc.

TDIGEST.REVRANK key value [value ...]

This is similar to TDIGEST.RANK, but returns, for each input *value*, an estimation of the *number* of (observations *larger* than a given value and half the observations equal to the given value).

The following query retrieves the number of latencies that are larger than 0, 1, 2, 3, 4, 5, and 6 milliseconds respectively:

```
TDIGEST.REVRANK t 0 1 2 3 4 5 6
1) "15"
2) "14"
3) "13"
4) "10"
5) "7"
6) "2"
7) "-1"
```

The query contains seven latency values; hence the reply is an array of seven reverse ranks.

Once again, you can see that all estimations in this example are accurate: the number of latencies larger than 0 plus half the latencies that are equal to 0 is 15. Similarly, the number of latencies larger than 1 plus half the latencies that are equal to 1 is 14. There are no latencies equal to or larger than 6; hence the resulting reverse rank is -1.

We can see that TDIGEST.RANK(v) + TDIGEST.REVRANK(v) for any v between the minimum and the maximum observation – is equal to the number of observations.

Calculating the average measurement value is a common operation. However, sometimes measurements are noisy or contain invalid values. For example, consider a noisy, invalid latency of 999999999 milliseconds. When this is possible, a common practice is to calculate the average value of all observations ignoring outliers. For example, you might want to calculate the average value between the 20th percentile and the 80th percentile.

To estimate the mean value between the specified fractions, use `TDIGEST.TRIMMED_MEAN`:

TDIGEST.TRIMMED_MEAN key lowFraction highFraction

```
TDIGEST.TRIMMED_MEAN t 0.2 0.8
"3.8888888888888888"
TDIGEST.TRIMMED_MEAN t 0.1 0.9
"3.7692307692307692"
TDIGEST.TRIMMED_MEAN t 0  1
"3.6666666666666665"
```

Sometimes, it is useful to merge t-digest data structures. For example, suppose we measure the latencies for three servers, each with its own t-digest, but then we want to calculate the 90%, 95%, and 99% latencies for all the servers combined.

Use this command to merge multiple t-digest data structures into a single one:

TDIGEST.MERGE destKey numKeys sourceKey... [COMPRESSION compression] [OVERRIDE]

```
TDIGEST.CREATE s1
TDIGEST.ADD s1 1 2 3 4 5
TDIGEST.CREATE s2
TDIGEST.ADD s2 6 7 8 9 10
TDIGEST.MERGE sM 2 s1 s2
TDIGEST.BYRANK sM 0 1 2 3 4 5 6 7 8 9
1) "1"
2) "2"
3) "3"
4) "4"
5) "5"
6) "6"
7) "7"
8) "8"
9) "9"
```

Use `TDIGEST.MIN` and `TDIGEST.MAX` to retrieve the minimal and maximal values in the t-digest data structure, respectively.

Both return `nan` when the data structure is empty.

```
TDIGEST.MIN t
"1"
TDIGEST.MAX t
"5"
```

Both commands return accurate results and are equivalent to `TDIGEST.BYRANK key 0` and `TDIGEST.BYREVRANK key 0` respectively.

Use `TDIGEST.INFO` to retrieve additional information about the t-digest, including the number of observations added to the data structure and the number of bytes allocated for the data structure.

To empty a t-digest data structure and re-initialize it:

```
TDIGEST.RESET key
```

Next Steps

t-digest extends Redis's growing set of probabilistic data structures, which helps you address more use cases related to streaming data and huge datasets. It does so with sub-millisecond latency, sub-linear memory requirements, and in an extremely accurate fashion.

This blog post is a general overview. All t-digest commands are explained on redis.io.

Download the latest version from our download center or install it via FlatHub or Snapcraf.

Related Posts

Hello, Redis Stack

Learn More

Count-Min Sketch: The Art and Science of Estimating Stuff

Learn More

Probabilistic Data Structures in Redis

Learn More