APOLLO GROUP℠

How Apollo Group Evaluated MongoDB

Brig Lamoreaux, Forward Engineering Project Lead, Apollo Group

10gen | the MongoDB company

How Apollo Group Evaluated MongoDB

By Brig Lamoreaux, Forward Engineering Project Lead at Apollo Group

Introduction: A Strategic Initiative

For most people, Apollo Group is best known as the parent company of the University of Phoenix. We educate 350,000+ students a year, reinventing higher education for today's working learner.

Always a leader in harnessing technology in the service of its students, Apollo Group found itself at a critical juncture in 2011. Our organization was planning a strategic initiative to create a cloud-based learning management platform. This initiative would require significant changes to the existing IT infrastructure in order to accommodate a substantial increase in the number and diversity of users, data, and applications.

We set out to evaluate which technologies would be best suited to support the platform. As part of this process, we evaluated MongoDB, a NoSQL document-oriented database. We tested MongoDB in several cloud-based, on-premise, and hybrid configurations, under a variety of stress conditions. MongoDB's ease of use, performance, availability, and cost effectiveness exceeded our expectations, leading us to choose it as one of the platform's underlying data stores. This paper describes the process and outcomes of our assessment.

The Challenges

We faced the following challenges with our existing infrastructure while planning for the new platform:

- **Scalability.** We were unable to scale our current system to support the anticipated number of users and volume of content, which would increase significantly as we would add applications to the platform.
- **Technology Fit.** Much of the data targeted for the platform was semi-structured and thus not a natural fit with relational databases.

Our Approach

The Apollo IT Team driving the initiative knew that it was contending with an aging infrastructure. The Oracle system had been in place for nearly 20 years and would have neither the flexibility nor the capacity to meet our future needs.

We decided to look for a solution with a better technological and financial fit, and the team developed a short list of potential solutions. While we strongly favored a solution that was already in-house, we added MongoDB to the short list because our research indicated that it might provide excellent query performance with less investment in software licenses and hardware than other solutions. However, our primary concern with MongoDB was that we had no hands-on experience with it.

Apollo management tasked the Forward Engineering group within IT – my team – with assessing MongoDB. We responded with an evaluation process designed to determine in a rigorous yet time-sensitive manner whether it would suit our needs.

The Purpose of This Paper

Our goal in producing this paper is to help other organizations with the type of analysis we applied to MongoDB. In eight weeks, we were able to produce relevant and useful data on the behavior of MongoDB when deployed in the Amazon Elastic Compute Cloud (Amazon EC2) that we are eager to share with the community. This paper also identifies areas for additional research on the behavior of MongoDB under stress conditions.

The Evaluation Process

Each year, the Forward Engineering team evaluates dozens of new technologies for use within Apollo Group. Given this level of experience, we were confident in our ability to evaluate the capabilities of MongoDB and to determine whether it would meet our requirements.

The Process

Our goal was to complete the evaluation in eight weeks. We divided the evaluation into four phases of two weeks each. Each phase had pre-defined goals.

Table 1: Project Phase Schedule & Objectives

PHASE	TIMEFRAME	OBJECTIVES
Phase 1	2 weeks	- Launch a cross-functional team of stakeholders - Agree on goals and objectives - Gather query usage data from legacy Oracle system
Phase 2	2 weeks	- Develop MongoDB data model - Develop sample use case application and generic service layer to access data store - Stand up a small MongoDB server
Phase 3	2 weeks	- Stand up a five-node MongoDB deployment - Develop Apollo's runbook for MongoDB
Phase 4	2 weeks	- Performance test

Phase 0

At the outset, our mission was somewhat loosely defined: to learn about MongoDB and to determine its suitability as a data store.

FORM TEAM OF STAKEHOLDERS

First, we launched a cross-functional team of stakeholders to determine objectives and to guide the project. Stakeholders included representatives from Apollo's business side, the database administration group, application development, and Forward Engineering.

