WHITE PAPER

Redis HyperLogLog: Visualization and practical use with Node.js, Redis and Angular

Kyle J. Davis

CONTENTS

Executive Summary

One of the lesser known and understood structures in Redis is HyperLogLog. HyperLogLog is a powerful algorithm that can be used to estimate a count of unique items in an extremely space-efficient manner. This white paper provides an overview of the Redis HyperLogLog implementation, how it contrasts with similar structures and a practical example use of HyperLogLog.

Introduction

In April 2014, Salvatore Sanfilippo, the author of Redis, introduced the HyperLogLog (HLL) functions to Redis. Unlike the other structures in Redis (Strings, Sets, Sorted Sets, Hashes, and Lists), HyperLogLog isn't a direct storage routine, but rather an interface to an algorithmic structure. In addition, HyperLogLog is a relative newcomer, being first described in 2007 by Philippe Flajolet. Due to this recency and consequential unfamiliarity by many developers, the extremely useful Redis commands (PFADD, PFCOUNT, and PFMERGE) are undeservingly obscure.

What Does HyperLogLog Do?

Simply put, HyperLogLog can tell you an estimate of the number of unique items it's been supplied.

What's the big deal? You can do the same thing with a set and SCARD and the results will be 100% accurate. The difference lies in space efficiency.

Redis memory representation can be a complex topic, but with a set and SCARD, the size of the set is roughly proportional to the number of items in the set multiplied by the size of the items, a linear relationship. If you store 100 items in a set that are 100 bytes each, the set will be roughly 10kb.

HyperLogLog, on the other hand, uses hashing and the random distribution of bits to mathematically estimate the number of unique items that you've put into it. How it works is fascinating but unimportant for use as Redis abstracts the inner workings of the HyperLogLog algorithm. What you do need to know is:

- HyperLogLog is space efficient—the maximum size of the data structure is about 12kb.
- HyperLogLog is pretty quick.
- HyperLogLog has a low error rate—0.81%.
- HyperLogLog is stored, internally, as a normal string in Redis. You can call string commands on the HyperLogLog (GET, SET, STRLEN) to evaluate and manipulate the underlying data (although this is not very useful in real-terms).

All of this seems abstract until you start using it. To contrast the difference between HyperLogLog and set/SCARD, we will build a HyperLogLog Playground

Building the playground

The playground is built with Node.js and Redis with Angular.js (1.x) handling the interactivity. Let's first start with the server—the API needs to be able to:

- Simultaneously add a single item to both a set and to a HyperLogLog.
- Get the current size of both the set and the HyperLogLog.
- Simultaneously reset both the HyperLogLog and the set.

For simplicity sake, all the playground API commands always return the sizes of both the set and the HyperLogLog. Here is how the routes will look:

- **POST /item** Accept a string of any size in a JSON object (at "value"). Return the PFCOUNT and SCARD as well as the STRLEN of the HyperLogLog and a running total of number of bytes passed into the set.

- **GET /items** Return the PFCOUNT and SCARD as well as the STRLEN of the HyperLogLog and a running total of number of bytes passed into the set.
- **DELETE /items** Reset the HyperLogLog and set (DEL). Return the PFCOUNT and SCARD as well as the STRLEN of the HyperLogLog and a running total of number of bytes passed into the set.

As you might notice, since it's always returning the same information, we can use the same code—basically if Redis returns anything but an error it runs the same function after every request (storageInfo).

To serve the static assets (CSS/HTML/Javascript), the server also uses the built-in Express static middleware.

A couple of caveats:

- The running total of bytes in the set is managed on the app level—so if you SADD extra items through redis-cli, those bytes won't be counted.
- The storageInfo function is not atomic for code simplicity.

The client side part of the playground is a short Angular.js single page application. Angular uses a template that is HTML with a few extra attributes that are interpreted by the client-side Javascript. The client-side part of the app consists of two pages:

- **index.html** - The HTML file that calls all the Angular assets and contains our only view.
- **hllplayground.js** - The javascript file that contains our angular bootstrap code and our controller.

