Data Model Design for MongoDB

Release 3.2.4

MongoDB, Inc.

March 13, 2016

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.

CHAPTER 1

Data Modeling Introduction

On this page

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*.

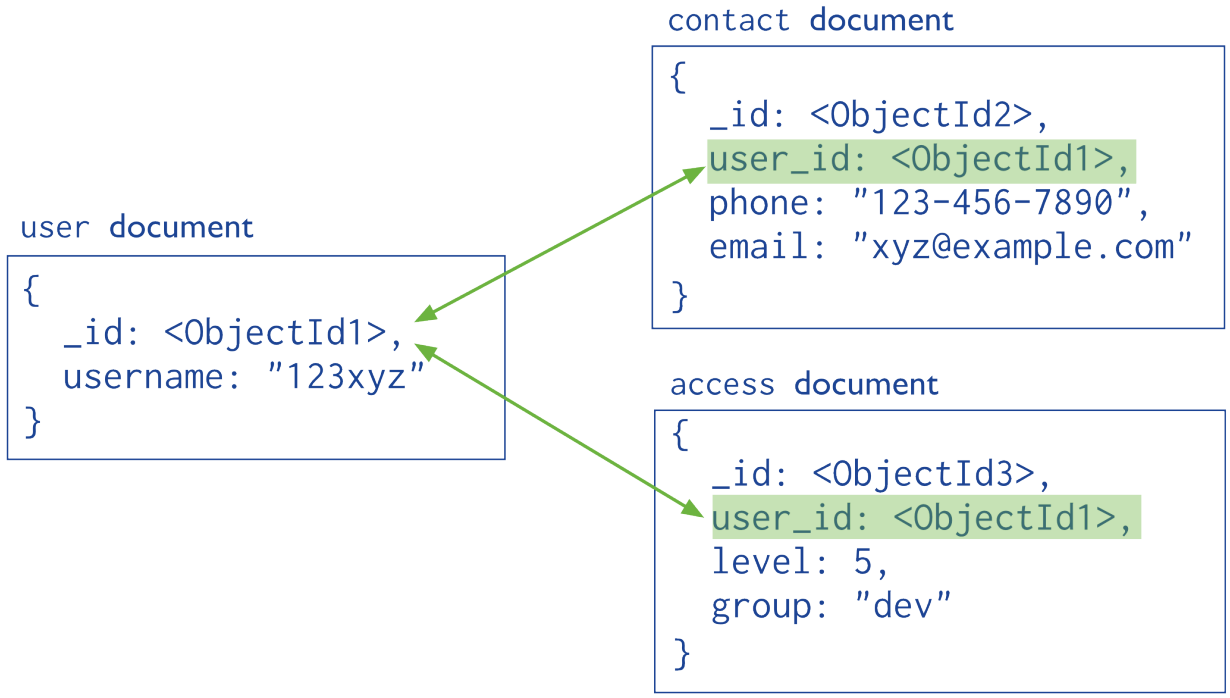
1.1.1 References

References store the relationships between data by including links or *references* from one document to another. Applications can resolve these *references* (page 39) to access the related data. Broadly, these are *normalized* data models.

See *Normalized Data Models* (page 12) for the strengths and weaknesses of using references.

1.1.2 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 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.


contact document
{
_id: <ObjectId2>,
user_id: <ObjectId1>,
phone: "123-456-7890",
email: "xyz@example.com"
}
user document
{
_id: <ObjectId1>,
username: "123xyz"
}
access document
{
_id: <ObjectId3>,
user_id: <ObjectId1>,
level: 5,
group: "dev"
}


{
_id: <ObjectId1>,
username: "123xyz",
contact: {
phone: "123-456-7890",
email: "xyz@example.com"
},
access: {
level: 5,
group: "dev"
}
}
Embedded sub-document
Embedded sub-document

See *Embedded Data Models* (page 11) for the strengths and weaknesses of embedding 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 14) 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.

For the MMAPv1 storage engine, if the document size exceeds the allocated space for that document, MongoDB relocates the document on disk. When using the MMAPv1 storage engine, growth consideration can affect the decision to normalize or denormalize data. See *Document Growth Considerations* (page 14) for more about planning for and managing document growth for MMAPv1.

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 `https://docs.mongodb.org/manual/core/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 13) for more information on these and other operational considerations that affect data model designs.

1.5 Additional Resources

- Thinking in Documents Part 1 (Blog Post)[1]

[1] https://www.mongodb.com/blog/post/thinking-documents-part-1?jmp=docs

CHAPTER 2

Document Validation

On this page

New in version 3.2.

MongoDB provides the capability to validate documents during updates and insertions. Validation rules are specified on a per-collection basis using the `validator` option, which takes a document that specifies the validation rules or expressions. Specify the expressions using any *query operators*, with the exception of `$near`, `$nearSphere`, `$text`, and `$where`.

Add document validation to an existing collection using the `collMod` command with the `validator` option. You can also specify document validation rules when creating a new collection using `db.createCollection()` with the `validator` option, as in the following:

```
db.createCollection( "contacts",
   { validator: { $or:
      [
         { phone: { $type: "string" } },
         { email: { $regex: /@mongodb\.com$/ } },
         { status: { $in: [ "Unknown", "Incomplete" ] } }
      ]
   }
} )
```

MongoDB also provides the `validationLevel` option, which determines how strictly MongoDB applies validation rules to existing documents during an update, and the `validationAction` option, which determines whether MongoDB should `error` and reject documents that violate the validation rules or `warn` about the violations in the log but allow invalid documents.

2.1 Behavior

Validation occurs during updates and inserts. When you add validation to a collection, existing documents do not undergo validation checks until modification.

2.1.1 Existing Documents

You can control how MongoDB handles existing documents using the `validationLevel` option.

By default, `validationLevel` is `strict` and MongoDB applies validation rules to all inserts and updates. Setting `validationLevel` to `moderate` applies validation rules to inserts and to updates to existing documents that fulfill the validation criteria. With the `moderate` level, updates to existing documents that do not fulfill the validation criteria are not checked for validity.

