MongoDB

Part 3: Query Implementation

Bingo Bango Mongo

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

CS488/588 Cloud & Cluster Data Management

Table of Contents

System Overview

The dataset used for our project was the publicly available Airbnb *listings* and *reviews* in .csv format. We have chosen to center our project on the comparison between different cloud providers on the free-tier of MongoDB Atlas. MongoDB Atlas provides a base cluster setup of three replica set members, the minimum for a MongoDB replica set. Three separate clusters were then spun up, one for each provider: Google Cloud Platform (GCP), AWS, and Azure. We attempted to keep the location of the clusters close together to reduce the impact of network latency on our results. Each location is as follows: GCP -> Iowa, AWS -> Virginia, Azure -> Virginia. Each of the clusters were setup with the same configurations and loaded with approximately 500 MB of data from our Airbnb dataset. We decided not to change the default sharding mechanism from 3 shards to several. Instead, we focused on comparing the cloud providers' performance when running the same queries without exceeding free tier restrictions -- the reason for capping our data to ~500MB. We used a subset of the United States data since the entirety far exceeded the target size for our database. For the *listings* data, we loaded listings from Minnesota, North Carolina, New Jersey, Nevada, Oregon, Texas, and Washington. For the *reviews* data, we loaded reviews from Oregon, specifically Portland.

Our project code and query output is located at https://github.com/PieMyth/CloudCluster.

Query Implementation and Results

We implemented five of the initial queries from our data model and query design. The one query from our initial design we did not implement was the update operation, Query 5. In its stead, we included an additional join query, Query 7.

Our benchmark for performance was to run the join queries 10 times and all other queries 25 times. Join queries take a long time, and 10 repetitions is sufficient for an average. We took an average of all queries and graphed the results, grouped by query and cloud service provider. The number of repetitions is specified in the graphs as **R=# of repetitions** and the benchmarked times are represented in milliseconds.

Query 1: Count query

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

This query is a simple aggregation using the *collection.countDocuments()* function exposed by the mongoDB C# API. This function wraps the passed-in query filter in a sum aggregation and returns the number of documents returned by the query. The filter passed in was a match on the *city* field for “Portland”, and a match on *bedrooms* field for any value greater than 2. The resulting document count was 526 listings.

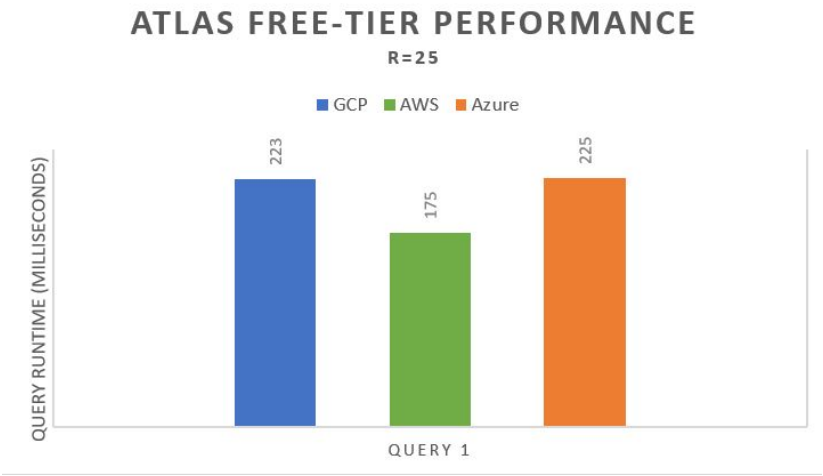

Figure 1: Query 1 average results in milliseconds averaged over 25 runs

Figure 1 displays the resulting average runtime performance on each Atlas free-tier option. The differences between GCP and Azure were negligible. On the other hand, AWS showed an approximately 29% quicker runtime compared to both GCP and Azure.

