Data Model Design for MongoDB

Release 2.4.9

MongoDB Documentation Project

January 27, 2014

Contents

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. This flexibility gives you data-modeling choices to match your application and its performance requirements.

Read the *Data Modeling Introduction* (page 3) document for a high level introduction to data modeling, and proceed to the documents in the *Data Modeling Concepts* (page 5) section for additional documentation of the data model design process. The *Data Model Examples and Patterns* (page 11) documents provide examples of different data models. In addition, the MongoDB Use Case Studies[1] provide overviews of application design and include example data models with MongoDB.

Data Modeling Introduction **(page 3)** An introduction to data modeling in MongoDB.

Data Modeling Concepts **(page 5)** The core documentation detailing the decisions you must make when determining a data model, and discussing considerations that should be taken into account.

Data Model Examples and Patterns **(page 11)** Examples of possible data models that you can use to structure your MongoDB documents.

Data Model Reference **(page 26)** Reference material for data modeling for developers of MongoDB applications.

[1] http://docs.mongodb.org/ecosystem/use-cases

1 Data Modeling Introduction

Data in MongoDB has a *flexible schema*. Unlike SQL databases, where you must determine and declare a table's schema before inserting data, MongoDB's *collections* do not enforce *document* structure. This flexibility facilitates the mapping of documents to an entity or an object. Each document can match the data fields of the represented entity, even if the data has substantial variation. In practice, however, the documents in a collection share a similar structure.

The key challenge in data modeling is balancing the needs of the application, the performance characteristics of the database engine, and the data retrieval patterns. When designing data models, always consider the application usage of the data (i.e. queries, updates, and processing of the data) as well as the inherent structure of the data itself.

1.1 Document Structure

The key decision in designing data models for MongoDB applications revolves around the structure of documents and how the application represents relationships between data. There are two tools that allow applications to represent these relationships: *references* and *embedded documents*.

References

References store the relationships between data by including links or *references* from one document to another. Applications can resolve these *references* (page 29) to access the related data. Broadly, these are *normalized* data models.

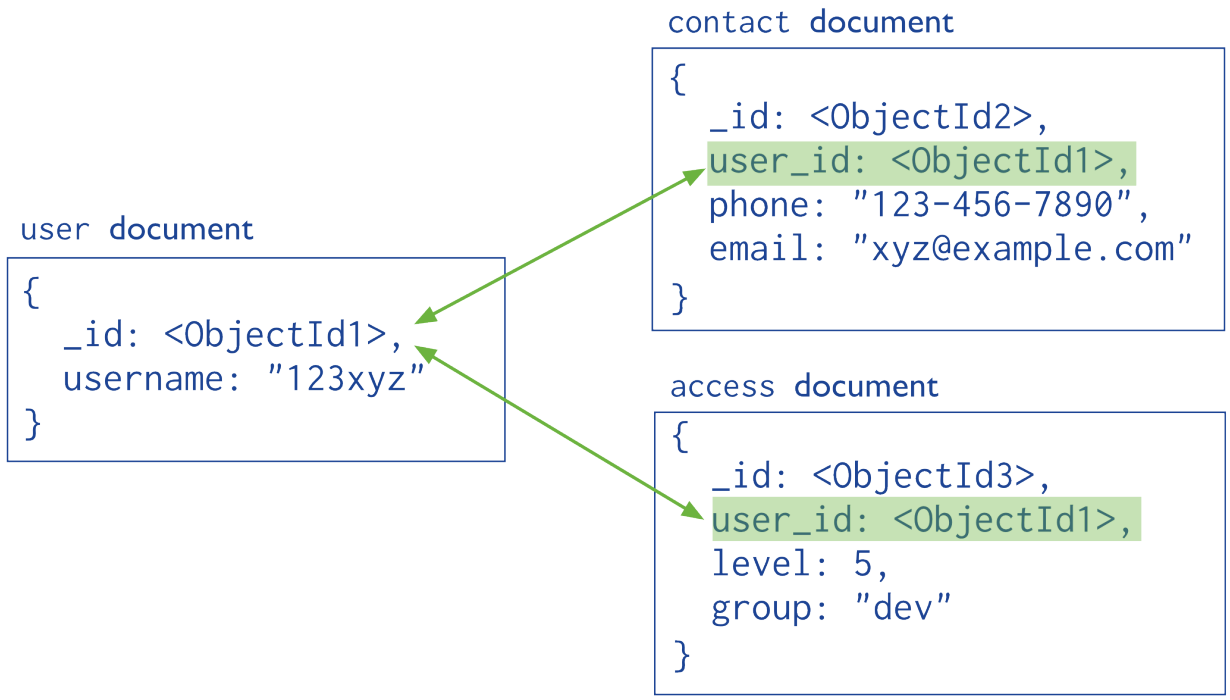

Figure 1: Data model using references to link documents. Both the `contact` document and the `access` document contain a reference to the `user` document.

See *Normalized Data Models* (page 6) for the strengths and weaknesses of using references.

Embedded Data

Embedded documents capture relationships between data by storing related data in a single document structure. MongoDB documents make it possible to embed document structures as sub-documents in a field or array within a document. These *denormalized* data models allow applications to retrieve and manipulate related data in a single database operation.

Figure 2: Data model with embedded fields that contain all related information.

See *Embedded Data Models* (page 5) for the strengths and weaknesses of embedding sub-documents.

1.2 Atomicity of Write Operations

In MongoDB, write operations are atomic at the *document* level, and no single write operation can atomically affect more than one document or more than one collection. A denormalized data model with embedded data combines all related data for a represented entity in a single document. This facilitates atomic write operations since a single write operation can insert or update the data for an entity. Normalizing the data would split the data across multiple collections and would require multiple write operations that are not atomic collectively.

However, schemas that facilitate atomic writes may limit ways that applications can use the data or may limit ways to modify applications. The *Atomicity Considerations* (page 8) documentation describes the challenge of designing a schema that balances flexibility and atomicity.

1.3 Document Growth

Some updates, such as pushing elements to an array or adding new fields, increase a *document's* size. If the document size exceeds the allocated space for that document, MongoDB relocates the document on disk. The growth consideration can affect the decision to normalize or denormalize data. See *Document Growth Considerations* (page 7) for more about planning for and managing document growth in MongoDB.

1.4 Data Use and Performance

When designing a data model, consider how applications will use your database. For instance, if your application only uses recently inserted documents, consider using `http://docs.mongodb.org/manualcore/capped-collections`. Or if your application needs are mainly read operations to a collection, adding indexes to support common queries can improve performance.

See *Operational Factors and Data Models* (page 6) for more information on these and other operational considerations that affect data model designs.

2 Data Modeling Concepts

When constructing a data model for your MongoDB collection, there are various options you can choose from, each of which has its strengths and weaknesses. The following sections guide you through key design decisions and detail various considerations for choosing the best data model for your application needs.

For a general introduction to data modeling in MongoDB, see the *Data Modeling Introduction* (page 3). For example data models, see *Data Modeling Examples and Patterns* (page 11).

Data Model Design **(page 5)** Presents the different strategies that you can choose from when determining your data model, their strengths and their weaknesses.

Operational Factors and Data Models **(page 6)** Details features you should keep in mind when designing your data model, such as lifecycle management, indexing, horizontal scalability, and document growth.

GridFS **(page 9)** GridFS is a specification for storing documents that exceeds the *BSON*-document size limit of 16MB.

2.1 Data Model Design

Effective data models support your application needs. The key consideration for the structure of your documents is the decision to *embed* (page 5) or to *use references* (page 6).

Embedded Data Models

With MongoDB, you may embed related data in a single structure or document. These schema are generally known as "denormalized" models, and take advantage of MongoDB's rich documents. Consider the following diagram:

Embedded data models allow applications to store related pieces of information in the same database record. As a result, applications may need to issue fewer queries and updates to complete common operations.

In general, use embedded data models when:

- you have "contains" relationships between entities. See *Model One-to-One Relationships with Embedded Documents* (page 12).
- you have one-to-many relationships between entities. In these relationships the "many" or child documents always appear with or are viewed in the context of the "one" or parent documents. See *Model One-to-Many Relationships with Embedded Documents* (page 13).

In general, embedding provides better performance for read operations, as well as the ability to request and retrieve related data in a single database operation. Embedded data models make it possible to update related data in a single atomic write operation.

However, embedding related data in documents may lead to situations where documents grow after creation. Document growth can impact write performance and lead to data fragmentation. See *Document Growth* (page 7) for

Figure 3: Data model with embedded fields that contain all related information.

details. Furthermore, documents in MongoDB must be smaller than the `maximum BSON document size`. For bulk binary data, consider *GridFS* (page 9).

To interact with embedded documents, use *dot notation* to "reach into" embedded documents. See *query for data in arrays* and *query data in sub-documents* for more examples on accessing data in arrays and embedded documents.

Normalized Data Models

Normalized data models describe relationships using *references* (page 29) between documents.

In general, use normalized data models:

- when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.
- to represent more complex many-to-many relationships.
- to model large hierarchical data sets.

References provides more flexibility than embedding. However, client-side applications must issue follow-up queries to resolve the references. In other words, normalized data models can require more roundtrips to the server.

See *Model One-to-Many Relationships with Document References* (page 14) for an example of referencing. For examples of various tree models using references, see *Model Tree Structures* (page 16).

2.2 Operational Factors and Data Models

Modeling application data for MongoDB depends on both the data itself, as well as the characteristics of MongoDB itself. For example, different data models may allow applications to use more efficient queries, increase the throughput of insert and update operations, or distribute activity to a sharded cluster more effectively.

