**MongoDB Introduction**

MongoDB is a powerful, flexible, and scalable data store. It combines the ability to scale out with many of the most useful features of relational databases, such as secondary indexes, range queries, and sorting. MongoDB is also incredibly featureful: it has tons of useful features such as built-in support for MapReduce-style aggregation and geospatial indexes.

There is no point in creating a great technology if it’s impossible to work with, so a lot of effort has been put into making MongoDB easy to get started with and a pleasure to use. MongoDB has a developer-friendly data model, administrator-friendly configuration options, and natural-feeling language APIs presented by drivers and the database shell. MongoDB tries to get out of your way, letting you program instead of worrying about storing data.

A Rich Data Model

MongoDB is a *document-oriented* database, not a relational one. The primary reason for moving away from the relational model is to make scaling out easier, but there are some other advantages as well. The basic idea is to replace the concept of a “row” with a more flexible model, the “document.” By allowing embedded documents and arrays, the document-oriented approach makes it possible to represent complex hierarchical relationships with a single record. This fits very naturally into the way developers in modern object-oriented languages think about their data.

MongoDB is also schema-free: a document’s keys are not predefined or fixed in any way. Without a schema to change, massive data migrations are usually unnecessary. New or missing keys can be dealt with at the application level, instead of forcing all data to have the same shape. This gives developers a lot of flexibility in how they work with evolving data models.

Easy Scaling

Data set sizes for applications are growing at an incredible pace. Advances in sensor technology, increases in available bandwidth, and the popularity of handheld devices that can be connected to the Internet have created an environment where even smallscale applications need to store more data than many databases were meant to handle.

A terabyte of data, once an unheard-of amount of information, is now commonplace. As the amount of data that developers need to store grows, developers face a difficult decision: how should they scale their databases? Scaling a database comes down to the choice between scaling up (getting a bigger machine) or scaling out (partitioning data across more machines). Scaling up is often the path of least resistance, but it has drawbacks: large machines are often very expensive, and eventually a physical limit is reached where a more powerful machine cannot be purchased at any cost. For the type of large web application that most people aspire to build, it is either impossible or not cost-effective to run off of one machine. Alternatively, it is both extensible and economical to scale *out*: to add storage space or increase performance, you can buy another commodity server and add it to your cluster.

MongoDB was designed from the beginning to scale out. Its document-oriented data model allows it to automatically split up data across multiple servers. It can balance data and load across a cluster, redistributing documents automatically. This allows developers to focus on programming the application, not scaling it. When they need more capacity, they can just add new machines to the cluster and let the database figure out how to organize everything.

Tons of Features…

It’s difficult to quantify what a feature is: anything above and beyond what a relational database provides? Memcached? Other document-oriented databases? However, no matter what the baseline is, MongoDB has some really nice, unique tools that are not (all) present in any other solution.

*Indexing*

MongoDB supports generic secondary indexes, allowing a variety of fast queries, and provides unique, compound, and geospatial indexing capabilities as well.

*Stored JavaScript*

Instead of stored procedures, developers can store and use JavaScript functions and values on the server side.

*Aggregation*

MongoDB supports MapReduce and other aggregation tools.

*Fixed-size collections*

Capped collections are fixed in size and are useful for certain types of data, such as logs.

*File storage*

MongoDB supports an easy-to-use protocol for storing large files and file metadata. Some features common to relational databases are not present in MongoDB, notably joins and complex multirow transactions. These are architectural decisions to allow for scalability, because both of those features are difficult to provide efficiently in a distributed system.

…Without Sacrificing Speed

Incredible performance is a major goal for MongoDB and has shaped many design decisions. MongoDB uses a binary wire protocol as the primary mode of interaction with the server (as opposed to a protocol with more overhead, like HTTP/REST). It adds dynamic padding to documents and preallocates data files to trade extra space usage for consistent performance. It uses memory-mapped files in the default storage engine, which pushes the responsibility for memory management to the operating system. It also features a dynamic query optimizer that “remembers” the fastest way to perform a query. In short, almost every aspect of MongoDB was designed to maintain high performance. Although MongoDB is powerful and attempts to keep many features from relational systems, it is not intended to do everything that a relational database does. Whenever possible, the database server offloads processing and logic to the client side (handled either by the drivers or by a user’s application code). Maintaining this streamlined design is one of the reasons MongoDB can achieve such high performance.

