KARPAGAMACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)
(For the candidates admitted from 2017 onwards)
DEPARTMENT OF CS, CA & IT

SUBJECT NAME : MONGODB
SEMESTER : II
SUBJECT CODE :19CSP203 **CLASS: I M.Sc CS**

Instruction Hours / week:L: 4 T: 0 P: 0 **Marks:** Internal:**40**External:**60**Total: **100**
End Semester Exam :3 Hours

Course Objectives
To provide students the knowledge and skills to master the NoSQL database mongoDB.

Course Outcomes(COs)
1. To provide students the right skills and knowledge needed to developApplications on mongoDB

2. To provide students the right skills and knowledge needed to run Applications on mongoDB

Unit I- GETTING STARTED

A database for the modern web – MongoDB through the JavaScript shell – Writing programs using MongoDB.

Unit II - APPLICATION DEVELOPMENT

Document-oriented data – Principles of schema design – Designing an e-commerce data model – Nuts and bolts on databases, collections, and documents. Queries and aggregation – E-commerce queries – MongoDB"s query language – Aggregating orders – Aggregation in detail.

Unit III - UPDATES, ATOMIC OPERATIONS, AND DELETES

A brief tour of document updates – E-commerce updates – Atomic document processing – MongoDB updates and deletes. Indexing and query optimization: Indexing theory – Indexing in practice – Query optimization.

Unit IV – REPLICATION

Overview – Replica sets – Master-slave replication – Drivers and replication. Shading: Overview – A sample shard cluster – Querying and indexing a shard cluster – Choosing a shard key – sharding in production.

Unit V - DEPLOYMENT AND ADMINISTRATION

Deployment – Monitoring and diagnostics – Maintenance – Performance troubleshooting

SUGGESTED READINGS

1. Kyle Banker. (2012). MongoDB in Action. Manning Publications Co.
2. Rick Copeland. (2013). MongoDB Applied Design Patterns, 1st Edition, O"Reilly Media Inc.
3. GautamRege, (2012). Ruby and MongoDB Web Development Beginner's Guide. Packt Publishing Ltd
4. Mike Wilson.. (2013). Building Node Applications with MongoDB and Backbone, O"Reilly Media Inc.
5. David Hows. (2009). The definitive guide to MongoDB, 2nd edition, Apress Publication, 8132230485
6. Shakuntala Gupta Edward. 2016. Practical Mongo DB , 2nd edition, Apress Publications, 2016, ISBN 1484206487

WEBSITES

1. http://www.mongodb.org/about/production-deployments/
2. http://docs.mongodb.org/ecosystem/drivers/
3. http://www.mongodb.org/about/applications/
4. http://www.mongodb.org/

KARPAGAM ACADEMY OF HIGHER EDUCATION
(Deemed to be University)
(Established Under Section 3 of UGC Act, 1956)
(For the candidates admitted from 2019 onwards)
DEPARTMENT OF CS, CA & IT
LESSON PLAN
SUBJECT NAME : MONGODB (19CSP203)
SEMESTER : II

		UNIT I	
SI.NO	**Lecture Duration (Hr)**	**Topics to be covered**	**Support Materials**
1	1	Getting Started	T1:1
2	1	A database for the modern web	T1:3
3	1	A database for the modern web	T1:18
4	1	MongoDB through the JavaScript shell	T1:29
5	1	MongoDB through the JavaScript shell	T1:39, W1
6	1	Writing programs using MongoDB	T1:52
7	1	Recapitulation and Discussion of Important Question	
Total no. of Hours Planned for Unit I			**7**
		UNIT II	

SI.NO	Lecture Duration (Hr)	Topics to be covered	Support Materials
1	1	APPLICATION DEVELOPMENT - Document-oriented data	T1:71
2	1	Principles of schema design , Designing an e-commerce data model	T1:75, R2:169
3	1	Nuts and bolts on databases, collections, and documents	T1:84
4	1	Queries and aggregation- E-commerce queries	T1:99, W2
5	1	MongoDB‟s query language	T1:103
6	1	Aggregating orders, Aggregation in detail	T1:120
7	1	Recapitulation and Discussion of Important Question	
Total Periods Planned for Unit II			**7**
		UNIT III	
SI.NO	**Lecture Duration (Hr)**	**Topics to be covered**	**Support Materials**
1	1	UPDATES, ATOMIC OPERATIONS, AND DELETES - A brief tour of document updates	T1:158
2	1	E-commerce updates	T1:162, R1:193
3	1	Atomic document processing	T1:171

4	1	MongoDB updates and deletes	T1:179
5	1	Indexing and query optimization: Indexing theory	T1:198
6	1	Indexing in practice	T1:207
7	1	Query optimization	W2
7	1	Recapitulation and Discussion of Important Question	7
Total Periods Planned for Unit III			
		UNIT IV	
SI.NO	**Lecture Duration (Hr)**	**Topics to be covered**	**Support Materials**
1	1	REPLICATION- Overview, Replica sets	T1:297
2	1	Master Slave Replication – Drivers and Replication	T1:324
3	1	Shading: Overview	T1:334 R3:312
4	1	A sample shard cluster	T1:343
5	1	Querying and indexing a shard cluster	T1:355,w 2
6	1	Choosing a shard key	T1:359, w2

7	2	Sharding in production	T1:365
8	1	Recapitulation and Discussion of Important Question	
Total Periods Planned for Unit IV			**8**
UNIT V			
SI.NO	**Lecture Duration (Hr)**	**Topics to be covered**	**Support Materials**
1	1	DEPLOYMENT AND ADMINISTRATION - Deployment	
2	1	Monitoring and diagnostics	
3	1	Monitoring and diagnostics	
4	1	Maintenance	
5	1	Maintenance	
6	1	Performance troubleshooting	
7	1	Recapitulation and Discussion of Important Question	
8	1	Discussion of Previous ESE Question Papers	
9	1	Discussion of Previous ESE Question Papers	
10	1	Discussion of Previous ESE Question Papers	
Total Periods Planned for Unit V			**10**
Total Periods			**40**

Text Book

T1	Kyle Banker. (2012). MongoDB in Action. Manning Publications Co.

References

R1	Rick Copeland. (2013). MongoDB Applied Design Patterns, 1st Edition, O‟Reilly Media Inc.
R2	Mike Wilson.(2013). Building Node Applications with MongoDB and Backbone, O‟Reilly Media Inc.
R3	Shakuntala Gupta Edward. 2016. Practical Mongo DB , 2nd edition, Apress Publications, 2016, ISBN 1484206487

Web Sites

w1	http://www.mongodb.org/
w2	W3schools.com/mongodb

UNIT I
SYLLABUS

Getting Started: A database for the modern web – MongoDB through the JavaScript shell – Writing programs using MongoDB.

Getting Started: A database for the modern web

MongoDB is a database management system designed to rapidly develop web appli- cations and internet infrastructure. The data model and persistence strategies are built for high read-and-write throughput and the ability to scale easily with automatic failover. Whether an application requires just one database node or dozens of them, MongoDB can provide surprisingly good performance. If you've experienced difficul- ties scaling relational databases, this may be great news. But not everyone needs to operate at scale. Maybe all you've ever needed is a single database server.

MongoDB stores its information in documents rather than rows. What's a document? Here's an example:

```
{
  _id: 10,
  username: 'peter',
  email: 'pbbakkum@gmail.com'
}
```

This is a pretty simple document; it's storing a few fields of information about a user (he sounds cool). What's the advantage of this model? Consider the case where you'd like to store multiple emails for each user. In the relational world, you might create a separate table of email addresses and the users to which they're associated. MongoDB gives you another way to store these:

```
{
  _id: 10,
```

```
  username: 'peter',
  email: [
    'pbbakkum@gmail
    .com',
    'pbb7c@virginia.ed
    u'
  ]
}
```

MongoDB's document format is based on JSON, a popular scheme for storing arbi- trary data structures. JSON is an acronym for *JavaScript Object Notation*. As you just saw, JSON structures consist of keys and values, and they can nest arbitrarily deep. They're analogous to the dictionaries and hash maps of other programming languages.

A document-based data model can represent rich, hierarchical data structures. It's often possible to do without the multitable joins common to relational databases normalized relational data model, the information for any one product might be divided among dozens of tables.

Built for the internet

The history of MongoDB is brief but worth recounting, for it was born out of a much more ambitious project. In mid-2007, a startup in New York City called 10gen began work on a platform-as-a-service (PaaS), composed of an application server and a data- base, that would host web applications and scale them as needed. Like Google's App Engine, 10gen's platform was designed to handle the scaling and management of hardware and software infrastructure automatically, freeing developers to focus solely on their application code. 10gen ultimately discovered that most developers didn't feel comfortable giving up so much control over their technology stacks, but

users did

10gen has since changed its name to MongoDB, Inc. and continues to sponsor the database's development as an open source project. The code is publicly available and free to modify and use, subject to the terms of its license, and the community at large is encouraged to file bug reports and submit patches. Still, most of MongoDB's core developers are either founders or employees of MongoDB, Inc., and the project's roadmap continues to be determined by the needs of its user community and the overarching goal of creating a database that combines the best features of relational databases and distributed key-value stores.

MongoDB's key features

A database is defined in large part by its data model. In this section, you'll look at the document data model, and then you'll see the features of MongoDB that allow you to operate effectively on that model.

Document data model

MongoDB's data model is document-oriented. If you're not familiar with documents in the context of databases, the concept can be most easily demonstrated by an exam- ple. A JSON document needs double quotes everywhere except for numeric values. The following listing shows the JavaScript version of a JSON document where double quotes aren't necessary.

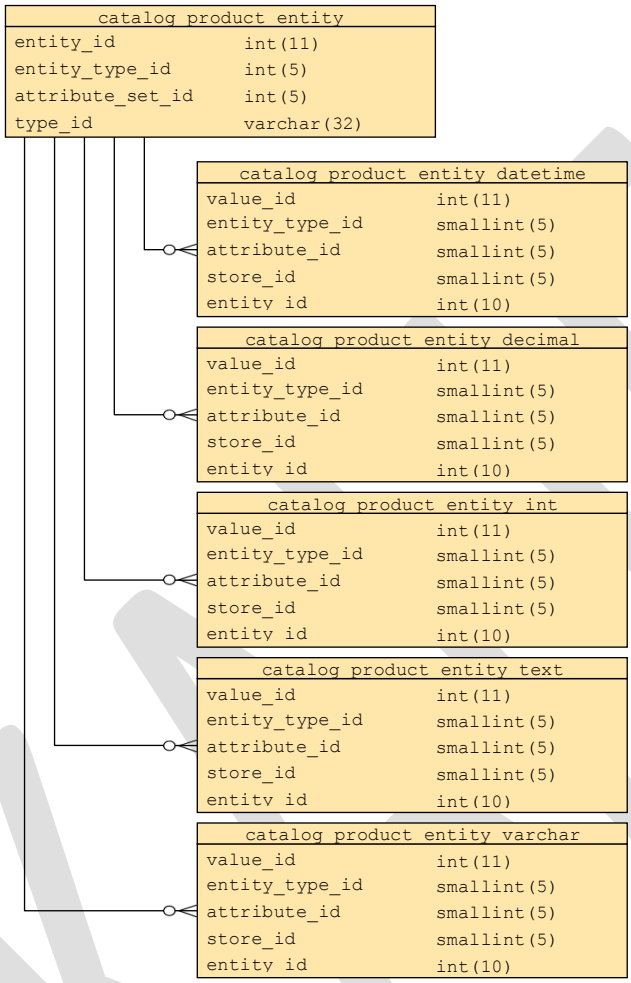
Listing 1.1 A document representing an entry on a social news site

```
{
  _id: ObjectID('4bd9e8e17cefd644108961bb'), title: 'Adventures in
  Databases',
  url: 'http://example.com/databases.txt', author: 'msmith',
```

```
vote_count: 20,
tags: ['databases', 'mongodb', 'indexing'], image: {
  url: 'http://example.com/db.jpg', caption: 'A database.',
  type: 'jpg', size: 75381, data: 'Binary'
},
                                comments: [
                                    {
      user: 'bjones',
      text: 'Interesting article.'
    },
    {
      user: 'sverch',
      text: 'Color me skeptical!'
    }
  ]
}
```

SCHEMA-LESS MODEL ADVANTAGES

This lack of imposed schema confers some advantages. First, your application code, and not the database, enforces the data's structure. This can speed up initial applica- tion development when the schema is changing frequently.



product catalog. There's no way of knowing in advance what attributes a product will have, so the application will need to account for that variability.

