EE 595 Final Project
6 May 2020; Los Angeles; CA

# A NoSQL Database and CDN for Video Delivery System

Yani Jin (yanijin@usc.edu)

Xinyi Cai (caixinyi@usc.edu)

Ryan Afranji (afranji@usc.edu)

## Problem Description

With the explosive growth of user's requirement for video streaming services and the development of video sharing platforms, media-services providers are seeking an optimal video storage and delivery pattern meeting their own specific demands. From end users' point of view, high availability and low latency are the two main factors they would consider when they choose a video provider, except for the content itself. For a video provider, based on how large of its audience coverage and how many contents it would like to supply, scalability is taken into consideration as well as availability and efficiency. In this paper, we provide a feasible pattern for media-services providers which promises a high performance in the above mentioned three aspects.

## Project Timeline

Phase I - March 4:

1. Came up with a design plan for integrating CDN and Cassandra to provide higher quality service.
2. Converted a local PC into a web server which also connects to Cassandra.
3. Designed the workflow and Cassandra logical data model for our web application.

Phase II - April 10:

1. Successfully made connections among web servers, load balancer and cache proxies.
2. Built a web server connecting to Cassandra database.

Phase III - May 6:

1. Implemented more complex queries on the Cassandra database, and designed a more friendly web page.
2. Established proxies among multiple continents, and added a self-built GeoDNS to the edge of our CDN.
3. Connected all components together forming a complete workflow.

EE 595 Final Project
6 May 2020; Los Angeles; CA

# A NoSQL Database and CDN for Video Delivery System

***Abstract*—End users of the Internet always want to access Internet contents faster and more stable. With the explosive growth of the variety of contents on the Internet as well as user's requirements for content delivery, internet service providers choose to use CDN to offer a better Internet service. Most websites and Internet contents that people interact with are spreading over multiple physical locations. The physical distance between the origin server and the end user will cause large transmission delay. In assistance of CDN, user's experience in terms of speed would be highly increased. Apache Cassandra database could be the right choice of database in terms of scalability and high availability without compromising performance. It supports replicating across multiple datacenters and providing lower latency at the same time. There are already many successful examples in industry using Cassandra effectively. In this paper we will discuss how to build our CDN with Cassandra database and provide content delivery service with high quality.**

***Keywords—Cassandra; Nginx; Load Balancing; Caching; Geo-aware DNS.***

## I. Introduction and Background

The internet has become one of the most important factors of human life. Every day, almost every human being has a large interaction with the Internet. One trend that continues year after year is in the increase of data consumption. A large cause of this is attributed to streaming services. There is a massive migration of people ditching their cable/satellite television providers to get their media through internet based streaming service. Another trend is that camera quality is increasing so the size of files to be distributed by these streaming services are also getting larger.

To handle significant network traffic on streaming services sites, they need to handle requests to their distributed systems in an effective way. This is done through the use of Content Delivery Networks (CDNs) and databases. An efficient tool to establish a CDN is Nginx, which is utilized in our system. When developers are making decisions on which database to use in the case of distributed systems (CDN could be regarded as a distributed system), they refer to CAP theorem and think about desired features of their system first. For an application providing a world wide streaming service, it must make sure users from all over the world could access it. Also, even numerous videos getting uploaded to it, it could handle the increase of video source easily with low cost. In a word, high availability and partition-tolerance are essential to streaming service. This is the reason why we choose Cassandra as our database.

As we've mentioned, the other essential factor which could accelerate the video streaming service is CDN. A carefully designed CDN could not only deliver content to end users quickly and efficiently but also handle more traffic coming from clients. By utilizing techniques such as load balancing and caching, a CDN could distribute incoming requests and deliver cached content if applicable, without adding traffic going through web servers.

This paper will analyze how the system we built to meet these demands. In section II, the framework of the problem will be analyzed. This includes the problem description, challenges, and analysis of our solution. In section III, the implementation is described followed by experiments performed on the system and its results. Lastly, section IV concludes the paper with an analysis of ten papers related to this topic.