Simple Administration

MongoDB tries to simplify database administration by making servers administrate themselves as much as possible. Aside from starting the database server, very little administration is necessary. If a master server goes down, MongoDB can automatically failover to a backup slave and promote the slave to a master. In a distributed environment, the cluster needs to be told only that a new node exists to automatically integrate and configure it.

MongoDB’s administration philosophy is that the server should handle as much of the configuration as possible automatically, allowing (but not requiring) users to tweak their setups if needed.

But Wait, That’s Not All…

Throughout the course of the book, we will take the time to note the reasoning or motivation behind particular decisions made in the development of MongoDB. Through those notes we hope to share the philosophy behind MongoDB. The best way to summarize the MongoDB project, however, is through its main focus—to create a full-featured data store that is scalable, flexible, and fast.

Getting Started

MongoDB is very powerful, but it is still easy to get started with. In this chapter we’ll introduce some of the basic concepts of MongoDB:

• A *document* is the basic unit of data for MongoDB, roughly equivalent to a row in a relational database management system (but much more expressive).

• Similarly, a *collection* can be thought of as the schema-free equivalent of a table.

• A single instance of MongoDB can host multiple independent *databases*, each of which can have its own collections and permissions.

• MongoDB comes with a simple but powerful JavaScript *shell*, which is useful for the administration of MongoDB instances and data manipulation.

• Every document has a special key, "\_id", that is unique across the document’s collection.

Documents

At the heart of MongoDB is the concept of a *document*: an ordered set of keys with associated values. The representation of a document differs by programming language, but most languages have a data structure that is a natural fit, such as a map, hash, or dictionary. In JavaScript, for example, documents are represented as objects:

{"greeting" : "Hello, world!"}

This simple document contains a single key, "greeting", with a value of "Hello, world!". Most documents will be more complex than this simple one and often will contain multiple key/value pairs:

{"greeting" : "Hello, world!", "foo" : 3}

This example is a good illustration of several important concepts:

• Key/value pairs in documents are ordered—the earlier document is distinct from the following document:

{"foo" : 3, "greeting" : "Hello, world!"}

• Values in documents are not just “blobs.” They can be one of several different data types (or even an entire embedded document

In this example the value for "greeting" is a string, whereas the value for "foo" is an integer.

The keys in a document are strings. Any UTF-8 character is allowed in a key, with a few notable exceptions:

• Keys must not contain the character \0 (the null character). This character is used to signify the end of a key.

• The . and $ characters have some special properties and should be used only in certain circumstances, as described in later chapters. In general, they should be considered reserved, and drivers will complain if they are used inappropriately.

• Keys starting with \_ should be considered reserved; although this is not strictly enforced.

MongoDB is type-sensitive and case-sensitive. For example, these documents are distinct:

{"foo" : 3}

{"foo" : "3"}

As are as these:

{"foo" : 3}

{"Foo" : 3}

A final important thing to note is that documents in MongoDB cannot contain duplicate keys. For example, the following is not a legal document:

{"greeting" : "Hello, world!", "greeting" : "Hello, MongoDB!"}

Collections

A *collection* is a group of documents. If a document is the MongoDB analog of a row in a relational database, then a collection can be thought of as the analog to a table.

Schema-Free

Collections are *schema-free*. This means that the documents within a single collection can have any number of different “shapes.” For example, both of the following documents could be stored in a single collection:

{"greeting" : "Hello, world!"}

{"foo" : 5}

Note that the previous documents not only have different types for their values (string versus integer) but also have entirely different keys. Because any document can be put into any collection, the question often arises: “Why do we need separate collections at all?” It’s a good question—with no need for separate schemas for different kinds of documents, why *should* we use more than one collection? There are several good reasons:

• Keeping different kinds of documents in the same collection can be a nightmare for developers and admins. Developers need to make sure that each query is only returning documents of a certain kind or that the application code performing a query can handle documents of different shapes. If we’re querying for blog posts, it’s a hassle to weed out documents containing author data.