Example

Consider the following documents in a `contacts` collection:

```
{
   "_id": "125876"
   "name": "Anne",
   "phone": "+1 555 123 456",
   "city": "London",
   "status": "Complete"
},
{
   "_id": "860000",
   "name": "Ivan",
   "city": "Vancouver"
}
```

Issue the following command to add a validator to the `contacts` collection:

```
db.runCommand( {
   collMod: "contacts",
   validator: { $or: [ { phone: { $exists: true } }, { email: { $exists: true } } ] },
   validationLevel: "moderate"
} )
```

The `contacts` collection now has a validator with the `moderate` validationLevel. If you attempted to update the document with `_id` of `125876`, MongoDB would apply validation rules since the existing document matches the criteria. In contrast, MongoDB will not apply validation rules to updates to the document with `_id` of `860000` as it does not meet the validation rules.

To disable validation entirely, you can set `validationLevel` to `off`.

2.1.2 Accept or Reject Invalid Documents

The `validationAction` option determines how MongoDB handles documents that violate the validation rules.

By default, `validationAction` is `error` and MongoDB rejects any insertion or update that violates the validation criteria. When `validationAction` is set to `warn`, MongoDB logs any violations but allows the insertion or update to proceed.

Example

The following example creates a `contacts` collection with a validator that specifies that inserted or updated documents should match at least one of three following conditions:

- the `phone` field is a string
- the `email` field matches the regular expression
- the `status` field is either `Unknown` or `Incomplete`.

```
db.createCollection( "contacts",
   {
      validator: { $or:
         [
            { phone: { $type: "string" } },
            { email: { $regex: /@mongodb\.com$/ } },
            { status: { $in: [ "Unknown", "Incomplete" ] } }
         ]
      },
      validationAction: "warn"
   }
)
```

With the validator in place, the following insert operation fails the validation rules, but since the `validationAction` is `warn`, the write operation logs the failure and succeeds.

```
db.contacts.insert( { name: "Amanda", status: "Updated" } )
```

The log includes the full namespace of the collection and the document that failed the validation rules, as well as the time of the operation:

```
2015-10-15T11:20:44.260-0400 W STORAGE  [conn3] Document would fail validation collection: example.co
```

2.2 Restrictions

You cannot specify a validator for collections in the `admin`, `local`, and `config` databases.

You cannot specify a validator for `system.*` collections.

2.3 Bypass Document Validation

User can bypass document validation using the `bypassDocumentValidation` option. For a list of commands that support the `bypassDocumentValidation` option, see *3.2-rel-notes-document-validation*.

For deployments that have enabled access control, to bypass document validation, the authenticated user must have `bypassDocumentValidation` action. The built-in roles `dbAdmin` and `restore` provide this action.

2.4 Additional Information

See also:

`collMod`, `db.createCollection()`, `db.getCollectionInfos()`.

CHAPTER 3

Data Modeling Concepts

Consider the following aspects of data modeling in MongoDB:

Data Model Design **(page 11)** 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 13)** Details features you should keep in mind when designing your data model, such as lifecycle management, indexing, horizontal scalability, and document growth.

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 19).

3.1 Data Model Design

On this page

Effective data models support your application needs. The key consideration for the structure of your documents is the decision to *embed* (page 11) or to *use references* (page 12).

3.1.1 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 20).
- 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 21).

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. With the MMAPv1 storage engine, document growth can impact write performance and lead to data fragmentation.

In version 3.0.0, MongoDB uses *power-of-2-allocation* as the default allocation strategy for MMAPv1 in order to account for document growth, minimizing the likelihood of data fragmentation. See *power-of-2-allocation* for details. Furthermore, documents in MongoDB must be smaller than the `maximum BSON document size`. For bulk binary data, consider `GridFS`.

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

3.1.2 Normalized Data Models

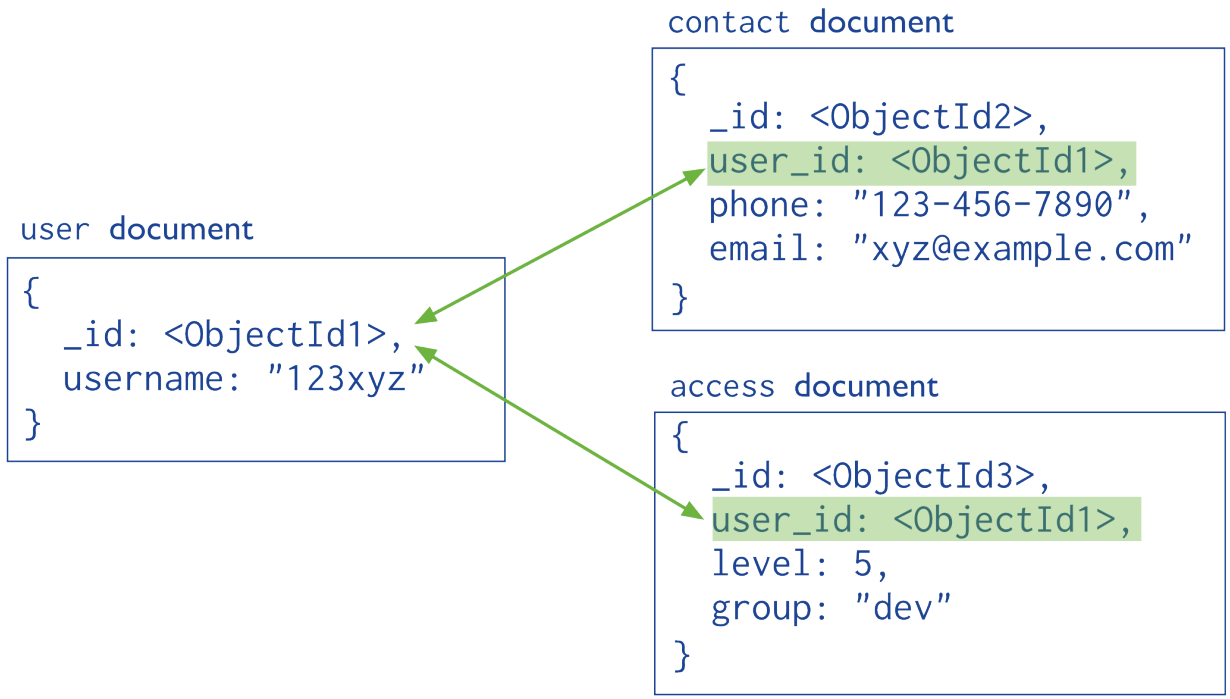
Normalized data models describe relationships using *references* (page 39) between documents.