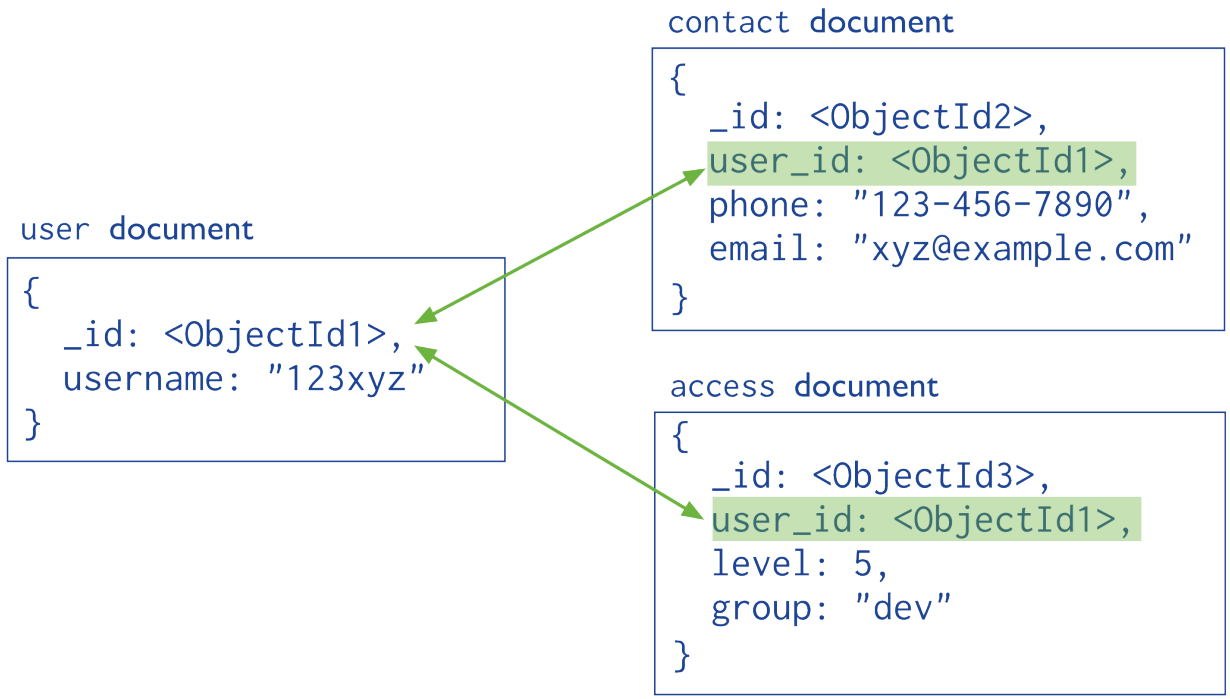

Figure 4: Data model using references to link documents. Both the `contact` document and the `access` document contain a reference to the `user` document.

These factors are *operational* or address requirements that arise outside of the application but impact the performance of MongoDB based applications. When developing a data model, analyze all of your application's `read operations` and `write operations` in conjunction with the following considerations.

Document Growth

Some updates to documents can increase the size of documents size. These updates include pushing elements to an array (i.e. `$push`) and adding new fields to a document. If the document size exceeds the allocated space for that document, MongoDB will relocate the document on disk. Relocating documents takes longer than *in place updates* updates and can lead to fragmented storage. Although MongoDB automatically `adds padding to document allocations` to minimize the likelihood of relocation, data models should avoid document growth when possible.

For instance, if your applications require updates that will cause document growth, you may want to refactor your data model to use references between data in distinct documents rather than a denormalized data model.

MongoDB adaptively adjusts the amount of automatic padding to reduce occurrences of relocation. You may also use a *pre-allocation* strategy to explicitly avoid document growth. Refer to Pre-Aggregated Reports Use Case[2] for an example of the *pre-allocation* approach to handling document growth.

[2] http://docs.mongodb.org/ecosystem/use-cases/pre-aggregated-reports

Atomicity

In MongoDB, operations are atomic at the *document* level: no **single** write operation can change more than one document or more than one collection. [3] Ensure that your application stores all fields with atomic dependency requirements in the same document. If the application can tolerate non-atomic updates for two pieces of data, you can store these data in separate documents.

A data model that embeds related data in a single document facilitates these kinds of atomic operations. For data models that store references between related pieces of data, the application must issue separate read and write operations to retrieve and modify these related pieces of data.

See *Model Data for Atomic Operations* (page 24) for an example data model that provides atomic updates for a single document.

Sharding

MongoDB uses *sharding* to provide horizontal scaling. These clusters support deployments with large data sets and high-throughput operations. Sharding allows users to *partition* a *collection* within a database to distribute the collection's documents across a number of `mongod` instances or *shards*.

To distribute data and application traffic in a sharded collection, MongoDB uses the *shard key*. Selecting the proper *shard key* has significant implications for performance, and can enable or prevent query isolation and increased write capacity. It is important to consider carefully the field or fields to use as the shard key.

See `http://docs.mongodb.org/manual/core/sharding-introduction` and `http://docs.mongodb.org/manual/core/sharding-shard-key` for more information.

Indexes

Use indexes to improve performance for common queries. Build indexes on fields that appear often in queries and for all operations that return sorted results. MongoDB automatically creates a unique index on the `_id` field.

As you create indexes, consider the following behaviors of indexes:

- Each index requires at least 8KB of data space.
- Adding an index has some negative performance impact for write operations. For collections with high write-to-read ratio, indexes are expensive since each insert must also update any indexes.
- Collections with high read-to-write ratio often benefit from additional indexes. Indexes do not affect un-indexed read operations.
- When active, each index consumes disk space and memory. This usage can be significant and should be tracked for capacity planning, especially for concerns over working set size.

See `http://docs.mongodb.org/manual/applications/indexes` for more information on indexes as well as `http://docs.mongodb.org/manual/tutorial/analyze-query-plan/`. Additionally, the MongoDB `database profiler` may help identify inefficient queries.

Large Number of Collections

In certain situations, you might choose to store related information in several collections rather than in a single collection.

[3] Document-level atomic operations include all operations within a single MongoDB document record: operations that affect multiple subdocuments within that single record are still atomic.

Consider a sample collection `logs` that stores log documents for various environment and applications. The `logs` collection contains documents of the following form:

```
{ log: "dev", ts: ..., info: ... }
{ log: "debug", ts: ..., info: ...}
```

If the total number of documents is low, you may group documents into collection by type. For logs, consider maintaining distinct log collections, such as `logs.dev` and `logs.debug`. The `logs.dev` collection would contain only the documents related to the dev environment.

Generally, having a large number of collections has no significant performance penalty and results in very good performance. Distinct collections are very important for high-throughput batch processing.

When using models that have a large number of collections, consider the following behaviors:

- Each collection has a certain minimum overhead of a few kilobytes.
- Each index, including the index on `_id`, requires at least 8KB of data space.
- For each *database*, a single namespace file (i.e. `<database>.ns`) stores all meta-data for that database, and each index and collection has its own entry in the namespace file. MongoDB places `limits on the size of namespace files`.
- MongoDB has `limits on the number of namespaces`. You may wish to know the current number of namespaces in order to determine how many additional namespaces the database can support. To get the current number of namespaces, run the following in the `mongo` shell:

  ```
  db.system.namespaces.count()
  ```

  The limit on the number of namespaces depend on the `<database>.ns` size. The namespace file defaults to 16 MB.

  To change the size of the *new* namespace file, start the server with the option `--nssize <new size MB>`. For existing databases, after starting up the server with `--nssize`, run the `db.repairDatabase()` command from the `mongo` shell. For impacts and considerations on running `db.repairDatabase()`, see `repairDatabase`.

Data Lifecycle Management

Data modeling decisions should take data lifecycle management into consideration.

The `Time to Live or TTL feature` of collections expires documents after a period of time. Consider using the TTL feature if your application requires some data to persist in the database for a limited period of time.

Additionally, if your application only uses recently inserted documents, consider `http://docs.mongodb.org/manual/core/capped-collections`. Capped collections provide *first-in-first-out* (FIFO) management of inserted documents and efficiently support operations that insert and read documents based on insertion order.

2.3 GridFS

GridFS is a specification for storing and retrieving files that exceed the *BSON*-document *size limit* of 16MB.

Instead of storing a file in a single document, GridFS divides a file into parts, or chunks, [4] and stores each of those chunks as a separate document. By default GridFS limits chunk size to 256k. GridFS uses two collections to store files. One collection stores the file chunks, and the other stores file metadata.

[4] The use of the term *chunks* in the context of GridFS is not related to the use of the term *chunks* in the context of sharding.

When you query a GridFS store for a file, the driver or client will reassemble the chunks as needed. You can perform range queries on files stored through GridFS. You also can access information from arbitrary sections of files, which allows you to "skip" into the middle of a video or audio file.

GridFS is useful not only for storing files that exceed 16MB but also for storing any files for which you want access without having to load the entire file into memory. For more information on the indications of GridFS, see *faq-developers-when-to-use-gridfs*.

Implement GridFS

To store and retrieve files using *GridFS*, use either of the following:

- A MongoDB driver. See the `drivers` documentation for information on using GridFS with your driver.
- The `mongofiles` command-line tool in the `mongo` shell. See `http://docs.mongodb.org/manualreference/program/mongofiles`.

GridFS Collections

GridFS stores files in two collections:

- `chunks` stores the binary chunks. For details, see *The chunks Collection* (page 32).
- `files` stores the file's metadata. For details, see *The files Collection* (page 33).

GridFS places the collections in a common bucket by prefixing each with the bucket name. By default, GridFS uses two collections with names prefixed by `fs` bucket:

- `fs.files`
- `fs.chunks`

You can choose a different bucket name than `fs`, and create multiple buckets in a single database.

Each document in the `chunks` collection represents a distinct chunk of a file as represented in the GridFS store. Each chunk is identified by its unique *ObjectID* stored in its `_id` field.

For descriptions of all fields in the `chunks` and `files` collections, see *GridFS Reference* (page 32).

GridFS Index

GridFS uses a *unique*, *compound* index on the `chunks` collection for the `files_id` and `n` fields. The `files_id` field contains the `_id` of the chunk's "parent" document. The `n` field contains the sequence number of the chunk. GridFS numbers all chunks, starting with 0. For descriptions of the documents and fields in the `chunks` collection, see *GridFS Reference* (page 32).