Ad hoc queries

Ad hoc queries are easy to take for granted if the only databases you've

ever used have been relational. But not all databases support dynamic queries. For instance, key-value stores are queryable on one axis only: the value's key.

A SQL query would look like this:

```
SELECT * FROM posts
  INNER JOIN posts_tags ON posts.id =
  posts_tags.post_id INNER JOIN tags ON
  posts_tags.tag_id == tags.id
  WHERE tags.text = 'politics' AND posts.vote_count > 10;
```

The equivalent query in MongoDB is specified using a document as a matcher. The special $gtkey indicates the greater-than condition:

```
db.posts.find({'tags': 'politics', 'vote_count': {'$gt': 10}});
```

Indexes

A critical element of ad hoc queries is that they search for values that you don't know when you create the database.

Indexes in MongoDB are implemented as a *B-tree* data structure. B-tree indexes, also used in many relational databases, are optimized for a variety of queries, includ- ing range scans and queries with sort clauses. But WiredTiger has support for log- structured merge-trees (LSM) that's expected to be available in the MongoDB 3.2 pro- duction release.

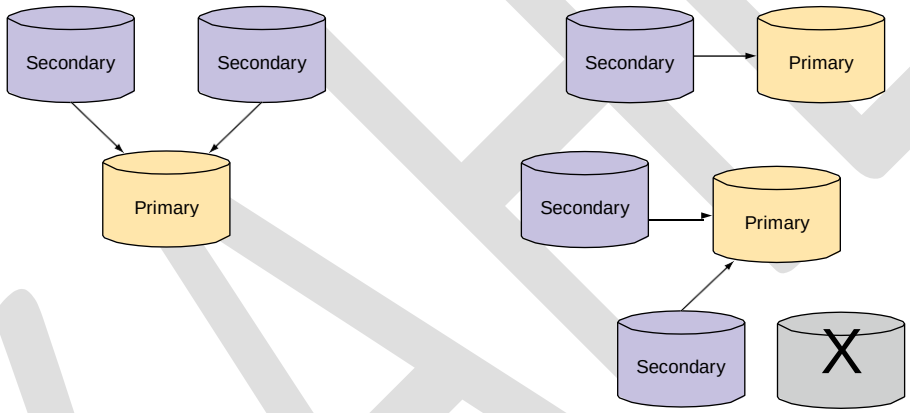
Replication

MongoDB provides database replication via a topology known as a replica set. *Replica sets* distribute data across two or more machines for redundancy and automate failover in the event of server and network outages. Additionally, replication is used to scale database reads. If you

have a read-intensive application, as is commonly the case on the web, it's possible to spread database reads across machines in the replica set cluster.

Speed and durability

To understand MongoDB's approach to durability, it pays to consider a few ideas first. In the realm of database systems there exists an inverse relationship between write speed and durability. *Write speed* can be understood as the volume of inserts, updates, and deletes that a database can process in a given time frame. *Durability* refers to level of assurance that these write operations have been made permanent.

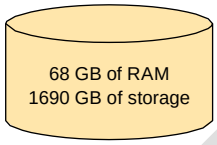


Scaling

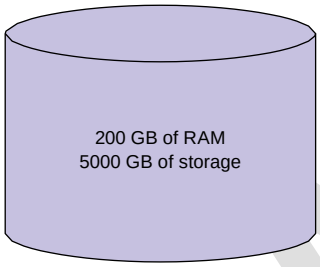
It then makes sense to consider scaling *horizontally*, or *scaling out* (see figure 1.4). Instead of beefing up a single node, scaling horizontally means distributing the data- base across multiple machines. A horizontally scaled architecture can run on many smaller, less expensive machines, often reducing your hosting costs.

MongoDB was designed to make horizontal scaling manageable. It does so via a range-based partitioning mechanism, known as *sharding*, which automatically manages

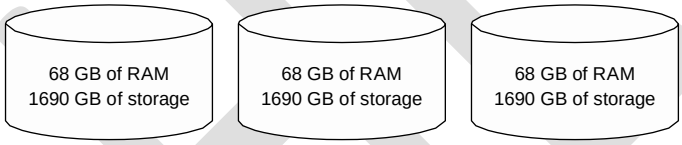
Original database



Scaling up increases the capacity of a single machine.



Scaling out adds more machines of similar size.



MongoDB's core server and tools

MongoDB is written in C++ and actively developed by MongoDB, Inc. The project compiles on all major operating systems, including Mac OS X, Windows, Solaris, and most flavors of Linux. Precompiled binaries are available for each of these platforms at http://mongodb.org. MongoDB is open source and licensed under the GNU-Affero General Public License (AGPL).

Core server

The core database server runs via an executable called mongod (mongodb.exe on Win- dows). The mongod server process receives commands over a network socket using a custom binary protocol. All the data files for a mongodprocess are stored by default in /data/db on Unix-like systems and in c:\data\db on Windows. Command-line tools

MongoDB is bundled with several command-line utilities:

- mongodump and mongorestore—Standard utilities for backing up and restoring a database. mongodump saves the database's data in its native BSON format and thus is best used for backups only; this tool has the advantage of being usable for hot backups, which can easily be restored with mongorestore.
- mongoexport and mongoimport—Export and import JSON, CSV, and TSV[7] data; this is useful if you need your data in widely supported formats. mongoimport can also be good for initial imports of large data sets, although before importing, it's often desirable to adjust the data model to take best advantage of MongoDB. In such cases, it's easier to import the data through one of the drivers using a custom script.
- mongosniff—A wire-sniffing tool for viewing operations sent to the database. It essentially translates the BSON going over the wire to human-readable shell statements.
- mongostat—Similar to iostat, this utility constantly polls MongoDB and the system to provide helpful stats, including the number of operations per second (inserts, queries, updates, deletes, and so on), the amount of virtual memory allocated, and the number of connections to the server.

- mongotop—Similar to top, this utility polls MongoDB and shows the amount of time it spends reading and writing data in each collection.
- mongoperf—Helps you understand the disk operations happening in a running MongoDB instance.

- mongooplog—Shows what's happening in the MongoDB oplog.
- Bsondump—Converts BSON files into human-readable formats including JSON. MongoDB versus other databases

The number of available databases has exploded, and weighing one against another can be difficult. Fortunately, most of these databases fall under one of a few catego- ries. In table 1.1, and in the sections that follow, we describe simple and sophisticated key-value stores, relational databases, and document databases, and show how these compare with MongoDB.

Table 1.1 Database families

	Examples	Data model	Scalability model	Use cases
Simple key-value stores	Memcached	Key-value, where the value is a binary blob.	Variable. Mem-cached can scale across nodes, converting all available RAM into a single, mono-lithic	Caching. Web ops.

			datastore.	
Sophisticated key-value stores	HBase, Cassandra, Riak KV, Redis, CouchDB	Variable. Cassandra uses a key- value structure known as a *col- umn*. HBase and Redis store binary blobs. CouchDB stores JSON documents.	Eventually consistent, multinode distribution for high availability and easy failover.	High-throughput verticals (activity feeds, message queues). Caching. Web ops.
Relational data- bases	Oracle Database, IBM DB2, Micro- soft SQL Server, MySQL, PostgreSQL	Tables.	Vertical scaling. Limited support for clustering and manual partition-ing.	System requiring transactions (banking, finance) or SQL. Normal-ized data model.

RELATIONAL DATABASES

Popular relational databases include MySQL, PostgreSQL, Microsoft SQL Server, Oracle Database, IBM DB2, and so on; some are open-source and some are proprietary. MongoDB and rela- tional databases are both capable of representing a rich data model. Where relational databases use fixed-

schema tables, MongoDB has schema-free documents. Most rela- tional databases support secondary indexes and aggregations.

DOCUMENT DATABASES

Few databases identify themselves as document databases. As of this writing, the clos- est open-source database comparable to MongoDB is Apache's CouchDB. CouchDB's document model is similar, although data is stored in plain text as JSON, whereas MongoDB uses the BSON binary format. Like MongoDB, CouchDB supports secondary indexes; the difference is that the indexes in CouchDB are defined by writing map- reduce functions, a process that's more involved than using the declarative syntax used by MySQL and MongoDB. They also scale differently.

Use cases and production deployments

wEB APPLICATIONS

MongoDB can be a useful tool for powering a high-traffic website. This is the case with *The Business Insider (TBI)*, which has used MongoDB as its primary datastore since January 2008. TBI is a news site, although it gets substantial traffic, serving more than a million unique page views per day.

History of MongoDB

When the first edition of *MongoDB in Action* was released, MongoDB 1.8.x was the most recent stable version, with version 2.0.0 just around the corner. With this second edi- tion, 3.0.x is the latest stable version.[11]

A list of the biggest changes in each of the official versions is shown below. You should always use the most recent version available, if possible, in which case this list isn't particularly useful. If not, this list may help you determine how your version dif- fers from the content of this book. This is by no means an exhaustive list, and because of space constraints, we've listed only the top four or five items for each release.

VERSION 1.8.X (NO LONGER OFFICIALLY SUPPORTED)

- *Sharding*—Sharding was moved from "experimental" to production-ready status.
- *Replica sets*—Replica sets were made production-ready.
- *Replica pairs deprecated*—Replica set pairs are no longer supported by MongoDB, Inc.
- *Geo search*—Two-dimensional geo-indexing with coordinate pairs (2D indexes) was introduced.

VERSION 2.0.X (NO LONGER OFFICIALLY SUPPORTED)

- *Journaling enabled by default*—This version changed the default for new data- bases to enable journaling. Journaling is an important function that prevents data corruption.

- *$and queries*—This version added the $andquery operator to complement the $oroperator.
- *Sparse indexes*—Previous versions of MongoDB included nodes in an index for every document, even if the document didn't contain any of the fields being tracked by the index. Sparse indexing adds only document nodes that have rel- evant fields. This feature significantly reduces index size. In some cases this can improve performance because smaller indexes can result in more efficient use of memory.
- *Replica set priorities*—This version allows "weighting" of replica set members to ensure that your best servers get priority when electing a new primary server.
- *Collection level compact/repair*—Previously you could perform compact/repair only on a database; this enhancement extends it to individual collections.

VERSION 2.2.X (NO LONGER OFFICIALLY SUPPORTED)

- *Aggregation framework*—This version features the first iteration of a facility to make analysis and transformation of data much easier and

more efficient. In many respects this facility takes over where map/reduce leaves off; it's built on a pipeline paradigm, instead of the map/reduce model (which some find diffi- cult to grasp).

