10 WiredTiger and pluggable storage

This chapter covers

- WiredTiger
- Pluggable storage engines
- A comparison between MMAPv1 and WiredTiger

With version 3.0, MongoDB introduced the Pluggable Storage Engine API as one of its major changes. In this chapter, we'll talk about what exactly it is and why it has been added to MongoDB. We'll talk about WiredTiger, a pluggable storage engine that's bundled with MongoDB, and compare it with the default storage engine that MongoDB has used up until version 3.0. We'll compare the two engines in terms of speed, disk use, and latency. We'll also introduce several other pluggable storage engines that are expected to become interesting alternatives. For the more advanced readers, we'll uncover the technology behind pluggable storage engines.

10.1 Pluggable Storage Engine API

An application programming interface (API) is a relatively strict set of routines, protocols, and tools for building software applications. As an example, you should be aware by now that MongoDB offers an API that allows other software to interact with MongoDB without using the MongoDB shell: each of the MongoDB drivers

that you've been using use the API provided by MongoDB to add driver functionality. They allow your application to communicate with the MongoDB database and to perform the basic CRUD operations on your documents in the database.

A storage engine is an interface between the database and the hardware. A storage engine doesn't change how you perform your queries in the shell or in the driver, and it doesn't interfere with MongoDB at the cluster level. But storage engines interfere with how data is written to, deleted from, and read from disk, as well as which data structure will be used for storing the data.

The Pluggable Storage Engine API allows third parties to develop storage engines for MongoDB. Before the Pluggable Storage Engine API, the only storage engine available to MongoDB was MMAPv1.

MongoDB still uses the MMAPv1 storage engine, and it's still the default storage engine in version 3.0 and later. The MMAPv1 storage engine is based on memory mapping, and has been a stable solution for MongoDB so far. One drawback to MMAPv1 that you'll notice soon if you have a lot of data to store is that it quickly consumes an enormous amount of disk space as your data set grows, to the extent that it preallocates 2 GB blocks every time it needs to grow in size. But preallocation is done by most database systems, and MongoDB is no exception. It does this in small, growing increments at first, but once it becomes larger than 2 GB, every next increment will preallocate another 2 GB, so as a system administrator you'll have to keep this in mind when managing disk space for your servers.

The database administrator has to choose from the different storage alternatives, which dictate how data is stored on disk. Since version 3.0, it's now possible to tell MongoDB to use a different module for storage, and that's what the Pluggable Storage Engine API does. It provides functions that MongoDB needs to use to store data. MongoDB 3.0 comes bundled with an alternative to MMAPv1, which is WiredTiger. We'll talk more about WiredTiger and how you can switch to using it in a later section in this chapter, but first let's consider why MongoDB has offered the ability to use different storage engines.

10.1.1 Why use different storages engines?

Let's consider two different applications:

- A news site, like Huffington Post
- A social media site, like Twitter or Fac

On news sites you'll see news articles. The Huffington Post averages 1,200 editorial pieces per day, but they're read by tens of millions of people around the world.[1] Bring this into contrast with a social media site where people share their own stories, which are much shorter than news articles. Twitter tweets are at most 140 characters, and Fac or Google+ status updates are short as well. These two different

[1] According to a 2013 article on DigiDay: http://digiday.com/publishers/whos-winning-at-volume-in-publishing/.

use cases have different requirements in terms of database storage and access, as table 10.1 shows.

News sites have much less data to delve into compared to social media sites, and for many visitors, the front page looks the same. Social media sites, on the other hand, have to go through millions and millions of tweets or status updates. Every visitor has their own feed, which should show only those tweets and updates that matter to the visitor. In addition to delivering millions of status updates for different visitors, the social media platforms also need to be able to store millions of new tweets every day.[2]

Table 10.1 Different requirements for different cases/users

	News site	Social media site
Number of documents	Hundreds of articles	Millions of updates
Average size	A few kilobytes	Tens of bytes
Dynamic content	None—same for every visitor	Content depends on visitor

For news sites, the application needs to collect the same articles over and over again for every user visiting the news site at the same time. Many database systems have a query cache that will quickly deliver the data that was requested by the same query a few minutes ago. Such news site applications can also make use of an external in-memory cache system such as Memcached or Redis to deliver the same data at high speeds. But these technologies will not help social media sites where the requested data is different every time, even per visitor. Such applications need a different kind of storage system that has much better performance when reading filtered data from a humongous set of records. Social media sites also need a storage system that has excellent write performance to be able to store millions of new records every day. News sites don't need this kind of performance because their number of write operations only runs in the mere thousands.

To cater to these different kinds of systems, MongoDB has implemented the concept of a pluggable storage engine so that the database administrators or system engineers can choose the storage engine that gives the best performance for their use case. In the next section we'll introduce a storage plugin that's bundled with MongoDB: WiredTiger.

10.2 WiredTiger

WiredTiger is a high-performance, scalable, open source data engine that focuses on multicore scalability and optimal RAM use. Multicore scaling is achieved by using

[2] According to Domo, in 2014, every minute of the day over 2.4 million pieces of content were shared on Facebook, and over 270,000 tweets were sent on Twitter. These numbers translate to over 3.5 billion shares per day, and 400 million tweets per day.

modern programming techniques such as hazard pointers[3] and lock-free algorithms,[4] resulting in more work done by each CPU core than alternative engines.

WiredTiger was developed by Michael Cahill and Keith Bostic, both architects at Sleepycat Software, which was founded by Bostic and his wife. At Sleepycat Software they designed and developed the Berkeley DB, the most widely used embedded data-management software in the world.

10.2.1 Switching to WiredTiger

Before you start using WiredTiger, make sure you're running a 64-bit system, with a 64-bit OS, because this is required. This should be the case in most modern computer equipment. Also, when setting up MongoDB to use WiredTiger, it's crucial that you start the MongoDB server with the WiredTiger configuration on a fresh dbPath directory. If you start the server with a dbPath that's in the MMAPv1 structure, it won't start. This is because the storage structure of MMAPv1 isn't compatible with that of WiredTiger, and there's no on-the-fly conversion available between storage structures. But there's a way to migrate your MMAPv1-based databases to WiredTiger, and vice versa, using `mongodump` and `mongorestore`. See chapter 13 to learn more about dumping and restoring your databases.

