```
{
    _id: ObjectId(...),
    node_id: ObjectId(...),
    slug: '34db',
    posted: ISODateTime(...),
    author: {
              id: ObjectId(...),
              name: 'Rick'
             },
    text: 'This is so bogus ... '
}
```

This form is only suitable for displaying comments in chronological order. Comments store the following:

- The `node_id` field that references the node parent
- A URL-compatible `slug` identifier
- A `posted` timestamp
- An `author` subdocument that contains a reference to a user's profile in the `id` field and their name in the `name` field
- The full `text` of the comment

In order to support threaded comments, we need to use a slightly different structure:

```
{
    _id: ObjectId(...),
    node_id: ObjectId(...),
    parent_id: ObjectId(...),
    slug: '34db/8bda'
    full_slug: '2012.02.08.12.21.08:34db/2012.02.09.22.19.16:8bda',
    posted: ISODateTime(...),
    author: {
              id: ObjectId(...),
              name: 'Rick'
             },
    text: 'This is so bogus ... '
}
```

This structure:

- Adds a `parent_id` field that stores the contents of the `_id` field of the parent comment
- Modifies the `slug` field to hold a path composed of the parent or parent's slug and this comment's unique slug
- Adds a `full_slug` field that combines the slugs and time information to make it easier to sort documents in a threaded discussion by date

Operation: Post a new comment

To post a new comment in a chronologically ordered (i.e., without discussion threading) system, we just need to use a regular `insert()`:

```
slug = generate_pseudorandom_slug()
db.comments.insert({
    'node_id': node_id,
    'slug': slug,
    'posted': datetime.utcnow(),
    'author': author_info,
    'text': comment_text })
```

To insert a comment for a system with threaded comments, we first need to generate the appropriate `slug` and `full_slug` values based on the parent comment:

```
posted = datetime.utcnow()

# generate the unique portions of the slug and full_slug
slug_part = generate_pseudorandom_slug()
full_slug_part = posted.strftime('%Y.%m.%d.%H.%M.%S') + ':' + slug_part
# load the parent comment (if any)
if parent_slug:
    parent = db.comments.find_one(
        {'node_id': node_id, 'slug': parent_slug })
    slug = parent['slug'] + '/' + slug_part
    full_slug = parent['full_slug'] + '/' + full_slug_part
else:
    slug = slug_part
    full_slug = full_slug_part

# actually insert the comment
db.comments.insert({
    'node_id': node_id,
    'slug': slug,
    'full_slug': full_slug,
    'posted': posted,
    'author': author_info,
    'text': comment_text })
```

Operation: View paginated comments

To view comments that are not threaded, we just need to select all comments participating in a discussion and sort by the `posted` field. For example:

```
cursor = db.comments.find({'node_id': node_id})
cursor = cursor.sort('posted')
cursor = cursor.skip(page_num * page_size)
cursor = cursor.limit(page_size)
```

Since the `full_slug` field contains both hierarchical information (via the path) and chronological information, we can use a simple sort on the `full_slug` field to retrieve a threaded view:

```
cursor = db.comments.find({'node_id': node_id})
cursor = cursor.sort('full_slug')
cursor = cursor.skip(page_num * page_size)
cursor = cursor.limit(page_size)
```

To support these queries efficiently, maintain two compound indexes on `node_id, posted` and `node_id, full_slug`:

```
>>> db.comments.ensure_index([
...     ('node_id', 1), ('posted', 1)])
>>> db.comments.ensure_index([
...     ('node_id', 1), ('full_slug', 1)])
```

Operation: Retrieve comments via direct links

To directly retrieve a comment, without needing to page through all comments, we can select by the `slug` field:

```
comment = db.comments.find_one({
    'node_id': node_id,
    'slug': comment_slug})
```

We can also retrieve a "subdiscussion," or a comment and all of its descendants recursively, by performing a regular expression prefix query on the `full_slug` field:

```
import re

subdiscussion = db.comments.find_one({
    'node_id': node_id,
    'full_slug': re.compile('^' + re.escape(parent_full_slug)) })
subdiscussion = subdiscussion.sort('full_slug')
```

Since we've already created indexes on `{ node_id: 1, full_slug: 1 }` to support retrieving subdiscussions, we don't need to add any other indexes here to achieve good performance.

