




# Design Twitter

Oliver



# 纯属抛砖引玉

- 欢迎大牛补充！
- Disclaimer: 不做商业用途。本整理纯属用于非盈利学习。部分资料来源于互联网。



花花酱



huahualeetcode

# System Design



## Twitter

大家好, 今天做客来讲一下一道经典的

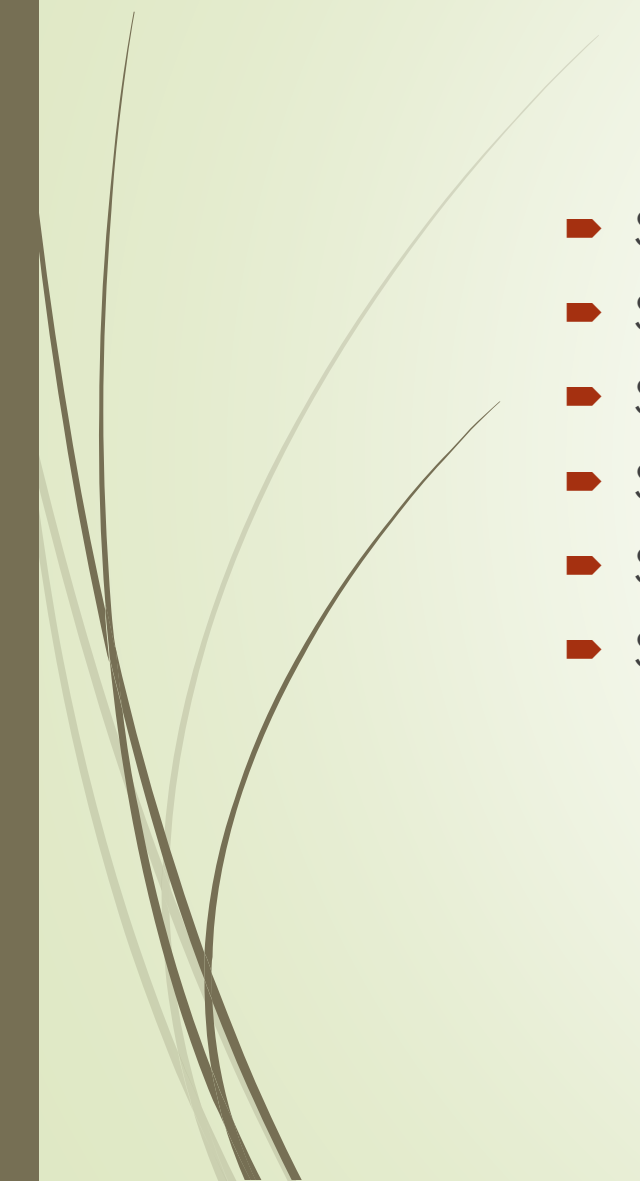


# Interview Signals

- Working solution (过滤掉那压根都行不通的，不work的。)
- Analysis & communication (过滤掉埋头乱写，拉都拉不回来的。)
- Trade-off Pros/ Cons(过滤掉读死书的。招灵活变通，经验丰富，见招拆招的。)
- Knowledge Base(摸底过滤掉一知半解的。招深不可测的。)



# Overview (skip)

- Step 1: Clarify the requirements
  - Step 2: Capacity estimation
  - Step 3: APIs
  - Step 4: High-level System Design
  - Step 5: Data Storage
  - Step 6: Scalability
- 



# Step 1: Clarify the requirement

- ▶ Type 1: Functional Requirement 发推等各种tangible的功能。
- ▶ Tweet
  - ▶ Create
  - ▶ Delete
- ▶ Timeline/ Feed
  - ▶ Home
  - ▶ User
- ▶ Follow a user
- ▶ Like a tweet
- ▶ Search tweets

# Step 1: Clarify the requirement

- ▶ Type 2: Non-Functional Requirement 通用的，可用性等intangible的功能。
- ▶ Consistency 一致性
  - ▶ Every read receives the most recent write or an error
  - ▶ Sacrifice: eventual-consistency. (慢一点不会死人。)
- ▶ Availability 可用性
  - ▶ Every request receives a (non-error) response, without the guarantee that it contains the most recent write
  - ▶ Scalable 是重点，由scalability才能保证availability。
    - ▶ Performance: low latency
- ▶ Partition tolerance (Fault Tolerance) 分区容错
  - ▶ The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.



## Step 2: Capacity Estimation

- 初步Assumption:
- 200 mil DAU, 100 mil new tweets
- Each user: visit home timeline 5 times; other user timeline 3 times
- Each timeline/ page has 20 tweets
- Each tweet has size 280 (140 characters) bytes, metadata 30 bytes
  - Per photo: 200 KB, 20% tweets have images.
  - Per video: 2MB, 10% tweets have video, 30% videos will be watched.



## Step 2: Capacity Estimation

- Storage Estimate
- Write Size Daily:
  - Text:
    - $100 \text{ M new tweets} * (280 + 30) \text{ Bytes/ tweet} = 31 \text{ GB / day}$
  - Image:
    - $100 \text{ M new tweets} * 20 \% \text{ has image} * 200 \text{ KB / image} = 4 \text{ TB / day}$
  - Video:
    - $100 \text{ M new tweets} * 10 \% \text{ has video} * 2 \text{ MB / video} = 20 \text{ TB / day}$
- Total
  - $31 \text{ GB text, } 4 \text{ TB} + 20 \text{ TB} = 24 \text{ TB S3 Objects}$

## Step 2: Capacity Estimation

- Bandwidth Estimate
- Daily Read Tweets Volume:
  - $200M * (5 \text{ home visit} + 3 \text{ user visit}) * 20 \text{ tweets/ page} = 32B \text{ tweets/ day}$
- Daily Read Bandwidth:
  - Text:  $32 \text{ B} * 280 \text{ bytes} / 86400 = 100 \text{ MB/s}$
  - Image:  $32 \text{ B} * 20\% * 200 \text{ KB per image} / 86400 = 14 \text{ GB/s}$
  - Video:  $32 \text{ B} * 10\% * 2 \text{ MB per video} / 86400 * 30\% \text{ got watched} = 21 \text{ GB/s}$
  - Total: 36 GB/s



