

[nkallen](#)

Magic Scaling Sprinkles

- [About](#)
- [Archives](#)
- [RSS Feed](#)

Introducing Cache Money

In [Uncategorized](#) on **December 11, 2008** at **6:25 am**

Pre-requisite: please read my article on [Write-through caching](#) to understand why this is useful.

Most caching solutions in the Rails world involve something like Cache-Fu: an alternative API to ActiveRecord that explicitly annotates all call sites with cache rules.

- `User.find(1)` becomes `User.get_cache(1)`
- `User.find(:all, ...)` becomes `User.get_cache("query_name", :ttl => 5.minutes) { User.find(:all, ...)}`


I hate this kind of interface, which places the burden on the caller and meekly surrenders any attempt at encapsulation. Your codebase will be littered with haphazard cache rules in your controllers, views, and models.

But even worse are the explicit cache expiry rules. As you cache non-trivial queries, you'll have to find all the of the writes in your system that could possibly invalidate the results of the query. It's a tedious and onerous effort: after hundreds of hours of debugging you'll finally get `expire_cache` in just the right places.

A solution to this brittle, messy coding style is now available, and ready for production use. [`Cache](#)

[Money`](#) is a plugin for ActiveRecord that **transparently** provides [write-through and read-through caching](#) functionality using Memcached. With [`Cache Money`](#), queries are **automatically cached** for you; and similarly, **cache expiry happens automatically** as `after_save` and `after_destroy` events.

This doesn't just apply to trivial queries. Very complex, sophisticated queries are handled effortlessly; the vast majority of ActiveRecord usage is transparently materialized, indexed, and kept fresh in Memcached. Here are some examples:

- `User.find(1)`
- `User.find(:all, :conditions => { :screen_name => 'bob' })`
- `Friendship.find(:all, :conditions => ['friendships.creator_id = ? AND friendships.receiver_id = ?', ...])`
- `users.direct_messages`
- `users.direct_messages.find(1)`
- `users.direct_messages.count`
- `User.find(:all, :limit => 10,  rder => 'id DESC')`

All of these, and much more will automatically be cached and kept fresh as you write to the database. [This greatly lessens the load on your database and makes your site impervious to catastrophic replication lag.](#)

The way [`Cache Money`](#) works is by materializing the equivalent of database indices in Memcached. It's as if you store your indices in a distributed hash table instead of an in-process BTree. Just as with a database, you declare your indices:

```
class User < ActiveRecord::Base
  index :screen_name
end
```

```
class Friendship
  index [:creator_id, :receiver_id]
end
```

```
class DirectMessage
  index :user_id
  index [:user_id, :id]
end
```

There are lots of configurable options like TTLs:

```
index :user_id, :ttl => 1.day
```

You can also specify limits to ensure that your indices do not grow too large:

```
index :user_id, :limit => 500, :buffer => 20
```

(This keeps a rolling window of 500 items. The `buffer` option indicates how many “extra” you want to keep around in case of deletes in order to maintain at least 500 items. If more than 20 are deleted,

the index will be repopulated to ensure there are at least 500 items in it).

This is just the tip of the iceberg. Many advanced utilities are included for even more sophisticated use, including **shared locks** to deal with distributed computations on shared memory, **simulated transactions in Memcached** (which obviates the need for locks in most cases), high-performance **mocks for your tests**, and in process-caches to minimize network operations during a single request-response cycle.

A version of this code is in production use at Twitter and is one part of the reason Twitter's uptime has improved so much over the last several months. This is real, pragmatic, unmagical, production-ready code that can be a big part of your Rails scaling strategy. It is designed with massive datasets and real-world operational challenges in mind. And it's almost effortless to use, since it requires no changes to how you use ActiveRecord.

[Check out `Cache Money` on github.](#)

Possibly related posts: (automatically generated)

- [Good article for performance in asp.net](#)

► [View 14 Comments](#)

« [Before Write-Through Cacheing is an Essential Part of a Healthy Scaling Strategy November 24, 2008](#)

- ◦ [About](#)

The blog of Nick Kallen, Systems Architect at Twitter, Inc. Follow me on Twitter

- ◦ [Archives](#)

- [December 2008](#)
- [November 2008](#)
- [October 2008](#)
- [Uncategorized](#)

- ◦ [Etc](#)

Search

- [RSS Feed](#)

[Blog at WordPress.com](#). Theme: **DePo Masthead** by [Derek Powazek](#)

⌵