All you need to do to enable WiredTiger in your MongoDB installation is to set the storage configuration in your default YAML configuration file (see appendix A for more information about YAML) as follows:

```
storage:
    dbPath: "/data/db"
    journal:
        enabled: true
    engine: "wiredTiger"
    wiredTiger:
        engineConfig:
            cacheSizeGB: 8
            journalCompressor: none
        collectionConfig:
            blockCompressor: none
        indexConfig:
            prefixCompression: false
```

[3] In multithreading programming it's important to keep track of which memory blocks need to be accessible by the threads, and whether any thread is accessing it. Hazard pointers are a list of pointers to such memory blocks that are being accessed by a thread, and other threads are prohibited from modifying or deleting the pointers and the memory blocks they point to, as long as they're in the hazard pointers list.

[4] Resource locking is an important concept in multithreaded programming. Lock-free algorithms are programming patterns to avoid a program being stuck because several threads are waiting for each other to release their locks, and to guarantee that the program as a whole makes progress.

This is the basic, noncompressed configuration to enable WiredTiger for your MongoDB installation. Table 10.2 shows what the options do.

Table 10.2 Various options of the MongoDB configuration file

Option Name	Description
`dbPath`	The path where your database files are stored. Defaults to `/data/db`.
`journal.enabled`	Whether to enable journaling or not. It's recommended to enable this as it may save data that was lost during a power outage and hasn't been synchronized to disk. Defaults to true on 64-bit systems.
`engine`	Which storage engine to use. Defaults to `mmapv1`. To use WiredTiger, set this to `wiredTiger`.
`wiredTiger`	This is where WiredTiger-specific options are set.
`engineConfig.cacheSize`	This is how much RAM memory WiredTiger needs to reserve for the in-memory data, which would then serve as a cache to rapidly serve your data. Defaults to half the physical RAM on your system, at least 1 GB.
`engineConfig.journalCompressor`	Tells WiredTiger what kind of compressor to use for the journaling data. Defaults to `snappy`, but this is best set to `none` to achieve the best performance.
`collectionConfig.blockCompressor`	This tells WiredTiger what kind of compressor to use for the collection data. The three supported options are `none`, `snappy`, and `zlib`. You'll see in the benchmarks which is the best option for you. Defaults to `snappy`.
`indexConfig.prefixCompression`	This tells WiredTiger whether to use compression for its index data. Defaults to `true`.

10.2.2 Migrating your database to WiredTiger

Because you can't run MongoDB with the WiredTiger set up on a data directory that's in MMAPv1 format, you'll need to migrate your database to the WiredTiger installation. This is basically done by creating dumps and restoring from them (see chapter 13 for more information):

1. Create a MongoDB dump of your MMAPv1-based database:

```
$ mkdir ~/mongo-migration
$ cd ~/mongo-migration
$ mongodump
```

   This will create a dump directory called mongo-migration in your home directory.

2. Stop the `mongod` instance, and make sure there's no `mongod` process running:

```
$ ps ax | grep mongo
```

3. Update the MongoDB configuration to use WiredTiger, as described in the previous section.
4. Move the MMAPv1-based database away. Assuming `dbPath` is set to /data/db:

```
$ sudo mv /data/db /data/db-mmapv1
```

5. Create a fresh directory and give the MongoDB user/group (assuming `mongodb`) permission to write to it:

```
$ mkdir /data/db
$ chown mongodb:mongodb /data/db
$ chmod 755 /data/db
```

6. Start your WiredTiger-enabled MongoDB instance.
7. Import your dump into the WiredTiger-enabled MongoDB instance:

```
$ cd ~/mongo-migration
$ mongorestore dump
```

Now you should have your database in your newly configured MongoDB environment. If you examine the WiredTiger-based /data/db and compare it with your old MMAPv1-based /data/db-mmapv1, you'll see considerable differences in how they're managed. For one, if you have large databases, you'll notice that the disk use of both directories differs greatly. You'll see that when we do the benchmark testing in the next section.

To convert your WiredTiger-based database back into MMAPv1, repeat the process, but now you'll make a dump of the data in your WiredTiger storage, stopping the MongoDB instance, changing the configuration to use MMAPv1, starting the MongoDB instance with this configuration, and restoring the data dump into the MMAPv1-enabled instance.

An alternative method is to run a second MongoDB instance with a WiredTiger configuration, and add this instance to your replica set, together with your existing MMAPv1 instance. Replica sets are covered in chapter 11.

10.3 Comparison with MMAPv1

How does WiredTiger's performance compare to a MongoDB instance with MMAPv1? In this chapter, you'll test three WiredTiger configurations against MMAPv1 using a couple of JavaScript and shell scripts. You'll test the following configurations:

- Default configuration using MMAPv1
- Normal WiredTiger, no compression

- WiredTiger with snappy compression
- WiredTiger with zlib compression

Zlib and snappy are compression algorithms. The former is an abstraction of the DEFLATE compression algorithm, which is an LZ77 variant that uses Huffman coding. Zlib is very common on many software platforms, and is the basis for the gzip file compression program. Snappy is developed by Google and is widely used in Google's projects such as BigTable. Snappy is more of an intermediate solution, which doesn't aim for maximum compression, but instead goes for high speeds, along with reasonable compression.

You're going to test the insert performance first to fill up the test databases. Then you'll use these test databases to measure the read performance. When going through the benchmark testing in the following sections, please keep in mind that it's difficult to draw the correct conclusions from these test results. The data sets in these tests are much smaller than real-world examples, and in real-world examples you'd have many more different search filters to collect the data that you need. There would also be a larger variety in the data structures in real-world examples, which will affect the compression algorithms in use. The tests in this chapter are only simple examples to give you a general idea. They'll also hopefully give you some insight into how you'd proceed testing the MongoDB instances with your own data and the filters that your application uses.

10.3.1 Configuration files

To be able to compare the disk use of the different storage configurations, you'll use different database paths in the storage configuration files. In a production system, however, you'd end up with the common database path for a MongoDB installation on your machine.