## II. Framework

### *A.* Formal Problem Description

With the explosive growth of user's requirement for video streaming services and the development of video sharing platforms, media-services providers are seeking an optimal video storage and delivery pattern meeting their own specific demands. From end users' point of view, high availability and low latency are the two main factors they would consider when they choose a video provider, except for the content itself. For a video provider, based on how large of its audience coverage and how many contents it would like to supply, scalability is taken into consideration as well as availability and efficiency.

### *B.* Challenges of the Problems

The first challenge is to figure out where we should build our system. As we came up with the system design, we found there would be more than four machines, including web servers, load balancer, proxies and name servers. Due to the need of placing proxies in different locations, we even have to ensure that we are capable of deciding where our machine is Geographically. We finally chose to develop the whole system on AWS EC2 instances after comparing the feasibility and cost of multiple plans.

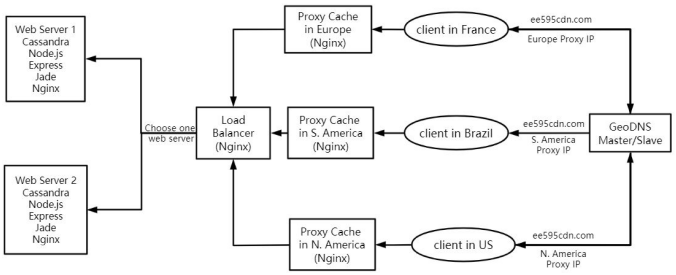
Since we planned to build every component from scratch, the next challenge we met was learning new concepts and technologies related to each of them. For example, we compared a lot of proxy software just to find one fitting our requirements and chose Nginx in the end. Even for a simple

configuration, we had probably done a lot of research to make a sensible decision.

Another challenge was integrating so many pieces of hardware devices, debugging them and taking performance tests on them as a union. Unlike other course work we have done, this project covers various roles in a computer network, from backend to frontend, from web servers to DNS. Because of this characteristic, we spent plenty of time figuring out how components talk to each other.

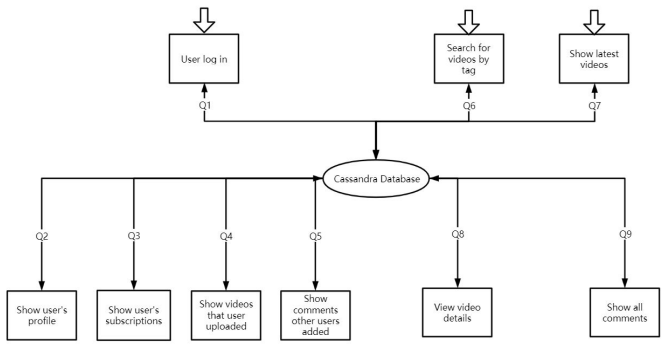
### C. Analysis of Our Solution

As shown in the figure below, our system consists of two identical web servers, one load balancer, three proxies in different continents and a geo-aware domain name system, all of which running on AWS EC2 instances.




When a client makes a request to our domain, our GeoDNS will direct it to its nearest proxy. For example, if a client is in Europe, he will be directed to the proxy sitting in Sweden. In default cases, i.e., if the client is in a continent that doesn't have our proxy, his request will go to the proxy in North America automatically. Once requests have reached their matching proxies, those proxies will deliver valid content to the client if they have, otherwise, they will go fetch the valid response from their upstream server, the load balancer. Until now, the load balancer will make a decision on to which web server the request will go. And, finally, the web server will respond to the request.

As for the database design behind web servers, we have some queries here. This database is designed on a Cassandra. The user could send requests to Cassandra database from using different queries, like finding the user by email address when the user logs in, searching for videos by tag and showing the latest videos when the user opens the website.




