Using Redis as a Cache Backend in Magento

Written by:
Alexey Samorukov
Aleksandr Lozhechnik
Kirill Morozov

Table of Contents

Magento EC^g

Problems with the TwoLevels

Cache Backend

Problems with the TwoLevels Cache Backend

The most common cache storage recommendation for Magento users running multiple web nodes is to implement the TwoLevels backend – that is, to use Memcache together with the database. However, this configuration often causes the following issues:

1. First, the core_cache_tag table constantly grows. On average, each web site has about 5 million records. If a system has multiple web sites and web stores with large catalogs, it can easily grow to 15 million records in less than a day. Insertion into core_cache_tag leads to issues with the MySQL server, including performance degradation. In this context, a tag is an identifier that classifies different types of Magento cache objects.

2. Second, the TwoLevels backend is more difficult to maintain because two services are required to make it work which makes it difficult to analyze cache content when necessary. Further, memcached itself has limitations such as a maximum object size and fixed bucket sizes which also contribute to difficult maintenance

3. Finally, the TwoLevels backend does not scale well, since using the database as part of the cache backend adds additional load to the master database server. Additionally, there is no reliable method for memcached replication.

Magento ECg

Confirming the Issue

Confirming The Issue

The issues with the TwoLevels Backend approach were reported several times; based on these reports, the Magento Expert Consulting Group began looking for alternative approaches for a reliable and scalable cache backend solution. First, the issue was replicated using a merchant's production database and a web-crawler based on the url_rewrites table. Additionally, we also tried to disable all of the re-indexing processes to ensure they would not skew the testing results. After five hours of crawling, we noted that the core_cache_table contained more than **10 million records**, and that under high concurrency, INSERT queries to the same table were visible in the MySQL process list.

Our tests showed that administrative operations performed significantly slow, taking more than one and a half minutes, and that most of this time was spent in conversation with memcached. Ngrep was used to monitor memcached traffic, and a typical line from the captured memcached traffic appeared as follows:

```
delete 0ed_CONTAINER_BREADCRUMBS_3A295717FC57647DA5111A28AC0F43D7
```

This confirmed that the constantly growing core_cache_tag table issue could be reproduced and was not related to the merchant's custom modules. The only affected items were created by Magento's Full Page Cache module.

Magento ECg

Solving the Issue Using the

Redis Cache Backend

Solving The Issue Using The Redis Cache Backend

To integrate Magento versions 1.12.x (Enterprise) or 1.7.x (Community) and earlier with Redis, you must integrate the third-party backend code Cm_Cache_Backend_Redis. With Enterprise 1.13.x and Community 1.8.x, no third-party code is required.

This code was installed in our web servers and a Redis server with minimal configuration changes. Some of the benefits we expected from this cache backend and Redis server were:

- Redis supports different types of data and it's easy to implement a tagging system using SET and HASH objects. This was implemented in the backend.
- The backend also supports tag-based cache cleanup without 'foreach' loops. This was implemented as well.
- The backend has garbage collector support, but by default it is not enabled. We did not enable it in our tests.
- Redis supports on-disk save and master/slave replication. This is a very popular feature and has been requested by many of our customers, and it is not supported by Memcached. With the replication setup, it's possible to avoid a single point of failure and to achieve high availability.
- The PHP-Redis extension also works for PHP session storage, making it possible to completely replace memcached with Redis.
- Redis provides much better eviction control and its backend is written with eviction support in mind.
- Redis supports multiple databases that use the same server instance so we can use different databases for Cache, FPC, and sessions without starting many processes listening on different ports.
- It is very easy to save the Redis database (using the BGSAVE command) and load it in a development environment later. It's also possible to get the list of full REDIS keys and dump their content, which makes debugging quite easy.
- Redis supports monitoring functionality to watch requests in real-time. The ability to see all the requests processed by the server is useful to spot bugs in an application both when using Redis as a database and when using it as a distributed caching system.

Magento EC^g

Testing and Benchmarking

The Solution

Testing and Benchmarking the Solution

We tested Redis using the exact same test system as we used with memcached. The difference is that with Redis it was not possible to produce millions of objects in the cache because Redis' architecture is more efficient. After six hours of web crawling, we were able to get approximately **50,000 records** in the cache.