- *TTL collections*—Collections in which the documents have a time-limited lifespan are introduced to allow you to create caching models such as those provided by Memcached.
- *DB level locking*—This version adds database level locking to take the place of the global lock, which improves the write concurrency by allowing multiple opera- tions to happen simultaneously on different databases.
- *Tag-aware sharding*—This version allows nodes to be tagged with IDs that reflect their physical location. In this way, applications can control where data is stored in clusters, thus increasing efficiency (read-only nodes reside in the same data center) and reducing legal jurisdiction issues (you store data required to remain in a specific country only on servers in that country).

VERSION 2.4.X (OLDEST STABLE RELEASE)

- *Enterprise version*—The first subscriber-only edition of MongoDB, the Enterprise version of MongoDB includes an additional authentication module that allows the use of Kerberos authentication systems to manage database login data. The free version has all the other features of the Enterprise version.

- *Aggregation framework performance*—Improvements are made in the performance of the aggregation framework to support real-time analytics; chapter 6 explores the Aggregation framework.

- *Text search*—An enterprise-class search solution is integrated as an experimental feature in MongoDB; chapter 9 explores the new text search features.
- *Enhancements to geospatial indexing*—This version includes support for polygon intersection queries and GeoJSON, and features

an improved spherical model supporting ellipsoids.

- *V8 JavaScript engine*—MongoDB has switched from the Spider Monkey JavaScript engine to the Google V8 Engine; this move improves multithreaded operation and opens up future performance gains in MongoDB's JavaScript-based map/ reduce system.

VERSION 2.6.X (STABLE RELEASE)

- *$text queries*—This version added the $text query operator to support text search in normal find queries.
- *Aggregation improvements*—Aggregation has various improvements in this ver- sion. It can stream data over cursors, it can output to collections, and it has many new supported operators and pipeline stages, among many other features and performance improvements.

Additional resources

- *Improved wire protocol for writes*—Now bulk writes will receive more granular and detailed responses regarding the success or failure of individual writes in a batch, thanks to improvements in the way errors are returned over the network for write operations.
- *New update operators*—New operators have been added for update operations, such as $mul, which multiplies the field value by the given amount.
- *Sharding improvements*—Improvements have been made in sharding to better handle certain edge cases. Contiguous chunks can now be merged, and dupli- cate data that was left behind after a chunk migration can be cleaned up auto- matically.
- *Security improvements*—Collection-level access control is supported in this ver- sion, as well as user-defined roles. Improvements have also been made in SSL and x509 support.
- *Query system improvements* —Much of the query system has been refactored. This improves performance and predictability of queries.
- *Enterprise module*—The MongoDB Enterprise module has improvements and extensions of existing features, as well as support for auditing.

VERSION 3.0.X (NEWEST STABLE RELEASE)

- The MMAPv1 storage engine now has support for collection-level locking
- Replica sets can now have up to 50 members.
- Support for the WiredTiger storage engine; WiredTiger is only available in the 64-bit versions of MongoDB 3.0.
- The 3.0 WiredTiger storage engine provides document-level locking and compression.
- Pluggable storage engine API that allows third parties to develop storage engines for MongoDB.
- Improved explain functionality.
- SCRAM-SHA-1 authentication mechanism.
- The ensureIndex() function has been replaced by the createIndex() function and should no longer be used.

This topic covers

- Using CRUD operations in the MongoDB shell
- Building indexes and using `explain()`
- Understanding basic administration
- Getting help

Diving into the MongoDB shell

MongoDB's JavaScript shell makes it easy to play with data and get a tangible sense of documents, collections, and the database's particular query language. Think of the following walkthrough as a practical introduction to MongoDB.

Starting the shell

Follow the instructions in appendix A and you should quickly have a working MongoDB installation on your computer, as well as a running mongod instance. Once you do, start the MongoDB shell by running the mongo executable:

mongo

If the shell program starts successfully, your screen will look like figure 2.1. The shell heading displays the version of MongoDB you're running, along with some additional information about the currently selected database.

```
10:25 $ mongo
MongoDB shell version: 3.0.4
connecting to: test
>
```

Databases, collections, and documents

MongoDB divides collections into separate *databases*. Unlike the usual overhead that databases produce in the SQL world, databases in MongoDB are just namespaces to distinguish between collections. To query MongoDB, you'll need to know the data- base (or namespace) and collection you want to query for documents. If no other database is specified on startup, the shell selects a default database called test. As a way of keeping all the subsequent tutorial exercises under the same namespace, let's start by switching to the tutorial database:

```
> use tutorial
switched to db tutorial
```

The document contains a single key and value for storing Smith's username.

Inserts and queries

To save this document, you need to choose a collection to save it to.

Appropriately enough, you'll save it to the userscollection. Here's how:

```
> db.users.insert({username:
"smith"}) WriteResult({ "nInserted"
: 1 })
```

NOTE Note that in our examples, we'll preface MongoDB shell commands with a >so that you can tell the difference between the command and its output.

You may notice a slight delay after entering this code. At this point, neither the tuto- rial database nor the users collection has been created on disk. The delay is caused by the allocation of the initial data files for both.

If the insert succeeds, you've just saved your first document. In the default MongoDB configuration, this data is now guaranteed to be inserted even if you kill the shell or suddenly restart your machine. You can issue a query to see the new document:

```
> db.users.find()
```

Since the data is now part of the users collection, reopening the shell and running the query will show the same result. The response will look something like this:

```
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

PASS A QUERY PREDICATE

Now that you have more than one document in the collection, let's look at some slightly more sophisticated queries. As before, you can still query for all the docu- ments in the collection:

```
> db.users.find()
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

You can also pass a simple query selector to the find method. A query selector is a document that's used to match against all documents in the collection. To query for all documents where the username is jones, you pass a simple document that acts as your query selector like this:

```
> db.users.find({username: "jones"})
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

Updating documents

```
> db.users.find({username: "smith"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

OPERATOR UPDATE

The first type of update involves passing a document with some kind of operator description as the second argument to the update function. In this section, you'll see an example of how to use the $set operator, which sets a single field to the spec- ified value.

Suppose that user Smith decides to add her country of residence. You can record this with the following update:

```
> db.users.update({username: "smith"}, {$set: {country:
"Canada"}}) WriteResult({ "nMatched" : 1, "nUpserted" : 0,
"nModified" : 1 })
```

Deleting data

If given no parameters, a remove operation will clear a collection of all its docu- ments. To get rid of, say, a foocollection's contents, you enter:

```
> db.foo.remove()
```

You often need to remove only a certain subset of a collection's documents, and for that, you can pass a query selector to the remove() method. If you want to remove all users whose favorite city is Cheyenne, the expression is straightforward:

```
> db.users.remove({"favorites.cities":
"Cheyenne"}) WriteResult({ "nRemoved" : 1 })
```

Note that the remove() operation doesn't actually delete the collection; it merely removes documents from a collection. You can think of it as being analogous to SQL's DELETE command.

If your intent is to delete the collection along with all of its indexes, use the drop() method:

```
> db.users.drop()
```

Basic administration

Getting database information

showdbsprints a list of all the databases on the system:

```
> show dbs
admin    (empty)
local
          0.078
GB tutorial
0.078GB
```

showcollections displays a list of all the collections defined on the current data- base.[4] If the tutorial database is still selected, you'll see a list of the collections you worked with in the preceding tutorial:

```
> show
collections
numbers
system.indexes
users
```

The one collection that you may not recognize is system.indexes. This is a special collection that exists for every database. Each entry in

system.indexes defines an index for the database, which you can view using the getIndexes()method, as you saw earlier.

But MongoDB 3.0 deprecates direct access to the system.indexes collections; you should use createIndexes and listIndexes instead. The getIndexes() Java- Script method can be replaced by the db.runCommand({"listIndexes": "numbers"}) shell command.

For lower-level insight into databases and collections, the stats()method proves useful. When you run it on a database object, you'll get the following output:

```
> db.stats()
{
    "db" : "tutorial",
    "collections" : 4,
    "objects" : 20010,
    "avgObjSize" : 48.0223888055972,
    "dataSize" : 960928,
    "storageSize" : 2818048,
    "numExtents" : 8,
    "indexes" : 3,
    "indexSize" : 1177344,
    "fileSize" : 67108864,
    "nsSizeMB" : 16,
    "extentFreeList" :
    {
        "num" : 0,
        "totalSize" : 0
    },
    "dataFileVersion" :
        { "major" : 4,
        "minor" : 5
    },
    "ok" : 1
}
```

This topic covers

- Introducing the MongoDB API through Ruby
- Understanding how the drivers work
- Using the BSON format and MongoDB network protocol
- Building a complete sample application

MongoDB through the Ruby lens

Installing and connecting

Once you have RubyGems installed, run:

```
gem install mongo
```

You'll start by connecting to MongoDB. First, make sure that mongodis running by running the mongoshell to ensure you can connect. Next, create a file called connect.rb and enter the following code:

```
require
'rubygems'
require 'mongo'

$client = Mongo::Client.new([ '127.0.0.1:27017' ], :database =>
'tutorial') Mongo::Logger.logger.level = ::Logger::ERROR
$users =
$client[:users] puts
'connected!'
```

The first two require statements ensure that you've loaded the driver. The next three lines instantiate the client to localhost and connect to the tutorial database, store a ref- erence to the userscollection in the $usersvariable, and print the string connected!. We place a $in front of each variable to make it global so that it'll be accessible out- side of the connect.rb script. Save

the file and run it:

```
$ ruby connect.rb
D, [2015-06-05T12:32:38.843933 #33946] DEBUG -- : MONGODB | Adding
    127.0.0.1:27017 to the cluster. | runtime: 0.0031ms
D, [2015-06-05T12:32:38.847534 #33946] DEBUG -- : MONGODB |
COMMAND |
    namespace=admin.$cmd selector={:ismaster=>1} flags=[]
    limit=-1 skip=0 project=nil | runtime: 3.4170ms
connected!
```

Inserting documents in Ruby

To run interesting MongoDB queries you first need some data, so let's create some (this is the C in CRUD). All of the MongoDB drivers are designed to use the most natu- ral document representation for their language. In JavaScript, JSON objects are the obvious choice, because JSON is a document data structure; in Ruby, the hash data structure makes the most sense. The native Ruby hash differs from a JSON object in only a couple of small ways; most notably, where JSON separates keys and values with a colon, Ruby uses a hash rocket (=>).[2]

Here's an example:

```
$ irb -r ./connect.rb
irb(main):017:0> id = $users.insert_one({"last_name" => "mtsouk"})
=> #<Mongo::Operation::Result:70275279152800
documents=[{"ok"=>1, "n"=>1}]> irb(main):014:0>
$users.find().each do |user|
irb(main):015:1* puts user
irb(main):016:1> end
```

```
{"_id"=>BSON::ObjectId('55e3ee1c5ae119511d000000'),
"last_name"=>"knuth"}
{"_id"=>BSON::ObjectId('55e3f13d5ae119516a000000'),
"last_name"=>"mtsouk"}
=> #<Enumerator: #<Mongo::Cursor:0x70275279317980
@view=#<Mongo::Collection::View:0x70275279322740
```

namespace='tutorial.users @selector={} @options={}>>:each>

Updates and deletes

$users.find({"last_name" => "smith"}).update_one({"$set" => {"city" => "Chicago"}})

This update finds the first user with a last_name of smith and, if found, sets the value of city to Chicago. This update uses the $set operator. You can run a query to show the change:

$users.find({"last_name" => "smith"}).to_a

Database commands

First, you instantiate a Ruby database object referencing the admin database. You then pass the command's query specification to the command method:

```
$admin_db = $client.use('admin')
$admin_db.command({"listDatabases" => 1})
```

Note that this code still depends on what we put in the connect.rb script above because it expects the MongoDB connection to be in $client. The response is a Ruby hash listing all the existing databases and their sizes on disk:

```
#<Mongo::Operation::Result:70112905054200 documents=[{"databases"=>[
{
    "name"=>"local",
    "sizeOnDisk"=>83886
    080.0, "empty"=>false
},
{
    "name"=>"tutorial",
"sizeOnDisk"=>83886
080.0, "empty"=>false
```

```
},
{
    "name"=>"admin",
    "sizeOnDisk"=>1.0,
    "empty"=>true
}], "totalSize"=>167772160.0, "ok"=>1.0}]>
 => nil
```

How the drivers work

All MongoDB drivers perform three major functions. First, they generate Mon- goDB object IDs. These are the default values stored in the _id field of all documents. Next, the drivers convert any language-specific representation of documents to and from BSON, the binary data format used by MongoDB. In the previous examples, the driver serializes all the Ruby hashes into BSON and then deserializes the BSON that's returned from the database back to Ruby hashes.