## Step 3: System APIs

- 帮助我们确认需求都cover到了。
- `postTweet(userToken, String tweet)`
- `deleteTweet(userToken, String tweetId)`
- `likeOrUnlikeTweet(userToken, String tweetId, bool like)`
- `readHomeTimeline(userToken, int pageSize, opt String pageToken)`
- `readuserTimeline(userToken, String targetUserId, int pageSize, opt String pageToken)`

## Step 4: High-level

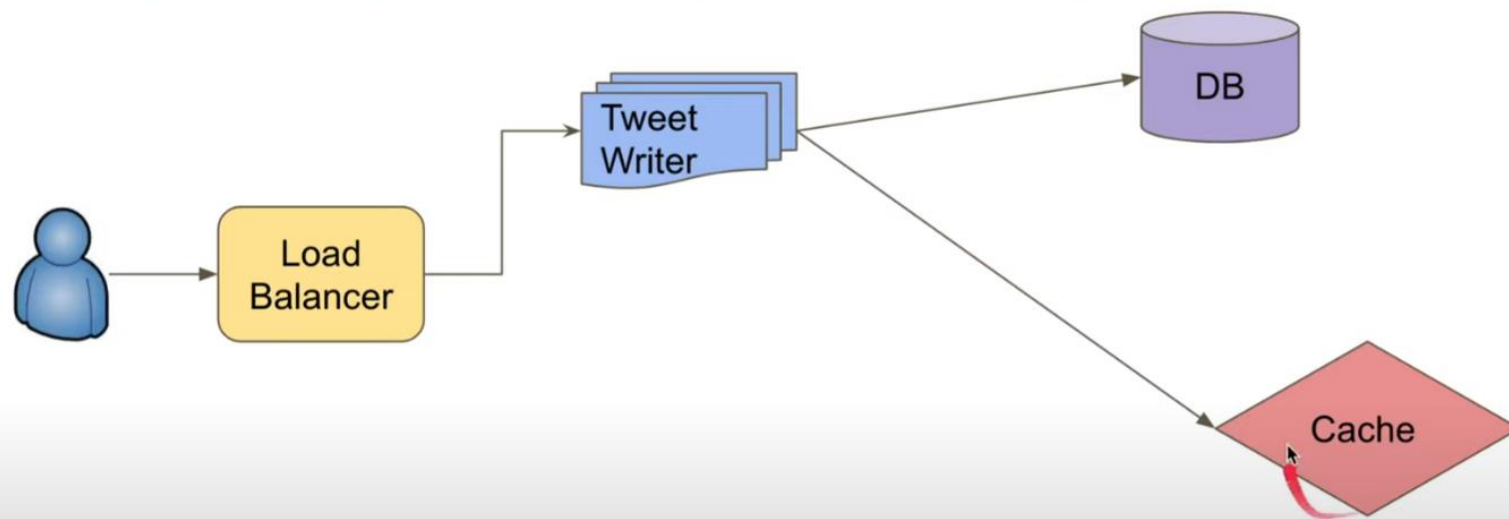


花花酱



huahualeetcode

### Step 4: High-level System Design



Scenario 1: Post tweets

马上我们会揭晓为什么还是要写到cache

## Step 4: High-level

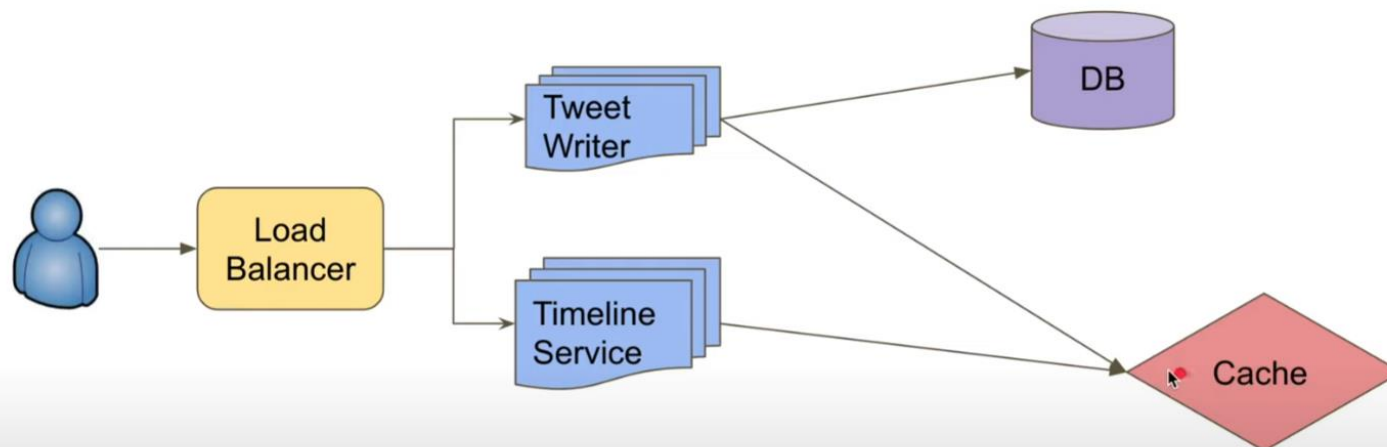


花花酱



huahualeetcode

### User Timeline



Scenario 2: Visit User Timeline

- Need very low latency (~200ms)
- Too time consuming for querying tweets and

这样Timeline Service就可以直接从cache里面读最新的tweets

## Step 4: High-level

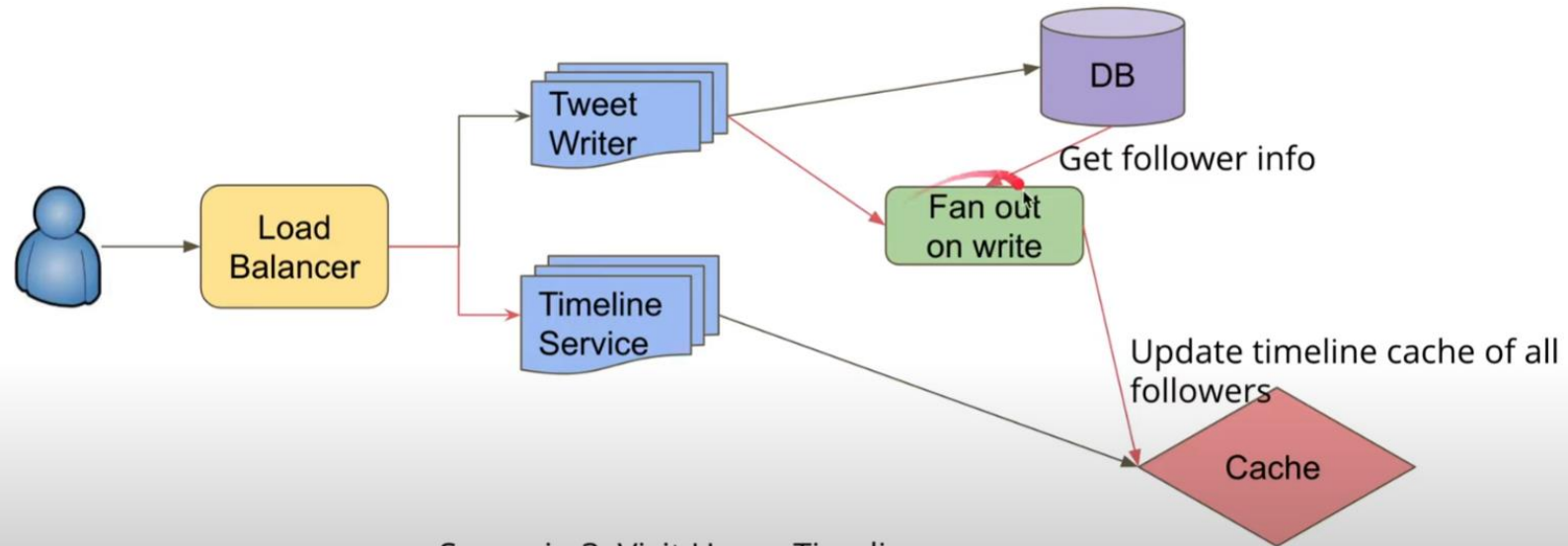


花花酱



huahualeetcode

### Home Timeline



Scenario 3: Visit Home Timeline

Twitter管这个方法叫Fan Out on Write





花花酱



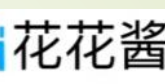
huahualeetcode

## Home Timeline (cont'd)

Naive solution: Pull mode

- How
  - Fetch tweets from N followers from DB, merge and return
- Pros
  - Write is fast:  $O(1)$
- Cons
  - Read is slow:  $O(N)$  DB reads

坏处就是读的时候比较慢