• It is much faster to get a list of collections than to extract a list of the types in a collection. For example, if we had a type key in the collection that said whether each document was a “skim,” “whole,” or “chunky monkey” document, it would be much slower to find those three values in a single collection than to have three separate collections and query for their names

• Grouping documents of the same kind together in the same collection allows for data locality. Getting several blog posts from a collection containing only posts will likely require fewer disk seeks than getting the same posts from a collection containing posts and author data.

• We begin to impose some structure on our documents when we create indexes. (This is especially true in the case of unique indexes.) These indexes are defined per collection. By putting only documents of a single type into the same collection, we can index our collections more efficiently. As you can see, there are sound reasons for creating a schema and for grouping related types of documents together. MongoDB just relaxes this requirement and allows developers more flexibility.

Naming

A collection is identified by its name. Collection names can be any UTF-8 string, with a few restrictions:

• The empty string ("") is not a valid collection name.

• Collection names may not contain the character \0 (the null character) because this delineates the end of a collection name.

• You should not create any collections that start with *system.*, a prefix reserved for system collections. For example, the *system.users* collection contains the database’s users, and the *system.namespaces* collection contains information about all of the database’s collections.

• User-created collections should not contain the reserved character $ in the name. The various drivers available for the database do support using $ in collection names because some system-generated collections contain it. You should not use $ in a name unless you are accessing one of these collections.

One convention for organizing collections is to use namespaced subcollections separated by the . character. For example, an application containing a blog might have a collection named *blog.posts* and a separate collection named *blog.authors*. This is for organizational purposes only—there is no relationship between the *blog* collection (it doesn’t even have to exist) and its “children.” Although subcollections do not have any special properties, they are useful and incorporated into many MongoDB tools:

• GridFS, a protocol for storing large files, uses subcollections to store file metadata separately from content chunks.

• The MongoDB web console organizes the data in its DBTOP section by subcollection

• Most drivers provide some syntactic sugar for accessing a subcollection of a given collection. For example, in the database shell, db.blog will give you the *blog* collection, and db.blog.posts will give you the *blog.posts* collection.

Subcollections are a great way to organize data in MongoDB, and their use is highly

recommended.

Databases

In addition to grouping documents by collection, MongoDB groups collections into *databases*. A single instance of MongoDB can host several databases, each of which can be thought of as completely independent. A database has its own permissions, and each database is stored in separate files on disk. A good rule of thumb is to store all data for a single application in the same database. Separate databases are useful when storing data for several application or users on the same MongoDB server.

Like collections, databases are identified by name. Database names can be any UTF-8 string, with the following restrictions:

• The empty string ("") is not a valid database name.

• A database name cannot contain any of these characters: ' ' (a single space), ., $, /,

\, or \0 (the null character).

• Database names should be all lowercase.

• Database names are limited to a maximum of 64 bytes.

One thing to remember about database names is that they will actually end up as files on your filesystem. This explains why many of the previous restrictions exist in the first place.

There are also several reserved database names, which you can access directly but have

special semantics. These are as follows:

*admin*

This is the “root” database, in terms of authentication. If a user is added to the *admin* database, the user automatically inherits permissions for all databases.

There are also certain server-wide commands that can be run only from the *admin* database, such as listing all of the databases or shutting down the server.

*local*

This database will never be replicated and can be used to store any collections that should be local to a single server.

*config*

When Mongo is being used in a sharded setup, the *config* database is used internally to store information about the shards. By prepending a collection’s name with its containing database, you can get a fully qualified collection name called a *namespace*. For instance, if you are using the *blog.posts* collection in the *cms* database, the namespace of that collection would be cms.blog.posts. Namespaces are limited to 121 bytes in length and, in practice, should be less than 100 bytes long. For more on namespaces and the internal representation of collections in MongoDB.