IDENTIFY GOALS AND OUTCOMES

After some discussion, the Stakeholder Team decided that the evaluation process should meet the following goals:

- » Learn how to design and deploy a large MongoDB farm, and document it in a runbook for MongoDB.
- » Learn how to maintain and troubleshoot production-scale deployments of MongoDB, documenting it in the runbook.
- » Determine how Apollo should organize and train its teams to support a MongoDB deployment. This meant understanding the different roles needed, (e.g., system administrator, developer, database administrator) and the expertise required of each person on the team.
- » Answer the team's questions about MongoDB (summarized in Table 2).

The stakeholder team sought to answer the following questions about MongoDB to determine whether it would be a suitable data store for the platform.

Table 2: Evaluation Questions for MongoDB

Resiliency	Is MongoDB robust enough to be a critical component in Apollo's next-generation platform? If failures occur, how does MongoDB respond?
Stability	MongoDB is relatively new. Is it high-quality enough to support our infrastructure without unexpected failures?
Adaptability of Data Model	If the data model needs to change, can this be done quickly and efficiently in MongoDB? How do changes to the data model impact the applications and services that consume it?
Performance	Does MongoDB perform well enough to serve a massive application and user base, without creating delays and a poor user experience? What is the performance of MongoDB when deployed on Amazon EC2? Is response time acceptable? Is throughput sufficient?
Configuration Flexibility	How suitable is MongoDB for a hybrid deployment with both cloud-based and on-premise components?
Time to Implement	How long does it take to install and deploy a production MongoDB configuration?
Administrator Functionality	How difficult is it to administer MongoDB, including tasks like performing backups, adding and removing indexes, and changing out hardware?
Training	What current and ongoing training do Apollo operations staff and developers need if we adopt MongoDB?
Data Migration & Movement	How should we migrate data from our current Oracle data stores into MongoDB? Once deployed, how should we load data into MongoDB on an ongoing basis? How can data be retrieved from MongoDB?
Conformity with Company & Industry Standards	Since MongoDB is not yet a corporate standard for Apollo, does it fit well with the rest of Apollo Group's technical infrastructure? Is MongoDB an industry de-facto standard? If not, is it well positioned to become one?
Quality & Availability of Support	If something goes wrong with our MongoDB configuration, can we get qualified, top-notch assistance – even in the middle of the night or on a holiday?

IDENTIFY A REALISTIC USE CASE

Shortly after organizing our stakeholder team and setting our goals, the team identified a good use case that would let us conduct an apples-to-apples comparison with our Oracle configuration.

In creating a use case, the team focused on commonly-used, discrete functionality from the Oracle system:

» For a specific student, retrieve a list of classes in which the student is enrolled.

» For a specific class, retrieve a list of the students enrolled.

» For a specific instructor, retrieve a list of the classes taught.

In Oracle, the data required to serve these queries was stored in 6 tables.

GATHER BASELINE METRICS FROM LEGACY SERVER

In order to have a basis for comparison, we pulled the actual historical usage logs for the student, course, and instructor tables in the legacy Oracle system. These logs covered the previous four weeks; they contained the queries triggered and their performance from whenever students, instructors, or administrators examined class schedules and class rosters.

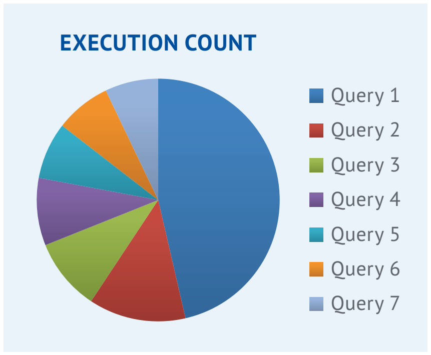
ANALYZE LEGACY QUERY PERFORMANCE & IDENTIFY EVALUATION METRICS

Upon analyzing the query usage data from the legacy Oracle system, we discovered the following (see Figure 1):

» The Oracle system was able to perform about 450 queries per second with acceptable response times. The production MongoDB system would need to handle at least the same load.

» Logs revealed that Query 1 – a query to return all courses for a given student – dominated all other queries. Four weeks of historical data showed that Query 1 was executed 15.6 million times out of 33.6 million query executions total. Furthermore, the top 5 queries comprised over 85% of all queries. Query 1 was responsible for nearly 50% of all query executions. Queries 1 through 5 were cumulatively responsible for over 85% of all query executions.

