100/100

MongoDB Overview

Bingo Bango Mongo

Michelle Duer *Ian Percy* *Billy Haugen* *Yves Wienecke* *Tejas Menon*

CS488/588 Cloud & Cluster Data Management

Table of Contents

Data Model

MongoDB is a document store system with data modeling that allows for embedded documents (aka *nesting*), object references, and a “flexible schema” [1]. These characteristics differ from relational databases by allowing more complex data representations through embeddings that also allow for clean BSON representation, “a binary JSON-like format” [2, p. 18]. The object references, also called links, reduces redundancy and improves data representation and navigation. The flexible schema, which some would call schema-less, allows for memory savings and more flexible data representation.

To expand on the flexible schema using an example, let us consider a site that stores data for users. How might data storage for two users, *user1* and *user2*, compare on either system? In a relational database management system, both user1 and user2 would require identical schemas with details {*A*, *B*, *C*, *D*, *E*} where empty data may be represented by a null value. MongoDB, on the other hand, can have a document with user1’s details: {*A*, *B*, *C*}, while a second document contains user2 details: {*A*, *B*, *D*, *E*} without storing null values when the user details differ.

Applying MongoDB to our AirBnB dataset, we can make many different data models. For example, a flat one-to-one model where an id is used as a primary key in both a small document of listing details and as a primary key for a significantly larger document with all of the remaining listing details. This simple data model could look as follows:

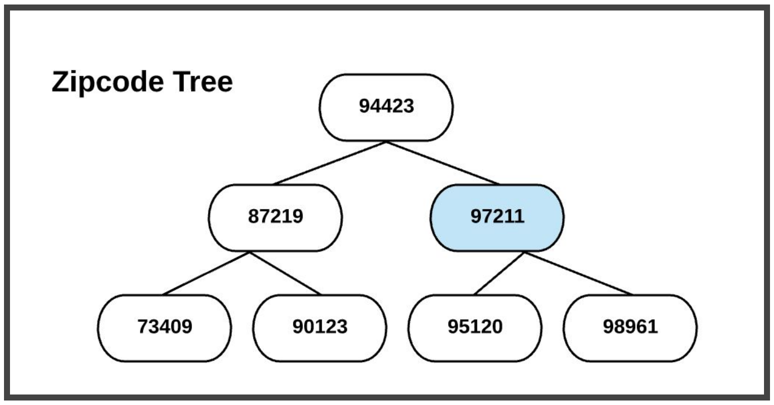
Example Data Model 1

```
{
  _id: 12899,
  listing_url: https://www.airbnb.com/rooms/12899,
  name: “Alberta Arts 2 bedroom suite charming 1906 house”
}

{
  _id: 12899,
  host_id: 49682,
  host_name: “Ali and David”,
  host_since: 2009-10-29,
  host_location: “Portland”,
  host_is_superhost: t,
  neighborhood: “Concordia”,
  city: “Portland”,
  zipcode: 97211,
  …
  review_scores_accuracy: 10,
  reviews_per_month: 4.51
  …
  minimum_nights: 2,
  maximum_nights: 730,
  has_availability: t,
  availability_30: 10,
}
```

However, this approach results in an overwhelming list of details. A developer accessing these fields will waste time finding relevant details. Readability can be improved by breaking down the large document into smaller ones, perhaps a *host* document and a *location* document with a unifying listing *_id*.

An alternate model can take advantage of the embeddings, flexible schema, and consider the most frequent use case. Other than the original write operation to create a listing, most of the data may be accessed by a potential renter. A tree data structure can be constructed where each node is a zip code defined in the data model as *_id*. Each zip code *_id* contains a list of references to different tenant listings. Finally, a referenced object contains embeddings related to aggregate details. The following is a data model incorporating both the use case and MongoDB features:

Example Data Model 2



```
{
  _id: 97211,
  list: {
    1: {...},
    2: {...},
    3: {
      listing: 12899,
    }
    …
    N: {...},
  },
  children: [95120, 98961]
}
```

```
{
 listing_id: 12899,
  host: {
     host_id: 49682,
      host_name: “Ali and David”
      host_since: 2009-10-29,
      host_location: “Portland”,
      host_is_superhost: t,
  }
  location: {
      neighborhood: “Concordia”,
      city: “Portland”,
      zipcode: 97211,
  }
  …
  reviews: {
      review_scores_accuracy: 10,
      reviews_per_month: 4.51
   }
}
{
  listing_id: 12899,
  reservations: {
      minimum_nights: 2,
      maximum_nights: 730,
      has_availability: t,
      availability_30: 10,
   }
}
```

Narrowing down areas of interest for the potential renter is much easier using the zipcode tree -- these keys can also be used as partitioning values, discussed later.

When the subset of listing references are retrieved, the links allow for organized object access. Within each listing object, the embedded documents allow a developer to get relevant data regarding host details, location, or reviews, instead of needing to scan a long, difficult-to-parse list. The big picture of such a data model is that the application client has a better experience due to the efficient lookup and the developer has an easier time creating an organized UI, and the embedded documents are essentially like JSON. Because the data model is flexible and doesn't require a schema, further indexes could also be added to optimize operations based on use cases. Another benefit is that the model is easier for a new developer to learn compared to the first, more complicated model.

Example Functions

Examples of insertion and update functions for the second model:

```
db.zipcodes.insertOne({ _id: 97211, list: [] })

db.zipcodes.update(
  { _id: 97211 },
  { $push: { list: 12899 } }
)

db.listings.insertOne({
    listing_id: 12899,
    host: { host_id: 49682, host_name: “Ali and David”, host_since: 2009-10-29, host_location:
“Portland”, host_is_superhost: t }
     location: { neighborhood: “Concordia”, city: “Portland”, zipcode: 97211 }
   …
   reviews: { review_scores_accuracy: 9, reviews_per_month: 4.30 }
})
```

