redislabs
home of redis

whitepaper

Using Redis As a Time Series Database: Why and How

Author: Dr.Josiah Carlson, Author of "Redis in Action"

Table of Contents

Executive Summary

Redis has been used for storing and analyzing time series data since its creation. Initially intended as a buffer and destination for logging, Redis has grown to include five explicit and three implicit structures/types that offer different methods for analyzing data with Redis. This whitepaper intends to introduce the reader to the most flexible method of using Redis for time series analysis.

A Note on Race Conditions and Transactions

While individual commands in Redis are atomic, multiple commands executed sequentially are not necessarily atomic, and may have data race conditions that could lead to incorrect behavior. To address this limitation, this whitepaper will use “transactional pipelines” and “Lua scripts” to prevent data race conditions.

With Redis and the Python client we are using to access Redis, a “transactional pipeline” (usually called a “transaction” or “**`MULTI/EXEC`** transaction” in the context of other clients) is constructed by calling the .pipeline() method on a Redis connection without arguments, or with a boolean True argument. Under the covers, the pipeline will collect all of the commands that are passed until the **`.execute()`** method is called. When the **`.execute()`** method is called, the client sends Redis the **`MULTI`** command, followed by all of the collected commands, and finally **`EXEC`**. When this group of commands is executed by Redis, Redis does so without being interrupted by any other commands, thus ensuring atomic execution.

As an additional option for providing atomic operations over a series of commands, Redis offers server-side Lua scripting. Generally speaking, Lua scripting behaves much like stored procedures in relational databases, limited to using Lua and an explicit Redis API from Lua for execution. Much like transactions, scripts in Lua generally cannot be interrupted during execution[1], though an unhandled error will cause your Lua script to terminate prematurely. Syntax-wise, we load a Lua script into Redis by calling the **`.register_script()`** method on a Redis connection object. That will return an object that can be used like a function to call the script inside Redis, instead of another method on the Redis connection, and uses a combination of the **`SCRIPT LOAD`** and **`EVALSHA`** commands to load and execute the script.

Use cases

One of the first questions brought up when talking about Redis and its use as a time-series database is “what is the use or purpose of a time-series database?” The use-cases around time series databases are more related to the data involved - specifically that your data is structured as a series of events or samples of one or more values or metrics over time. A few examples include (but are not limited to):

- Sell price and volume of a traded stock
- Total value and delivery location of an order placed at an online retailer
- Actions of a user in a video game
- Data gathered from sensors embedded inside IoT devices

[1] Read-only scripts can be interrupted if you have enabled the lua-time-limit configuration option, and the script has been executing for longer than the configured limit.

We could keep going, but basically if something happened or you made a measurement, you can record that with a timestamp. Once you have collected some events, you can perform analysis over those events either in real-time as they are collected, or after the fact as part of a more complex query.

Advanced Analysis Using a Sorted Set with Hashes

The most flexible method for storing and analyzing data as a time series combines two different structures in Redis; the Sorted Set and the Hash.

In Redis, the Sorted Set is a structure that combines the features of a hash table with those of a sorted tree (internally Redis uses a skiplist, but you can ignore that detail for now). Briefly, each entry in a Sorted Set is a combination of a string "member" and a double "score". The member acts as a key in the hash, with the score acting as the sorted value in the tree. With this combination, you can access members and scores directly by member or score value, and there are several methods for accessing the members and scores by order based on score value[2].

Storing events

In many ways, storing time series data as a combination of one or more Sorted Sets and some Hashes is one of the most common uses of Redis available today. It represents an underlying building block used to implement a wide variety of applications; from social networks like Twitter, to news sites like Reddit and Hacker News, all the way to an almost-complete relational object mapper on top of Redis itself.

For this example, let's say that we are receiving events that represent user actions on a website with 4 shared properties among all events, and a variable number of other properties depending on the event type. Our known properties are going to be: id, timestamp, type, and user. To store each individual event, we are going to use a Redis Hash, whose key is derived from the event id. For generating event ids, we can use one of a number of different sources, but for now we will generate the id using a counter in Redis. Using 64 bit Redis on a 64 bit platform will allow for 2^{63}-1 events to be generated over time, primarily limited to available memory.

When we have our data ready for recording/insertion, we will store our data as a hash, then insert a member/score pair into our Sorted Set that will map our event id (member) to the event timestamp (score). The code for recording an event is as follows.

```
def record_event(conn, event):
    id = conn.incr('event:id')
    event['id'] = id
    event_key = 'event:{id}'.format(id=id)

    pipe = conn.pipeline(True)
    pipe.hmset(event_key, event)
    pipe.zadd('events', **{id: event['timestamp']})
    pipe.execute()
```

[2] When scores are equal, items are sub-ordered by the lexicographic ordering of the members themselves.