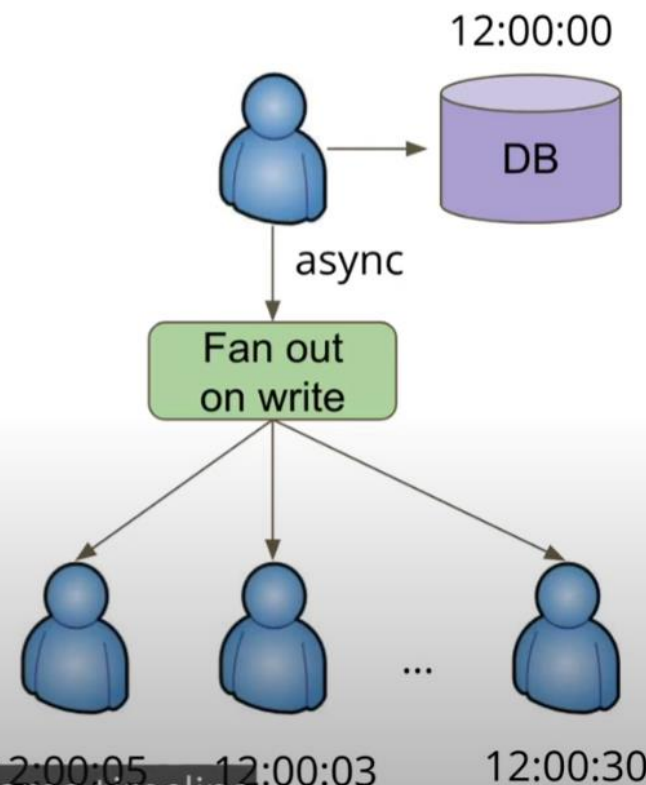


huahualeetcode

## Home Timeline (cont'd)

Better solution: Push mode

- How
  - Maintain a feed list in cache for each user
  - Fanout on write
- Pros
  - Read is fast:  $O(1)$  from the feed list in cache
- Cons
  - Write needs more efforts:  $O(N)$  write for each new tweet
    - Async tasks
  - Delay in showing latest tweets (**eventual consistency**)



所以只需要 $O(1)$ 的时间就能得到要show给这个user的home timeline





花花酱

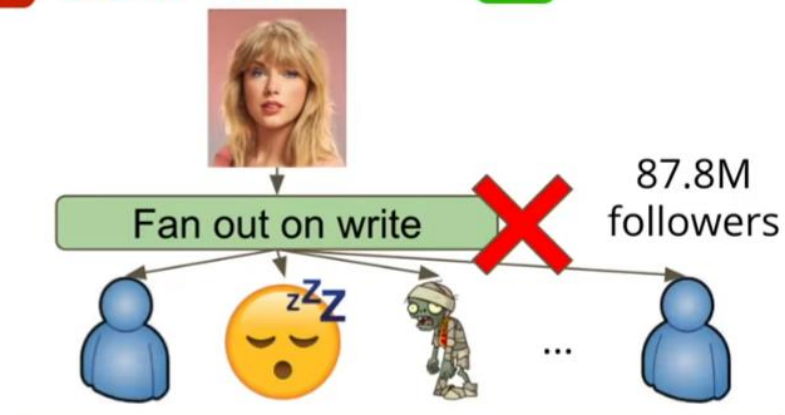


huahualeetcode

## Home Timeline (cont'd)

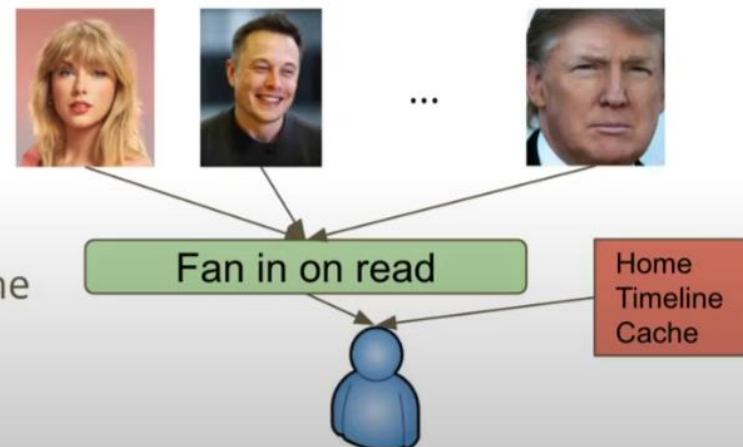
### Fan out on write

- Not efficient for users with huge amount of followers (~>10k)



### Hybrid Solution

- Non-hot users:
  - fan out on write (push): write to user timeline cache
  - do not fanout on non-active users
- Hot users:
  - **fan in on read (pull)**: read during timeline request from tweets cache, and aggregate with results from non-hot users



对于大V（粉丝大于1万个）而言



## Step 5: Data Storage

User Table	
PK	userID: Integer
	name: Varchar (256) email: Varchar (100) creationTime: DateTime lastLogin: DateTime isHotUser: Boolean

Tweet Table	
PK	tweetID: Integer
	userID: Integer creationTime: DateTime content: varchar (140) ...