Approach: Embedding All Comments

This design embeds the entire discussion of a comment thread inside of its parent node document.

Consider the following prototype `topic` document:

```
{ _id: ObjectId(...),
  ...,
  metadata: {
    ...
    comments: [
        { posted: ISODateTime(...),
          author: { id: ObjectId(...), name: 'Rick' },
          text: 'This is so bogus ... ' },
        ... ],
    }
}
```

This structure is only suitable for a chronological display of all comments because it embeds comments in chronological order. Each document in the array in the `comments` contains the comment's date, author, and text.

To support threading using this design, we would need to embed comments within comments, using a structure more like the following:

```
{ _id: ObjectId(...),
  ... lots of topic data ...
  metadata: {
    ...,
    replies: [
      { posted: ISODateTime(...),
         author: { id: ObjectId(...), name: 'Rick' },
         text: 'This is so bogus ... ',
         replies: [
             { author: { ... }, ... },
             ... ]
      }
      ... ]
  }
}
```

Here, the `replies` field in each comment holds the subcomments, which can in turn hold subcomments.

Operation: Post a new comment

To post a new comment in a chronologically ordered (i.e., unthreaded) system, we need the following `update`:

```
db.cms.nodes.update(
    { ... node specification ... },
    { '$push': { 'metadata.comments': {
        'posted': datetime.utcnow(),
        'author': author_info,
        'text': comment_text } } } )
```

The `$push` operator inserts comments into the `comments` array in correct chronological order. For threaded discussions, the update operation is more complex. To reply to a comment, the following code assumes that it can retrieve the *path* as a list of positions, for the parent comment:

```
if path != []:
    str_path = '.'.join('replies.%d' % part for part in path)
```

```
        str_path += '.replies'
    else:
        str_path = 'replies'
    db.cms.nodes.update(
        { ... node specification ... },
        { '$push': {
            'metadata.' + str_path: {
                'posted': datetime.utcnow(),
                'author': author_info,
                'text': comment_text } } } )
```

This constructs a field name of the form metadata.replies.0.replies.2... as str_path and then uses this value with the $push operator to insert the new comment into the replies array.

Operation: View paginated comments

To view the comments in a nonthreaded design, we need to use the $slice operator:

```
node = db.cms.nodes.find_one(
    { ... node specification ... },
    { ... some fields relevant to your page from the root discussion ...,
      'metadata.comments': { '$slice': [ page_num * page_size, page_size ] }
    })
```

To return paginated comments for the threaded design, we must retrieve the whole document and paginate the comments within the application:

```
node = db.cms.nodes.find_one(... node specification ...)

def iter_comments(obj):
    for reply in obj['replies']:
        yield reply
        for subreply in iter_comments(reply):
            yield subreply

paginated_comments = itertools.slice(
    iter_comments(node),
    page_size * page_num,
    page_size * (page_num + 1))
```

Operation: Retrieve a comment via direct links

Instead of retrieving comments via slugs as in "Approach: One Document per Comment" (page 111), the following example retrieves comments using their position in the comment list or tree. For chronological (i.e., nonthreaded) comments, we'll just use the $slice operator to extract a single comment, as follows:

```
node = db.cms.nodes.find_one(
    {'node_id': node_id},
    {'comments': { '$slice': [ position, position ] } })
comment = node['comments'][0]
```

For threaded comments, we must know the correct path through the tree in our application, as follows:

```
node = db.cms.nodes.find_one(... node specification ...)
current = node.metadata
for part in path:
    current = current.replies[part]
comment = current
```

Approach: Hybrid Schema Design

In the "hybrid approach," we store comments in "buckets" that hold about 100 comments. Consider the following example document:

```
{ _id: ObjectId(...),
  node_id: ObjectId(...),
  page: 1,
  count: 42,
  comments: [ {
        slug: '34db',
        posted: ISODateTime(...),
        author: { id: ObjectId(...), name: 'Rick' },
        text: 'This is so bogus ... ' },
    ... ]
}
```

Each document maintains `page` and `count` data that contains metadata regarding the page number and the comment count in this page, in addition to the `comments` array that holds the comments themselves.

