



Evaluation of YouTube's Design

Let's understand how our design decision fulfills the requirements.




We'll cover the following

- Fulfilling requirements
- Trade-offs
 - Consistency
 - Distributed cache
 - Bigtable versus MySQL
 - Public versus private CDN
 - Duplicate videos
- Future scaling
- Web server

Fulfilling requirements

Our proposed design needs to fulfill the requirements we mentioned in the previous lessons. Our main requirements are smooth streaming (low latency), availability, and reliability. Let's discuss them one by one.

1. **Low latency/Smooth streaming** can be achieved through these strategies:

- Geographically distributed cache servers at the ISP level to keep the most viewed content. 
- Choosing appropriate storage systems for different types of data. 
For example, we'll can use Bigtable for thumbnails, blob storage for 

videos, and so on.

- Using caching at various layers via a distributed cache management system.
- Utilizing content delivery networks (CDNs) that make heavy use of caching and mostly serve videos out of memory. A CDN deploys its services in close vicinity to the end users for low-latency services.

2. **Scalability:** We've taken various steps to ensure scalability in our design as depicted in the table below. The horizontal scalability of web and application servers will not be a problem as the users grow. However, MySQL storage cannot scale beyond a certain point. As we'll see in the coming sections, that may require some restructuring.
3. **Availability:** The system can be made available through redundancy by replicating data to as many servers as possible to avoid a single point of failure. Replicating data across data centers will ensure high availability, even if an entire data center fails because of power or network issues. Furthermore, local load balancers can exclude any dead servers, and global load balancers can steer traffic to a different region if the need arises.
4. **Reliability:** YouTube's system can be made reliable by using data partitioning and fault-tolerance techniques. Through data partitioning, the non-availability of one type of data will not affect others. We can use redundant hardware and software components for fault tolerance. Furthermore, we can use the heartbeat protocol to monitor the health of servers and omit servers that are faulty and erroneous. We can use a variant of consistent hashing to add or remove servers seamlessly and reduce the burden on specific servers in case of non-uniform load.

Quiz



Question

Isn't the load balancer a single point of failure (SPOF)?

Show Answer ▼

| Requirements | Techniques |
|--------------|--|
| Scalability | <ul style="list-style-type: none"> • Load balancers to multiplex between servers. • Ability to horizontally add (or remove) web servers according to our current needs. • Addition of multiple storage units specific to the required types of data. • Serving from different colocation sites and CDNs. • Separating read/write operations on different servers. |
| Availability | <ul style="list-style-type: none"> • Replication of content on different sites. • Important data is persisted on replicated data stores so that we could re-spawn the service in case of major disruption or failures. • Using local and global load balancers. |
| Performance | <ul style="list-style-type: none"> • Lighttpd for serving videos/static content. • Caching at each layer (file system, database, cluster, application server, web server). • Addition of multiple storage units specific to the required types of data. • CDNs. • Using an appropriate programming language to perform specific tasks—for example, using C for encryption and Python otherwise. |

How YouTube achieves scalability, availability, and good performance

Trade-offs

Let's discuss some of the trade-offs of our proposed solution.

Consistency

Our solution prefers high availability and low latency. However, strong consistency can take a hit because of high availability (see the [CAP theorem](#)). Nonetheless, for a system like YouTube, we can afford to let go of strong consistency. This is because we don't need to show a consistent feed to all the users. For example, different users subscribed to the same channel may not

see a newly uploaded video at the same time. It's important to mention that we'll maintain strong consistency of user data. This is another reason why we've decoupled user data from video metadata.



Distributed cache

We prefer a distributed cache over a centralized cache in our YouTube design. This is because the factors of scalability, availability, and fault-tolerance, which are needed to run YouTube, require a cache that is not a single point of failure. This is why we use a distributed cache. Since YouTube mostly serves static content (thumbnails and videos), Memcached is a good choice because it is open source and uses the popular Least Recently Used (LRU) algorithm. Since YouTube video access patterns are long-tailed, LRU-like algorithms are suitable for such data sets.

Bigtable versus MySQL

Another interesting aspect of our design is the use of different storage technologies for different data sets. Why did we choose MySQL and Bigtable?

The primary reason for the choice is performance and flexibility. The number of users in YouTube may not scale as much as the number of videos and thumbnails do. Moreover, we require storing the user and metadata in structured form for convenient searching. Therefore, MySQL is a suitable choice for such cases.

However, the number of videos uploaded and the thumbnails for each video would be very large in number. Scalability needs would force us to use a custom or NoSQL type of design for that storage. One could use alternatives to GFS and Bigtable, such as HDFS and Cassandra.