The configuration file for the MMAPv1 instance, called mmapv1.conf, is as follows, in YAML format:

```
storage:
    dbPath: "./data-mmapv1"
    directoryPerDB: true
    journal:
        enabled: true
systemLog:
    destination: file
    path: "./mongodb-server.log"
    logAppend: true
    timeStampFormat: iso8601-utc
net:
    bindIp: 127.0.0.1
    port: 27017
    unixDomainSocket:
        enabled : true
```

For the WiredTiger configurations, it's the same as the previous code, except for the storage part. For the no-compression version you'll use the following storage configuration for WiredTiger, naming it wiredtiger-uncompressed.conf:

```
storage:
    dbPath: "./data-wt-uncompressed"
    directoryPerDB: true
    journal:
        enabled: true
    engine: "wiredTiger"
    wiredTiger:
        engineConfig:
            cacheSizeGB: 8
            journalCompressor: none
        collectionConfig:
            blockCompressor: none
        indexConfig:
            prefixCompression: false
```

For the snappy instance, you'll use the following configuration for storage; what's different from the uncompressed WiredTiger instance is shown in bold. This file is called "wiredtiger-snappy.conf":

```
storage:
    dbPath: "./data-wt-zlib"
    directoryPerDB: true
    journal:
        enabled: true
    engine: "wiredTiger"
    wiredTiger:
        engineConfig:
            cacheSizeGB: 8
            journalCompressor: none
        collectionConfig:
            blockCompressor: snappy
        indexConfig:
            prefixCompression: true
```

Finally, for the zlib instance, you'll use the following storage configuration, naming it wiredtiger-zlib.conf:

```
storage:
    dbPath: "./data-wt-snappy"
    directoryPerDB: true
    journal:
        enabled: true
    engine: "wiredTiger"
    wiredTiger:
        engineConfig:
            cacheSizeGB: 8
            journalCompressor: none
        collectionConfig:
            blockCompressor: zlib
```

```
        indexConfig:
            prefixCompression: true
```

If you have a legacy MongoDB installation in some directory, and these configuration files in a configs directory, you can use this MongoDB installation with these configurations by, for example running the following for the MMAPv1 configuration:

```
$ bin/mongod --config configs/mmapv1.conf &
```

This should run a MongoDB instance in the background with the given configuration.

10.3.2 Insertion script and benchmark script

You'll use the following JavaScript code to fill up the benchmark database with documents that have four fields of different types, an array field with eight document elements with four differently typed fields each, and eight subdocuments, also with four differently typed fields each. This is by no means a real-world situation, and compression algorithms will work differently with different data sets that are more heterogeneous than this test data:

```
for (var j = 0; j < 10000; j++) {
    var r1 = Math.random();

    // A nice date around year 2000
    var dateFld = new Date(1.5e12 * r1);
    var intFld = Math.floor(1e8 * r1);
    // A nicely randomized string of around 40 characters
    var stringFld = Math.floor(1e64 * r1).toString(36);
    var boolFld = intFld % 2;

    doc = {
        random_date: dateFld,
        random_int: intFld,
        random_string: stringFld,
        random_bool: boolFld
    }

    doc.arr = [];

    for (var i = 0; i < 16; i++) {
        var r2 = Math.random();

        // A nice date around year 2000
        var dateFld = new Date(1.5e12 * r2);
        var intFld = Math.floor(1e8 * r2);
        var stringFld = Math.floor(1e64 * r2).toString(36);
        var boolField = intFld % 2;

        if (i < 8) {
            doc.arr.push({
                date_field: dateFld,
                int_field: intFld,
                string_field: stringFld,
                bool_field: boolFld
            });
```

```
        } else {
            doc["sub" + i] = {
                date_field: dateFld,
                int_field: intFld,
                string_field: stringFld,
                bool_field: boolFld
            };
        }
    }

    db.benchmark.insert(doc);
}
```

This piece of JavaScript code, stored in insert.js, will insert 10,000 documents into the benchmark database. This script will be run by the following batch script that will go through all the four configurations, and does the same insert job for each configuration, 16 times:

```
#!/bin/bash

export MONGO_DIR=/storage/mongodb
export NUM_LOOPS=16

configs=(
    mmapv1.conf
    wiredtiger-uncompressed.conf
    wiredtiger-snappy.conf
    wiredtiger-zlib.conf
)

cd $MONGO_DIR
for config in "${configs[@]}"; do
    echo "===== RUNNING $config ====="
    echo "Cleaning up data directory"
    DATA_DIR=$(grep dbPath configs/$config | awk -F\" '{ print $2 }')
    rm -rf $MONGO_DIR/$DATA_DIR/*

    echo -ne "Starting up mongod... "
    T="$(date +%s)"
    ./bin/mongod --config configs/$config &

    # wait for mongo to start
    while [ 1 ]; do
        ./bin/mongostat -n 1 > /dev/null 2>&1
        if [ "$?" -eq 0 ]; then
            break
        fi
        sleep 2
    done
    T="$(($(date +%s)-T))"
    echo "took $T seconds"

    T="$(date +%s)"
    for l in $(seq 1 $NUM_LOOPS); do
        echo -ne "\rRunning import loop $l"
        ./bin/mongo benchmark --quiet insert.js >/dev/null 2>&1
```

```
    done
    T="$(($(date +%s)-T))"

    echo
    echo "Insert performance for $config: $T seconds"

    echo -ne "Shutting down server... "
    T="$(date +%s)"
    ./bin/mongo admin --quiet --eval "db.shutdownServer({force: true})" >/
     dev/null 2>&1

    while [ 1 ]; do
        pgrep -U $USER mongod > /dev/null 2>&1
        if [ "$?" -eq 1 ]; then
            break
        fi
        sleep 1
    done
    T="$(($(date +%s)-T))"
    echo "took $T seconds"

    SIZE=$(du -s --block-size=1 $MONGO_DIR/$DATA_DIR | cut -f1)
    SIZE_MB=$(echo "scale=2; $SIZE/(1024*1024)" | bc)
    echo "Disk usage for $config: ${SIZE_MB}MB"
done
```

This script assumes a legacy MongoDB installation in the bin directory inside the same directory as the script, and starts up a MongoDB instance according to the given configuration parameter. Therefore, you should make sure that you stop your MongoDB instance before running this script.

