

100/100

Great report! A bit long,  
but it was well written  
and easy to read.

Please see my  
comments, which are  
mostly suggestions for  
future writing and your  
current experiment,

# MongoDB

## Part 2: Data Models & Queries

Bingo Bango Mongo

*Michelle Duer*

*Ian Percy*

*Billy Haugen*

*Yves Wienecke*

*Tejas Menon*

CS488/588 Cloud & Cluster Data Management

## Table of Contents

Data Sets	2
Data Model	3
Storage	6
Limitations and possible workarounds	8
MongoDB C# Driver	10
Atlas Import	11
Queries	13
Query 1: Count query	13
Query 2: Sorted Subset	13
Query 3: Subset-search	14
Query 4: Average	16
Query 5: "Join"	16
Query 6: Updating	17
References	19

Designing a data model for a noSQL database is fundamentally different from designing a traditional relational database data model. RDBs require a set schema, normalization, and careful planning to ensure consistency throughout the database. These models typically consist of a variety of tables that relate to each other and can be joined to produce an encompassing view of the data. On the other hand, noSQL databases allow for a flexible schema and often favor availability over consistency. In document store databases such as mongoDB, data is more often stored in a single file, as joining files for a query drastically reduces performance. When crafting a design for such a database, some key considerations include the following: query performance, scalability, and partitions for sharding. In this report, we will discuss our dataset, the data model we designed to fit this dataset, and queries that we plan to run against our completed mongoDB database.

## Data Sets

Any good database model design requires sufficient understanding of the data that the model is trying to fit. Our group chose to proceed with information scraped by Murray Cox from the popular website for temporary housing accommodations, Airbnb [1]. The data is partitioned by city and includes six comma-separated files and one geojson file. This makes it easier to adapt the magnitude of information we process to our project. Three of the .csv files are detailed information about listings, reviews, and calendar data. Two of the .csv files are summarized versions of listings and reviews. The last .csv file lists the neighborhoods in each city, which goes along with the .geojson file containing the geospatial description of the neighborhoods (used for cleaning neighborhood names). With this information, we hope to gain a data-informed insight on the types of housing available for rent on this website, along with the reviews for these listings. For our project, we will focus on data from the detailed listings and reviews. The summary, calendar, and neighborhood files do not provide additional information

relevant for our project. These files are not a reflection on how Airbnb actually stores their data, but rather how Cox gathered and represented the data scraped from the original website.

We leveraged the *urllib.request* library in python in order to scrape the Airbnb data page and request each of the detailed .csv files. These files are in a compressed format of about 5 GB. When uncompressed, these files expand to over 22 MB for listings and 113 MB for reviews. The listing file has 106 columns, with each column fitting into one of the following groupings: listing metadata; housing details such as size, amount of people the space accommodates, address, and a brief description of the space; accommodation details like price, minimum and maximum days to stay, and house rules; host information and metadata; and listing notes, including notes about transportation. The reviews file connects to the listings file via the `listing_id` column, and includes information about the date of a review, the name of the reviewer, and the review itself. While the latter file is flat and obeys a format not unusual for an RDB, the former file breaks the first rule of normalization with a few columns that have embedded documents, for example `host_verifications`, `amenities`, and `jurisdiction names`.

## Data Model

Considering Murray Cox's presentation of Airbnb data, our group decided upon maintaining a flat model of the data. This flat file format works well with mongoDB, because most of the information is stored in one file rather than multiple files, which reduces the amount of joins needed to complete many listing queries. MongoDB excels with embedded documents, which functions like a partially-merged dataset, but the 15 MB document (row) limit discourages large embedded documents. This rules out considerations of embedding reviews in the listing file. First of all, reviews may grow unbounded for a listing and can be large chunks of text, and this negatively impacts

read performance for querying reviews and can push to the 15 MB limit. The reviews file is significantly larger than the listing file. Second, this greatly reduces performance for querying based on reviewer or review date, because it would require reading through each listing row and then unpacking each review for each row or creating a multikey index, instead of taking advantage of indexes on a separate reviews table. While we could embed review ids and remove listing id from the review file, this would negatively impact update performance whenever a review or listing is removed. Listings are tied to a single host, so embedding host information would add complexity without adding any advantages.