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 round trips to the server.

See *Model One-to-Many Relationships with Document References* (page 22) for an example of referencing. For examples of various tree models using references, see *Model Tree Structures* (page 24).



3.1.3 Additional Resources

- Thinking in Documents Part 1 (Blog Post)[1]
- Thinking in Documents (Presentation)[2]
- Schema Design for Time Series Data (Presentation)[3]
- Socialite, the Open Source Status Feed - Storing a Social Graph (Presentation)[4]
- MongoDB Rapid Start Consultation Services[5]

3.2 Operational Factors and Data Models

[1] https://www.mongodb.com/blog/post/thinking-documents-part-1?jmp=docs
[2] http://www.mongodb.com/presentations/webinar-back-basics-1-thinking-documents?jmp=docs
[3] http://www.mongodb.com/presentations/webinar-time-series-data-mongodb?jmp=docs
[4] http://www.mongodb.com/presentations/socialite-open-source-status-feed-part-2-managing-social-graph?jmp=docs
[5] https://www.mongodb.com/products/consulting?jmp=docs#rapid_start

On this page

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.

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.

3.2.1 Document Growth

Changed in version 3.0.0.

Some updates to documents can increase the size of documents. These updates include pushing elements to an array (i.e. `$push`) and adding new fields to a document.

When using the MMAPv1 storage engine, document growth can be a consideration for your data model. For MMAPv1, if the document size exceeds the allocated space for that document, MongoDB will relocate the document on disk. With MongoDB 3.0.0, however, the default use of the *power-of-2-allocation* minimizes the occurrences of such re-allocations as well as allows for the effective reuse of the freed record space.

When using MMAPv1, if your applications require updates that will frequently cause document growth to exceeds the current power of 2 allocation, you may want to refactor your data model to use references between data in distinct documents rather than a denormalized data model.

You may also use a *pre-allocation* strategy to explicitly avoid document growth. Refer to the Pre-Aggregated Reports Use Case[6] for an example of the *pre-allocation* approach to handling document growth.

See `https://docs.mongodb.org/manual/core/mmapv1` for more information on MMAPv1.

3.2.2 Atomicity

In MongoDB, operations are atomic at the *document* level. No **single** write operation can change more than one document. Operations that modify more than a single document in a collection still operate on one document at a time. [7] 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 32) for an example data model that provides atomic updates for a single document.

[6] https://docs.mongodb.org/ecosystem/use-cases/pre-aggregated-reports

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

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

3.2.4 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 8 kB 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 `https://docs.mongodb.org/manual/applications/indexes` for more information on indexes as well as `https://docs.mongodb.org/manual/tutorial/analyze-query-plan/`. Additionally, the MongoDB `database profiler` may help identify inefficient queries.

3.2.5 Large Number of Collections

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

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 8 kB 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 using the `mmapv1` storage engine 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`.

3.2.6 Collection Contains Large Number of Small Documents

You should consider embedding for performance reasons if you have a collection with a large number of small documents. If you can group these small documents by some logical relationship *and* you frequently retrieve the documents by this grouping, you might consider "rolling-up" the small documents into larger documents that contain an array of embedded documents.

"Rolling up" these small documents into logical groupings means that queries to retrieve a group of documents involve sequential reads and fewer random disk accesses. Additionally, "rolling up" documents and moving common fields to the larger document benefit the index on these fields. There would be fewer copies of the common fields *and* there would be fewer associated key entries in the corresponding index. See `https://docs.mongodb.org/manual/core/indexes` for more information on indexes.

However, if you often only need to retrieve a subset of the documents within the group, then "rolling-up" the documents may not provide better performance. Furthermore, if small, separate documents represent the natural model for the data, you should maintain that model.

3.2.7 Storage Optimization for Small Documents

Each MongoDB document contains a certain amount of overhead. This overhead is normally insignificant but becomes significant if all documents are just a few bytes, as might be the case if the documents in your collection only have one or two fields.

Consider the following suggestions and strategies for optimizing storage utilization for these collections:

- Use the `_id` field explicitly.

  MongoDB clients automatically add an `_id` field to each document and generate a unique 12-byte *ObjectId* for the `_id` field. Furthermore, MongoDB always indexes the `_id` field. For smaller documents this may account for a significant amount of space.

  To optimize storage use, users can specify a value for the `_id` field explicitly when inserting documents into the collection. This strategy allows applications to store a value in the `_id` field that would have occupied space in another portion of the document.

  You can store any value in the `_id` field, but because this value serves as a primary key for documents in the collection, it must uniquely identify them. If the field's value is not unique, then it cannot serve as a primary key as there would be collisions in the collection.

- Use shorter field names.

  Note: Shortening field names reduces expressiveness and does not provide considerable benefit for larger documents and where document overhead is not of significant concern. Shorter field names do not reduce the size of indexes, because indexes have a predefined structure.

  In general, it is not necessary to use short field names.

  MongoDB stores all field names in every document. For most documents, this represents a small fraction of the space used by a document; however, for small documents the field names may represent a proportionally large amount of space. Consider a collection of small documents that resemble the following:

  ```
  { last_name : "Smith", best_score: 3.9 }
  ```

  If you shorten the field named `last_name` to `lname` and the field named `best_score` to `score`, as follows, you could save 9 bytes per document.

  ```
  { lname : "Smith", score : 3.9 }
  ```

- Embed documents.

  In some cases you may want to embed documents in other documents and save on the per-document overhead. See *Collection Contains Large Number of Small Documents* (page 16).

3.2.8 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 `https://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.

CHAPTER 4

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 19)** Examples for modeling relationships between documents.

- ***Model One-to-One Relationships with Embedded Documents*** **(page 20)** Presents a data model that uses *embedded documents* (page 11) to describe one-to-one relationships between connected data.
- ***Model One-to-Many Relationships with Embedded Documents*** **(page 21)** Presents a data model that uses *embedded documents* (page 11) to describe one-to-many relationships between connected data.
- ***Model One-to-Many Relationships with Document References*** **(page 22)** Presents a data model that uses *references* (page 12) to describe one-to-many relationships between documents.