The GridFS index allows efficient retrieval of chunks using the `files_id` and `n` values, as shown in the following example:

```
cursor = db.fs.chunks.find({files_id: myFileID}).sort({n:1});
```

See the relevant `driver` documentation for the specific behavior of your GridFS application. If your driver does not create this index, issue the following operation using the `mongo` shell:

```
db.fs.chunks.ensureIndex( { files_id: 1, n: 1 }, { unique: true } );
```

Example Interface

The following is an example of the GridFS interface in Java. The example is for demonstration purposes only. For API specifics, see the relevant `driver` documentation.

By default, the interface must support the default GridFS bucket, named `fs`, as in the following:

```
// returns default GridFS bucket (i.e. "fs" collection)
GridFS myFS = new GridFS(myDatabase);

// saves the file to "fs" GridFS bucket
myFS.createFile(new File("/tmp/largething.mpg"));
```

Optionally, interfaces may support other additional GridFS buckets as in the following example:

```
// returns GridFS bucket named "contracts"
GridFS myContracts = new GridFS(myDatabase, "contracts");

// retrieve GridFS object "smithco"
GridFSDBFile file = myContracts.findOne("smithco");

// saves the GridFS file to the file system
file.writeTo(new File("/tmp/smithco.pdf"));
```

3 Data Model Examples and Patterns

The following documents provide overviews of various data modeling patterns and common schema design considerations:

Model Relationships Between Documents **(page 12)** Examples for modeling relationships between documents.

- ***Model One-to-One Relationships with Embedded Documents*** **(page 12)** Presents a data model that uses *embedded documents* (page 5) to describe one-to-one relationships between connected data.
- ***Model One-to-Many Relationships with Embedded Documents*** **(page 13)** Presents a data model that uses *embedded documents* (page 5) to describe one-to-many relationships between connected data.
- ***Model One-to-Many Relationships with Document References*** **(page 14)** Presents a data model that uses *references* (page 6) to describe one-to-many relationships between documents.

Model Tree Structures **(page 16)** Examples for modeling tree structures.

- ***Model Tree Structures with Parent References*** **(page 17)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 6) to "parent" nodes in "child" nodes.
- ***Model Tree Structures with Child References*** **(page 18)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 6) to "child" nodes in "parent" nodes.

See *Model Tree Structures* (page 16) for additional examples of data models for tree structures.

Model Specific Application Contexts **(page 24)** Examples for models for specific application contexts.

- ***Model Data for Atomic Operations*** **(page 24)** Illustrates how embedding fields related to an atomic update within the same document ensures that the fields are in sync
- ***Model Data to Support Keyword Search*** **(page 25)** Describes one method for supporting keyword search by storing keywords in an array in the same document as the text field. Combined with a multi-key index, this pattern can support application's keyword search operations

3.1 Model Relationships Between Documents

Model One-to-One Relationships with Embedded Documents **(page 12)** Presents a data model that uses *embedded documents* (page 5) to describe one-to-one relationships between connected data.

Model One-to-Many Relationships with Embedded Documents **(page 13)** Presents a data model that uses *embedded documents* (page 5) to describe one-to-many relationships between connected data.

Model One-to-Many Relationships with Document References **(page 14)** Presents a data model that uses *references* (page 6) to describe one-to-many relationships between documents.

Model One-to-One Relationships with Embedded Documents

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 5) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *embedded* (page 5) documents to describe relationships between connected data.

Pattern

Consider the following example that maps patron and address relationships. The example illustrates the advantage of embedding over referencing if you need to view one data entity in context of the other. In this one-to-one relationship between `patron` and `address` data, the `address` belongs to the `patron`.

In the normalized data model, the `address` document contains a reference to the `patron` document.

```
{
   _id: "joe",
   name: "Joe Bookreader"
}

{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA",
   zip: 12345
}
```

If the `address` data is frequently retrieved with the `name` information, then with referencing, your application needs to issue multiple queries to resolve the reference. The better data model would be to embed the `address` data in the `patron` data, as in the following document:

```
{
   _id: "joe",
   name: "Joe Bookreader",
   address: {
              street: "123 Fake Street",
              city: "Faketon",
              state: "MA",
              zip: 12345
            }
}
```

With the embedded data model, your application can retrieve the complete patron information with one query.

Model One-to-Many Relationships with Embedded Documents

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 5) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *embedded* (page 5) documents to describe relationships between connected data.

Pattern

Consider the following example that maps patron and multiple address relationships. The example illustrates the advantage of embedding over referencing if you need to view many data entities in context of another. In this one-to-many relationship between `patron` and `address` data, the `patron` has multiple `address` entities.

In the normalized data model, the `address` documents contain a reference to the `patron` document.

```
{
   _id: "joe",
   name: "Joe Bookreader"
}

{
   patron_id: "joe",
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA",
   zip: 12345
}

{
   patron_id: "joe",
   street: "1 Some Other Street",
   city: "Boston",
   state: "MA",
   zip: 12345
}
```

If your application frequently retrieves the `address` data with the `name` information, then your application needs to issue multiple queries to resolve the references. A more optimal schema would be to embed the `address` data entities in the `patron` data, as in the following document:

```
{
   _id: "joe",
   name: "Joe Bookreader",
   addresses: [
                {
                  street: "123 Fake Street",
                  city: "Faketon",
                  state: "MA",
                  zip: 12345
                },
```

```
                {
                  street: "1 Some Other Street",
                  city: "Boston",
                  state: "MA",
                  zip: 12345
                }
              ]
}
```

With the embedded data model, your application can retrieve the complete patron information with one query.

Model One-to-Many Relationships with Document References

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 5) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *references* (page 6) between documents to describe relationships between connected data.

Pattern

Consider the following example that maps publisher and book relationships. The example illustrates the advantage of referencing over embedding to avoid repetition of the publisher information.

Embedding the publisher document inside the book document would lead to **repetition** of the publisher data, as the following documents show:

```
{
   title: "MongoDB: The Definitive Guide",
   author: [ "Kristina Chodorow", "Mike Dirolf" ],
   published_date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
   publisher: {
              name: "O'Reilly Media",
              founded: 1980,
              location: "CA"
            }
}

{
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
   publisher: {
              name: "O'Reilly Media",
              founded: 1980,
              location: "CA"
            }
}
```

To avoid repetition of the publisher data, use *references* and keep the publisher information in a separate collection from the book collection.

When using references, the growth of the relationships determine where to store the reference. If the number of books per publisher is small with limited growth, storing the book reference inside the publisher document may sometimes be useful. Otherwise, if the number of books per publisher is unbounded, this data model would lead to mutable, growing arrays, as in the following example:

```
{
   name: "O'Reilly Media",
   founded: 1980,
   location: "CA",
   books: [12346789, 234567890, ...]
}

{
    _id: 123456789,
    title: "MongoDB: The Definitive Guide",
    author: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate("2010-09-24"),
    pages: 216,
    language: "English"
}

{
   _id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English"
}
```

To avoid mutable, growing arrays, store the publisher reference inside the book document:

```
{
   _id: "oreilly",
   name: "O'Reilly Media",
   founded: 1980,
   location: "CA"
}

{
   _id: 123456789,
   title: "MongoDB: The Definitive Guide",
   author: [ "Kristina Chodorow", "Mike Dirolf" ],
   published_date: ISODate("2010-09-24"),
   pages: 216,
   language: "English",
   publisher_id: "oreilly"
}

{
   _id: 234567890,
   title: "50 Tips and Tricks for MongoDB Developer",
   author: "Kristina Chodorow",
   published_date: ISODate("2011-05-06"),
   pages: 68,
   language: "English",
```

```
   publisher_id: "oreilly"
}
```

3.2 Model Tree Structures

MongoDB allows various ways to use tree data structures to model large hierarchical or nested data relationships.

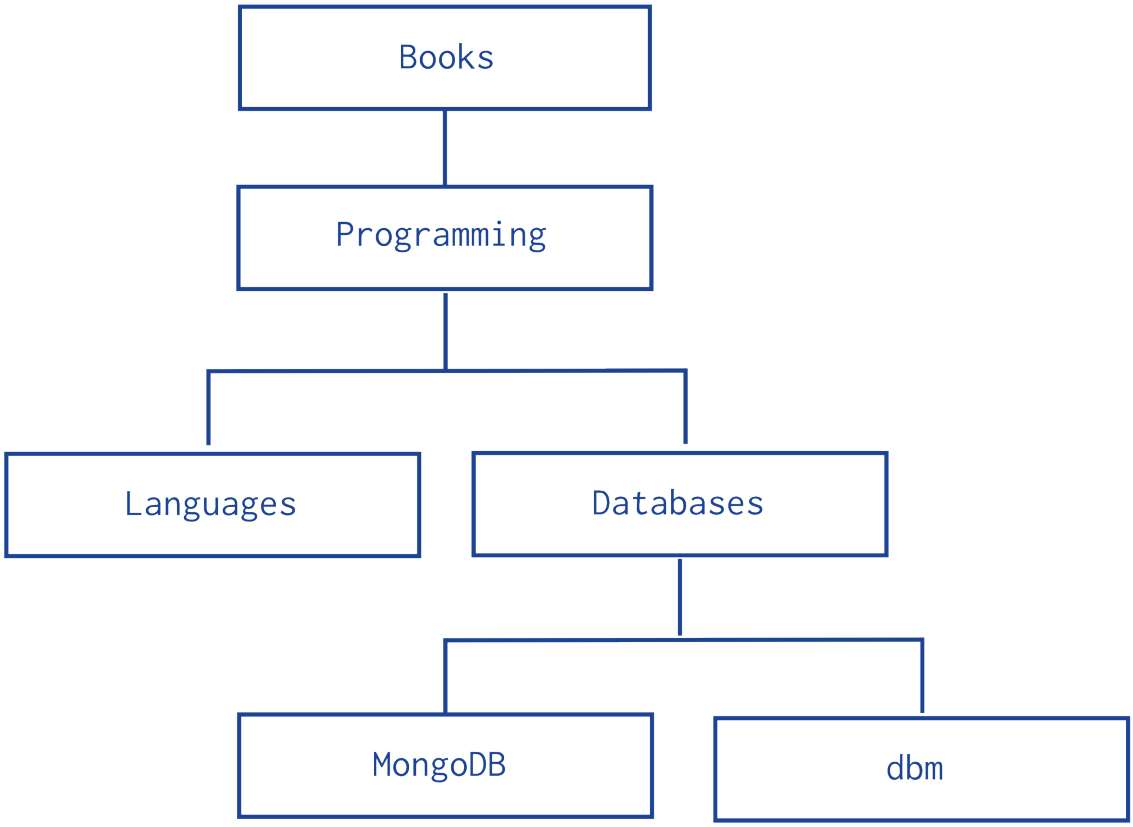

Figure 5: Tree data model for a sample hierarchy of categories.