10.3.3 Insertion benchmark results

After running this batch script, you'll see output that looks like the following. The timings may be different on each machine, depending on the hardware and even on the operating system's kernel. This output was generated on a QuadCore i5-3570K running at 3.4 GHz, with 16 GB of RAM, and the storage medium being an ext4-formatted LVM partition, spread over two Hitachi Deskstar T7K250 250 GB disks connected to a SATA port, with a 7200 RPM rotational speed. The system runs on an Ubuntu 14.04 machine with a default Linux kernel 3.13.0 for the Intel 64-bit architecture:

```
===== RUNNING mmapv1.conf =====
Cleaning up data directory
Starting up mongod... took 102 seconds
Running import loop 16
Insert performance for mmapv1.conf: 105 seconds
Shutting down server... took 1 seconds
Disk usage for mmapv1.conf: 4128.04MB
===== RUNNING wiredtiger-uncompressed.conf =====
Cleaning up data directory
Starting up mongod... took 2 seconds
Running import loop 16
Insert performance for wiredtiger-uncompressed.conf: 92 seconds
Shutting down server... took 3 seconds
```

```
Disk usage for wiredtiger-uncompressed.conf: 560.56MB
===== RUNNING wiredtiger-snappy.conf =====
Cleaning up data directory
Starting up mongod... took 1 seconds
Running import loop 16
Insert performance for wiredtiger-snappy.conf: 93 seconds
Shutting down server... took 2 seconds
Disk usage for wiredtiger-snappy.conf: 380.27MB
===== RUNNING wiredtiger-zlib.conf =====
Cleaning up data directory
Starting up mongod... took 2 seconds
Running import loop 16
Insert performance for wiredtiger-zlib.conf: 104 seconds
Shutting down server... took 3 seconds
Disk usage for wiredtiger-zlib.conf: 326.67MB
```

It timed the server startup, the duration of the insert job, and the server shutdown processes, and measured the disk use of the storage directory after shutdown. You can see the results in table 10.3.

Table 10.3 Comparing MMAPv1 and WiredTiger operations

	MMAPv1	WT	WT snappy	WT zlib
Starting up	102 sec	2 sec	1 sec	2 sec
Insert job	105 sec	92 sec	93 sec	104 sec
Shut down	1 sec	3 sec	2 sec	3 sec
Disk use	4128.04 MB	560.56 MB	380.27 MB	326.67 MB

From this table it appears that WiredTiger has a tremendous gain in the time to start up the server and initialize the storage directory. But this is a one-time process, and therefore not a crucial measurement. Another test of the startup and shutdown times of the MongoDB instances with a preinitialized storage directory showed that the MMAPv1 startup took four seconds. You'll see this in a test in the next section, where you'll put the read performance to the test.

There's little gain to be had for the insert job when using WiredTiger, although the difference can be considerable when dealing with more data than this data set. Also keep in mind that this test doesn't test running the insert job using multiple Mongo client connections simultaneously.

The most striking feature of these test results is the disk use: WiredTiger without compression uses less than 15% of what MMAPv1 uses, and if you add compression, it's less than 10%! You'll also see that the snappy compression configuration finds a good middle ground between the insert speeds and the disk use. It's almost as fast as a bare WiredTiger instance with no compression, but still gains 180 MB on this data set. The zlib configuration has better compression, but in exchange for this, it takes 10 seconds more to do the insert job.

10.3.4 Read performance scripts

Up until now you've benchmarked the write performance of the different storage configurations, but for some applications you might be more interested in the read performance. Here's the simple JavaScript that will fetch all records in the benchmark collection and sequentially go through each of them. Note that this doesn't benchmark searches and filtering, which would need a predefined set of values to search for:

```
c = db.benchmark.find();
while(c.hasNext()) c.next();
```

This simple JavaScript is put in a read.js file and run by the following read.sh shell script, which is similar to the insert script:

```
#!/bin/bash

export MONGO_DIR=/storage/mongodb
export NUM_LOOPS=16

configs=(
    mmapv1.conf
    wiredtiger-uncompressed.conf
    wiredtiger-snappy.conf
    wiredtiger-zlib.conf
)

sudo echo "Acquired root permissions"

cd $MONGO_DIR
for config in "${configs[@]}"; do
    echo "===== RUNNING $config ====="
    echo "Clearing memory caches"
    sync
    echo 3 | sudo tee /proc/sys/vm/drop_caches

    echo -ne "Starting up mongod... "
    T="$(date +%s)"
    ./bin/mongod --config configs/$config &

    # wait for mongo to start
    while [ 1 ]; do
        ./bin/mongostat -n 1 > /dev/null 2>&1
        if [ "$?" -eq 0 ]; then
            break
        fi
        sleep 2
    done
    T="$(($(date +%s)-T))"
    echo "took $T seconds"

    rm -f timings-${config}.txt
    T="$(date +%s)"
    for l in $(seq 1 $NUM_LOOPS); do
        echo -ne "\rRunning read loop $l"
        /usr/bin/time -f "%e" -o timings-${config}.txt -a --quiet ./bin/mongo
     benchmark --quiet read.js >/dev/null 2>&1
```

```
    done
    T="$(($(date +%s)-T))"

    echo
    echo "Read performance for $config: $T seconds"

    echo -ne "Shutting down server... "
    T="$(date +%s)"
    ./bin/mongo admin --quiet --eval "db.shutdownServer({force: true})" >/
     dev/null 2>&1

    while [ 1 ]; do
        pgrep -U $USER mongod > /dev/null 2>&1
        if [ "$?" -eq 1 ]; then
            break
        fi
        sleep 1
    done
    T="$(($(date +%s)-T))"
    echo "took $T seconds"
done
```

This script also times the startup and shutdown processes of the MongoDB server instance again, so you can see that it starts up faster with a storage directory that's already initialized. Because the script needs to clear the memory caches to get an accurate timing of cold fetches—that is, fetches that aren't from the cache, but from the storage system itself—you'll need to enter your password to give it `sudo` access so it can flush the memory cache.

