The ability to sort results by referencing data stored in HASHes works well when you have a string or number that represents the actual sort order you are interested in. But what if your sort order could be a composite of a few different scores? In this article based on chapter 7 of Redis in Action, you’ll learn about ways to combine multiple scores using SETs and ZSETs, which can offer greater flexibility than calling SORT.

To save 35% on your next purchase use Promotional Code carlson0735 when you check out at http://www.manning.com.

You may also be interested in...

Sorted Indexes

When you use SORT and fetch data to sort by from HASHes, the HASHes behave very much like rows in a relational database. If we were to instead pull all of the updated times for our sample project articles into a ZSET, we could similarly order our articles by updated times by intersecting a result SET with our update time ZSET with ZINTERSTORE, using an aggregate of MAX. This works because SETs can participate as part of a ZSET intersection or union as though every element has a score of 1.

Sorting search results with ZSETs

SETs can actually be provided as arguments to the ZSET commands ZINTERSTORE and ZUNIONSTORE. When we pass SETs to these commands, Redis will consider the SET members to have scores of 1. For now, we aren’t going to worry about the scores of SETs in our operations, but we will later.

In this part, we will talk about using SETs and ZSETs together for a two part search + sort operation. When you’ve finished reading this part, you will understand why and how we would want to combine scores together as part of a document search.

Suppose we have already performed a search and have our result SET. We could sort our results with the SORT command, but that means that we can only sort based on a single value at a time. The short answer is that sometimes it simplifies our problem.

Let’s say that we want to add the ability to vote on our knowledge base articles to say that they were useful. Well, we could put the vote count in the article hash, and use SORT as we did before. That’s reasonable. But what if we also wanted to sort based on a combination of recency AND votes? We could pre-define the "score" increase for each vote. But if we don’t have enough information about how much scores should increase with each vote, then picking a score early on will just force us to have to recalculate later when we find the right number.

Instead, we will keep a ZSET of the times that articles were last updated, as well as a ZSET for the number of votes that an article has received. Both will use the article ids of the knowledge base articles as members of the ZSETs, with update times or vote count as scores, respectively. Passing similar arguments to an updated search_and_zsort() function defined in listing 1 to calculate the resulting sort order for only update times, only vote counts, or almost any relative balance between the two.

For Source Code, Sample Chapters, the Author Forum and other resources, go to http://www.manning.com/carlson/
Listing 1 An updated function to search and sort based on votes and updated times

```python
def search_and_zsort(conn, query, id=None, ttl=300, update=1, vote=0, #1
                     start=0, num=20, desc=True):
    #1 Like before, we'll optionally take a previous result id for pagination if the result is still available
    #2 We will refresh the search result's TTL if possible
    #3 If our search result expired, or if this is the first time we've searched, perform the standard SET search
    #4 Set up the scoring adjustments for balancing update time and votes. Remember: votes can be adjusted to 1, 10, 100, or higher depending on the sorting result desired.
    #5 Intersect using our helper function that we define in listing
    #6 Fetch the size of the result ZSET
    #7 Handle fetching a "page" of results
    #8 Return the results and the id for pagination

    if id and not conn.expire(id, ttl): #2
        id = None #2
    if not id: #3
        id = parse_and_search(conn, query, ttl=ttl) #3
    scored_search = {
        'id': 0, #4
        'sort:updated': update, #4
        'sort:votes': vote #4
    }
    id = zintersect(conn, scored_search, ttl) #5
    pipeline = conn.pipeline(True)
    pipeline.zcard(id) #6
    if desc: #7
        pipeline.zrevrange(id, start - 1, start - num) #7
    else:
        pipeline.zrange(id, start, start + num - 1) #7
    results = pipeline.execute() #8

    return results[0], results[1], id
```

Our `search_and_zsort()` works very much like `search_and_sort()`, differing primarily in how we sort/order our results. Rather than calling SORT, we instead perform a ZINTERSTORE operation, balancing the search result SET, the updated time ZSET, and the vote ZSET.

As part of `search_and_zsort()`, we used a helper function for handling the creation of a temporary id, the ZINTERSTORE call, and setting the expiration time of the result ZSET. You can see the `zinterstore()` and `zunionstore()` helper functions in listing 2.

