redislabs
home of redis

whitepaper

Redis for Geospatial Data

Author: Itamar Haber, Chief Developer Advocate, Redis Labs

Table of Contents

Executive Summary

Geospatial data represents a complex real time big data challenge. Redis' new geospatial indexes introduce an extremely efficient and simple way of using geospatial data in new applications. Redis' versatile nature makes it an exceptionally proficient database for combining geospatial data with other types of data to solve complex application scenarios. Redis' flexibility with Lua scripting and the new Redis Modules, allow you to extend it even further to an infinite variety of application scenarios. This whitepaper provides an overview of how geospatial indexes are implemented in Redis and how they can be used in application scenarios.

Introduction

What is Geospatial Data

The world is irreversibly more connected than ever. As technology starts to extend itself to physical scenarios rather than cyber ones, location, distances and the related data processing become increasingly important. The term geospatial, or just spatial, refers to data which has a geographical or spatial aspect. Handling this data is complex because of the nature of location data.

A location is primarily identified by a pair of coordinates, x and y (longitude and latitude), in the 2-dimensional space. Depending on the data, a 3rd coordinate can be used to denote elevation (or altitude when measured from sea level). Lastly, data can be extended to the 4th dimension for performing time series geospatial analysis.

Why Bother with Geospatial Data

As mobile technology continues to connect the world, and as the Internet of smart "things" gains greater power and intelligence, location plays an important role in several applications. A mobile service might want to localize the advertisements it shows its users based on their location. A more complicated scenario is a navigation service displaying local services as the user passes through locations, in time for them to make appropriate choices and then selecting the right route to guide them. An even more sophisticated example is where this same navigation service also displays the favorite locations or entertainment options of the user's closest friends from when they visited.

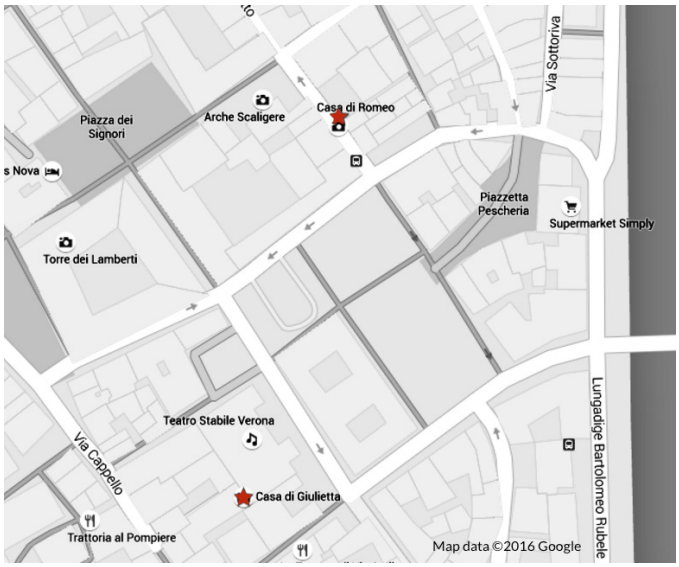
More industrial scenarios could involve managing and routing fleets optimally based on changing weather conditions. When you have hundreds or thousands of connected "things" like sensors or meters, using the location information to gain local intelligence and take immediate action like auto-replacement of defective parts or shutdown during danger levels, becomes important.

For our whitepaper here, we will use the not very technically complicated example of location-based social networks.

Romeo and Juliet are using the latest ShakeDatApp (Shakespeare's dating application) which offers instant notifications of personalized potential for love and drama in your immediate vicinity (100 meters), with just a simple shake of your mobile device.

Juliet, of the House of Capulet, loves ice cream and long walks in the rain while living at Via Cappello 23 Verona, Italy - 10.9962165,45.4419226.

Romeo, with similar interests in desserts and rain, kicks back at the House of Montague at Via Arche Scaligere 4 Verona, Italy - 10.9971645,45.4435245.



What Makes Geospatial Special

The advent of location-aware technology make processing geospatial data a real-time, big data challenge. Applications that use and manage geospatial data effectively serve, at minimal latency, a high number of requests for:

- Fetching the location (or "Wherefore art thou?")
- Location updates ("I am here")
- Searching data by location ("Who is near this location?")

Simple reads (fetch location) and writes (update location) are challenging at scale. Searching compounds the challenge. The key to satisfy the above requirements is maintaining effective indexes for the data. An effective index is one that can facilitate speedy searches and isn't expensive to maintain (in terms of memory and compute power).