Query 2: Sorted Subset

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

This query took advantage of mongoDB indexing to efficiently retrieve the top five most reviewed listings. A *_createIndexDescending(collection, newIndex)* method was created specifically to create descending indexes on a specified field. In this case, the new index was *number_of_reviews* from the *listings* collection. Once the index was created, a descending sort was applied as this results in more efficient queries. Finally, a limit of five was placed on the query results and the top-five reviewed locations resulted. The following displays the console output (without the platform benchmarking):

```
2019-12-04 20:52:03.2816 | Creating new index on number_of_reviews, descending
2019-12-04 20:52:03.3659 | Top five listings with sorted number of reviews:
2019-12-04 20:52:04.5987 | (1) ID: 1260528, number of reviews: 987, Description: A Guest suite located in Portland, OR.
2019-12-04 20:52:04.5987 | (2) ID: 329997, number of reviews: 848, Description: A Guesthouse located in Nashville, TN.
2019-12-04 20:52:04.5987 | (3) ID: 1021139, number of reviews: 836, Description: A Apartment located in Austin, TX.
2019-12-04 20:52:04.5987 | (4) ID: 25002, number of reviews: 747, Description: A Guest suite located in Seattle, WA.
2019-12-04 20:52:04.5987 | (5) ID: 3861673, number of reviews: 743, Description: A Guest suite located in Seattle, WA.
2019-12-04 20:52:04.5987 |
Query run time: 00:00:01.3278018
```

Figure 2: Query 2 sorted subset console output

You can see that the review values are descending and, interestingly, the most reviewed location is a guest suite located in Portland, OR.

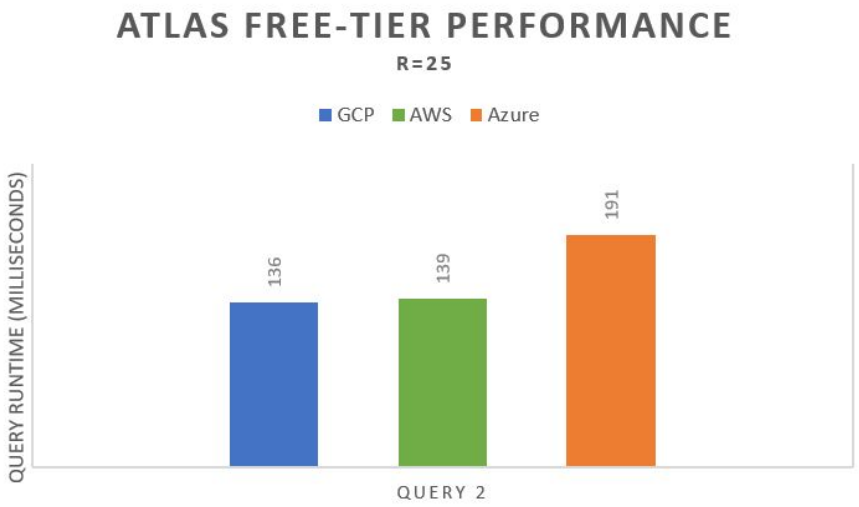

Figure 3: Query 2 average results in milliseconds averaged over 25 runs

The performance results were trivial between GCP and AWS with GCP performing several milliseconds better. Azure, on the other hand, had comparatively poor performance with an almost 40% increase compared to GCP. It is difficult to speculate why there were differences for Azure and more research would be needed to provide insight into this performance gap.

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?

This query implementation avoided nested group and lookup operations by using saved results within C# variables. With this approach, slow joins were avoided and simpler calculations, less suitable for MongoDB queries, could be dealt with in C#. The first query filters all documents

with *property_type* of “house” that were updated within a week from the *calendar_updated* text field from the *listings* collection. Applying a numeric range filter on textual data would be problematic, but this text field contained a finite number of string combinations so an intermediate *group* query could be applied. For all listings updated in less than a week, a pattern existed where they contained “days”, “yesterday” or “today”. After enumerating these possibilities with a regular expression, the query successfully filtered all houses updated within a week.