The data fits with a noSQL storage model due to the variation in listing details. For instance, some listings just have the basic details - price, minimum stay, address, etc. - but a significant amount of listings may or may not have additional information, such as notes, weekly\_price, or Airbnb license. MongoDB's flexible schema suits this attribute uncertainty well. We could embed amenities, host\_verification, and jurisdiction names, as they include multiple elements within one listing (ex. Amenities may expand to include internet, television, bedding, etc.). However, these columns are not relevant to our queries and thus provide more complexity without adding benefit to our project.

Utilizing a simple, flat data model will streamline the process of importing data into mongoDB Atlas clusters. After downloading the detailed listings and reviews files, we used a python script to import the data. This was the simplest and most flexible method for importing thanks to libraries such as pandas, which infers and processes data types from the files and avails transformation functionality down the line if more pre-processing is performed on the data. The current script utilizes a connection string exposed by the database, and then add new records from the data files into the Atlas cluster, which we can configure to host on either AWS or GCP.

cool!

Our group will put an index on the zipcode column of the listings table, and another index on the listing\_id column of the review table. Considering real world use cases for Airbnb users looking to travel, they are likely to prioritize location filtering or searching by geolocation. Having an index on the zipcode allows for range indexes based on city and/or state, which may filter MBs of data from other locations. Because reviews are most often queried in the context of a certain listing, having an index on this column may greatly boost query performance as well.

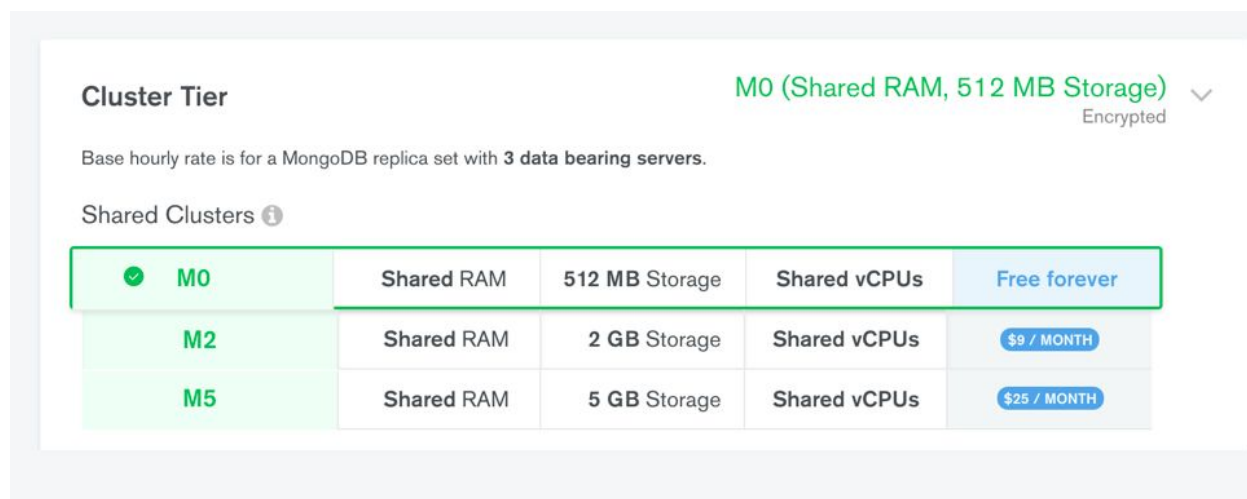
While the data is imported in a flat format, this does not mean that it will be less performant when queried. The amount of columns in a table does not significantly impact database performance [2]. This may be due to the BSON format in which MongoDB stores data, which is designed for better traversability by storing extra information about data types, string lengths, and embedded document lengths [6]. The queries may, for the size of the data, perform in some capacity that would be reasonable enough for the majority of Airbnb use cases.

Our group would like to explore the performance implications for the entire dataset and simplicity of embedding documents with similar information -- for instance, host information or listing information -- and multikey indexes on these documents versus indexes on the expanded fields of the flat model. Some of the alternative data models that we considered address this area. For instance, embedding geographical information, such as zipcode, city, and/or state, may simplify reads as these columns are often grouped together. A user would want to look at a specific city, cities nearby or proximity to certain attractions. This embedding may be extended to consider use cases where frequently accessed data is in a top-level document, while data that is rarely accessed is stored in a lower-level, embedded document. As mentioned earlier, this

could show some performance improvements when querying by a zip code field for location.