Follower Table	
PK	userID1: Integer userID2: Integer

PK: primary key

follower table存储的是user ID



# Data Storage

- SQL database
  - E.g., user table
- NoSQL database
  - E.g., timelines
- File system
  - Media file: image, audio, video



# Step 6: Scalability

- Identify potential bottlenecks
- Discussion solutions, focusing on tradeoffs
  - Data sharding
    - Data store, cache
  - Load balancing
    - E.g., user <-> application server; application server <-> cache server; application server <-> db
  - Data caching
    - Read heavy



今天我们主要会着重讲一下怎么进行data sharding



# Consistency patterns

- With multiple copies of the same data, we are faced with options on how to synchronize them so clients have a consistent view of the data. Recall the definition of consistency from the CAP theorem - Every read receives the most recent write or an error.





# Consistency patterns



- **Weak consistency**

- After a write, reads may or may not see it. A best effort approach is taken.
- This approach is seen in systems such as memcached. Weak consistency works well in real time use cases such as VoIP, video chat, and realtime multiplayer games. For example, if you are on a phone call and lose reception for a few seconds, when you regain connection you do not hear what was spoken during connection loss.

- **Eventual consistency**

- After a write, reads will eventually see it (typically within milliseconds). Data is replicated asynchronously.
- This approach is seen in systems such as DNS and email. Eventual consistency works well in highly available systems.

- **Strong consistency**

- After a write, reads will see it. Data is replicated synchronously.
- This approach is seen in file systems and RDBMSes. Strong consistency works well in systems that need transactions.



# Availability patterns

- There are two complementary patterns to support high availability: **fail-over** and **replication**.
-



# Fail-over



- **Active-passive**

- With active-passive fail-over, heartbeats are sent between the active and the passive server on standby. If the heartbeat is interrupted, the passive server takes over the active's IP address and resumes service.
- The length of downtime is determined by whether the passive server is already running in 'hot' standby or whether it needs to start up from 'cold' standby. Only the active server handles traffic.
- Active-passive failover can also be referred to as master-slave failover.

- **Active-active**

- In active-active, both servers are managing traffic, spreading the load between them.
- If the servers are public-facing, the DNS would need to know about the public IPs of both servers. If the servers are internal-facing, application logic would need to know about both servers.
- Active-active failover can also be referred to as master-master failover.

- **Disadvantage(s): failover**

- Fail-over adds more hardware and additional complexity.
- There is a potential for loss of data if the active system fails before any newly written data can be replicated to the passive.





# Replication



- **Master-slave and master-master**
- This topic is further discussed in the [Database](#) section:
  - [Master-slave replication](#)
  - [Master-master replication](#)