Model Tree Structures with Parent References **(page 17)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 6) to "parent" nodes in "child" nodes.

Model Tree Structures with Child References **(page 18)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 6) to "child" nodes in "parent" nodes.

Model Tree Structures with an Array of Ancestors **(page 20)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 6) to "parent" nodes and an array that stores all ancestors.

Model Tree Structures with Materialized Paths **(page 21)** Presents a data model that organizes documents in a tree-like structure by storing full relationship paths between documents. In addition to the tree node, each document stores the `_id` of the nodes ancestors or path as a string.

Model Tree Structures with Nested Sets **(page 23)** Presents a data model that organizes documents in a tree-like structure using the *Nested Sets* pattern. This optimizes discovering subtrees at the expense of tree mutability.

Model Tree Structures with Parent References

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 5) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents by storing *references* (page 6) to "parent" nodes in children nodes.

Pattern

The *Parent References* pattern stores each tree node in a document; in addition to the tree node, the document stores the id of the node's parent.

Consider the following hierarchy of categories:

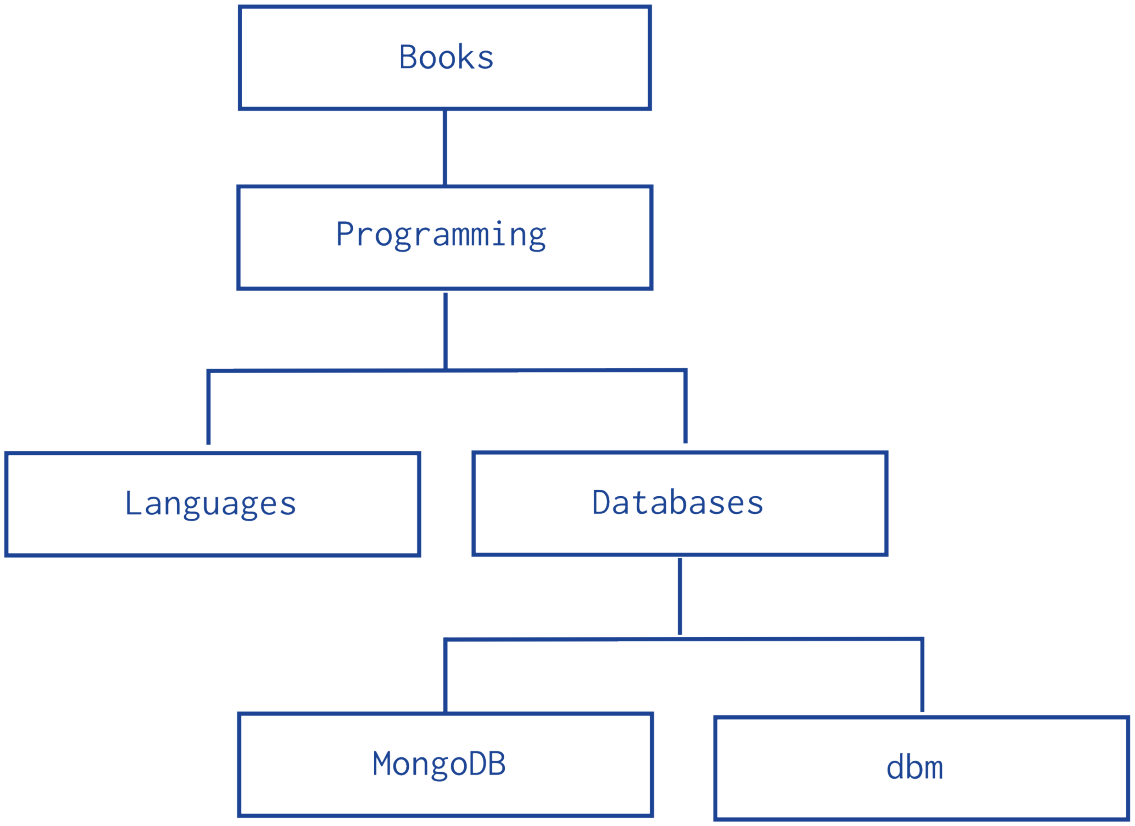

Figure 6: Tree data model for a sample hierarchy of categories.

The following example models the tree using *Parent References*, storing the reference to the parent category in the field `parent`:

```
db.categories.insert( { _id: "MongoDB", parent: "Databases" } )
db.categories.insert( { _id: "dbm", parent: "Databases" } )
db.categories.insert( { _id: "Databases", parent: "Programming" } )
```

```
db.categories.insert( { _id: "Languages", parent: "Programming" } )
db.categories.insert( { _id: "Programming", parent: "Books" } )
db.categories.insert( { _id: "Books", parent: null } )
```

- The query to retrieve the parent of a node is fast and straightforward:

  ```
  db.categories.findOne( { _id: "MongoDB" } ).parent
  ```

- You can create an index on the field `parent` to enable fast search by the parent node:

  ```
  db.categories.ensureIndex( { parent: 1 } )
  ```

- You can query by the `parent` field to find its immediate children nodes:

  ```
  db.categories.find( { parent: "Databases" } )
  ```

The *Parent Links* pattern provides a simple solution to tree storage but requires multiple queries to retrieve subtrees.

Model Tree Structures with Child References

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 5) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents by storing *references* (page 6) in the parent-nodes to children nodes.

Pattern

The *Child References* pattern stores each tree node in a document; in addition to the tree node, document stores in an array the id(s) of the node's children.

Consider the following hierarchy of categories:

The following example models the tree using *Child References*, storing the reference to the node's children in the field `children`:

```
db.categories.insert( { _id: "MongoDB", children: [] } )
db.categories.insert( { _id: "dbm", children: [] } )
db.categories.insert( { _id: "Databases", children: [ "MongoDB", "dbm" ] } )
db.categories.insert( { _id: "Languages", children: [] } )
db.categories.insert( { _id: "Programming", children: [ "Databases", "Languages" ] } )
db.categories.insert( { _id: "Books", children: [ "Programming" ] } )
```

- The query to retrieve the immediate children of a node is fast and straightforward:

  ```
  db.categories.findOne( { _id: "Databases" } ).children
  ```

- You can create an index on the field `children` to enable fast search by the child nodes:

  ```
  db.categories.ensureIndex( { children: 1 } )
  ```

- You can query for a node in the `children` field to find its parent node as well as its siblings:

  ```
  db.categories.find( { children: "MongoDB" } )
  ```

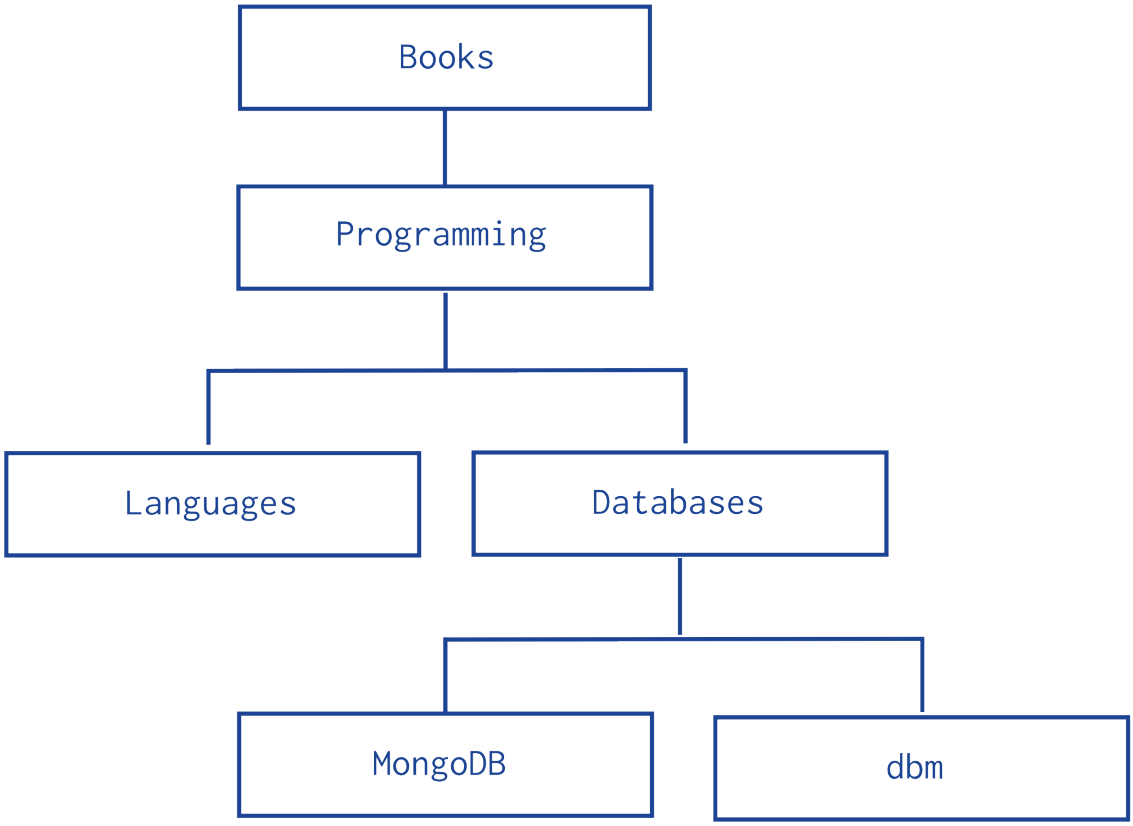

Figure 7: Tree data model for a sample hierarchy of categories.