10.3.5 Read performance results

When running the script, you'll see an output similar to the following:

```
===== RUNNING mmapv1.conf =====
Clearing memory caches
3
Starting up mongod... took 3 seconds
Running read loop 16
Read performance for mmapv1.conf: 33 seconds
Shutting down server... took 1 seconds
===== RUNNING wiredtiger-uncompressed.conf =====
Clearing memory caches
3
Starting up mongod... took 2 seconds
Running read loop 16
Read performance for wiredtiger-uncompressed.conf: 23 seconds
Shutting down server... took 2 seconds
===== RUNNING wiredtiger-snappy.conf =====
Clearing memory caches
3
Starting up mongod... took 3 seconds
Running read loop 16
Read performance for wiredtiger-snappy.conf: 21 seconds
Shutting down server... took 1 seconds
```

```
===== RUNNING wiredtiger-zlib.conf =====
Clearing memory caches
3
Starting up mongod... took 2 seconds
Running read loop 16
Read performance for wiredtiger-zlib.conf: 21 seconds
Shutting down server... took 1 seconds
```

The `3`'s on their own lines are from the echo to clear the memory caches, so you can ignore those. You can now clearly see that the startup and shutdown times are similar for all MongoDB configurations, as shown in table 10.4.

Table 10.4 Comparing shutdown and startup times of various configurations

	MMAPv1	WT	WT snappy	WT zlib
Starting up	3 sec	2 sec	3 sec	2 sec
Read job	33 sec	23 sec	21 sec	21 sec
Shut down	1 sec	2 sec	1 sec	1 sec

The read job took at least 10 seconds longer on the MMAPv1 configuration than on the WiredTiger configurations. If you check the timings files, you'll see why. The timings have been graphed in figure 10.1.

It's clear that the first iteration took the longest because each subsequent iteration will take the results directly from the memory cache. For the cold fetch, MMAPv1 is clearly the slowest. The compressed configurations of WiredTiger have the best performance during the cold fetch. But for cached results, MMAPv1 is slightly faster than

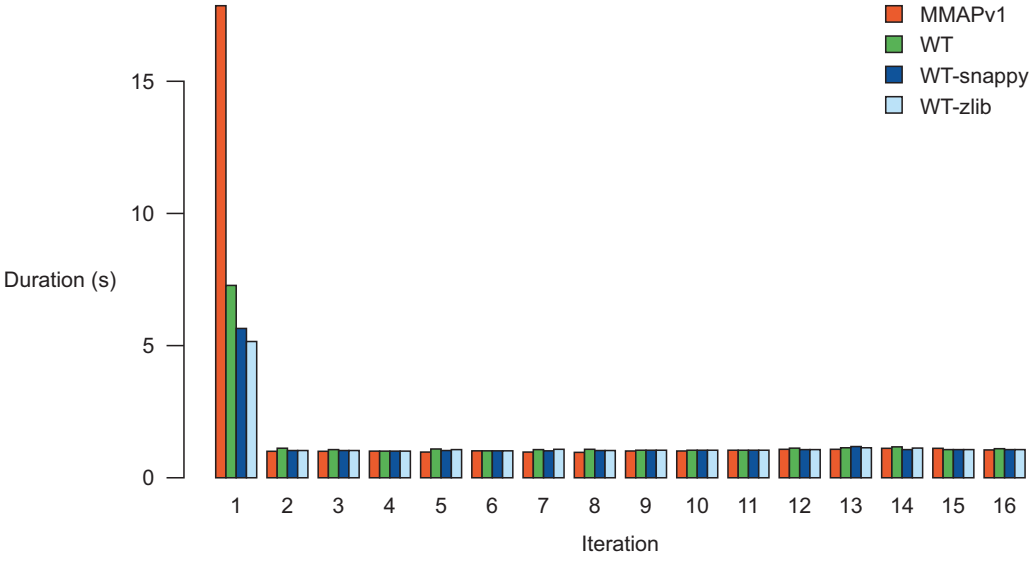

Figure 10.1 The read performance of MMAPv1 and WiredTiger

the WiredTiger alternatives. The graph doesn't show this clearly, but the timings files do, and their contents are shown in the table 10.5.

Table 10.5 Timing fetching operations using various storage engines (all times are in seconds)

	MMAPv1	WT	WT-snappy	WT-zlib
1	17.88	7.37	5.67	5.17
2	1.01	1.1	1.05	1.05
3	1	1.05	1.02	1.08
4	1.03	1.03	1.05	1.08
5	0.99	1.07	1.04	1.08
6	1.03	1.03	1.05	1.08
7	0.96	1.06	0.99	1.07
8	0.99	1.08	1.01	1.06
9	0.97	1.03	1.02	1.03
10	0.96	1.03	1.03	1.03
11	0.99	1.03	1.06	1.06
12	1.01	1.07	1.06	1.07
13	0.98	1.04	1.08	1.06
14	1.01	1.08	1.03	1.07
15	1.08	1.05	1.05	1.05
16	1.02	1.06	1.04	1.06

10.3.6 Benchmark conclusion

How much better is WiredTiger is than MMAPv1 in different aspects? You've seen the server startup and shutdown times, the insertion of thousands of averagely sized documents, and the fetching and iterating through these documents, repeatedly. You've also looked at the disk use of the storage directories.

You haven't tested how the MongoDB instances perform when there are several clients connecting and making requests at the same time. You also haven't tested the random search and filtering performances. These two are what happens most in a real-world example, and require a more complicated benchmark testing setup than what is used in this chapter. We hope that the benchmark examples give you some insight into how you want to benchmark the other aspects of the MongoDB installations.

From the results in this chapter, you can conclude that there's a tremendous gain to be had in terms of disk use. For small-scale applications where resource use is a concern, this will be the deciding factor, and you should go with a compressed version of WiredTiger. The zlib version will give the best performance-versus-cost ratio. For critical

applications where extra storage cost is worthwhile, the WiredTiger configuration without compression, or, if needed, using the snappy compression algorithm, will give slightly better speeds over the zlib configuration.

Even when disk storage isn't an issue for enterprise users, the speed of cold fetches will often be an important factor. This is especially true for social network sites where every visitor will have specific filters, so that cache misses will often occur.

Again, remember that the benchmarks in this chapter aren't totally representative of real-world situations, and therefore no hard conclusions should be drawn from the results of these benchmarks. But we hope that these benchmarks will give you a basic idea of what you can do with the data in your application, and that you'll be able to tune the benchmark scripts to best match the workload in your application. This way you'll be able to draw better conclusions as to what storage engine is better for your application's specific use case.