Figure 1: Actual Oracle Query Usage



Phase 1

Phase 1 involved implementing our use case with real-world data on a single-server configuration of MongoDB.

DESIGN THE DATA MODEL

The most fundamental difference between Oracle and MongoDB is the data model. In Oracle, all data is stored in relational tables, and most queries require joins of these tables. MongoDB, on the other hand, uses neither tables nor joins. Instead, it uses a document-based approach where data that is usually accessed together is also stored together in the same MongoDB document.

In Oracle, our use case's data was stored in 6 relational tables with several indexes, and accessed via complex SQL queries that used several joins. We needed to transform this data into a MongoDB document-based data model. Creating an optimized data model is critically important – if not done well, one can lose many of the benefits of MongoDB, such as reducing the number of queries and reducing the amount of reads and writes to disk.

Fortunately, the task came together easily, because we focused on how the data was being used. Armed with our query and table usage data from the Oracle system (from Phase 0), and guided by knowledgeable 10gen consultants, we designed a data model that was optimized for the most common queries.

As mentioned earlier, a single query accounted for nearly 50% of the query executions. We thus designed a data model that represented the exact fields from this query. We then progressed to the next most common query. This query was very similar to the first, and asked for all the students in a given course. In fact, the top five queries – which cumulatively accounted for 85% of all query executions – were just variations of the same basic query.

As a result, we were able to reduce the original 6 relational tables with numerous indexes in Oracle to just one collection with 2 indexes in MongoDB (see Figure 2). For our use case, data that is most frequently accessed together is also stored together. So, user data (students and instructors) and course data are stored as user-course pairs. In a relational database this would be stored differently, with separate tables for users and courses.

Figure 2: Our Simplified MongoDB Data Model

```
{
  "_id": "8738728763872",
  "role" : "Student",
  "user :{
    "id" : "b7ed789f198a",
    "firstName" : "Rick",
    "lastName" : "Matin"
  },
  "course" : {
    "dateRange" : {
      "startDate" : ISODate("2011-12-30T07:00:00Z"),
      "endDate" : ISODate("2012-01-30T07:00:00Z")
     },
     "courseId" : "734234274",
     "code" : "MATH/101",
     "title" : "Introduction to Mathematics"
  }
}
```

DESIGN THE SERVICE LAYER FOR DATA STORE ACCESS
To do an apples-to-apples comparison of Oracle and MongoDB, we designed a common service layer that our student roster application could use to access either data store without code changes. This service layer allowed the test application to create, read, update, and delete records individually, and also performed some basic data aggregations.

STAND UP SMALL MONGODB SERVER
With our MongoDB data model designed, our next task was to get a small MongoDB server running. This was a simple process that took half a day.

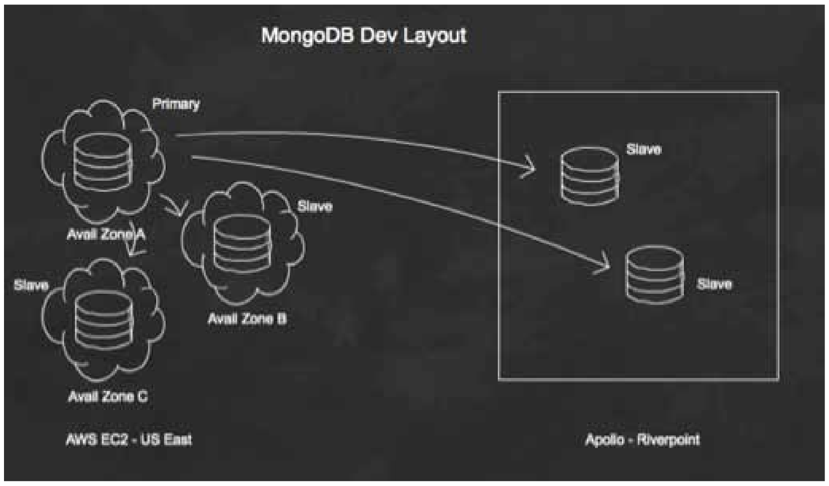
» **Install and Activate.** We quickly spun up a blank virtual machine[1] on Amazon EC2, and then installed MongoDB on it.