翻译

MongoDB介绍

MongoDB是一种强大，灵活，可拓展的数据存世方式。它拓展了关系型数据库的众多可用功能，如辅助索引，范围查询和排序。MongoDB的功能非常吩咐，比如内置的对MapReduce式聚合的支持，以及对地理空间索引的支持。

要是不能用的话，再牛的技术也是空谈，MongoDB致力于容易上手，便于使用。MongoDB的数据抹胸对于开发者来说非常友好，配置选项对于管理员来说也很轻松，并且有驱动程序和数据库shell提供的自然语言式的API。MongoDB会为你扫除障碍，让你关注编程本身而不是为了存储数据烦恼。

丰富的数据模型

MongoDB是面向对象的数据库，不是关系型数据库。放弃关系模型的主要原因是为了获得更加方便的拓展性，当然还有其他好处。

基本的思路就是将原来的行的概念换成更加灵活的文档模型。面向文档的方式可以将文档或者数组内嵌进来，所以一条记录就可以表示非常复杂的层次关系。使用面向对象语言的开发者恰恰这么看待数据，所以感觉非常自然。

MongoDB没有模式：文档的键不会事先定义也不会固定不变。由于没有模式需要更改，通常不需要迁移大量数据。不必将所有数据都放到一个模子里面，应用层可以处理新增或者丢失的键，这样开发者可以非常容易地变更数据模型。

容易拓展

应用数据集的大小在飞速增加。传感器技术的发展，贷款的增加，以及可连接到特网的手持设备的普及使得当下即便很小的应用也要存储大盆数据，最大到很多数据库都应付不来。T级别的数据原来是闻所未闻的，现在已经司空见惯了。

由于开发者要存储的数据不断增长，他们面临一个非常困难的选择：该如何扩一展他们的数据库？升级呢（买台更好的机器），还是扩展呢（将数据分散到很多机器上）？升级通常是最省力气的做法，但是问题也显而易见：大型机一般都非常昂贵，最后达到了物理极限的话花多少钱都买不到更好的机器。对于大多数人希望构建的大型Web应用来说，这样做既不现实也不划算。而扩展就不同了，不但经济而且还能持续添加：想要增加存储空间或者提升性能，只需要买台一般的服务器加入集群就好了。

MongoDB从最初设计的时候就考虑到了扩展的问题。它所采用的面向文档的数据模型使其可以自动在多台服务器之间分割数据。它还可以平衡集群的数据和负载，自动重排文档。这样开发者就可以专注于编写应用，而不是考虑如何扩展。要是需要更大的容最．只需在集群中添加新机器，然后让数据库来处理剩下的事。

丰富的功能

很难界定什么才算作一个功能：上面提及的算是功能吗？关系型数据库做不到的算吗？都可以说Memcached做不到的呢？其他面向文档的数据库做不到的又如何呢？但无论界定的标准是什么，都可以说MongoDB拥有一些真正独特的、好用的工具，其他方案不具备或不完全具备这些工具。

1. 索引

MongoDB支持通用辅助索引，能进行多种快速查询，也提供唯一的、复合的和地理空间索引能力。

1. 存储Javascript

开发人员不必使用存储过程了，可以直接在服务端存取JavascriPt的函数和值。

1. 聚合

MongoDB支持MapReduce和其他聚合工具，

1. 固定集合

集合的大小是有上限的，这对某些类型的数据（比如口志）特别有用。

1. 文件存储

MongoDB支持用一种容易使用的协议存储大型文件和文件的元数据．

有些关系型数据库的常见功能MongoDB并不具备，比如联接（join）和复杂的多行事务。这个架构上的考虑是为了提高拓展性，因为这两个功能实在很难在一个分布式系统上实现。

不牺牲速度

卓越的性能是MongoDB的主要目标，也极大地影响了设计上的很多决定。MongoDB使用MongoDB传输协议作为与服务器交互的主要方式（与之对应的协议需要更多的开销，如HT'I'PlREST）。它对文档进行动态填充，预分配数据文件，用空间换取性能的稳定。默认的存储引擎中使用了内存映射文件，将内存管理工作交给操作系统去处理。动态查询优化器会“记住”执行查询最高效的方式。

总之，MongoDB在各个方面都充分考虑了性能。虽然MongoDB功能强大，尽量保持关系型数据库的众多特性，但是它并不是要具备所有的关系型数据库的功能。它尽可能地将服务器端处理逻辑交给客户端（由驱动程序或者用户的应用程序处理）。这样精简的设计使得MongoDB获得了非常好的性能。

