

# Getting Creative With Memcached and AR-extensions

Daniel Lockhart, Newzwag

Nov 19, 2008

# Our Situation:

- Using conventional Memcached techniques (caching AR queries, rendered views, etc..), we were able to eliminate almost all of the reads from the database.
- Writes were still still taking quite a bit of time, as up to four separate tables were being written to each time a user answered a question.

# Abstract Solution:

- Cache the new information that needs to be written to the database and periodically write all of it at once.

# Our Requirements:

- Ability to store both new records and updates to existing records in Memcached.
- Thread-safe - needs to be able to handle simultaneous reading/writing of data.
- Ability to retrieve all of the new data periodically to commit to the database.
- No stale data.

# Memcached

## Limitations:

- No data redundancy - if Memcached goes down or runs out of space, new records and updates will be lost.
- No built in locking/mutex type capability (other than using set/add)
- No ability to iterate over data or determine what keys have been stored.

# Solution:

# Data Redundancy

- Ignore it! If we commit changes to the database frequently, we can limit the damage from lost data.
- **The minimal risk of lost data is acceptable for our project.**

# Solution:

## No locking/mutex

- There is one aspect of Memcached that is atomic/thread-safe: incr and decr
- This feature can be leveraged in creative ways to overcome our other limitations.

# Using Incr and Decr

- Some of our data updates are simple counters: how many people have answered a question, how many people chose each answer, etc.
- For this type of data, we can use incr and decr directly.

# Incr and Decr for Question Stats

- Each attribute gets its own key:  
answer\_count\_27 will be the total number of answers for a question with ID 27, while correct\_count\_27 would be the count of correct answers.
- Example:  
CACHE.incr("answer\_count\_27", 1)  
CACHE.incr("correct\_count\_27", 1) if answer.is\_correct?

# Incr and Decr for Question Stats

- Since we know all of the question IDs, we can easily periodically retrieve all of the keys to write the counters to the database. (Bulk DB updating will be explained soon)

```
Question.find(:all).map {|x| "correct_count_#{x.id}"}
```

Stretching the use of incr and decr:

# UserActions

- Our UserAction model stores information about each action a user takes - a new record is created for every request.
- None of the attributes lend themselves to be counted with incr and decr.

# Caching a UserAction

- When rails starts, initialize a counter and store the starting value as the lower bounds.

```
CACHE.add("ua_counter", "0", 0, true)  
Cache.put("ua_lower_bounds", 0)
```

# Caching a UserAction

- When a UserAction needs to be created, call `incr` on the counter. Since this call is atomic, each request will be guaranteed to return a unique value.

```
ua_number = CACHE.incr("ua_counter")
```

# Caching a UserAction

- Store the marshaled UserAction object with a key created from the counter value (i.e. ua\_2, ua\_3, etc..)

```
Cache.add("ua_{ua_number}", ua)
```

# Writing to the database

- Call `incr` on the counter to get an upper bounds.
- Using this number and our lower bounds number we recorded when we started rails, we can reconstruct an array of all of the keys that store the individual `UserAction` objects.

```
keys = []  
(lower_bounds...upper_bounds).each do |x|  
  keys << "ua_#{x}"  
end
```

# Writing to the database

- Use `Cache.get_multi` to retrieve all of the `UserAction` objects.

```
user_action_objects = CACHE.get_multi(keys)
```

# Writing to the database

- Store previous upper bounds as our new lower bounds for the next time we import.

```
Cache.put("ua_lower_bounds", upper_bounds)
```

# Ar-Extensions

- Ar-Extensions - the best RoR plugin since sliced\_bread.rb
- Bulk importing - commit multiple records in a single database call.
- Very simple syntax:

```
UserAction.import user_action_objects, {:timestamp => false, :validate => false}|
```

# Updating Existing Records with AR-Extensions

- Using AR-extensions with MySQL allows you to update multiple records in a single call.
- This is accomplished by utilizing the MySQL feature **ON DUPLICATE KEY UPDATE**.

# Updating User Records

- Each user record includes columns for a total score, number of questions answered, longest streak, etc.
- These attributes are cached in Memcached. When a user record is retrieved from the DB, the object attributes are updated from the cache. When the attributes are updated, they are written to the cache instead of the DB.

# Update Example

```
User.import [:id, :score, :correct_count], [[1,500,6], [2,300,5]],  
:or_duplicate_key_update => [:score, :correct_count]
```

This word was not found in the spelling dictionary.

# Flaws

- No redundancy of the data.
- You can never bring a Memcached server down or add a new one to the pool while people are playing.
- If the background updating process stops for any reason, updates do not occur and there is a possibility that the Memcached server could fill up.

# Possible Solutions

- **MirroredMemcached:** Write the same data to two or more Memcached server clusters. This would slow the updates down slightly, but would still be an improvement over writing directly to the DB.
- A consistent hashing algorithm might mitigate the damage of losing a Memcached server instance.

# Is it worth it?

- Any attempt to offload database writes into a single query(that I can think of) will be at some risk of data loss.
- If the data is stored in memory, it can be lost; if it is stored on disk, it will be slow to update.
- Each app has to strike its own balance between speed and data protection.