

---

# MongoDB Use Cases

*Release 2.0.5*

## MongoDB Documentation Project

May 15, 2012

### Contents

<b>1</b>	<b>Creating a Role-Playing Game</b>	<b>ii</b>
1.1	Overview . . . . .	ii
	Problem . . . . .	ii
	Solution . . . . .	ii
	Schema . . . . .	iii
1.2	Operations . . . . .	iv
	Loading Character Data . . . . .	v
	Displaying Armor and Weapon Data . . . . .	v
	Displaying Character Attributes, Inventory, and Room Information . . . . .	vii
	Picking Up an Item From a Room . . . . .	vii
	Removing an Item from a Container . . . . .	viii
	Moving the Character to a Different Room . . . . .	ix
	Buying an Item . . . . .	ix
1.3	Sharding . . . . .	x
<b>2</b>	<b>Serving and Tracking Online Advertisements</b>	<b>x</b>
2.1	Overview . . . . .	x
	Problem . . . . .	x
	Solution . . . . .	x
2.2	Serving Basic Ads . . . . .	x
	Schema . . . . .	xi
	Choosing an Ad to Serve . . . . .	xi
	Making an Ad Campaign Inactive . . . . .	xii
	Sharding . . . . .	xii
2.3	Adding Frequency Capping . . . . .	xii
	Schema . . . . .	xiii
	Choosing an Ad to Serve . . . . .	xiii
	Sharding . . . . .	xiv
2.4	Keyword Targeting . . . . .	xiv
	Schema . . . . .	xv
	Choosing a Group of Ads to Serve . . . . .	xv
<b>3</b>	<b>Storing Updates and Profiles for Social Networking Sites</b>	<b>xvi</b>
3.1	Overview . . . . .	xvi
	Problem . . . . .	xvi
	Solution . . . . .	xvi

	Schema . . . . .	xvii
3.2	Operations . . . . .	xx
	Viewing a News Feed or Wall Posts . . . . .	xx
	Commenting on a Post . . . . .	xxi
	Creating a New Post . . . . .	xxiii
	Maintaining the Social Graph . . . . .	xxiv
3.3	Sharding . . . . .	xxv

---

# 1 Creating a Role-Playing Game

## 1.1 Overview

This document outlines the basic patterns and principles for using MongoDB as a persistent storage engine for an online role-playing game. It contains an example data schema and an overview of basic operations to provide basic game mechanics.

### Problem

In designing an online game, there is a need to store various data about the player's character. Some of the attributes might include:

**Character attributes** These might include intrinsic characteristics such as strength, dexterity, charisma, etc., as well as variable characteristics such as health, mana (if your game includes magic), etc.

**Character inventory** If your game includes the ability for the player to carry around objects, you will need to keep track of the items carried.

**Character location** If your game allows the player to move their character from one location to another, the character needs to track this.

In addition, you need to store all this data for large numbers of players who might be playing the game simultaneously, and this data needs to be both readable and writeable with minimal latency in order to ensure responsiveness during gameplay.

In addition to the above data, you also need to store data for:

**Items** Includes in-game objects that characters might interact with such as weapons, armor, and treasure.

**Locations** Includes in-game settings where the characters and objects may exist during game play. Locations may include: rooms, halls, market places, buildings, and so forth.

Flexibility is a key feature for a game system. Particularly in early releases of a game, you may wish to change game mechanics as you receive feedback from your players. As you implement these changes, you need to be able to migrate your persistent data from one format to another with minimal or no downtime.

### Solution

The solution presented in this case study assumes that the read and write performance are equally important and must be accessible with minimal latency.

## Schema

Ultimately, the particulars of your schema depends on the particular design of your game. When designing your schema, you should attempt to encapsulate all the commonly used data into a small number of objects in order to minimize the number of queries to the database and the number of seeks in a query. Encapsulating all player state into a `character` collection, item data into an `item` collection, and location data into a `location` collection satisfies both these criteria.

## Characters

In a role-playing game, then, a typical character state document might look like the following:

```
{
  _id: ObjectId('...'),
  name: 'Tim',
  character: {
    intrinsics: {
      strength: 10,
      dexterity: 16,
      intelligence: 17,
      charisma: 8 },
    class: 'mage',
    health: 212,
    mana: 152
  },
  location: {
    id: 'maze-1',
    description: 'a maze of twisty little passages...',
    exits: {n:'maze-2', s:'maze-1', e:'maze-3'},
    players: [
      { id:ObjectId('...'), name:'grue' },
      { id:ObjectId('...'), name:'Tim' }
    ],
    inventory: [
      { qty:1, id:ObjectId('...'), name:'scroll of cause fear' }]
  },
  gold: 523,
  armor: [
    { id:ObjectId('...'), region:'head'},
    { id:ObjectId('...'), region:'body'},
    { id:ObjectId('...'), region:'feet'}],
  weapons: [ {id:ObjectId('...'), hand:'both' } ],
  inventory: [
    { qty:1, id:ObjectId('...'), name:'backpack', inventory: [
      { qty:4, id:ObjectId('...'), name: 'potion of healing'},
      { qty:1, id:ObjectId('...'), name: 'scroll of magic mapping'},
      { qty:2, id:ObjectId('...'), name: 'c-rations' } ]},
    { qty:1, id:ObjectId('...'), name:"wizard's hat", bonus:3},
    { qty:1, id:ObjectId('...'), name:"wizard's robe", bonus:0},
    { qty:1, id:ObjectId('...'), name:"old boots", bonus:0},
    { qty:1, id:ObjectId('...'), name:"quarterstaff", bonus:2} ]
}
```

There are a few things to note about this document:

1. Encapsulate information that relates to the character's location in the `location` attribute within the character state document. This allows the game system to render the room without making a second query to get room

information.