Why Redis is Perfect for Geospatial Data

Redis is a data structure store, often used as a database, a cache and a message broker. Data structures are like "Lego" building blocks; helping developers achieve specific functionality with the least amount of complexity, least overhead on the network and the least latency because operations are executed extremely efficiently in memory, right next to where the data is stored. Redis includes data structures such as Hashes, Sets, Sorted sets, Lists, Strings, Bitmaps and HyperLogLogs. These are highly optimized and each provides specialized commands that help you execute complex functionality effortlessly and with very little code. These data structures make Redis extremely powerful and allow Redis-based applications to handle extreme volumes of operations at very low latency.

While other disk-based databases do have built-in capability to handle geospatial data, to handle high volume, frequent real-time updates(like when in transit) at extremely low latencies would require several tens or hundreds of server instances. Redis, runs in memory, and can handle extremely high throughput (millions of operations/second) of updates and reads, with relatively little hardware. See https://redislabs.com/benchmarks for more details.

With the introduction of geospatial sets and operations, Redis now has the added advantage that location-specific indexing, searching and sorting can all take place with extreme simplicity, right next to where the data is stored, in-memory. This enables reduced code complexity, reduced network bandwidth consumption and overall faster execution.

When combined with other Redis capabilities, some functionality becomes almost magically simple to implement. For example: by melding the new Geo Sets and PubSub together, it is near trivial to set up a real time tracking system in which every update to a member's position is sent to all interested parties.

If your application relies on or uses geospatial data, Redis provides the simplest, fastest way to store, process and analyze it.

Redis Geospatial Indices

The Geo Set

The Geo Set is the basis for working with geospatial data in Redis—it is a data structure that is specialized for managing geospatial indices. Each Geo Set is made up of one or more members, with each member consisting of a unique identifier and a longitude/latitude pair. Similar to all of Redis' data structures, Geo Sets are manipulated and queried using a subset of simple-to-use and at the same time highly-optimal commands.

Internally, Geo Sets are implemented with another Redis data structure called a Sorted Set. Sorted Sets exhibit a good space-time balance by consuming a linear amount of RAM while providing logarithmic computing complexity for most operations.

Creating and Adding to the Index

The Redis command for adding members to a geospatial index is called `GEOADD`. This command is used both for creating new sets as well as adding members. The following example demonstrates its use:

Redis Command Example

```
GEOADD locations 10.9971645 45.4435245 Romeo
```

Node Redis Example

```
redis.geoadd('locations', '10.9971645', '45.4435245', 'Romeo');
```

The above tells Redis to use a Geo Set called "locations" for storing the coordinates of the member named "Romeo". In case that the "locations" data structure doesn't exist, it will first be created by Redis. The new member will be added to the index if and only if it does not exist in the set.

It is also possible to add multiple members to the index with a single call to `GEOADD`. By batching multiple operations in a single command, this form of invocation reduces the load on the database and the network - an example:

Redis Command Example

```
GEOADD locations 10.9971645 45.4435245 Mercutio 10.9962165 45.4419226
Juliet
```

Node Redis Example

```
redis.geoadd('locations', '10.9971645', '45.4435245', 'Mercutio',
'10.9962165','45.4419226', 'Juliet');
```

Updating the Index

After a member and its coordinates have been recorded in the index, Redis allows to update that member's location. Updating members in a Geo Set is done by calling the same command used for adding them, namely `GEOADD`. When called with existing members, `GEOADD` simply updates the spatial data that is associated with each member with the new values. Therefore, once Romeo exits the house to begin his evening stroll, his updated location can be recorded with the following:

Redis Command Example

```
GEOADD locations 10.999216 45.4432923 Romeo
```

Node Redis Example

```
redis.geoadd('locations', '10.999216', '45.4432923', 'Romeo');
```

Removing Members from the Index

After having been added to the index, members may need to be deleted from it at a later time. To facilitate the deletion of members from the Geo Set, Redis provides the `ZREM` command. To delete a member (or more) from the set, `ZREM` is called with the appropriate key name followed by the members to be deleted from it. For example:

Redis Command Example

```
ZREM locations Mercutio
```

Node Redis Example

```
redis.zrem('locations', 'Mercurio');
```

Deleting the Index

The geospatial index may be deleted entirely. Since the index is stored as a Redis key, Redis' `DEL` command can be used to delete it.

Reading from the Index

The data in a Geo Set index can be read in several ways. First, the index can be used for scanning through all of the members in it, whether in one big bulk or in several smaller chunks. Redis provides two commands that can be used for iterating through the entire index - `ZRANGE` and `ZSCAN`. However, because these can be used to cover all of the indexed elements, this type of access to the data is mostly reserved for offline, non-critical operations (for example ETL and reporting processes).