In the record_event() function, we receive an event, get a new calculated id from Redis, assign it to the event, and generate the key where the event will be stored by concatenating the string ‘event’ and the new id, separated by a colon[3]. We then create a pipeline, and prepare to set all of the data for the event, as well as preparing to add the event id/timestamp pair to the sorted set. After the transactional pipeline has finished executing, our event is recorded and stored in Redis.

Event Analysis

From here, we have many options for analyzing our time series. We can scan the newest or oldest event ids with **`ZRANGE`**[4], maybe later pulling the events themselves for analysis. We can get the 10 or even 100 events immediately before or after a timestamp with **`ZRANGEBYSCORE`** combined with the **`LIMIT`** argument. We can count the number of events that occurred in a specific time period with **`ZCOUNT`**. Or we can even implement our analysis as a Lua script. See the below for an example that counts the number of different types of events over a provided time range, using a Lua script.

```
import json

def count_types(conn, start, end):
    counts = count_types_lua(keys=['events'], args=[start, end])
    return json.loads(counts)

count_types_lua = conn.register_script('''
local counts = {}
local ids = redis.call('ZRANGEBYSCORE', KEYS[1], ARGV[1], ARGV[2])
for i, id in ipairs(ids) do
    local type = redis.call('HGET', 'event:' .. id, 'type')
    counts[type] = (counts[type] or 0) + 1
end

return cjson.encode(counts)
''')
```

The function defined as count_types() prepares arguments to pass to the wrapped Lua script, and decodes the json-encoded mapping of event types to their counts. The Lua script first sets up a table of results (the counts variable), then reads the list of event ids in the desired time range with **`ZRANGEBYSCORE`**. After getting the ids, the script reads the type property from each event one at a time, incrementing the event count table as it goes along, finally returning a json-encoded mapping when finished.

[3] While we generally use colons as name/namespace/data separators when operating with Redis data in this whitepaper, you can feel free to use whatever character you like. Other users of Redis use a period “.”, semicolon “;”, and more. Picking some character that doesn’t usually appear in your keys or data is a good idea.

[4] ZRANGE and ZREVRANGE offer the ability to retrieve elements from a Sorted Set based on their sorted position, indexed 0 from the minimum score in the case of ZRANGE, and indexed 0 from the maximum score in the case of ZREVRANGE.

Performance Considerations and Data Modeling

As written, our method to count different event types in the specified time range works, but requires actively reading the type attribute from every event in the time range. For time ranges with only a few hundred or a few thousand events, this analysis will be reasonably fast, but what happens when our time range includes tens of thousands or even millions of events? Quite simply, Redis will block while calculating the result.

One method of addressing performance issues resulting from long script execution when analyzing event streams is to consider the queries that need to be answered in advance. In particular, if you know that you need to query for event counts of each type over arbitrary time ranges, you can keep an additional Sorted Set for each event type, each of which would only store id/timestamp pairs for events for that specific type. Then when you need to count the number of events of each type, you can perform a series of `ZCOUNT` or equivalent calls[5] and return that result instead. Let's look at what a record_event() function would look like if it also wrote to Sorted Sets based on event type.

```
def record_event_by_type(conn, event):
    id = conn.incr('event:id')
    event['id'] = id
    event_key = 'event:{id}'.format(id=id)
    type_key = 'events:{type}'.format(type=event['type'])

    ref = {id: event['timestamp']}
    pipe = conn.pipeline(True)
    pipe.hmset(event_key, event)
    pipe.zadd('events', **ref)
    pipe.zadd(type_key, **ref)
    pipe.execute()
```

Many of the same things happen in the new `record_event_by_type()` function as happened in the old `record_event()` function, though there are some new operations. In the new function, we also calculate a `type_key`, which is where we will store the index entry for this event in the type-specific Sorted Set. After preparing to add the id/timestamp pair to the events Sorted Set, we also prepare to add the id/timestamp pair to the `type_key` Sorted Set, and then perform all of our data insertions as before.

From here, to count events of the 'visit' type between two time ranges, we only need to call the `ZCOUNT` command with the specific key for the event type we want to count, along with the start and ending timestamps.