Model Tree Structures **(page 24)** Examples for modeling tree structures.

- ***Model Tree Structures with Parent References*** **(page 25)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 12) to "parent" nodes in "child" nodes.
- ***Model Tree Structures with Child References*** **(page 26)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 12) to "child" nodes in "parent" nodes.

See *Model Tree Structures* (page 24) for additional examples of data models for tree structures.

Model Specific Application Contexts **(page 32)** Examples for models for specific application contexts.

- ***Model Data for Atomic Operations*** **(page 32)** 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 33)** 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.

4.1 Model Relationships Between Documents

Model One-to-One Relationships with Embedded Documents **(page 20)** Presents a data model that uses *embedded documents* (page 11) to describe one-to-one relationships between connected data.

Model One-to-Many Relationships with Embedded Documents **(page 21)** Presents a data model that uses *embedded documents* (page 11) to describe one-to-many relationships between connected data.

Model One-to-Many Relationships with Document References **(page 22)** Presents a data model that uses *references* (page 12) to describe one-to-many relationships between documents.

4.1.1 Model One-to-One Relationships with Embedded Documents

On this page

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 11) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *embedded* (page 11) 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.

4.1.2 Model One-to-Many Relationships with Embedded Documents

On this page

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 11) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *embedded* (page 11) 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.

4.1.3 Model One-to-Many Relationships with Document References

On this page

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 11) for a full high level overview of data modeling in MongoDB.

This document describes a data model that uses *references* (page 12) 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"
}
```

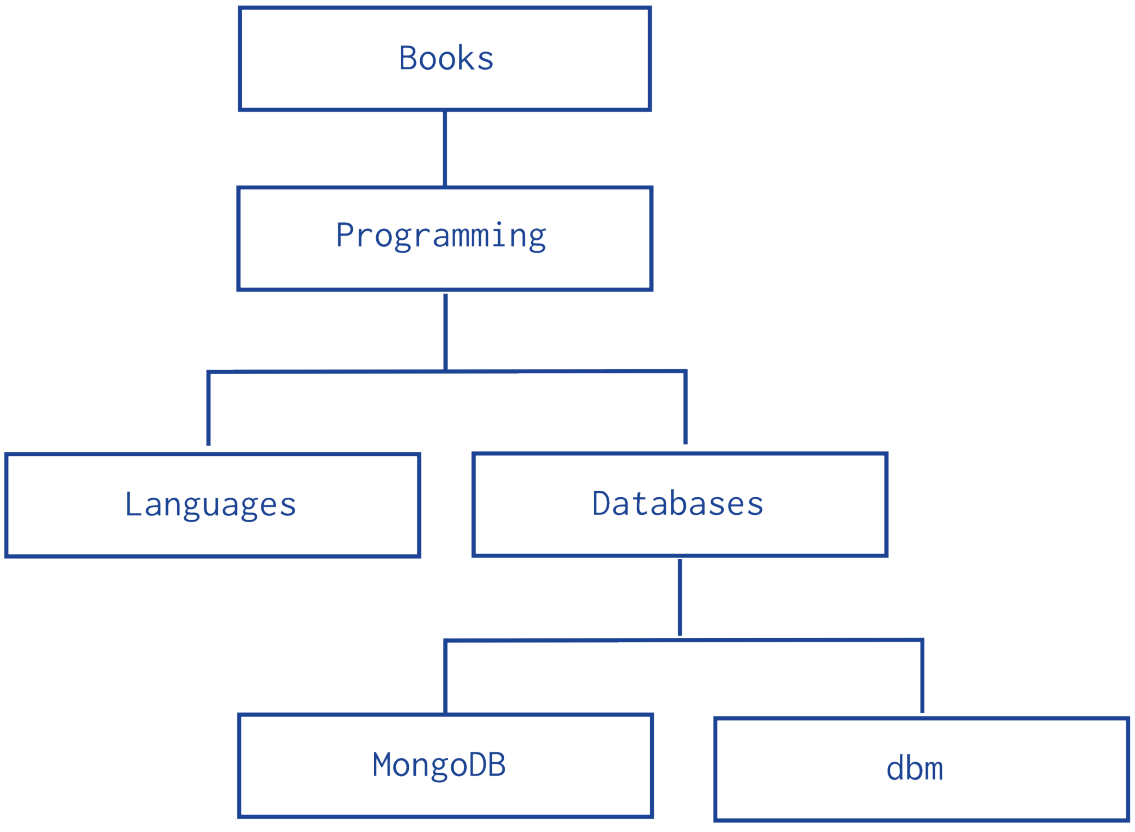
To avoid mutable, growing arrays, store the publisher reference inside the book document:

```
{
   _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"
}
```

4.2 Model Tree Structures

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



Model Tree Structures with Parent References **(page 25)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 12) to "parent" nodes in "child" nodes.

Model Tree Structures with Child References **(page 26)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 12) to "child" nodes in "parent" nodes.

Model Tree Structures with an Array of Ancestors **(page 27)** Presents a data model that organizes documents in a tree-like structure by storing *references* (page 12) to "parent" nodes and an array that stores all ancestors.

Model Tree Structures with Materialized Paths **(page 29)** 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 31)** 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.

4.2.1 Model Tree Structures with Parent References

On this page

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 11) 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 12) 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:

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
  ```

