**Redis Schema 0.3**

**Base schema**:

This schema is based on Redis Schema 0.2 and improved for data integrity, space efficiency and runtime efficiency.

Given following table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | name | gender | age | Log-in |
| 1 | A | M | 12 | 0 |
| 2 | B | F | 11 | 0 |

To implement this table in Redis, we are going to use similar schema proposed by Ryan Briones (https://www.slideshare.net/ryanbriones/the-beauty-of-simplicity-mastering-

database-design-with-redis) with one list and multiple hash table:

|  |  |  |
| --- | --- | --- |
| Type | Key | Value |
| list | ID | {1,2} |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Type | Key | Field1 | Value1 | Field2 | Value2 | Field3 | Value3 | Field4 | Value4 |
| hash | Student\_1 | name | A | age | 12 | gender | M | Log-in | 0 |
| hash | Student\_2 | name | B | age | 11 | gender | F | Log-in | 0 |

**Operations**:

**Sort**:

Sort operation will be broken down in to 2-level operation:

1. SORT ID BY Student\_\*->[attribute field]
2. Load all hash based on sorted ID

This sort operation will be 0 runtime load on local and also need one time I/O of the whole table. If certain sort has been operated many times, the sorted ID can be temporarily stored in Redis.

**Add**:

1. Add new profile hash
2. Add new primary key to list

**Search**:

O(n) operation linear search in the front-end is the current option. Finding the corresponding ID and remove hash and element from list will be sufficient. Alternative will be sets with the key of searching attributes, so this is a time-space trade-off problem now.

**Delete**:

Will be O(1) operation after search operation.

**Comparison**:

This section serves to compare Ryan Briones, schema 0.2 and schema 0.3.

We define table row numbers as n and col number as l

**Ryan Briones**:

Using strings to save primary key-attribute pairs and one set for primary key.

Pros:

* Pretty intuitive

Cons:

* when a sort on non-primary attribute is requested, a new set has to be created on that attribute. This will make constant I/O on a new set of size n. This can be very costly on multiple sort requests.
* String pairs will generate huge amount of keys and potentially long length keys, which is supper space in efficiency.

**Schema 0.2**:

Saves all table columns to lists. Using linked-list order to keep data consistency.

Pros:

* Saves a lot of spaces. All data has no duplicates in Redis.
* No new data structure needs to be generate for sorting

Cons:

* Difficult to grouping. Especially during cross sorting (sort…by…get…), require large local memory to hold data and run matching locally.
* Easy to get data corruption and hard to recover. Since there is no data structure saving rows information, if an insertion/deletion operation is incomplete and passed through. Whole database is corrupted.

**Schema 0.3**:

Saves all tuples to hash and save primary keys to a list.

Pros:

* Hash is the most space efficient data type in Redis, which is more efficient than set. It is also easier to check data integrity or reference check based on fields.
* No new data structure needs to be generated for sort operation
* No local operation needed at all.
* Hard to corrupt, easy to recover integrity.

**Redis Schema V0.31**

This documentation will focus more on different operation and how the operation workloads are distributed among clients and servers. All schema examples are from V0.3.

**Sort**:

Sort operation will be 100% run on server inside of Redis with only I/O operation for client. This allows data streaming of certain attributes in sorted order in real time. This also allows server to observe all sort requests and saving frequent sorted table as an option to save total workloads.

**Precise Search**:

This section is based on the assumed implementation of above sort method. To reduce precise search from O(n) to more efficient O(log n). We can take advantage of temporary sorted table from above section. Assume we are searching name and sorted table of name is generated from previous sort request. The precise search operation will be as follow:

*queryResult, queryID <= sort ID by student\_\*->name get student\_\*->name get student\_\*->ID*

*index = binary search (queryResult)*

*result <= hgetall student\_+queryID[index]*

The cost will be *O(log(N)+N+M\*log(M))* for the first time and O(log(N)) for the rest. This also generates sorted result in the process.

**Range search**:

This procedure will essentially be two times precise search finding the end point.

**Delete by field/ Filtering**:

Now deletion can be implemented not only by keys but also by fields (e.g: delete all people with age 18). The complexity is also reduced to *O(log(N)+A)* where *A* is the amount of filtering elements.

**Cross table operation**:

Now we add another table called history:

|  |  |  |
| --- | --- | --- |
| time | name | event |
| 10:00 | A | Log-in |
| 10:01 | A | Log-out |
| 12:21 | B | Log-in |
| 13:30 | A | Log-in |
| 13:10 | B | Log-out |