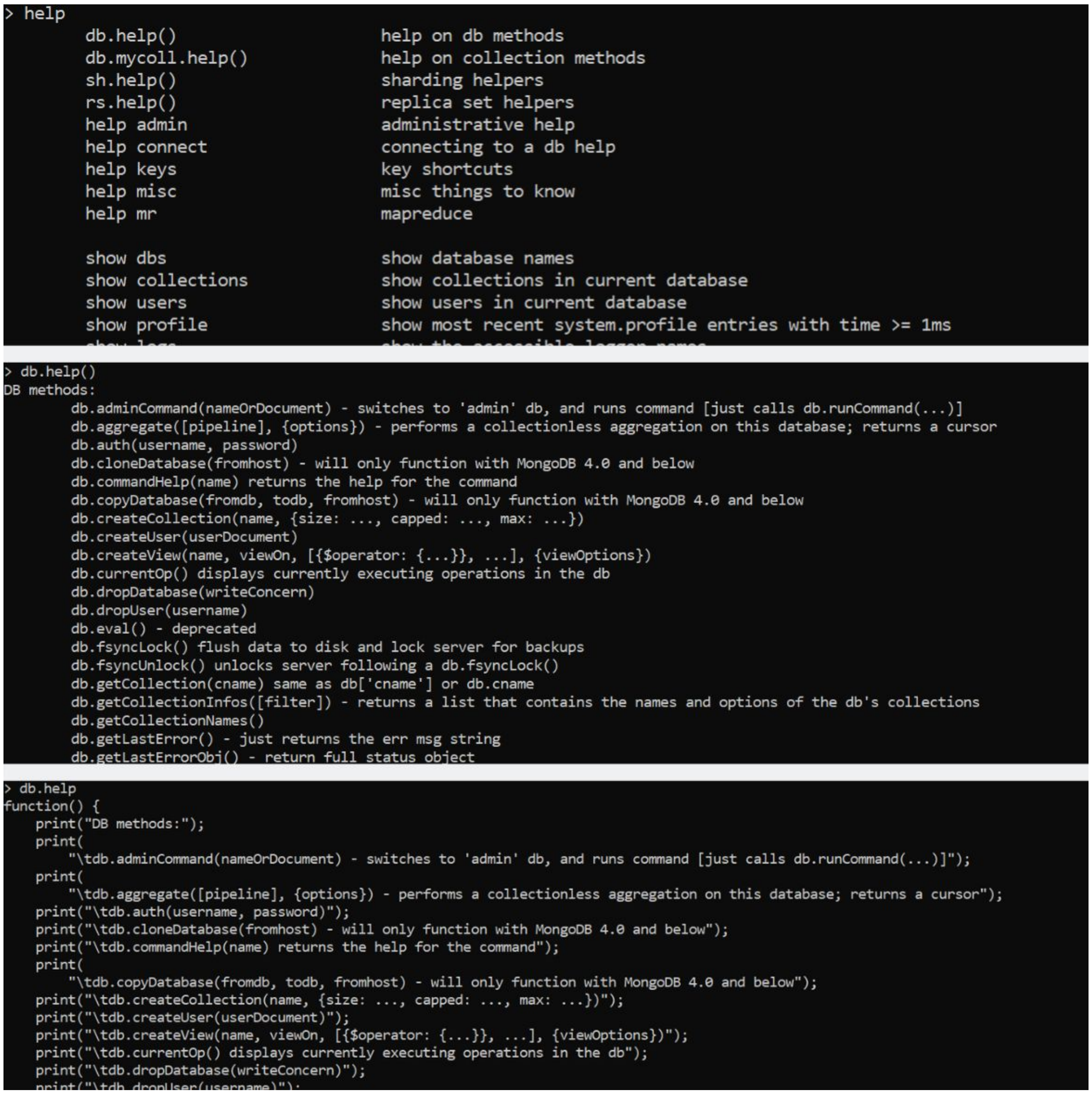
```
db.listings.update(
   { listing_id: 12899 },
   { $set: {
        reviews.review_scores_accuracy: 10,
        reviews.reviews_per_month: 4.51
     }
  }
)

db.listings.insertOne({
    listing_id: 12899,
    reservations: { minimum_nights: 2, maximum_nights: 730, has_availability: t,
availability_30: 10 }
})
```

User Interface

One of the critical characteristics of a database is the user interface. This is the face of the database: the APIs which the database reveals, the various functions for managing and querying entries, and communication between the database and the user. ‘User’ in our usage of the term will refer to administrators interacting with the database through the mongo shell, and developers who use drivers to connect to the database by means of programming languages such as C#, Python, and Node.js. In this section, we will focus on the mongo shell interface and official documentation. There are other tools which simplify MongoDB implementation and management, including: the compass GUI, cloud service provider tools, atlas, and drivers for programming languages. However, the user interface for these tools depends more on the tool’s implementation rather than mongoDB, so we will not include them in our analysis of the database system user interface.

First, we will consider the mongo database installation and set-up process. The mongo manual in the official documentation [1] provides clear, step-by-step information with pictures for installing the database and running it manually or setting it up as a service. The nav bar makes it easy to select installation information for a specific operating system. From start to finish, this process takes about 15-20 minutes and most of this time is spent waiting for the database to initialize.

Following installation, the user may run an instance of the mongo database server and client with the commands *mongod* and *mongo*, respectively. From the client side, the user enters the mongo shell and may interact with the database through various commands. In order to see all commands available, the user may enter the *help* command. To view help for a certain database entity or command, the user may append *help()* to view all commands and usage. For commands that end with parentheses, the user may omit the parentheses to view the code implementation.



```
> help
        db.help()                    help on db methods
        db.mycoll.help()             help on collection methods
        sh.help()                    sharding helpers
        rs.help()                    replica set helpers
        help admin                   administrative help
        help connect                 connecting to a db help
        help keys                    key shortcuts
        help misc                    misc things to know
        help mr                      mapreduce

        show dbs                     show database names
        show collections             show collections in current database
        show users                   show users in current database
        show profile                 show most recent system.profile entries with time >= 1ms
```

```
> db.help()
DB methods:
        db.adminCommand(nameOrDocument) - switches to 'admin' db, and runs command [just calls db.runCommand(...)]
        db.aggregate([pipeline], {options}) - performs a collectionless aggregation on this database; returns a cursor
        db.auth(username, password)
        db.cloneDatabase(fromhost) - will only function with MongoDB 4.0 and below
        db.commandHelp(name) returns the help for the command
        db.copyDatabase(fromdb, todb, fromhost) - will only function with MongoDB 4.0 and below
        db.createCollection(name, {size: ..., capped: ..., max: ...})
        db.createUser(userDocument)
        db.createView(name, viewOn, [{$operator: {...}}, ...], {viewOptions})
        db.currentOp() displays currently executing operations in the db
        db.dropDatabase(writeConcern)
        db.dropUser(username)
        db.eval() - deprecated
        db.fsyncLock() flush data to disk and lock server for backups
        db.fsyncUnlock() unlocks server following a db.fsyncLock()
        db.getCollection(cname) same as db['cname'] or db.cname
        db.getCollectionInfos([filter]) - returns a list that contains the names and options of the db's collections
        db.getCollectionNames()
        db.getLastError() - just returns the err msg string
        db.getLastErrorObj() - return full status object
```

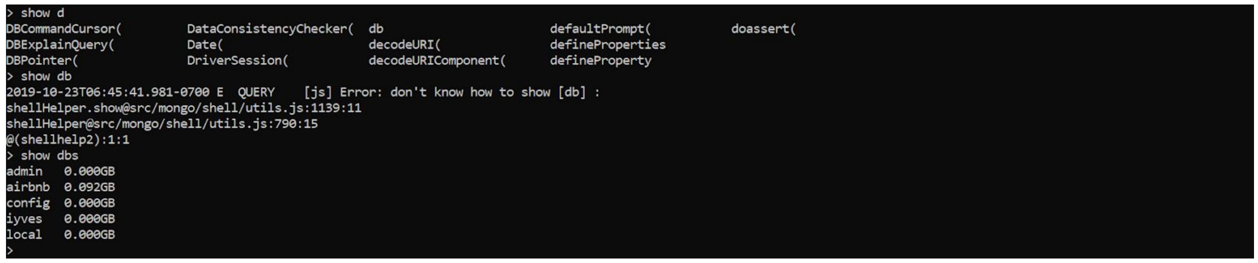
```
> db.help
function() {
    print("DB methods:");
    print(
        "\tdb.adminCommand(nameOrDocument) - switches to 'admin' db, and runs command [just calls db.runCommand(...)]");
    print(
        "\tdb.aggregate([pipeline], {options}) - performs a collectionless aggregation on this database; returns a cursor");
    print("\tdb.auth(username, password)");
    print("\tdb.cloneDatabase(fromhost) - will only function with MongoDB 4.0 and below");
    print("\tdb.commandHelp(name) returns the help for the command");
    print(
        "\tdb.copyDatabase(fromdb, todb, fromhost) - will only function with MongoDB 4.0 and below");
    print("\tdb.createCollection(name, {size: ..., capped: ..., max: ...})");
    print("\tdb.createUser(userDocument)");
    print("\tdb.createView(name, viewOn, [{$operator: {...}}, ...], {viewOptions})");
    print("\tdb.currentOp() displays currently executing operations in the db");
    print("\tdb.dropDatabase(writeConcern)");
    print("\tdb.dropUser(username)");
```

Figure 1: Various ways of running the help, in order of increasing granularity

The MongoDB Shell

Although the mongo shell help command provides information for many functions, the user must search through all of these functions in order to find a certain one; there is no way to search and limit the help result. Appending help() provides a finer grain of usage information, and the mongo documentation [1] has extensive details with examples. The combination of these help commands and examples makes it easier to understand what a MongoDB query is doing and how to craft a similar query.

