



build unknown

NEW! Try out our new standalone bitmapist-server, which improves memory efficiency 443 times and makes your setup much cheaper to run (and more scaleable). It's fully compatible with bitmapist that runs on Redis.

bitmapist: a powerful analytics library for Redis

This Python library makes it possible to implement real-time, highly scalable analytics that can answer following questions:

- Has user 123 been online today? This week? This month?
- Has user 123 performed action "X"?
- How many users have been active this month? This hour?
- How many unique users have performed action "X" this week?
- How many % of users that were active last week are still active?
- How many % of users that were active last month are still active this month?
- What users performed action "X"?

This library is very easy to use and enables you to create your own reports easily.

Using Redis bitmaps you can store events for millions of users in a very little amount of memory (megabytes). You should be careful about using huge ids as this could require larger amounts of memory. Ids should be in range $[0, 2^{32})$.

Additionally bitmapist can generate cohort graphs that can do following: * Cohort over user retention
* How many % of users that were active last [days, weeks, months] are still active? * How many % of users that performed action X also performed action Y (and this over time) * And a lot of other things!

If you want to read more about bitmaps please read following:

- <http://blog.getspool.com/2011/11/29/fast-easy-realtime-metrics-using-redis-bitmaps/>
- <http://redis.io/commands/setbit>
- http://en.wikipedia.org/wiki/Bit_array
- <http://www.slideshare.net/crashlytics/crashlytics-on-redis-analytics>

Installation

Can be installed very easily via:

```
1 $ pip install bitmapist
```

Ports

- PHP port: <https://github.com/jeremyFreeAgent/Bitter>

Examples

Setting things up:

```
1 from datetime import datetime, timedelta
2 from bitmapist import setup_redis, delete_all_events, mark_event,\
3                     MonthEvents, WeekEvents, DayEvents, HourEvents,\
4                     BitOpAnd, BitOpOr
5
6 now = datetime.utcnow()
7 last_month = datetime.utcnow() - timedelta(days=30)
```

Mark user 123 as active and has played a song:

```
1 mark_event('active', 123)
2 mark_event('song:played', 123)
```

Answer if user 123 has been active this month:

```
1 assert 123 in MonthEvents('active', now.year, now.month)
2 assert 123 in MonthEvents('song:played', now.year, now.month)
3 assert MonthEvents('active', now.year, now.month).has_events_marked()
   == True
```

How many users have been active this week?:

```
1 print(len(WeekEvents('active', now.year, now.isocalendar()[1])))
```

Iterate over all users active this week:

```
1 for uid in WeekEvents('active'):
2     print(uid)
```

If you're interested in “current events”, you can omit extra `now.whatever` arguments. Events will be populated with current time automatically.

For example, these two calls are equivalent:

```
1
2 MonthEvents('active') == MonthEvents('active', now.year, now.month)
```

Additionally, for the sake of uniformity, you can create an event from any datetime object with a `from_date` static method.

```
1
2 MonthEvents('active').from_date(now) == MonthEvents('active', now.year,
   now.month)
```

Get the list of these users (user ids):

```
1 print(list(WeekEvents('active', now.year, now.isocalendar()[1])))
```

There are special methods `prev` and `next` returning “sibling” events and allowing you to walk through events in time without any sophisticated iterators. A `delta` method allows you to “jump” forward or backward for more than one step. Uniform API allows you to use all types of base events (from hour to year) with the same code.

```
1
2 current_month = MonthEvents()
3 prev_month = current_month.prev()
4 next_month = current_month.next()
5 year_ago = current_month.delta(-12)
```

Every event object has `period_start` and `period_end` methods to find a time span of the event. This can be useful for caching values when the caching of “events in future” is not desirable:

```
1
```

```
2 ev = MonthEvent('active', dt)
3 if ev.period_end() < now:
4     cache.set('active_users_<...>', len(ev))
```

As something new tracking hourly is disabled (to save memory!) To enable it as default do::

```
1 import bitmapist
2 bitmapist.TRACK_HOURLY = True
```

Additionally you can supply an extra argument to `mark_event` to bypass the default value::

```
1 mark_event('active', 123, track_hourly=False)
```

Unique events

Sometimes the date of the event makes little or no sense, for example, to filter out your premium accounts, or in A/B testing. There is a `UniqueEvents` model for this purpose. The model creates only one Redis key and doesn't depend on the date.

You can combine unique events with other types of events.