Operation: Post a new comment

In order to post a new comment, we need to `$push` the comment onto the last page and `$inc` that page's comment count. Consider the following example that adds a comment onto the last page of comments for some node:

```
def post_comment(node, comment):
    result = db.comment_pages.update(
        { 'node_id': node['_id'],
          'page': node['num_comment_pages'],
          'count': { '$lt': 100 } }, ❶
        { '$inc': { 'count': 1 },
          '$push': { 'comments': comment } },
        upsert=True)

    if not result['updatedExisting']:
        db.cms.nodes.update(
            { '_id': node['_id'],
              'num_comment_pages': node['num_comment_pages'] },
            { '$inc': { 'num_comment_pages': 1 } }) ❷
        db.comment_pages.update(
            { 'node_id': node['_id'],
              'page': node['num_comment_pages'] + 1},
            { '$inc': { 'count': 1 },
              '$push': { 'comments': comment } },
            upsert=True) ❸
```

There are a few things to note about this code:

❶ The first update will only `$push` a comment if the page is not yet full.

❷ If the last comment page *is* full, we need to increment the `num_comment_pages` property in the node (so long as some other process has not already incremented that property).

❸ We also need to re-run the update to `$push` the comment onto the newly created comment page. Here, we've dropped the `count` constraint to make sure the `$push` goes through. (While it's technically possible that 100 other concurrent writers were adding comments and the new page is already full, it's highly unlikely, and the application works just fine if there happen to be 101 comments on a page.)

To support the update operations, we need to maintain a compound index on `node_id, page` in the `comment_pages` collection:

```
>>> db.comment_pages.ensure_index([
...     ('node_id', 1), ('page', 1)])
```

Operation: View paginated comments

The following function defines how to paginate comments where the number of comments on a page is not known precisely (i.e., with *roughly* 100 comments, as in this case):

```
def find_comments(discussion_id, skip, limit):
    query = db.comment_pages.find(
        { 'node_id': node_id } )
    query = query.sort('page')
    for page in query:
        new_skip = skip - page['count']
        if new_skip >= 0:
            skip = new_skip
            continue
        elif skip > 0:
            comments = page['comments'][skip:]
        else:
            comments = page['comments']
        skip = new_skip
        for comment in comments:
            if limit == 0:
                break
            limit -= 1
            yield comment
        if limit == 0: break
```

Here, we iterate through the pages until our skip requirement is satisfied, then yield comments until our limit requirement is satisfied. For example, if we have three pages of comments with 100, 102, 101, and 22 comments on each page, and we wish to retrieve comments where skip=300 and limit=50, we'd use the following algorithm:

Skip	Limit	Discussion
300	50	Page 0 has 100 comments, so skip -= 100.
200	50	Page 1 has 102 comments, so skip -= 102.
98	50	Page 2 has 101 comments, so set skip=0 and return last 3 comments.
0	47	Page 3 has 22 comments, so return them all and set limit-= 22.
0	25	There are no more pages; terminate loop.

Operation: Retrieve a comment via direct links

To retrieve a comment directly without paging through all preceding pages of commentary, we'll use the slug to find the correct page, and then use application logic to find the correct comment:

```
page = db.comment_pages.find_one(
    { 'node_id': node_id,
      'comments.slug': comment_slug},
    { 'comments': 1 })
for comment in page['comments']:
    if comment['slug'] = comment_slug:
        break
```

To perform this query efficiently, we'll need a new index on node_id, comments.slug (this is assuming that slugs are only guaranteed unique *within* a node):

```
>>> db.comment_pages.ensure_index([
...    ('node_id', 1), ('comments.slug', 1)])
```

Sharding Concerns

For all of the architectures just discussed, we will want the node_id field to participate in any shard key we pick.

