CHAPTER 3

Mimicking Transactional Behavior

Relational database schemas often rely on the existence of atomic multistatement transactions to ensure data consistency: either all of the statements in a group succeed, or all of the statements fail, moving the database from one self-consistent state to another. When trying to scale relational databases over multiple physical servers, however, transactions must use a two-phase commit protocol, which significantly slows down transactions that may span multiple servers. MongoDB, in not allowing multidocument atomic transactions, effectively side-steps this problem and substitutes another one: how to maintain consistency in the *absence* of transactions.

In this chapter, we'll explore how MongoDB's document model and its atomic update operations enable an approach that maintains consistency where a relational database would use a transaction. We'll also look at how we can use an approach known as *compensation* to mimic the transactional behavior of relational databases.

The Relational Approach to Consistency

One of the goals of relational database normalization is the ability to make atomic changes to a single row, which maintains the domain-level consistency of your data model. Although normalization goes a long way toward such consistency enforcement, there are some types of consistency requirements that are difficult or impossible to express in a single SQL statement:

- Deleting a row in a one-to-many relationship should also delete the many rows joined to it. For instance, deleting an order from the system should delete its subordinate rows.
- Adjusting the quantity of a line item on an order should update the order total cost (assuming that cost is stored in the order row itself).

- In a bank account transfer, the debit from the sending account and the credit into the receiving account should be an atomic operation where both succeed or both fail. Additionally, other simultaneous transactions should not see the data in an incomplete state where either the debit or credit has not yet completed.

To address situations such as these, relational databases use atomic multistatement transactions, where a group of updates to a database either all succeed (via `COMMIT`) or all fail (via `ROLLBACK`). The drawback to multistatement transactions is that they can be quite slow if you are using a distributed database. However, it is possible to maintain consistency across multiple servers in a distributed database using a two-phase commit protocol, summarized as follows:

1. Each server prepares to execute the transaction. In this stage, all the updates are computed and guaranteed not to cause consistency violations within the server.
2. Once all servers have executed the "prepare" step, each server then applies the updates that are part of the transaction.

The drawback to a two-phase commit is that it can significantly slow down your application. Since each server guarantees that the transaction can be completed at the end of the prepare step, the server will typically maintain a set of locks on data to be modified. These locks must then be held until all the *other* servers have completed *their* prepare step, which may be a lengthy process.

MongoDB, designed from the beginning with an eye toward distributed operation, "solves" this problem by giving up on the idea of multidocument transactions. In MongoDB, each update to a document stands alone.

Compound Documents

MongoDB's document model and its update operators combine to enable operations that would require transactions in relational databases. For example, consider deleting an order from a database where each order contains multiple line items. In a relational database, we could use a transaction to ensure that the order "cleans up after itself":

```
BEGIN TRANSACTION;
DELETE FROM orders WHERE id='11223';
DELETE FROM order_items WHERE order_id='11223';
COMMIT;
```

Since this is such a common use case, many relational database systems provide cascading constraints in the table-creation logic that do this automatically. For instance, we may have designed our tables using the following SQL:

```
CREATE TABLE `orders` (
    `id` CHAR(5) NOT NULL,
    ...
    PRIMARY KEY(`id`))

CREATE TABLE `order_items` (
    `order_id` CHAR(5) NOT NULL,
    `sku` CHAR(8) NOT NULL,
    ...
    PRIMARY KEY(`order_id`, `sku`),
    FOREIGN KEY(`order_id`) REFERENCES orders.id
        ON DELETE CASCADE)
```

In this case, we could execute a simpler SQL statement:

```
DELETE FROM orders WHERE id='11223';
```

However, despite the fact that we're not explicitly calling `BEGIN` and `COMMIT`, the database system is still doing the work of a full, multitable transaction.