» **Populate with Course Data.** We used approximately 300,000 very simple records. We used a Python script to import the data.

After 4 hours, our small MongoDB server was operational and fully loaded with our test data. We were ready to test the implementation of our use case, the student roster application.

RUN USE CASE ON SINGLE SERVER
We ran our student roster test case against the single MongoDB instance. All the queries executed correctly and we experienced no problems. While we were not measuring performance at this point, we noted that the system seemed quite responsive and returned results quickly.

Figure 3: Hybrid Configuration — Multi-Node MongoDB Clusters



Phase 2

Having successfully deployed and tested MongoDB on a single Amazon EC2 server, we moved on to standing up various large MongoDB configurations. Our aim in this phase was to learn how to deploy multi-node MongoDB clusters that used replica sets, and how to deploy MongoDB in a hybrid cloud/on-premise configuration.

Our operations teams built a hybrid configuration of geographically distributed nodes that were both in Amazon EC2 and on-premise in an Apollo data center. We deployed five nodes total: three nodes on Amazon EC2 (one master and two slaves) and two slaves within

[1] The server and clients were Amazon EC2 m1.large instances: 7.5 GB memory, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 50 GB instance storage, 64-bit platform, I/O Performance: High, EBS-Optimized Available: 500 Mbps

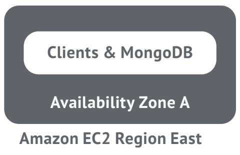
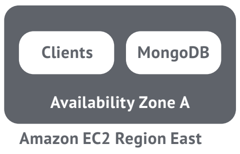
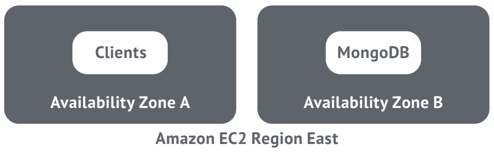
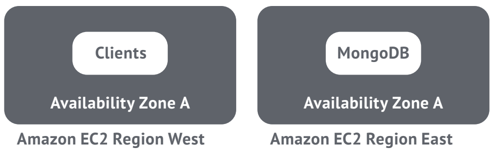
an Apollo Group data center. All the Amazon EC2 nodes were within the same Amazon EC2 Region, but each within different availability zones.

In standing up these various large configurations, we developed Chef and Puppet scripts that let us quickly deploy new MongoDB farms and add replica sets, as well as monitor the servers. We also created a runbook to instruct administrators on how to install MongoDB instances and keep them running.

Phase 3

Whereas the first three phases centered around getting familiar with MongoDB and how to set it up, Phase 3 focused on performance testing. MongoDB performed flawlessly in our preliminary setups, but we did not yet know how it would perform in a production-scale deployment. In particular, we wanted to learn how using various Amazon EC2 availability zones and regions affected overall performance.

Table 3: Results of Performance Testing Across Various MongoDB Configurations on Amazon EC2. See appendix for more detail, tables, and graphs.

	CONFIGURATION	RESULTS OVERVIEW
A	**Clients and MongoDB on Same Amazon EC2 Instance**  Clients & MongoDB Availability Zone A Amazon EC2 Region East	**Typical Response Time:** 0-1.7 ms **Maximum Throughput:** 9,000 queries/sec *CPU-bound.* **Typical CPU Utilization:** 100%
B	**Clients and MongoDB on Separate Amazon EC2 Instances, but within Same Amazon EC2 Availability Zone**  Clients MongoDB Availability Zone A Amazon EC2 Region East	**Typical Response Time:** 1.2-8.5 ms **Maximum Throughput:** 12,000 queries/sec **Typical CPU Utilization:** 80%
C	**Clients and MongoDB in Separate Availability Zones, but within One Amazon EC2 Region**  Clients Availability Zone A MongoDB Availability Zone B Amazon EC2 Region East	**Typical Response Time:** 1.2-10.6 ms **Maximum Throughput:** 12,200 queries/sec **Typical CPU Utilization:** 85% *Approximately the same response time, throughput, and CPU utilization as Configuration B.*
D	**Clients and MongoDB in Different Amazon EC2 Regions**  Clients Availability Zone A Amazon EC2 Region West MongoDB Availability Zone A Amazon EC2 Region East	**Typical Response Time:** 85.6-87.3 ms **Maximum Throughput:** 1,600 queries/sec **Typical CPU Utilization:** 2%. Very low. *EC2 instance was unstressed.* *East coast-west coast network was bottleneck in this configuration – EC2 instances were not stressed. Response times were much higher than when instances were located within a single Amazon EC2 region (configurations B & C).*

During this phase, we compared query response time and query throughput on several single-server MongoDB configurations that differed by Amazon EC2 Region and Availability Zone. For each configuration, we had varying numbers of clients query the server for a random student record. This let us see how the configurations responded in both unloaded and saturated states.