In the load balancer, the default algorithm is round robin. It sends requests to the backend servers in a rotational fashion. It is one of the simplest methods for delivering requests. Round robin load balancers would assume that all servers are basically the same from every aspect. This algorithm might not be very sensible when the server in order still has a lot of connections waiting to be answered. Based on Nginx's customers' feedback, we change the algorithm to least connections. Least connections algorithm also effectively distributes workload across servers according to their server remaining capacity. A more powerful server fulfills requests more quickly, so at any given moment it's likely to have a smaller number of connections still being processed. Least Connections sends each request to the server with the smallest number of current active connections, and so is more likely to send requests to powerful servers. With the least connection method, the load balancer will compare the current number of active connections among those available servers. After that, the load balancer will send the request to the server which has the least active connections.

As for proxy caches, two key issues are considered; one is where and in what architecture to store the cache content in proxies, the other is specifying the cache policies. For the first one, we need to create a directory for cached responses, allocate a specific amount of memory for Shared Memory Space, specify directory structure levels, define a period of time after which we should release the cached response, limit the space for cached response. For the other, setting cache validity, content refreshing and when to deliver stale content is needed.

A fundamental DNS contains a pair of master name server and slave name server. To make a DNS geo-aware, these two name servers should also be deployed as split horizon name servers. With the help of BIND's view clause, BIND based name servers are capable of achieving the functionality of geographic mapping and being so called split horizon name servers.

## III. Implementation and Experimental Results

In this section, we'll talk about how the whole system designed and implemented in detail. By the end of this section, we display the performance of our system in the aspects of time consuming.

### A. Experimental Setup

For the webserver, many packages were used to set it up. First and most importantly, Cassandra was used as the database of this project. The packages used to support the webpage were installed using NPM. Express was used to generate a web server template to create the website. The template generated a basic Javascript server and a basic Jade file to style the webpage. These files were modified to complete the desired tasks of the webpage. Cassandra-driver was used in the javascript file to contact and make queries to the Cassandra database. This webserver ran through the use of Node.js. Lastly, this webpage runs on the local host, so Nginx

is used to forward this web server to a publicly accessible IP address.

As for the load balancer, it is built on Nginx. It balances the traffic between multiple web servers. We need to set some parameters in the file sitting in Nginx sites-available, like algorithm, IP address, listening port.

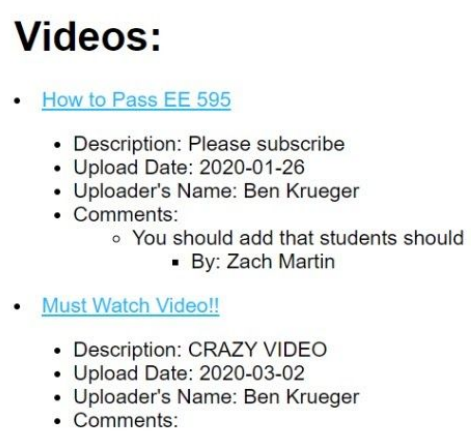
We also built our cache proxies in different continents using Nginx. Besides determining proxies' upstream server, the most important portion is to define how proxies cache content on the way of videos flowing to end users. Cache policies will be explained in software implementation.

As for the domain name systems, we built master and slave name servers on BIND. An updated GeoIP access control list is necessary for name servers to enable geographic mapping. As we need to take a performance test on our system, a real domain name is required. Hence, we bought a domain name called ee595cdn.com and modified the DNS configuration to manage name servers on our own instead of under the control of the domain name seller.

### B. Software Implementation

For the webserver, the implementation starts with populating the Cassandra database. First, a keyspace called *library* was created to store all the tables. All the tables include the tables listed in the flowchart above that diagrams the database. The command *CREATE TABLE IF NOT EXISTS* created new tables and *INSERT INTO* was used to populate the tables with the appropriate rows. Each EC2 instance hosting the web server had a copy of this database. Cassandra ran locally on these machines allowing the web server to contact it in a low latency manner as opposed to the higher latency operation of having to access it via the network.