Next, a projection reduced the total width of each document thereby reducing total network transfer and latency [1]. Finally, the total number of documents were stored in a C# variable and the query repeated with an additional filter to limit all documents without a strict cancellation policy indicated by the keyword *strict* within the *cancellation_policy* field. The resulting total number of documents from this query could be used with the previous value to compute a percentage for strict cancellation policy for “house” listings. Incidentally, this value hovered around ~50% as indicated by the results.

```
Query began at: 12/7/2019 6:29:47 PM
The percentage of listings with a strict cancellation policy is 51.84139%
The percentage of listings with a strict cancellation policy is 51.84139%
The percentage of listings with a strict cancellation policy is 51.84139%
The percentage of listings with a strict cancellation policy is 51.84139%
The percentage of listings with a strict cancellation policy is 51.84139%
The percentage of listings with a strict cancellation policy is 51.84139%
The percentage of listings with a strict cancellation policy is 51.84139%
The percentage of listings with a strict cancellation policy is 51.84139%
The percentage of listings with a strict cancellation policy is 51.84139%
The percentage of listings with a strict cancellation policy is 51.84139%
```

Figure 4: Query 3 subset search console output

Viewing the results as a comparison between the three cloud providers, Azure’s consistently slow performance is notable. Since the trials are repeated and the difference is not trivial, one can surmise it may be a hardware/network handicap of some sort; a verifiable reason is unavailable with classroom research. The general slower performance for this query compared to others (without a lookup) can be explained by the additional query requirement, and possibly the use of regular expressions, which adds overhead to each document evaluation.

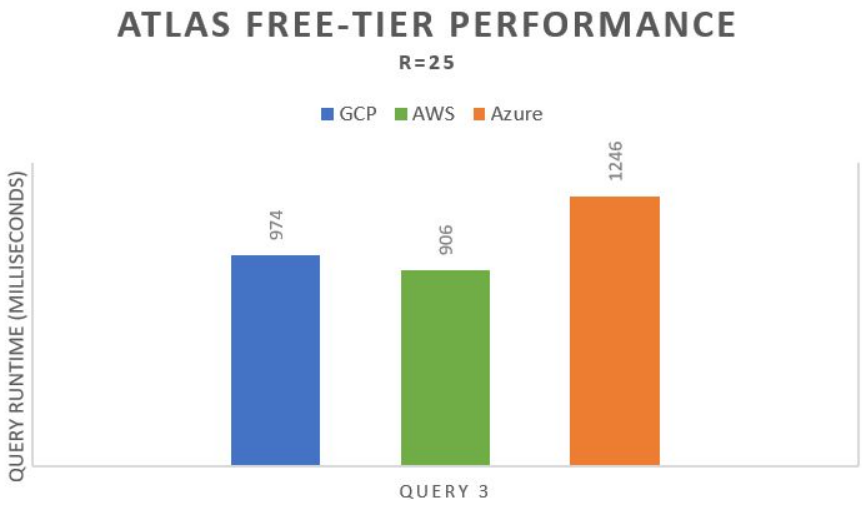

Figure 5: Query 3 results in milliseconds averaged over 25 runs

Query 4: Average

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

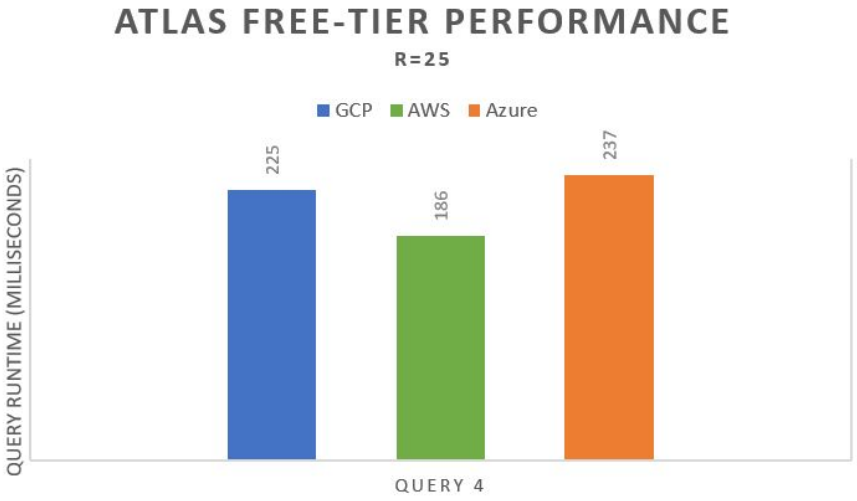

Figure 6: Query 4 average results in milliseconds averaged over 25 runs

The results between the average query run on the different platforms are almost negligible as there is only a 51ms difference between the fastest and slowest average over the query. That being said, AWS did outperform its competitors. It should also be noted that this query had to remove null values where the host's response rate was unavailable. In this case, the whole document was dropped from the averaging query to avoid a "nullable" return value.

```
Average response percentage for Airbnbs over 1000: 97.041016406562619
```