There are also several other environmental factors that depend on the hardware and software (OS kernel) configuration of your system. They haven't been taken into account in this chapter, but they may affect the performance of these benchmarks. When comparing storage engines, one should remember to fix all the environmental factors, even though some storage systems may perform better with a certain operating system setting, which may have a negative effect on other storage systems. Therefore, you should be cautious when drawing conclusions from such benchmarks.

10.4 Other examples of pluggable storage engines

We talked about WiredTiger in the previous sections and how it performs against MMAPv1 in terms of disk use and read/write speeds. In this section we'll introduce you to several other storage engines that are available.

An example is RocksDB, developed by Fac, and based on Google's Level DB with significant inspiration from Apache's Hbase. RocksDB is a key-value store that was developed to exploit the full potential of high read/write rates offered by Flash or RAM subsystems to offer low-latency database access by using an LSM tree engine.[5] This makes RocksDB a good storage plugin for applications that need high write performance, such as social media applications. MongoDB can be built with RocksDB support so that you can also enjoy the performance of RocksDB. For more information on how to do this, check out the announcement on the RocksDB blog.[6]

Tokutek's TokuFT (formerly known as TokuKV) is another key-value store that uses fractal tree indexing[7] instead of the default B-tree data structure used in WiredTiger.

---

[5] Log Structure Merge trees are data structures that outperform B-trees in the most useful record updating scenarios. The WiredTiger team compared B-trees vs. LSM trees: https://github.com/wiredtiger/wiredtiger/wiki/Btree-vs-LSM. For more information on LSM trees, see https://en.wikipedia.org/wiki/Log-structured_merge-tree.

[6] See http://rocksdb.org/blog/1967/integrating-rocksdb-with-mongodb-2/ for more information.

[7] Fractal tree indexing is a Tokutek innovation that keeps write performance consistent while your data set grows larger. See http://www.tokutek.com/resources/technology/.

Before MongoDB version 3.0, Tokutek offered a fork of MongoDB called TokuMX, which uses their TokuFT technology as the storage engine. Because MongoDB supports pluggable storage engines, Tokutek is offering a storage engine–only solution based on the MongoDB Pluggable Storage API, called TokuMXse,[8] where se stands for storage engine. While the TokuMX fork incorporates many features, such as clustering indexes and fast read-free updates by updating the MongoDB codebase, the TokuMXse version can't have them because of how the Pluggable Storage Engine API is currently designed. In this sense, TokuMXse will empower your standard MongoDB installation with reliably high-performance storage and compression.

Tokutek has been acquired by Percona, so all the TokuFT-related development will be done by Percona. Percona has announced an experimental build of MongoDB using TokuMXse.[9]

A last example that's familiar to MySQL users is InnoDB. This engine is used by MySQL as an alternative to the sequential MyISAM storage engine. InnoDB makes use of a transaction log that's replayed when a crash has occurred to get the latest operations from before the crash synchronized to disk, similar to how journaling works in WiredTiger and MMAPv1. MyISAM has to go through the whole database after a crash to repair any indexes or tables that haven't been flushed to disk, and as the database grows in size, this has an impact on the availability of the database. InnoDB doesn't have that problem and therefore offers better availability on bigger databases. The InnoDB technology isn't partial to MySQL. There's a fork from InnoDB called XtraDB, which is used by MariaDB, and it's possible that there will be an InnoDB module for MongoDB in the future.

10.5 Advanced topics

In this section we'll talk about the more advanced topics to get a better understanding of how storage engines work and why there are so many different ones with different technologies out there. The material covered in this section is absolutely not necessary to get a MongoDB instance running with one of the storage engines, but provides nice background information for those who are curious to know more.

10.5.1 How does a pluggable storage engine work?

The MongoDB source code comes with special classes to deal with storage modules. At the time of writing, the MongoDB source code was exposing the following classes:

- `StorageEngine`—A virtual class that forms the basis of the storage engines
- `MMAPV1Engine`—The MMAPv1 storage engine plugin class
- `KVStorageEngine`—The key-value-based storage engine class

[8] Announced in January 2015 by Tokutek: http://www.tokutek.com/2015/01/announcing-tokumxse-v1-0-0-rc-0/.

[9] Percona announcement in May 2015: https://www.percona.com/blog/2015/05/08/mongodb-percona-tokumxse-experimental-build-rc5-available/.

- `KVEngine`—The key-value engine that's used by the `KVStorageEngine` class
- MMAPV1DatabaseCatalogEntry—The catalog entry class for MMAPv1-based databases
- `RecordStore`—The base class for record store objects
- `RecordStoreV1Base`—The base class for MMAPv1 record store objects; the MMAPV1DatabaseCatalogEntry uses these
- `WiredTigerKVEngine`—The WiredTiger engine
- `WiredTigerFactory`—The factory class that creates a `KVStorageEngine` that uses the `WiredTigerKVEngine` class for its key-value engine
- `WiredTigerRecordStore`—The record store class that's used by the `WiredTigerKVEngine` class

You can see these classes in figure 10.2.