Angular.js is being called from a CDN as is chart.js, a Javascript charting library. Chart.js is being Angular-fied by the angular-chart module.

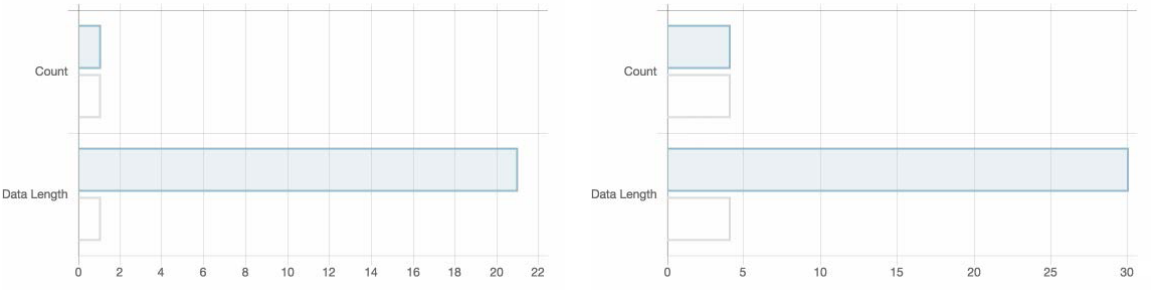
In the controller, $http is used to make run-of-the-mill HTTP requests to the server.

Using the Playground

When you start the playground you'll see two graphs. On the left is the history and the right is the current state of the set and HyperLogLog. The history graph will be empty as it is managed for just the browser session.

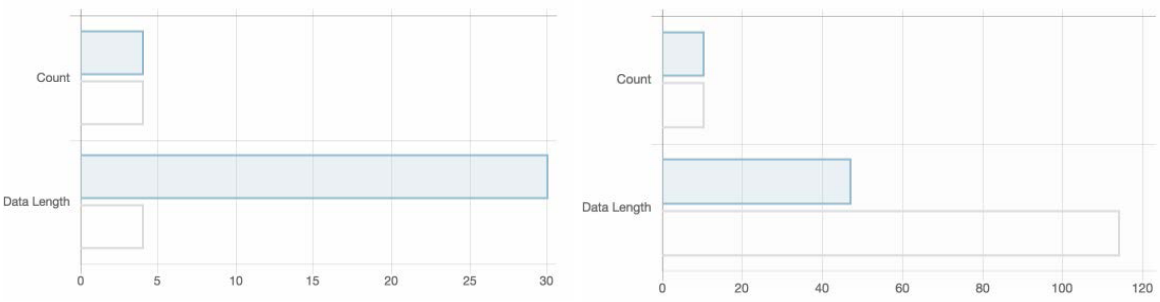
Under the graphs are a tabular representation of the data. Below the table is a form where you can enter any value you'd like to store both in the set and in the HyperLogLog as well as a few buttons. Finally below the form is a history for the session.

To start out, just start entering strings into the text field and click "submit." After submitting you'll see the graphs instantly change. On an empty HLL and set, add a single character. You'll see the size of the HLL jump—this is due to the overhead of the structure as well as the bits being stored. The set (represented by the lighter bar), at this point will be smaller, but the count will be the same as estimation errors are unlikely with only one item in a HyperLogLog.