Figure 7: Query 4 average console output

The query was implemented by using a pipeline consisting of both a filter and an aggregate step. The filter step was created to drop documents that did not meet the minimum of $1000/night cost or did not have a host response rate field. With the valid documents that met the requirements, a grouping was performed by the "_id" field, which was set to "NULL". When the group by field is set to null in mongoDB, this groups all the documents together into one group for the aggregate function. This is then used to get the average of the host response rate field across all valid documents. The result is returned as a document consisting of the "nullable" *_id* field and the *avg* field that holds the the query average. To successfully achieve this with the C# driver, the returned document must be specified via the *Single()* function attached to the end of the query. This allows reasonable field access with a single BSON document. For instance, *result["avg"]* would return the average that was kept in the *avg* field, which is shown above in Figure 7 with a 97% average.

Query 6: First Join

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

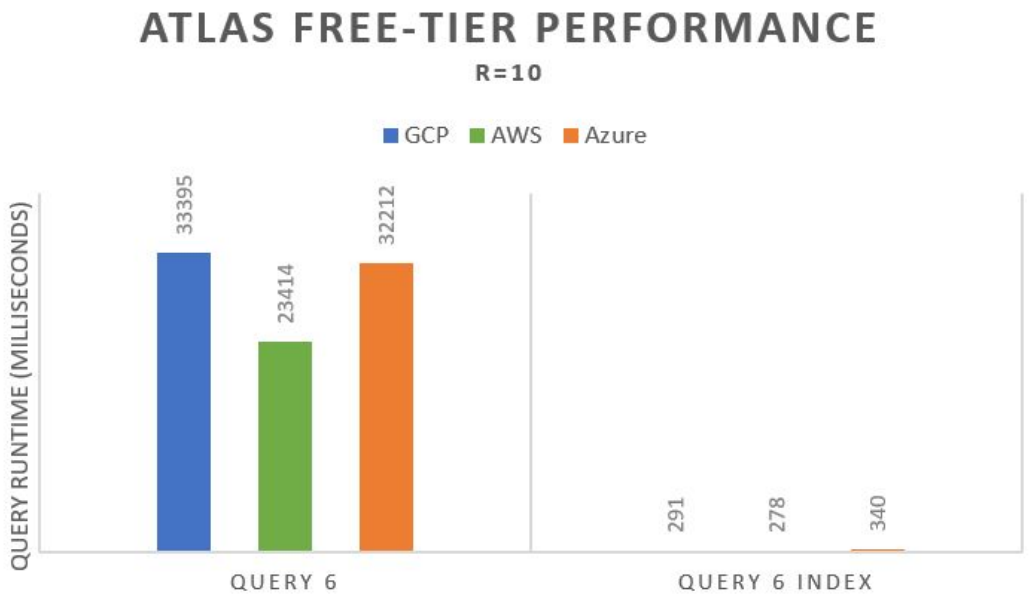

Figure 8: Query 6 join results (ms) with/without indexes on join keys averaged over 10 runs

This query started our investigation into the join ability that MongoDB provides through the *Lookup* operation. While it is not as robust an implementation as a relational database, it does provide the ability to compare one collection to another. We approached this query first by indexing on the join keys: “id” field from *listings* and “listing_id” field from *reviews*.

When attempting to complete many memory intensive steps in a single query, the query often ran out of working memory, so we determined that an optimization should be applied. The left side of Figure 8 shows us an average execution time of ~23,000ms at minimum and ~34,000ms at maximum. The AWS platform performed roughly 35% faster than GCP and 31% faster than Azure. The execution time for all three is long enough to discount network lag as the cause of performance difference.

We then completed the same join after applying indexes on the join keys to compare the benchmarks. The right side of Figure 8 above shows us those results. Again we see the AWS benchmark (the middle of the three results) performing faster than GCP and Azure. However, since they are all close to ~300ms, the numbers do not reveal a compelling reason behind the difference between the three cloud providers when the operation is fully optimized with the join indexes. We do see concrete proof that if a *Lookup* needs to be performed in a MongoDB, indexes provide more optimal performance.