2. The `armor` and `weapons` attributes contain little information about the actual items worn or carried. The `inventory` holds this data, and because `inventory` already exists in this document, there is no need to duplicate information within a document.
3. `inventory` contains all information details required for rendering the items in the character's possession. This includes weapons, armor, as well as any enchantments (i.e. `bonus` values) as well as the `quantity`. By embedding this data in the character record means you don't have to perform a separate query to fetch item details necessary for display.

## Items

The item schema should include all details about the items available in the game:

```
{
  _id: ObjectId('...'),
  name: 'backpack',
  bonus: null,
  inventory: [
    { qty:4, id:ObjectId('...'), name: 'potion of healing'},
    { qty:1, id:ObjectId('...'), name: 'scroll of magic mapping'},
    { qty:2, id:ObjectId('...'), name: 'c-rations' } ]],
  weight: 12,
  price: 160,
  ...
}
```

These documents will contain roughly identical information as the `inventory` attribute of the character documents. Additionally, these documents will store data that the game will need to access only sporadically, such as `weight` and `price`.

## Location

The location schema specifies the state of the world in the game:

```
{
  id: 'maze-1',
  description: 'a maze of twisty little passages...',
  exits: {n:'maze-2', s:'maze-1', e:'maze-3'},
  players: [
    { id:ObjectId('...'), name:'grue' },
    { id:ObjectId('...'), name:'Tim' } ],
  inventory: [
    { qty:1, id:ObjectId('...'), name:'scroll of cause fear' } ],
}
```

The `location` field stores the same information as the `location` attribute of the character documents. The application will use `location` as the system of record for interactions between multiple characters or between characters and non-inventory items.

## 1.2 Operations

For this online gaming system, because all state embedded in single documents for `character`, `item`, and `location` most operations are relatively simple. The majority of queries revolve around checking the character

state using a query on the `_id` field and extracting relevant information for display. Additionally, the game application will frequently update attributes about the character. This section describes procedures for performing these queries, extractions, and updates.

As a general rule, in particular you should try *not* to load the `location` or `item` documents except when absolutely necessary.

The examples that follow use the Python programming language and the [PyMongo driver](#) for MongoDB, but you can implement this system using any language you choose.

## Loading Character Data

The most basic operation in this system is loading the character state.

Use the following query to load the `character` document from MongoDB:

```
>>> character = db.characters.find_one({'_id': character_id})
```

In this case, the default index that MongoDB supplies on the `_id` field is sufficient for good performance of this query.

## Displaying Armor and Weapon Data

In order to save space, the `character` schema described above stores item details only in the `inventory` attribute, storing *ObjectId references* in other locations. To display these item details, as on a character summary window, you need to merge the information from the `armor` and `weapons` attributes with information from the `inventory` attribute.

Suppose, for instance, that your code displays armor data using the following Jinja2 template:

```
<div>
  <h2>Armor</h2>
  <dl>
    {% if value.head %}
      <dt>Helmet</dt>
      <dd>{{value.head[0].description}}</dd>
    {% endif %}
    {% if value.hands %}
      <dt>Gloves</dt>
      <dd>{{value.hands[0].description}}</dd>
    {% endif %}
    {% if value.feet %}
      <dt>Boots</dt>
      <dd>{{value.feet[0].description}}</dd>
    {% endif %}
    {% if value.body %}
      <dt>Body Armor</dt>
      <dd><ul>{% for piece in value.body %}
        <li>piece.description</li>
      {% endfor %}</ul></dd>
    {% endif %}
  </dl>
</div>
```

In this case, you want the `description` fields above to render as text, such as “+3 wizard’s hat.” The context passed to the template above, then, would be of the following form:

```
{
    "head": [ { "id":..., "description": "+3 wizard's hat" } ],
    "hands": [],
    "feet": [ { "id":..., "description": "old boots" } ],
    "body": [ { "id":..., "description": "wizard's robe" } ],
}
```

To build this structure, use the following helper functions:

```
def get_item_index(inventory):
    """Given an inventory attribute, recursively build up an item
    index (including all items contained within other items)
    """

    result = {}
    for item in inventory:
        result[item['_id']] = item
        if 'inventory' in item:
            result.update(get_item_index(item['inventory']))
    return result

def describe_item(item):
    """Add a 'description' field to the given item"""

    result = dict(item)
    if item['bonus']:
        description = '%+d %s' % (item['bonus'], item['name'])
    else:
        description = item['name']
    result['description'] = description
    return result

def get_armor_for_display(character, item_index):
    """Given a character document, return an 'armor' value
    suitable for display"""

    result = dict(head=[], hands=[], feet=[], body=[])
    for piece in character['armor']:
        item = describe_item(item_index[piece['id']])
        result[piece['region']].append(item)
    return result
```

To display the armor, then, you would use the following code:

```
>>> item_index = get_item_index(
...     character['inventory'] + character['location']['inventory'])
>>> armor = get_armor_for_display(character, item_index)
```

This operation builds an index for the items the character is actually carrying in inventory in addition to the items that the player might interact with in the room. Similarly, in order to display the weapon information, you need to build a structure that resembles the following:

```
{
    "left": None,
    "right": None,
    "both": { "description": "+2 quarterstaff" }
}
```

The helper function is similar to that for `get_armor_for_display`:

```
def get_weapons_for_display(character, item_index):
    '''Given a character document, return a 'weapons' value
    suitable for display'''

    result = dict(left=None, right=None, both=None)
    for piece in character['weapons']:
        item = describe_item(item_index[piece['id']])
        result[piece['hand']] = item
    return result
```

To actually display weapons, then, you would use the following code:

```
>>> armor = get_weapons_for_display(character, item_index)
```

## Displaying Character Attributes, Inventory, and Room Information

To display information about the character's attributes, inventory, and surroundings, you also need to extract fields from the character state. In this case, however, the schema defined above keeps all the relevant information for display embedded in those sections of the document. The code for extracting this data, then, is the following:

```
>>> attributes = character['character']
>>> inventory = character['inventory']
>>> room_data = character['location']
```