The second type of read access to the index is for fetching members' coordinates, and to achieve that Redis provides two commands. The first of these commands is `GEOPOS`, which returns the coordinates for a given member in a Geo Set. Assuming that Romeo is keeping with his walk, the answer regarding his current whereabouts can is provided by executing the following:

Redis Command Example

```
GEOPOS locations Romeo

1)    1) 10.999164
      2) 45.442681
```

Node Redis Example

```
redis.geopos('locations', 'Romeo', function(err, reply) {
});
```

In the example above, the first of the lines is the query, whereas the following lines are the database's response. Redis provides another command called `GEOHASH` that reports members locations. While both practically perform the same function, the difference between them is that `GEOHASH's` output is encoded as a standard Geohash (more on Geohashes below).

Another use for data that's stored in the index is computing distances between members. For any two members in the Geo Set, Redis' `GEODIST` command will compute and return the distance between them.

Searching the Index

The last, and perhaps most useful, type of read access that the geospatial index enables is searching the data by its location. The most common of such searches is for finding indexed members within a certain distance of a given location. For that purpose, Redis provides the `GEORADIUS` command.

As the name suggests, `GEORADIUS` performs a search within a circle given by its center's location and its radius and returns the members that fall inside it. Another Redis command, `GEORADIUSBYMEMBER`, serves the same purpose but accepts one of the indexed members as the circle's center. The following is an example of the search executed by ShakeDatApp when

Romeo reaches into his pocket and shakes his mobile has he passes the entrance to Teatro Stabile:

Redis Command Example

```
GEORADIUSBYMEMBER locations Romeo 100 m
1) “Juliet”
```

Node Redis Example

```
redis.georadiusbymember(‘locations’, ‘Romeo’, ‘100’, ‘m’,
function(err, reply) {
});
```

The search command also supports sorting the replies from nearest to furthest (the default) or vice versa, as well as returning the location and distance of each reply. Redis also allows storing the reply in another Set for further processing (such as paging and Set operations).

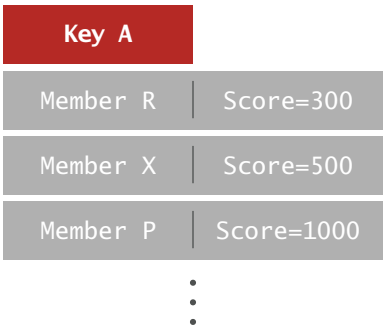
Technical details

This section describes the implementation and technical details of Redis’ Geo Sets. Internally, Geo Sets are implemented using another Redis data structure called a Sorted Set.

Sorted Sets

A Redis Sorted Set is a data structure that stores a score for each member in it, allowing for ordering and searching for members by their rank and score. The Set’s members are Redis Strings[1] and scores are 64-bit floating point numbers[2].

The following is an illustration of the Sorted Set data structure:



The Sorted Set data structure exhibits O(N) asymptotic space complexity, which means its memory consumption is linearly proportional to the number (and sizes) of the members in it. The computational complexity of most Sorted Set operations is logarithmic, or O(logN). Logarithmic complexity means that the processing cost of operations such as adding, removing and searching for members is kept relatively low, even for extremely large sets[3].

[1] String sizes can be up to 512MB and are binary-safe
[2] Redis uses the IEEE 754 floating point number representation allowing for the precise inclusive integer range between -2^{53} to 2^{53}
[3] The upper limit on the number of members in a Redis Set is 4,294,967,296 (or 2^{32})

Geo Sets

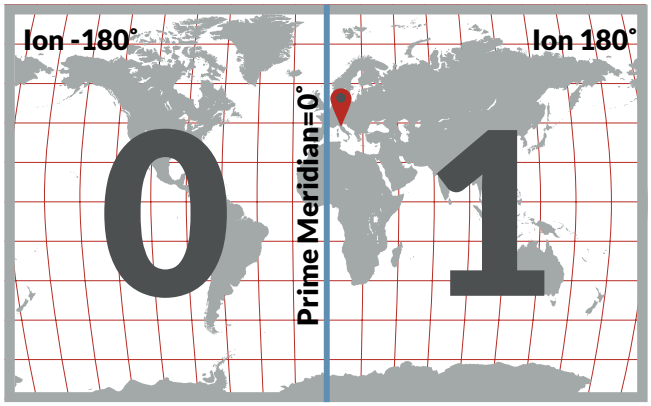
A rose by any other name would smell as sweet, and Geo Sets are just Sorted Sets by a different name. The difference, however, also extends to how the score is used—while with regular Sorted Sets the score can be set to any arbitrary numerical aspect of the data, with Geo Sets it is used for storing the location.