For applications that use the "one document per comment" approach, we'll use the slug (or full_slug, in the case of threaded comments) fields in the shard key to allow the mongos instances to route requests by slug:

```
>>> db.command('shardcollection', 'dbname.comments', {
...     'key' : { 'node_id' : 1, 'slug': 1 } })
{ "collectionsharded" : "dbname.comments", "ok" : 1 }
```

In the case of comments that are fully embedded in parent content, the comments will just participate in the sharding of their parent document.

For hybrid documents, we can use the page number of the comment page in the shard key along with the node_id to prevent a single discussion from creating a giant, unsplittable chunk of comments. The appropriate command for this is as follows:

```
>>> db.command('shardcollection', 'dbname.comment_pages', {
...     key : { 'node_id' : 1, 'page': 1 } })
{ "collectionsharded" : "dbname.comment_pages", "ok" : 1 }
```

CHAPTER 7

Online Advertising Networks

In this chapter, we'll examine building an online advertising network that connects advertisers and media websites. Advertisers provide the ads for display, with each ad designed for a particular *ad zone*. Media sites, on the other hand, provide content pages for display with various regions marked for serving ads. When the media site displays a page, it makes a request to the ad network for one or more ads to display in its ad zones.

As part of the ad serving, the ad network records the number of pageviews of each ad in order to track statistics for the ad, which may then also be used to bill the advertiser.

Solution Overview

This solution is structured as a progressive refinement of the ad network, starting out with the basic data storage requirements and adding more advanced features to the schema to support more advanced ad targeting. The key performance criterion for this solution is the latency between receiving an ad request and returning the (targeted) ad to be displayed.

Design 1: Basic Ad Serving

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` to be served.
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 to be displayed, recording the pageview for the ad as well.

This design uses the `site_id` and `zone_id` submitted with the ad request, as well as information stored in the ad inventory collection, to make the ad targeting decisions. Later examples will build on this, allowing more advanced ad targeting.

Schema Design

A very basic schema for storing ads available to be served 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 },
    ... ]
}
```

Note that for each site-zone combination you'll be storing a list of ads, sorted by their *eCPM* values.

eCPM, CPM, CPC, CTR, etc.

The world of online advertising is full of somewhat cryptic acronyms. Most of the decisions made by the ad network in this chapter will be based on the eCPM, or effective cost per mille. This is a synthetic measure meant to allow comparison between *CPM* (cost per mille) ads, which are priced based on the number of impressions, and *CPC* (cost per click) ads, which are priced per click.

The eCPM of a CPM ad is just the CPM. Calculating the eCPM of a CPC ad is fairly straightforward, based on the *CTR* (click-through rate), which is defined as the number of clicks per ad impression. The formula for eCPM for a CPC ad then is:

$eCPM = CPC \times CTR \times 1000$

In this chapter, we'll assume that the eCPM is already known for each ad, though you'll obviously need to calculate it in a real ad network.

Operation: Choose an Ad to Serve

The query we'll use to choose which ad to serve selects a compatible ad and sorts by the advertiser's `ecpm` bid in order to maximize the ad network's profits:

```
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}) ❶
    if site is None: return None
    if len(site['ads']) == 0: return None
    ecpm_groups = groupby(site['ads'], key=lambda ad:ad['ecpm']) ❷
    ecpm, ad_group = ecpm_groups.next()
    return choice(list(ad_group))❸
```

❶ First, we find a compatible site and zone for the ad request.

❷ Next, we group the ads based on their eCPM. This step requires that the ads already be sorted by descending eCPM.

❸ Finally, we randomly select an ad from the most expensive ad group.

In order to execute the ad choice with the lowest latency possible, we need to maintain a compound index on `site_id`, `zone_id`:

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

Operation: Make an Ad Campaign Inactive

One case we need to deal with is making a campaign inactive. This may happen for a variety of reasons. For instance, the campaign may have reached its end date or exhausted its budget for the current time period. In this case, the logic is fairly straightforward:

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

This update statement first selects only those ad zones that had available ads from the given `campaign_id` and then uses the `$pull` modifier to remove them from rotation.

To execute the multiupdate quickly, we'll keep an index on the `ads.campaign_id` field:

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

Sharding Concerns

In order 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 (and the highest possible throughput) in the ad selection operation, the shard key needs to be chosen to allow MongoDB to route the ad.zone query to a single shard. In this case, a good approach is to shard on the site_id, zone_id combination:

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

Design 2: Adding Frequency Capping

One problem with the logic described in "Design 1: Basic Ad Serving" (page 121) is that it will tend to display the same ad over and over again until the campaign's budget is exhausted. To mitigate this, advertisers may wish to limit the frequency with which a given user is presented a particular ad. This process is called frequency capping and is an example of user profile targeting in advertising.

In order to perform frequency capping (or any type of user targeting), the ad network needs to maintain a profile for each visitor, typically implemented as a cookie in the user's browser. This cookie, effectively a user_id, is then transmitted to the ad network 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 Design

In order to use the user profile data, we 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: [ ... ],
        ...
      }
  }
```

There are a few things to note about the user profile:

- All data is embedded in a single profile document. When you need to query this data (detailed next), you don't necessarily know which advertiser's ads you'll be showing, so it's a good practice to embed all advertisers in a single document.
- The event information is grouped by event type within an advertiser, and sorted by timestamp. This allows rapid lookups of a stream of a particular type of event.

Operation: Choose an Ad to Serve

The query we'll use to choose which ad to serve now needs to iterate through ads in order of profitability and select the "best" ad that also satisfies the advertiser's targeting rules (in this case, the frequency cap):

```
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}) ❶
    if site is None or len(site['ads']) == 0: return None
    ads = ad_iterator(site['ads']) ❷
    user = db.ad.user.find_one({'user_id': user_id}) ❸
    if user is None:
        # any ad is acceptable for an unknown user
        return ads.next()
    for ad in ads: ❹
        advertiser_id = ad['campaign_id'].split(':', 1)[0]
        if ad_is_acceptable(ad, user[advertiser_id]):
            return ad
    return None
```

❶ Here we once again find all ads that are targeted to that particular site and ad zone.

❷ Next, we have factored out a Python generator that will iterate through all the ads in order of profitability.

❸ Now we load the user profile for the given user. If there is no profile, we return the first ad in the iterator.

❹ Finally, we iterate through each of the ads and check it using the ad_is_acceptable function.

Here's our ad_iterator generator:

```
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
```

This generator yields the ads in an order that both maximizes profitability and randomly shuffles ads of the same eCPM. Finally, here's our ad filter 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
```

This function examines all the user's ad impressions for the current day and rejects an ad that has been displayed to that user.

In order to retrieve the user profile with the lowest latency possible, there needs to be 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 user profile:

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

Design 3: Keyword Targeting

Where frequency capping in the previous section is an example of user profile targeting, you may also wish to perform content targeting so that the user receives relevant ads for the particular page being viewed. The simplest example of this is targeting ads at the result of a search query. In this case, a list of keywords is sent to the choose_ad() call along with the site_id, zone_id, and user_id.

Schema Design

In order to choose relevant ads, we'll need to expand the ad.zone collection to store relevant 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_eclass_1',
      ad_unit_id: 'search1',
      keywords: [ 'car', 'performance' ],
      ecpm: 200 },
    ... ]
}
```