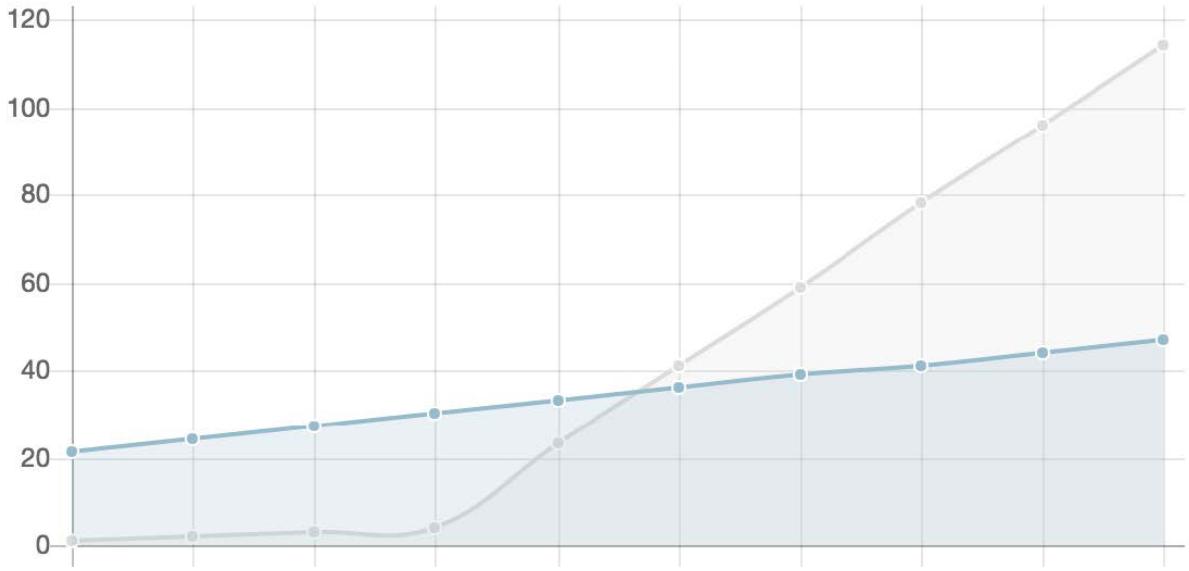
Left: with one single character entry. Right: with four single character entries. The dark bar is a HyperLogLog, the light bar is a Set.

Now, let's click the "Random" button a few times. This is taking a javascript random number (Math.random) and submitting it to the server.



Left: Before clicking "Random." Right: after clicking "Random" six times. The dark bar is a HyperLogLog, the light bar is a Set.

After clicking the random button six time (which inserted six, ~18-bytes strings) the set has really grown and surpassed the storage taken up by the HyperLogLog data and overhead. Looking at the other graph you can see the linear nature of the set/SCARD method overtaking the HLL.



After four single byte items and six 18-byte items. The dark area is the Hyperloglog size while the light area is the size taken up by the set.

While the HyperLogLog is growing, it isn't totally relational to the size of the items being inserted, and, in fact, will eventually cap out at around 12kb. Redis automatically run-length encodes HyperLogLogs so they are smaller when few items have been added.

Recall earlier that HyperLogLogs can only estimate the number of unique items, so far we haven't seen anything—the counts are still the same. On the playground table, the HyperLogLog count cell text turn red when the count is not the same as the SCARD count. You can have many items before you see an estimation error.

Method	Count	Data Length
HyperLogLog	75	219
Set	76	1738

HyperLogLog missing a count.

You can go and check out the playground source on github. You'll need to grab the repo and run npm install. Put your connection details into a node_redis connection options object and save it as a JSON file (outside of the project's directory!). Launch the server by running:

```
$ node index.js connection pathtoyourconnection.json
```

You should shortly see "server ready" in your terminal. To start exploring the HyperLogLog playground point your browser at http://localhost:8012/

Practical Application of HyperLogLog

Now that we've established the function of the Redis HyperLogLog API with the HyperLogLog Playground, let's build a more useful application using the HyperLogLog algorithm. In applications like Swarm or Facebook Places users can "check-in" to a location and you can see how many unique users have checked into that location.