A/B testing example:

```
1
2 active_today = DailyEvents('active')
3 a = UniqueEvents('signup_form:classic')
4 b = UniqueEvents('signup_form:new')
5
6 print("Active users, signed up with classic form", len(active & a))
7 print("Active users, signed up with new form", len(active & b))
```

Generic filter example

```
1
2 def premium_up(uid):
3     # called when user promoted to premium
4     ...
5     mark_unique('premium', uid)
6
7
8 def premium_down(uid):
9     # called when user loses the premium status
10    ...
11    unmark_unique('premium', uid)
12
13 active_today = DailyEvents('active')
14 premium = UniqueEvents('premium')
15
```

```
16 # Add extra Karma for all premium users active today,
17 # just because today is a special day
18 for uid in premium & active_today:
19     add_extra_karma(uid)
```

To get the best of two worlds you can mark unique event and regular bitmapist events at the same time.

```
1 def premium_up(uid):
2     # called when user promoted to premium
3     ...
4     mark_event('premium', uid, track_unique=True)
```

Perform bit operations

How many users that have been active last month are still active this month?

```
1 active_2_months = BitOpAnd(
2     MonthEvents('active', last_month.year, last_month.month),
3     MonthEvents('active', now.year, now.month)
4 )
5 print(len(active_2_months))
6
7 # Is 123 active for 2 months?
8 assert 123 in active_2_months
```

Alternatively, you can use standard Python syntax for bitwise operations.

```
1 last_month_event = MonthEvents('active', last_month.year, last_month.
    month)
2 this_month_event = MonthEvents('active', now.year, now.month)
3 active_two_months = last_month_event & this_month_event
```

Operators `&`, `|`, `^` and `~` supported.

Work with nested bit operations (imagine what you can do with this ;-))!

```
1 active_2_months = BitOpAnd(
2     BitOpAnd(
3         MonthEvents('active', last_month.year, last_month.month),
4         MonthEvents('active', now.year, now.month)
5     ),
6     MonthEvents('active', now.year, now.month)
7 )
8 print(len(active_2_months))
9 assert 123 in active_2_months
10
11 # Delete the temporary AND operation
12 active_2_months.delete()
```

Deleting

If you want to permanently remove marked events for any time period you can use the `delete()` method:

```
1 last_month_event = MonthEvents('active', last_month.year, last_month.month)
2 last_month_event.delete()
```

If you want to remove all bitmapist events use:

```
1 bitmapist.delete_all_events()
```

When using Bit Operations (ie `BitOpAnd`) you can (and probably should) delete the results unless you want them cached. There are different ways to go about this:

```
1 active_2_months = BitOpAnd(
2     MonthEvents('active', last_month.year, last_month.month),
3     MonthEvents('active', now.year, now.month)
4 )
5 # Delete the temporary AND operation
6 active_2_months.delete()
7
8 # delete all bit operations created in runtime up to this point
9 bitmapist.delete_runtime_bitop_keys()
10
11 # delete all bit operations (slow if you have many millions of keys in
12   Redis)
13 bitmapist.delete_temporary_bitop_keys()
```

bitmapist cohort

With `bitmapist cohort` you can get a form and a table rendering of the data you keep in `bitmapist`. If this sounds confusing please look at `Mixpanel`.

Here's a simple example of how to generate a form and a rendering of the data you have inside `bitmapist`:

```
1 from bitmapist import cohort
2
3 html_form = cohort.render_html_form(
4     action_url='/_Cohort',
5     selections1=[ ('Are Active', 'user:active'), ],
6     selections2=[ ('Task completed', 'task:complete'), ]
7 )
8 print(html_form)
```

```

9
10 dates_data = cohort.get_dates_data(select1='user:active',
11                                   select2='task:complete',
12                                   time_group='days')
13
14 html_data = cohort.render_html_data(dates_data,
15                                    time_group='days')
16
17 print(html_data)
18
19 # All the arguments should come from the FORM element (html_form)
20 # but to make things more clear I have filled them in directly

```

This will render something similar to this:

People actions
 Show me people who and then came back and

Group by
 Group by

		0	1	2	3	4	5
19 Nov, 2012	43800	70.66%	51.53%	47.85%	33.68%	31.58%	20.16%
20 Nov, 2012	48340	67.31%	46.69%	33.7%	31.11%	19.01%	21.45%
21 Nov, 2012	47790	67.57%	37.1%	34.42%	20.61%	23.23%	47.73%
22 Nov, 2012	35440	65.27%	42.55%	24.38%	25.4%	47.94%	46.44%
23 Nov, 2012	37410	64.82%	26.6%	26.62%	49.91%	46.54%	45.9%
24 Nov, 2012	25130	61.16%	37.13%	50.58%	46.92%	45.36%	42.66%
25 Nov, 2012	29980	60.51%	53.9%	48.77%	46.2%	44.3%	40.13%

Contributing

Please see our guide [here](#)

Local Development

We use Poetry for dependency management & packaging. Please see [here](#) for setup instructions.

Once you have Poetry installed, you can run the following to install the dependencies in a virtual environment:

```
1 poetry install
```

Testing

To run our tests will need to ensure a local redis server is installed.

We use pytest to run unittests which you can run in a poetry shell with

```
1 poetry run pytest
```

Releasing new versions

- Bump version in `pyproject.toml`
- Update the CHANGELOG
- Commit the changes with a commit message “Version X.X.X”
- Tag the current commit with `vX.X.X`
- Create a new release on GitHub named `vX.X.X`
- GitHub Actions will publish the new version to PIP for you

Legal

Copyright: 2012 by Doist Ltd.

License: BSD