Locations are described by a pair of coordinates, the longitude and the latitude, whereas the score is a single number. The main functionality provided by Geo Sets on top of regular Sorted Sets is performing on-the-fly encoding and decoding of coordinate pairs and numerical scores. Redis implements the Geohash system for translating between these two representations.

Geohash

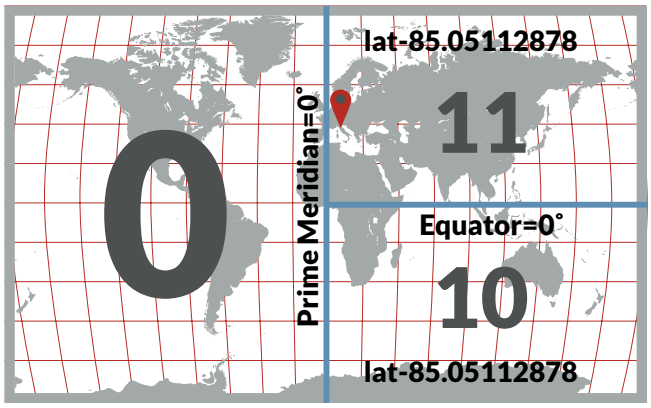
Geohash is a latitude/longitude geocode system invented by Gustavo Niemeyer when writing the web service at geohash.org, and put into the public domain. Redis uses this system for mapping between pairs of valid[4] coordinates and their hash values. While the original Geohash system uses base-32 representation of the hash value (for humane reasons), Redis simply stores the hash's numerical representation as the member's score in the Geo Set.

The way that the Geohash system works is by dividing the world to rectangular cells where each such cell is uniquely identified by its hash value. The cells' hashes are computed by interleaving the information from both location coordinates into a single value. Consider Juliet and the world:

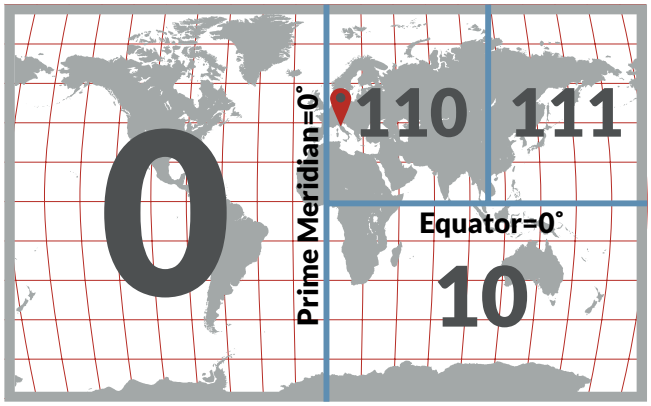


Geohash first tests the longitude to see whether it is on the Prime Meridian's left or right. Because Juliet is the Sun in the East, the hash's most significant bit is turned on to 1. Next, the same logic is applied to Juliet's latitude:

[4] As specified by EPSG:900913 / EPSG:3785 / OSGEO:41001, valid longitudes are between -180 and 180 degrees and latitudes are between -85.05112878 and 85.05112878 degrees



Verona lies north to the Equator, so the Geohash's next bit is also turned on to yield the binary value of 11. The Geohashing process continues by dividing that hash's cell according to the longitude again, like so:



By repeating this, the resulting binary value can represent any location with increasing degree of accuracy. The accuracy error only depends on the number of iterations that are performed, that is the hash's length.

As noted earlier, Redis' scores can accurately express integers that are up to 53-bit long. This allows for 26 Geohash iterations that produce hash values that are 52-bit long. A 52-bit Geohash provides an accuracy error of 0.6 m, which is more than adequate for anything short of military-grade weaponry.

Geohash search

Geohashes are more than just a clever encoding method - their structure makes it possible to search efficiently. As shown above, the hash's length determines its accuracy. Put differently, the shorter that the hash is, the bigger the area that it covers. That means that searching for locations within a given cell is simply a matter of retrieving all the locations that share that cell's prefix. Consider, for example Juliet's binary Geohash value:

*11010000001110101011011000000111111000011110000011*11

The italicized part is that hash's longest possible prefix. Permuting the last two (least significant) bits provides that cell's 3 immediate neighbours, namely:

*11010000001110101011011000000111111000011110000011*00

*11010000001110101011011000000111111000011110000011*01

*11010000001110101011011000000111111000011110000011*10

Searching for all locations that have that prefix is therefore a range search for all scores between `1101..1100` and `1101..1111`. By using a shorter Geohash search prefix the search is done in a bigger cell.