The *Child References* pattern provides a suitable solution to tree storage as long as no operations on subtrees are necessary. This pattern may also provide a suitable solution for storing graphs where a node may have multiple parents.

Model Tree Structures with an Array of Ancestors

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 5) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents using *references* (page 6) to parent nodes and an array that stores all ancestors.

Pattern

The *Array of Ancestors* pattern stores each tree node in a document; in addition to the tree node, document stores in an array the id(s) of the node's ancestors or path.

Consider the following hierarchy of categories:

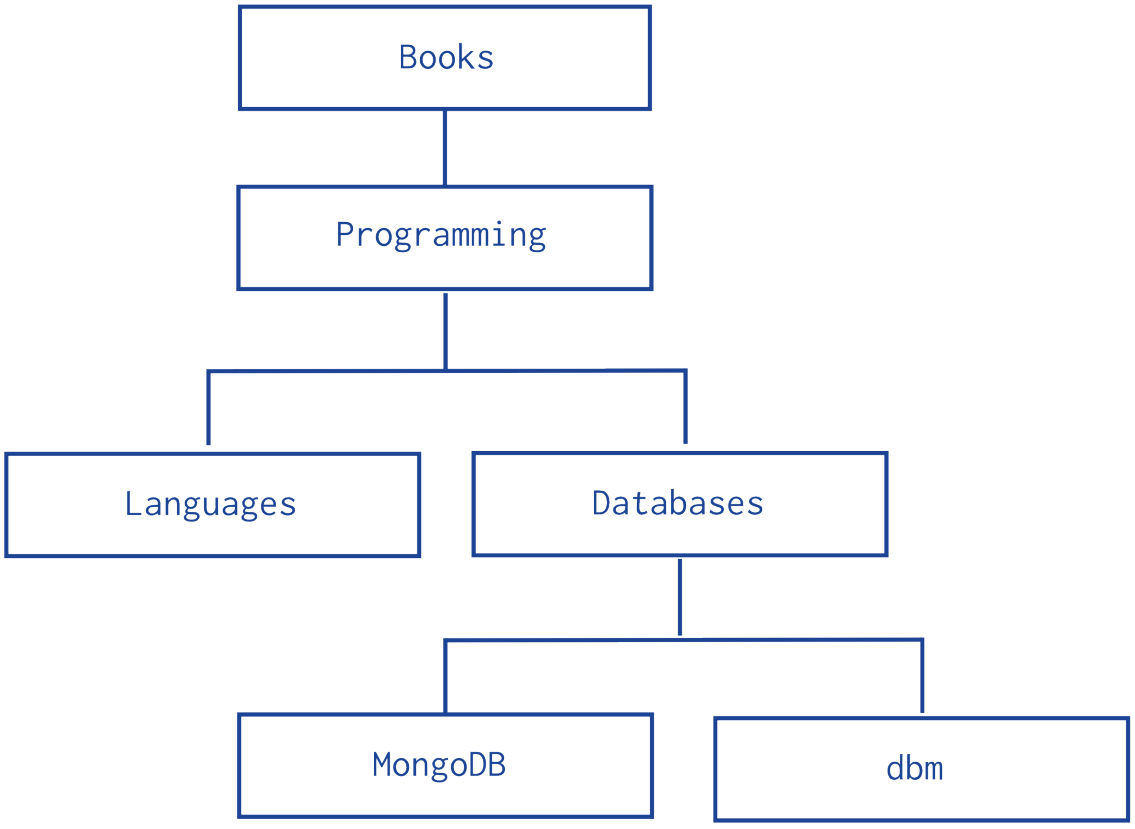

Figure 8: Tree data model for a sample hierarchy of categories.

The following example models the tree using *Array of Ancestors*. In addition to the `ancestors` field, these documents also store the reference to the immediate parent category in the `parent` field:

```
db.categories.insert( { _id: "MongoDB", ancestors: [ "Books", "Programming", "Databases" ], parent: "
db.categories.insert( { _id: "dbm", ancestors: [ "Books", "Programming", "Databases" ], parent: "Data
db.categories.insert( { _id: "Databases", ancestors: [ "Books", "Programming" ], parent: "Programming
db.categories.insert( { _id: "Languages", ancestors: [ "Books", "Programming" ], parent: "Programming
db.categories.insert( { _id: "Programming", ancestors: [ "Books" ], parent: "Books" } )
db.categories.insert( { _id: "Books", ancestors: [ ], parent: null } )
```

- The query to retrieve the ancestors or path of a node is fast and straightforward:

  ```
  db.categories.findOne( { _id: "MongoDB" } ).ancestors
  ```

- You can create an index on the field `ancestors` to enable fast search by the ancestors nodes:

  ```
  db.categories.ensureIndex( { ancestors: 1 } )
  ```

- You can query by the field `ancestors` to find all its descendants:

  ```
  db.categories.find( { ancestors: "Programming" } )
  ```

The *Array of Ancestors* pattern provides a fast and efficient solution to find the descendants and the ancestors of a node by creating an index on the elements of the ancestors field. This makes *Array of Ancestors* a good choice for working with subtrees.

The *Array of Ancestors* pattern is slightly slower than the *Materialized Paths* (page 21) pattern but is more straightforward to use.

Model Tree Structures with Materialized Paths

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 5) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents by storing full relationship paths between documents.

Pattern

The *Materialized Paths* pattern stores each tree node in a document; in addition to the tree node, document stores as a string the id(s) of the node's ancestors or path. Although the *Materialized Paths* pattern requires additional steps of working with strings and regular expressions, the pattern also provides more flexibility in working with the path, such as finding nodes by partial paths.

Consider the following hierarchy of categories:

The following example models the tree using *Materialized Paths*, storing the path in the field `path`; the path string uses the comma `,` as a delimiter:

```
db.categories.insert( { _id: "Books", path: null } )
db.categories.insert( { _id: "Programming", path: ",Books," } )
db.categories.insert( { _id: "Databases", path: ",Books,Programming," } )
db.categories.insert( { _id: "Languages", path: ",Books,Programming," } )
db.categories.insert( { _id: "MongoDB", path: ",Books,Programming,Databases," } )
db.categories.insert( { _id: "dbm", path: ",Books,Programming,Databases," } )
```

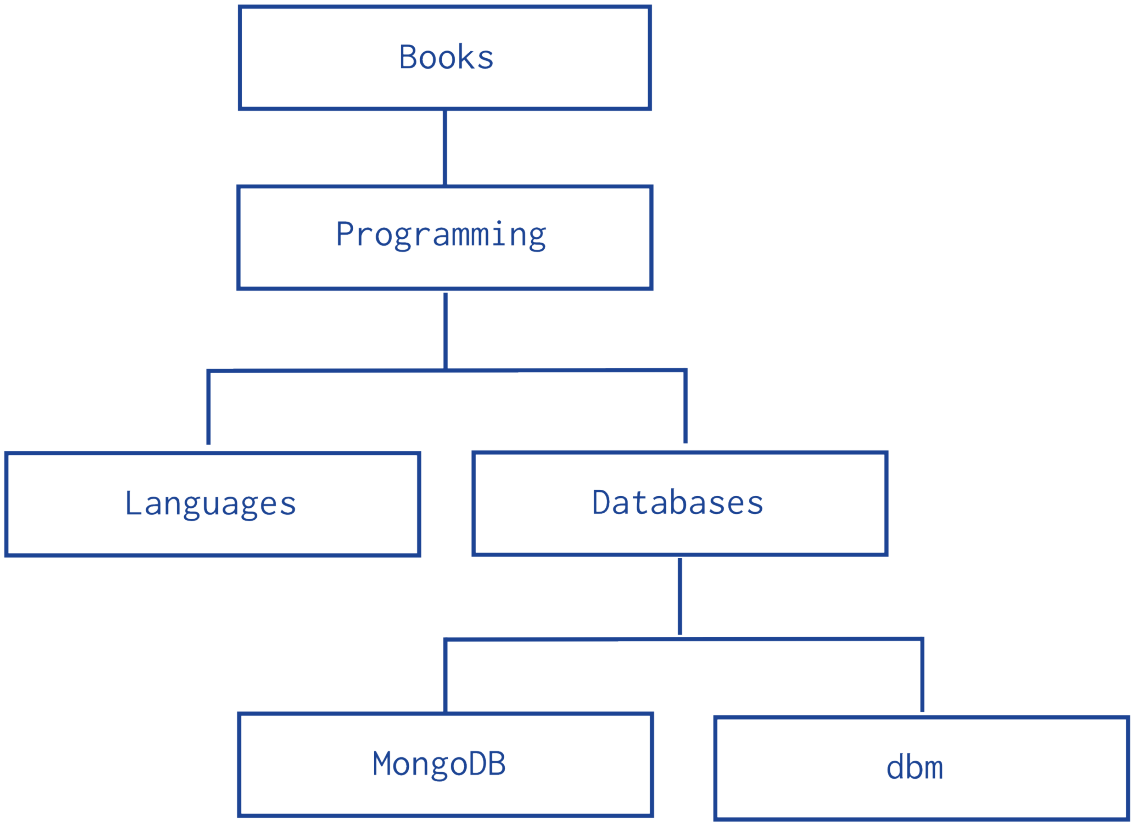

Figure 9: Tree data model for a sample hierarchy of categories.

- You can query to retrieve the whole tree, sorting by the field `path`:

  ```
  db.categories.find().sort( { path: 1 } )
  ```

- You can use regular expressions on the `path` field to find the descendants of `Programming`:

  ```
  db.categories.find( { path: /,Programming,/ } )
  ```

- You can also retrieve the descendants of `Books` where the `Books` is also at the topmost level of the hierarchy:

  ```
  db.categories.find( { path: /^,Books,/ } )
  ```

- To create an index on the field `path` use the following invocation:

  ```
  db.categories.ensureIndex( { path: 1 } )
  ```

  This index may improve performance depending on the query:

  - For queries of the `Books` sub-tree (e.g. `http://docs.mongodb.org/manual^,Books,/`) an index on the `path` field improves the query performance significantly.
  - For queries of the `Programming` sub-tree (e.g. `http://docs.mongodb.org/manual,Programming,/`), or similar queries of sub-tress, where the node might be in the middle of the indexed string, the query must inspect the entire index.