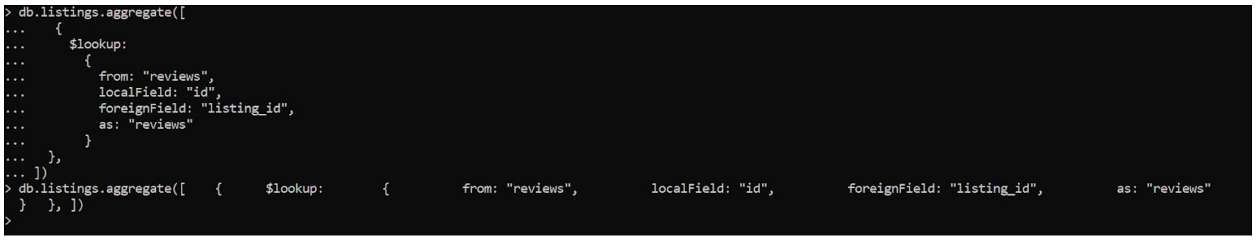
Not entirely unlike the Windows command prompt and Linux shell, the mongo shell supports tab completion and command history. The user can type a command and press tab to finish the command or show completion suggestions. Unfortunately, this feature is neither consistent, nor accurate; certain commands will not tab complete for no apparent reason, and tab completion does not show all possible completions. For example, given a database that has a collection named 'users', the following commands do not tab complete: *'db.users.find().'*, *'db.users.find().coun'*. The following commands suggest incorrect completions: *'show d', 'db.users.find()'*.



```
> show d
DBCommandCursor(        DataConsistencyChecker(  db                     defaultPrompt(         doassert(
DBExplainQuery(         Date(                    decodeURI(             defineProperties
DBPointer(              DriverSession(           decodeURIComponent(    defineProperty
> show db
2019-10-23T06:45:41.981-0700 E  QUERY    [js] Error: don't know how to show [db] :
shellHelper.show@src/mongo/shell/utils.js:1139:11
shellHelper@src/mongo/shell/utils.js:790:15
@(shellhelp2):1:1
> show dbs
admin   0.000GB
airbnb  0.092GB
config  0.000GB
iyves   0.000GB
local   0.000GB
>
```

Figure 2: Incorrect tab completion suggestions for the command 'show d'

Another feature shared between the command line and shell is a command history. Pressing the up and down arrow keys allows the user to navigate through recently submitted queries, which may be convenient for submitting similar queries in succession or re-submitting a complex query. However, the mongo shell replaces any new line characters with spaces, makes the query significantly harder to read and navigate. Also, the user cannot navigate backwards and re-enter the new lines -- the mongo shell will instead run the command. For these reasons, the best option is to craft queries in a word editor and then paste them into the shell.



```
> db.listings.aggregate([
...    {
...      $lookup:
...        {
...          from: "reviews",
...          localField: "id",
...          foreignField: "listing_id",
...          as: "reviews"
...        }
...    },
... ])
> db.listings.aggregate([    {      $lookup:        {          from: "reviews",          localField: "id",          foreignField: "listing_id",          as: "reviews"
  }   }, ])
>
```

Figure 3: The mongo shell history removing the formatting for multi-line queries in the command history

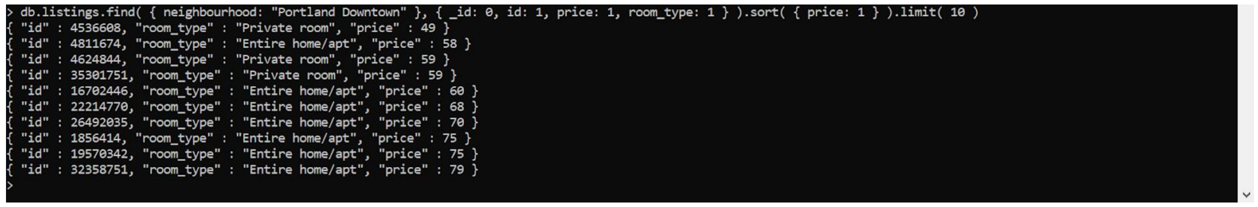
MongoDB Query Language

The MongoDB query language is quite distinct from other database query languages, traditional and cloud alike. It can best be described as a combination of method chaining, ECMAScript and JSON. Data flows from left to right, top to bottom and operations are specified in a JSON-like format, surrounded by curly brackets. For users familiar with these standards and formats, the mongo query language is easy to learn. For those not familiar, the mongoDB documentation [1] provides a lot of examples, and some examples include an embedded web-based console, through which the user may test simple commands. Additionally, the authors of the documentation included a SQL-to-mongo chart to understand how terminology changes between the two languages.

Writing database queries for a single collection is fairly simple. Using MongoDB CRUD operations [12], users can find an overview of basic collection manipulation queries, such as inserting, updating, removing, and querying documents. For example, for a collection, ‘listings’, that provides information about entry listings in the homestay app

Airbnb, the following query will find the listing id, price, and room type of the top ten cheapest places to stay in downtown Portland city:

```
db.listings.find( { neighbourhood: "Portland Downtown" }, { _id: 0, id: 1, price: 1,
room_type: 1 } ).sort( { price: 1 } ).limit( 10 )
```



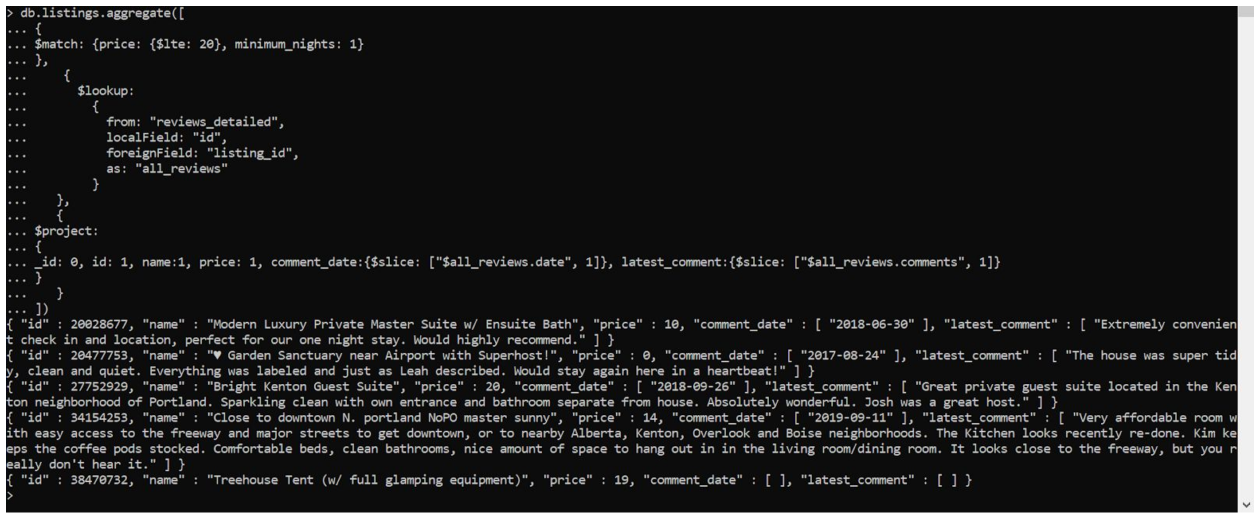
```
> db.listings.find( { neighbourhood: "Portland Downtown" }, { _id: 0, id: 1, price: 1, room_type: 1 } ).sort( { price: 1 } ).limit( 10 )
{ "id" : 4536608, "room_type" : "Private room", "price" : 49 }
{ "id" : 4811674, "room_type" : "Entire home/apt", "price" : 58 }
{ "id" : 4624844, "room_type" : "Private room", "price" : 59 }
{ "id" : 35301751, "room_type" : "Private room", "price" : 59 }
{ "id" : 16702446, "room_type" : "Entire home/apt", "price" : 60 }
{ "id" : 22214770, "room_type" : "Entire home/apt", "price" : 68 }
{ "id" : 26492035, "room_type" : "Entire home/apt", "price" : 70 }
{ "id" : 1856414, "room_type" : "Entire home/apt", "price" : 75 }
{ "id" : 19570342, "room_type" : "Entire home/apt", "price" : 75 }
{ "id" : 32358751, "room_type" : "Entire home/apt", "price" : 79 }
>
```