## Picking Up an Item From a Room

Use the following procedure to update the character state and the global location state when the player picks up an item from the room and adds it to their inventory:

```
def pick_up_item(character, item_index, item_id):
    '''Transfer an item from the current room to the character's inventory'''

    item = item_index[item_id]
    character['inventory'].append(item)
    db.character.update(
        { '_id': character['_id'] },
        { '$push': { 'inventory': item },
          '$pull': { 'location.inventory': { '_id': item['id'] } } })
    db.location.update(
        { '_id': character['location']['id'] },
        { '$pull': { 'inventory': { 'id': item_id } } })
```

While the above code may be for a single-player game, if you allow multiple players, or non-player characters, to pick up items at the same time, you may introduce a problem when two characters attempt to pick up an item simultaneously. To guard against that, use the `location` collection to “break ties.” In this case, the code is now the following:

```
def pick_up_item(character, item_index, item_id):
    '''Transfer an item from the current room to the character's
    inventory'''

    item = item_index[item_id]
    character['inventory'].append(item)
    result = db.location.update(
        { '_id': character['location']['id'],
          'inventory.id': item_id },
```

```

        { '$pull': { 'inventory': { 'id': item_id } } },
        safe=True)
    if not result['updatedExisting']:
        raise Conflict()
    db.character.update(
        { '_id': character['_id'] },
        { '$push': { 'inventory': item },
          '$pull': { 'location': { '_id': item['id'] } } })

```

By ensuring that the item is present before removing it from the room in the update call above, you guarantee that only one player/non-player characters can pick up the item.

## Removing an Item from a Container

In the game described here, the backpack item can contain other items. You might further suppose that some other items may be similarly hierarchical (e.g. a chest in a room). Suppose that the player wishes to move an item from one of these “containers” into their active inventory as a prelude to using it. In this case, you need to update both the character state and the item state:

```

def move_to_active_inventory(character, item_index, container_id, item_id):
    '''Transfer an item from the given container to the character's active
    inventory'''

    result = db.item.update(
        { '_id': container_id,
          'inventory.id': item_id },
        { '$pull': { 'inventory': { 'id': item_id } } },
        safe=True)
    if not result['updatedExisting']:
        raise Conflict()
    item = item_index[item_id]
    container = item_index[container_id]
    character['inventory'].append(item)
    container['inventory'] = [
        item for item in container['inventory']
        if item['_id'] != item_id ]
    db.character.update(
        { '_id': character['_id'] },
        { '$push': { 'inventory': item } } )
    db.character.update(
        { '_id': character['_id'], 'inventory.id': container_id },
        { '$pull': { 'inventory.$.inventory': { 'id': item_id } } } )

```

In this code, you:

- Ensure that the update is valid, in this case that the item is actually contained within the container. Abort with an error if the operation is invalid.
- Update the in-memory `character` document's inventory, adding the item.
- Update the in-memory `container` document's inventory, removing the item.
- Update the `character` document.
- In the case that the character is moving an item from a container *in his own inventory*, update the character's inventory representation of the container.



## Moving the Character to a Different Room

Use the following operation to update the character's state with a new location:

```
def move(character, direction):
    '''Move the character to a new location'''

    # Remove character from current location
    db.location.update(
        { '_id': character['location']['id'] },
        { '$pull': { 'players': { 'id': character['_id'] } } })
    # Add character to new location, retrieve new location data
    new_location = db.location.find_and_modify(
        { '_id': character['location']['exits'][direction] },
        { '$push': { 'players': {
            'id': character['_id'],
            'name': character['name'] } } },
        new=True)
    character['location'] = new_location
    db.character.update(
        { '_id': character['_id'] },
        { '$set': { 'location': new_location } })
```

This operation updates the old room, the new room, and the character document to reflect the new state.

## Buying an Item

For a character to buy an item, you need to: add the item to the character's inventory, decrement the character's gold, increment the shopkeeper's gold, and update the room:

```
def buy(character, shopkeeper, item_id):
    '''Pick up an item, add to the character's inventory, and transfer
    payment to the shopkeeper'''

    price = db.item.find_one({'_id': item_id}, {'price':1})['price']
    result = db.character.update(
        { '_id': character['_id'],
          'gold': { '$gte': price } },
        { '$inc': { 'gold': -price } },
        safe=True )
    if not result['updatedExisting']:
        raise InsufficientFunds()
    try:
        pick_up_item(character, item_id)
    except:
        # Add the gold back to the character
        result = db.character.update(
            { '_id': character['_id'] },
            { '$inc': { 'gold': price } } )
        raise
    character['gold'] -= price
    db.character.update(
        { '_id': shopkeeper['_id'] },
        { '$inc': { 'gold': price } } )
```

This code ensures that the character has sufficient gold to pay for the item using the `updatedExisting` check from the *picking up items* operation. This handles the potential race condition if an item picked up, and can “roll back” the

removal of gold from the character's wallet if the item becomes unavailable during the transaction.

## 1.3 Sharding

If your system needs to scale beyond a single MongoDB instance node, you will want to use a *shard cluster*, which takes advantage of MongoDB's *sharding* functionality.

### See Also:

“/faq/sharding” and the “[Sharding](#)” wiki page.

Because all items are always retrieved by `_id`, the choice *shard key* is straightforward. To shard the `character` and `location` collections, the commands would be the following:

```
>>> db.command('shardcollection', 'character', {
...     'key': { '_id': 1 } })
{ "collectionsharded" : "character", "ok" : 1 }
>>> db.command('shardcollection', 'location', {
...     'key': { '_id': 1 } })
{ "collectionsharded" : "location", "ok" : 1 }
```

## 2 Serving and Tracking Online Advertisements

### 2.1 Overview

This document outlines basic patterns and principles for using MongoDB as a persistent storage engine for an online advertising network. In particular, this document focuses on the task of deciding *which* ad to serve when a user visits a particular site.