In general, MongoDB showed exemplary performance. When clients were within the same Amazon EC2 region – even if spread out across multiple availability zones – response time and throughput handily met our requirements. When clients and the MongoDB instance were in different Amazon EC2 regions, though, throughput decreased by over 80% and response time increased 50x.

What Apollo Group Learned about MongoDB

The findings from any evaluation process are both project- and company-specific. Depending on the goals and evaluation methodology, experiences may be different. Nevertheless, we made several general findings about MongoDB that are likely applicable to the community:

1. **The key to MongoDB performance is thoughtful data model design.** To move correctly from a relational model to a document-oriented model, one must be data-driven. Measure how often each type of query is triggered in the relational system, and then design the MongoDB data model to optimize returning the data of the most common queries.

2. **Design the deployment to match data usage.** A system with many reads should be designed and deployed different than a high-write system.

3. **MongoDB is a great match for cloud-based, on-premise, and hybrid deployments.**

4. **MongoDB has excellent availability.** It is able to deal with run-of-the-mill hardware failures, such as losing hard disks (which was important because our legacy Oracle system experienced hard disk failures every 2 weeks), as well as losing entire geographic sites or Amazon EC2 Regions.

5. **Latency between different Amazon EC2 Availability Zones within the same EC2 Region is very low.** This is a distinct advantage for users of this particular public cloud solution. Latency significantly increases for servers in different Amazon EC2 Regions.

6. **At 85% CPU utilization, the behavior of MongoDB changes, and performance levels off.** We posited that this might be due to MongoDB throttling itself. This is an area that we believe could benefit from further research.

Evaluation of MongoDB Suitability

Our investigation revealed the following answers to the questions posed by our stakeholder team in Phase 0:

Table 4: Results of Evaluation Questions

Resiliency	MongoDB's architecture proved to be highly resilient. It can survive multiple-node failures and even site-wide failures. The replica set feature works well and is easy to use. As such, we are confident that MongoDB would allow our systems to operate even if one of Apollo's two data centers were inaccessible.
Stability	During our testing, MongoDB was stable and easy to integrate with Apollo Group's technical infrastructure.
Adaptability of Data Model	It was easy and fast for us to change the data model as needed without impacting users or applications. This was one of MongoDB's biggest strengths over relational databases.
Performance	MongoDB performed very well – as well as or even better than the Oracle system. Under heavy load, CPU utilization leveled off at 85%.
Configuration Flexibility	Using Amazon EC2, we successfully deployed MongoDB in cloud-based, on-premise, and hybrid configurations.

Time to Implement	On Amazon EC2, we were able to bring up our first MongoDB configuration within hours. We then automated this process by creating Chef and Puppet scripts that could spin up dozens of MongoDB nodes in the cloud within minutes.
Administrator Functionality	A two-day training from 10gen provided the operations team the skills needed to evaluate our use case.
Training	Application developers became proficient in MongoDB with just a half day of training.
Data Migration & Movement	We populated MongoDB by exporting data from Oracle and using simple Python scripts (based on well-documented examples from 10gen) to load the data into MongoDB.
Conformity with Company & Industry Standards	While MongoDB is not yet a corporate standard at Apollo, it met the criteria for future inclusion on the standards list. MongoDB's immense momentum relative to all other NoSQL solutions was reassuring.
Quality & Availability of Support	10gen provided invaluable assistance in this evaluation. They have dozens of very knowledgeable and helpful engineers and run a very professional support operation. We are confident in their ability to provide enterprise-grade support, even in the middle of the night or on a long holiday weekend. Further, MongoDB has a very large, active, and helpful community with which we could engage.