Our goal was to:

- Compare memcached+db with Redis + PHPRedis
- Compate Redis + PHPRedis with Redis but not PHPRedis

Our tests showed that the cache-related operations were much faster, the database had no hanging queries, and no errors or problems were found. After five hours of testing, we tried to save a product in the Admin Panel and found that total time spent in cache was less than one second, and the product save took only **six seconds** with a remote connection and single server configuration.

We monitored Redis traffic using ngrep.

More Research into the TwoLevels Backend Issue

We decided to replicate the memcached issue in a new Magento EE1.13 installation. To do this, we created a sample database which contained 100,000 SKUs and eight stores. Our tests used two virtual machines (1W+1DB). We used **enterprise_url_rewrite** as the data source for the web crawler to get enough tags in the **core_cache_tags** table. After about seven hours of pre-heating the cache, we got about 100,000 tags:

```
mysql> select count(distinct(tag)) from core_cache_
tag;
+----------------------+
| count(distinct(tag)) |
+----------------------+
|                92761 |
+----------------------+
1 row in set (2.47 sec)
```

After this operation, we found that the product save took about 20 seconds. Most of the time was spent in the in memcached partial invalidation loop, not in the database. When compared with the same amount of tags in Redis, the same operation took less than one second. We decided to test Full Page Cache performance by regularly fetching cached page on Redis and memcached. Results follow:

Magento Enterprise 1.13 and Memcached

```
$ siege http://m13.local/          -b -c 50 -v -t 15s -q
Lifting the server siege...        done.
Transactions:                      3744 hits
Availability:                      100.00 %
Elapsed time:                      14.64 secs
Data transferred:                  35.91 MB
Response time:                     0.19 secs
Transaction rate:                  255.74 trans/sec
Throughput:                        2.45 MB/sec
Concurrency:                       49.60
Successful transactions:           3744
Failed transactions:               0
Longest transaction:               0.45
Shortest transaction:              0.08
```

Magento Enterprise Edition 1.13 with Redis and the PHP Redis Extension

```
$ siege http://m13.local/          -b -c 50 -v -t 15s -q
Lifting the server siege...        done.
Transactions:                      4384 hits
Availability:                      100.00 %
Elapsed time:                      14.71 secs
Data transferred:                  42.01 MB
Response time:                     0.17 secs
Transaction rate:                  298.03 trans/sec
Throughput:                        2.86 MB/sec
Concurrency:                       49.66
Successful transactions:           4384
Failed transactions:               0
Longest transaction:               0.38
Shortest Transaction:              0.06
```

As you can see, with FPC enabled the Redis back end is faster and provides better response time. We also tested Redis in plain PHP mode, without the PHPRedis extension enabled:

```
[root@DE1P1MGDB102 data1]# siege http://m13.local/  -b -c 50 -v -t 15s -q
Lifting the server siege...         done.
Transactions:                       4324 hits
Availability:                       100.00 %
Elapsed time:                       14.94 secs
Data transferred:                   41.43 MB
Response time:                      0.17 secs
Transaction rate:                   289.42 trans/sec
Throughput:                         2.77 MB/sec
Concurrency:                        49.70
Successful transactions:            4324
Failed transactions:                0
Longest transaction:                0.37
Shortest transaction:               0.06
```

These results are still better compared to memcached, but it is also clear that PHPRedis adds some additional non-critical performance benefits.

Potential Issues with Redis as a Cache Backend

There are some possible issues with using Redis as a cache backend which should be mentioned.

- First, Redis is not yet widely adopted as memcached which means that not as many hosting providers will support it.
- Additionally, the PHPRedis extension is not available on many OS repositories and is in active development on github. This situation should improve soon because PHPRedis is now also available as a PECL package. The package also supports a standalone mode, which is written in pure PHP. Performance testing needs to be done on the standalone mode to quantify any performance improvements.
- Finally, Magento needs to discuss integrating PHPRedis. This code is licensed under the New BSD License. Magento EE 1.13 and Magento CE 1.8 integrated this code, so it should not be an issue in those versions.