## Storage

MongoDB Atlas is a cloud database solution for MongoDB deployment on a user dependent platform – GCP, AWS or Azure– for which preset server-architecture configurations can be automatically set-up and configured. In addition to leveraging common use-cases with commercial deployment, it offers easier development and set-up workflows, along with more reliable, although less tunable, behavior and running costs. It completely automates the set-up process via a database interfacing web client. The user can choose between a region (or multi-region in which case free testing is unavailable) and select a cluster configuration – free tier provides a 3-node data-bearing replica set, with a default replication factor of three and disabled data sharding due to low server count and single region deployment.



The screenshot shows the 'Cluster Tier' selection interface in MongoDB Atlas. At the top, 'M0 (Shared RAM, 512 MB Storage)' is selected, with a dropdown arrow and 'Encrypted' text. Below this, a note states: 'Base hourly rate is for a MongoDB replica set with 3 data bearing servers.' Under the 'Shared Clusters' heading, a table lists three tiers: M0, M2, and M5. The M0 tier is highlighted with a green border and a checkmark, indicating it is the selected free tier. The M2 and M5 tiers are also highlighted with green borders. The table columns are: Tier, Configuration, Storage, vCPUs, and Price.

Cluster Tier	Configuration	Storage	vCPUs	Price
<input checked="" type="checkbox"/> M0	Shared RAM	512 MB Storage	Shared vCPUs	Free forever
<input type="checkbox"/> M2	Shared RAM	2 GB Storage	Shared vCPUs	\$9 / MONTH
<input type="checkbox"/> M5	Shared RAM	5 GB Storage	Shared vCPUs	\$25 / MONTH

Figure 1 Free tier cluster is limited to 512 MB storage and given replica-set [3]

<b>Cluster Tier</b>	You must select the <b>M0</b> cluster tier to deploy a Free Tier cluster.
<b>Cluster Memory</b>	You cannot configure memory for <b>M0</b> Free Tier or <b>M2/M5</b> shared clusters.
<b>Cluster Storage</b>	You cannot configure storage size for <b>M0</b> Free Tier or <b>M2/M5</b> shared clusters.
<b>Replication Factor</b>	Replication Factor is set to <b>3 Nodes</b> and cannot be modified for <b>M0</b> Free Tier, or <b>M2/M5</b> shared clusters.
<b>Replica Set Tags</b>	<b>M0</b> Free Tier, and <b>M2/M5</b> shared clusters are not configured with pre-defined <a href="#">replica set tags</a> .
<b>Do You Want A Sharded Cluster</b>	You cannot deploy a <b>M0</b> Free Tier or <b>M2/M5</b> shared cluster as a Sharded Cluster.
<b>Do You Want To Enable Backup</b>	You cannot enable backups on <b>M0</b> Free Tier clusters.

*Figure 2 Free-tier (or low tier) restrictions [4]*

So interesting if you can do the comparison.

Given these restrictions however, the main benefit of Atlas is its abstraction of tedious server set-up and hidden platform specific activities. Additionally, our ideal use-case in our project is comparing AWS vs GCP performance running MongoDB, therefore, removing ourselves from server specifics whilst ensuring identical server configuration allows for a fair comparison between each platform without having to delve into details given our free-credit limitations and project purpose.

By default, Atlas is configured on AWS and GCP to implement a similar cluster to the one shown in Figure 3, however, specific networking settings relating to VPC are unmodifiable and inaccessible from the Atlas web client as those are handled by the cloud providers. Atlas provides standard data redundancy mechanisms and low-to-average bandwidth and memory usage for adequate comparison between the two providers on low-end hardware e.g. t2-micro instances in AWS or f1-micro in GCP.

Just a feedback here and a suggestion.  
these machines do not have the same specifications! fi-micro < t2-micro.  
If you cannot select machine types I think it is OK to make the experiment as this is a class project, but I would clarify the difference in the hardware used in each platform. Who knows, maybe the weaker machine outperforms the stronger one!  
Also, make sure you know and report on the disk specifications as disk speed is a highly influential factor in query performance. I suggest you change the comparison to be between the free tier in GCD and AWS (your findings can be really useful for other students/learners who can only use the free tier).