If you built this app using only Redis sets, with each user, the app records the places they've checked into in Redis using a set. It could look something like this:

```
userlocations:[userid] [locationid]
```

Adding a visit to a user would be a single command:

```
> SADD userlocations:[userid] [currentlocationid]
```

It's quick and easy to see if a user has been to a location before:

```
> SISMEMBER userlocations:[userid] [currentlocationid]
```

And if you want to know what location's they've been to, you can issue a SMEMBERS command:

```
> SMEMBERS userlocations:[userid]
```

This is all pretty manageable—each user is going to have a relatively small number of locations they've visited. Top users may have thousands of visited locations, but most will have a few dozen.

Now, on the other side, how do you deal with the users that have visited a location? Let's assume we use a similar setup for the locations as we do for the users—a Redis set:

```
location:[alocationid] [ [auserid], [anotheruserid] … ]
```

We could then find the number of unique users by doing a SCARD command

```
> SCARD location:[alocationid]
```

You could check to see if you've checked into the location by running SISMEMBER:

```
> SISMEMBER location:alocationId myuserid
```

For many locations, the number of unique visitors will be small—the local pizza restaurant might rack up a thousand unique visitors, for example. But what happens if you have a popular location?

At time of writing Walt Disney World on Facebook Places has over 5.4 million unique visitors that have checked in. If we store all the visitors in a set for this location, we could be in real trouble.

Let's do the math to see how much memory Walt Disney World's set would take up. To get a good number, let's flesh out our data. Just for an example, let's say we're storing our user IDs as 32-character hexadecimal strings. We'll also make the assumption that each character takes up a byte of memory when stored in a set (as mentioned earlier Redis memory representation is not that cut-and-dry, but stay with me).

```
5,400,000 * 32 = 172,800,000 (~173 mb)
```

That's a lot of in-memory storage for a single location. What happens if you have thousands of locations? Millions? Not to mention that as your service grows, you're going to increase the storage as:

1. Your user base grows
2. Number of locations grows
3. Number of check-ins increase Scaling could easily become unmanageable.

As you might have guessed, we can ease this problem with HyperLogLog. Taking stock of the situation when it comes to the data about checking in, you really only need to know the following things:

- Have I checked into this location before?
- How many unique users have checked into this location previously?

The first question can be answered with just the user's own data. The second question we can answer with PFCOUNT. It does come with caveats—first, you won't be able to answer "Who has been to this location?" but you'll likely want to store this information somewhere (maybe not in Redis, given the size of the set at scale), but it's certainly not needed to render the page. Page rendering is the critical, high-volume bottleneck.

Secondly, you have to be aware that the answer that PFCOUNT supplies is only an estimate. It's going to be close, but it may not be right on the money. In an app like this, an error rate of 0.81% is likely more than acceptable at large scales.

Despite the above mentioned caveats we're gaining some huge benefits. The maximum size that the HyperLogLog can take up is 12kb per location. All of your check-in information will grow at a nice, linear rate to the number of locations. If you have 1,000 locations, the sum of your location data will be ~12mb. 10,000? ~120mb. It doesn't matter if every one of those 10,000 locations are a Walt Disney World-scale number of check-ins, it'll be no more than ~120mb for 10,000 locations.

Now, let's say that you have a few locations—Walt Disney World, Epcot Center, Magic Kingdom, Hollywood Studios, and Animal Kingdom. In this example, the latter four all make up Walt Disney World. If you wanted to find the count of all of these locations. With sets, this could become a massive computational undertaking to use SUNION. PFCOUNT allows you to supply multiple arguments to get an on-the-fly unique count across multiple HyperLogLogs. The performance isn't as good as a single key, but it's better than SUNION, especially at scale. In the same vein if you wanted to remove the comprising parks and merge everything into one park, you can call PFMERGE to combine all the locations together without sacrificing or double counting anyone. Handy!

Check-dis Example

Now, let's take that idea and make it into the start of a web app. We're going to build it using the same basic stack as the HyperLogLog playground: Node.js, Express, Angular.js and Redis (the NEAR stack). We're not going to get into the weeds of authentication, so we can simulate part of that by having each session login. As long as you don't reload the page it will hold on to your "user."