### Problem

You want to create an advertising network to serve ads to online media sites. As part of this service, you also want to track which ads are available, and decide on a particular to serve to a particular zone.

### Solution

This case study describes a basic advertising service and then refines this application to support more advanced ad targeting. The key performance requirements for this solution is the latency between receiving a request and returning the (targeted) ad for display.

The examples that follow use the Python programming language and the [PyMongo driver](#) for MongoDB, but you can implement this system using any language you choose.

### 2.2 Serving Basic Ads

A basic ad serving algorithm consists of the following steps:

1. The network receives a request for an ad, specifying at a minimum the `site_id` and `zone_id`.
2. The network consults its inventory of ads available to display and chooses an ad based on various business rules.
3. The network returns the actual ad for display, possibly recording the decision.

This design uses the `site_id` and `zone_id` with the ad request, as well as information stored in the ad inventory collection, to make the ad targeting decisions. This design also provides flexibility for additional functionality later.

## Schema

The schema for storing available ads consists of a single collection, `ad.zone`:

```
{
  _id: ObjectId(...),
  site_id: 'cnn',
  zone_id: 'banner',
  ads: [
    { campaign_id: 'mercedes:c201204_sclass_4',
      ad_unit_id: 'banner23a',
      ecpm: 250 },
    { campaign_id: 'mercedes:c201204_sclass_4',
      ad_unit_id: 'banner23b',
      ecpm: 250 },
    { campaign_id: 'bmw:c201204_eclass_1',
      ad_unit_id: 'banner12',
      ecpm: 200 },
    ... ]
}
```

For each (site, zone) combination you'll store a list of ads, sorted by their `ecpm` values.

## Choosing an Ad to Serve

### Querying

The query that the application uses to choose an add to serve selects a compatible ad and sorts by the advertiser's `ecpm` bid in order to maximize the ad network's profits. This query resembles the following:

```
from itertools import groupby
from random import choice

def choose_ad(site_id, zone_id):
    site = db.ad.zone.find_one({
        'site_id': site_id, 'zone_id': zone_id})
    if site is None: return None
    if len(site['ads']) == 0: return None
    ecpm_groups = groupby(site['ads'], key=lambda ad:ad['ecpm'])
    ecpm, ad_group = ecpm_groups.next()
    return choice(list(ad_group))
```

### Indexing

To execute the ad choice with the lowest latency possible, create a compound index on (site\_id, zone\_id):

```
>>> db.ad.zone.ensure_index([
...     ('site_id', 1),
...     ('zone_id', 1) ])
```

## Making an Ad Campaign Inactive

### Updating

Ad systems must adjust automatically to the requirements of an ad campaign: if the campaign has reached its end date or exhausted its budget the system must automatically cease serving those ads. The following operation implements this feature:

```
def deactivate_campaign(campaign_id):
    db.ad.zone.update(
        { 'ads.campaign_id': campaign_id },
        { '$pull': { 'ads', { 'campaign_id': campaign_id } } },
        multi=True)
```

The update statement selects only those ad zones that have available ads from the given `campaign_id` and then uses the `$pull` operator to remove them from rotation.

### Indexing

To support the multi-update, you should maintain an index on the `ads.campaign_id` field:

```
>>> db.ad.zone.ensure_index('ads.campaign_id')
```

### Sharding

To scale beyond the capacity of a single replica set, you will need to shard the `ad.zone` collection. To maintain the lowest possible latency in the ad selection operation, choose a the *shard key* that allows MongoDB to route the `ad.zone` query to a single shard or limited number of shards. Using the `(site_id, zone_id)` field combination as a shard key fulfills this requirement:

```
>>> db.command('shardcollection', 'ad.zone', {
...     'key': {'site_id': 1, 'zone_id': 1} })
{ "collectionsharded": "ad.zone", "ok": 1 }
```

## 2.3 Adding Frequency Capping

One problem with the logic described in the basic application design is that that it will tend to display the same ad repeatedly until it exhausts the campaign's budget. To mitigate this, the system may want to limit the frequency that it presents a single user with a specific ad. This “frequency capping” is an example of user profile targeting in advertising.

To provide frequency capping, or any type of user targeting, the ad system must maintain a profile for each visitor, typically implemented as a cookie in the user's browser. The system uses this cookie, effectively a `user_id`, when logging impressions, clicks, conversions, etc., as well as the ad serving decision. This section focuses on how that profile data impacts the ad serving decision.

### Schema

In order to use the user profile data, you need to store it. In this case, it's stored in a collection `ad.user`:

```
{
  _id: 'cookie_value',
  advertisers: {
    mercedes: {
      impressions: [
        { date: ISODateTime(...),
          campaign: 'c201204_sclass_4',
          ad_unit_id: 'banner23a',
          site_id: 'cnn',
          zone_id: 'banner' } },
        ... ],
      clicks: [
        { date: ISODateTime(...),
          campaign: 'c201204_sclass_4',
          ad_unit_id: 'banner23a',
          site_id: 'cnn',
          zone_id: 'banner' } },
        ... ],
      bmw: [ ... ],
      ...
    }
  }
}
```

This schema:

- Segments profile information by advertiser. Typically advertising data is sensitive information that the service can't share between advertisers.
- Embeds all data in a single profile document. When you need to query this data, the application won't know which advertiser's ads it's showing, so it makes sense to store all data in a single document.
- Groups event information by type within an advertiser, and sorts by timestamp. This facilitates rapid lookups of a stream of a particular type of event.