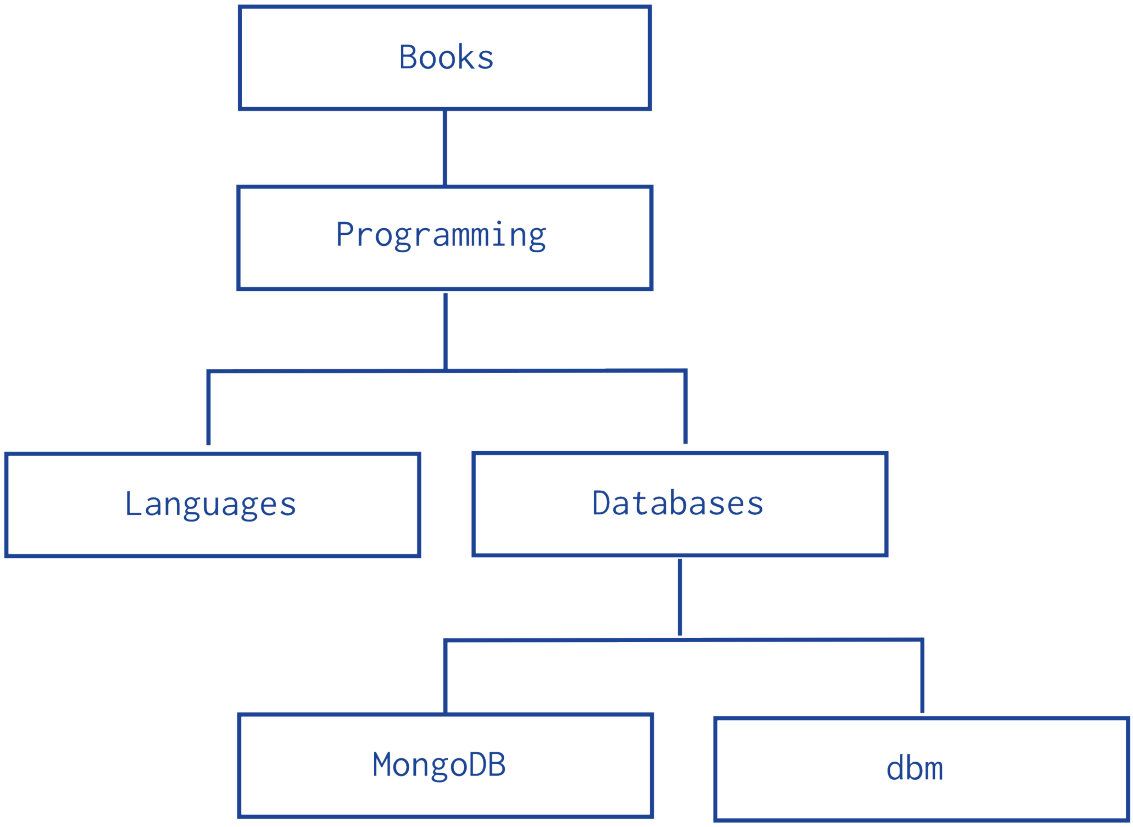
- You can create an index on the field `parent` to enable fast search by the parent node:

  ```
  db.categories.createIndex( { 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.



4.2.2 Model Tree Structures with Child References

On this page

Overview

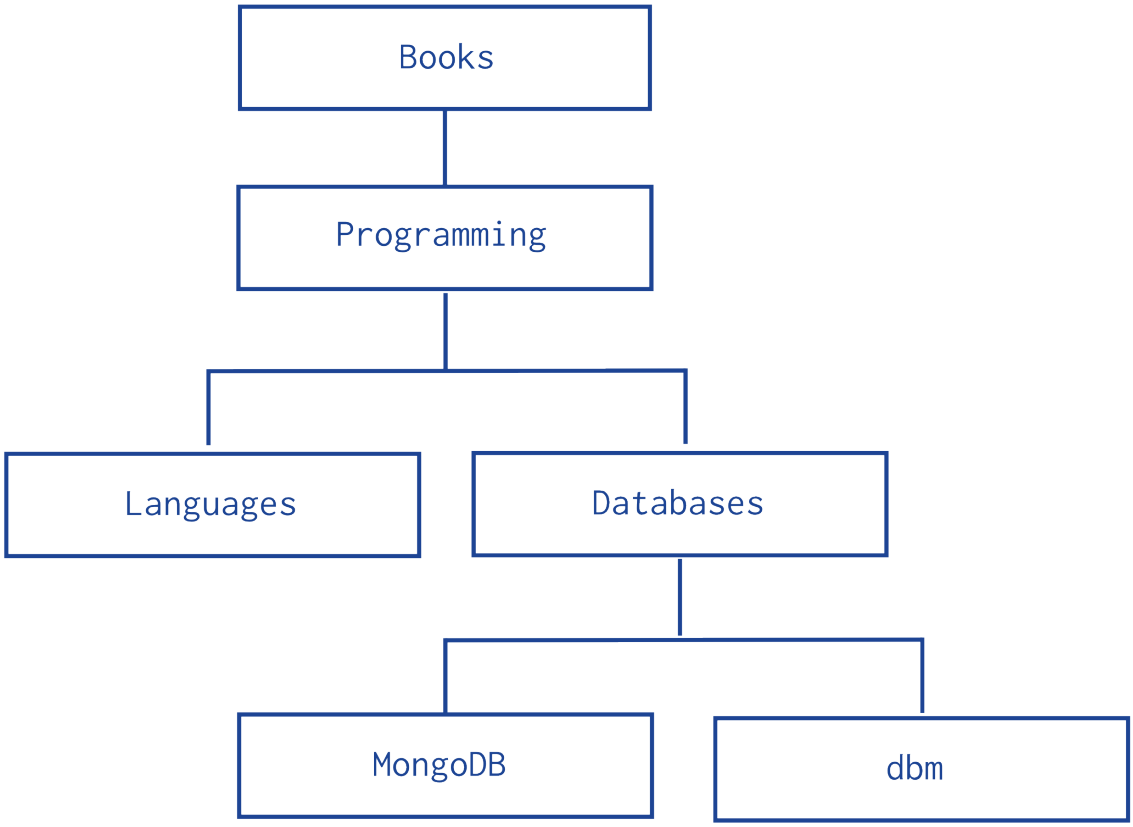
Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 11) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents by storing *references* (page 12) 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.createIndex( { 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" } )
  ```

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.