The basic functions here are:

- List the locations
- Check-in to a location
- View the count of a single location

Just for fun, let's add in some other features that will help illustrate what you can do with HyperLogLog.

- Send random "check-ins" with fake email addresses in bulk
- See the combined total of several locations

On the server-side, I'll try and keep it as minimal as possible.

- **PUT /location/[location-key]** check-in to a location, with the user's email in the body of the request, return the current number of checkins
- **GET /location/[location-key]** return the current number of checkins
- **GET /locations** return a list of all the locations available
- **GET /multi-location/[location-key+location-key2+...]** get the total of the locations listed in the parameter.

The "user"'s email is hashed into MD5—this will be our emulation of a unique user ID field. We're also only allowing users to check-in to certain locations by checking to see if a location is a member of a set. You'll need to add some locations (thematically, I'm using Disney parks):

```
> SADD locations [mynewlocationidentifier]
```

Grab the script from github and run:

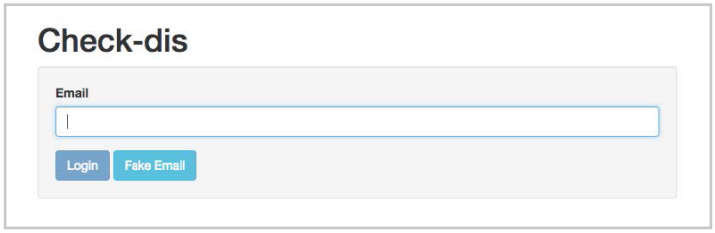
```
$ npm install
```

I'm using the middleware pattern pretty extensively in this script. If you're not familiar, it's basically an intermediary step between a route and sending a HTTP response—this step can modify or add to data about the request or, alternately, it can reject a route and send an error. This pattern is especially useful in Redis as it's friendly to asynchronous calls and Redis is so fast that you can easily do even a hand full of round-trips without getting outside of the acceptable range for response time.

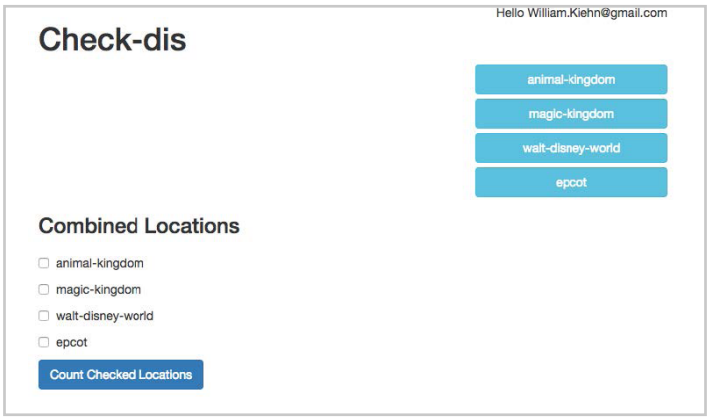
To get the script started, run:

```
$ node index.js credentials ../yourcredentialsfile.json
```

The **your-credentials-file.json** is a node_redis connection object stored as a JSON file. Make sure you store this file in a safe place (outside of your project directory). Once running, you'll see the "app started" message in your terminal. You can point your browser at http://localhost:8013/. You should see something like this:



If you click the “Fake Email” button you can generate fake but valid emails. We’re using the faker library to do this magic. Put your own email or generate a fake one and click “Login.” You should see the dashboard:



Click on one of the locations on the right hand part of the screen—this brings up where you can check-in to that location.