Public versus private CDN



Our design relies on CDNs for low latency serving of the content. However, CDNs can be private or public. YouTube can choose between any one of the two options.

>

This choice is more of a cost issue than a design issue. However, for areas where there is little traffic, YouTube can use the public CDN because of the following reasons:

1. Setting up a private CDN will require a lot of CAPEX.
2. For rather little viral traffic in certain regions, there will not be enough time to set up a new CDN.
3. There may not be enough users to sustain the business.

However, YouTube can consider building its own CDN if the number of users is too high, since public CDNs can prove to be expensive if the traffic is high. Private CDNs can also be optimized for internal usage to better serve customers.

Duplicate videos

The current YouTube design doesn't handle duplicate videos that have been uploaded by a user or spammers. Duplicated videos take extra space, which leads to a trade-off. As a result, we either waste storage space or face an additional complexity to the upload process for handling duplicate videos.

Let's perform some calculations to resolve this problem. Assume that 50 out of 500 hours of videos uploaded to YouTube are duplicates. Considering that one minute of video requires 6 MB of storage space, the duplicated content will take up the following storage space:

?

```
(50 x 60) minutes x 6 MB/min = 18 GB
```

Tt

If we avoid video duplication, we can save up to 9.5 petabytes of storage space in a year. The calculations are as follows:

C

```
18 GB/min * (60 * 24 * 365) total minutes in an year = 9.5 Peta Bytes
```

>

Storage space being wasted, and other computational costs are not the only issues with duplicate videos. An important aspect of duplicate videos is the copyright issue. No content creator would want their content plagiarized. Therefore, it's plausible to add the complexity of handling duplicate videos to the design of YouTube.

Duplication can be solved with simple techniques like locality-sensitive hashing. However, there can be complex techniques like Block Matching Algorithms (BMAs) and phase correlation to find duplications. Implementing this solution can be quite complex in a huge database of videos. We may have to use technologies like artificial intelligence (AI).

Future scaling

So far, we've focused on the design and analysis of the proposed design for YouTube. In reality, the design of YouTube is quite complex and requires advanced systems. In this section, we'll focus on the pragmatic structure of data stores and the web server.

We'll begin our discussion with some limitations in terms of scaling YouTube. In particular, we'll consider what design changes we'll have to make if the traffic load to our service goes up by, say, a few folds.

We already know that we'll have to scale our existing infrastructure, which includes the below elements:

- Web servers
- Application servers
- Datastores
- Placing load balancers among each of the layers above
- Implementing distributed caches

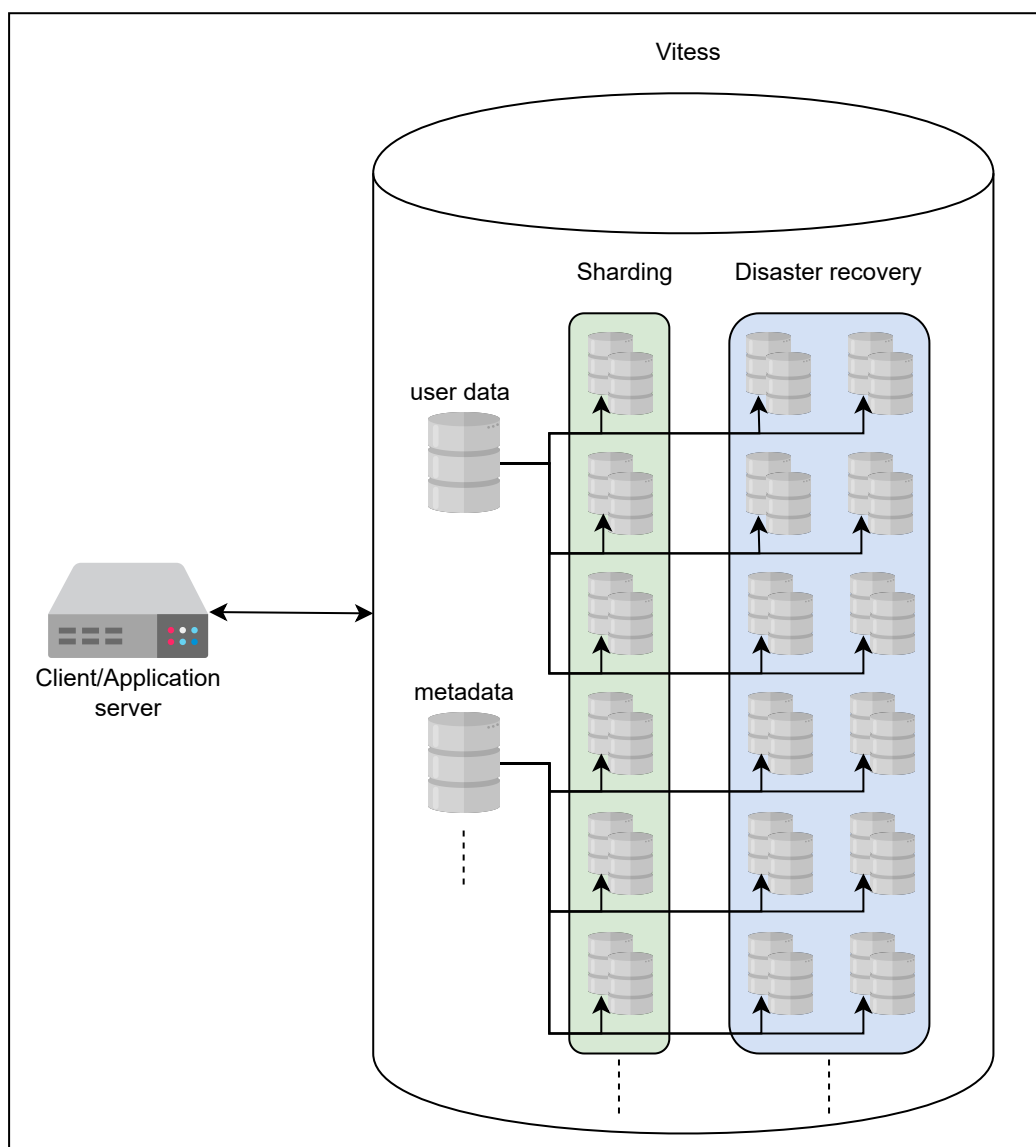
?