## Choosing an Ad to Serve

### Querying

Use the following query to choose an ad to serve. This operation must iterate through ads in order of “desireability” and then select the “best” ad that also satisfies the advertiser’s targeting rules. In this example the frequency cap is the targeting rule.

```
from itertools import groupby
from random import shuffle
from datetime import datetime, timedelta

def choose_ad(site_id, zone_id, user_id):
    site = db.ad.zone.find_one({
        'site_id': site_id, 'zone_id': zone_id})
    if site is None or len(site['ads']) == 0: return None
    ads = ad_iterator(site['ads'])
    user = db.ad.user.find_one({'user_id': user_id})
    if user is None:
        # any ad is acceptable for an unknown user
        return ads.next()
    for ad in ads:
        advertiser_id = ad['campaign_id'].split(':', 1)[0]
```

```

        if ad_is_acceptable(ad, user[advertiser_id]):
            return ad
    return None

def ad_iterator(ads):
    '''Find available ads, sorted by ecpm, with random sort for ties'''
    ecpm_groups = groupby(ads, key=lambda ad:ad['ecpm'])
    for ecpm, ad_group in ecpm_groups:
        ad_group = list(ad_group)
        shuffle(ad_group)
        for ad in ad_group: yield ad

def ad_is_acceptable(ad, profile):
    '''Returns False if the user has seen the ad today'''
    threshold = datetime.utcnow() - timedelta(days=1)
    for event in reversed(profile['impressions']):
        if event['timestamp'] < threshold: break
        if event['detail']['ad_unit_id'] == ad['ad_unit_id']:
            return False
    return True

```

The `chose_ad()` function provides the framework for your ad selection process. The query fetches site first, and then passes it to the `ad_iterator()` function. This yields ads in order of desirability. Then, `ad_is_acceptable()` checks each add to determine if it meets the advertiser's rules.

The `ad_is_acceptable()` function then iterates over all impressions in the user's profile, from most recent to oldest, within a certain threshold time period, which is 1 day in the sample above. If the same `ad_unit_id` appears in the impression stream, the function rejects the ad. Otherwise the service shows the first acceptable ad to the user.

## Indexing

To retrieve the user profile with the lowest latency possible, this operation requires an index on the `_id` field, which MongoDB supplies by default.

## Sharding

When sharding the `ad.user` collection, choosing the `_id` field as a *shard key* allows MongoDB to route queries and updates to the profile:

```

>>> db.command('shardcollection', 'ad.user', {
...     'key': {'_id': 1 } })
{ "collectionsharded": "ad.user", "ok": 1 }

```

## 2.4 Keyword Targeting

Where frequency is an example of user profile targeting, you may want the system to target ads so that the user receives ads relevant to the page they're viewing. For example, you might want to target ads based on a search query. In this case, the system sends a list keywords to the `choose_ad()` function with the `site_id`, `zone_id`, and `user_id`.

## Schema

To choose relevant ads, you will expand the `ad.zone` collection to store keywords for each ad:

```
{
  _id: ObjectId(...),
  site_id: 'cnn',
  zone_id: 'search',
  ads: [
    { campaign_id: 'mercedes:c201204_sclass_4',
      ad_unit_id: 'search1',
      keywords: [ 'car', 'luxury', 'style' ],
      ecpm: 250 },
    { campaign_id: 'mercedes:c201204_sclass_4',
      ad_unit_id: 'search2',
      keywords: [ 'car', 'luxury', 'style' ],
      ecpm: 250 },
    { campaign_id: 'bmw:c201204_eiclass_1',
      ad_unit_id: 'search1',
      keywords: [ 'car', 'performance' ],
      ecpm: 200 },
    ... ]
}
```

## Choosing a Group of Ads to Serve

The system will then choose a number of ads that match the keywords used in the search. The following `choose_ads()` and `ad_iterator()` implementations will return and iterate over ads in descending order of preference:

```
def choose_ads(site_id, zone_id, user_id, keywords):
    site = db.ad.zone.find_one({
        'site_id': site_id, 'zone_id': zone_id})
    if site is None: return []
    ads = ad_iterator(site['ads'], keywords)
    user = db.ad.user.find_one({'user_id': user_id})
    if user is None: return ads
    advertiser_ids = (
        ad['campaign_id'].split(':', 1)[0]
        for ad in ads )
    return (
        ad for ad, advertiser_id in izip(
            ads, advertiser_ids)
        if ad_is_acceptable(ad, user[advertiser_id]) )

def ad_iterator(ads, keywords):
    """Find available ads, sorted by score, with random sort for
    ties"""

    keywords = set(keywords)
    scored_ads = [
        (ad_score(ad, keywords), ad)
        for ad in ads ]
    score_groups = groupby(
        sorted(scored_ads), key=lambda score, ad: score)
    for score, ad_group in score_groups:
        ad_group = list(ad_group)
```

```

        shuffle(ad_group)
        for ad in ad_group: yield ad

def ad_score(ad, keywords):
    '''Compute a desirability score based on the ad ecpm and
    keywords'''

    matching = set(ad['keywords']).intersection(keywords)
    return ad['ecpm'] * math.log(
        1.1 + len(matching))

# ad_is_acceptable
def ad_is_acceptable(ad, profile):
    '''Returns False if the user has seen the ad today'''
    threshold = datetime.utcnow() - timedelta(days=1)
    for event in reversed(profile['impressions']):
        if event['timestamp'] < threshold: break
        if event['detail']['ad_unit_id'] == ad['ad_unit_id']:
            return False
    return True

```

With these implementations, ads you must sort ads according to some score. Here, `ad_score()` computes this value based on a combination of the `ecpm` value for the ad and the number of keywords matched. In more sophisticated implementations you may choose to customize the value of some keywords, but this is beyond the scope of this document. Because the ad service must sort all ads at display time, you may find performance issues if you if there are a large number of ads competing for the same display slot.