Magento EC^g

Implementing and Testing Redis in a Production Environment

Implementing and Testing Redis in a Production Environment

With the help of one of our merchants, we were able to implement and test Redis as a backend cache solution for Magento Enterprise on their production environment. This merchant was an excellent fit for the test, as they had been experiencing the previously mentioned issues with the two-level back end, and were running Magento Enterprise Edition 1.12.0.2 with a limited number of local modules. Additionally, the client's environment had four web nodes and two database servers, on which they hosted a single website with twelve stores and close to 30,000 SKUs. Up to nine people could be logged in to the Admin Panel at the same time.

Product Test Configurations

Redis Configuration

For the production environment test, Redis version 2.6.13 was installed and configured from the CentALT repository.

The Redis configuration was as follows:

```
daemonize yes
pidfile /var/run/redis/redis.pid
port 6379
timeout 0
loglevel notice
logfile /var/log/redis/redis.log
databases 2
rdbcompression no
dbfilename dump.rdb
dir /var/lib/redis/
slave-serve-stale-data yes
maxmemory 8gb
maxmemory-policy volatile-lru
appendonly no
appendfsync everysec
no-appendfsync-on-rewrite no
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
slowlog-log-slower-than 10000
```

```
slowlog-max-len 1024
list-max-ziplist-entries 512
list-max-ziplist-value 64
set-max-intset-entries 512
zset-max-ziplist-entries 128
zset-max-ziplist-value 64
activerehashing yes
```

As is shown in the configuration list, memory usage was limited to 8GB with the volatile-lru eviction policy. This was done in order to test whether volatile-lru was the best option as well as to test how evictions affect Redis performance.

Magento Configuration

The code for the test can be found at http://github.com/colinmollenhour/Cm_Cache_Backend_Redis. The Credis library was copied to the app/ directory and the Redis.php file from Cm_Cache_Backend_Redis was installed to the app/code/community/Cm/Cache/Backend/directory. A sample local.xml which was used is provided as Appendix A to this article.

The Magento configuration shows that as far as Full Page Cache is concerned, we are disabling data compression because, as discussed earlier, Full Page Cache already uses compression, and we don't need Redis to compress data on top of it. In addition, note the usage of different databases for Full Page Cache and Magento cache for better performance and easier debugging.

Production Test Results

After approximately 12 hours of testing with the TwoLevels backend solution, the core_cache_tag table had more than 15 million records and the product save action was timing out. We found that the main cause of this behavior was the mass delete from memcached.

After installing and enabling the Redis cache backend, we found that the cache worked faster and that the product save action was no longer affected by its size. However, we did find a major issue: the cache size was continuously growing.

Production Test Issue: Continuously Growing Cache Size

Root Cause and Conclusion

After reaching the defined limit in the Redis configuration, our test experienced a major slowdown caused by Redis mass evictions. A short-term solution was to flush the Redis cache. After analyzing

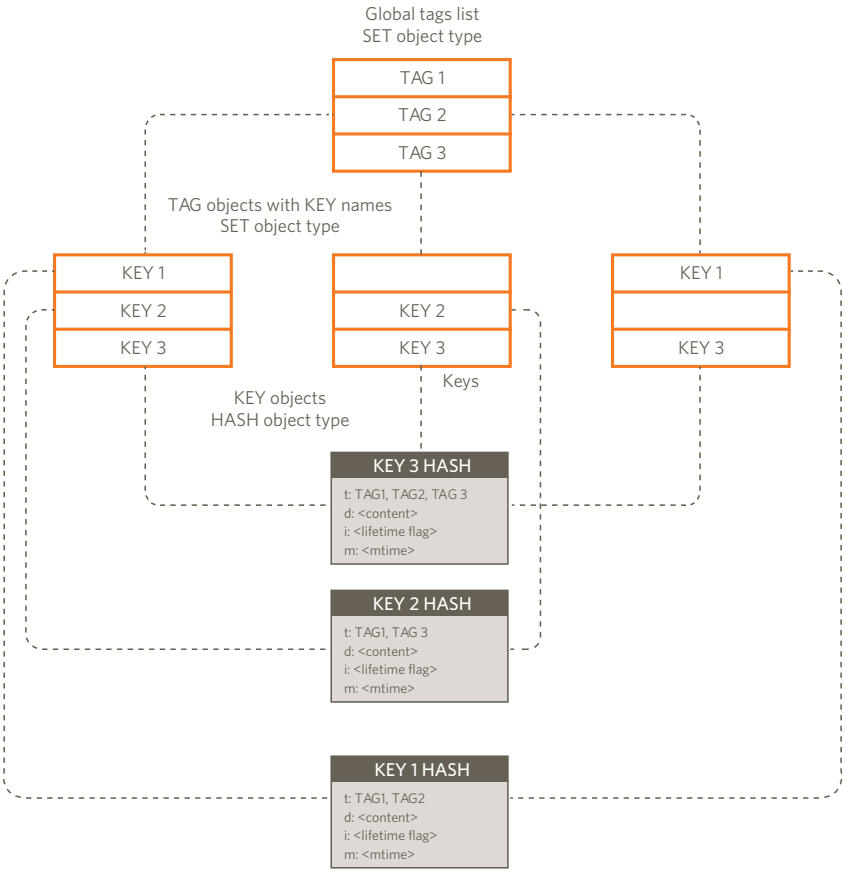
cache content using the BGSAVE command and restoring the cache locally, we found that most of the Full Page Cache objects have a maximum possible TTL setting defined in the Redis backend as one month. (Note: While analyzing the cache content, we also created a cache dump that can be converted to CSV using the RDB tools project.)