Possible improvements for the query could be adding additional indexes on the filtered fields. Specifically, these fields would be the “city”, “bedrooms”, and “housing_type” since we are filtering for all Portland homes with more than 3 bedrooms.

Query 7: Second Join

Query: Returns the listings of the person who has travelled the most based on their number of reviews.

This query continues our investigation of joins. To reduce the complexity of the query, we broke it into two separate queries to the database: (1) retrieve the traveller that has the greatest number of documents in the *reviews* collection, and (2) retrieve all listings for which this traveller submitted a review in chronological order. This query was run without any indexes, and then with the same indexes as specified in Query 6.

The first query did a sum aggregation of documents for each unique *reviewer_id*. Then, it sorted the results in descending order and returned the first result. For our dataset, the *reviewer_id* with the associated name “Celeste” was returned [3]. Next, the second query first filtered out any documents in the *reviews* collection that were not submitted by the *reviewer_id* for Celeste,

and sorted the documents in chronological order. It then performed a join with the *listings* collection and returned the dates and location for each document. A total of 53 documents were returned.

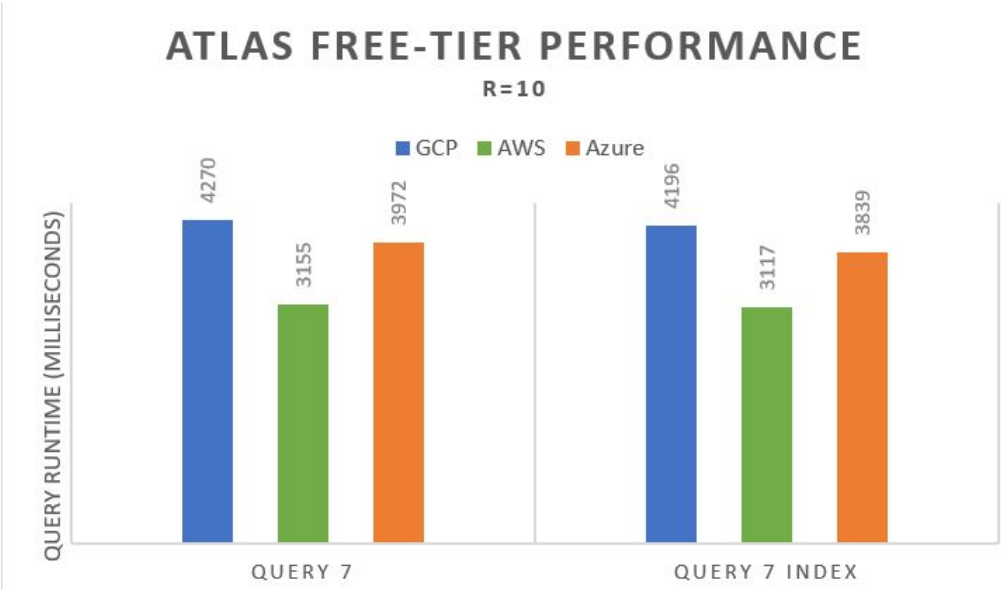

Figure 9: Query 7 join results (ms) with/without indexes on join keys averaged over 10 runs

Using an index on this query showed negligible improvement in query runtime performance. This is because the join played a relatively insignificant role in the query runtime, only joining 53 documents from the *reviews* collection with the *listings* collection. In this case, the aggregation performed in the first query produced the greatest overhead, as the *reviews* collection is significantly larger. An index on the *reviewer_id* field would improve aggregation performance, but it would require having enough space in memory to store the index, other indexes and the query's working set.

In both trials with and without an index, AWS performed significantly better than both GCP and Azure. AWS was ~35% faster than GCP, and ~24% faster than Azure. Additionally, Azure showed an ~8% quicker runtime than GCP. The length of the query runtime rules out network latency as a significant factor in these differences. From the results of our other queries, Azure performs equally or slightly worse than GCP. Comparing with the previous join query, GCP fares worse than Azure, but our data is insufficient in explaining why this performance switch occurs.