    For these queries an index *may* provide some performance improvement *if* the index is significantly smaller than the entire collection.

Model Tree Structures with Nested Sets

Overview

Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 5) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree like structure that optimizes discovering subtrees at the expense of tree mutability.

Pattern

The *Nested Sets* pattern identifies each node in the tree as stops in a round-trip traversal of the tree. The application visits each node in the tree twice; first during the initial trip, and second during the return trip. The *Nested Sets* pattern stores each tree node in a document; in addition to the tree node, document stores the id of node's parent, the node's initial stop in the `left` field, and its return stop in the `right` field.

Consider the following hierarchy of categories:

The following example models the tree using *Nested Sets*:

```
db.categories.insert( { _id: "Books", parent: 0, left: 1, right: 12 } )
db.categories.insert( { _id: "Programming", parent: "Books", left: 2, right: 11 } )
db.categories.insert( { _id: "Languages", parent: "Programming", left: 3, right: 4 } )
db.categories.insert( { _id: "Databases", parent: "Programming", left: 5, right: 10 } )
db.categories.insert( { _id: "MongoDB", parent: "Databases", left: 6, right: 7 } )
db.categories.insert( { _id: "dbm", parent: "Databases", left: 8, right: 9 } )
```

You can query to retrieve the descendants of a node:

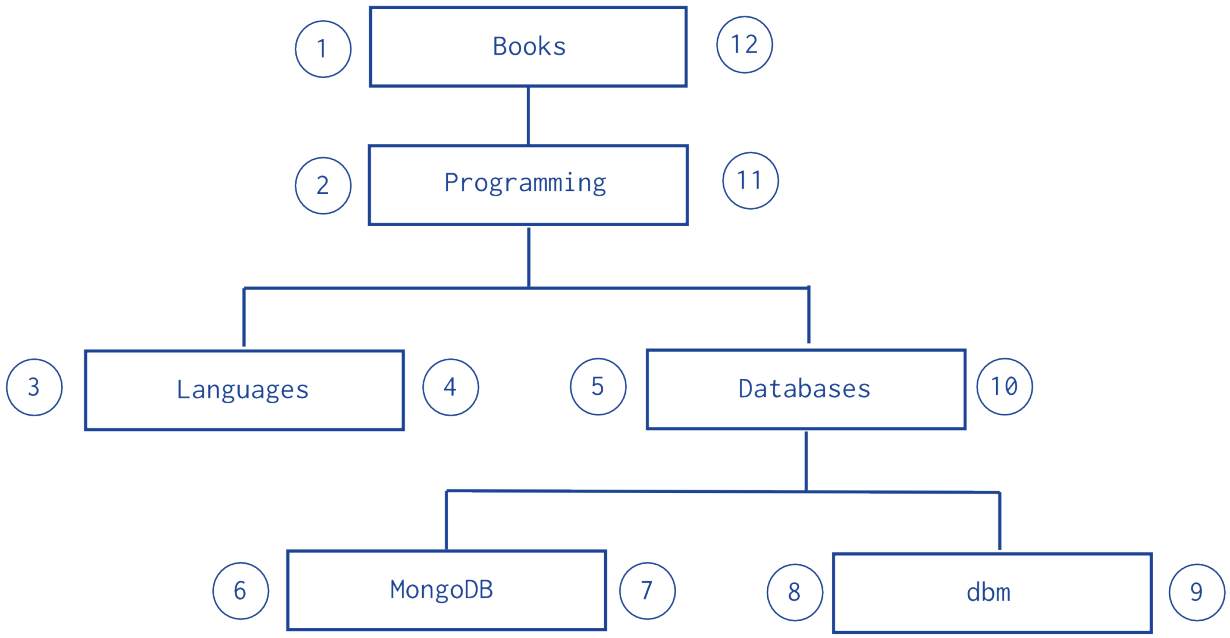

Figure 10: Example of a hierarchical data. The numbers identify the stops at nodes during a roundtrip traversal of a tree.

```
var databaseCategory = db.categories.findOne( { _id: "Databases" } );
db.categories.find( { left: { $gt: databaseCategory.left }, right: { $lt: databaseCategory.right } } )
```

The *Nested Sets* pattern provides a fast and efficient solution for finding subtrees but is inefficient for modifying the tree structure. As such, this pattern is best for static trees that do not change.

3.3 Model Specific Application Contexts

Model Data for Atomic Operations **(page 24)** Illustrates how embedding fields related to an atomic update within the same document ensures that the fields are in sync

Model Data to Support Keyword Search **(page 25)** Describes one method for supporting keyword search by storing keywords in an array in the same document as the text field. Combined with a multi-key index, this pattern can support application's keyword search operations

Model Data for Atomic Operations

Pattern

Consider the following example that keeps a library book and its checkout information. The example illustrates how embedding fields related to an atomic update within the same document ensures that the fields are in sync.

Consider the following `book` document that stores the number of available copies for checkout and the current checkout information:

```
book = {
          _id: 123456789,
          title: "MongoDB: The Definitive Guide",
```

```
          author: [ "Kristina Chodorow", "Mike Dirolf" ],
          published_date: ISODate("2010-09-24"),
          pages: 216,
          language: "English",
          publisher_id: "oreilly",
          available: 3,
          checkout: [ { by: "joe", date: ISODate("2012-10-15") } ]
        }
```

You can use the `db.collection.findAndModify()` method to atomically determine if a book is available for checkout and update with the new checkout information. Embedding the `available` field and the `checkout` field within the same document ensures that the updates to these fields are in sync:

```
db.books.findAndModify ( {
   query: {
             _id: 123456789,
             available: { $gt: 0 }
           },
   update: {
              $inc: { available: -1 },
              $push: { checkout: { by: "abc", date: new Date() } }
            }
} )
```

Model Data to Support Keyword Search

Note: Keyword search is *not* the same as text search or full text search, and does not provide stemming or other text-processing features. See the *Limitations of Keyword Indexes* (page 26) section for more information.

In 2.4, MongoDB provides a text search feature. See `http://docs.mongodb.org/manual/core/index-text` for more information.

If your application needs to perform queries on the content of a field that holds text you can perform exact matches on the text or use `$regex` to use regular expression pattern matches. However, for many operations on text, these methods do not satisfy application requirements.

This pattern describes one method for supporting keyword search using MongoDB to support application search functionality, that uses keywords stored in an array in the same document as the text field. Combined with a *multi-key index*, this pattern can support application's keyword search operations.

Pattern

To add structures to your document to support keyword-based queries, create an array field in your documents and add the keywords as strings in the array. You can then create a *multi-key index* on the array and create queries that select values from the array.

Example

Given a collection of library volumes that you want to provide topic-based search. For each volume, you add the array `topics`, and you add as many keywords as needed for a given volume.

For the `Moby-Dick` volume you might have the following document:

```
{ title : "Moby-Dick" ,
  author : "Herman Melville" ,
  published : 1851 ,
```

```
  ISBN : 0451526996 ,
  topics : [ "whaling" , "allegory" , "revenge" , "American" ,
    "novel" , "nautical" , "voyage" , "Cape Cod" ]
}
```

You then create a multi-key index on the `topics` array:

```
db.volumes.ensureIndex( { topics: 1 } )
```

The multi-key index creates separate index entries for each keyword in the `topics` array. For example the index contains one entry for `whaling` and another for `allegory`.

You then query based on the keywords. For example:

```
db.volumes.findOne( { topics : "voyage" }, { title: 1 } )
```

Note: An array with a large number of elements, such as one with several hundreds or thousands of keywords will incur greater indexing costs on insertion.

Limitations of Keyword Indexes

MongoDB can support keyword searches using specific data models and *multi-key indexes*; however, these keyword indexes are not sufficient or comparable to full-text products in the following respects:

- *Stemming*. Keyword queries in MongoDB can not parse keywords for root or related words.
- *Synonyms*. Keyword-based search features must provide support for synonym or related queries in the application layer.
- *Ranking*. The keyword look ups described in this document do not provide a way to weight results.
- *Asynchronous Indexing*. MongoDB builds indexes synchronously, which means that the indexes used for keyword indexes are always current and can operate in real-time. However, asynchronous bulk indexes may be more efficient for some kinds of content and workloads.

4 Data Model Reference

Documents **(page 27)** MongoDB stores all data in documents, which are JSON-style data structures composed of field-and-value pairs.

Database References **(page 29)** Discusses manual references and DBRefs, which MongoDB can use to represent relationships between documents.

GridFS Reference **(page 32)** Convention for storing large files in a MongoDB Database.

ObjectId **(page 34)** A 12-byte BSON type that MongoDB uses as the default value for its documents' `_id` field if the `_id` field is not specified.

BSON Types **(page 36)** Outlines the unique *BSON* types used by MongoDB. See BSONspec.org[5] for the complete BSON specification.

[5] http://bsonspec.org/

4.1 Documents

MongoDB stores all data in documents, which are JSON-style data structures composed of field-and-value pairs:

```
{ "item": "pencil", "qty": 500, "type": "no.2" }
```

Most user-accessible data structures in MongoDB are documents, including:

- All database records.
- `Query selectors`, which define what records to select for read, update, and delete operations.
- `Update definitions`, which define what fields to modify during an update.
- `Index specifications`, which define what fields to index.
- Data output by MongoDB for reporting and configuration, such as the output of the `serverStatus` and the *replica set configuration document*.

Document Format

MongoDB stores documents on disk in the *BSON* serialization format. BSON is a binary representation of *JSON* documents, though contains more data types than does JSON. For the BSON spec, see bsonspec.org[6]. See also *BSON Types* (page 36).

The `mongo` JavaScript shell and the `MongoDB language drivers` translate between BSON and the language-specific document representation.

