Design pattern for memcached data caching

Asked 16 years, 1 month ago Modified 14 years, 4 months ago Viewed 8k times

Part of PHP Collective

10

It's easy to wrap optional memcached caching around your existing database queries. For example:

Old (DB-only):

```
function getX
    x = get from db
    return x
end
```

New (DB with memcache):

```
function getX
    x = get from memcache
    if found
      return x
    endif

    x = get from db
    set x in memcache
    return x
end
```

The thing is though, that's not always how you want to cache. For instance take the following two queries:

```
-- get all items (recordset)
SELECT * FROM items;

-- get one item (record)
SELECT * FROM items WHERE pkid = 42;
```

If I was to use the above pseudo-code to handle the caching, I would be storing all fields of item 42 twice. Once in the big record set and once on its own. Whereas I'd rather do something like this:

```
SELECT pkid FROM items;
```

and cache that index of PK's. Then cache each record individually as well.

So in summary, the data access strategy that will work best for the DB doesn't neatly fit the memcache strategy. Since I want the memcache layer to be optional (i.e. if memcache is down, the site still works) I kind of want to have the best of both worlds, but to do so, I'm pretty sure I'll need to maintain a lot of the queries in 2 different forms (1. fetch index, then records; and 2. fetch recordset in one query). It gets more complicated with pagination. With the DB you'd do LIMIT/OFFSET SQL queries, but with memcache you'd just fetch the index of PK's and then batch-get the relevant slice of the array.

I'm not sure how to neatly design this, does anyone have any suggestions?

Better yet, if you've come up against this yourself. How do you handle it?

PHP php database design-patterns oop memcached

Share

Improve this question

Follow

edited Nov 10, 2008 at 0:12

asked Nov 10, 2008 at 0:01

Jamie

4 Answers

Sorted by: Highest score (default)

4

If you're using a cache then, to get the most out of it, you have to accept that your data will always be stale to an extent, and that some portions of the data will be out of sync with each other. Trying to keep all the records up to date by maintaining a single copy is something best left to relational databases, so if this is the behaviour you need then you're probably better off with a powerful 64-bit DB server with a lot of RAM so it can perform its own internal caching.

If you can accept stale data (which you'll need to if real scalability is important) then one approach is to just throw the whole result set into the cache; don't worry about duplication. RAM is cheap. If you find your cache is getting full then just buy more RAM and/or cache servers. For example if you have a query that represents items 1-24 in a set filtered by conditions X and Y then use a cache key that contains all this information, and then when asked for that same search again just return the entire result set from the cache. You either get the full result set from the cache in one hit, or you go to the database.

The hardest thing is working out how much data can be stale, and how stale it can be without either (a) people noticing too much, or (b) breaking business requirements such as minimum update intervals.

This approach works well for read-mostly applications, particularly ones that have paged queries and/or a finite set of filter criteria for the data. It also means that your application works exactly the same with the cache on or off, just with 0% hit rate when the cache is off. It's the approach we take at blinkBox in almost all cases.

Share Improve this answer Follow

answered Nov 10, 2008 at 2:01

Greg Beech

136k ● 45 ● 209 ● 250

3

Read about the Identity Map pattern. This is a way to make sure you only keep one copy of a given row in your application space. Whether you store it in memcached or just plain objects, this is a way to handle what you want. I would guess that Identity Map is best used when you typically fetch one row at a time.

When you fetch whole subsets of a table, then you have to process each row individually. You might frequently have the dilemma of whether you're getting the best use out of your cache, because if 99% of your rows are in the cache but one requires fetching from the database, you have to run the SQL query anyway (at least once).

You could transform the SQL query to fetch only rows that aren't in the cache, but it's nontrivial to perform this transformation automatically without making the SQL query more costly.

Share Improve this answer Follow

answered Nov 10, 2008 at 1:38

Bill Karwin

561k ● 87 ● 698 ● 854

1

Well, I guess that's something you'll have to live with. Memcahced will work the best if you don't really do stuff in batches. For example it's great for stuff like "where are the things for this user? Here is a bunch of things for this user." This doesn't really mean that this query doesn't do batches. Of course it will - if some of the user stuff is stuff like his/her posts.

I guess the problem you'll have is cases where you are mixing queries that need to get an item from the DB on its own and some that get bunch of the same kind of the previous items.

There is always a flip side to the situation. If you really want to get hairy with your implementation you can change your batch queries to not include the items already present in memcached.Very very ugly...

In my opinion it always comes down to "which queries do I **really** want to cache?"

EDIT:

The way I would go about this is:

- Single-item query - if in memcached, use that one, otherwise fetch from DB and update memcached.
- Batch query - don't worry about which items are in memcached, just get everything and update memcached.

This of course assumes that the batch queries already take hell a lot more time to complete and so it I'm already spending so much time I can live with external lookups to already cached items.

However, eventually, your cache will contain a lot of the items if you use the batch queries a lot. Therefore you'll have to strike the balance for determining at which point you still want to perform the database lookups. Good thing is if the batch query is earlier in the life cycle of your applications, then everything will be cached earlier. After the first batch query you can tell yourself that you don't need to fetch from DB anymore unless the data in the cache is invalidated by updates or deletes.

Share

Improve this answer

Follow

edited Nov 10, 2008 at 1:59

answered Nov 10, 2008 at 1:22

Cem Catikkas

7,174 ● 4 ● 30 ● 33

Thanks for your answer Cem. Assuming that I am only caching what I really need to cache. Would you have any ideas how to manage the two strategies (DB and DB+memcache) with the least duplication of code? – doekman Nov 10, 2008 at 1:37

1

Here's my understanding of how NHibernate (and therefore probably Hibernate) does it. It has 4 caches:

- row cache: this caches DB rows. The cache key is TableName#id, the other entries are the row values.
- query cache: this caches the results returned for a particular query. The cache key is the query with parameters, the data is a list of the TableName#id row keys that were returned as query results.
- collections cache: this caches the child objects of any given parent (which NHibernate allows to be lazy-loaded.) So if you access myCompany.Employees, the employees collection will be cached in the collections cache. The cache key is CollectionName#entityId, the data is a list of the TableName#id row keys for the child rows.

- table update cache: a list of each table and when it was last updated. If a table was updated after the data was cached, the data is considered stale.

This is a pretty flexible solution, is very efficient space-wise, and guarantees that the data won't be stale. The disadvantage is that a single query can require several round-trips to the cache, which can be a problem if the cache server is on the network.

answered Aug 24, 2010 at 13:16

zcrar70

3,174 ● 2 ● 25 ● 18