The different components needed to support a basic website were generated using Express. After this generation, the file index.js stores the Javascript file that runs the backend web server of our main webpage. The first thing done in this webserver is set up a connection to the cassandra database. It takes the ip address (127.0.0.1 because it is running locally) and its unique name to establish the connection. Once connected, a variable called *client* stores an object that can be used to make queries to the database via the command *execute*. A query is first made to get the list of videos in the database. Next, another query is made to associate the *user_id* of the video's uploader with their first and last name which is stored in a different table from the video table. Lastly, another query is made to load the comments of the video and it also makes the same query as above to associate the commentor's *user_id* with their first and last name stored in a different table. During each of these queries, their needed output data is added to the data structure that was outputted from the video query. Thus, all the data needed for the webpage can be found in this data structure.




The above screenshot of the webpage is the result of the queries. This page was assembled using Jade. The data from the data structure in the previous paragraph is passed to the file *index.jade*. *index.jade* is passed to a client's web browser and explains how to load the web page on it. First thing this file does is list all the videos from the data structure. As shown above the two videos, "How to Pass EE 595" and "Must Watch Video!!" are listed. Under each of the listings, their description and upload date are listed. Next, the uploader's name is listed. Lastly, if the video has any comments, the comments will be listed with the name of the person who created the comment.

*npm start* is ran on the prompt of the EC2 instance to begin running this web server. This will run the web server on the localhost of the instance on the default port of 3000. Nginx is then configured to set up a *proxy_pass* to pass the webserver through the localhost on port 3000 to the publicly accessible ip address of port 80. The web server can now be accessed externally via its public ip address.

For the load balancer, we need to make some configuration in the Nginx file. The load balancer is running on an EC2 instance to provide a public IP address for others. Load balancer here is used to balance the traffic among multiple clients. And in our case, the traffic is given from the proxies or the clients. In the *upstream backend server* block, we defined the algorithm used in this load balancer as well as the IP addresses of the backend servers. In the *server* block, we defined the listening port 80 which is default for HTTP traffic. *Proxy_pass* is used to record the path to the backend server.

We have three proxies in total, they are located in North America, South America and Europe respectively. For clients in continents other than these three, they will be directed to the proxy in North America by default.

When configuring our proxies using nginx, there are 4 things that need to be set up. Firstly, at least, proxies need to know their upstream server, i.e. the load balancer. Second one is to set up a shared memory zone, which stores the cache keys and metadata such as usage timers. This zone is set to be a 2 level directory with a size of 10 megabytes in total which

could store 80,000 keys. Thirdly, we need a place to store the cached content. For each proxy, we set the cache size to 1 Gigabyte. As youtube statistic shows, the average video duration is 4.3 minutes. For a standard quality video, i.e., 480p, it will be 36 megabytes per video. Hence our proxy could cache roughly 300 videos. The last one is cache policies. We only cached the content if the response is a success or redirection, which means the http status code starts with 2 or 3. Any item could remain in the cache without being accessed in an hour. After that, the cached content will be erased from disk. If the content is not fresh anymore but is still in the disk, our proxy will go fetch data from the original server automatically. However, there might be a server error or connection error on its way to fetch the new fresh content. In this case, our proxies will temporarily use the stale content they have.

For our geo-aware domain name system, we use BIND to build a master and a slave name server. To connect these two, we allow master to transfer zone files to slave and enable salve to get updated zone files from master.To make DNS geo-aware, firstly, we got GeoIP access control list online. Basically, the ACL defines names for different IP ranges. For example, it notates IP addresses that belong to Mexico as MX. Then in the BIND configuration, view clauses are designed to direct requests coming from different countries to different proxies. Take North America as an example, we matched requests coming from the US, Canada and Mexico to the proxy sitting in North America. For simplicity purposes, we didn't include all countries in North America.Last but not the least, zone files in name servers contain our resource records. And the most important record inside this file is to map our domain to the IP address of a proper proxy. At this point, the DNS servers introduce themselves to the world saying they are authoritative to the domain name ee595cdn.com.