Document Structure

MongoDB documents are composed of field-and-value pairs and have the following structure:

```
{
   field1: value1,
   field2: value2,
   field3: value3,
   ...
   fieldN: valueN
}
```

The value of a field can be any of the BSON *data types* (page 36), including other documents, arrays, and arrays of documents. The following document contains values of varying types:

```
var mydoc = {
               _id: ObjectId("5099803df3f4948bd2f98391"),
               name: { first: "Alan", last: "Turing" },
               birth: new Date('Jun 23, 1912'),
               death: new Date('Jun 07, 1954'),
               contribs: [ "Turing machine", "Turing test", "Turingery" ],
               views : NumberLong(1250000)
             }
```

The above fields have the following data types:

- `_id` holds an *ObjectId*.
- `name` holds a *subdocument* that contains the fields `first` and `last`.

[6] http://bsonspec.org/

- `birth` and `death` hold values of the *Date* type.
- `contribs` holds an *array of strings*.
- `views` holds a value of the *NumberLong* type.

Field Names

Field names are strings. Field names cannot contain null characters, dots (`.`) or dollar signs (`$`). See *faq-dollar-sign-escaping* for an alternate approach.

BSON documents may have more than one field with the same name. Most `MongoDB interfaces`, however, represent MongoDB with a structure (e.g. a hash table) that does not support duplicate field names. If you need to manipulate documents that have more than one field with the same name, see the `driver documentation` for your driver.

Some documents created by internal MongoDB processes may have duplicate fields, but *no* MongoDB process will *ever* add duplicate fields to an existing user document.

Document Limitations

Documents have the following attributes:

- The maximum BSON document size is 16 megabytes.

  The maximum document size helps ensure that a single document cannot use excessive amount of RAM or, during transmission, excessive amount of bandwidth. To store documents larger than the maximum size, MongoDB provides the GridFS API. See `mongofiles` and the documentation for your `driver` for more information about GridFS.

- *Documents* (page 27) have the following restrictions on field names:
  - The field name `_id` is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array.
  - The field names **cannot** start with the `$` character.
  - The field names **cannot** contain the `.` character.
- MongoDB does not make guarantees regarding the order of fields in a BSON document. Drivers and MongoDB will reorder the fields of a documents upon insertion and following updates.

  Most programming languages represent BSON documents with some form of *mapping type*. Comparisons between mapping type objects typically, depend on order. As a result, the only way to ensure that two documents have the same set of field and value pairs is to compare each field and value individually.

The `_id` Field

The `_id` field has the following behavior and constraints:

- In documents, the `_id` field is always indexed for regular collections.
- The `_id` field may contain values of any *BSON data type* (page 36), other than an array.

  Warning: To ensure functioning replication, do not store values that are of the BSON regular expression type in the `_id` field.

The following are common options for storing values for `_id`:

- Use an *ObjectId* (page 34).
- Use a natural unique identifier, if available. This saves space and avoids an additional index.
- Generate an auto-incrementing number. See `http://docs.mongodb.org/manualtutorial/create-an-auto-inc`
- Generate a UUID in your application code. For a more efficient storage of the UUID values in the collection and in the `_id` index, store the UUID as a value of the BSON `BinData` type.

  Index keys that are of the `BinData` type are more efficiently stored in the index if:

  - the binary subtype value is in the range of 0-7 or 128-135, and
  - the length of the byte array is: 0, 1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 20, 24, or 32.
- Use your driver's BSON UUID facility to generate UUIDs. Be aware that driver implementations may implement UUID serialization and deserialization logic differently, which may not be fully compatible with other drivers. See your driver documentation[7] for information concerning UUID interoperability.

Note: Most MongoDB driver clients will include the `_id` field and generate an `ObjectId` before sending the insert operation to MongoDB; however, if the client sends a document without an `_id` field, the `mongod` will add the `_id` field and generate the `ObjectId`.

Dot Notation

MongoDB uses the *dot notation* to access the elements of an array and to access the fields of a subdocument.

To access an element of an array by the zero-based index position, concatenate the array name with the dot (`.`) and zero-based index position, and enclose in quotes:

```
'<array>.<index>'
```

To access a field of a subdocument with *dot-notation*, concatenate the subdocument name with the dot (`.`) and the field name, and enclose in quotes:

```
'<subdocument>.<field>'
```

See also:

- *read-operations-subdocuments* for dot notation examples with subdocuments.
- *read-operations-arrays* for dot notation examples with arrays.

4.2 Database References

MongoDB does not support joins. In MongoDB some data is *denormalized*, or stored with related data in *documents* to remove the need for joins. However, in some cases it makes sense to store related information in separate documents, typically in different collections or databases.

MongoDB applications use one of two methods for relating documents:

1. *Manual references* (page 30) where you save the `_id` field of one document in another document as a reference. Then your application can run a second query to return the embedded data. These references are simple and sufficient for most use cases.
2. *DBRefs* (page 31) are references from one document to another using the value of the first document's `_id` field collection, and optional database name. To resolve DBRefs, your application must perform additional queries

[7] http://api.mongodb.org/

to return the referenced documents. Many `drivers` have helper methods that form the query for the DBRef automatically. The drivers [8] do not *automatically* resolve DBRefs into documents.

Use a DBRef when you need to embed documents from multiple collections in documents from one collection. DBRefs also provide a common format and type to represent these relationships among documents. The DBRef format provides common semantics for representing links between documents if your database must interact with multiple frameworks and tools.

Unless you have a compelling reason for using a DBRef, use manual references.

Manual References

Background

Manual references refers to the practice of including one *document's* `_id` field in another document. The application can then issue a second query to resolve the referenced fields as needed.

Process

Consider the following operation to insert two documents, using the `_id` field of the first document as a reference in the second document:

```
original_id = ObjectId()

db.places.insert({
    "_id": original_id,
    "name": "Broadway Center",
    "url": "bc.example.net"
})

db.people.insert({
    "name": "Erin",
    "places_id": original_id,
    "url":  "bc.example.net/Erin"
})
```

Then, when a query returns the document from the `people` collection you can, if needed, make a second query for the document referenced by the `places_id` field in the `places` collection.

Use

For nearly every case where you want to store a relationship between two documents, use *manual references* (page 30). The references are simple to create and your application can resolve references as needed.

The only limitation of manual linking is that these references do not convey the database and collection name. If you have documents in a single collection that relate to documents in more than one collection, you may need to consider using *DBRefs* (page 31).

[8] Some community supported drivers may have alternate behavior and may resolve a DBRef into a document automatically.

DBRefs

Background

DBRefs are a convention for representing a *document*, rather than a specific reference type. They include the name of the collection, and in some cases the database, in addition to the value from the `_id` field.

Format

DBRefs have the following fields:

`$ref`
: The `$ref` field holds the name of the collection where the referenced document resides.

`$id`
: The `$id` field contains the value of the `_id` field in the referenced document.

`$db`
: *Optional.*

  Contains the name of the database where the referenced document resides.

  Only some drivers support `$db` references.

Example

DBRef document would resemble the following:

```
{ "$ref" : <value>, "$id" : <value>, "$db" : <value> }
```

Consider a document from a collection that stored a DBRef in a `creator` field:

```
{
  "_id" : ObjectId("5126bbf64aed4daf9e2ab771"),
  // .. application fields
  "creator" : {
                  "$ref" : "creators",
                  "$id" : ObjectId("5126bc054aed4daf9e2ab772"),
                  "$db" : "users"
                }
}
```

The DBRef in this example, points to a document in the `creators` collection of the `users` database that has `ObjectId("5126bc054aed4daf9e2ab772")` in its `_id` field.

Note: The order of fields in the DBRef matters, and you must use the above sequence when using a DBRef.

Support

C++ The C++ driver contains no support for DBRefs. You can transverse references manually.

C# The C# driver provides access to DBRef objects with the MongoDBRef Class[9] and supplies the FetchDBRef Method[10] for accessing these objects.

[9] http://api.mongodb.org/csharp/current/html/46c356d3-ed06-a6f8-42fa-e0909ab64ce2.htm
[10] http://api.mongodb.org/csharp/current/html/1b0b8f48-ba98-1367-0a7d-6e01c8df436f.htm

Java The DBRef[11] class provides supports for DBRefs from Java.

JavaScript The `mongo` shell's `JavaScript` interface provides a DBRef.

Perl The Perl driver contains no support for DBRefs. You can transverse references manually or use the MongoDBx::AutoDeref[12] CPAN module.

PHP The PHP driver does support DBRefs, including the optional `$db` reference, through The MongoDBRef class[13].

Python The Python driver provides the DBRef class[14], and the dereference method[15] for interacting with DBRefs.

Ruby The Ruby Driver supports DBRefs using the DBRef class[16] and the deference method[17].

Use

In most cases you should use the *manual reference* (page 30) method for connecting two or more related documents. However, if you need to reference documents from multiple collections, consider a DBRef.

4.3 GridFS Reference

GridFS stores files in two collections:

- `chunks` stores the binary chunks. For details, see *The chunks Collection* (page 32).
- `files` stores the file's metadata. For details, see *The files Collection* (page 33).

GridFS places the collections in a common bucket by prefixing each with the bucket name. By default, GridFS uses two collections with names prefixed by `fs` bucket:

- `fs.files`
- `fs.chunks`

You can choose a different bucket name than `fs`, and create multiple buckets in a single database.

See also:

GridFS (page 9) for more information about GridFS.

The `chunks` Collection

Each document in the `chunks` collection represents a distinct chunk of a file as represented in the *GridFS* store. The following is a prototype document from the `chunks` collection.:

```
{
  "_id" : <ObjectId>,
  "files_id" : <ObjectID>,
  "n" : <num>,
  "data" : <binary>
}
```

A document from the `chunks` collection contains the following fields:

[11] http://api.mongodb.org/java/current/com/mongodb/DBRef.html
[12] http://search.cpan.org/dist/MongoDBx-AutoDeref/
[13] http://www.php.net/manual/en/class.mongodbref.php/
[14] http://api.mongodb.org/python/current/api/bson/dbref.html
[15] http://api.mongodb.org//python/current/api/pymongo/database.html#pymongo.database.Database.dereference
[16] http://api.mongodb.org//ruby/current/BSON/DBRef.html
[17] http://api.mongodb.org//ruby/current/Mongo/DB.html#dereference

`chunks.`**`_id`**
: The unique *ObjectID* of the chunk.