# Availability in numbers

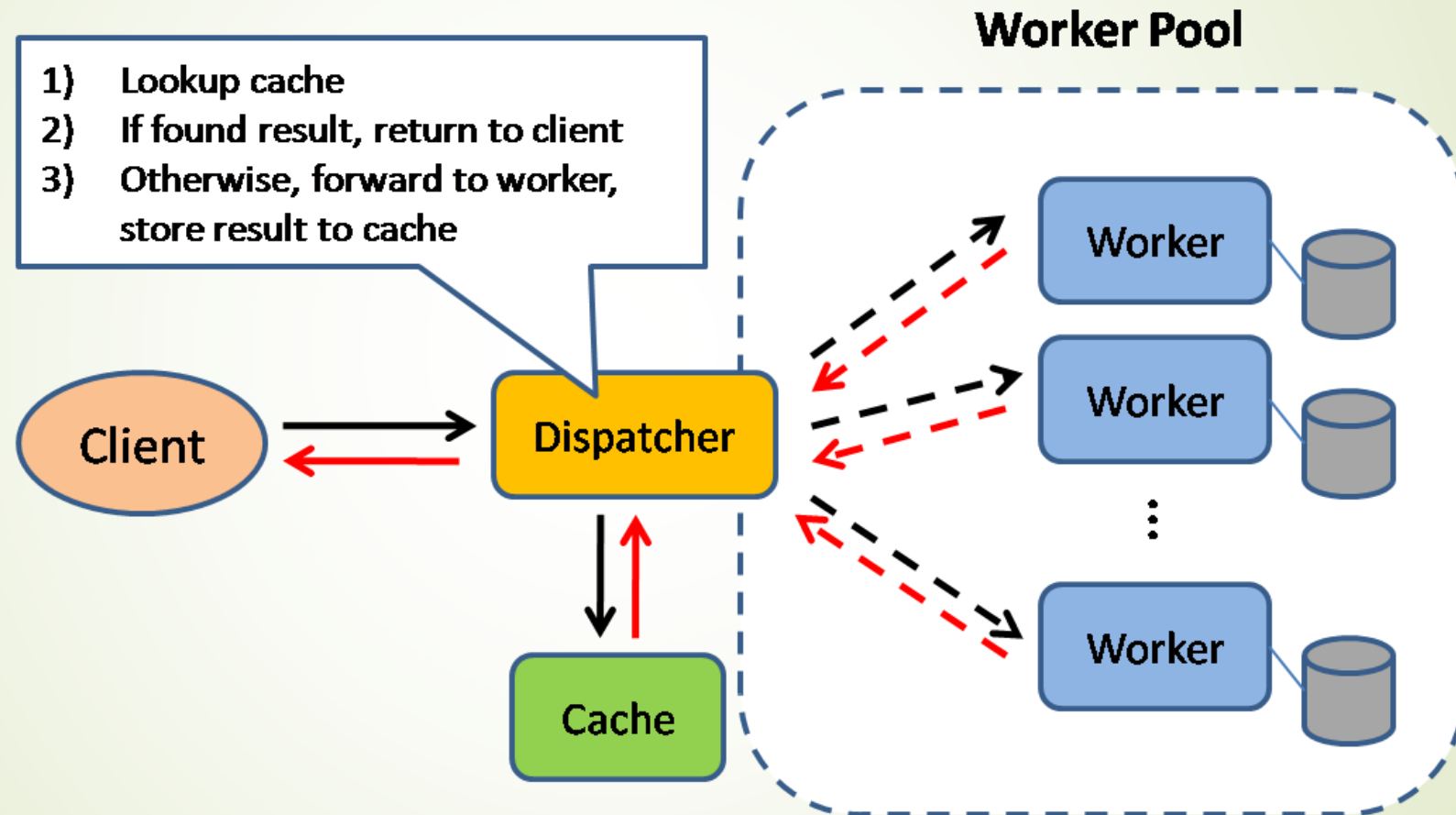


- Availability is often quantified by uptime (or downtime) as a percentage of time the service is available. Availability is generally measured in number of 9s--a service with 99.99% availability is described as having four 9s.
- 99.9% availability - three 9s
  - Duration      Acceptable downtime
  - Downtime per year    8h 45min 57s
  - Downtime per month 43m 49.7s
  - Downtime per week   10m 4.8s
  - Downtime per day    1m 26.4s
- 99.99% availability - four 9s
  - Duration      Acceptable downtime
  - Downtime per year    52min 35.7s
  - Downtime per month 4m 23s
  - Downtime per week   1m 5s
  - Downtime per day    8.6s

# Availability in parallel vs in sequence

- If a service consists of multiple components prone to failure, the service's overall availability depends on whether the components are in sequence or in parallel.
- In sequence
- Overall availability decreases when two components with availability < 100% are in sequence:
  - $\text{Availability (Total)} = \text{Availability (Foo)} * \text{Availability (Bar)}$
  - If both Foo and Bar each had 99.9% availability, their total availability in sequence would be 99.8%.
- In parallel
- Overall availability increases when two components with availability < 100% are in parallel:
  - $\text{Availability (Total)} = 1 - (1 - \text{Availability (Foo)}) * (1 - \text{Availability (Bar)})$
  - If both Foo and Bar each had 99.9% availability, their total availability in parallel would be 99.9999%.

# Cache





# Cache



- Caching improves page load times and can reduce the load on your servers and databases. In this model, the dispatcher will first lookup if the request has been made before and try to find the previous result to return, in order to save the actual execution.
- Databases often benefit from a uniform distribution of reads and writes across its partitions. Popular items can skew the distribution, causing bottlenecks. Putting a cache in front of a database can help absorb uneven loads and spikes in traffic.



# Client caching

- Caches can be located on the client side (OS or browser), server side, or in a distinct cache layer.
- **CDN caching**
- CDNs are considered a type of cache.
- **Web server caching**
- Reverse proxies and caches such as Varnish can serve static and dynamic content directly. Web servers can also cache requests, returning responses without having to contact application servers.
- **Database caching**
- Your database usually includes some level of caching in a default configuration, optimized for a generic use case. Tweaking these settings for specific usage patterns can further boost performance.



# Application caching

- In-memory caches such as Memcached and Redis are key-value stores between your application and your data storage. Since the data is held in RAM, it is much faster than typical databases where data is stored on disk. RAM is more limited than disk, so cache invalidation algorithms such as least recently used (LRU) can help invalidate 'cold' entries and keep 'hot' data in RAM.
- Redis has the following additional features:
  - Persistence option
  - Built-in data structures such as sorted sets and lists
- There are multiple levels you can cache that fall into two general categories: **database queries** and **objects**:
  - Row level
  - Query-level
  - Fully-formed serializable objects
  - Fully-rendered HTML
- Generally, you should try to avoid file-based caching, as it makes cloning and auto-scaling more difficult.





## When to update the cache

- Since you can only store a limited amount of data in cache, you'll need to determine which cache update strategy works best for your use case.





## Cache-aside

The application is responsible for reading and writing from storage. The cache does not interact with storage directly. The application does the following:

- Look for entry in cache, resulting in a cache miss

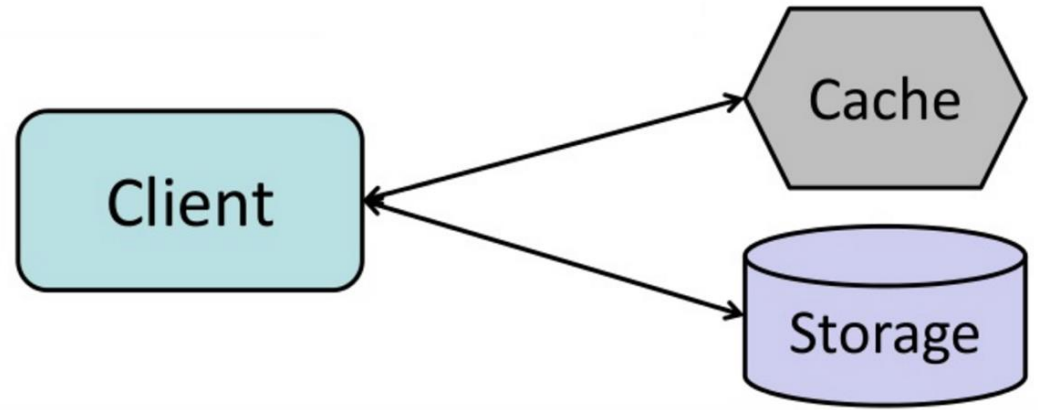
- Load entry from the database

- Add entry to cache

- Return entry

```
def get_user(self, user_id):  
    user = cache.get("user.{0}", user_id)  
    if user is None:  
        user = db.query("SELECT * FROM users WHERE user_id = {0}", user_id)  
        if user is not None:  
            key = "user.{0}".format(user_id)  
            cache.set(key, json.dumps(user))  
    return user
```