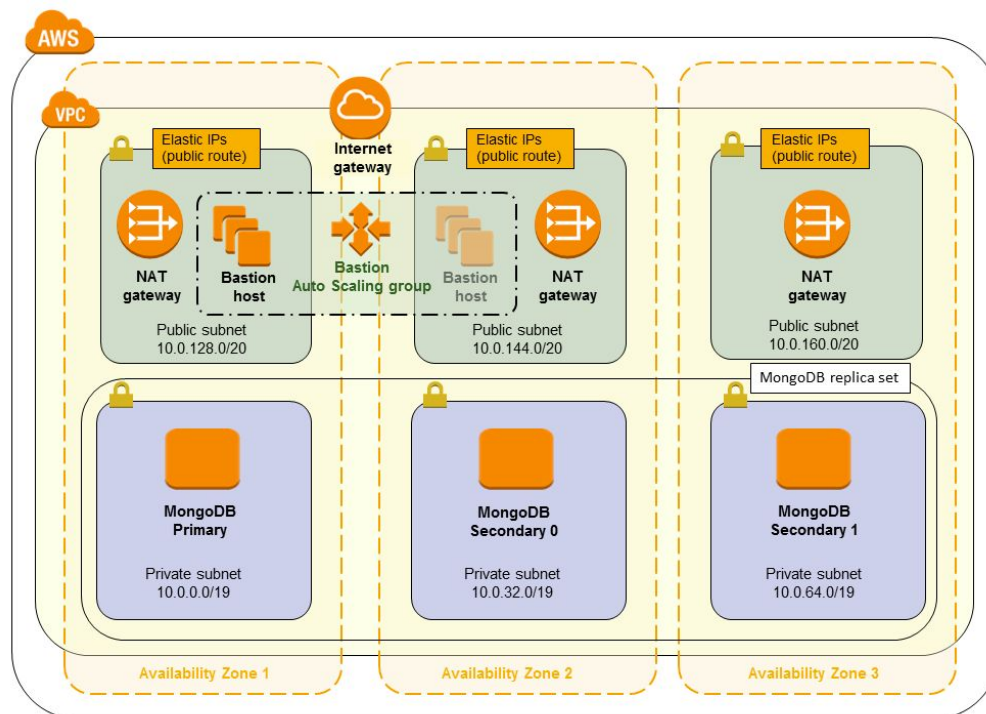


Figure 3 AWS Quick-start architecture with MongoDB [5]

## Limitations and possible workarounds

Certain limitations in the low and free tier options for Atlas might restrict usage of all the Airbnb data mainly due to query performance, storage restrictions and sharding capabilities.

Limitation	Tentative Workaround
Disabled sharding could result in slower query performance and lower throughput	Upgrade instance for a limited time period using free credit and manually split collections into instances based on zip_code field.



<p>Flat-file import structure could result in slower performance for join queries on large collections (example: Query 5 that involves a join between listing and reviews, discussed later on)</p>	<ul style="list-style-type: none"><li>· Purge total data size (to within free tier limitations), and restrict query coverage to fields in a single file</li><li>· Use covered queries based on indices</li><li>· Join in C# post querying flat-files in MongoDB. This will reduce overall server-load and hand-over time consuming computation to the client. This however has the drawback of complex data management in C# with possible sub-optimal algorithms. Also, comparison between the two providers will also depend on the C# execution time which might affect results.</li></ul>
<p>Comparison b/w GCP and AWS</p> <p>Queries that run very quickly have the problem of unreliable/anomalous execution times which would make fair judgment of each platform difficult. The delay may occur due to a temporary slow connection/busy servers (since memory and computation is</p>	<ul style="list-style-type: none"><li>· Use same region for each cloud provider</li><li>· Complex queries (design may be sub-optimal) should be tested for the sake of measuring performance. If these execute in minutes rather than a few seconds, any inherent delays in the server configuration are more likely to be represented by the time taken.</li></ul>

shared on the free tier) or be dependent on region.