`chunks.`**`files_id`**
: The `_id` of the "parent" document, as specified in the `files` collection.

`chunks.`**`n`**
: The sequence number of the chunk. GridFS numbers all chunks, starting with 0.

`chunks.`**`data`**
: The chunk's payload as a *BSON* binary type.

The `chunks` collection uses a *compound index* on `files_id` and `n`, as described in *GridFS Index* (page 10).

The `files` Collection

Each document in the `files` collection represents a file in the *GridFS* store. Consider the following prototype of a document in the `files` collection:

```
{
  "_id" : <ObjectID>,
  "length" : <num>,
  "chunkSize" : <num>
  "uploadDate" : <timestamp>
  "md5" : <hash>

  "filename" : <string>,
  "contentType" : <string>,
  "aliases" : <string array>,
  "metadata" : <dataObject>,
}
```

Documents in the `files` collection contain some or all of the following fields. Applications may create additional arbitrary fields:

`files.`**`_id`**
: The unique ID for this document. The `_id` is of the data type you chose for the original document. The default type for MongoDB documents is *BSON ObjectID*.

`files.`**`length`**
: The size of the document in bytes.

`files.`**`chunkSize`**
: The size of each chunk. GridFS divides the document into chunks of the size specified here. The default size is 256 kilobytes.

`files.`**`uploadDate`**
: The date the document was first stored by GridFS. This value has the `Date` type.

`files.`**`md5`**
: An MD5 hash returned from the filemd5 API. This value has the `String` type.

`files.`**`filename`**
: Optional. A human-readable name for the document.

`files.`**`contentType`**
: Optional. A valid MIME type for the document.

`files.`**`aliases`**
: Optional. An array of alias strings.

`files.metadata`
: Optional. Any additional information you want to store.

4.4 ObjectId

Overview

ObjectId is a 12-byte *BSON* type, constructed using:

- a 4-byte value representing the seconds since the Unix epoch,
- a 3-byte machine identifier,
- a 2-byte process id, and
- a 3-byte counter, starting with a random value.

In MongoDB, documents stored in a collection require a unique *_id* field that acts as a *primary key*. Because ObjectIds are small, most likely unique, and fast to generate, MongoDB uses ObjectIds as the default value for the `_id` field if the `_id` field is not specified. MongoDB clients should add an `_id` field with a unique ObjectId. However, if a client does not add an `_id` field, `mongod` will add an `_id` field that holds an ObjectId.

Using ObjectIds for the `_id` field provides the following additional benefits:

- in the `mongo` shell, you can access the creation time of the `ObjectId`, using the `getTimestamp()` method.
- sorting on an `_id` field that stores `ObjectId` values is roughly equivalent to sorting by creation time.

Important: The relationship between the order of `ObjectId` values and generation time is not strict within a single second. If multiple systems, or multiple processes or threads on a single system generate values, within a single second; `ObjectId` values do not represent a strict insertion order. Clock skew between clients can also result in non-strict ordering even for values, because client drivers generate `ObjectId` values, *not* the `mongod` process.

Also consider the *Documents* (page 27) section for related information on MongoDB's document orientation.

ObjectId()

The `mongo` shell provides the `ObjectId()` wrapper class to generate a new ObjectId, and to provide the following helper attribute and methods:

- `str`

  The hexadecimal string value of the `ObjectId()` object.

- `getTimestamp()`

  Returns the timestamp portion of the `ObjectId()` object as a Date.

- `toString()`

  Returns the string representation of the `ObjectId()` object. The returned string literal has the format "`ObjectId(...)`".

  Changed in version 2.2: In previous versions `toString()` returns the value of the ObjectId as a hexadecimal string.

- `valueOf()`

Returns the value of the `ObjectId()` object as a hexadecimal string. The returned string is the `str` attribute.

Changed in version 2.2: In previous versions `valueOf()` returns the `ObjectId()` object.

Examples

Consider the following uses `ObjectId()` class in the `mongo` shell:

Generate a new ObjectId

To generate a new ObjectId, use the `ObjectId()` constructor with no argument:

```
x = ObjectId()
```

In this example, the value of `x` would be:

```
ObjectId("507f1f77bcf86cd799439011")
```

To generate a new ObjectId using the `ObjectId()` constructor with a unique hexadecimal string:

```
y = ObjectId("507f191e810c19729de860ea")
```

In this example, the value of `y` would be:

```
ObjectId("507f191e810c19729de860ea")
```

- To return the timestamp of an `ObjectId()` object, use the `getTimestamp()` method as follows:

Convert an ObjectId into a Timestamp

To return the timestamp of an `ObjectId()` object, use the `getTimestamp()` method as follows:

```
ObjectId("507f191e810c19729de860ea").getTimestamp()
```

This operation will return the following Date object:

```
ISODate("2012-10-17T20:46:22Z")
```

Convert ObjectIds into Strings

Access the `str` attribute of an `ObjectId()` object, as follows:

```
ObjectId("507f191e810c19729de860ea").str
```

This operation will return the following hexadecimal string:

```
507f191e810c19729de860ea
```

To return the value of an `ObjectId()` object as a hexadecimal string, use the `valueOf()` method as follows:

```
ObjectId("507f191e810c19729de860ea").valueOf()
```

This operation returns the following output:

```
507f191e810c19729de860ea
```

To return the string representation of an `ObjectId()` object, use the `toString()` method as follows:

```
ObjectId("507f191e810c19729de860ea").toString()
```

This operation will return the following output:

```
ObjectId("507f191e810c19729de860ea")
```

4.5 BSON Types

BSON is a binary serialization format used to store documents and make remote procedure calls in MongoDB. The BSON specification is located at bsonspec.org[18].

BSON supports the following data types as values in documents. Each data type has a corresponding number that can be used with the `$type` operator to query documents by BSON type.

Type	Number
Double	1
String	2
Object	3
Array	4
Binary data	5
Object id	7
Boolean	8
Date	9
Null	10
Regular Expression	11
JavaScript	13
Symbol	14
JavaScript (with scope)	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

When comparing values of different *BSON* types, MongoDB uses the following comparison order, from lowest to highest:

1. MinKey (internal type)
2. Null
3. Numbers (ints, longs, doubles)
4. Symbol, String
5. Object

[18] http://bsonspec.org/

6. Array
7. BinData
8. ObjectID
9. Boolean
10. Date, Timestamp
11. Regular Expression
12. MaxKey (internal type)

Note: MongoDB treats some types as equivalent for comparison purposes. For instance, numeric types undergo conversion before comparison.

To determine a field's type, see *check-types-in-shell*.

If you convert BSON to JSON, see *bson-json-type-conversion-fidelity* for more information.

The next sections describe special considerations for particular BSON types.

ObjectId

ObjectIds are: small, likely unique, fast to generate, and ordered. These values consists of 12-bytes, where the first four bytes are a timestamp that reflect the ObjectId's creation. Refer to the *ObjectId* (page 34) documentation for more information.

String

BSON strings are UTF-8. In general, drivers for each programming language convert from the language's string format to UTF-8 when serializing and deserializing BSON. This makes it possible to store most international characters in BSON strings with ease. [19] In addition, MongoDB `$regex` queries support UTF-8 in the regex string.

Timestamps

BSON has a special timestamp type for *internal* MongoDB use and is **not** associated with the regular *Date* (page 38) type. Timestamp values are a 64 bit value where:

- the first 32 bits are a `time_t` value (seconds since the Unix epoch)
- the second 32 bits are an incrementing `ordinal` for operations within a given second.

Within a single `mongod` instance, timestamp values are always unique.

In replication, the *oplog* has a `ts` field. The values in this field reflect the operation time, which uses a BSON timestamp value.

Note: The BSON Timestamp type is for *internal* MongoDB use. For most cases, in application development, you will want to use the BSON date type. See *Date* (page 38) for more information.

If you create a BSON Timestamp using the empty constructor (e.g. `new Timestamp()`), MongoDB will only generate a timestamp *if* you use the constructor in the first field of the document. [20] Otherwise, MongoDB will generate an empty timestamp value (i.e. `Timestamp(0, 0)`.)

[19] Given strings using UTF-8 character sets, using `sort()` on strings will be reasonably correct. However, because internally `sort()` uses the C++ `strcmp` api, the sort order may handle some characters incorrectly.

[20] If the first field in the document is `_id`, then you can generate a timestamp in the *second* field of a document.

Changed in version 2.1: `mongo` shell displays the Timestamp value with the wrapper:

```
Timestamp(<time_t>, <ordinal>)
```

Prior to version 2.1, the `mongo` shell display the Timestamp value as a document:

```
{ t : <time_t>, i : <ordinal> }
```

Date

BSON Date is a 64-bit integer that represents the number of milliseconds since the Unix epoch (Jan 1, 1970). The official BSON specification[21] refers to the BSON Date type as the *UTC datetime*.

Changed in version 2.0: BSON Date type is signed. [22] Negative values represent dates before 1970.

Example

Construct a Date using the `new Date()` constructor in the `mongo` shell:

```
var mydate1 = new Date()
```

Example

Construct a Date using the `ISODate()` constructor in the `mongo` shell:

```
var mydate2 = ISODate()
```

Example

Return the `Date` value as string:

```
mydate1.toString()
```

Example

Return the month portion of the Date value; months are zero-indexed, so that January is month `0`:

```
mydate1.getMonth()
```

---

[21] http://bsonspec.org/#/specification

[22] Prior to version 2.0, `Date` values were incorrectly interpreted as *unsigned* integers, which affected sorts, range queries, and indexes on `Date` fields. Because indexes are not recreated when upgrading, please re-index if you created an index on `Date` values with an earlier version, and dates before 1970 are relevant to your application.

Index