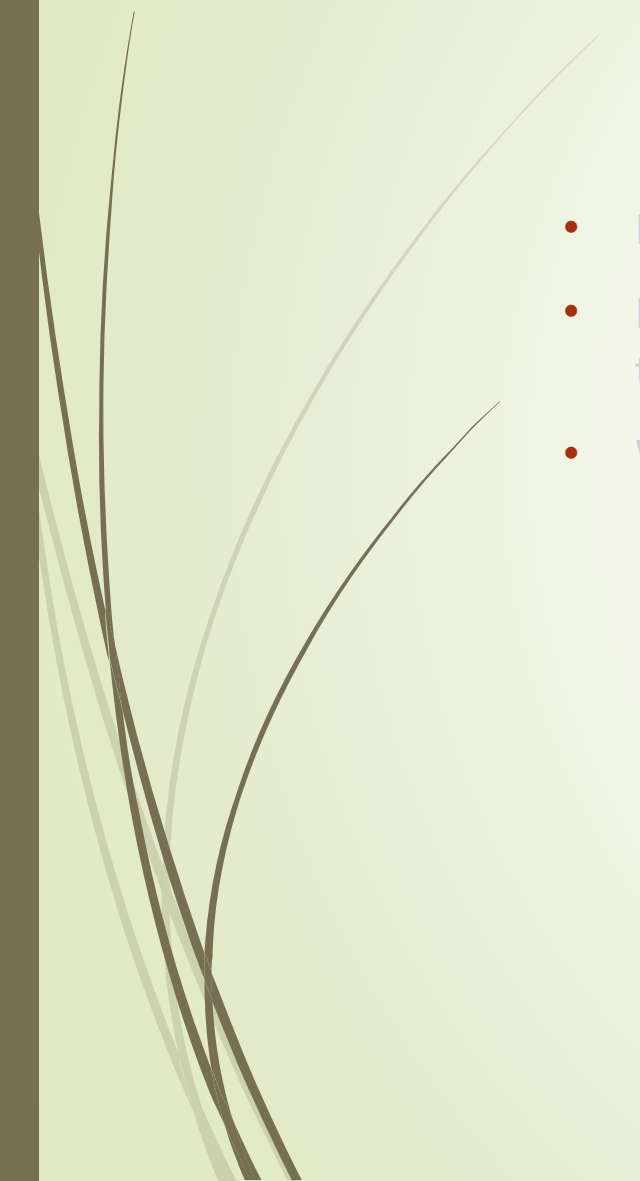
Memcached is generally used in this manner.



Subsequent reads of data added to cache are fast. Cache-aside is also referred to as lazy loading. Only requested data is cached, which avoids filling up the cache with data that isn't requested.



## Disadvantage(s): cache-aside

- Each cache miss results in three trips, which can cause a noticeable delay.
  - Data can become stale if it is updated in the database. This issue is mitigated by setting a time-to-live (TTL) which forces an update of the cache entry, or by using write-through.
  - When a node fails, it is replaced by a new, empty node, increasing latency.
- 

# Write-through

- The application uses the cache as the main data store, reading and writing data to it, while the cache is responsible for reading and writing to the database:

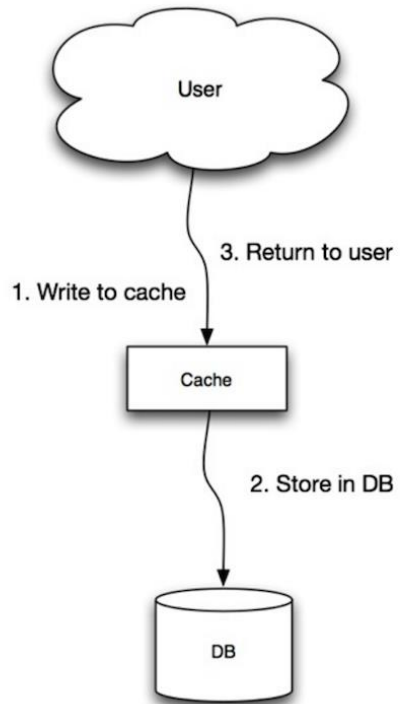
- Application adds/updates entry in cache
- Cache synchronously writes entry to data store
- Return
- Application code:

- `set_user(12345, {"foo":"bar"})`

- Cache code:

- ```
def set_user(user_id, values):  
    user = db.query("UPDATE Users WHERE id = {0}", user_id, values)  
    cache.set(user_id, user)
```

- Write-through is a slow overall operation due to the write operation, but subsequent reads of just written data are fast. Users are generally more tolerant of latency when updating data than reading data. Data in the cache is not stale.





## Disadvantage(s): write through

- When a new node is created due to failure or scaling, the new node will not cache entries until the entry is updated in the database. Cache-aside in conjunction with write through can mitigate this issue.
- Most data written might never be read, which can be minimized with a TTL.

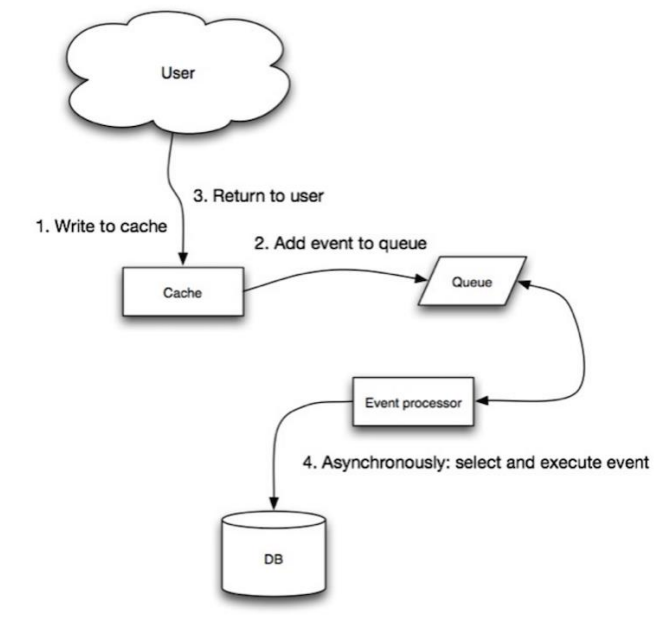
# Write-behind (write-back)

➤ In write-behind, the application does the following:

- Add/update entry in cache
- Asynchronously write entry to the data store, improving write performance

➤ **Disadvantage(s): write-behind**

- There could be data loss if the cache goes down prior to its contents hitting the data store.
- It is more complex to implement write-behind than it is to implement cache-aside or write-through.