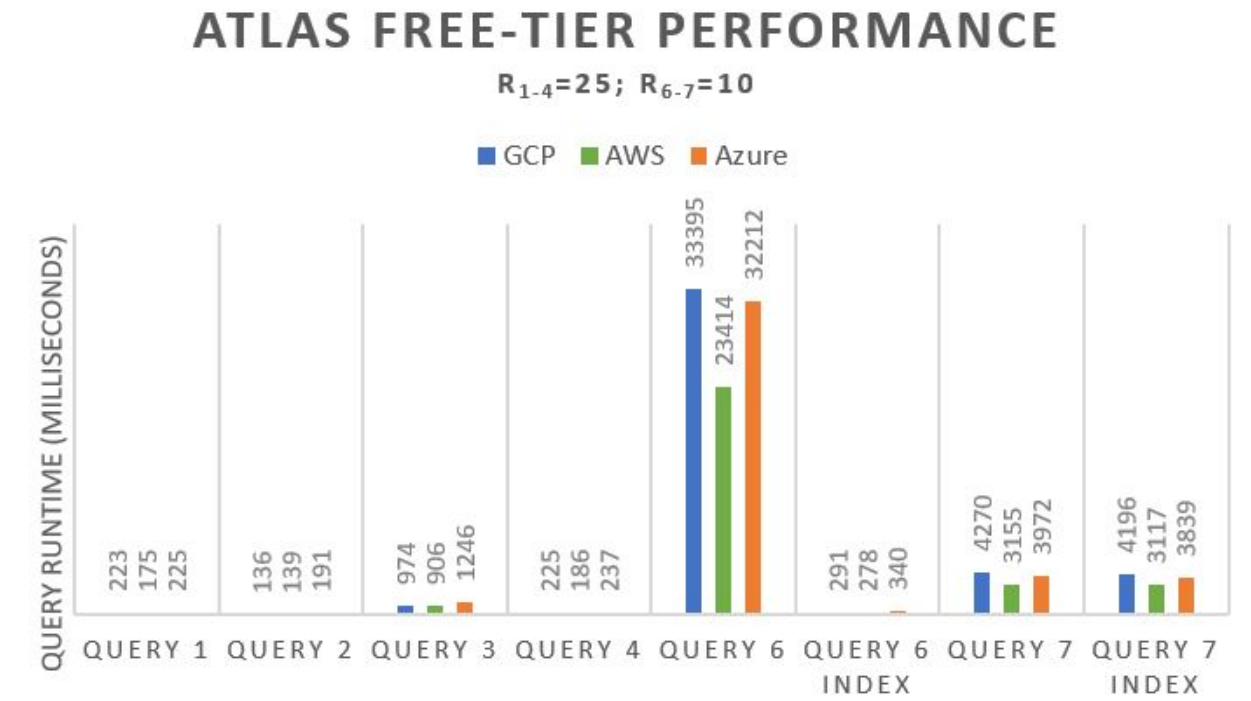

Figure 10: All queries

Critique of Data Model and Query Plans

When we first wrote our design plan, we created a data model that would take advantage of embedded documents as well as the tree structures that are available in mongoDB. We had a couple reasons for implementing a relatively flat data model instead: (1) a direct key-document approach seemed reasonable considering the format of the existing data, and (2) we faced challenges with importing data (specifically with C#) that made us simplify our objectives.

If we were to re-design our model, we would likely compare benchmarking from this research with a denormalized model incorporating embedded documents and reflecting specific AirBnB use cases. It would be interesting to see what impact the data model has on the performance and if, by creating a more structured model with layered embeddings, the performance would improve or degrade.

Further preprocessing of data may have also aided certain queries. For Query 3, splitting the textual fields into an array type or making them numeric may have resulted in a more reliable

query calculation. The regular expressions that were used in Query 3 are problematic for the real-world case where data input can change depending on the sources.

While mongoDB performs very poorly with joins, having an index on the join columns can significantly improve runtime performance, as seen in Query 6. However, there must be sufficient space in memory for all indexes on each collection, as well as the working set for queries. While an index on the *reviewer_id* field in the *reviews* collection would speed up aggregation in Query 7, the data is so large that our use case for the reviews does not justify having such an index for our data model. Our data model had one foreign key in the *reviews* collection, *listing_id*, to the *listings* collection, *id*. This is a common join, and the performance gains are so significant, as seen in Figure 10, that an index on these columns works well with our data model.