[5] `ZCOUNT` as a command does count the values in a range of data in a Sorted Set, but does so by starting at one endpoint and incrementally walking through the entire range. For ranges with many items, this can be quite expensive. As an alternative, you can use `ZRANGEBYSCORE` and `ZREVRANGEBYSCORE` to discover the members at both the starting and ending points of the range. By using `ZRANK` on the members of both ends, you can discover the indices of those members in the Sorted Set. And with both indices, a quick subtraction of the two (plus one) will return the same answer with far less computational overhead, even if it may take a few more calls to Redis.

```
def count_type(conn, type, start, end):
    type_key = 'events:{type}'.format(type=type)
    return conn.zcount(type_key, start, end)
```

If we knew all of the possible event types in advance, we could call the above **`count_type()`** function for each different type, and construct the table we did earlier in **`count_types()`**. For those cases where we don't know all of the possible event types in advance, or may be adding event types in the future, we can add each type to a Set structure, then later use the Set to discover all unique event types. Our updated record event function would read as follows.

```
def record_event_types(conn, event):
    id = conn.incr('event:id')
    event['id'] = id
    event_key = 'event:{id}'.format(id=id)
    type_key = 'events:{type}'.format(type=event['type'])

    ref = {id: event['timestamp']}
    pipe = conn.pipeline(True)
    pipe.hmset(event_key, event)
    pipe.zadd('events', **ref)
    pipe.zadd(type_key, **ref)
    pipe.sadd('event:types', event['type'])
    pipe.execute()
```

The only change from earlier is that we are adding the event type to the Set named '**`event:types'`**. And then to update our **`count_types()`** function from earlier...

```
def count_types_fast(conn, start, end):
    event_types = conn.smembers('event:types')
    counts = {}
    for event_type in event_types:
        counts[event_type] = conn.zcount(
            'events:{type}'.format(type=event_type), start, end)
    return counts
```

For more than small numbers of events in a time range, this new **`count_types_fast()`** function is going to be faster than the earlier **`count_types()`** function simply because **`ZCOUNT`** is faster than fetching each event type from a hash.

Redis as a Datastore

While the analytics tools built into Redis with its commands and Lua scripting are flexible and performant, some types of time series analysis benefit from specialized computational methods, libraries, or tools. For those cases, storing data in Redis can still make a lot of sense - as Redis is incredibly fast for storing and retrieving data.

As an example, there are only 1.2 million per-minute samples of 10 years of pricing data for a single stock symbol, which is easily stored in Redis. But to calculate almost any nontrivial function over the data with a Lua script inside Redis could take porting/debugging pre-existing optimized libraries for doing the same thing to Redis. Instead with Redis as a datastore, you can fetch time ranges of data (using the Sorted Set as your index to get keys, then the keys to get hash data like before), drop that into your existing optimized kernels for moving averages, price volatility, etc.

Why not use a relational database instead? Speed. Redis stores everything in RAM and in optimized data-structures (as in our example sorted-set). This combination of in memory optimized data-structures not only performs 3 orders of magnitude faster than even SSD-backed databases, but can also perform 1-2 orders of magnitude faster than a simple in-memory key-value datastores, or those that store data serialized in-memory.

Conclusion and Next Steps

When using Redis for time series analytics, and really any sort of analytics, it can make sense to record certain common attributes and values among different events in a common location, to aid in searching for events having those common attributes and values. We did this above with per-event-type Sorted Sets, and just started talking about using Sets. While this whitepaper primarily addresses the use of Sorted Sets, there are more structures to Redis, and there are far more options for using Redis in your analysis. Other commonly-used structures used for analytics in addition to Sorted Sets + Hashes include (but are not limited to): bitmaps, array-indexed byte strings, HyperLogLogs, Lists, Sets, and the soon to be released geo indexed Sorted Set commands[6].

Adding related data structures for more specific data access patterns is a subject that you will revisit periodically when using Redis. Almost invariably, the form that you choose to store your data in will be both a function of and a limiting factor to the types of queries you can perform. This is important because unlike a typically more familiar relational database, the queries and operations that are available in Redis are restricted based on the types used to store data.

Moving on from these few examples of analyzing time series data, you can read more about methods of building indexes for finding related data in chapter 7 of Redis in Action in the eBooks section at RedisLabs.com. In chapter 8 of Redis in Action, there is an almost complete implementation of a social network like Twitter, including followers, lists, timelines, and a streaming server, all of which are good starting points for understanding how Redis can be used to store and answer queries about timelines and events in a time series.

[6] Much like the Z*LEX commands introduced in Redis 2.8.9, which uses Sorted Sets to provide limited prefix searching over Sorted Sets, the currently-unreleased Redis 3.2 will offer limited geographic searching and indexing with GEO* commands.