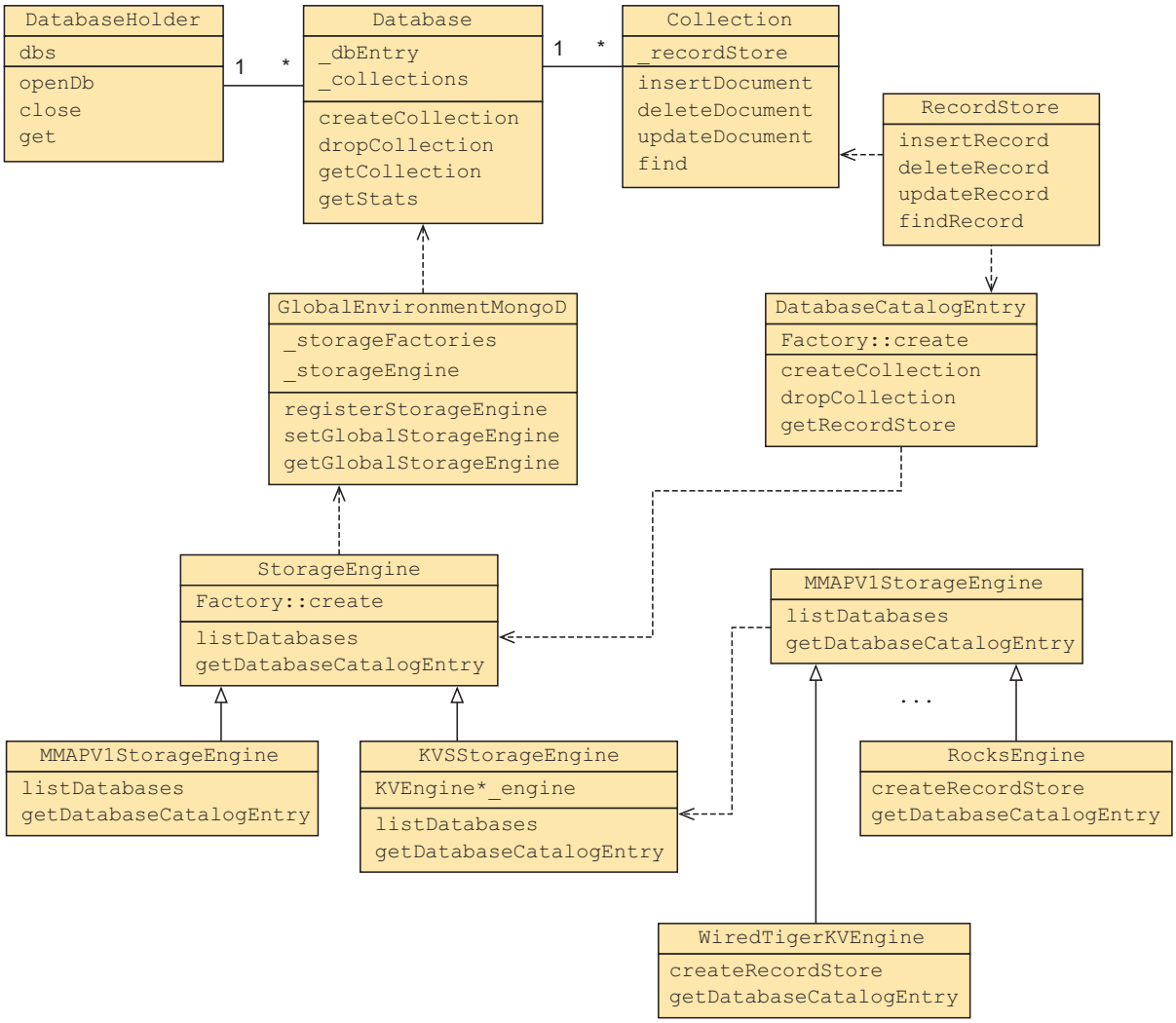

Figure 10.2 MongoDB multiple storage engines support mechanisms

There are three levels:

- *The storage engine at the top*—This works as the interface for MongoDB to the storage plugin.
- *The storage management*—This is the catalog entry class in the MMAPv1 case, and the key-value store (`KVEngine`) in the key-value-based storage engine case.
- *The record store*—This manages the actual MongoDB data, such as collections and entries. At this level, it communicates with the actual storage engine for the CRUD operations.

Together, these levels form the translator, if you will, between how MongoDB sees their data structures and how the storage engines store them. What MongoDB cares about is its own data structure—BSON—and its own scripting language based on JavaScript, which is used by the MongoDB drivers to communicate with MongoDB. How the data is stored or collected is left to the storage engine.

10.5.2 Data structure

In the MongoDB Pluggable Storage Engine API there are base classes for key-value storage systems. A key-value storage system is a storage system that guarantees fast access to the records by looking up the keys in the collection. You can tell the storage system to index certain keys for faster lookup.

A common data structure for storing such key-value data is a B-tree, which is also used in the WiredTiger storage plugin. B-trees were invented by Rudolf Bayer and Ed McCreight while working at Boeing Research Labs. There are speculations that the B in B-tree stands for Bayer, named after Rudolf Bayer, but others say that it stands for Boeing, where it was developed. In any case, the origin of the name is a mystery, but what's more interesting is how it works.

The power of the B-tree lies in the fact that disk-based storage is done in blocks. Each piece of data that's stored on disk is put in a block on the filesystem. In a real-world situation, these blocks are usually 4 KB in size, which is the default for many filesystems such as ExtFS and NTFS. A larger block size will allow for much larger files to be stored on the disk, but then we're talking about files larger than 4 TB.

A B-tree storage system uses these blocks as its nodes to store the indexes to the data elements. Each node can hold at most 4096 bytes of index information, and this index information is basically sorted. In the following examples, assume that each node can hold at most three indexes, but keep in mind that in a real-world example that number is much larger.

A B-tree starts with a root node, and within this node, you can store data records using their keys. These data records may have pointers to the actual value assigned to the key, or may have that value inside the data record. The more interesting part is how these records are found, because that's what needs to be done fast. Each node has several data records, indexed by an index key, and between each record, there's a diamond that's a placeholder for a pointer to another node containing other data

records. The diamond placeholder to the left of the 12 goes to a whole tree of data records with index values less than 12. Similarly, the whole tree that's pointed to by the diamond between 12 and 65 contains records with index values all between 12 and 65. This way, when doing a search, an algorithm will quickly know which node to traverse into.

For this to work well, it's important that the index keys in each node of a B-tree structure be sorted. To optimize the disk seek times, it's preferable to store as much data as you can within a node that's optimally the same size as a disk block; in most real-world examples this size is 4096 bytes.

Imagine an empty database with an empty root node. You start adding records with certain keys in an arbitrary order, and once the node becomes full, you create new nodes for the next data records to be added. Let's say you have the following keys to be added in this order:

{ 2, 63, 42, 48, 1, 62, 4, 24, 8, 23, 58, 99, 38, 41, 81, 30, 17, 47, 66 }

Remember that you use a maximum node size of three elements. You'll have a root node with the keys shown in figure 10.3, which are the first three elements from the example sequence, in sorted order.