简便的管理

MongoDB尽量让服务器自治来简化数据库的管理。除了启动数据库服务器之外，几乎没有什么必要的管理操作。如果主服务器挂掉了，MongoDB会自动切换到备份服务器上，并且将备份服务器提升为活跃服务器。在分布式环境下，集群只需要知道有新增加的节点，就会自动集成和配置新节点。

MongoDB的管理理念就是尽可能地让服务器自动配置，让用户能在需要的时候调

整设置（但不强制）。

其他内容

在本书中，我们还会花些时间追溯一下在开发MongoDB的过程中一些决定的原因和动机，希望通过这种方式来阐释MongoDB的理念。毕竟，MongoDB的愿景是对自身最好的连释―建立一种灵活、高效、易于扩展、功能完备的数据库。

入门

MongoDB非常强大，同时也很容易上手。本章会介绍一些MongoDB的基本概念。

1. 文档是MongoDB中数据的基本单元，非常类似于关系数据库管理系统中的行（但是比行要复杂得多）。
2. 类似地，集合可以被看做是没有模式的表。
3. MongoDB的单个实例可以容纳多个独立的数据库，每一个都有自己的集合和权限。
4. MongoDB自带简洁但功能强大的Javascriptshell，这个工具对于管理MongooB实例和操作数据作用非常大。
5. 每一个文档都有一个特殊的键”idll，它在文档所处的集合中是唯一的。

文档

文档是MongoDB的核心概念。多个键及其关联的值有序地放置在一起便是文档。

每种编程语言表示文档的方法不太一样，但大多数编程语言都有相通的一种数据结

构，比如映射、散列或字典。例如，在JavascriPt里面，文档表示为对象：

{“greeting”: “Hello,world!''}

这个文档只有一个键greeting，其对应的值为Hello, world。绝大多数

情况下，文档会比这个简单的例子复杂得多，经常会包含多个键/值对：

{"greeting" : "Hello, world!", "foo" : 3}

这个例子很好地解释了几个十分重要的概念。

1. 文档中的键l值对是有序的，上面的文档和下面的文档是完全不同的：

{"foo" : 3, "greeting" : "Hello, world!"}

1. 文档中的值不仅可以是在双引号里面的字符串，还可以是其他几种数据类型（甚至可以是整个嵌入的文档）。这个例子中，greeting的值是个字符串，而foo，的值是个整数。

文档的键是字符串。除了少数例外情况，键可以使用任意UTF一8字符。

1. 键不能含有＼0（空字符）。这个字符用来表示键的结尾。
2. ．和$有特别的意义，只有在特定环境下才能使用，后面的章节会详细说明．通常来说就是被保留了，使用不当的话，驱动程序会提示。
3. 以下划线“\_”开头的键是保留的，虽然这个并不是严格要求的。

MongoDB不但区分类型，也区分大小写。例如，下面的两个文档是不同的：

{"foo":3}

{"foo":"3”,}

以下的文档也是不同的：

{"foo,,:3}

{"Foo":3}

还有一个非常重要的事项需要注意，MongoDB的文档不能有重复的键。例如，下

面的文档是非法的：

{"greeting":"Hello, world!", “greeting": "Hello, ongoDB!"}

集合

集合就是一组文档。如果说MongoDB中的文档类似于关系型数据库中的行，那么集合就如同表。

无模式

集合是无模式的。这意味着一个集合里面的文档可以是各式各样的。例如，下面两个文档可以存在于同一个集合里面：

{"greeting":"Hello, world!”,}

{"foo":"5”,}

注意，上面的文档不光是值的类型不同（字符串和整数），它们的键也是完全不一样的。因为集合里面可以放置任何文档，随之而来的一个问题是：“还有必要使用多个集合吗？”问得好！要是没必要对各种文档划分模式，那么为什么还要使用多个集合呢？下面是一些理由。

