Redis Schema Design

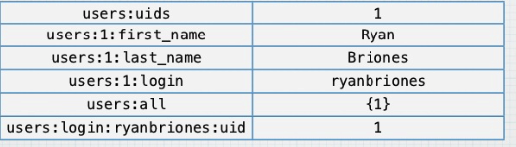
Since Redis is NoSql database, focusing on key-value pair storage. The original data structure is very one-dimensional focused and linear. To improve it and achieve functionality of tabular data structure, further schematic design has to be implemented on the basic data structure. The following tends to offer an alternative schematic design, comparing to what has been proposed by Ryan Briones:

<https://www.slideshare.net/ryanbriones/the-beauty-of-simplicity-mastering-database-design-with-redis>

**Example**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| uid | First-name | Last-name | Log-in | age |
| 1 | Ryan | Briones | ryanbriones | 30 |
| 2 | Terry | Andrew | terryandrew | 25 |
| 3 | Sad | Men | sadmen | 12 |

**Ryan Briones schema**:



Primary keys are put into one set, and every new pair is created for every relation between primary key and an attribute.

Issues:

* 2 data points are created for each attribute
* keys are supper long (a lot more bits if original data space is large)
* new data pairs need to be created for new sort on columns, e.g:

To sort last name, a new key **users:last\_names** need to be created

**Alternative**:

Only save columns in list, comparison with Ryan Briones schema:

|  |  |
| --- | --- |
| Set keys | Set value |
| Users:uids | 1,2,3 |
| Users:1:first\_name | Ryan |
| Users:1:last\_name | Briones |
| Users:1:login | ryanbriones |
| Users:1:age | 30 |
| Users:2:first\_name | Terry |
| Users:2:last\_name | Andrew |
| Users:2:login | terryandrew |
| Users:2:age | 25 |
| Users:3:first\_name | Sad |
| Users:3:last\_name | Men |
| Users:3:login | sadmen |
| Users:3:age | 12 |

|  |  |  |  |
| --- | --- | --- | --- |
| List keys | Value[0] | Value[1] | Value[2] |
| Users:uids | 1 | 2 | 3 |
| Users:first\_name | Ryan | Terry | Sad |
| Users:last\_name | Biones | Andrew | Men |
| Users:login | ryanbriones | terryandrew | sadmen |
| Users:age | 30 | 25 | 12 |

**Advantages**:

* Old schema has 26 data points and new schema has 20 data points. The difference will increase when more data is added (every new person profile will make 5 data points difference). This is mentioned in memory optimization of Redis. The fewer the keys, the less of memory space it is going to take.
* List will keep data consistency since it is ordered and rpush/lpush has consistent insertion location.
* Shorter keys
* Easier sort operation, example of sorting name by age for alternative schema:

**SORT Users:first\_name by Users:age**

For old schema, Users\_age has to be created for sorting.

**Efficiency and space trade-offs**:

The above schema is mainly used to fast retrieve only one column information. To retrieve whole tabular information, the total complexity would be ***column number \* sort operation time***, which can be costly if the table has huge number of columns. To improve the efficiency, reference data pairs can be created for fast retrieve.

Based on the above example, new schema would be added as:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Hash keys | First\_name | Last\_name | Log-in | Age |
| 1 | Ryan | Biones | ryanbriones | 30 |
| 2 | Terry | Andrew | terryandrew | 25 |
| 3 | Sad | Men | sadmen | 12 |

Where the hash keys have to be unique primary keys in table. All the information retrieval based on sort operation can now be reduced to one-time sort operation time. The procedure is described in the following:

**SORT [primary keys] by [sort attribute]**

**MGET [primary keys]**

This will retrieve whole table information based on sort in one-time sort operation. However, this will double the data space.

**Other operations:**

**Add**:

Use RPUSH for all new data to keep consistency.

**Search**:

Node.js supports search and return index.

**Delete**:

Use returned index to delete data from all lists.