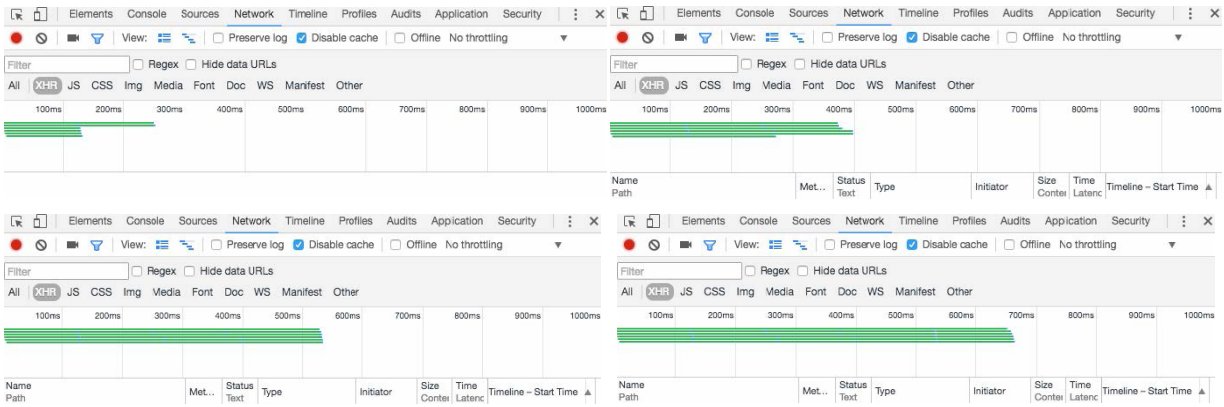
Click the “Check-in” button—notice that the check-ins will increment. You’ll also see a status below that indicates you’ve checked-in. If you click it multiple times, you still get the notice that you’ve checked in but the number won’t increment (as expected).

Now, you might have noticed that the number of check-ins is 6515 in the above screenshot. This isn’t the result of spending hours checking-in random users. One can use the random check-in buttons—these send a random email addresses to the server in bulk. They are sent as parallel requests.

If you click on “Random Check-ins (100)” you might notice these check-ins were sent all at once (100 HTTP Requests) but they are coming back one at a time. This actually is illustrating a few different things:

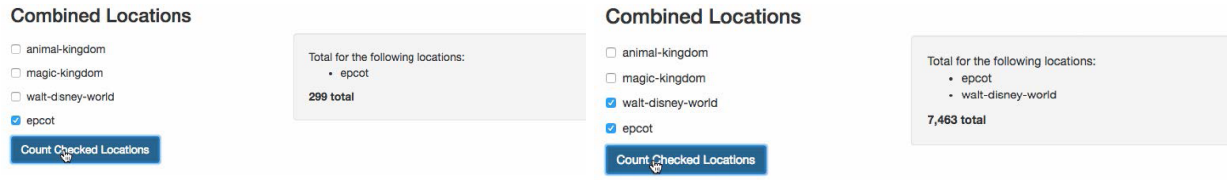
- Browsers only have a limited number of ports available
- Node is single thread
- Redis is single thread.

You can see how the requests go out and responses flow in with Chrome's DevTools:



Clicking the "Random Check-ins (100)" 5 or 6 times in a row and watch how the data flows in neatly. It's an interesting way to visually see how load occurs and the server deals with the strain (although maybe not the most scientific).

You can also combine location totals to get a combined unique count over multiple HyperLogLogs



It's even pretty quick though PFCOUNT with multiple keys is considered a slow HyperLogLog operation.

Conclusion

In our first section, the HyperLogLog playground, we explored the differences and similarities between sets and HyperLogLog. This script can be helpful in visualising the concepts of HyperLogLog and modifying the script can help you understand if your use-case would work for HyperLogLog.

The second section, the Check-dis app, is an example of a more direct practical use-case for the HyperLogLog functions. While the example code is far from a complete application, the underlying concepts illustrate how HyperLogLog can more efficiently scale this type of application.

The HyperLogLog commands unlock a very powerful algorithm available in Redis. These commands can save both on computational and space resources when compared to using sets.

700 E El Camino Real, Suite 250
Mountain View, CA 94040
(415) 930-9666
redis.com