The drivers' final function is to communicate with the database over a TCP socket using the MongoDB wire protocol. The details of the protocol are beyond the scope of this discussion. But the style of socket communication, in particular whether writes on the socket wait for a response, is important, and we'll explore the topic in this section.

Object ID generation

Every MongoDB document requires a primary key. That key, which must be unique for all documents in each collection, is stored in the document's _id field. Developers are free to use their own custom values as the _id, but when not provided, a MongoDB object ID will be used. Before sending a document to the server, the driver checks whether the _idfield is present. If the field is missing, an object ID will be generated and stored as _id.

MongoDB object IDs are designed to be globally unique, meaning they're guaran- teed to be unique within a certain context. How can this be guaranteed? Let's exam- ine this in more detail.

You've probably seen object IDs in the wild if you've inserted documents into MongoDB, and at first glance they appear to be a string of mostly random text, like 4c291856238d3b19b2000001.

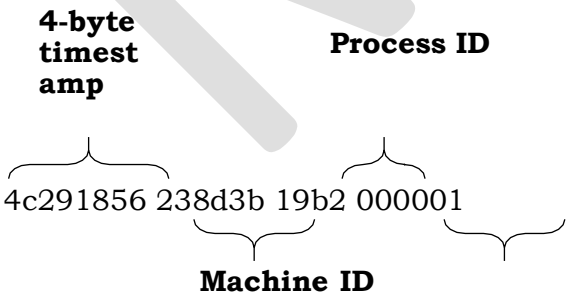


Figure 3.1 MongoDB object ID format

representation of 12 bytes, and actually stores some useful information. These bytes have a specific structure, as illustrated in figure 3.1.

The most significant four bytes carry a standard Unix (epoch) timestamp[3]. The next three bytes store the machine ID, which is followed by a two-byte process ID. The final three bytes store a process-local counter that's incremented each time an object ID is generated. The counter means that ids generated in the same process and second won't be duplicated.

Why does the object ID have this format? It's important to understand that these IDs are generated in the driver, not on the server. This is different than many RDBMSs, which increment a primary key on the server, thus creating a bottleneck for the server generating the key. If more than one driver is generating IDs and inserting docu- ments, they need a way of creating unique identifiers without talking to each other. Thus, the timestamp, machine ID, and process ID are included in the identifier itself to make it extremely unlikely that IDs will overlap.

You may already be considering the odds of this happening. In practice, you would encounter other limits before inserting documents at the rate required to overflow the counter for a given second (2^{24} million per second). It's slightly more conceivable (though still unlikely) to imagine that ifyou had many drivers distributed across many machines, two machines could have the same machine ID. For example, the Ruby driver uses the following:

@@machine_id = Digest::MD5.digest(Socket.gethostname)[0, 3]

For this to be a problem, they would still have to have started the MongoDB driver's process with the same process ID, and have the same counter value in a given second. In practice, don't worry about duplication; it's extremely unlikely.

One of the incidental benefits of using MongoDB object IDs is that they include a timestamp. Most of the drivers allow you to extract the timestamp, thus providing the document creation time, with resolution to the nearest second, for free. Using the Ruby

driver, you can call an object ID's generation_time method to get that ID's creation time as a Ruby Time object:

```
irb> require 'mongo'
irb> id = BSON::ObjectId.from_string('4c291856238d3b19b2000001')
=> BSON::ObjectId('4c291856238d3b19b2000001')
irb> id.generation_time
=> 2010-06-28 21:47:02 UTC
```

Naturally, you can also use object IDs to issue range queries on object creation time. For instance, if you wanted to query for all documents created during June 2013, you could create two object IDs whose timestamps encode those dates and then issue a range query on _id. Because Ruby provides methods for generating object IDs from any Time object, the code for doing this is trivial:[4]

```
jun_id = BSON::ObjectId.from_time(Time.utc(2013, 6, 1))
jul_id = BSON::ObjectId.from_time(Time.utc(2013, 7,
1)) @users.find({'_id' => {'$gte' => jun_id, '$lt' =>
jul_id}})
```

As mentioned before, you can also set your own value for _id. This might make sense in cases where one of the document's fields is important and always unique. For instance, in a collection of users you could store the username in _id rather than on object ID. There are advantages to both ways, and it comes down to your preference as a developer.

Building a simple application

Next you'll build a simple application for archiving and displaying Tweets. You can imagine this being a component in a larger application that allows users to keep tabs on search terms relevant to their businesses. This example will demonstrate how easy it is to consume JSON from an API like Twitter's

and convert that to MongoDB docu- ments. If you were doing this with a relational database, you'd have to devise a schema in advance, probably consisting of multiple tables, and then declare those tables. Here, none of that's required, yet you'll still preserve the rich structure of the Tweet documents, and you'll be able to query them effectively.

Let's call the app TweetArchiver. TweetArchiver will consist of two components: the archiver and the viewer. The archiver will call the Twitter search API and store the relevant Tweets, and the viewer will display the results in a web browser.

Setting up

This application requires four Ruby libraries. The source code repository for this chap- ter includes a file called Gemfile, which lists these gems. Change your working directory

```
gem install bundler bundle install
```

This will ensure the bundler gem is installed. Next, install the other gems using Bundler's package management tools. This is a widely used Ruby tool for ensuring that the gems you use match some predetermined versions: the versions that match our code examples.

Our Gemfile lists the mongo, twitter, bson and sinatra gems, so these will be installed. The mongo gem we've used already, but we include it to be sure we have the right version. The twitter gem is useful for communicating with the Twitter API.

We provide the source code for this example separately, but introduce it gradually to help you understand it. We recommend you experiment and try new things to get the most out of the example.

It'll be useful to have a configuration file that you can share between the archiver and viewer scripts. Create a file called config.rb (or copy it from the source code) that looks like this:

```
DATABASE_HOST =
'localhost'
```

```
DATABASE_PORT = 27017
DATABASE_NAME = "twitter-archive" COLLECTION_NAME = "tweets"
TAGS = ["#MongoDB", "#Mongo"]

CONSUMER_KEY = "replace me"
CONSUMER_SECRET = "replace me" TOKEN = "replace me" TOKEN_SECRET = "replace me"
```

First you specify the names of the database and collection you'll use for your applica- tion. Then you define an array of search terms, which you'll send to the Twitter API.

Twitter requires that you register a free account and an application for accessing the API, which can be accomplished at http://apps.twitter.com. Once you've regis- tered an application, you should see a page with its authentication information, per- haps on the API keys tab. You will also have to click the button that creates your access token. Use the values shown to fill in the consumer and API keys and secrets.

Gathering data

The next step is to write the archiver script. You start with a TweetArchiver class. You'll instantiate the class with a search term. Then you'll call the update method on the TweetArchiver instance, which issues a Twitter API call, and save the results to a MongoDB collection.

Let's start with the class's constructor:

```
def initialize(tag)
  connection = Mongo::Connection.new(DATABASE_HOST, DATABASE_PORT) db = connection[DATABASE_NAME]
  @tweets = db[COLLECTION_NAME]
  @tweets.ensure_index([['tags', 1], ['id', -1]])
```

```
  @tag = tag
  @tweets_found = 0


  @client = Twitter::REST::Client.new do
    |config| config.consumer_key        =
    API_KEY config.consumer_secret    =
    API_SECRET config.access_token    =
    ACCESS_TOKEN
    config.access_token_secret =
    ACCESS_TOKEN_SECRET
  en
d
end
```

The initialize method instantiates a connection, a database object, and the collec- tion object you'll use to store the Tweets.

You're creating a compound index on tags ascending and id descending. Because you're going to want to query for a particular tag and show the results from newest to oldest, an index with tags ascending and id descending will make that query use the index both for filtering results and for sorting them. As you can see here, you indicate index direction with 1 for *ascending* and -1 for *descending*. Don't worry if this doesn't make sense now—we discuss indexes with much greater depth in chapter 8.

You're also configuring the Twitter client with the authentication information from config.rb. This step hands these values to the Twitter gem, which will use them when calling the Twitter API. Ruby has somewhat unique syntax often used for this sort of con- figuration; the configvariable is passed to a Ruby block, in which you set its values.

In the future, Twitter may change its API so that different values are returned, which will likely require a schema change if you want to store these additional values. Not so with MongoDB. Its schema-less design allows you to save the document you get from the Twitter API without worrying about the exact format.

The Ruby Twitter library returns Ruby hashes, so you can pass these directly to your MongoDB collection object. Within your TweetArchiver, you add the following instance method:

```
def save_tweets_for(term)
  @client.search(term).each do
  |tweet|
    @tweets_found += 1
    tweet_doc =
    tweet.to_h
    tweet_doc[:tags] =
    term
    tweet_doc[:_id] =
    tweet_doc[:id]
    @tweets.insert_one(tweet_do
    c)
  en
d
end
```

Before saving each Tweet document, make two small modifications. To simplify later queries, add the search term to a tagsattribute. You also set the _id field to the ID of the Tweet, replacing the primary key of your collection and ensuring that each Tweet is added only once. Then you pass the modified document to the savemethod.

To use this code in a class, you need some additional code. First, you must config- ure the MongoDB driver so that it connects to the correct mongodand uses the desired database and collection. This is simple code that you'll replicate often as you use MongoDB. Next, you must configure the Twitter gem with your developer credentials. This step is necessary because Twitter restricts its API to registered developers. The next listing

also provides an update method, which gives the user feedback and calls save_tweets_for.

Listing 3.1 archiver.rb—A class for fetching Tweets and archiving them in MongoDB

```
$LOAD_PATH << File.dirname(_
FILE_) require 'rubygems'
require 'mongo'
require 'twitter'
require 'config'

class TweetArchiver

def initialize(tag)
client =
Mongo::Client.new(["#{DATABASE_HOST}:#{DATABASE_PORT}"],:
database => "#{DATABASE_NAME}")
@tweets =
client["#{COLLECTION_NAME}"]
@tweets.indexes.drop_all
@tweets.indexes.create_many([
{ :key => { tags: 1 }},
{ :key => { id: -1 }}
])
@tag = tag
@tweets_found = 0

    client = Twitter::REST::Client.new do |config| config.consumer_key =
      "#{API_KEY}" config.consumer_secret = "#{API_SECRET}"
      config.access_token = "#{ACCESS_TOKEN}"
      config.access_token_secret = "#{ACCESS_TOKEN_SECRET}"
    end end
```

Configure the Twitter client using the values found in config.rb.

```
def update
  puts "Starting Twitter search for '#{@tag}'..." save_tweets_for(@tag)
  print "#{@tweets_found} Tweets saved.\n\n" end

private
```

A user facing method to wrap save_tweets_for

```
  def save_tweets_for(term) @client.search(term).each do |tweet|
      @tweets_found += 1 tweet_doc = tweet.to_h tweet_doc[:tags] =
      term
      tweet_doc[:_id] = tweet_doc[:id] @tweets.insert_one(tweet_doc)
    end end
end
```

Search with the Twitter client and save the results to Mongo.