A developer new to MongoDB may approach an order management system by designing a relational-style schema:

```
// "orders" document
{
  _id: '11223',
  ...
}

// "order_items" document
{
  _id: ObjectId(...),
  order_id: '11223',
  sku: '...',
  ...
}
```

Deleting such an order, however, presents a problem. One approach is to use two non-atomic updates:

```
db.orders.remove({'_id': '1123'})
db.order_items.remove({'order_id': '11223'})
```

Of course, this can leave dangling `order_items` documents around if an exception occurs between the `remove` calls, and other processes can end up seeing orphaned `order_items` documents if they happen to query that collection between our operations. Alternatively, we could reverse the order:

```
db.order_items.remove({'order_id': '11223'})
db.orders.remove({'_id': '1123'})
```

Although this guarantees that we won't have "garbage" items in our database, it also introduces the problem of having partially deleted orders, where some or all of the line items are deleted but the order itself remains. A better approach is to simply embed the order items within the order document itself:

```
// "orders" document
{
  _id: '11223',
  ...
  items: [
    { sku: '...', ... },
    { sku: '...', ... },
    ...
  ]
}
```

Deleting the order, then, is as simple as the single statement:

```
db.orders.remove({'_id': '1123'})
```

Using Complex Updates

Although using document embedding in your MongoDB schema makes some "transactional" problems in relational databases easier to handle, there are other cases where we need something more. For instance, consider our order management system again. Suppose we wish to store the order total price as well as each line item's price so that we can easily display the order total without computing it each time. A document might look like the following:

```
// "orders" document
{
  _id: '11223',
  total: 500.94,
  ...
  items: [
    { sku: '123', price: 55.11, qty: 2 },
    { sku: '...', ... },
    ...
  ]
}
```

Now suppose we want to update the quantity of item 123 to 3. A naive approach might be to read the document from the database, update it in-memory, and then save it back. Unfortunately, this approach introduces race conditions between the loading of the order and saving it back. What we need is a way to atomically update the document *without* doing it in client application code. We can use MongoDB's atomic update operators to perform the same operation in a single step. We *might* do so with the following code:

```
def increase_qty(order_id, sku, price, qty):
    total_update = price * qty
    while True:
        db.orders.update(
            { '_id': order_id, 'items.sku': sku },
            { '$inc': {
                'total': total_update,
                'items.$.qty': qty } })
```

In this case, we still have a risk that another operation *removed* the line item we are interested in updating (perhaps in another browser window). To account for this case, we must detect whether our update actually succeeds by checking its return value. If the update failed, someone must have removed that line item and we must try to `$push` it onto the array with its new quantity:

```
def increase_qty(order_id, sku, price, qty):
    total_update = price * qty
    while True:
        result = db.orders.update(
            { '_id': order_id, 'items.sku': sku },
            { '$inc': {
                'total': total_update,
                'items.$.qty': qty } })
        if result['updatedExisting']: break
        result = db.orders.update(
            { '_id': order_id, 'items.sku': { '$ne': sku } },
            { '$inc': { 'total': 110.22 },
              '$push': { 'items': { 'sku': sku,
                                    'qty': qty,
                                    'price': price } } })
        if result['updatedExisting']: break
```

Optimistic Update with Compensation

There are some cases where it's just not possible to do your operation with a single `update()` statement in MongoDB. For instance, consider the account transfer problem where we must debit one account and credit another. In these cases, we are stuck making multiple updates, but we must ensure that our database is eventually consistent by examining all the places where we could have an error. A naive approach would simply store the account balance in each document and update them separately. Our documents would be quite simple:

```
{ _id: 1, balance: 100 }
{ _id: 2, balance: 0 }
```

The code to update them is likewise simple:

```
def transfer(amt, source, destination):
    result = db.accounts.update(
        { '_id': source, 'balance': { '$gte': amt } },
        { '$inc': { 'balance': -amt } })
    if not result['updatedExisting']:
        raise InsufficientFundsError(source)
    db.accounts.update(
        { '_id': destination },
        { '$inc': { 'balance': amt } } )
```

The problem with this approach is that, if an exception occurs between the source account being debited and the destination account being credited, the funds are lost.

> You should be exceedingly careful if you find yourself designing an application-level, two-phase commit protocol. It's easy to miss a particular failure scenario, and there are many opportunities to miss a race condition and introduce inconsistency into your data, by a small oversight in design or a bug in implementation. As a rule of thumb, whenever it's possible to structure your schema such that all your atomic updates occur *within* a document boundary, you should do so, but it's nice to know you can still fall back to two-phase commit if you absolutely have to.

A better approach to this problem is to emulate transactions in the data model. Our basic approach here will be to create a "transaction" collection containing documents that store the state of all outstanding transfers:

- Any transaction in the "new" state may be rolled back if it times out.
- Any transaction in the "committed" state will always (eventually) be retired.
- Any transaction in the "rollback" state will always (eventually) be reversed.