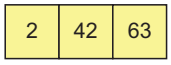

Figure 10.3 Root node with three elements

When you add the fourth element from the sequence, which is 48, it will create a new node between 42 and 63, and put 48 there, because 48 is between 42 and 63. The next element, 1, will be in the node before 2, because 1 is less than 2. Then the next element, 62, will join 48 in its node, because 62 lies between 42 and 63. The following three keys—4, 24, and 8—will go into the node that's pointed to between 2 and 42, because these numbers are all between 2 and 42. Within this new node, the keys will be sorted, as is always the case within nodes. After these first steps, the node tree graph will look like figure 10.4. Note how the numbers are sorted within each node.

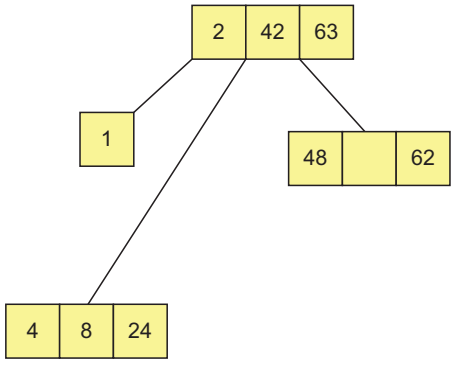

Figure 10.4 Early version of B-tree

You continue along the sequence. The next key, 23, is between 2 and 42 and will go into the node containing 4, 8, and 24, but because that node is already full, it will go into a new node between 8 and 24, because 23 is between 8 and 24. Then the next one, 58, will join 48 and 62. Key 99 will be in a new node pointed to after 63, because

99 is more than 63. Key 38, being between 2 and 42, will visit the node containing 4, 8, and 24, but because that node is full, it will go into a new node after 24, because 38 is more than 24. Key 41 joins key 38 in that new node at the second level. Key 81 joins 99, and 30 joins 38 and 41. Key 17 joins key 23 in the second-level node. Key 47 lies between 42 and 63, and so it goes to the node that holds 48, 58, and 62. Because that node is already full, it goes a level deeper, into a node before 48. Finally, key 66 will join 81 and 99, because 66 is more than 63. The final structure will look like figure 10.5, with the keys in each node in a sorted order.

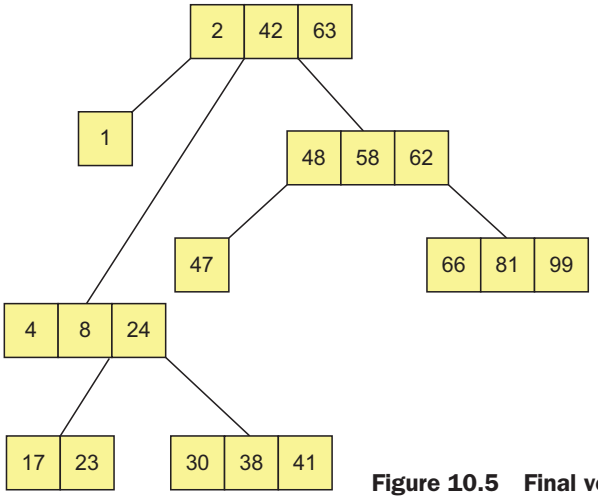

Figure 10.5 Final version of B-tree

Because the nodes contain the data in a sorted order, you will find out where you have to be very quickly. It will also be very fast at adding new records into the tree, as it will traverse through the tree at the same speed as finding a record, until it finds a free slot in a node, or until it needs to create a new node from a full node.

10.5.3 Locking

In the early days of MongoDB, locking was done for every connection, on a server level, using mutexes (mutual exclusions), a mechanism that ensures that multiple clients or threads can access the same resource—in this case, the database server—but not simultaneously. This is the worst method of locking, especially if you want to be a Big Data database engine, accessed by thousands of clients at the same time.

This changed with version 2.2, which is when database-level locking was implemented. The mutexes were applied on databases instead of on the whole MongoDB server instance, which was an improvement. But MongoDB threads would try to acquire subsequent write locks in a queue and deal with them in a serial fashion, letting the threads continue whenever the currently locking thread released its lock. While in a small-sized database, this can be rather fast, without any significant performance impact; in larger databases with thousands of write requests per second, this will become problematic and degrade the application's performance.

This changed with MongoDB version 3.0, which introduced collection-level locking for the MMAPv1 storage engine. This moved the locking mechanism a level lower, to collections. This means that multiple requests will be served simultaneously, without blocking each other, as long as they're writing into different collections.

With WiredTiger in version 3.0, MongoDB also supports document-level locking. This is an even more granular level of locking—multiple requests can now access the same collection simultaneously without blocking each other, as long as they aren't writing into the same document.

10.6 Summary

MongoDB version 3.0 introduced the concept of a pluggable storage engine architecture, and in this chapter you've seen what this entails and what it means for the system administrator or the application developer working with MongoDB. A real-world comparison has been given between a news site and a social network platform, both having different requirements for their database systems.

We compared the default engine that MongoDB used in previous versions, MMAPv1, with the newly bundled WiredTiger engine. WiredTiger performs well in both read and write operations, and offers document-level locking, whereas MMAPv1 doesn't go further than collection-level locking. WiredTiger isn't as greedy as MMAPv1 in terms of disk use, and it makes MongoDB a viable database system, even for small-scale applications. While MMAPv1 may be faster in certain situations, the performance versus cost ratio that WiredTiger offers far outweighs the performance versus cost ratio for MMAPv1.

While this comparison is based on a simple benchmark kit to give you a general idea of what benchmarking scripts do, it should give you insight into how to do your own benchmarks specific to your application's use case. The benchmarks are simple without taking into account several environmental factors, such as the hardware and software (OS kernel) configuration of your system. Tweaking these may affect the results of your benchmarks, so be cautious when comparing the results of the different storage engines.

You also learned about a few other storage engine platforms such as RocksDB and TokuFT, both of which are still in the experimental phase, but offer interesting features, because RocksDB is based on LSM trees and TokuFT is based on fractal trees. It will be interesting to see how they perform against WiredTiger in different situations.

You also read about some advanced concepts that come into play when designing a storage engine. Maximizing read and write performance, minimizing disk I/O, and offering optimal concurrency by using smart locking mechanisms on higher levels all play a crucial role in developing a storage engine.

Enough with the theory; the topic of the next chapter is used in true production databases—replication.