All that remains is to write a script to run the TweetArchiver code against each of the search terms. Create a file called update.rb (or copy it from the provided code) con- taining the following:

```
$LOAD_PATH << File.dirname(
FILE_) require 'config'
require 'archiver'

TAGS.each do |tag|
  archive =
```

```
  TweetArchiver.new(tag)
  archive.update
end
```

Next, run the update script:

```
ruby update.rb
```

You'll see some status messages indicating that Tweets have been found and saved. You can verify that the script works by opening the MongoDB shell and querying the col- lection directly:

```
> use twitter-archive
switched to db twitter-archive
> db.tweets.coun
t() 30
```

What's important here is that you've managed to store Tweets from Twitter searches in only a few lines of code.[5] Next comes the task of displaying the results.

Viewing the archive

You'll use Ruby's Sinatra web framework to build a simple app to display the results. Sinatra allows you to define the endpoints for a web application and directly specify the response. Its power lies in its simplicity. For example, the content of the index page for your application can be specified with the following:

```
get '/' do
  "respons
  e"
end
```

This code specifies that GET requests to the / endpoint of your application return the

value of response to the client. Using this format, you can write full web applications with many endpoints, each of which can execute arbitrary Ruby code before returning a response. You can find more information, including Sinatra's full documentation, at http://sinatrarb.com.

We'll now introduce a file called viewer.rb and place it in the same directory as the other scripts. Next, make a subdirectory called views, and place a file there called tweets.erb. After these steps, the project's file structure should look like this:

- config.rb
- archiver.rb
- update.rb
- viewer.rb
- /views
  - tweets.erb

Again, feel free to create these files yourself or copy them from the code examples. Now edit viewer.rb with the code in the following listing.

Listing 3.2 viewer.rb—Sinatra application for displaying the Tweet archive

```
$LOAD_PATH << File.dirname(_
FILE_) require 'rubygems'
require 'mongo'
require 'sinatra'
require 'config'
require 'open-uri'

configure do
  client = Mongo::Client.new(["#{DATABASE_HOST}:#{DATABASE_PORT}"],
  :database
      => "#{DATABASE_NAME}")
```

Required libraries

```
  TWEETS = client["#{COLLECTION_NAME}"]
end
```

```
get '/' do
  if params['tag']
    selector = {:tags => params['tag']} else
    selector = {} end
```

Instantiate collection
c for tweets

d Dynamically build query selector...

...or use
e blank selector

```
  @tweets = TWEETS.find(selector).sort(["id", -1]) erb :tweets
end
```

The first lines require the necessary libraries along with your config file B. Next there's a configuration block that creates a connection to MongoDB and stores a refer- ence to your tweets collection in the constant TWEETSc.

The real meat of the application is in the lines beginning with get'/'do. The code in this block handles requests to the application's root URL. First, you build your

query selector. If a tagsURL parameter has been provided, you create a query selector that restricts the result set to the given tags d. Otherwise, you create a blank selector, which returns all documents in the collection e. You then

issue the query f. By now, you should know that what gets assigned to the @tweetsvariable isn't a result set but a
cursor. You'll iterate over that cursor in your view.

The last line g renders the view file tweets.erb (see the next listing).

Listing 3.3 tweets.erb—HTML with embedded Ruby for rendering the Tweets

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <style>
    body
    {
      width: 1000px;
      margin: 50px
      auto;
      font-family: Palatino,
      serif; background-color:
      #dbd4c2; color: #555050;
    }
    h2 {
      margin-top: 2em;
      font-family: Arial, sans-
      serif; font-weight: 100;
    }
  </style>
</head>
<body>
<h1>Tweet Archive</h1>
<% TAGS.each do |tag| %>
  <a href="/?tag=<%= URI::encode(tag) %>"><%= tag %></a>
<% end %>
```

```
<% @tweets.each do |tweet| %>
  <h2><%= tweet['text'] %></h2>
  <p>
    <a href="http://twitter.com/<%= tweet['user']['screen_name'] %>">
      <%= tweet['user']['screen_name'] %>
    </a>
    on <%= tweet['created_at'] %>
  </p>
  <img src="<%= tweet['user']['profile_image_url'] %>" width="48" />
<% end %>
</body>
</html>
```

Most of the code is just HTML with some ERB (embedded Ruby) mixed in. The Sinatra app runs the tweets.erb file through an ERB processor and evaluates any Ruby code between <% and %> in the context of the application.

The important parts come near the end, with the two iterators. The first of these cycles through the list of tags to display links for restricting the result set to a given tag.

The second iterator, beginning with the @tweets.eachcode, cycles through each Tweet to display the Tweet's text, creation date, and user profile image. You can see results by running the application:

```
$ ruby viewer.rb
```

If the application starts without error, you'll see the standard Sinatra startup message that looks something like this:

```
$ ruby viewer.rb
[2013-07-05 18:30:19] INFO WEBrick 1.3.1
[2013-07-05 18:30:19] INFO ruby 1.9.3 (2012-04-20) [x86_64-darwin10.8.0]
== Sinatra/1.4.3 has taken the stage on 4567 for development with
     backup from WEBrick
[2013-07-05 18:30:19] INFO WEBrick::HTTPServer#start: pid=18465
```

port=4567

You can then point your web browser to http://localhost:4567. The page should look something like the screenshot in figure 3.2. Try clicking on the links at the top of the screen to narrow the results to a particular tag.

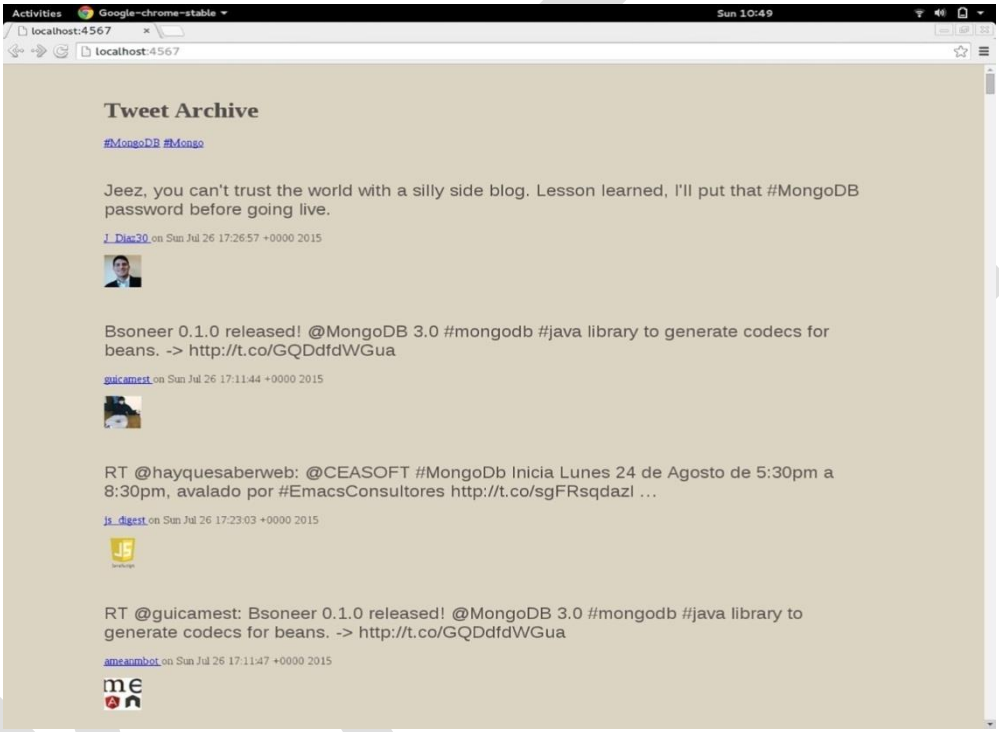


Figure 3.2 Tweet Archiver output rendered in a web browser

69

UNIT II
SYLLABUS

Application Development: Document-oriented data – Principles of schema design – Designing an e-commerce data model – Nuts and bolts on databases, collections, and documents. Queries and aggregation – E-commerce queries – MongoDB"s query language – Aggregating orders – Aggregation in detail.

This topic covers

- Schema design
- Data models for e-commerce
- Nuts and bolts of databases, collections, and documents

Principles of schema design

Database schema design is the process of choosing the best representation for a data set, given the features of the database system, the nature of the data, and the applica- tion requirements. The principles of schema design for relational database systems are well established. With RDBMSs, you're encouraged to shoot for a normalized data model,[1] which helps to ensure generic query ability and avoid updates to data that might result in inconsistencies. Moreover, the established patterns prevent developers from wondering how to model, say, one-to-many and many-to-many relationships.

What are your application access patterns?

What's the basic unit of data?

What are the capabilities of your database?

What makes a good unique id or primary key for a record?

Designing an e-commerce data model

E-commerce has the advantage of including a large number of famil- iar data modeling patterns. Plus, it's not hard to imagine how products, categories, product reviews, and orders are typically modeled in an RDBMS.

E-commerce has typically been done with RDBMSs for a couple of reasons. The first is that e-commerce sites generally require transactions, and transactions are an RDBMS staple.

The second is that, until recently, domains that require rich data models and sophisticated queries have been assumed to fit best within the realm of the RDBMS.

Schema basics

Products and categories are the mainstays of any e-commerce site. Products, in a nor- malized RDBMS model, tend to require a large number of tables. There's a table for basic product information, such as the name and SKU, but there will be other tables to relate shipping information and pricing histories.

This multitable schema will be facil- itated by the RDBMS's ability to join tables.

More concretely, listing 4.1 shows a sample product from a gardening store. It's advis- able to assign this document to a variable before inserting it to the database using db.products.insert(yourVariable) to be able to run the queries discussed over the next several pages.

Listing 4.1 A sample product document

```
{
  _id: ObjectId("4c4b1476238d3b4dd5003981"), slug: "wheelbarrow-9092",
  sku: "9092",
  name: "Extra Large Wheelbarrow", description: "Heavy duty
  wheelbarrow...", details: {
    weight: 47, weight_units: "lbs", model_num: 4039283402,
    manufacturer: "Acme", color: "Green"
  },
  total_reviews: 4,
  average_review: 4.5, pricing: {
                          retail: 589700,
                          sale: 489700,
```

```
    },
    price_history: [
                        {
                         retail: 529700,
                         sale: 429700,
        start: new Date(2010, 4, 1),
        end: new Date(2010, 4, 8)
                        },

                        {
                         retail: 529700,
                         sale: 529700,
        start: new Date(2010, 4, 9),
        end: new Date(2010, 4, 16)
```

b **Unique object ID**

c **Unique slug**

Nested
d **document**

e **One-to-many relationship**