Operation: Choose a Group of Ads to Serve

In the approach described here, we'll choose a number of ads that match the keywords used in the search, so the following code has been tweaked to return an iterator over ads in descending order of preference. We've also modified the ad_iterator to take the list of keywords as a second parameter:

```
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
    for ad in ads:
        advertiser_id = ad['campaign_id'].split(':', 1)[0]
        if ad_is_acceptable(ad, user[advertiser_id]):
            yield ad
    return None
```

Our ad_iterator method has been modified to allow us to score ads based on both their eCPM as well as their relevance:

```
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))

  def ad_is_acceptible(ad, profile):
    # same as above
```

The main thing to note in the preceding code is that ads must now be sorted according to some score, which in this case is computed based on a combination of the ecpm of the ad as well as the number of keywords matched. More advanced use cases may boost the importance of various keywords, but this goes beyond the scope of this use case. One thing to keep in mind is that because the ads are now being sorted at display time, there may be performance issues if a large number of ads are competing for the same display slot.

CHAPTER 8

Social Networking

In this chapter, we'll explore how you could use MongoDB to store the social graph for a social networking site. We'll look at storing and grouping followers as well as how to publish events to different followers with different privacy settings.

Solution Overview

Our solution assumes a *directed* social graph where a user can choose whether or not to follow another user. Additionally, the user can designate "circles" of users with which to share updates, in order to facilitate fine-grained control of privacy. The solution presented here is designed in such a way as to minimize the number of documents that must be loaded in order to display any given page, even at the expense of complicating updates.

The particulars of what type of data we want to host on a social network obviously depend on the type of social network we're designing, and is largely beyond the scope of this use case. In particular, the main variables that you would have to consider in adapting this use case to your particular situation are:

What data should be in a user profile?
: This may include gender, age, interests, relationship status, and so on for a "personal" social network, or may include resume-type data for a more "business-oriented" social network.

What type of updates are allowed?
: Again, depending on what flavor of social network you are designing, you may wish to allow posts such as status updates, photos, links, check-ins, and polls, or you may wish to restrict your users to links and status updates.

Schema Design

In the solution presented here, we'll 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:

```
{
  _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' },
      ...  },
    ...}
4  },
  blocked: ['gh1...0d']
}
```

There are a few things to note about this schema:

- Rather than using a "raw" `ObjectId` for the `_id` field, we'll use a base64-encoded version. Although we *can* use raw `ObjectId` values as keys, we can't use them to "reach inside" a document in a query or an update. By base64-encoding the `_id` values, we can use queries or updates that include the `_id` value like `circles.10gen.T4Y...AD`.
- We're storing the social graph bidirectionally in the `followers` and `circles` collections. While this is technically redundant, having the bidirectional connections is useful both for displaying the user's followers on the profile page, as well as propagating posts to other users, as we'll see later.
- In addition to the normal "positive" social graph, this schema above stores a block list that contains an array of user IDs for posters whose posts never appear on the user's wall or news feed.

- The particular profile data stored for the user is isolated into the `profile` subdocument, allowing us to evolve the profile's schema as necessary without worrying about name collisions with other parts of the schema that need to remain fixed for social graph operations.

Of course, to make the network interesting, it's necessary to add various types of posts. We'll put these 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 ... ]
}
```

Here, the post stores minimal author information (`by`), the post `type`, a timestamp `ts`, post details `detail` (which vary by post type), and a `comments` array. In this case, the schema 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, refer back to "Storing Comments" (page 111).

A couple of points are worthy of further discussion:

- Author information is truncated; just enough is stored in each `by` property to display the author name and a link to the author profile. If a user wants more detail on a particular author, we can fetch this information as they request it. Storing minimal information like this helps keep the document small (and therefore fast.)
- The visibility of the post is controlled via the `circles` property; any user that is part of one of the listed circles can view the post. The special values `*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 this polymorphic information into a subdocument is a good practice, helping to identify which parts of the document are common to all posts and which can vary. In this case, we would store different data for a photo post versus a status update, while still keeping the metadata (`_id`, `by`, `circles`, `type`, `ts`, and `comments`) the same.