Listing 2 Some helper functions for performing ZSET intersections and unions

```python
def _zset_common(conn, method, scores, ttl=30, **_scores):
    id = str(uuid.uuid4()) #1
    pipeline = conn.pipeline(True) #2
    execute = _scores.pop('_execute', True) #3
    for key, value in scores.iteritems(): #4
        _scores['idx:' + key] = value #4
    getattr(pipeline, method)(**_scores) #5
    if execute: #7
        pipeline.execute() #7
    return id #8

def zintersect(conn, items, ttl=30, **_scores): #9
    return _zset_common(conn, 'zintersect', items, ttl=ttl, **_scores) #9

def zunion(conn, items, ttl=30, **_scores): #10
    return _zset_common(conn, 'zunion', items, ttl=ttl, **_scores) #10
```

#1 Create a new temporary identifier
#2 Set up a transactional pipeline so that we have consistent results for each individual call
#3 Allow the passing of an argument to determine whether we should defer pipeline execution
#4 Add the 'idx:' prefix to our inputs
#5 Set up the call for one of the operations
#6 Instruct Redis to expire the ZSET in the future
#7 Actually execute the operation, unless explicitly instructed not to by the caller
These helper functions are very similar to SET-based helpers, the primary difference being that we are passing a dictionary through to specify scores, so we need to do a bit more work to properly prefix all of our input keys.

If we are going to try to fully replace SORT + HASHes with the more flexible ZSET, we run into one problem almost immediately, as scores in ZSETs must be floating point numbers. But we can handle this issue in many cases by converting our non-numeric data to numbers.

**Non-numeric sorting with ZSETs**

Typical comparison operations between strings will examine two strings character by character until one character is different, one string runs out of characters, or until they are found to be equal. In order to offer the same sort of functionality with string data, we need to turn strings into numbers. In this part, we will talk about methods of converting strings into numbers that can be used with Redis ZSETs in order to sort based on string prefixes. After reading this part, you should be able to sort your ZSET search results with strings.

Our first step in translating strings into numbers is understanding the limitations of what we can do. Because Redis uses IEEE 754 double precision floating point values to store scores, we are limited to at most 64 bits of storage. Due to some subtleties in the way doubles represent numbers, we can’t use all 64 bits.

Technically, we could use a bit more than the equivalent of 63 bits, but that doesn’t buy us significantly more than 63 bits, and for our case, we are only going to use 48 bits for the sake of simplicity. By using 48 bits, we limit ourselves to prefixes of 6 bytes on our data, which is often sufficient.

To convert our string into an integer, we are going to trim our string down to 6 characters (as necessary), converting each character into its ASCII value. In this part, we will talk about methods of converting strings into numbers that can be used with Redis ZSETs in order to sort based on string prefixes. After reading this part, you should be able to sort your ZSET search results with strings.

Listing 3 A function to turn a string into a numeric score

```python
def string_to_score(string, ignore_case=False):
    if ignore_case:
        #1 We can handle optional case-insensitive indexes easily, so we will
        string = string.lower()
    #1
    pieces = map(ord, string[:6])
    #2 Convert the first 6 characters of the string into their numeric values, null being 0, tab being 9, capital A being 65, etc.
    while len(pieces) < 6:
        #3 For strings that aren’t at least 6 characters long, we will add place-holding values to represent that the string was incomplete
        pieces.append(-1)
        #3
    score = 0
    #4 For each value in the converted string values, we add it to the score, taking into consideration that a null is different from a place holder
    for piece in pieces:
        score = score * 257 + piece + 1
        #4
    #5 Because we have an extra bit, we can also signify whether the string is exactly 6 characters or more, allowing us to differentiate ‘robber’ and ‘robbers’, though not ‘robbers’ and ‘robbery’
    return score * 2 + len(string) > 6
```

Most of our string_to_score() function should be pretty straightforward, except for maybe our use of -1 as a filler value for strings shorter than 6 characters, and our use of 257 as a multiplier that we use before adding each character value to the score. For many applications, being able to differentiate between “hello\\0” and “hello” can be important, so we take steps to differentiate the two, primarily by adding 1 to all ASCII values (making null 1),
and using 0 (-1 + 1) as a filler value for short strings. As a bonus, we use an extra bit to tell us whether a string is more than 6 characters long, which gives us a little more help for similar 6 character prefixes.¹

By mapping strings to scores, we are able to get a prefix comparison of a little more than the first 6 characters of our string. For non-numeric data, this is more or less what we can reasonably do without performing extensive numeric gymnastics and without running into issues with how a non-Python library transfers large integers (that may or may not have been converted to a double).

**Summary**

Limitations on SORT leads to using ZSETs to support more intricate forms of document SORTing, including combining scores for a composite sort order. In this article, we discussed sorting search results with ZSETs and methods of converting strings into numbers that can be used with Redis ZSETs.

¹ Our use of the filler for short strings uses .0337 additional bits, which when combined with the extra bit we used for longer strings, brings us to 49.0337 total bits used.
Here are some other Manning titles you might be interested in:

- **Big Data**
  Nathan Marz

- **Hadoop in Practice**
  Alex Holmes

- **HBase in Action**
  Nick Dimiduk and Amandeep Khurana

Last updated: August 13, 2012