4.2.3 Model Tree Structures with an Array of Ancestors

On this page

Overview

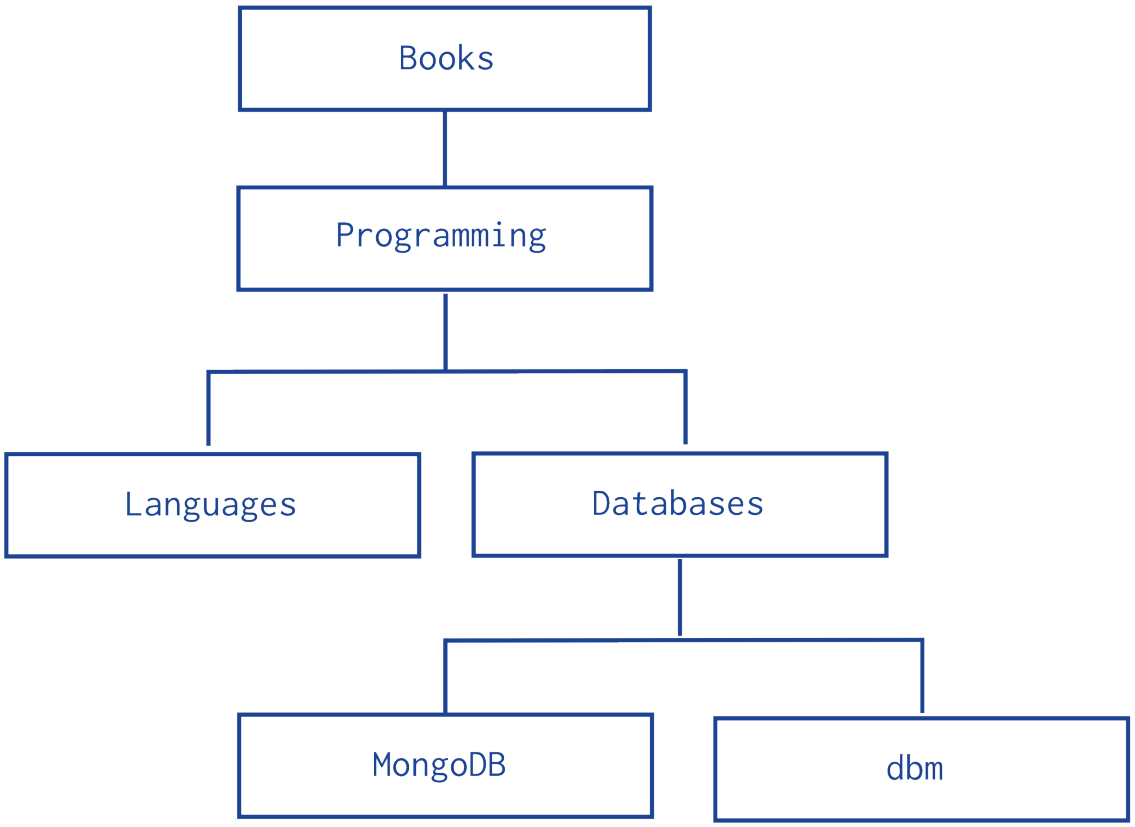
Data in MongoDB has a *flexible schema*. *Collections* do not enforce *document* structure. Decisions that affect how you model data can affect application performance and database capacity. See *Data Modeling Concepts* (page 11) for a full high level overview of data modeling in MongoDB.

This document describes a data model that describes a tree-like structure in MongoDB documents using *references* (page 12) 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:



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.createIndex( { 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 29) pattern but is more straightforward to use.

4.2.4 Model Tree Structures with Materialized Paths

On this page

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 11) for a full high level overview of data modeling in MongoDB.

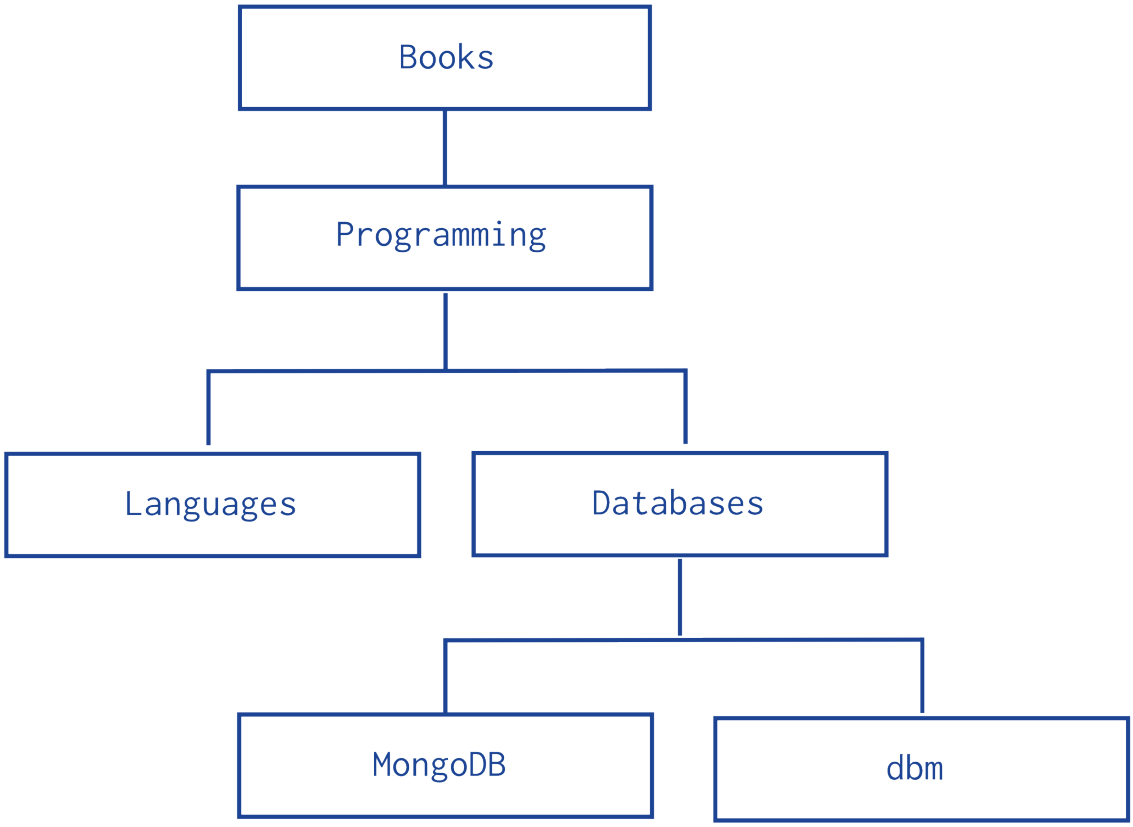
This document describes a data model that describes a tree-like structure in MongoDB documents by storing full relationship paths between documents.