Our conclusion was that Full Page Cache code does not set a TTL at all; therefore, the TTL is set by the cache backend. When using the file-system for backend cache storage, the TTL maximum amount is indefinite, while for Redis it is one month.

Work-around

Full Page Cache typically fills cache storage quickly. The workaround for the TTL issue was to extend the Redis backend with a new option to redefine the maximum TTL for undefined TTL objects. We specified a TTL of 12 hours. After about 14 hours, the cache size stabilized at about 2.5 GB +/- 500-600Mb. Performance was perfect, latency was improved and no additional problems were found.

A note about data compression: Redis provides transparent compression of stored data to minimize the network and memory usage. It's a good idea to keep it enabled for normal Magento cache; however, all Full Page Cache data is already compressed by the Full Page Cache code. Therefore, it does not make sense to enable Redis data compression while using Full Page Cache.



Tag Clean-up Procedure and Script

The Redis backend uses the SET object to store a global list of all tags. Every item in the SET corresponds to a TAG object (as well as a SET type) which stores a names-only list of the keys. Every KEY object (HASH type) stores key data and metadata. An example follows:

Only the KEY object has an expiration set (using i: <lifetime flag>). This prevents cache structure damage on evictions and avoids "lost" KEY objects. One concern is that if KEY objects expire, the corresponding TAG objects will not be purged from the global tag list. To cleanup "empty" tags, the backend implements a garbage collection process.

Implementing this garbage collection process to function from within Magento is not a good approach because the collector can take a long time to complete (typically minutes). To solve this, we created a simple wrapper (rediscli.php) to run the garbage collector outside of Magento or the Zend Framework, making it possible to use it on a standalone Redis server. This script is located on github.

```
rediscli.php supports command-line execution from crontab.
rediscli.php syntax:
Usage: rediscli.php <args>
    -s <server> - server address
    -p <port> - server port
    -v - show process status
    -d <databases> - list of the databases, comma separated
Example: rediscli.php -s 127.0.0.1 -p 6379 -d 0,1
```

Sample cron entry:

```
15 2 * * * /usr/bin/php /root/cm_redis_tools/rediscli.php -s 127.0.0.1 -p 6379 -d 0,1
```

Magento ECg

ECg Changes in Magento Enterprise Edition 1.13 and Community Edition 1.8

Changes in Magento Enterprise Edition 1.13 and Community Edition 1.8

Redis has been introduced natively to Magento Enterprise 1.13 with minimal changes which are mostly related to code formatting and naming conventions. In fact, the only change is the name of the class in the configuration (local.xml) – instead of the `<backend>Cm_Cache_Backend_Redis</backend>` line, `<backend>Mage_Cache_Backend_Redis</backend>` should be used.

Additional Links

Redis Server - http://redis.io/

Zend cache backend: https://github.com/colinmollenhour/Cm_Cache_Backend_Redis

Tools to analyze Redis database (RDB) - https://github.com/sripathikrishnan/redis-rdb-tools

PHP Redis extension - https://github.com/nicolasff/phpredis

Got Questions?

Contact ECG at consulting@magento.com