Tips for Evaluating MongoDB

The following four factors contributed to our success in doing an accurate, insightful evaluation of MongoDB:

1. **Work with 10gen and the MongoDB community.** We attended the MongoDB conference and 10gen trainings, and engaged 10gen consultants. Their assistance was vital to completing the evaluation within an aggressively short timeframe. 10gen's input was especially helpful when creating our runbook and designing our data model.

2. **Dive in and get your hands dirty.** MongoDB is open source, well documented, and easy to install. Thus, unlike many proprietary databases, there are no barriers to getting started right away. Hands-on experimentation was often the best way to answer our questions.

3. **Use the cloud to get started quickly.** We used Amazon EC2 to spin up development servers within hours, and to test a variety of geographically dispersed configurations.

4. **Identify the right metrics.** For your specific use case, comparing metrics such as input/output operations per second (IOPS) may give misleading results. What is most important is the amount of 'work' accomplished and how quickly it is done for your use case.

Tips for Evaluating a Software Solution

We in the Forward Engineering team at Apollo Group are experts in software evaluation, conducting about 30 POCs/evaluations per year. Here are our tips for running any type of software evaluation process:

- Identify and work closely with all relevant stakeholders.
- Clearly define the problem, your goals, and the areas on which you want to focus.
- Follow a rigorous, template-based evaluation process to ensure that you cover all bases across all POCs.
- Divide the evaluation into short, discrete phases.
- Identify a simple but relevant use case to ensure general understanding of the technical findings.
- Act fast and fail quickly to avoid spending time and resources on a solution that will fail in the long run.
- Leverage formal training, seminars, and targeted questions to vendors.
- Enroll users not involved in the project in training to assess accurately how long it will take to train novice users.
- Evaluate the ease – or difficulty – of training users.

Conclusion

A Challenge to the Community

While our evaluation of MongoDB was sufficient to justify deploying it as part of our platform, there are still numerous areas that call for further research. We invite members of the MongoDB community to expand on our research by evaluating the following situations, sharing their experiences with us at MongoResearch@apollogrp.edu, and communicating the results to the community at large.

In particular, we at Apollo Group are interested in:

» How changing replica sets, indexing, and sharding affects system performance under heavy load, and for configurations with large numbers of nodes.

» How MongoDB performs with queries more complicated than the ones we tested, such as queries that update or delete documents.

Summary

Given the overwhelmingly positive results of our evaluation, Apollo Group decided to move forward with MongoDB. Not only can MongoDB support the anticipated use cases for our platform, but it also functions well with large and critical parts of our architecture. While some aspects of our platform still use a relational database, we use MongoDB whenever possible because it is easy to use, has excellent performance, works well in a variety of cloud and on-premise configurations, and is highly cost-effective.

More information on how to get a MongoDB database up and running is available at the following links:

http://www.10gen.com/reference

http://www.10gen.com/presentations

http://www.mongodb.org/display/DOCS/Home

Appendix

Oracle Query Calls

The table below provides a breakdown of actual query usage over a one-month period for the legacy Oracle solution.