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," } )
```

- 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.createIndex( { path: 1 } )
  ```

  This index may improve performance depending on the query:

  - For queries from the root `Books` sub-tree (e.g. `https://docs.mongodb.org/manual/^,Books,/` or `https://docs.mongodb.org/manual/^,Books,Programming,/`), an index on the `path` field improves the query performance significantly.

– For queries of sub-trees where the path from the root is not provided in the query (e.g. `https://docs.mongodb.org/manual/,Databases,/`), or similar queries of sub-trees, 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.

4.2.5 Model Tree Structures with Nested Sets

On this page

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 11) 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:

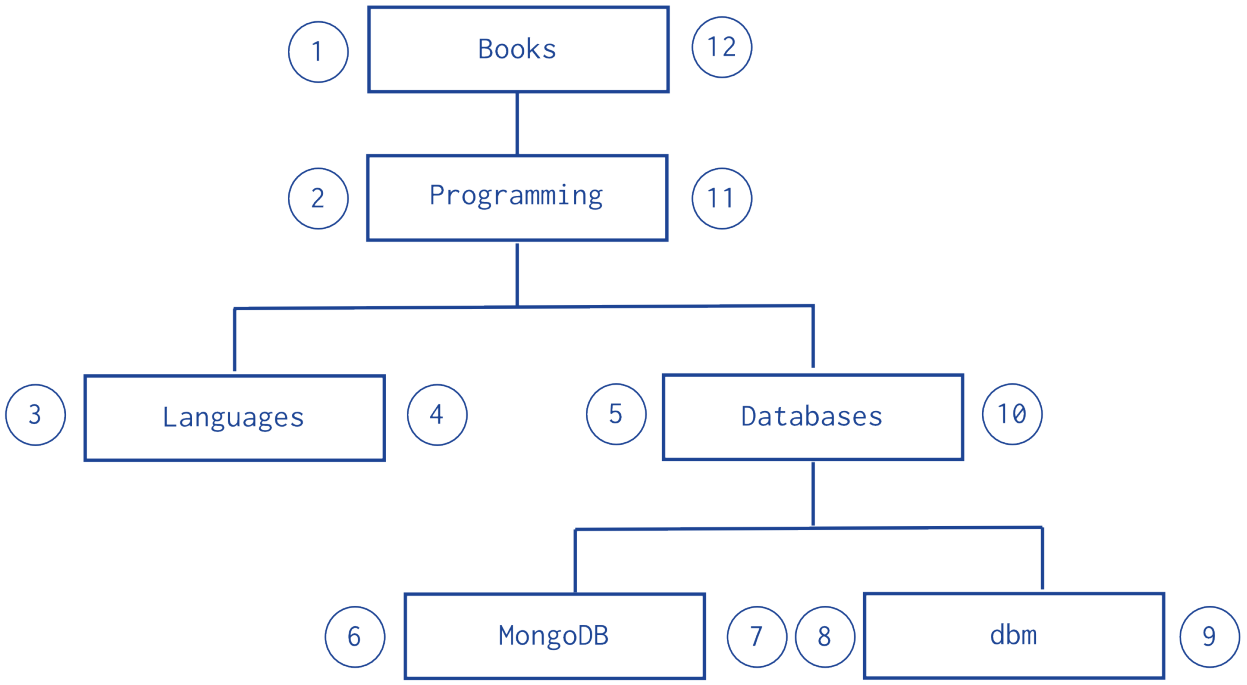
The following example models the tree using *Nested Sets*:

```
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:

```
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.



4.3 Model Specific Application Contexts

Model Data for Atomic Operations **(page 32)** 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 33)** 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 Monetary Data **(page 34)** Describes two methods to model monetary data in MongoDB.

Model Time Data **(page 36)** Describes how to deal with local time in MongoDB.

4.3.1 Model Data for Atomic Operations

On this page

Pattern

In MongoDB, write operations, e.g. `db.collection.update()`, `db.collection.findAndModify()`, `db.collection.remove()`, are atomic on the level of a single document. For fields that must be updated together, embedding the fields within the same document ensures that the fields can be updated atomically.

For example, consider a situation where you need to maintain information on books, including the number of copies available for checkout as well as the current checkout information.

The available copies of the book and the checkout information should be in sync. As such, embedding the `available` field and the `checkout` field within the same document ensures that you can update the two fields atomically.

```
{
    _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") } ]
}
```

Then to update with new checkout information, you can use the `db.collection.update()` method to atomically update both the `available` field and the `checkout` field:

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

The operation returns a `WriteResult()` object that contains information on the status of the operation:

```
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

The `nMatched` field shows that `1` document matched the update condition, and `nModified` shows that the operation updated `1` document.

If no document matched the update condition, then `nMatched` and `nModified` would be `0` and would indicate that you could not check out the book.

4.3.2 Model Data to Support Keyword Search

On this page

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 34) section for more information.

In 2.4, MongoDB provides a text search feature. See `https://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.createIndex( { 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.

4.3.3 Model Monetary Data

On this page

Overview

MongoDB stores numeric data as either IEEE 754 standard 64-bit floating point numbers or as 32-bit or 64-bit signed integers. Applications that handle monetary data often require capturing fractional units of currency. However, arithmetic on floating point numbers, as implemented in modern hardware, often does not conform to requirements for monetary arithmetic. In addition, some fractional numeric quantities, such as one third and one tenth, have no exact representation in binary floating point numbers.

Note: Arithmetic mentioned on this page refers to server-side arithmetic performed by `mongod` or `mongos`, and not to client-side arithmetic.

This document describes two ways to model monetary data in MongoDB:

- *Exact Precision* (page 35) which multiplies the monetary value by a power of 10.
- *Arbitrary Precision* (page 36) which uses two fields for the value: one field to store the exact monetary value as a non-numeric and another field to store a floating point approximation of the value.

Use Cases for Exact Precision Model

If you regularly need to perform server-side arithmetic on monetary data, the exact precision model may be appropriate. For instance:

- If you need to query the database for exact, mathematically valid matches, use *Exact Precision* (page 35).
- If you need to be able to do server-side arithmetic, e.g., `$inc`, `$mul`, and `aggregation framework arithmetic`, use *Exact Precision* (page 35).

Use Cases for Arbitrary Precision Model

If there is no need to perform server-side arithmetic on monetary data, modeling monetary data using the arbitrary precision model may be suitable. For instance:

- If you need to handle arbitrary or unforeseen number of precision, see *Arbitrary Precision* (page 36).
- If server-side approximations are sufficient, possibly with client-side post-processing, see *Arbitrary Precision* (page 36).

Exact Precision

To model monetary data using the exact precision model:

1. Determine the maximum precision needed for the monetary value. For example, your application may require precision down to the tenth of one cent for monetary values in `USD` currency.

2. Convert the monetary value into an integer by multiplying the value by a power of 10 that ensures the maximum precision needed becomes the least significant digit of the integer. For example, if the required maximum precision is the tenth of one cent, multiply the monetary value by 1000.
3. Store the converted monetary value.

For example, the following scales `9.99 USD` by 1000 to preserve precision up to one tenth of a cent.