The basic search described above is used in implementing the more robust points-in-radius search. Given the circle's center and radius, Redis performs a Geohash search of the cell containing the center and the 8 cells that surround it (total of 9 cells). The distance from the center to each member found is checked, and if it is less or equal to the radius then it is included in the reply.

Advanced Redis Geospatial Capabilities with Lua

Lua is a powerful, fast, lightweight, embeddable scripting language. One of Redis' most powerful features is its ability to execute Lua scripts. Such scripts are executed atomically, by the data store's engine itself, and as near as possible to the data itself. Scripts are mainly used for reducing the network communications between the database and its clients, but they are also extremely useful for extending the built-in Redis functionality with custom logic.

The following section describes several such extensions made specifically for geospatial indices. All of the following extensions are a part of an open source Lua library called geo.lua. The library that can be found at: https://github.com/RedisLabs/geo.lua.

Elevation: The 3rd Geospatial Dimension

The Geohash, and consequently the Geo Set, indexes only two of the spatial dimensions. Some applications, such as air traffic control's, require use of the third coordinate that describes a location's elevation.

For managing 3-dimensional geospatial data, a combination of two Redis data structures is needed: a Geo Set and a Sorted Set. The Geo Set is used as described in this document, that is for storing pairs of longitudes and latitudes, whereas the additional Sorted Set stores only the elevation as each member's score.

Redis' transactions can be used for ensuring the consistency between the two sets when modifying their contents. Searching is done independently on each set and then the two result sets are intersected by Redis to provide the relevant responses. For additional information refer to geo.lua's xyzsets submodule.

Path Length

A common use for geospatial data is for computing the length a of a path that is defined as a list of locations. The algorithm for computing the length of path is straightforward:

1. Begin with length equals 0 at the first location in the path
2. Add the distance between the current location and the next one to the length
3. Move to the next location
   If the current location isn't the last one, go to step 2

Redis' `GEODIST` command accepts two arguments only, which makes it suitable for use in step 2. However, running the path length algorithm from a client that is connected to the database remotely would be suboptimal due to the need for sending each command to the server (as well as returning its reply).

In this case, the client requires an aggregate and not the raw data itself, which makes it a good candidate for implementation using Lua as exemplified by `GEOPATHLEN` command in the geo.lua library.

Polygon Search

While bounding-circle searches are the most common of geospatial searches, the next common type of filters uses polygons instead. A polygon is defined as any path that consists of four or more locations and whose first and last locations are identical. Polygons are commonly for modeling any spatial elements that are not points or lines, including geographical elements, man-made structures and national borders.

Similarly to computing a path's length, there are algorithms to perform searches for points inside polygons. As Redis doesn't have this ability built-in, such algorithms' implementation can be made either in the application's code or with a Lua script. Like with the `GEOPATHLEN` command, a Lua script is the preferable implementation in this case due to performance considerations.

The library's `GEOMETRYFILTER` script performs a polygon search with the following algorithm:

1. Compute the search polygon's axis-aligned bounding box.
2. Perform a `GEORADIUS` search from the box's center, using half of the box's diagonal as radius
3. Filter the points that are outside the bounding box
4. Filter points that are outside the polygon using the PNPOLY algorithm

Refer to the library's section about 2D geometries and polygon search for the actual implementation and details.

Real-time Tracking and Geofencing

Redis has many capabilities, and while each by itself has many uses, the real magic begins when these are used together for producing far greater results. This example leverages on Redis' PubSub mechanism, which is essentially a message broadcasting service. PubSub allows creating any number of channels, to which clients can subscribe and listen for messages. Other clients can then create and publish message to these channels, thus having Redis broadcast their message to all subscribers.

By melding the Geo Sets and PubSub together, it is near trivial to set up a real time tracking system in which every update to a member's position is sent to all interested parties. It is also just as easy to store such updates in a queue (using a Redis list or a Sorted Set) for reliable processing by event consumers. These techniques are covered in the Location Updates section of the Lua library.

Furthermore, by adding polygon search to the mix, it is possible to limit these updates only to those concerning specific events such as entry, exit and movement in defined areas. This type of capability is referred to as geofencing.

Geo Capabilities in Redis Versions

Geospatial capabilities are available with Redis version 3.2 that can be obtained here.

Redis Cloud (Redis as an enterprise-class service) and Redis Labs Enterprise Cluster (downloadable enterprise grade software) also support this version. Redis Cloud and Redis Labs Enterprise Cluster are service and software based on Redis Labs' technology that enhances open source Redis with enterprise class high availability, seamless scalability and effortless management.

See www.redislabs.com for additional details.