## 3 Storing Updates and Profiles for Social Networking Sites

### 3.1 Overview

This document outlines the basic patterns and principles for using MongoDB as a persistent storage engine for a social networking website. In particular, this case study focuses on the task of storing and displaying user updates.

#### Problem

You want to create an social network where users can create profiles with personal or professional information. Then, users will be able to create posts and updates, of various types, that the “walls” or news feeds of their friends.

#### Solution

This document assumes that relationships between users in the social network are a *directed* social graph where users can choose whether or not to follow another user who follows them. Additionally, the user may group the people they follow into “circles” to selectively limit the distribution of their information and help them control their privacy.

The solution below attempts to minimize the number of documents that the application must load to display any given page, even at the expense of complicating update operations.

The type of data that you will store and host depends on the type of community and site you’re building. These decisions are largely beyond the scope of this document. However, consider the following questions in light of the following issues:

**What data will you include in a user profile?** For a non-professional community and site, this may include age, interests, relationship status, etc. or more resume-like data for more “business-oriented” communities/networks.



**What type of updates will you allow?** The kinds of content that users can submit and include also depends on the kind of community and social networking site you’re building. You may want to allow users to post status updates, photos, links, location check-ins, and polls, or you may wish to restrict your users to just links and status updates.

## Schema

To model the data for the social networking site, you will use two main “independent” collections and three “dependent” collections to store user profile data and posts.

### Independent Collections

The first collection, `social.user`, stores the social graph information for a given user along with the user’s profile data. It uses a schema in the following form:

```
{
  _id: 'T4Y...AC', // base64-encoded ObjectId
  name: 'Rick',
  profile: { ... age, location, interests, etc. ... },
  followers: {
    "T4Y...AD": { name: 'Jared', circles: [ 'python', 'authors' ] },
    "T4Y...AF": { name: 'Bernie', circles: [ 'python' ] },
    "T4Y...AI": { name: 'Meghan', circles: [ 'python', 'speakers' ] },
    ...
  },
  circles: {
    "10gen": {
      "T4Y...AD": { name: 'Jared' },
      "T4Y...AE": { name: 'Max' },
      "T4Y...AF": { name: 'Bernie' },
      "T4Y...AH": { name: 'Paul' },
      ... },
    ... }
  },
  blocked: ['gh1...0d']
}
```

Consider the following features of this schema:

- Rather than using a “raw” `ObjectId` for your `_id` field, you’ll use a base64-encoded version. This allows you to use `_id` values as keys in subdocuments, which both reduces the memory footprint of these subdocuments as well as speeding up some operations.
- The schema represents the social graph bidirectionally in the `followers` and `circles` sub-documents. While this is redundant, representing the connection in both directions makes it easier to display the user’s followers on the profile page and helps in propagating posts to other users, as shown below.
- In addition to the normal “positive” social graph, this schema also stores a “block list” that contains an array of user ids for posters whose posts a user has prohibited from appearing on *their* wall or news feed.
- The schema stores all of the user’s profile data in the `profile` sub-document, so that you may iterate the schema can evolve as necessary without affecting other fields.

Posts, of various types, reside in the `social.post` collection:

```
{
  _id: ObjectId(...),
```

```

by: { id: "T4Y...AE", name: 'Max' },
circles: [ '*public*' ],
type: 'status',
ts: ISODateTime(...),
detail: {
  text: 'Loving MongoDB' },
comments: [
  { by: { id:"T4Y...AG", name: 'Dwight' },
    ts: ISODateTime(...),
    text: 'Right on!' },
    ... all comments listed ... ]
}

```

The `by` field holds minimal required author information, while the `type` field stores the post type and the `detail` array contains specific information for the type. The `ts` field stores the post's. `comments` stores an array of documents with comments that embeds all comments on a post as a time-sorted flat array. For a more in-depth exploration of the other approaches to storing comments, please see the `/use-cases/storing-comments` document.

This schema includes the following considerations:

- Truncate all author information and store enough data in each `by` property to render the author name and a link to the author's profile. If users want more detail on a particular author, the application can fetch this information separately. Storing only minimally required information helps keep the document small and makes most operations fast.
- Control post visibility with the `circles` property: any user that is part of a circle listed in the array can view the post. Special values like `"*public*"` and `"*circles*"` allow the user to share a post with the whole world or with any users in any of the posting user's circles, respectively.
- Different types of posts may contain different types of data in the `detail` field. Isolating the subset of fields that can change into a sub-document is good practice, and here you would store the data for a photo post differently from the data for a status update. All the metadata, e.g. `_id`, `by`, `circles`, `type`, `ts`, and `comments`, remain the same.

## Dependent Collections

To optimize rendering content for display, this application design includes three “dependent” collections in addition to two “independent” collections. The first dependent collection, `social.wall`, provides a cache for displaying a “wall” or feed that contains posts created by or directed to a particular user. The schema for the documents in the `social.wall` collection is:

```

{
  _id: ObjectId(...),
  user_id: "T4Y...AE",
  month: '201204',
  posts: [
    { id: ObjectId(...),
      ts: ISODateTime(...),
      by: { id: "T4Y...AE", name: 'Max' },
      circles: [ '*public*' ],
      type: 'status',
      detail: { text: 'Loving MongoDB' },
      comments_shown: 3,
      comments: [
        { by: { id: "T4Y...AG", name: 'Dwight',
          ts: ISODateTime(...),
          text: 'Right on!' },