Another suggestion for this or future reports, run experiments few times and compute averages if possible

## MongoDB C# Driver

Interacting with the database over C# is provided via a connection string generated by Atlas upon provisioning a server cluster.

Connect to TestCluster1

✓ Setup connection security ✓ Choose a connection method Connect

1 Choose your driver version

DRIVER VERSION

Node.js 3.0 or later

2 Add your connection string into your application code

Connection String Only Full Driver Example

mongodb+srv://brad:<password>@testcluster1-cbml1.mongodb.net/?retryWrites=true&appName=test

Copy

Replace <password> with the password for the brad user.  
When entering your password, make sure that any special characters are URL encoded.

Having trouble connecting? [View our troubleshooting documentation](#)

Figure 4 Connection String in Atlas [8]

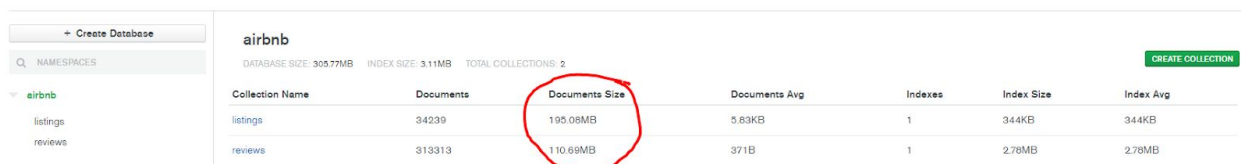
In Visual Studio, three packages are required for the driver: **MongoDB.Bson**, **MongoDB.Driver.Core**, and **MongoDB.Driver**. Once these are installed and linked

(they have dependencies on one another), a client object **MongoClient** can be generated and the connection string passed to its constructor to connect to a specified database. The data was already imported via a python script described earlier, so the C# serves strictly as a driver for running queries such as CRUD operations. We also observed that attempting to import the data through C# was fairly restrictive and time-consuming, so a python script was ideal and optimal for a rarely run operation. This was performed through **MongoClient.GetCollection()** and **IMongoDatabase.RunCommand()** functions, the first to get a collection from a database object and the second to execute queries on it.

Yet another suggestion for future reports that may save you some time! I think the highlighted text can be skipped. However, reporting on your experience of importing data using C# and python can be interesting/informative to the reader.

## Atlas Import

This data has been loaded into a cluster with one database name “airbnb” with two collections, “listings” and “reviews”. Here is the summary showing total data loaded in the database:



Collection Name	Documents	Documents Size	Documents Avg	Indexes	Index Size	Index Avg
listings	34239	195.08MB	5.63KB	1	344KB	344KB
reviews	313313	110.69MB	371B	1	2.78MB	2.78MB

First we show the data in the listings collection. For this we use a sample aggregation query in the Atlas Aggregation GUI for finding all listings with greater than 2 bedrooms (limited down to a set number of attributes for ease of reading):

The screenshot shows the Atlas Aggregation GUI with two stages: **\$project** and **\$match**.

**\$project stage:** The query is `{ bedrooms:1, id:1, zipcode:1 }` (marked with a red '1'). The output shows three documents with fields `_id`, `id`, `zipcode`, and `bedrooms`.

**\$match stage:** The query is `{ bedrooms:{$gt:2} }` (marked with a red '2'). The output shows three documents with fields `_id`, `id`, `zipcode`, and `bedrooms`.

The first step is a `$project` where we choose the columns we wish to output. The second step outlines to only output when the number of bedrooms is greater than 2 as we see from the documents on the bottom reflect.

Next we will show the example query on the reviews collection. For this we will use the Atlas Aggregation GUI again. This will not use a `$project` because the number of columns is already small. However, we will return a query with dates greater than August 01, 2019:

The screenshot shows the Atlas Aggregation GUI with a **\$match** stage. The query is `{ date: {$gt: ISODate('2019-08-01T00:00:00')} }` (marked with a red '1'). The output shows three documents with fields `_id`, `listing_id`, `reviewer_id`, `reviewer_name`, `comments`, and `date`.

This query uses the `ISODate` helper function to match the dates properly. We see results returned from the query.

## Queries

### Query 1: Count query

Query:

Find the number of listings available with greater than 2 bedrooms in Portland.