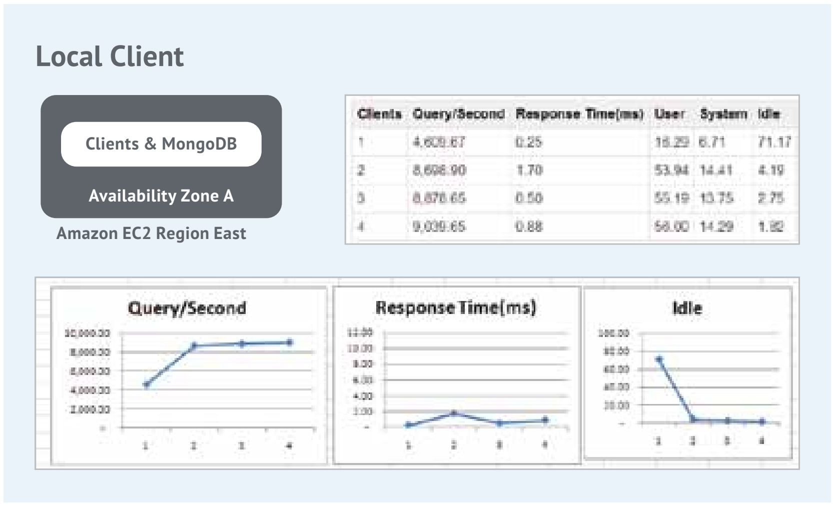
Table 5: Oracle Query Calls

QUERY	EXECUTIONS	PERCENTAGE
Query 1	15,572,099	46%
Query 2	4,339,293	13%
Query 3	3,232,297	10%
Query 4	3,016,176	9%
Query 5	2,541,686	8%
Query 6	2,485,334	7%
Query 7	2,384,839	7%
Query 8	33,571,724	100%

Results: Configuration A – Clients and MongoDB on Same Amazon EC2 Instance

Figure 4 shows the results from load tests in which the test clients and MongoDB were on the same EC2 instance. The instance quickly became CPU-bound with 10,000 queries per second and near-zero response times.

Figure 4: Load Test Results–Clients and MongoDB on Same Amazon EC2 Instance



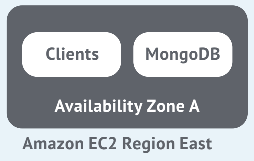
Clients	Query/Second	Response Time(ms)	User	System	Idle
1	4,609.67	0.25	16.29	6.71	71.17
2	8,696.90	1.70	53.94	14.41	4.19
3	8,878.65	0.50	55.19	13.75	2.75
4	9,039.65	0.88	56.00	14.29	1.92

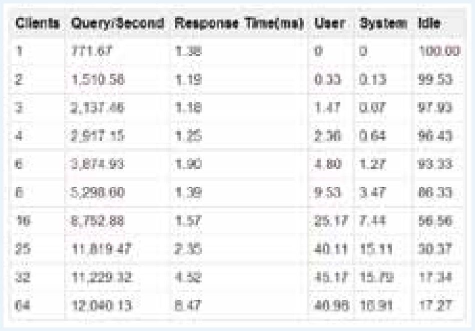
Results: Configuration B – Clients and MongoDB on Separate Amazon EC2 Instances, but within Same Amazon EC2 Availability Zone

Figure 5 shows results from a load test in which the test clients and MongoDB were on separate instances but were still located in the same Amazon EC2 Availability Zone. The instance peaked at 80% CPU utilization, delivering 12,000 queries per second. Response time grew as clients were added, as the instance maxed out at 12,000 queries per second.

Figure 5: Load Test Results–Clients and MongoDB on Separate Instances, but within Same Amazon EC2 Availability Zone

Zone Clients





Clients	Query/Second	Response Time(ms)	User	System	Idle
1	771.67	1.38	0	0	100.00
2	1,510.58	1.19	0.33	0.13	99.53
3	2,137.46	1.18	1.47	0.07	97.93
4	2,917.15	1.25	2.36	0.64	96.43
6	3,874.93	1.90	4.80	1.27	93.33
6	5,298.00	1.39	9.53	3.47	86.33
16	8,752.88	1.57	25.17	7.44	56.56
25	11,819.47	2.35	40.11	15.11	30.37
32	11,229.32	4.52	45.17	15.79	17.34
64	12,040.13	8.47	40.98	16.91	17.27

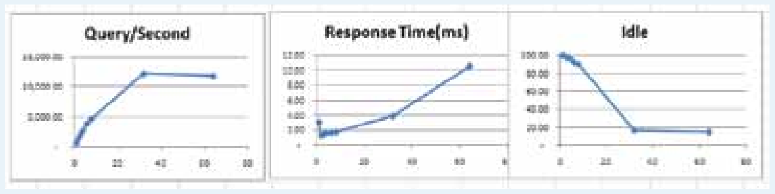
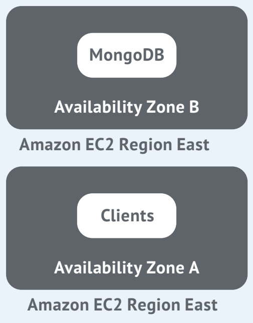
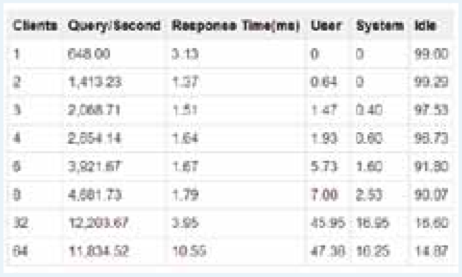