```

```

        ... only last 3 comments listed ...
    ]
},
{ id: ObjectId(...), s
  ts: ISODateTime(...),
  by: { id: "T4Y...AE", name: 'Max' },
  circles: [ '*circles*' ],
  type: 'checkin',
  detail: {
    text: 'Great office!',
    geo: [ 40.724348, -73.997308 ],
    name: '10gen Office',
    photo: 'http://....' },
  comments_shown: 1,
  comments: [
    { by: { id: "T4Y...AD", name: 'Jared' },
      ts: ISODateTime(...),
      text: 'Wrong coast!' },
    ... only last 1 comment listed ...
  ]
},
{ id: ObjectId(...),
  ts: ISODateTime(...),
  by: { id: "T4Y...g9", name: 'Rick' },
  circles: [ '10gen' ],
  type: 'status',
  detail: {
    text: 'So when do you crush Oracle?' },
  comments_shown: 2,
  comments: [
    { by: { id: "T4Y...AE", name: 'Max' },
      ts: ISODateTime(...),
      text: 'Soon... ;-)' },
    ... only last 2 comments listed ...
  ]
},
...
]
}

```

Consider the following features of this schema:

- Each post contains a limited number of comments (i.e. 3 or 4.) Limiting the number of comments minimizes the size of the document. To display more comments on a post, your application can query the `social.post` collection for full details.
- The `social.wall` collection would contain one document per month per `social.user` document. Then, the system can keep a “page” of recent posts in the initial page view, and can fetch additional views as needed.
- Once again, the `by` sub-document stores only the minimum amount of information required for display. This helps keep the document small.
- The `comments_shown` field stores the number of comments. This allows updates in the future to identify posts with more than a certain number of comments because the `$size` query operator does not allow inequality comparisons.

The second dependent collection is `social.news`, which collects posts from people the user follows. These documents duplicate information from the documents in the `social.wall` collection:

```
{
    _id: ObjectId(...),
    user_id: "T4Y...AE",
    month: '201204',
    posts: [ ... ]
}
```

## 3.2 Operations

The data model presented above optimizes for read performance at the possible expense of write performance. To As a result, you should ideally provide a queueing system for processing updates which may take longer than your desired web request latency.

The examples that follow use the Python programming language and the [PyMongo driver](#) for MongoDB, but you could easily implement this using any language of your choose.

### Viewing a News Feed or Wall Posts

#### Querying

The most common operations on this site are probably displaying a particular user's news feed and displaying a user's wall posts. Since the `social.news` and `social.wall` collections optimize for these operations, the query can be quite simple. Since these two collections share a schema, viewing the posts for a news feed or a wall are similar operations and can use the same code:

```
def get_posts(collection, user_id, month=None):
    spec = { 'user_id': viewed_user_id }
    if month is not None:
        spec['month'] = {'$lte': month}
    cur = collection.find(spec)
    cur = cur.sort('month', -1)
    for page in cur:
        for post in reversed(page['posts']):
            yield page['month'], post
```

The function `get_posts` above retrieves all the posts on a particular user's wall or news feed in reverse-chronological order. This requires special handling to achieve the reverse-chronological ordering efficiently:

- The data model sorts `posts` within a month in chronological order, so your application must reverse the order of these posts must before displaying them.
- As a users pages through their walls, you want your application to be able to avoid fetching the most recent few months from the server for every request. Th code specifies the first month to fetch in the `month` argument, passing this in as an `$lte` expression in the query.
- The generator yields the post's month for use in subsequent calls to `get_posts` in addition to yielding the post itself.

Also consider privacy configuration when selecting posts for display. To handle privacy issues, your application will need filter functions to process the posts generated by `get_posts`. The first filter determines if a user can view a given post on their wall:

```
def visible_on_own_wall(user, post):
    '''if poster is followed by user, post is visible'''
    for circle, users in user['circles'].items():
```

```

    if post['by']['id'] in users: return True
    return False

```

You might also want your application to provide an “incoming” page that contains all posts directed to a user even if the user doesn’t follow the poster, unless the user blocks the poster. See the following filter:

```

def visible_on_own_incoming(user, post):
    '''if poster is not blocked by user, post is visible'''
    return post['by']['id'] not in user['blocked']

```

When viewing a news feed or another user’s wall, use the following operation to check permissions using the value of the post’s circles field:

```

def visible_post(user, post):
    if post['circles'] == ['*public*']:
        # public posts always visible
        return True
    circles_user_is_in = set(
        user['followers'].get(post['by']['id'] []))
    if not circles_user_is_in:
        # user is not circled by poster; post is invisible
        return False
    if post['circles'] == ['*circles*']:
        # post is public to all followed users; post is visible
        return True
    for circle in post['circles']:
        if circle in circles_user_is_in:
            # User is in a circle receiving this post
            return True
    return False

```

## Indexing

To support quick retrieval of pages, you’ll need an index on (user\_id, month) in both the social.news and social.wall collections. Use the following operation at the Python/PyMongo shell.

```

>>> for collection in (db.social.news, db.social.wall):
...     collection.ensure_index([
...         ('user_id', 1),
...         ('month', -1)])

```

## Commenting on a Post

### Updating

In addition to viewing walls and news feeds, the next most common activity is creating new posts is the next most common action taken on social networks. To create a comment by user on a given post containing the given text, you’ll need to execute code similar to the following:

```

from datetime import datetime

def comment(user, post_id, text):
    ts = datetime.utcnow()
    month = ts.strftime('%Y%m')
    comment = {

```

```

        'by': { 'id': user['id'], 'name': user['name'] }
        'ts': ts,
        'text': text }
# Update the social.posts collection
db.social.post.update(
    { '_id': post_id },
    { '$push': { 'comments': comment } } )
# Update social.wall and social.news collections
db.social.wall.update(
    { 'posts.id': post_id },
    { '$push': { 'comments': comment },
      '$inc': { 'comments_shown': 1 } },
    upsert=True,
    multi=True)
db.social.news.update(
    { 'posts.id': _id },
    { '$push': { 'comments': comment },
      '$inc': { 'comments_shown': 1 } },
    upsert=True,
    multi=True)

```

---

**Note:** The presence of a couple of `multi=True` update statements means that this operation can potentially take a long time. This feature makes this function a good candidate for processing ‘out of band’ with the regular request-response flow of your application.