The purpose of this query to allow us to query data from a particular subset of our documents and then perform a selection from that point. Limiting to Portland is a logical starting point since it is local to us. Choosing 2 bedrooms may need to be adjusted depending on the quantity of data returned to have a higher granularity of number bedrooms or increasing the number of cities in the query.

Index:

The index choice here would be based on the number of bedrooms. This provides a practical use case for a search as well as an index for the Airbnb dataset, because users will often search for a certain number of rooms available. We choose to implement an index on the attribute *bedrooms*.

Note here that the best index for this query is on (location, #bedrooms). The second best configuration is having two separate indexes one for each attribute assuming MongoDB makes use of that (relational DBs probably do).

Implementation Strategy:

1. Perform a filter where the city is equal to "Portland" (\$eq)
2. Perform a secondary filter where the number of bedrooms is greater than 2 (\$gt)

### Query 2: Sorted Subset

Query:

Find the listings with the top 5 highest number of reviews for the entire dataset.

The purpose of this query is to perform a search over the entire dataset. This search will allow us to see performance implications of sorting an entire dataset to find the top number of reviews.

Index:

The index that will provide significant efficiency here is that of an index on the attribute *number\_of\_reviews*. For the purpose of this query, this will provide significant performance increase. However, from a practical standpoint would a database have this as an index? If time allows, comparison will be completed on this query with an index and without because a sort would be expensive. We choose to implement an index on the *number\_of\_reviews*.

Implementation Strategy:

1. Perform a sort (\$sort) on the number\_of\_reviews decreasing (-1).
2. Project a limited number of columns to add additional complexity.
3. Limit to the top 5 results (\$limit).

Optional. If time allows, perform a grouping before the sorting to see performance impact of grouping by city.

### **Query 3: Subset-search**

Query:

For all listings that are classified as a house that have been updated within a week, what percentage of these have a strict cancellation policy?

The purpose of this query is to again test the system when performing aggregation calculations on a subset of documents. Once we have narrowed down based on the

specified criteria, we will then perform a percentage calculation on the final set based on the string attribute *cancellation\_policy*.

Index:

This query introduces the first time in our report so far that we can benefit from having multiple indices. The first choice is to have an index on the *property\_type*, which is a logical choice for a housing application. Another choice is based on the *calendar\_updated* field. However, this doesn't warrant an actual use case outside of the query because presumably a user will be choosing a property based on features and amenities besides when the calendar was last updated. Lastly, an index on the *cancellation\_policy* would be another valid use case for the application and the query. Users might be interested in the cancellation policy if they are impulsive vacationers. Overall we choose to implement indices on *property\_type* and *cancellation\_policy*.

Implementation Strategy:

1. Perform a filter on the data to only grab documents where *property\_type* is equal to "House".
2. Perform another filter where the *calendar\_updated* is less than one week. This will require more effort for string comparison when needing to consider values such as "today", "6 days ago", "one week ago".
3. Capture the number of documents in this resulting set.
4. Perform a grouping on the filtered set to group on the *cancellation\_policy* and count using \$group and \$sum. This can then be formatted to get this count divided by the total number of documents to gather a percentage.



#### Query 4: Average

Query:

Find the average host response rate for listings with a price per night over \$1000

The purpose of this query is again to filter based on a particular subset of listings and perform an aggregation. The price point of \$1000 may need to be adjusted once we determine the cardinality of the data, but it is the starting point.

Index:

A possible index that has a valid business use case here would be the *price* attribute. This will be accessed frequently by the users as price is a critical deciding point for choosing a lodging. The next possible choice here could be based on *host\_response\_rate*, but this doesn't hold much of a business case outside of the query. We choose to implement an index on the attribute *price*.

Implementation Strategy:

1. Perform a filter to find listings with *price* greater than \$1000 (\$gt)
2. Perform an average aggregation for the average *host\_response\_rate*

#### Query 5: "Join"

Query:

Return the most recent review for all listings from Portland with greater than 3 bedrooms that is also a house.

This query's purpose is to exploit the lack of efficient joins in MongoDB. Mongo uses what is called a "Lookup". From here we can search from one table to another to find

matching values. This query will combine two different types of documents one from the *listings* collection to the *reviews* collection. These are stored separately in the Airbnb database.