Our transaction collection contains documents of the following format:

```
{
  _id: ObjectId(...),
  state: 'new',
  ts: ISODateTime(...),
  amt: 55.22,
  src: 1,
  dst: 2
}
```

Our account schema also changes just a bit to store the pending transaction IDs along with each account:

```
{ _id: 1, balance: 100, txns: [] }
{ _id: 2, balance: 0, txns: [] }
```

The top-level transfer function transfers an amount from one account to another as before, but we have added a maximum amount of time allowable for the transaction to complete. If a transaction takes longer, it will eventually be rolled back by a periodic process:

```
def transfer(amt, source, destination, max_txn_time):
    txn = prepare_transfer(amt, source, destination)
    commit_transfer(txn, max_txn_time)
```

Note that in the preceding code we now have a two-phase commit model of our transfer: first the accounts are prepared, then the transaction is committed. The code to prepare the transfer is as follows:

```
def prepare_transfer(amt, source, destination):
    # Create a transaction object
    now = datetime.utcnow()
    txnid = ObjectId()
    txn = {
      '_id': txnid,
      'state': 'new',
      'ts': datetime.utcnow(),
      'amt': amt,
      'src': source,
      'dst': destination }
    db.transactions.insert(txn)

    # "Prepare" the accounts
    result = db.accounts.update(
        { '_id': source, 'balance': { '$gte': amt } },
        { '$inc': { 'balance': -amt },
          '$push': { 'txns': txn['_id'] } })
    if not result['updatedExisting']:
        db.transaction.remove({'_id': txnid})
        raise InsufficientFundsError(source)
    db.accounts.update(
        { '_id': dest },
        { '$inc': { 'balance': amt },
          '$push': { 'txns': txn['_id'] } })
    return txn
```

There are two key insights applied here:

- The source and destination accounts store a list of pending transactions. This allows us to track, in the account document, whether a particular transaction ID is pending.
- The transaction itself must complete during a certain time window. If it does not, a periodic process will roll outstanding transactions back or commit them based on the last state of the transaction. This handles cases where the application or database crashes in the middle of a transaction.

Here's our function to actually commit the transfer:

```
def commit_transfer(txn, max_txn_time):
    # Mark the transaction as committed
    now = datetime.utcnow()
    cutoff = now - max_txn_time
    result = db.transaction.update(
        { '_id': txnid, 'state': 'new', 'ts': { '$gt': cutoff } },
        { '$set': { 'state': 'commit' } })
    if not result['updatedExisting']:
        raise TransactionError(txn['_id'])
        else:
            retire_transaction(txn['_id'])
```

The main purpose of this function is to perform the atomic update of transaction state from new to commit. If this update succeeds, the transaction will eventually be retired, even if a crash occurs immediately after the update. To actually retire the transaction, then, we use the following function:

```
    def retire_transaction(txn_id):
        db.accounts.update(
                { '_id': txn['src'], 'txns._id': txn_id },
                { '$pull': { 'txns': txn_id } })
        db.accounts.update(
                { '_id': txn['dst'], 'txns._id': txn['_id'] },
                { '$pull': { 'txns': txn_id } })
        db.transaction.remove({'_id': txn_id})
```

Note that the retire_transaction is *idempotent*: it can be called any number of times with the same txn_id with the same effect as calling it once. This means that if we have a crash at any point before removing the transaction object, a subsequent cleanup process can still retire the transaction by simply calling retire_transaction again.

We now need to take care of transactions that have timed out, or where the commit or rollback process has crashed in a periodic cleanup task:

```
    def cleanup_transactions(txn, max_txn_time):
        # Find & commit partially-committed transactions
        for txn in db.transaction.find({ 'state': 'commit' }, {'_id': 1}):
            retire_transaction(txn['_id'])

        # Move expired transactions to 'rollback' status:
        cutoff = now - max_txn_time
        db.transaction.update(
            { '_id': txnid, 'state': 'new', 'ts': { '$lt': cutoff } },
            { '$set': { 'state': 'rollback' } })
        # Actually rollback transactions
        for txn in db.transaction.find({ 'state': 'rollback' }):
            rollback_transfer()
```