Tt

☾

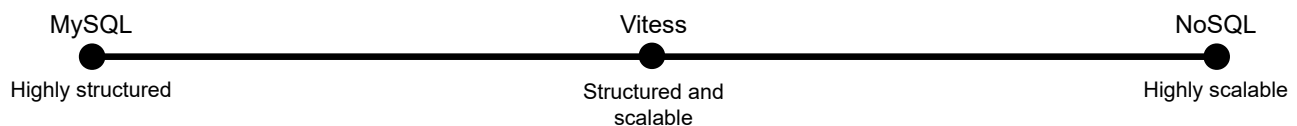
Any infrastructure mentioned above requires some modifications and adaptation to the application-level logic. For example, if we continue to increase our data in MySQL servers, it can become a choke point. To effectively use a sharded database, we might have to make changes to our database client to achieve a good level of performance and maintain the ACID (atomicity, consistency, isolation, durability) properties. However, even if we continue to change to the database client as we scale, its complexity may reach a point where it is no longer manageable. Also note that we haven't incorporated a disaster recovery mechanism into our design yet.

To resolve the problems above, YouTube has developed a solution called Vitess.



Vitess system for scalability

The key idea in Vitess is to put an abstraction on top of all the database layers, giving the database client the illusion that it is talking to a single database server. The single database in this case is the Vitess system. Therefore, all the database-client complexity is migrated to and handled by Vitess. This maintains the ACID properties because the internal database in use is MySQL. However, we can enable scaling through partitioning. Consequently, we'll get a MySQL structured database that gives the performance of a NoSQL storage system. At the same time, we won't have to live with a rich database client (application logic). The following illustration highlights how Vitess is able to achieve both scalability and structure.



Vitess on the scalability versus structure spectrum

One could imagine using techniques like data denormalization instead of the Vitess system. However, data denormalization won't work because it comes at the cost of reduced writing performance. Even if our work is read-intensive, as the system scales, writing performance will degrade to an unbearable limit.

Web server

A **web server** is an extremely important component and, with scale, a custom web server can be a viable solution. This is because most commercial or open-source solutions are general purpose and are developed with a wide range of users in mind. Therefore, a custom solution for such a successful service is desirable.

Let's try an interesting exercise to see which server YouTube currently uses. Click on the terminal below and execute the following command:


```
lynx -head -dump http://www.youtube.com | grep ^Server
```

Terminal 1



Terminal

Loading...

Note: ESF is a custom web server developed by Google, and as of early 2022, it is widely in use in the Google ecosystem because out-of-the-box solutions were not enough for YouTube's needs.

[← Back](#)[Next →](#)

Design of YouTube

The Reality Is More Complicated



Mark as Completed