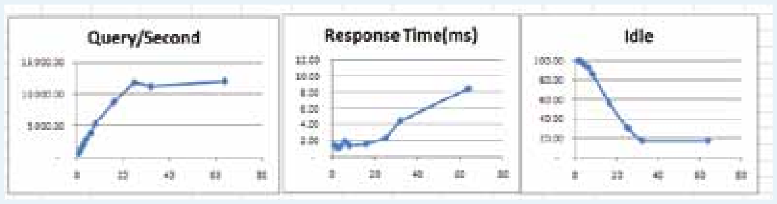
Figure 6: Load Test Results–Clients and MongoDB in Separate Availability Zones, but within One Amazon EC2 Region

Region Clients





Clients	Query/Second	Response Time(ms)	User	System	Idle
1	648.00	3.13	0	0	99.00
2	1,413.23	1.27	0.64	0	99.29
3	2,068.71	1.51	1.47	0.40	97.53
4	2,654.14	1.64	1.93	0.60	98.73
6	3,921.67	1.67	5.73	1.60	91.30
8	4,681.73	1.79	7.00	2.53	90.07
32	12,203.67	3.95	45.95	16.95	15.60
64	11,834.52	10.55	47.38	10.25	14.87



Results: Configuration C – Clients and MongoDB in Separate Availability Zones, but within One Amazon EC2 Region

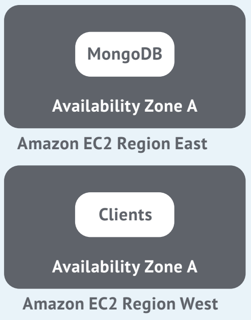
The results from Configuration C were very similar to those of the previous configuration. In this configuration, the clients and MongoDB were on separate instances in separate Amazon EC2 Availability Zones, but within the same Amazon EC2 Region.

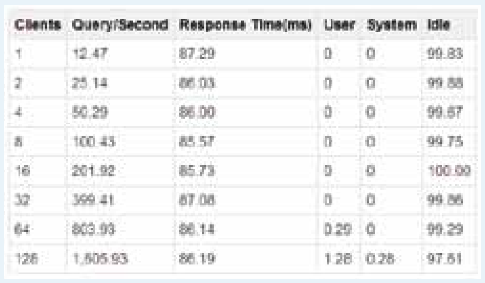
Results: Configuration D – Clients and MongoDB in Different Amazon EC2 Regions

In a load test with clients and MongoDB in different Amazon EC2 Regions, the response time increased by a factor of 50. The network was the bottleneck in this configuration; the instance wasn't stressed, as evidenced by very low CPU utilization, even with 128 concurrent clients.

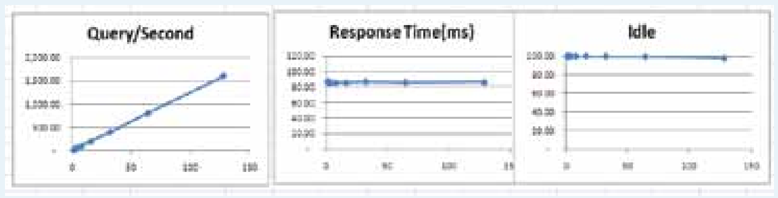
Figure 7: Load Test Results–Clients and MongoDB in Different Amazon EC2 Regions

Different Regions





Clients	Query/Second	Response Time(ms)	User	System	Idle
1	12.47	87.29	0	0	99.83
2	25.14	86.03	0	0	99.88
4	50.29	86.00	0	0	99.67
8	100.43	85.57	0	0	99.75
16	201.92	85.73	0	0	100.00
32	399.41	87.08	0	0	99.86
64	803.93	86.14	0.29	0	99.29
128	1,605.93	86.19	1.28	0.28	97.61



10gen | the MongoDB company

Published by 10gen, Inc. with the permission of the Apollo Group