### C. *Experimental Results*

This following chart displays some parameters when accessing the target website using the CDN designed in this project. This test is based on the situation when we type an URL into a browser. The web performance test analyzes the load time of a single asset, when we type in ee595cdn.com, it returns a breakdown of some attributes. Here it estimates parameters like status code, DNS resolve time and Connect time. The status code returns the HTTP status code, which is 200 in this chart. DNS here means the lookup time the system used to return from the domain name system. Connect is the time it takes to connect to the web server. People normally regard connection time under 200 ms as a good result.

| Location | Status | DNS | Connect |
|---|---|---|---|
| Frankfurt | 200 | 4.3 ms | 155.3 ms |
| New York | 200 | 6.1 ms | 76.3 ms |
| Miami | 200 | 127.8 ms | 72.3 ms |
| Dallas | 200 | 127.7 ms | 47.3 ms |
| San Francisco | 200 | 28.6 ms | 3.8 ms |
| Seattle | 200 | 63.5 ms | 26.9 ms |
| Toronto | 200 | 17.6 ms | 70.3 ms |
| London | 200 | 253.1 ms | 26.3 ms |
| Paris | 200 | 255.9 ms | 159.8 ms |
| Amsterdam | 200 | 253.3 ms | 151.4 ms |

This following chart shows the IP addresses of the proxies when the clients want to have access to the web server. Since the clients sit in different locations around the world, they will access a different proxy cache. The way they choose a proxy cache is based on a Geo-aware method.

| Location | IP |
|---|---|
| Frankfurt | 13.53.85.246 |
| New York | 54.177.253.119 |
| Miami | 54.177.253.119 |
| Dallas | 54.177.253.119 |
| San Francisco | 54.177.253.119 |
| Seattle | 54.177.253.119 |
| Toronto | 54.177.253.119 |
| London | 54.177.253.119 |
| Paris | 13.53.85.246 |
| Amsterdam | 54.177.253.119 |

Chart below shows the comparison between two cases. The first row is the situation where CDN combined with Geo-aware DNS, the second row is the situation where CDN works without Geo-aware DNS. With the help of Geo-aware DNS, the client's request could be directed to the closest proxy cache, while without it the request will be directed to a default proxy.

DNS resolve time will affect the time it takes for the host server to receive and process a request for a webpage. If the DNS server information is available in browser cache, or if the DNS name server is available at a location close to the client, then the process is relatively simple. DNS connect time indicates the time it takes for the client to get connected with the web server. Here we could see that when the CDN has

Geo-aware DNS, the connection time is much shorter than the one without it. And this could also help with download time. Since these two cases are different in routing, they will access different IP addresses as well.

| | Total time(ms) | Resolve time(ms) | Connect time(ms) | Download time(ms) | IP address |
|---|---|---|---|---|---|
| Client | 261 | 155 | 35 | 71 | 13.53.85.246 |
| | 605 | 155 | 146 | 304 | 13.52.240.150 |

We also make comparisons among the experiment results with other commercialized CDNs. On the average, the commercialized CDNs listed here will have a latency around 30.9 ms. From the chart below, some of our clients get a better result than these commercialized CDNs while others are not. The difference is mainly caused by the cache in the browser and DNS resolution time.

| Service | Latency(ms) |
|---|---|
| Azure CDN from Verizon | 26.5 |
| CacheFly CDN | 22 |
| CloudFlare | 29 |
| Akamai CDN | 37.5 |
| Alibaba Cloud CDN | 39.5 |

## IV. Readings

*1) Survey on NoSQL Database*

This paper begins by listing some of the needs of modern databases. These include, the need to handle concurrent reads and writes in a low latency manner, the ability to scale up efficiently, and the need to be highly available. Traditional relational databases are slow, have a difficult time scaling, and have limited capacity. NoSQL was developed to solve these problems. This paper then provides summaries for different NoSQL databases that fall under these categories: key-value databases, column-oriented databases, and document databases. The paper concludes by stating that each of these databases have their niche benefits and companies should consider these benefits to pick the optimal database.

*2) NoSQL Databases: MongoDB vs Cassandra*

The paper has a pretty self-explanatory name; it provides a comparison of MongoDB and Cassandra. The paper begins by going into depth of why relational databases are not able to handle modern uses well and that NoSQL databases solve relational databases' shortcomings. Due to the nature of both being NoSQL databases, MongoDB and Cassandra share many similarities. The differences between the two include MongoDB being more optimized for consistency and partition tolerance while Cassandra is more optimized for consistency and availability. MongoDB replicates using a master-slave technique while Cassandra uses peer-to-peer.