Figure 4: Result from running the above single-collection query above

The query follows an ordering of find({ WHERE }, { SELECT }).sort({ ORDERBY }).limit(LIMIT). This is slightly different from the traditional SQL ordering, but still flows logically. First, the user specifies the criteria for selecting documents, then the user chooses which fields to display of those documents, and can finally specify sort order and limit the amount of results returned to the screen.

Due to the flexibility of NoSQL systems, creating queries in MongoDB that span across multiple collections is noticeably more difficult compared to SQL. The following query will combine the ‘listings’ collection with a ‘reviews_detailed’ collection by using listing id as a foreign key, and will return the latest review for all homestay listings with prices up to $20 where the minimum amount of nights to stay is one night:

```
db.listings.aggregate([
   {
         $match: {price: {$lte: 20}, minimum_nights: 1}
   },
   {
        $lookup:
        {
                from: "reviews_detailed",
                localField: "id",
                foreignField: "listing_id",
                as: "all_reviews"
        }
  },
  {
         $project:
         {
                _id: 0, id: 1, name:1, price: 1, comment_date:{$slice:
["$all_reviews.date", 1]}, latest_comment:{$slice: ["$all_reviews.comments", 1]}
         }
  }
])
```



```
> db.listings.aggregate([
... {
... $match: {price: {$lte: 20}, minimum_nights: 1}
... },
...     {
...       $lookup:
...         {
...           from: "reviews_detailed",
...           localField: "id",
...           foreignField: "listing_id",
...           as: "all_reviews"
...         }
...    },
...    {
... $project:
... {
... _id: 0, id: 1, name:1, price: 1, comment_date:{$slice: ["$all_reviews.date", 1]}, latest_comment:{$slice: ["$all_reviews.comments", 1]}
... }
...    }
... ])
{ "id" : 20028677, "name" : "Modern Luxury Private Master Suite w/ Ensuite Bath", "price" : 10, "comment_date" : [ "2018-06-30" ], "latest_comment" : [ "Extremely convenien
t check in and location, perfect for our one night stay. Would highly recommend." ] }
{ "id" : 20477753, "name" : "♥ Garden Sanctuary near Airport with Superhost!", "price" : 0, "comment_date" : [ "2017-08-24" ], "latest_comment" : [ "The house was super tid
y, clean and quiet. Everything was labeled and just as Leah described. Would stay again here in a heartbeat!" ] }
{ "id" : 27752929, "name" : "Bright Kenton Guest Suite", "price" : 20, "comment_date" : [ "2018-09-26" ], "latest_comment" : [ "Great private guest suite located in the Ken
ton neighborhood of Portland. Sparkling clean with own entrance and bathroom separate from house. Absolutely wonderful. Josh was a great host." ] }
{ "id" : 34154253, "name" : "Close to downtown N. portland NoPO master sunny", "price" : 14, "comment_date" : [ "2019-09-11" ], "latest_comment" : [ "Very affordable room w
ith easy access to the freeway and major streets to get downtown, or to nearby Alberta, Kenton, Overlook and Boise neighborhoods. The Kitchen looks recently re-done. Kim ke
eps the coffee pods stocked. Comfortable beds, clean bathrooms, nice amount of space to hang out in in the living room/dining room. It looks close to the freeway, but you r
eally don't hear it." ] }
{ "id" : 38470732, "name" : "Treehouse Tent (w/ full glamping equipment)", "price" : 19, "comment_date" : [ ], "latest_comment" : [ ] }
>
```

Figure 5: Result from running the aforementioned multi-collection query

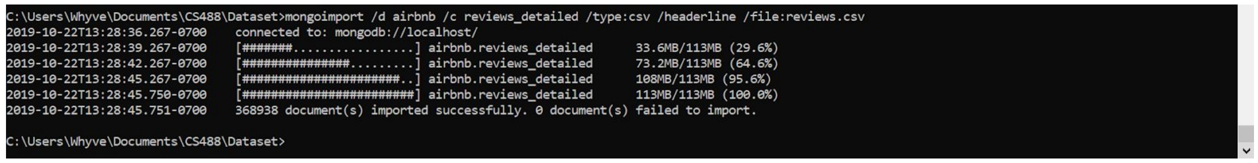
This example makes use of the *aggregate* method, which substantially differs from SQL joins and merges. The *aggregate* method uses a pipeline that can consist of stages, expressions, and accumulators [4]. Each section of the pipeline does one of these manipulations and passes the output to the next stage. In the example above, the pipeline consists of three steps:

1. **$match** - limits the documents returned from the '*listings*' collection
2. **$lookup** - left outer join the *'reviews_detailed'* collection with the *'listings'* collection on the 'listing_id' foreign key.
3. **$project** - SELECT fields and choose the first element ($slice) for two fields that consist of arrays

Although pipelines require a very different approach to query design when compared to SQL, the separation of the query into steps that flow from top to bottom fit logically well with user expectations of a pipeline. The abstraction and separation of query parts into steps simplifies large, complex queries. This makes the query easier to comprehend, maintain, and modify.

Nevertheless, there are some peculiarities that negatively impact the MongoDB shell user interface. When running a query in SQL, the column display order matches the order specified in the SELECT line, but in MongoDB, the column display order is always alphabetical unless the user crafts a workaround by cleverly changing the query. Another peculiarity is in the .distinct() method. In SQL, the DISTINCT keyword in the SELECT line will return distinct rows of a specified column. In MongoDB, however, the .distinct() method will apply the distinct method before doing the query. Instead, the user must place the distinct as a step in a pipeline in order to get the expected result. A final peculiarity is in the the .count() method. This method can be appended to the end of any collection or the .find() method to return a count of the documents that the query returns. However, this method cannot be appended to the end of an .aggregate() method; the count method must be added as a step in the pipeline. While these peculiarities do not largely impact the usability of the mongoDB shell, they are usage inconsistencies that break the expectations of the user.