# Refresh-ahead

- You can configure the cache to automatically refresh any recently accessed cache entry prior to its expiration.
- Refresh-ahead can result in reduced latency vs read-through if the cache can accurately predict which items are likely to be needed in the future.
- **Disadvantage(s): refresh-ahead**
  - Not accurately predicting which items are likely to be needed in the future can result in reduced performance than without refresh-ahead.



## Disadvantage(s): cache

- Need to maintain consistency between caches and the source of truth such as the database through cache invalidation.
- Cache invalidation is a difficult problem, there is additional complexity associated with when to update the cache.
- Need to make application changes such as adding Redis or memcached.





## Source(s) and further reading

- [From cache to in-memory data grid](#)
- [Scalable system design patterns](#)
- [Introduction to architecting systems for scale](#)
- [Scalability, availability, stability, patterns](#)
- [Scalability](#)
- [AWS ElastiCache strategies](#)
- [Wikipedia](#)



# Sharding

Why?

- Impossible to store/process all data in a single machine

How?

- Break large tables into smaller shards on multiple servers

Pros

- **Horizontal scaling**

Cons

- Complexity (distributed query, resharding)

那么解决Scalability问题的一个重要的工具或者方法的话就是Sharding

| Col1 | Col2 | Col3 |
|------|------|------|
| A    |      |      |
| B    |      |      |
| C    |      |      |
| D    |      |      |
| E    |      |      |

Horizontal  
shards

| Col1 | Col2 | Col3 |
|------|------|------|
| A    |      |      |
| C    |      |      |

| Col1 | Col2 | Col3 |
|------|------|------|
| E    |      |      |

| Col1 | Col2 | Col3 |
|------|------|------|
| B    |      |      |
| D    |      |      |

# Sharding (Cont'd)

Option 1: Shard by tweet's creation time

Pros:

- Limited shards to query

Cons:

- Hot/Cold data issue
- New shards fill up quickly

| TweetId | DateTime   |
|---------|------------|
| 100000  | 12/23/2020 |
| 100001  | 12/23/2020 |
| 100002  | 12/23/2020 |
| 100003  | 12/23/2020 |
| ....    |            |

Hot: High qps

| TweetId | DateTime   |
|---------|------------|
| 1000    | 12/23/2012 |
| 1001    | 12/23/2012 |
| 1002    | 12/23/2012 |
| 1003    | 12/23/2012 |
| ....    |            |

Cold: Low qps

这个很明显 (Shard) by creation time就会有这样的问题

# Sharding (Cont'd)

Option 2: Shard by hash(userId): store all the data of a user on a single shard

Pros:

- Simple
- Query user timeline is straightforward

Cons:

- Home timeline still needs to query multiple shards
- Non-uniform distribution of storage
  - User data might not be able to fit into a single shard
- Hot users
- Availability

那么这是Shard by hash(userId) 比之前的creation time要稍微好一点

# Sharding (Cont'd)

Option 3: Shard by hash(tweetId)

Pros:

- Uniform distribution
- High availability

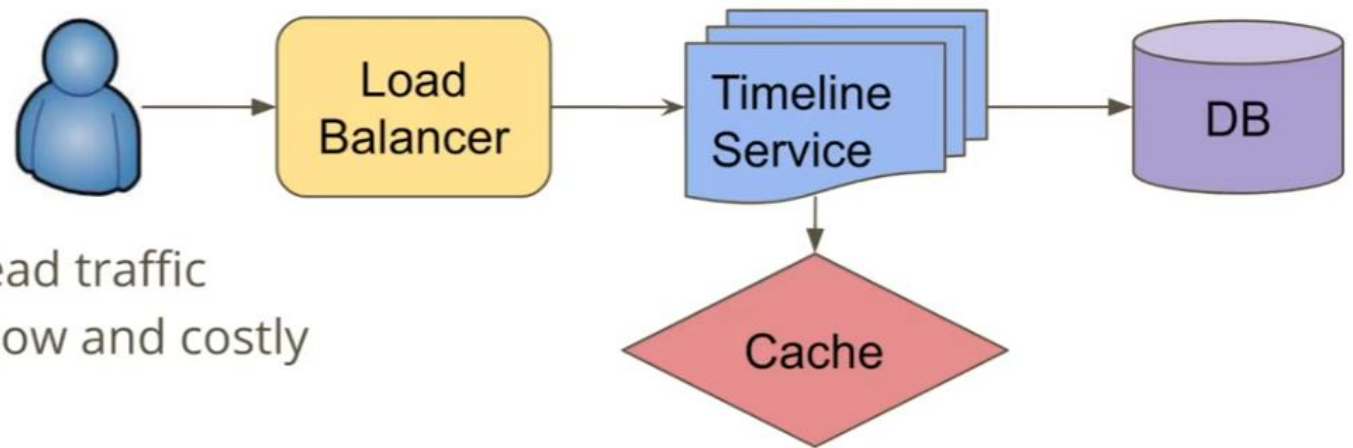
Cons:

- Need to query all shards in order to generate user / home timeline

这样的话我们就可以几乎做到uniform distribution实现这个high availability



# Caching



- Why?
  - Social networks have heavy read traffic
  - (Distributed) queries can be slow and costly
- How
  - Store hot / precomputed data in memory, reads can be much faster
- Timeline service
  - User timeline:  $\text{user\_id} \rightarrow \{\text{tweet\_id}\}$  # Varies a lot,  $T = 1\text{k} \sim 100\text{k}$ , Trump:  $\sim 60\text{k}$
  - Home timeline:  $\text{user\_id} \rightarrow \{\text{tweet\_id}\}$  # This can be huge,  $\text{sum}(\text{followee's tweets})$
  - Tweets:  $\text{tweet\_id} \rightarrow \text{tweet}$  # Common data that can be shared
- Topics
  - Caching policy
  - Sharding
  - Performance

那么我们把一些hot和precomputed的数据存储在memory里面



# Caching 的考察范围

- Policy:
  - LRU, LFU
- Cache也可能要做Sharding。
- Cache比db大吗？
- Cache和db谁是source of truth？



怎么优化？How to Scale?



# Optimization 1: Data Sharding

Why?

- Horizontal scaling: distribute our database load to multiple servers making it scalable

How?

- Sharding by **video\_id** via **consistent hashing**
  - Pros:
    - Uniform distribution (no hot user problem)
  - Cons:
    - Need to query all shards in order to get a user's videos.
    - Hot videos can make shards experience very high traffic.

Shard by videoid的好处就相比Shard by userID没有hot user的问题



# Optimization 2: Data Replication

Why?