```
{ price: 9990, currency: "USD" }
```

The model assumes that for a given currency value:

- The scale factor is consistent for a currency; i.e. same scaling factor for a given currency.
- The scale factor is a constant and known property of the currency; i.e applications can determine the scale factor from the currency.

When using this model, applications must be consistent in performing the appropriate scaling of the values.

For use cases of this model, see *Use Cases for Exact Precision Model* (page 35).

Arbitrary Precision

To model monetary data using the arbitrary precision model, store the value in two fields:

1. In one field, encode the exact monetary value as a non-numeric data type; e.g., `BinData` or a `string`.
2. In the second field, store a double-precision floating point approximation of the exact value.

The following example uses the arbitrary precision model to store `9.99 USD` for the price and `0.25 USD` for the fee:

```
{
  price: { display: "9.99", approx: 9.9900000000000002, currency: "USD" },
  fee: { display: "0.25", approx: 0.2499999999999999, currency: "USD" }
}
```

With some care, applications can perform range and sort queries on the field with the numeric approximation. However, the use of the approximation field for the query and sort operations requires that applications perform client-side post-processing to decode the non-numeric representation of the exact value and then filter out the returned documents based on the exact monetary value.

For use cases of this model, see *Use Cases for Arbitrary Precision Model* (page 35).

4.3.4 Model Time Data

On this page

Overview

MongoDB *stores times in UTC* by default, and will convert any local time representations into this form. Applications that must operate or report on some unmodified local time value may store the time zone alongside the UTC timestamp, and compute the original local time in their application logic.

Example

In the MongoDB shell, you can store both the current date and the current client's offset from UTC.

```
var now = new Date();
db.data.save( { date: now,
                offset: now.getTimezoneOffset() } );
```

You can reconstruct the original local time by applying the saved offset:

```
var record = db.data.findOne();
var localNow = new Date( record.date.getTime() -  ( record.offset * 60000 ) );
```

CHAPTER 5

Data Model Reference

Database References **(page 39)** Discusses manual references and DBRefs, which MongoDB can use to represent relationships between documents.

5.1 Database References

On this page

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:

- *Manual references* (page 40) 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 related data. These references are simple and sufficient for most use cases.
- *DBRefs* (page 40) are references from one document to another using the value of the first document's `_id` field, collection name, and, optionally, its database name. By including these names, DBRefs allow documents located in multiple collections to be more easily linked with documents from a single collection.

  To resolve DBRefs, your application must perform additional queries to return the referenced documents. Many `drivers` have helper methods that form the query for the DBRef automatically. The drivers [1] do not *automatically* resolve DBRefs into documents.

  DBRefs provide a common format and type to represent relationships among documents. The DBRef format also provides common semantics for representing links between documents if your database must interact with multiple frameworks and tools.

Unless you have a compelling reason to use DBRefs, use manual references instead.

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

5.1.1 Manual References

Background

Using manual references is 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 40). 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 names. If you have documents in a single collection that relate to documents in more than one collection, you may need to consider using DBRefs.

5.1.2 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 name, 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 documents resemble the following document:

```
{ "$ref" : <value>, "$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.

Driver Support for DBRefs

C	The C driver contains no support for DBRefs. You can traverse references manually.
C++	The C++ driver contains no support for DBRefs. You can traverse references manually.
C#	The C# driver supports DBRefs using the MongoDBRef[2] class and `FetchDBRef` and `FetchDBRefAs` methods.
Haskell	The Haskell driver contains no support for DBRefs. You can traverse references manually.
Java	The DBRef[3] class provides support for DBRefs from Java.
JavaScript	The `mongo` shell's `JavaScript` interface provides a DBRef.
Node.js	The Node.js driver supports DBRefs using the DBRef[4] class and the dereference[5] method.
Perl	The Perl driver supports DBRefs using the MongoDB::DBRef[6] class. You can traverse references manually.
PHP	The PHP driver supports DBRefs, including the optional `$db` reference, using the MongoDBRef[7] class.
Python	The Python driver supports DBRefs using the DBRef[8] class and the dereference[9] method.
Ruby	The Ruby driver supports DBRefs using the DBRef[10] class and the dereference[11] method.
Scala	The Scala driver contains no support for DBRefs. You can traverse references manually.

[2] https://api.mongodb.org/csharp/current/html/T_MongoDB_Driver_MongoDBRef.htm
[3] https://api.mongodb.org/java/current/com/mongodb/DBRef.html
[4] http://mongodb.github.io/node-mongodb-native/api-bson-generated/db_ref.html
[5] http://mongodb.github.io/node-mongodb-native/api-generated/db.html#dereference
[6] https://metacpan.org/pod/MongoDB::DBRef
[7] http://www.php.net/manual/en/class.mongodbref.php/
[8] https://api.mongodb.org/python/current/api/bson/dbref.html
[9] https://api.mongodb.org/python/current/api/pymongo/database.html#pymongo.database.Database.deref eren ce
[10] https://api.mongodb.org/ruby/current/BSON/DBRef.html
[11] https://api.mongodb.org/ruby/current/Mongo/DB.html#dereference-instance_method

Use

In most cases you should use the *manual reference* (page 40) method for connecting two or more related documents. However, if you need to reference documents from multiple collections, consider using DBRefs.