Importing and exporting files to and from the mongo database is an awkward moment: there is no command within the shell to do export or import (although this isn't specific to the MongoDB shell). This requires exiting the mongo shell and running the command *'mongoimport'* or *'mongoexport'*. These commands are fortunately simple and intuitive to use, which makes the import and export process relatively painless.



```
C:\Users\Whyve\Documents\CS488\Dataset>mongoimport /d airbnb /c reviews_detailed /type:csv /headerline /file:reviews.csv
2019-10-22T13:28:36.267-0700    connected to: mongodb://localhost/
2019-10-22T13:28:39.267-0700    [#######.................] airbnb.reviews_detailed      33.6MB/113MB (29.6%)
2019-10-22T13:28:42.267-0700    [###############.........] airbnb.reviews_detailed      73.2MB/113MB (64.6%)
2019-10-22T13:28:45.267-0700    [######################..] airbnb.reviews_detailed      108MB/113MB (95.6%)
2019-10-22T13:28:45.750-0700    [########################] airbnb.reviews_detailed      113MB/113MB (100.0%)
2019-10-22T13:28:45.751-0700    368938 document(s) imported successfully. 0 document(s) failed to import.

C:\Users\Whyve\Documents\CS488\Dataset>
```

*Figure 6: Importing a 113MB .csv file to the 'reviews_detailed' collection in the '*airbnb*' database*

Overall, the user interface of mongoDB is good. The user must learn the unique query language of the MongoDB shell, but this is not too difficult thanks to its similarities to features of programming languages popular in the industry. Help commands and the MongoDB documentation is easy to read, well organized, and offers varying levels of granularity. The added feature of viewing a command's code implementation is very helpful for understanding underlying data connections. While there are some peculiarities that may surprise or frustrate users coming from a SQL background, the logical flow of the aggregate function pipeline simplifies complex queries.

Indexes

Despite being a NoSQL DBMS, MongoDB permits the use of indices on one field or subfields in a collection. Without indices, a particular scan would need to traverse all documents during query execution offering no alternatives for MongoDB to perform a more efficient search [13].

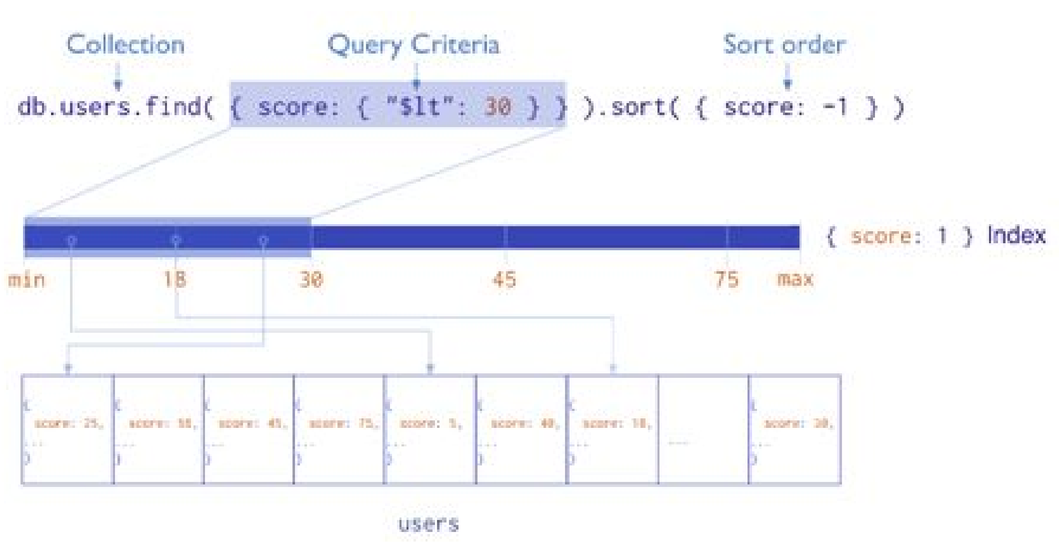
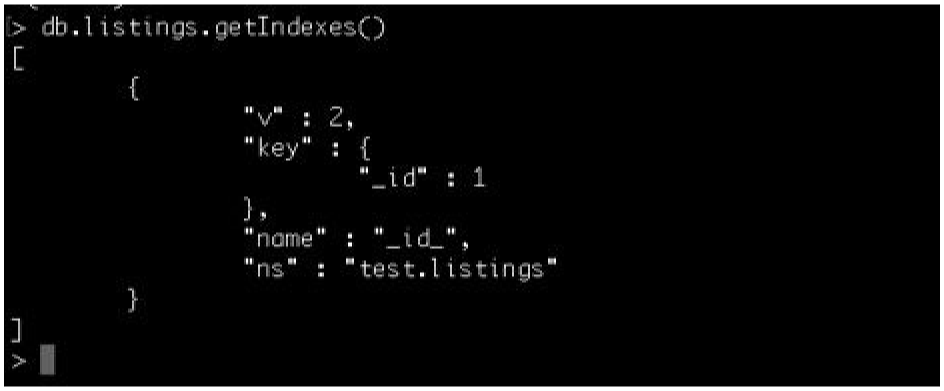


Figure 7: Index data structure allows query optimization [13]

An index orders tuples (or documents in this case), in the order of the field that is indexed (by means of an ordered data structure such as a B-Tree), which usually means that the creation of multiple indices duplicates some data in the collection. For this reason, developers aim to create indices only on fields commonly queried or ones that are filtered upon; range based/equality searches are the primary operations which benefit the most. Also, MongoDB assigns, by default, an index on the document ***_id*** field to ensure uniqueness: the primary key should uniquely identify a document, and creating an index on this field checks this enforcement in O(log n) [13]. Being the primary index, documents are physically present on disk in this order, making any object lookup using this key an extremely efficient operation. Maintaining this fundamental order means that this index cannot be dropped. Hence, no other field can serve as a primary index (a collection can only exist physically in one order).

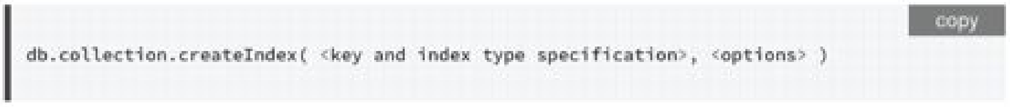
Using the Mongo shell below on the ***listings*** data from the Airbnb dataset, we can see that the collection specifies the ***_id*** as the default primary key sorted in ascending order.



```
> db.listings.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "test.listings"
        }
]
>
```

Figure 8: Default key

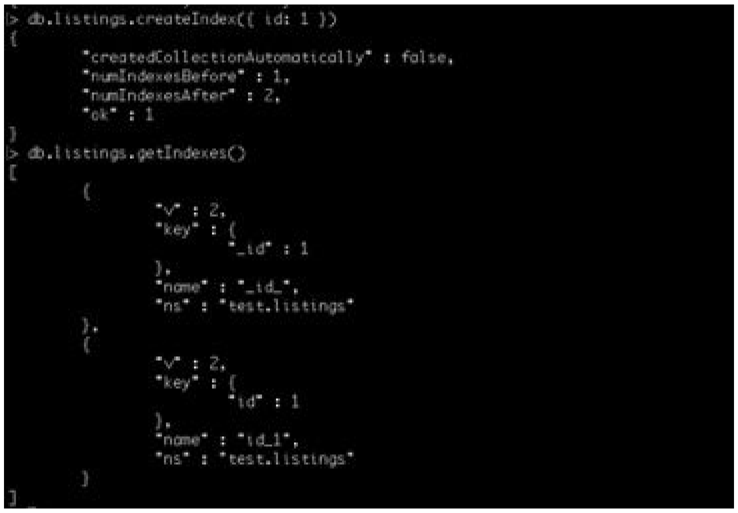
Adding a user-defined secondary index is even simpler:



```
db.collection.createIndex( <key and index type specification>, <options> )
```

Figure 9: Creating an Index [13]

The results for creating an index on the ***listings*** data for the ***id*** field is shown below. Even though this is a secondary index, specifying the primary key during the dataset import will preclude MongoDB from applying its default Object key field ***_id***.