*3) NoSQL Database Design Using UML Conceptual Data Model Based on Peter Chen's Framework*

A problem with NoSQL databases in large-scale systems is that it is hard to have consistent data management. This paper presents a conceptual data model for building databases. This conceptual data model is independent of which database you are using (can be applied to relational, NoSQL, etc.) The details of it are too much to include in this summary but this paper provides ways of organizing a database and provides an example of an e-commerce's business database and the queries made to it.

*4) Node.js Paradigms and Benchmarks*

Traditionally, web servers used thread-based solutions to handle their increase in traffic. This had issues though which included it being difficult to program for non expert programmers and synchronization. This was solved by using event-based solutions which greatly increased performance by doing things such as implementing non-blocking I/O operations in the form of callbacks. Node.js is an event-based model. Based on the results in their experiments, Node.js performed much better than Apache, which is an implementation of the thread-based model.

*5) Improving Response Time for Cassandra with Query Scheduling*

This paper analyzes databases that use distributed Key Value Stores supporting databases. It defines range queries as a key that returns multiple values and a single query as one that returns one value per query. The paper identifies that range queries are slower and require more resources thus slowing down the response time of single queries. This paper achieves better response time for single queries by prioritizing scheduling them to execute before range queries. They also were able to parallelize the range queries to improve their response time as well. This project was built in Cassandra.

*6) Improving Network Scalability Using NoSql Database*

NoSQL databases scale much better over commodity hardware as opposed to traditional databases. Older databases handle scaling by using more advanced hardware; this technique had its limitation to scaling so the NoSQL commodity hardware approach was adopted. Cloud services such as Facebook used NoSQL databases because they are

able to sacrifice strong consistency for improved responsiveness for their clients. Another benefit that improves scaling is that some NoSQL databases such as Cassandra are implemented in such a way that they are able to avoid having significant amounts of empty blocks in the databases thus avoiding taking up memory for unused data.

*7) Improving Database System Performance by Applying NoSQL*

This paper compares relational database management systems (RDBMS) with NoSQL. RDBMS focuses on providing ACID (atomicity, consistency, isolation, durability). These properties are great for when the data must be strongly consistent and durable but ensuring these properties degrades performance. For services that can afford to relax these properties, they can boost performance of their system. NoSQL chooses to relax these properties. Testing proves that not only does NoSQL perform better, but in some cases it offers an extraordinary performance boost.

*8) Improving Database Performance on Simultaneous Multithreading Processors*

As simultaneous multithreading processors share processor resources, different from the regular multiprocessor systems, developers have to take advantage of new technologies to implement databases. This paper explores three methods to deploy databases on SMT and evaluate each of them. First one is distributing independent operations to threads, so that sharing resources won't make a mess. Second one is implementing operators in a multithread manner. The last one makes use of a work-ahead set to ensure data needed by other threads are cached to memory in advance.

*9) Content Delivery Networks Status and Trends*

This paper explains the current status and future developing trends of CDN. At the beginning, CDN was firstly designed for the reason that the web cannot handle large content transmission among large networks. With the web's growth, CDNs could improve network performance as well as offer fast and stable services. This result could be achieved by distributing content to the cache servers. These servers are generally sitting close to the end users. CDN could reduce the workload of the original server, reduce the latency for clients and improve the network stability at the same time.

*10) Managing Service Performance in the Cassandra Distributed Storage System*

In this paper, the authors mainly explain the details of quality-of-service infrastructure in Cassandra distributed storage systems. The implementation of this design is on Amazon EC2 Cloud. The architecture in this paper is able to handle the dynamic adaptation problem. It is solved by monitoring service performance. Using targeted measurement to produce some tables to deliver the benchmark performance over different configurations of Cassandra. Although this architecture is built on the basis of Cassandra context, its method could also be applied to other NoSQL systems.