Index:

The lack of efficient joins in MongoDB makes the arguments for indices intriguing for this particular query. We will benefit from having indices on the city (attribute *city*) and the number of bedrooms (attribute *bedrooms*). These are logical choices from the report standpoint and a business standpoint. The next consideration is to create an index on the attributes that we will perform the “join” or Lookup on. The attributes in question are the *id* from listings and the *listing\_id* from reviews. To make the lookup more efficient, having indices on both of these attributes is logical. We choose to have indices on *bedrooms*, *city*, *id* and *listing\_id* (from reviews).

I do not know much about MongoDB, but from a relational DB point of view, the index on id is not useful in this case. You will use bedroom and city to bring the desired documents from listing then take the ids from these documents and use the index on reviews.listing\_id to retrieve related reviews.

Implementation Strategy:

1. Perform a filter on the data to find all listings with city equal to “Portland” and greater than 3 bedrooms.
2. Perform a lookup between the listings and reviews tables with *id* and *listing\_id* as the lookup criteria.
3. For each host *id*, return the max *date* from the reviews (\$max). (This possibly can be completed before the lookup for efficiency)

## Query 6: Updating

Query:

Update all listings that have more than 2 bedrooms and more than 2 bathrooms from Portland to require guest phone verification (field is *require\_guest\_phone\_verification* with values ‘t’ and ‘f’).

This query will test one of the CRUD functions for MongoDB. We are updating a column that is not in use by any of the other queries, so its impact will be minimized when this query runs. Execution time here will depend on the cardinality returned from our selection predicate and use of tactical indices.

#### Index:

For this exercise, assuming that we already have individual indices on both the number of bedrooms and the number of bathrooms, attributes *bedrooms* and *bathrooms* respectively. We will also create a compound index for the combination of *bedrooms* and *bathrooms*. This choice reflects that users will likely be concerned with the number of both of those during every search. We will not include the index on *require\_guest\_phone\_verification* field for the purpose of this query, but if time allows we will run an additional query to see the response time of the update if *require\_guest\_phone\_verification* was indeed an indexed field.

#### Implementation Strategy:

1. Perform an updateMany on all listings from Portland with greater than 1 bedrooms and 2 bathrooms to set the *require\_guest\_phone\_verification* to true.

Many of your queries can be tested using two configurations: the first one is having one compound index, and the second one is having multiple single-attribute indexes. This can be an interesting experiment (just a suggestion).

## References

- [1] M. Cox, “Get the Data - Inside Airbnb. Adding data to the debate.” *Inside Airbnb*, Sep. 14, 2019. [Online]. Available: <http://insideairbnb.com/get-the-data.html>. [Accessed: Nov. 13, 2019].
- [2] MongoDB, Inc. “Operational Factors and Data Models,” *mongoDB.com*, Aug. 13, 2019. [Online]. Available: <https://docs.mongodb.com/manual/core/data-model-operations/#large-number-of-collections>. [Accessed: Nov. 13, 2019].
- [3] MongoDB, Inc. “New to MongoDB Atlas,” *mongoDB.com*, Nov. 8, 2019 [Online]. Available: <https://www.mongodb.com/blog/post/new-to-mongodb-atlas--get-started-with-free-database-tier-on-microsoft-azure>. [Accessed: Nov. 13, 2019]
- [4] MongoDB, Inc. “Atlas M0 (Free Tier), M2, and M5 Limitations,” *mongoDB.com*. [Online]. Available: <https://docs.atlas.mongodb.com/reference/free-shared-limitations/>. [Accessed: Nov 13, 2019]
- [5] Amazon, Inc. “Architecture,” *amazon.aws.com*. [Online]. Available: <https://docs.aws.amazon.com/quickstart/latest/mongodb/architecture.html>. [Accessed: Nov 13, 2019]
- [6] Bsonspec “BSON (Binary JSON): FAQ,” *bsonspec.org*. [Online]. Available: <http://bsonspec.org>. [Accessed: Nov 13, 2019]
- [7] MongoDB, Inc. “Set Up Atlas Connectivity,” *mongoDB.com*. [Online]. Available: <https://docs.mongodb.com/guides/cloud/connectionstring/>. [Accessed: Nov 13, 2019]