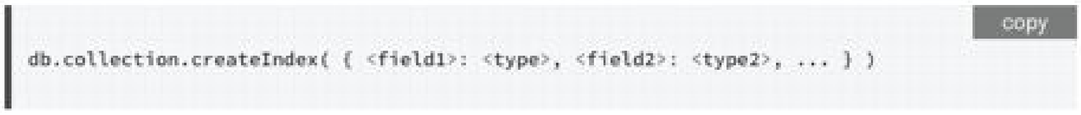


```
> db.listings.createIndex({ id: 1 })
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
> db.listings.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "test.listings"
        },
        {
                "v" : 2,
                "key" : {
                        "id" : 1
                },
                "name" : "id_1",
                "ns" : "test.listings"
        }
]
```

Figure 10: id index created in ascending order

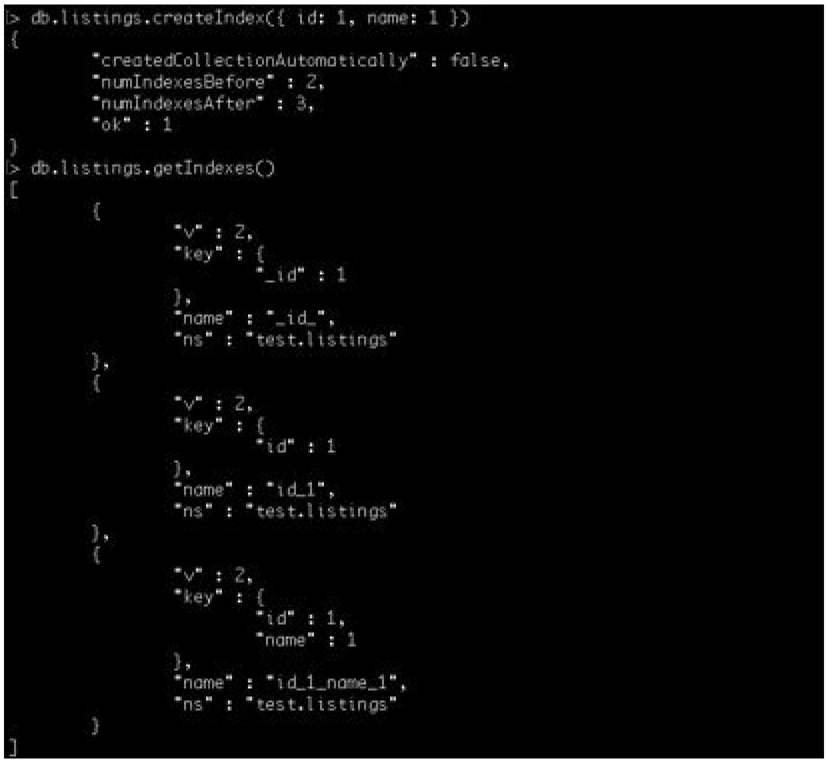
MongoDB supports a plethora of other index types -- single and multifield. With both these types, a user must specify a sort order: ***1*** for ascending and ***-1*** for descending. Using a ***sort()*** operation with single field indexes may use the index independent of the sort order (traversal is possible in either order). However, in compound indexes, the sort() operation uses the index only for the first field in the compound index- documents are ordered first by the sort order of the first key and then the sort order specified by the second.

Creating multi field indexes is a natural extension of the first:



```
db.collection.createIndex( { <field1>: <type>, <field2>: <type2>, ... } )
```

Figure 11: Multi Field Indexes [13]



```
> db.listings.createIndex({ id: 1, name: 1 })
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 2,
        "numIndexesAfter" : 3,
        "ok" : 1
}
> db.listings.getIndexes()
[
        {
                "v" : 2,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "test.listings"
        },
        {
                "v" : 2,
                "key" : {
                        "id" : 1
                },
                "name" : "id_1",
                "ns" : "test.listings"
        },
        {
                "v" : 2,
                "key" : {
                        "id" : 1,
                        "name" : 1
                },
                "name" : "id_1_name_1",
                "ns" : "test.listings"
        }
]
```

Figure 12: Order by id, then by name

Compound indexing is useful in the situation where ***<field1>*** is dependent on ***<field2>*** in the sense that common query calculations refer to both, order dependent, in range and equality matches. Once the first index narrows down results logarithmically from the first field, a self-similar step can be applied on all tuples (documents) matching the second criterion.

Another notable property of indexes in MongoDB is the covering index. This assures that if a certain query only applies fields that have already been indexed, results will be procured directly from disk without needing to consult local memory.

Further, MongoDB allows indexing in a few more ways [13]:

- Multikey Index
  - If the field that is indexed is of type ***array,*** MongoDB creates an index for every element of the array. This allows for a user to perform efficient searches on particular elements of an array that may themselves be BSON objects.

- Geospatial Index
  - N-dimensional coordinate fields may need to be sorted in various forms for quick spatial searches. For example, an ordering that keeps tuples with coordinates close to each other would result in faster zip code lookups.

- Text Index
  - Indexing based on fields that store strings that are ***root*** words (not “a”, “the”, “or” etc.) allow ordering based on the word that is to be indexed.

- Hashed Index
  - In the case of a sharded cluster set, indexing by the hashed value of a certain field results in pseudo-random, similarly sized buckets (assuming an adequate hash function); the tuples for which can be assigned to separate shards. Parallelization of tasks are easier in this configuration as equality-based searches can commence on multiple shards at a time. However, range searches cannot be performed in this case.

MongoDB is unusual as a NoSQL database that is also a proponent of ***ACID*** properties much like traditional RDBMs. The ability to apply various indexing methods for specific use cases allows for quick, reliable results for a variety of data types and configurations.

Consistency

Consistency is the ability of a database to ensure that transactions comply with specific sets of rules or guarantees. In terms of databases, consistency will relate to the properties of transactions. These specific properties are encapsulated in the abbreviation 'ACID' (atomicity, consistency, isolation, and durability). MongoDB allows for atomic transactions, as each operation can be wrapped in a WiredTiger transaction. WiredTiger is the default storage engine for MongoDB which uses "document-level concurrency control for write operations" [6]. WriteTiger will maintain locks and intent locks, and when a write conflict occurs, the conflict will result in a retry of the failed operation. Since version 3.2, WiredTiger has been incorporated into MongoDB and this allows for operations on a single document to be atomic. MongoDB now allows for atomicity of reads and writes for multiple documents starting in version 4.2. Atomicity will enable visibility outside of a transaction when a transaction commits [6]. While ACID transactions are fully supported, a user doesn't need to necessarily use transactions for single document writes since they are already atomic.

MonogoDB can be setup to use standalone *mongod* (i.e. the primary daemon process for the database server), replica sets, or sharded clusters. To promote high availability, MongoDB allows for the creation of replica sets which are a "consensus group, where each node maintains a logical copy of the database state" [7]. The replica set provides a single primary node and then a set of secondary nodes. Writes filter through the primary node of the replica set, and this primary node is the only node capable of confirming the success or failure of writes to the replica set, depending on the type of write chosen this could be only to the primary or to both the primary and secondary nodes.

Along with transactions being supported, MongoDB has variable levels of consistency that it offers. As denoted by the CAP theorem, there is a need to decide between either consistency or availability when designing a database, especially with cloud systems. MongoDB allows deviation from traditional database consistency guarantees with a one size fits all design. Application developers and database admins can then modify certain tunable features to better fit specific use cases. The main tunable features are included in the following: Read Preference, Readconcern, and Writeconcern. For multi-document transactions, the Writeconcern is set at the transaction level.