- Scale to handle heavy read traffic by making the same data available in multiple machines
- Improve availability

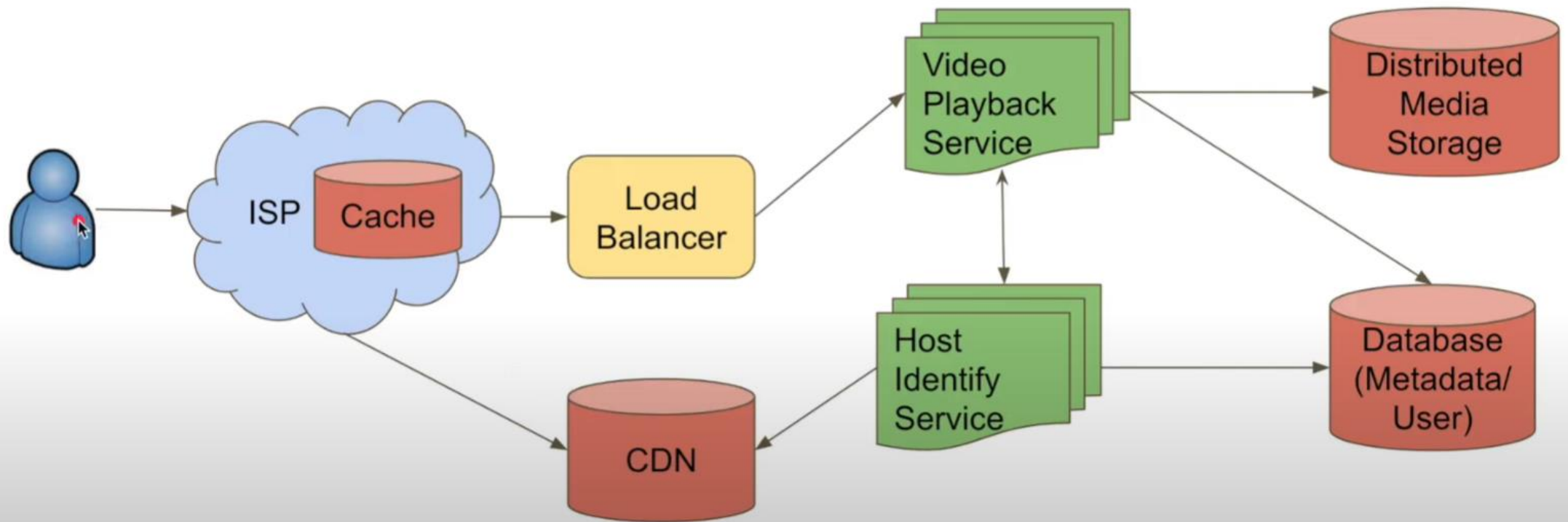
How?

- Primary-secondary configuration
  - Write to primary and then propagate to all secondaries
  - Read from secondary

Cons?

- Break consistency: data consistency. 这里有个知识点就是怎么进行拷贝

## Optimization 3: Caching (Cont'd)



Stream video (通过ISP) 然后再到达用户

## Optimization 4: CDN

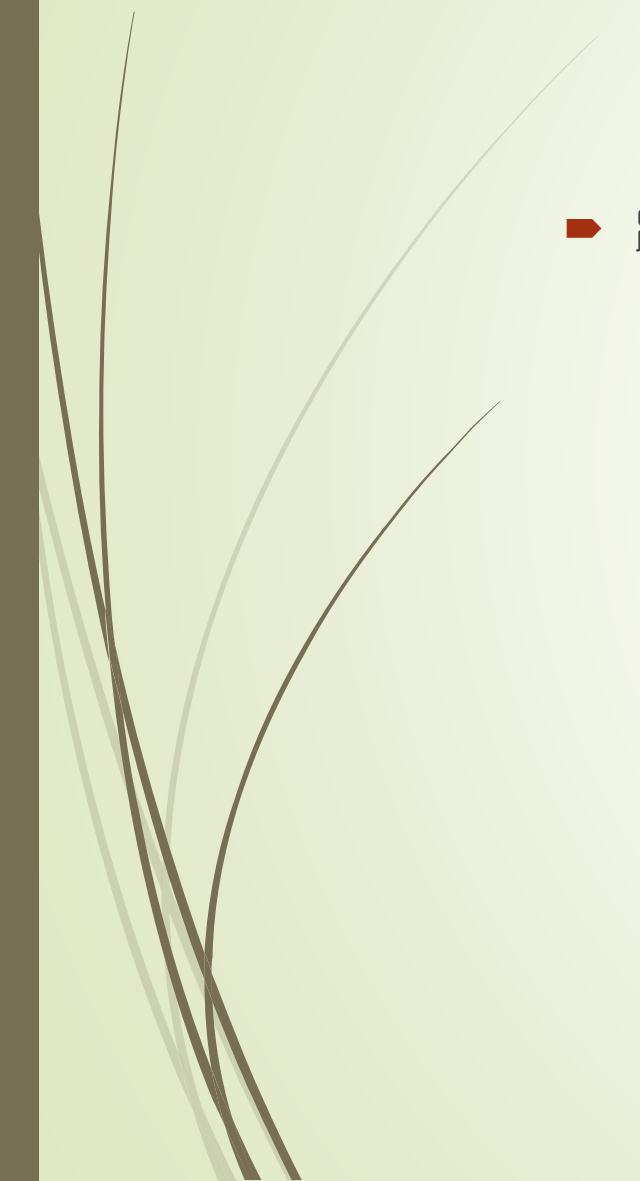
- Predict locations where people would prefer to watch a video.
  - Copy the video to the CDNs closest to the predicted locations in advance. So the videos are ready to serve the users when requested.
  - Copy the video at off-peak time.

YouTube预测到你现在会观看者的视频然后所以提前把这个视频



# Q & A

➡ 踊跃发言





# Thank You!

- References:

- <https://www.youtube.com/watch?v=PMCdWr6ejpw&list=PLLuMmzMTgVK4RuSJjXUxjeUt3-vSyA1Or>
- <https://github.com/donnemartin/system-design-primer>
- <https://www.youtube.com/channel/UC9vLsnF6QPYuH51njmllooCQ>
- <https://www.youtube.com/user/braveheartcy>
- <https://www.educative.io/courses/grokking-the-system-design-interview/3j6NnJrpp5p>