1. 把各种各样的文档都混在一个集合里面，无论对于开发者还是管理员来说都是盗梦。开发者要么确保每次查询只返回需要的文档种类，要么让执行查询的应用程序来处理所有不同类型的文档。如果查询博客文章还要glJ除那些含有作者数据的文档，就很令人恼火。
2. 在一个集合里面查询特定类型的文档在速度上也很不划算，分开做多个集合要快得多。例如，集合里面有个标注类型的键，现在查询其值为“skim”、“whole”或“monkey”的文档，就会非常慢。如果按照名字分割成3个集合的话，查询会快很多
3. 把同种类型的文档放在一个集合里，这样数据会更加集中。从只含有博客文章的集合里面查询几篇文章，会比从含有文章和作者数据的集合里面查出几篇文章少消耗磁盘寻道操作。
4. 当创建索引的时候，文档会有附加的结构（尤其是有唯一索引的时候）。索引是按照集合来定义的。把同种类型的文档放入同一个集合里面，可以使索引更加有效。你已经看到了，的确有很多理由创建一个模式把相关类型的文档规整到一起。但MongoDB对此还是不做强制要求，让开发者更有灵活性．

命名

我们可以通过名字来标识集合．集合名可以是满足下列条件的任意UTF-8字符串。

1. 集合名不能是空字符串””。
2. 集合名不能含有\0字符（空字符），这个字符表示集合名的结尾。
3. 集合名不能以“system.”开头，这是为系统集合保留的前缀。例如system.users这个集合保存着数据库的用户信息，system.namespace集合保存着所有数据库集合的信息。
4. 用户创建的集合名字不能含有保留字符$。有些驱动程序的确支持在集合名里面包

含$，这是因为某些系统生成的集合中包含该字符。除非你要访向这种系统创建的集合，否则千万不要在名字里出现$。

子集合

组织集合的一种惯例是使用“.”字符分开的按命名空间划分的子集合。例如，一个带有博客功能的应用可能包含两个集合，分别是blog.post。和blog.authors。这样做的目的只是为了使组织结构更好些，也就是说blog这个集合（这里根本就不需要存在）及其子集合没有任何关系。

虽然子集合没有特别的地方，但还是很有用，很多MongoDB工具中都包含子集合。

1. GridFs是一种存储大文件的协议，使用子集合来存储文件的元数据，这样就与内

容块分开了。

1. MongoDB的web控制台通过子集合的方式将数据组织在DBTOP部分。
2. 绝大多数驱动程序都提供语法糖，为访问指定集合的子集合提供方便。例如，在数据库shell里面，db.blog代表blog集合，db.blog.posts，代表blog.posts集合。

在MongoDB中使用子集合来组织数据是很好的方法，在此强烈推荐。

数据库

MongoDB中多个文档组成集合，同样多个集合可以组成数据库．一个MongoDB实例可以承载多个数据库，它们之间可视为完全独立的。每个数据库都有独立的权限控制，即便是在磁盘上，不同的数据库也放置在不同的文件中。将一个应用的所有数据都存储在同一个数据库中的做法就很好．要想在同一个MongQDB服务器上存放多个应用或者用户的数据，就要使用不同的数据库了。

和集合一样，数据库也通过名字来标识。数据库名可以是满足以下条件的任意

UTF-8字符串。

1. 不能是空字符串（""）。
2. 不得含有”（空格）、．、$、/、\、和\0（空字符）。
3. 应全部小写。
4. 最多64字节。

要记住一点，数据库名最终会变成文件系统里的文件，这也就是有如此多限制的原因。

有一些数据库名是保留的，可以直接访问这些有特殊作用的数据库．这些数据库如

下所示。

1. Admin

从权限的角度来看，这是“root”数据库。要是将一个用户添加到这个数据库，这个用户自动继承所有数据库的权限。一些特定的服务器端命令也只能从这个数据库运行，比如列出所有的数据库或者关闭服务器。

1. Local

这个数据永远不会被复制，可以用来存储限于本地单台服务器的任意集合。

1. Config

当Mongo用于分片设置时，config数据库在内部使用，用干保存分片的相关信息。

把数据库的名字放到集合名前面，得到就是集合的完全限定名，称为命名空间．例如，如果你在cms数据库中使用blog.posts集合，那么这个集合的命名空间就是cms.blog.posts。命名空间的长度不得超过121字节，在实际使用当中应该小于100字节。