At a high level, the difference between Read Preference and Readconcern is that Read Preference outlines where the read happens from and Readconcern outlines what is being read. Readconcern is a tunable feature that controls the consistency and isolation of data reads [8]. A couple notable options for this feature include “local” and “majority”. “Local” is chosen by default, and will return data from the replica set, but there are no guarantees that the data being read has been written to the majority of the replica set at that moment in time. “Majority” returns read data that has been acknowledged by the majority of the replica set members. In contrast to Readconcern, Read Preference outlines where data is read from and will read from the primary node by default [5]. Another option available for Read Preference is to read from the nearest node, possibly a secondary, thus reducing latency. However, reading from anything but the primary when majority of nodes have not reached a consensus introduces the possibility of returning stale data.

Writeconcern is another tunable feature that MongoDB provides. The value chosen here can either be numeric or a defined set of string values with various meanings. The chosen value will represent the number of data bearing nodes that must acknowledge a write before it is “successful” [9]. The default value is 1, which requires only the primary replica set node to acknowledge the write before returning success to the client. Any value over 1 indicates the number of other data-bearing nodes that must acknowledge the write as well, these will be secondary nodes. Writeconcern also has the option to require a “majority” value, which means that the write operation must have propagated to the majority of replica set members before returning success.

Write Concern and Read Concern Example

```
db.inventory.insert(
        { sku: "03902390", qty : 100, category: "Shirt" },
        { writeConcern: { w: 2, j: true, wtimeout: 5000 } }
)
```

Above is an example of a write concern with a number of nodes chosen greater than one [9]. The wtimeout is the millisecond value to wait for operation completion before throwing an error. Here, we see a Writeconcern parameterized such that two members of the replica set have to acknowledge the write within 5000 milliseconds or throw an error. The j option specifies that we request the write operation to be written to the on-disk journal, discussed following the next example.

```
db.restaurants.find( { _id: 5}).readConcern("linearizable”).maxTimeMS(10000)
```

Above is an example of using a ReadConcern with the “linearizable” setting, which will confirm with secondary replica set members that the read operation is coming from a primary node [8]. When a “linearizable” read is used following a “majority” write, then that operation is able to read its own writes.

MongoDB implements journaling and oplogs to help maintain checkpoints and transaction details. Journaling is written ahead to on-disk journal files. These journal files are maintained for each client write operation, which can contain both index modification details as well as data modification details [10]. MongoDB also uses an oplog, which will serialize all write operations that come into the system. The oplog is used for replica sets, and keeps records of any operations on the primary's oplog [11]. Secondary replica set members will copy and apply the operations asynchronously. A key feature of the oplog is that each operation is idempotent. Oplogs may be used in crash recovery to retry the transactions, and may be used to rollback in the event of a failure.

MongoDB offers consistency on a tunable scale and also atomic transactions for both single and multi-document operations. Choosing Writeconcern and Readconcern values that require more nodes to respond and agree before completion of the operation leads to a higher guarantee of consistency. With everything taken into consideration, MongoDB states that it guarantees strong consistency when used with the default values. These default values use the primary node for both write and read operations. Any choices to read or write to nodes other than the primary can lead to eventual or casual consistency. However, as suggested by the CAP theorem, these consistency requirements will negatively impact availability, as there is often a tradeoff between consistency and availability. Having strict consistency requirements may not be necessary for all applications, but requiring both a “majority” for Readconcern and Writeconcern will promote consistency and avoid stale data from returned to the client.

Scalability and Replication

In terms of scaling, there are two terms that get used: *horizontal* and *vertical scaling*. Each of these terms are solutions to the problem of a growing database, but they solve the problem in different ways. MongoDB explicitly moves away from vertical scaling in favor of utilizing more servers for both horizontal scaling and replication. With regards to the CAP theorem, MongoDB prioritizes tolerance of network partition and consistency over availability. The latter is not always an acceptable compromise for certain applications, and is somewhat unusual for a NoSQL database. To accomplish this, MongoDB considers many edge cases and scenarios, such as dealing with network partitioning in the event of an outage between different locations. Sharding, for example, has already been implemented and allows for a very flexible server cluster setup.

Scalability in MongoDB is very user and developer friendly; it has built-in support for sharding with many commands that help create shards in a quick and easy fashion. The database is configured with a config server, routers, and replica sets or shards as shown in the figure below. The client connects to a router, which interacts with both the config servers and the shards. To avoid a bottleneck, the client does not interact with the config servers and, instead, interacts with the routers.

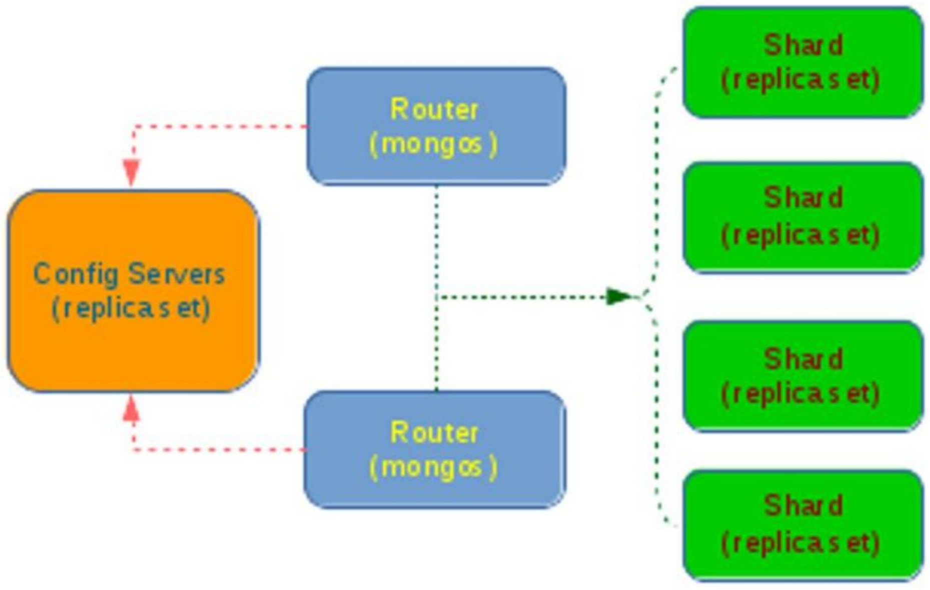


Figure 13: visualizing the cluster system [3]

As seen in Figure 13, the config servers are isolated from the shards through the routers. The config servers hold the metadata and settings/configuration for the cluster setup and do not directly interact with clients. The clients interact with the routers, which in turn query the cluster system. This system scales compute and storage resources easily by adding shards and routers into the cluster as needed, flexibly adjusting to various use cases.

When it comes to data replication, availability, and fault tolerance, MongoDB once again has built in functionality for these problems as well as concepts for some potential edge cases that may arise when deciding on what node is called a primary. The system has a concept of primary and secondary servers. The primary, as the name suggests, is the main cluster that is used and has all the data manipulation and queries run against it. But in the event that the primary becomes unavailable, the remaining servers hold a poll to select which secondary gets promoted as the new primary by majority vote. However, in the event that there are an even number of duplicates, there can be another server called an arbiter with the sole purpose of guaranteeing a majority vote for creating the new primary server. The arbiter cannot be promoted to a primary or secondary server, as it does not store data like the other servers. However, the arbiter may not need dedicated hardware for its purpose. In fact “an arbiter does not store a copy of the data and requires fewer resources. As a result, you may run an arbiter on an application server or other shared process” [1].