Finally, in the case where we want to roll back a transfer, we must update the transaction object and *undo* the effects of the transfer:

```
    def rollback_transfer(txn):
        db.accounts.update(
                { '_id': txn['src'], 'txns._id': txn['_id'] },
                { '$inc': { 'balance': txn['amt'] },
                  '$pull': { 'txns': { '_id': txn['_id'] } } })
        db.accounts.update(
                { '_id': txn['dst'], 'txns._id': txn['_id'] },
                { '$inc': { 'balance': -txn['amt'] },
                  '$pull': { 'txns': { '_id': txn['_id'] } } })
        db.transaction.remove({'_id': txn['_id']})
```

Note in particular that the preceding code will only undo a transaction in an account if the transaction is still stored in that account's txns array. This makes the rollback of the transaction idempotent just like retiring a transaction via a commit.

Conclusion

The constraints of a toolset help to define patterns for solving problems. In the case of MongoDB, one of those constraints is the lack of atomic multidocument update operations. The patterns we use in MongoDB to mitigate the lack of atomic multidocument update operations include document embedding and complex updates for basic operations, with optimistic update with compensation available for when we really need a two-phase commit protocol. When designing your application to use MongoDB, more than in relational databases, you must keep in mind which updates you need to be atomic and design your schema appropriately.

CHAPTER 4

Operational Intelligence

The first use cases we'll explore lie in the realm of *operational intelligence*, the techniques of converting transactional data to actionable information in a business setting. Of course, the starting point for any of these techniques is getting the raw transactional data *into* your data store. Our first use case, "Storing Log Data" (page 37), deals with this part of the puzzle.

Once you have the data, of course, the first priority is to generate actionable reports on that data, ideally in real time with the data import itself. We address the generation of these reports in real time in "Pre-Aggregated Reports" (page 52).

Finally, we'll explore the use of more traditional batch aggregation in "Hierarchical Aggregation" (page 63) to see how MongoDB can be used to generate reports at multiple layers of your analytics hierarchy.

Storing Log Data

The starting point for any analytics system is the raw "transactional" data. To give a feel for this type of problem, we'll examine the particular use case of storing event data in MongoDB that would traditionally be stored in plain-text logfiles. Although plain-text logs are accessible and human-readable, they are difficult to use, reference, and analyze, frequently being stored on a server's local filesystem in an area that is generally inaccessible to the business analysts who need these data.

Solution Overview

The solution described here assumes that each server generating events can access the MongoDB instance and has read/write access to some database on that instance. Furthermore, we assume that the query rate for logging data is significantly lower than the insert rate for log data.

This case assumes that you're using a standard uncapped collection for this event data, unless otherwise noted. See "Capped collections" (page 51) for another approach to aging out old data.

Schema Design

The schema for storing log data in MongoDB depends on the format of the event data that you're storing. For a simple example, you might consider standard request logs in the combined format from the Apache HTTP Server. A line from these logs may resemble the following:

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" ...
```

The simplest approach to storing the log data would be putting the exact text of the log record into a document:

```
{
  _id: ObjectId(...),
 line: '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif ...
}
```

Although this solution does capture all data in a format that MongoDB can use, the data is neither particularly useful nor efficient. For example, if you need to find events on the same page, you would need to use a regular expression query, which would require a full scan of the collection. A better approach is to extract the relevant information from the log data into individual fields in a MongoDB *document.*

When designing the structure of that document, it's important to pay attention to the data types available for use in BSON, the MongoDB document format. Choosing your data types wisely can have a significant impact on the performance and capability of the logging system. For instance, consider the date field. In the previous example, `[10/Oct/2000:13:55:36 -0700]` is 28 bytes long. If you store this with the UTC timestamp BSON type, you can convey the same information in only 8 bytes.

Additionally, using proper types for your data also increases query flexibility. If you store date as a timestamp, you can make date range queries, whereas it's very difficult to compare two *strings* that represent dates. The same issue holds for numeric fields; storing numbers as strings requires more space and is more difficult to query.

Consider the following document that captures all data from the log entry:

```
{
     _id: ObjectId(...),
     host: "127.0.0.1",
     logname: null,
     user: 'frank',
     time:  ISODate("2000-10-10T20:55:36Z"),
     request: "GET /apache_pb.gif HTTP/1.0",
        status: 200,
        response_size: 2326,
        referrer: "[http://www.example.com/start.html](http://www.example.com/...",
        user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
    }
```

The is better, but it's quite a large document. When extracting data from logs and designing a schema, you should also consider what information you can omit from your log tracking system. In most cases, there's no need to track *all* data from an event log. To continue this example, here the most crucial information may be the host, time, path, user agent, and referrer, as in the following example document:

```
    {
        _id: ObjectId(...),
        host: "127.0.0.1",
        time:  ISODate("2000-10-10T20:55:36Z"),
        path: "/apache_pb.gif",
        referer: "[http://www.example.com/start.html](http://www.example.com/...",
        user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
    }
```

Depending on your storage and memory requirements, you might even consider omitting explicit time fields, since the BSON `ObjectId` implicitly embeds its own creation time:

```
    {
        _id: ObjectId('...'),
        host: "127.0.0.1",
        path: "/apache_pb.gif",
        referer: "[http://www.example.com/start.html](http://www.example.com/...",
        user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
    }
```

Operations

In this section, we'll describe the various operations you'll need to perform on the logging system, paying particular attention to the appropriate use of indexes and MongoDB-specific features.

Inserting a log record

The primary performance concerns for event-logging systems are:

- How many inserts per second it can support, which limits the event throughput
- How the system will manage the growth of event data, particularly in the case of a growth in insert activity

In designing our system, we'll primarily focus on optimizing insertion speed, while still addressing how we manage event data growth. One decision that MongoDB allows you to make when performing updates (such as event data insertion) is whether you want to trade off data safety guarantees for increased insertion speed. This section will explore the various options we can tweak depending on our tolerance for event data loss.

Write concern

MongoDB has a configurable *write concern*. This capability allows you to balance the importance of guaranteeing that all writes are fully recorded in the database with the speed of the insert.

For example, if you issue writes to MongoDB and do not require that the database issue any response, the write operations will return *very* fast (since the application needs to wait for a response from the database) but you cannot be certain that all writes succeeded. Conversely, if you require that MongoDB acknowledge every write operation, the database will not return as quickly but you can be certain that every item will be present in the database.

The proper write concern is often an application-specific decision, and depends on the reporting requirements and uses of your analytics application.

In the examples in this section, we will assume that the following code (or something similar) has set up an event from the Apache Log. In a real system, of course, we would need code to actually parse the log and create the Python `dict` shown here:

```
>>> import bson
>>> import pymongo
>>> from datetime import datetime
>>> conn = pymongo.Connection()
>>> db = conn.event_db
>>> event = {
...     _id: bson.ObjectId(),
...     host: "127.0.0.1",
...     time:  datetime(2000,10,10,20,55,36),
...     path: "/apache_pb.gif",
...     referer: "[http://www.example.com/start.html](http://www.example.com/...",
...     user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
...}
```

The following command will insert the `event` object into the `events` collection:

```
>>> db.events.insert(event, w=0)
```

By setting `w=0`, you do not require that MongoDB acknowledge receipt of the insert. Although this is the fastest option available to us, it also carries with it the risk that you might lose a large number of events before you notice.

If you want to ensure that MongoDB acknowledges inserts, you can omit the `w=0` argument, or pass `w=1` (the default) as follows:

```
>>> db.events.insert(event)
>>> # Alternatively, you can do this
>>> db.events.insert(event, w=1)
```

MongoDB also supports a more stringent level of write concern, if you have a lower tolerance for data loss. MongoDB uses an on-disk journal file to persist data before writing the updates back to the "regular" data files.

Since journal writes are significantly slower than in-memory updates (which are, in turn, much slower than "regular" data file updates), MongoDB batches up journal writes into "group commits" that occur every 100 ms unless overridden in your server settings. What this means for the application developer is that, on average, any individual writes with `j=True` will take around 50 ms to complete, which is generally even more time than it would take to replicate the data to another server. If you want to ensure that MongoDB not only *acknowledges* receipt of a write operation but also commits the write operation to the on-disk journal before returning successfully to the application, you can use the `j=True` option:

```
>>> db.events.insert(event, j=True)
```

It's important to note that the journal does *not* protect against any failure in which the disk itself might fail, since in that case the journal file itself can be corrupted. Replication, however, *does* protect against single-server failures, and is the recommended way to achieve real durability.

> `j=True` requires acknowledgment from the server, so `w=1` is implied unless you explicitly set `w=N` with N greater than 1.

You can require that MongoDB replicate the data to multiple secondary replica set members before returning:

```
>>> db.events.insert(event, w=2)
```

This will force your application to acknowledge that the data has replicated to two members of the replica set. You can combine options as well:

```
>>> db.events.insert(event, j=True, w=2)
```

In this case, your application will wait for a successful journal commit *and* a replication acknowledgment. This is the safest option presented in this section, but it is the slowest. There is always a trade-off between safety and speed.

Bulk inserts

If possible, you should use bulk inserts to insert event data. All write concern options apply to bulk inserts, but you can pass multiple events to the `insert()` method at once. Batch inserts allow MongoDB to distribute the performance penalty incurred by more stringent write concern across a group of inserts.

If you're doing a bulk insert and *do* get an error (either a network interruption or a unique key violation), your application will need to handle the possibility of a partial bulk insert. If your particular use case doesn't care about missing a few inserts, you can add the `continue_on_error=True` argument to insert, in which case the insert will insert as many documents as possible, and report an error on the *last* insert that failed.

If you use `continue_on_error=True` and *multiple* inserts in your batch fail, your application will only receive information on the *last* insert to fail. The take-away? You can sometimes amortize the overhead of safer writes by using bulk inserts, but this technique brings with it another set of concerns as well.

Finding all events for a particular page

The value in maintaining a collection of event data derives from being able to query that data to answer specific questions. You may have a number of simple queries that you may use to analyze these data.

As an example, you may want to return all of the events associated with a specific value of a field. Extending the Apache access log example, a common case would be to query for all events with a specific value in the `path` field. This section contains a pattern for returning data and optimizing this operation.

In this case, you'd use a query that resembles the following to return all documents with the `/apache_pb.gif` value in the `path` field:

```
>>> q_events = db.events.find({'path': '/apache_pb.gif'})
```

Of course, if you want this query to perform well, you'll need to add an index on the `path` field:

```
>>> db.events.ensure_index('path')
```

Aside: managing index size

One thing you should keep in mind when you're creating indexes is the size they take up in RAM. When an index is accessed randomly, as in the case here with our index on `path`, the entire index needs to be resident in RAM. In this particular case, the total number of distinct paths is typically small in relation to the number of documents, which will limit the space that the index requires.

To actually see the size of an index, you can use the `collstats` database command:

```
>>> db.command('collstats', 'events')['indexSizes']
```

There is actually another type of index that doesn't take up much RAM, and that's a *right-aligned* index. *Right-aligned* refers to the access pattern of a regular index, not a special MongoDB index type: in this case, most of the queries that use the index focus on the largest (or smallest) values in the index, so most of the index is never actually used. This is often the case with time-oriented data, where you tend to query documents from the recent past. In this case, only a very thin "sliver" of the index is ever resident in RAM at a particular time, so index size is of much less concern.

Finding all the events for a particular date

Another operation we might wish to do is to query the event log for all events that happened on a particular date, perhaps as part of a security audit of suspicious activity. In this case, we'll use a range query:

```
>>> q_events = db.events.find('time':
...     { '$gte':datetime(2000,10,10),'$lt':datetime(2000,10,11)})
```

This query selects documents from the `events` collection where the value of the `time` field represents a date that is on or after (i.e., `$gte`) `2000-10-10` but before (i.e., `$lt`) `2000-10-11`. Here, an index on the `time` field would optimize performance:

```
>>> db.events.ensure_index('time')
```

Note that this is a right-aligned index so long as our queries tend to focus on the recent history.

Finding all events for a particular host/date

Expanding on our "security audit" example, suppose we isolated the incident to a particular server and wanted to look at the activity for only a single server on a particular date. In this case, we'd use a query that resembles the following:

```
>>> q_events = db.events.find({
...     'host': '127.0.0.1',
...     'time': {'$gte':datetime(2000,10,10),'$lt':datetime(2000,10,11)}
... })
```

The indexes you use may have significant implications for the performance of these kinds of queries. For instance, you *can* create a compound index on the time-host field pair (noting that order matters), using the following command:

```
>>> db.events.ensure_index([('time', 1), ('host', 1)])
```

To analyze the performance for the above query using this index, MongoDB provides the `explain()` method. In Python for instance, we can execute `q_events.explain()` in a console. This will return something that resembles:

```
{ ..
  u'cursor': u'BtreeCursor time_1_host_1',
  u'indexBounds': {u'host': [[u'127.0.0.1', u'127.0.0.1']],
  u'time': [
      [ datetime.datetime(2000, 10, 10, 0, 0),
        datetime.datetime(2000, 10, 11, 0, 0)]]
  },
  ...
  u'millis': 4,
  u'n': 11,
  u'nscanned': 1296,
  u'nscannedObjects': 11,
  ... }
```

This query had to scan 1,296 items from the index to return 11 objects in 4 milliseconds. Conversely, you can test a different compound index with the `host` field first, followed by the `time` field. Create this index using the following operation:

```
>>> db.events.ensure_index([('host', 1), ('time', 1)])
```

Now, `explain()` tells us the following:

```
{ ...
  u'cursor': u'BtreeCursor host_1_time_1',
  u'indexBounds': {u'host': [[u'127.0.0.1', u'127.0.0.1']],
  u'time': [[datetime.datetime(2000, 10, 10, 0, 0),
      datetime.datetime(2000, 10, 11, 0, 0)]]},
  ...
  u'millis': 0,
  u'n': 11,
  ...
  u'nscanned': 11,
  u'nscannedObjects': 11,
  ...
}
```

Here, the query had to scan 11 items from the index before returning 11 objects in less than a millisecond. Although the index order has an impact on query performance, remember that index scans are *much* faster than collection scans, and depending on your other queries, it may make more sense to use the `{ time: 1, host: 1 }` index depending on usage profile.

Rules of index design

MongoDB indexes are stored in a data structure known as a B-tree. The details are beyond our scope here, but what you need to understand as a MongoDB *user* is that each index is stored in sorted order on all the fields in the index. For an index to be maximally efficient, the key should look just like the queries that use the index. Ideally, MongoDB should be able to traverse the index to the first document that the query returns and *sequentially walk* the index to find the rest.

Because of this sorted B-tree structure, then, the following rules will lead to efficient indexes:

- Any fields that will be queried *by equality* should occur first in the index definition.
- Fields used to sort should occur next in the index definition. If multiple fields are being sorted (such as `(last_name, first_name)`), then they should occur in the same order in the index definition.
- Fields that are queried by range should occur *last* in the index definition.

This leads to some unfortunate circumstances where our index cannot be used optimally:

- Whenever we have a range query on two or more properties, they cannot both be used effectively in the index.
- Whenever we have a range query combined with a sort on a different property, the index is somewhat less efficient than when doing a range and sort on the same property set.

In such cases, the best approach is to test with representative data, making liberal use of `explain()`. If you discover that the MongoDB query optimizer is making a bad choice of index (perhaps choosing to reduce the number of entries scanned at the expense of doing a large in-memory sort, for instance), you can also use the `hint()` method to tell it which index to use.

Counting requests by day and page

Finding requests is all well and good, but more frequently we need to *count* requests, or perform some other aggregate operation on them during analysis. Here, we'll describe how you can use MongoDB's *aggregation framework*, introduced in version 2.1, to select, process, and aggregate results from a large number of documents for powerful ad hoc queries. In this case, we'll count the number of requests per resource (i.e., page) per day in the last month.

To use the aggregation framework, we need to set up a *pipeline* of operations. In this case, our pipeline looks like Figure 4-1 and is implemented by the database command shown here:

```
>>> result = db.command('aggregate', 'events', pipeline=[
...         { '$match': {  ❶
...                 'time': {
...                     '$gte': datetime(2000,10,1),
...                     '$lt':  datetime(2000,11,1) } } },
...         { '$project': {  ❷
...                 'path': 1,
...                 'date': {
```