---

This function can result in unbounded document growth of the `social.wall` and `social.news` collections, because there is no limits on the number of comments. To compensate, run the following update statement periodically. This will truncate the number of displayed comments and control the size of these documents:

```
COMMENTS_SHOWN = 3
```

```

def truncate_extra_comments():
    db.social.news.update(
        { 'posts.comments_shown': { '$gt': COMMENTS_SHOWN } },
        { '$pop': { 'posts.$.comments': -1 },
          '$inc': { 'posts.$.comments_shown': -1 } },
        multi=True)
    db.social.wall.update(
        { 'posts.comments_shown': { '$gt': COMMENTS_SHOWN } },
        { '$pop': { 'posts.$.comments': -1 },
          '$inc': { 'posts.$.comments_shown': -1 } },
        multi=True)

```

## Indexing

To support updates to the `social.news` and `social.wall` collections, you’ll need to be able to locate both of the following types of documents:

- Documents containing a given post
- Documents containing posts displaying too many comments

This requires indexes on `posts.id` and `posts.comments_shown` fields in both collections. Create these indexes with the following form:

```
>>> for collection in (db.social.news, db.social.wall):
...     collection.ensure_index('posts.id')
...     collection.ensure_index('posts.comments_shown')
```

## Creating a New Post

### Inserting

Use the following operation to create a post and insert content in all required collections:

```
from datetime import datetime

def post(user, dest_user, type, detail, circles):
    ts = datetime.utcnow()
    month = ts.strftime('%Y%m')
    post = {
        'ts': ts,
        'by': { 'id': user['id'], name: user['name'] },
        'circles': circles,
        'type': type,
        'detail': detail,
        'comments': [] }
    # Update global post collection
    db.social.post.insert(post)
    # Copy to dest user's wall
    if user['id'] not in dest_user['blocked']:
        append_post(db.social.wall, [dest_user['id']], month, post)
    # Copy to followers' news feeds
    if circles == ['*public*']:
        dest_userids = set(user['followers'].keys())
    else:
        dest_userids = set()
        if circles == ['*circles*']:
            circles = user['circles'].keys()
        for circle in circles:
            dest_userids.update(user['circles'][circle])
    append_post(db.social.news, dest_userids, month, post)
```

The basic sequence of operations in the code above is the following:

1. The post first saved into the “system of record,” the `social.post` collection.
2. The recipient’s wall is updated with the post.
3. Updates news feeds of everyone who is “circled” in the post.

### Updating

Use the `append_post` function to update a particular wall or group of news feeds:

```
def append_post(collection, dest_userids, month, post):
    collection.update(
        { 'user_id': { '$in': sorted(dest_userids) },
          'month': month },
        { '$push': { 'posts': post } },
        multi=True)
```

## Indexing

To update the `social.wall` and `social.news` collections efficiently, you will need an index on both `user_id` and `month`. For these operations (`month, user_id`) is the optimal order of keys in the index. *However*, because your collections have an index on (`user_id, month`), which *must* be in that order to support the sort on `month`, adding a second index is unnecessary.

## Maintaining the Social Graph

### Updating

While maintaining the social graph is a relatively infrequent operation, it is absolutely essential. Use the following function to add a user, `other`, to the current user's, `self`, circles:

```
def circle_user(self, other, circle):
    circles_path = 'circles.%s.%s' % (circle, other['_id'])
    db.social.user.update(
        { '_id': self['_id'] },
        { '$set': { circles_path: { 'name': other['name'] } } })
    follower_circles = 'followers.%s.circles' % self['_id']
    follower_name = 'followers.%s.name' % self['_id']
    db.social.user.update(
        { '_id': other['_id'] },
        { '$push': { follower_circles: circle },
          '$set': { follower_name: self['name'] } })
```

---

**Note:** This does not add earlier posts by `other` to the `self` user's news feed or wall. To include these past posts is an expensive and complex operation beyond the scope of this use case.

---

### Removing

The following function provides support for *removing* users from a circle:

```
def uncircle_user(self, other, circle):
    circles_path = 'circles.%s.%s' % (circle, other['_id'])
    db.social.user.update(
        { '_id': self['_id'] },
        { '$unset': { circles_path: 1 } })
    follower_circles = 'followers.%s.circles' % self['_id']
    db.social.user.update(
        { '_id': other['_id'] },
        { '$pull': { follower_circles: circle } })
    # Special case -- 'other' is completely uncircled
    db.social.user.update(
        { '_id': other['_id'], follower_circles: {'$size': 0 } },
        { '$unset': { 'followers.' + self['_id'] } })
```

## Indexing

Because the update queries both include the `_id` field, these operations require no additional indexes.



### 3.3 Sharding

In order to scale beyond the capacity of a single replica set, you may eventually need to shard the collections described here. Since the `social.user`, `social.wall`, and `social.news` collections contain documents which are specific to a given user, the user's `_id` field is an appropriate shard key:

```
>>> db.command('shardcollection', 'social.user', {
...     'key': {'_id': 1 } } )
{ "collectionsharded": "social.user", "ok": 1 }
>>> db.command('shardcollection', 'social.wall', {
...     'key': {'user_id': 1 } } )
{ "collectionsharded": "social.wall", "ok": 1 }
>>> db.command('shardcollection', 'social.news', {
...     'key': {'user_id': 1 } } )
{ "collectionsharded": "social.news", "ok": 1 }
```

The posting user's `_id` is *not* the best choice for a shard key for the `social.post` collection because queries and updates to this table use the `_id` field. To shard the `social.post` collection on `_id`, then, you'll need to execute the following command:

```
>>> db.command('shardcollection', 'social.post', {
...     'key': {'_id': 1 } } )
{ "collectionsharded": "social.post", "ok": 1 }
```

#### See Also:

“/faq/sharding” and the “[Sharding](#) wiki page.