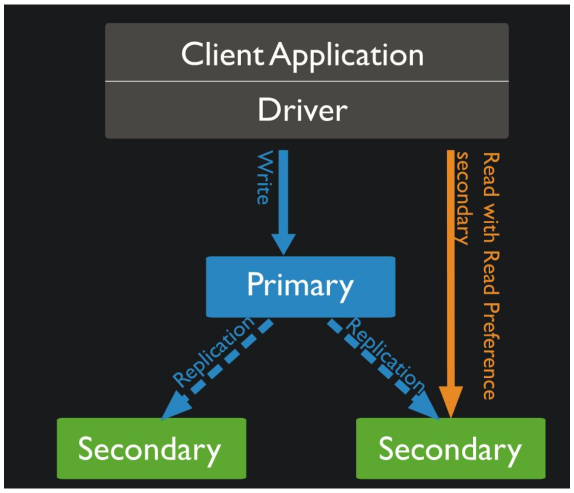
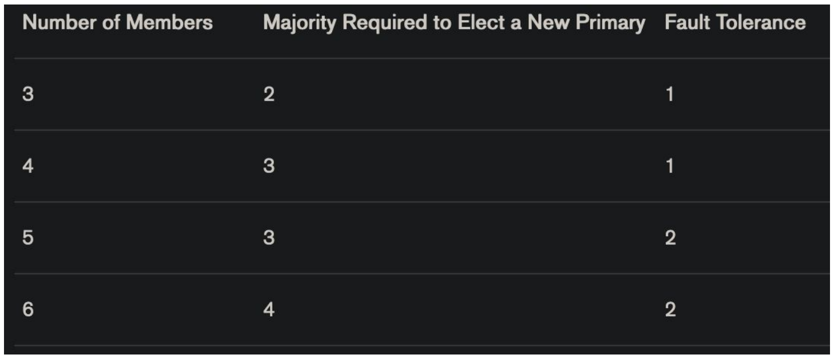


Figure 14: Showing how a user interacts with the cluster systems with replication [1]

As shown in Figure 14, the client application will normally write and read to the primary set of servers for the data. However, the client application may explicitly request to view from one of the secondary servers if desired. This may be used for a variety of reasons, each of which have their uses. For example, maybe the client application would like to initiate a timeout for the primary cluster, still service reads, and deal with electing a new primary for the system. The chart below in Figure 15 provides an example of the required number of replicas needed in order to elect a new primary in the event that a system goes down. When there are five members in total, MongoDB allows an increased fault tolerance of two machines and a majority of three for the new secondary to become the new primary.



Number of Members	Majority Required to Elect a New Primary	Fault Tolerance
3	2	1
4	3	1
5	3	2
6	4	2

Figure 15: Fault Tolerance for MongoDB clusters [1]

Setting Up Sharding and Replication Example

Following is a great example provided from Eugen Hoble, an experienced Java and J2EE programmer, in one of his posts [3] demonstrating the steps for setting up a MongoDB cluster with sharding:

1. Set up the config server
   ```
   mongod --configsvr --dbpath /config-server --port 8080
   ```
2. Set up the router servers
   ```
   mongos --configdb configserver1.myhost.com:8080
   ```
3. Start a shard server
   ```
   mongod --shardsvr
   ```
   -or if the shard is going to be part of a replica (replication)
   ```
   mongod --shardsvr --replSet replica1
   ```
4. Connect to router and add shard
   ```
   mongo --host router1.myhost.com --port 8080
   sh.addShard( "shard1.myhost.com:8080" )
   ```
5. Enable sharding
   ```
   mongo --host router1.example.com --port 8080
   sh.enableSharding("my_database")
   ```
6. Enable sharding on collections
   ```
   use my_database
   db.my_collection.ensureIndex( { _id : "hashed" } )
   ```

```
sh.shardCollection("my_database.my_collection", { "_id": "hashed" } )
```

Conclusion

MongoDB is a very appealing NoSQL alternative to traditional relational database systems. Flexible schemas allow for storing non-flat records or data with variable fields. Along with flexible schemas, there are a multitude of index options available to increase performance. In our research, we discovered that this database system takes a very unusual approach of prioritizing consistency over availability, which seems counterintuitive, considering the advantages of relaxed consistency that clusters offer. Nevertheless, tunable features allow users to configure their system per use case. Although interaction with the mongo database requires learning a new query language, the readability of official documentation and simplicity of database installation greatly reduces this overhead. This ease of user interface along with our implementation of the various possible data models should allow for a smoother start to our project using the AirBnB dataset.

References

[1] MongoDB, Inc., “The MongoDB 4.2 Manual,” *mongoDB.com*, Aug. 13, 2019. [Online]. Available: https://docs.mongodb.com/manual/. [Accessed: Oct. 17, 2019].

[2] R. Cattell, “Scalable SQL and NoSQL data Stores,” *ACM SIGMOD Record*, vol 39, no. 4, pp. 12-27, Dec. 2010. [Online]. Available: https://dl.acm.org/citation.cfm?id=1978919. [Accessed: Oct. 3, 2019].

[3] E. Hoble, “Divide and Conquer: High Scalability With MongoDB Sharding,” Divide and Conquer: High Scalability With MongoDB Sharding, Nov. 1, 2016. [Online]. Available: https://dzone.com/articles/divide-and-conquer-high-scalability-with-mongodb-t. [Accessed: Oct. 22, 2019].

[4] J. Kelly, “Mongo Aggregations in 5 Minutes,” *Universe Engineering*, para. 6, May 30, 2017. [Online]. Available: https://engineering.universe.com/mongo-aggregations-in-5-minutes-b8e1d9c274bb. [Accessed: Oct. 22, 2019].

[5] MongoDB, Inc. “Read Preference,” *mongoDB.com,* Aug. 13, 2019. [Online]. Available: https://docs.mongodb.com/manual/core/read-preference/. [Accessed: Oct. 23, 2019].

[6] MongoDB, Inc. “Transactions,” *mongoDB.com,* Aug. 13, 2019. [Online]. Available: https://docs.mongodb.com/manual/core/transactions/. [Accessed: Oct. 23, 2019].

[7] W. Schultz, T. Avitabile, and A. Cabral, “Tunable consistency in MongoDB,” Proceedings of the VLDB Endowment, vol. 12, no. 12, pp. 2071–2081, Jan. 2019.

[8] MongoDB, Inc. “Read Concern,” *mongoDB.com,* Aug. 13, 2019. [Online]. Available: https://docs.mongodb.com/manual/reference/read-concern/. [Accessed: Oct. 23, 2019].

[9] MongoDB, Inc. “Write Concern for Replica Sets,” *mongoDB.com,* Aug. 13, 2019. [Online]. Available: https://docs.mongodb.com/manual/core/replica-set-write-concern/. [Accessed: Oct. 23, 2019].

[10] MongoDB, Inc. “Journaling,” *mongoDB.com*, Aug. 13, 2019. [Online]. Available: https://docs.mongodb.com/manual/core/journaling. [Accessed: Oct. 23, 2019].

[11] MongoDB, Inc. “Replica Set Oplog,” *mongoDB.com,* Aug. 13, 2019. [Online]. Available: https://docs.mongodb.com/manual/core/replica-set-oplog/. [Accessed: Oct. 23, 2019].

[12] MongoDB, Inc. “MongoDB CRUD Operations,” *mongoDB.com*, Aug. 13, 2019. [Online]. Available: https://docs.mongodb.com/manual/crud/. [Accessed: Oct. 23, 2019].

[13] MongoDB, Inc. “Indexes,” *mongoDB.com*, Aug. 13, 2019. [Online]. Available: https://docs.mongodb.com/manual/indexes/. [Accessed: Oct. 24, 2019].