```
...                     'y': { '$year': '$time' },
...                     'm': { '$month': '$time' },
...                     'd': { '$dayOfMonth': '$time' } } } },
...          { '$group': { ❸
...                 '_id': {
...                     'p':'$path',
...                     'y': '$date.y',
...                     'm': '$date.m',
...                     'd': '$date.d' },
...                 'hits': { '$sum': 1 } } },
...          ])
```

This command aggregates documents from the events collection with a pipeline that:

❶ Uses the `$match` operation to limit the documents that the aggregation framework must process. `$match` is similar to a `find()` query. This operation selects all documents where the value of the `time` field represents a date that is on or after (i.e., `$gte`) `2000-10-10` but before (i.e., `$lt`) `2000-10-11`.

❷ Uses the `$project` operator to limit the data that continues through the pipeline. This operator:

- Selects the `path` field.
- Creates a `y` field to hold the year, computed from the `time` field in the original documents.
- Creates an `m` field to hold the month, computed from the `time` field in the original documents.
- Creates a `d` field to hold the day, computed from the `time` field in the original documents.

❸ Uses the `$group` operator to create new computed documents. This step will create a single new document for each unique path/date combination. The documents take the following form:

- The `_id` field holds a subdocument with the content's `path` field from the original documents in the `p` field, with the `date` fields from the `$project` as the remaining fields.
- The `hits` field uses the `$sum` statement to increment a counter for every document in the group. In the aggregation output, this field holds the total number of documents at the beginning of the aggregation pipeline with this unique date and path.

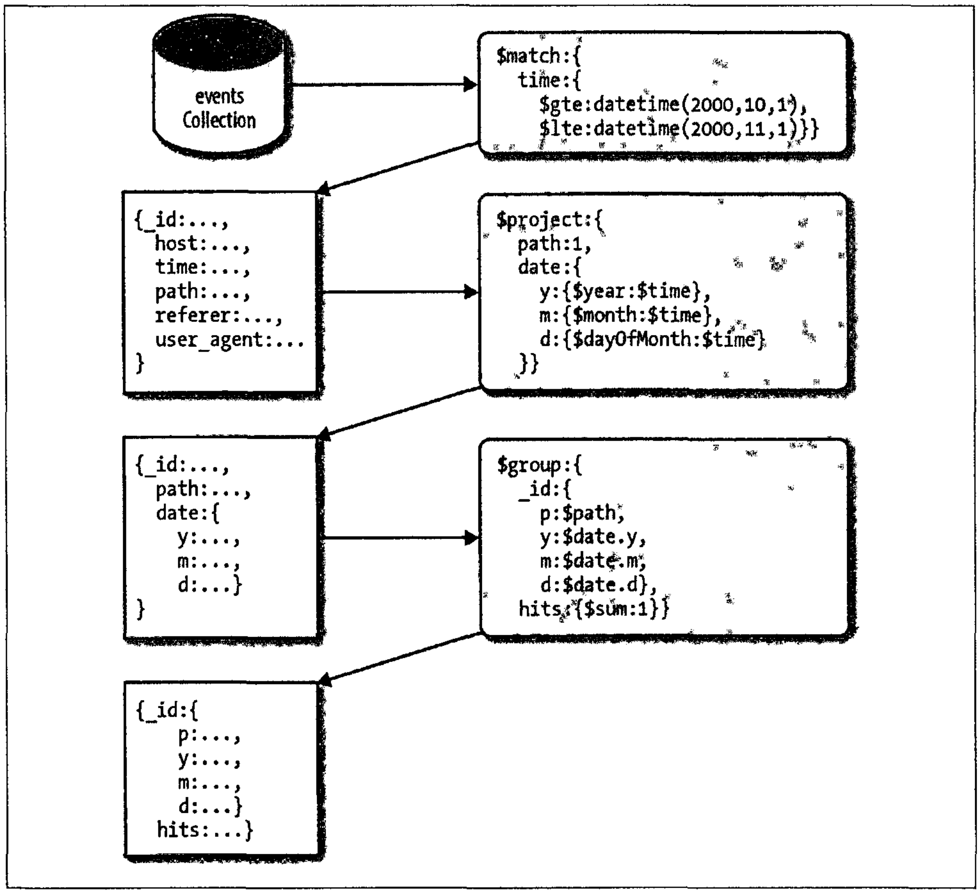

Figure 4-1. Aggregation pipeline

In sharded environments, the performance of aggregation operations depends on the shard key. Ideally, all the items in a particular `$group` operation will reside on the same server.

Although this distribution of documents would occur if you chose the `time` field as the shard key, a field like `path` also has this property and is a typical choice for sharding. See "Sharding Concerns" (page 48) for additional recommendations concerning sharding.

SQL equivalents

To translate statements from the aggregation framework to SQL, you can consider the `$match` equivalent to `WHERE`, `$project` to `SELECT`, and `$group` to `GROUP BY`.