```
      },
    ],

    primary_category: ObjectId("6a5b1476238d3b4dd5000048"),
    category_ids: [
      ObjectId("6a5b1476238d3b4dd5000048"),
      ObjectId("6a5b1476238d3b4dd5000049")
    ],
    main_cat_id: ObjectId("6a5b1476238d3b4dd5000048"),
    tags: ["tools", "gardening", "soil"],
}
```

Many-to-many relationship

ONE-TO-MANY RELATIONSHIPS

This is a one-to-many relationship, since a product only has one primary category, but a category can be the primary for many products.

MANY-TO-MANY RELATIONSHIPS

MongoDB doesn't support joins, so you need a different many-to-many strategy. We've defined a field called category_ids f containing an array of object IDs. Each object ID acts as a pointer to the _id field of some category document.

A RELATIONSHIP STRUCTURE

The next listing shows a sample category document. You can assign it to a new variable and insert it into the categories collection using db.categories.insert(newCategory). This will help you using it in forthcoming queries without having to type it again.

Listing 4.2 A category document

```
{
  _id:
  ObjectId("6a5b1476238d3b4dd50000
  48"), slug: "gardening-tools",
  name: "Gardening Tools",
  description: "Gardening gadgets galore!",
  parent_id:
  ObjectId("55804822812cb336b78728f9"),
    ancestors: [

        {
          name: "Home",
          _id:
          ObjectId("558048f0812cb336b78728f
          a"), slug: "home"
        },
```

```
        {
          name: "Outdoors",
          _id:
          ObjectId("55804822812cb336b78728
          f9"), slug: "outdoors"
        }
    ]
}
```

Nuts and bolts: On databases, collections, and documents

Databases

A database is a namespace and physical grouping of collections and their indexes. In this section, we'll discuss the details of creating and deleting databases. We'll also jump down a level to see how MongoDB allocates space for individual databases on the filesystem.

MANAGING DATABASES

There's no explicit way to create a database in MongoDB. Instead, a database is cre- ated automatically once you write to a collection in that database. Have a look at this Ruby code:

```
connection = Mongo::Client.new( [ '127.0.0.1:27017' ], :database =>
'garden' ) db = connection.database
```

Recall that the JavaScript shell performs this connection when you start it, and then allows you to select a database like this:

```
use garden
```

Assuming that the database doesn't exist already, the database has yet to be created on disk even after executing this code. All you've done is instantiate an instance of the class Mongo::DB, which represents a MongoDB database. Only when you write to a col- lection are the data files created. Continuing on in Ruby,

```
products = db['products']
products.insert_one({:name => "Extra Large Wheelbarrow"})
```

When you call insert_one on the products collection, the driver tells MongoDB to insert the product document into the garden.products collection. If that collec- tion doesn't exist, it's created; part of this involves allocating the garden database on disk.

You can delete all the data in this collection by calling:

```
products.find({}).delete_many
```

This removes all documents which match the filter {}, which is all documents in the collection. This command doesn't remove the collection itself; it only empties it. To remove a collection entirely, you use the drop method, like this:

```
products.drop
```

To delete a database, which means dropping all its collections, you issue a special com- mand. You can drop the garden database from Ruby like so:

```
db.drop
```

From the MongoDB shell, run the dropDatabase()method using JavaScript:

```
use garden
db.dropDatabas
e();
```

Be careful when dropping databases; there's no way to undo this operation since it erases the associated files from disk. Let's look in more detail at how databases store their data.

DATA FILES AND ALLOCATION

When you create a database, MongoDB allocates a set of data files on disk. All collec- tions, indexes, and other metadata for the database are stored in these files. The data files reside in whichever directory you designated as the dbpath when starting mongod. When left unspecified, mongod stores all its files in /data/db.[3] Let's see how this direc- tory looks after creating the garden database:

```
$ cd /data/db
$ ls -lah
drwxr-xr-x  81 pbakkum admin  2.7K 1 Jul 10:42 .
drwxr-xr-x   5 root    admin  170B 19 Sep 2012 ..
-rw-------   1 pbakkum admin   64M 1 Jul 10:43 garden.0
-rw-------   1 pbakkum admin  128M 1 Jul 10:42 garden.1
-rw-------   1 pbakkum admin   16M 1 Jul 10:43 garden.ns
-rwxr-xr-x   1 pbakkum admin    3B 1 Jul 08:31 mongod.lock
```

Collections

Collections are containers for structurally or conceptually similar documents. Here,

MANAGING COLLECTIONS

As you saw in the previous section, you create collections implicitly by inserting docu- ments into a particular namespace. But because more than one collection type exists, MongoDB also provides a command for creating collections. It provides this com- mand from the JavaScript shell:

```
db.createCollection("users")
```

When creating a standard collection, you have the option of preallocating a specific number of bytes. This usually isn't necessary but can be done like this in the Java- Script shell:

```
db.createCollection("users", {size: 20000})
```

Collection names may contain numbers, letters, or . characters, but must begin with a letter or number. Internally, a collection name is identified by its namespace name, which includes the name of the database it belongs to. Thus, the products collection is technically referred to as garden.productswhen referenced in a mes- sage to or from the core server. This fully qualified collection name can't be longer than 128 characters.

It's sometimes useful to include the . character in collection names to provide a kind of virtual namespacing. For instance, you can imagine a series of collections with titles like the following:

products.categor
ies
products.images
products.reviews

Keep in mind that this is only an organizational principle; the database treats collec- tions named with a .like any other collection.

Collections can also be renamed. As an example, you can rename the products col- lection with the shell's renameCollectionmethod:

```
db.products.renameCollection("store_products")
```

Listing 4.6 Simulating the logging of user actions to a capped collection

```
require 'mongo'


VIEW_PRODUCT = 0      # action type constants ADD_TO_CART = 1
CHECKOUT        = 2
PURCHASE        = 3



client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'garden')
```

```
    client[:user_actions].drop
actions = client[:user_actions, :capped => true, :size => 16384]
actions.create

500.times do |n|          # loop 500 times, using n as the iterator doc = {
    :username => "kbanker",
    :action_code => rand(4), # random value between 0 and 3, inclusive
    :time => Time.now.utc,
                                                          :n => n
                                                        }
  actions.insert_one(d
oc) end
```

First, you create a 16 KB capped collection called user_actionsusing client.[6] Next, you insert 500 sample log documents B. Each document contains a username, an action code (represented as a random integer from 0 through 3), and a timestamp. You've included an incrementing integer, n, so that you can identify which documents have aged out. Now you'll query the collection from the shell:

```
> use garden
> db.user_actions.coun
t(); 160
```

Even though you've inserted 500 documents, only 160 documents exist in the collec- tion.[7] If you query the collection, you'll see why:

```
db.user_actions.find().pretty();
{
    "_id" :
    ObjectId("51d1c69878b10e1a0e000040"
    ), "username" : "kbanker",
    "action_code" : 3,
    "time" : ISODate("2013-07-
    01T18:12:40.443Z"), "n" : 340
}
```

"_id" :
ObjectId("51d1c69878b10e1a0e000041"),
"username" : "kbanker",
"action_code" : 2,
"time" : ISODate("2013-07-
01T18:12:40.444Z"), "n" : 341

"_id" :
ObjectId("51d1c69878b10e1a0e000042"),
"username" : "kbanker",
"action_code" : 2,
"time" : ISODate("2013-07-
01T18:12:40.445Z"), "n" : 342

TTL COLLECTIONS

MongoDB also allows you to expire documents from a collection after a certain amount of time has passed. These are sometimes called time-to-live (TTL) collections, though this functionality is actually implemented using a special kind of index. Here's how you would create such a TTL index:

```
>
>
>
> db.reviews.createIndex({time_field: 1}, {expireAfterSeconds: 3600})
```

This command will create an index on time_field.

between time_field and the current time is greater than your expireAfterSeconds setting, then the document will be removed automatically. In this example, review documents will be deleted after an hour.

Using a TTL index in this way assumes that you store a timestamp in time_field.

Here's an example of how to do this:

```
> db.reviews.insert({
    time_field: new
    Date(),
    …
  })
```

SYSTEM COLLECTIONS

Part of MongoDB's design lies in its own internal use of collections. Two of these spe- cial system collections are system.namespaces and system.indexes. You can query the former to see all the namespaces defined for the current database:

```
> db.system.namespaces.find();
{ "name" : "garden.system.indexes" }
{ "name" : "garden.products.$_id_" }
{ "name" : "garden.products" }
{ "name" : "garden.user_actions.$_id_" }
{ "name" : "garden.user_actions", "options" : { "create" :
"user_actions", "capped" : true, "size" : 1024 } }
```

The first collection, system.indexes, stores each index definition for the current database. To see a list of indexes you've defined for the garden database, query the collection:

```
> db.system.indexes.find();
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "garden.products", "name" : "_id_" }
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "garden.user_actions", "name" :
"_id_" }
{ "v" : 1, "key" : { "time_field" : 1 }, "name" : "time_field_1", "ns" :
"garden.reviews", "expireAfterSeconds" : 3600 }
```

Documents and insertion

DOCUMENT SERIALIZATION, TYPES, AND LIMITS

All documents are serialized to BSON before being sent to MongoDB; they're later deserialized from BSON. The driver handles this process and translates it from and to the appropriate data types in its programming language. Most of the drivers provide a simple interface for serializing from and to BSON; this happens automatically when reading and writing documents. You don't need to worry about this normally, but we'll demonstrate it explicitly for educational purposes.

In the previous capped collections example, it was reasonable to assume that the sample document size was roughly 102 bytes. You can check this assumption by using the Ruby driver's BSON serializer:

```
doc = {
  :_id => BSON::ObjectId.new,
  :username => "kbanker",
  :action_code => rand(5),
  :time => Time.now.utc,
  :n => 1
}
bson = doc.to_bson
puts "Document #{doc.inspect} takes up #{bson.length} bytes as BSON"
```

Deserializing BSON is as straightforward with a little help from the StringIOclass.

Try running this Ruby code to verify that it works:

```
string_io = StringIO.new(bson)
deserialized_doc =
String.from_bson(string_io)
puts "Here's our document deserialized from
BSON:" puts deserialized_doc.inspect
```

STRINGS

All string values must be encoded as UTF-8. Though UTF-8 is quickly becoming the standard for character encoding, there are plenty of situations when an older encod- ing is still used. Users typically encounter issues with this when importing data gener- ated by legacy systems into MongoDB.

NUMBERS

BSON specifies three numeric types: double, int, and long. This means that BSON can encode any IEEE floating-point value and any signed integer up to 8 bytes in length. When serializing integers in dynamic languages, such as Ruby and Python, the driver will automatically determine whether to encode as an int or a long. In fact, there's only one common situation where a number's type must be made explicit: when you're inserting numeric data via the JavaScript shell. JavaScript, unhappily, natively

---

supports only a single numeric type called Number, which is equivalent to an IEEE 754 Double. Consequently, if you want to save a numeric value from the shell as an integer, you need to be explicit, using either NumberLong() or NumberInt(). Try this example:

```
db.numbers.save({n: 5});
db.numbers.save({n:
NumberLong(5)});
```

You've saved two documents to the numbers collection, and though their values are equal, the first is saved as a double and the second as a long integer. Querying for all documents where n is 5 will return both documents:

```
> db.numbers.find({n: 5});
{ "_id" : ObjectId("4c581c98d5bbeb2365a838f9"), "n" : 5 }
{ "_id" : ObjectId("4c581c9bd5bbeb2365a838fa"), "n" : NumberLong( 5 ) }
```

DATETIMES

The BSON datetime type is used to store temporal values. Time values are represented using a signed 64-bit integer marking milliseconds since the

Unix epoch. A negative value marks milliseconds prior to the epoch.[10]

VIRTUAL TYPES

What if you must store your times with their time zones? Sometimes the basic BSON types don't suffice. Though there's no way to create a custom BSON type, you can compose the various primitive BSON values to create your own virtual type in a sub- document. For instance, if you wanted to store times with zone, you might use a docu- ment structure like this, in Ruby:

```
{
  time_with_zone:
    { time: new
    Date(), zone:
    "EST"
  }
}
```

It's not difficult to write an application so that it transparently handles these compos- ite representations. This is usually how it's done in the real world. For example, Mongo-Mapper, an object mapper for MongoDB written in Ruby, allows you to define to_mongo and from_mongo methods for any object to accommodate these sorts of cus- tom composite types.

LIMITS ON DOCUMENTS

BSON documents in MongoDB v2.0 and later are limited to 16 MB in size. The limit exists for two related reasons. First, it's there to prevent developers from creating ungainly data models. Though poor data models are still possible with this limit, the 16 MB limit helps discourage schemas with oversized documents.

If you find yourself needing to store documents greater than 16 MB, consider whether your schema should split data into smaller documents, or whether a MongoDB document is even the right place to store such information—it may be better managed as a file.

The second reason for the 16 MB limit is performance-related. On the

server side, querying a large document requires that the document be copied into a buffer before being sent to the client. This copying can get expensive, especially (as is often the case) when the client doesn't need the entire document.[12] In addition, once sent, there's the work of transporting the document across the network and then deserializ- ing it on the driver side. This can become especially costly if large batches of multi-megabyte documents are being requested at once.

MongoDB documents are also limited to a maximum nesting depth of 100. Nesting occurs whenever you store a document within a document. Using deeply nested docu- ments—for example, if you wanted to serialize a tree data structure to a MongoDB

---

document—results in documents that are difficult to query and can cause problems during access. These types of data structures are usually accessed through recursive function calls, which can outgrow their stack for especially deeply nested documents.

BULK INSERTS

All of the drivers make it possible to insert multiple documents at once. This can be extremely handy if you're inserting lots of data, as in an initial bulk import or a migra- tion from another database system. Here's a simple Ruby example of this feature:

```
docs = [                          # define an array of documents
  { :username => 'kbanker' },
  { :username => 'pbakkum' },
  { :username => 'sverch' }
]
@col = @db['test_bulk_insert']
@ids = @col.insert_many(docs)   # pass the entire array to
insert puts "Here are the ids from the bulk insert:
#{@ids.inspect}"
```

Constructing Queries

This topic covers

- Querying an e-commerce data model
- The MongoDB query language in detail
- Query selectors and options

E-commerce queries

For instance, _idlookups shouldn't be a mystery at this point. But we'll also show you a few more sophisticated patterns, including querying for and displaying a category hierarchy, as well as providing filtered views of product listings.

Products, categories, and reviews

Most e-commerce applications provide at least two basic views of products and catego- ries. First is the product home page, which highlights a given product, displays reviews, and gives some sense of the product's categories. Second is the product listing page, which allows users to browse the category hierarchy and view thumbnails of all the products within a selected category. Let's begin with the product home page, in many ways the simpler of the two.

Imagine that your product page URLs are keyed on a product slug (you learned about these user-friendly permalinks in chapter 4). In that case, you can get all the data you need for your product page with the following three queries:

```
roduct = db.products.findOne({'slug': 'wheel-barrow-
9092'}) db.categories.findOne({'_id':
product['main_cat_id']}) db.reviews.find({'product_id':
product['_id']})
```

FINDONE VS. FIND QUERIES

The findOnemethod is similar to the following, though a cursor is returned even when you apply a limit:

```
db.products.find({'slug': 'wheel-barrow-9092'}).limit(1)
```

SKIP, LIMIT, AND SORT QUERY OPTIONS

Most applications paginate reviews, and for enabling this MongoDB provides skipand
limitoptions. You can use these options to paginate the review document like this:

```
db.reviews.find({'product_id': product['_id']}).skip(0).limit(12)


db.reviews.find({'product_id': product['_id']}).
                    sort({'helpful_votes': -1}).
                    limit(12)


page_number = 1
product = db.products.findOne({'slug': 'wheel-barrow-
9092'}) category = db.categories.findOne({'_id':
product['main_cat_id']}) reviews_count =
db.reviews.count({'product_id': product['_id']}) reviews =
db.reviews.find({'product_id': product['_id']}).
                            skip((page_number - 1) *
                            12). limit(12).
                            sort({'helpful_votes': -1})
```

MongoDB's query language

Query criteria and selectors

Query criteria allow you to use one or more query selectors to specify the query's results. MongoDB gives you many possible selectors. This section provides an overview.

SELECTOR MATCHING

The simplest way to specify a query is with a selector whose key-value pairs literally match against the document you're looking for. Here are a couple of examples:

```
db.users.find({'last_name': "Banker"})
db.users.find({'first_name': "Smith", birth_year:
1975})
```

RANGES

Table 5.1 shows the range query operators most commonly used in MongoDB.

Table 5.1 Summary of range query operators

Operator	Description
$lt	Less than
$gt	Greater than
$lte	Less than or equal
$gte	Greater than or equal

Beginners sometimes struggle with combining these operators. A common mistake is to repeat the search key:

```
db.users.find({'birth_year': {'$gte': 1985}, 'birth_year': {'$lte': 2015}})
```

The aforementioned query only takes into account the last condition. You can prop- erly express this query as follows:

```
db.users.find({'birth_year': {'$gte': 1985, '$lte': 2015}})
```

SET OPERATORS

Three query operators—$in, $all, and $nin—take a list of one or more values as their predicate, so these are called set operators. $in returns a document if any of the given values matches the search key.

Table 5.2 Summary of set operators

Operator	Description
$in	Matches if any of the arguments are in the referenced set
$all	Matches if all of the arguments are in the referenced set and is used in documents that contain arrays
$nin	Matches if none of the arguments are in the referenced set

If the following list of category IDs

```
[
  ObjectId("6a5b1476238d3b4dd5
  000048"),
  ObjectId("6a5b1476238d3b4dd5
  000051"),
  ObjectId("6a5b1476238d3b4dd5
  000057")
]
```

corresponds to the lawnmowers, hand tools, and work clothing categories, the query to find all products belonging to these categories looks like this:

```
db.products.find({
    'main_cat_id':
    {
      '$in': [
        ObjectId("6a5b1476238d3b4dd5
        000048"),
        ObjectId("6a5b1476238d3b4dd5
        000051"),
```

```
            ObjectId("6a5b1476238d3b4dd5
            000057")
        ]
    }
})
```

Table 5.3 Summary of Boolean operators

Operator	Description
$ne	Matches if the argument is not equal to the element
$not	Inverts the result of a match
$or	Matches if any of the supplied set of query terms is true
$nor	Matches if none of the supplied set of query terms are true
$and	Matches if all of the supplied set of query terms are true
$exists	Matches if the element exists in the document.

QUERYING FOR A DOCUMENT WITH A SPECIFIC KEY

The final operator we'll discuss in this section is $exists. This operator is necessary because collections don't enforce a fixed schema, so you occasionally need a way to query for documents containing a particular key. Recall that you'd planned to use

each product's detailsattribute to store custom fields. You might, for instance, store a color field inside the details attribute. But if only a subset of all products specify a set of colors, then you can query for the ones that don't like this:

```
db.products.find({'details.color': {$exists: false}})
```

The opposite query is also possible:

```
db.products.find({'details.color': {$exists: true}})
```

ARRAYS

Arrays give the document model much of its power. As you've seen in the e-commerce example, arrays are used to store lists of strings, object IDs, and even other docu- ments.

Arrays afford rich yet comprehensible documents; it stands to reason that MongoDB would let you query and index the array type with ease. And it's true: the simplest array queries look like queries on any other document type, as you can see in table 5.4.

Table 5.4 Summary of array operators

Operator	Description
$elemMatch	Matches if all supplied terms are in the same subdocument
$size	Matches if the size of the array subdocument is the same as the supplied literal value

Let's look at these arrays in action. Take product tags again. These tags are repre- sented as a simple list of strings:

```
{
  _id:
  ObjectId("4c4b1476238d3b4dd5003981
  "), slug: "wheel-barrow-9092",
  sku:  "9092",
  tags: ["tools", "equipment", "soil"]
}
```

Querying for products with the tag "soil"is trivial and uses the same syntax as query- ing a single document value:

```
db.products.find({tags: "soil"})
```

Importantly, this query can take advantage of an index on the tags field. If you build the required index and run your query with explain(), you'll see that a B-tree cursor[3] is used:

```
db.products.ensureIndex({tags: 1})
db.products.find({tags:
"soil"}).explain()
```

When you need more control over your array queries, you can use dot notation to query for a value at a particular position within the array. Here's how you'd restrict the previous query to the first of a product's tags:

```
db.products.find({'tags.0': "soil"})
```

REGULAR EXPRESSIONS

The $regexoperator is summarized here:

- $regex Match the element against the supplied regex term

MongoDB is a case-sensitive system, and when using a regex, unless you use the /i modifier (that is, /best|worst/i), the search will have to exactly match the case of the fields being searched. But one caveat is that if you do use /i, it will disable the use of indexes. If you want to do indexed case-insensitive search of the contents of string fields in documents, consider either storing a duplicate field with the contents forced to lowercase specifically for searching or using MongoDB's text search capabili- ties, which can be combined with other queries and does provide an indexed case- insensitive search.

MISCELLANEOUS QUERY OPERATORS

Two more query operators aren't easily categorized and thus deserve their own sec- tion. The first is $mod, which allows you to query documents matching a given modulo operation, and the second is $type, which matches values by their BSON type. Both are detailed in table 5.5.

Table 5.5 Summary of miscellaneous operators

Operator	Description
$mod [(quotient),(result)]	Matches if the element matches the result when divided by the quotient
$type	Matches if the element type matches a specified BSON type
$text	Allows you to performs a text search on the content of the fields indexed with a text index

For instance, $modallows you to find all order subtotals that are evenly divisible by 3 using the followingquery:

```
db.orders.find({subtotal: {$mod: [3, 0]}})
```

You can see that the $modoperator takes an array having two values. The first is the divisor and the second is the expected remainder. This query technically reads, "Find all documents with subtotals that return a remainder of 0 when divided by 3." This is a contrived example, but it demonstrates the idea. If you end up using the $mod opera- tor, keep in mind that it won't use an index.

The second miscellaneous operator, $type, matches values by their BSON type. I don't recommend storing multiple types for the same field within a collection, but if the situation ever arises, you have a query operator that lets you test against type.

Table 5.6 BSON types

BSON type	$type number	Example
Double	1	123.456
String (UTF-8)	2	“Now is the time”
Object	3	{ name:"Tim",age:"myob" }
Array	4	[123,2345,"string"]
Binary	5	BinData(2,"DgAAAEltIHNvbWUgYmluYXJ5")
ObjectId	7	ObjectId("4e1bdda65025ea6601560b50") true
Boolean	8	ISODate("2011-02-24T21:26:00Z")
Date	9	null
Null	10	/test/i
Regex	11	function() {return false;}
JavaScript	13	Not used; deprecated in the standard function
Symbol	14	(){return false;}
Scoped JavaScript	15	10
32-bit integer	16	{ "t" : 1371429067, "i" : 0 }
Timestamp	17	NumberLong(10)
64-bit integer	18	
Maxkey	127	
Minkey	255	
Maxkey	128	

		{"$maxKey": 1} { "$minKey" : 1} {"maxkey" : { "$maxKey" : 1 }}

PROJECTIONS

- Projections are most commonly defined as a set of

fields to return:

```
db.users.find({}, {'username': 1})
```

SORTING

```
db.reviews.find({}).sort({'rating': -1})
```

Naturally, it might be more useful to sort by helpfulness and then by rating:

```
db.reviews.find({}).sort({'helpful_votes':-1, 'rating': -1})
```

SKIP AND LIMIT

```
db.docs.find({}).skip(500000).limit(10).sort({date: -1})
```

becomes this:

```
previous_page_date = new Date(2013, 05, 05)
db.docs.find({'date': {'$gt': previous_page_date}}).limit(10).sort({'date': -1})
```

This topic covers

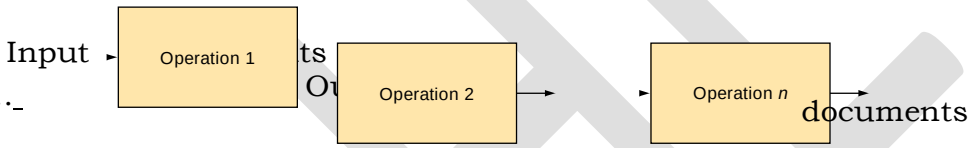
- Aggregation on the e-commerce data model
- Aggregation framework details
- Performance and limitations
- Other aggregation capabilities

Aggregation framework overview

A call to the aggregation framework defines a pipeline (figure 6.1), the *aggregation pipeline*, where the output from each step in the pipeline provides input to the next step. Each step executes a single operation on the input documents to transform the input and generate output documents.

Aggregation pipeline operations include the following:

- $project—Specify fields to be placed in the output document (projected).
- $match—Select documents to be processed, similar to find().



- $limit—Limit the number of documents to be passed to the next step.
- $skip—Skip a specified number of documents.
- $unwind—Expand an array, generating one output document for each array entry.
- $group—Group documents by a specified key.
- $sort—Sort documents.
- $geoNear—Select documents near a geospatial location.

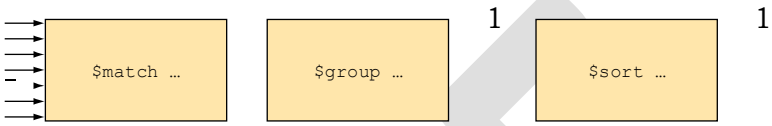
- $out—Write the results of the pipeline to a collection (new in v2.6).
- $redact—Control access to certain data (new in v2.6).

Most of these operators will look familiar if you've read the previous chapter on con- structing MongoDB queries. Because most of the aggregation framework operators work similarly to a function used for MongoDB queries, you should make sure you have a good understanding of section 5.2 on the MongoDB query language before continuing.

This code example defines an aggregation framework pipeline that consists of a match, a group, and then a sort:

db.products.aggregate([{$match: …}, {$group: …}, {$sort: …}])

This series of operations is illustrated in figure 6.2.



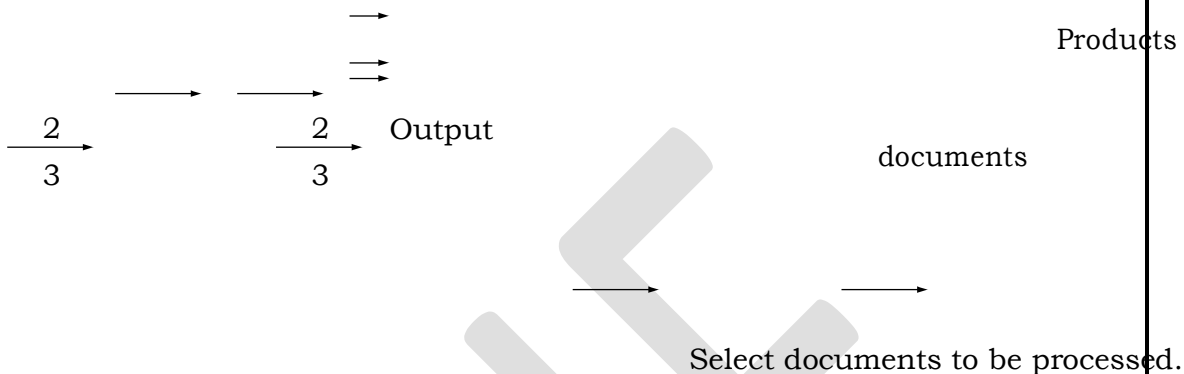


Table 6.1 SQL versus aggregation framework comparison

SQL command	Aggregation framework operator
SELECT	$project
	$group functions: $sum, $min, $avg, etc.
FROM	db.collectionName.aggregate(...)
JOIN	$unwind
WHER	$match
E	$group
GROU	$match
PBY	
HAVING	

Products, categories, and reviews

Now let's look at a simple example of how the aggregation framework can be used to summarize information about a product. Chapter 5 showed an example of counting the number of reviews for a given product using this query:

product = db.products.findOne({'slug': 'wheelbarrow-

```
9092'}) reviews_count = db.reviews.count({'product_id':
product['_id']})
```

Let's see how to do this using the aggregation framework. First, we'll look at a query that will calculate the total number of reviews for all products:

```
db.reviews.aggregate([
 {$group : {
 _id:'$product_id',
                count:{$sum:1}
                }}
]);
```

Group the input documents by product_id.

Count the number of reviews for each product.

This single operator pipeline returns one document for each product in your data- base that has a review, as illustrated here:

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003982"), "count" : 2 }
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"), "count" : 3 }
```

Outputs one document for each product

Next, add one more operator to your pipeline so that you select only the one prod- uct you want to get a count for:

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})
```

```
ratingSummary = db.reviews.aggregate([
 {$match : { product_id: product['_id']} },
 {$group : { _id:'$product_id',
```

Select only a single product.

```
]).next();
Count:{$sum:1} }}
```

Return the first document in the results.

is example returns the one product you're interested in and assigns it to the vari- able ratingSummary. Note that the result from the aggregation pipeline is a *cursor*, a pointer to your results that allows you to process results of almost any size, one docu- ment at a time. To retrieve the single document in the result, you use the next() func- tion to return the first document from the cursor:

```
{ "_id" : ObjectId("4c4b1476238d3b4dd5003981"), "count" : 3 }
```

The parameters passed to the $matchoperator, {'product_id':product['_id']}, should look familiar. They're the same as those used for the query taken from chap- ter 5 to calculate the count of reviews for a product:

```
db.reviews.count({'product_id': product['_id']})
```

CALCULATING THE AVERAGE REVIEW

To calculate the average review for a product, you use the same pipeline as in the pre- vious example and add one more field:

```
product = db.products.findOne({'slug': 'wheelbarrow-9092'})

                    ratingSummary = db.reviews.aggregate([
    {$match : {'product_id': product['_id']}},
    {$group : { _id:'$product_id', average:{$avg:'$rating'}, count:
        {$sum:1}}}
]).next();
```

Calculate the average rating for a product.

The previous example returns a single document and assigns it to the variable rating- Summarywith the content shown here:

```
{
    "_id" :
    ObjectId("4c4b1476238d3b4dd5003981
    "), "average" : 4.333333333333333,
    "count" : 3
}
```

This example uses the $avgfunction to calculate the average rating for the product. Notice also that the field being averaged, rating, is specified using '$rating'in the
$avgfunction. This is the same convention used for specifying the field for the $group _idvalue, where you used this:

```
_id:'$product_id'.
```

```
                count:{$sum:1}}}
                        ]).toArray();
```

As shown in this snippet, you've once again produced a count using the $sum func- tion; this time you counted the number of reviews for each rating. Also note that the result of this aggregation call is a cursor that you've converted to an array and assigned to the variable countsByRating.

SQL query

For those familiar with SQL, the equivalent SQL query would look something like this:

```
SELECT RATING, COUNT(*) AS COUNT
  FROM REVIEWS
 WHERE PRODUCT_ID = '4c4b1476238d3b4dd5003981'
 GROUP BY RATING
```

This aggregation call would produce an array similar to this:

```
[ { "_id" : 5, "count" : 5 },
  { "_id" : 4, "count" : 2 },
  { "_id" : 3, "count" : 1 } ]
```

JOINING COLLECTIONS

Next, suppose you want to examine the contents of your database and count the num- ber of products for each main category. Recall that a product has only one main cate- gory. The aggregation command looks like this:

```
db.products.aggregate([
    {$group : { _id:'$main_cat_id',
                count:{$sum:1}}}
]);
```

This command would produce a list of output documents. Here's an example:

```
{ "_id" : ObjectId("6a5b1476238d3b4dd5000048"), "count" : 2 }
```

option is to use the forEachfunction to process the cursor returned from the aggre- gation command and add the name using a pseudo-join. Here's an example:

```
b.mainCategorySummary.remove({});

db.products.aggregate([
    {$group : { _id:'$main_cat_id',
                count:{$sum:1}}}
]).forEach(function(doc){
```

Remove existing documents from mainCategorySummary collection

Read category for a result

```
    var category = db.categories.findOne({_id:doc._id});
    if (category !== null) {

        doc.category_name = category.name;
                        }
                        else {
        doc.category_name = 'not found';
                        }
    db.mainCategorySummary.insert(doc);
})
```

mainCategorySummary:

```
db.products.aggregate([
    {$group : { _id:'$main_cat_id',
                count:{$sum:1}}},
    {$out : 'mainCategorySummary'}
```

])

User and order

When the first edition of this book was written, the aggregation framework, first intro- duced in MongoDB v2.2, hadn't yet been released. The first edition used the MongoDB map-reducefunction in two examples, grouping reviews by users and summarizing sales by month. The example grouping reviews by user showed how many reviews each reviewer had and how many helpful votes each reviewer had on average. Here's what this looks like in the aggregation framework, which provides a much simpler and more intuitive approach:

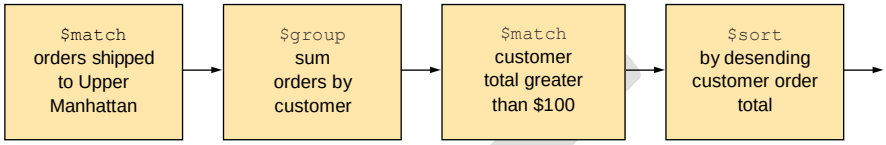
```
db.reviews.aggregate([
    {$group :
        {_id : '$user_id',
         count : {$sum : 1},
         avg_helpful : {$avg : '$helpful_votes'}}
    }
])
```

The result from this call looks like this:

```
{ "_id" :
    ObjectId("4c4b1476238d3b4dd50000
    03"), "count" : 1, "avg_helpful" : 10 }
{ "_id" :
    ObjectId("4c4b1476238d3b4dd50000
    02"), "count" : 2, "avg_helpful" : 4 }
{ "_id" :
    ObjectId("4c4b1476238d3b4dd50000
    01"), "count" : 2, "avg_helpful" : 5 }
```

FINDING BEST MANHATTAN CUSTOMERS

Now let's extend that query to find the highest spenders in Upper Manhattan. This pipeline is summarized in figure 6.5. Notice that the $match is the first step in the pipeline, greatly reducing the number of documents your pipeline has to process.



The query includes these steps:

- $match—Find orders shipped to Upper Manhattan.
- $group—Sum the order amounts for each customer.
- $match—Select those customers with order totals greater than $100.
- $sort—Sort the result by descending customer order total.