Lessons Learned

For the most part, we enjoyed using mongoDB for querying. Some of the biggest challenges were building up the driver, importing data, and the trial and error that went into connecting with the platforms before we chose Atlas. The overall mongoDB documentation was easy-to-use, however, documentation specific to the C# driver (with deprecated functionality) was more difficult to find or referred to earlier mongoDB versions. Learning to manage BSON data was also an initial challenge when trying to import data with C#.

While importing the data with C# was achieved by one of our team members, we ultimately chose to use a python script for importing data. Using a simple, reusable and easily modifiable script for importing data is something we would recommend for future users to save time, ease of pre-processing data, and ease of coding. As the import operation is infrequent, this seemed like a better decision for a time-boxed deadline. Additionally, the pandas python library offers functionality that simplifies table operations, which would prove useful for exploring data model modifications such as embedded documents.

Working with in C# was quite fun, but more challenging than another language like Python. This is partly because the majority of mongoDB driver examples are written in Python, which makes sense because it is an easier language to learn. This likely explains why there is a wider availability of resources and up-to-date documentation for the Python driver as well. We actively chose a harder challenge with C# (half of us had never used the language, and the other half had a handful of C# project experience), and it was quite rewarding after the initial effort. For someone who is looking for an easier approach or who wants to spend more time with the

database, we would likely recommend Python, however, we did not have any regrets for our language selection.

MongoDB was easy to learn and implement with simple queries, but as the queries became more complicated, the learning and implementation became noticeably steeper. Different approaches for the same query (e.g. using filters versus using mapReduce or using indexes) would result in dramatically different performance as we also demonstrated with comparisons with/without indexing.

One of our conclusions about the system was that it is so popular that it may be selected and implemented for small projects without much prior research. Our research left us slightly skeptical of mongoDB's claims of achieving all of CAP -- consistency, availability, and partition tolerance -- which we know to be theoretically impossible. Their marketing should probably be taken with a grain of salt with a lot of research to understand where these claims stem from and the configurations required to achieve different data store features.

In terms of cloud data management, we were fortunate that one of our team members took it upon himself to research Atlas further as we had initially pushed it aside. Based on his research and proof-of-concept, we returned to Atlas as a cloud service requiring less management. Its built-in ease of connecting to GCP, AWS, and Azure greatly simplified our stretch goals as well as broadening what we could benchmark. We would highly recommend Atlas to future users as a straightforward introduction to cloud data management with a clean GUI, and user-friendly guides to progressing to next-steps of cloud-management.

Attempts were made to connect directly with GCP, which was not too bad considering it targets the educational sector and has many guides to help individuals configure settings. AWS on the other hand, caused a lot of headache with poorly written guides and less-than-transparent terms for billing. Its appeal remains because of its success in industry as well as the results that were benchmarked in our research where it frequently outperformed its competitors, at least for the Altas free-tier offerings.

Given More Time

It would be worthwhile to continue research to understand why a performance gap exists between the different platforms. To benchmark more rigorously, we may want to modify our code for synced comparisons on separate platforms instead of consecutive runs. Given more experience with the challenging AWS platform, we might exclude Atlas and connect directly with the platform providers. We added stretch goals as we progressed through our project and some additional features and/or functionality we would want to explore would be:

- ➔ Creating a front-end application for selecting platforms, querying with and displaying results from our driver to further abstract things and make the functionality user-friendly
- ➔ Explore various data-models (perhaps with embedding)
- ➔ Explore different sharding configurations
- ➔ More of our stretch goals and additional documentation can be found in our README.md (https://github.com/PieMyth/CloudCluster/blob/master/README.md), specifically in the ***Stretch Goals*** section.

References

[1] MongoDB, Inc., “The MongoDB 4.2 Manual,” mongoDB.com, Aug. 13, 2019. [Online]. Available: https://docs.mongodb.com/manual/tutorial/optimize-query-performance-with-indexes-and-projections/. [Accessed: Dec. 7, 2019].