

System Design Primer - Complete Guide

- [The System Design Primer](#)
 - [Motivation](#)
 - [Learn how to design large-scale systems](#)
 - [Learn from the open source community](#)
 - [Prep for the system design interview](#)
 - [Anki flashcards](#)
 - [Coding Resource: Interactive Coding Challenges](#)
 - [Contributing](#)
 - [Index of system design topics](#)
 - [Study guide](#)
 - [How to approach a system design interview question](#)
 - [Step 1: Outline use cases, constraints, and assumptions](#)
 - [Step 2: Create a high level design](#)
 - [Step 3: Design core components](#)
 - [Step 4: Scale the design](#)
 - [Back-of-the-envelope calculations](#)
 - [Source\(s\) and further reading](#)
 - [System design interview questions with solutions](#)
 - [Design Pastebin.com \(or Bit.ly\)](#)
 - [Design the Twitter timeline and search \(or Facebook feed and search\)](#)
 - [Design a web crawler](#)
 - [Design Mint.com](#)
 - [Design the data structures for a social network](#)
 - [Design a key-value store for a search engine](#)
 - [Design Amazon's sales ranking by category feature](#)
 - [Design a system that scales to millions of users on AWS](#)
 - [Object-oriented design interview questions with solutions](#)
 - [System design topics: start here](#)
 - [Step 1: Review the scalability video lecture](#)
 - [Step 2: Review the scalability article](#)
 - [Next steps](#)
 - [Performance vs scalability](#)

- [Source\(s\) and further reading](#)
- [Latency vs throughput](#)
 - [Source\(s\) and further reading](#)
- [Availability vs consistency](#)
 - [CAP theorem](#)
 - [Source\(s\) and further reading](#)
- [Consistency patterns](#)
 - [Weak consistency](#)
 - [Eventual consistency](#)
 - [Strong consistency](#)
 - [Source\(s\) and further reading](#)
- [Availability patterns](#)
 - [Fail-over](#)
 - [Disadvantage\(s\): failover](#)
 - [Replication](#)
 - [Availability in numbers](#)
- [Domain name system](#)
 - [Disadvantage\(s\): DNS](#)
 - [Source\(s\) and further reading](#)
- [Content delivery network](#)
 - [Push CDNs](#)
 - [Pull CDNs](#)
 - [Disadvantage\(s\): CDN](#)
 - [Source\(s\) and further reading](#)
- [Load balancer](#)
 - [Layer 4 load balancing](#)
 - [Layer 7 load balancing](#)
 - [Horizontal scaling](#)
 - [Disadvantage\(s\): load balancer](#)
 - [Source\(s\) and further reading](#)
- [Reverse proxy \(web server\)](#)
 - [Load balancer vs reverse proxy](#)
 - [Disadvantage\(s\): reverse proxy](#)
 - [Source\(s\) and further reading](#)
- [Application layer](#)
 - [Microservices](#)
 - [Service Discovery](#)

- [Disadvantage\(s\): application layer](#)
 - [Source\(s\) and further reading](#)
- [Database](#)
 - [Relational database management system \(RDBMS\)](#).
 - [NoSQL](#)
 - [SQL or NoSQL](#)
- [Cache](#)
 - [Client caching](#)
 - [CDN caching](#)
 - [Web server caching](#)
 - [Database caching](#)
 - [Application caching](#)
 - [Caching at the database query level](#)
 - [Caching at the object level](#)
 - [When to update the cache](#)
 - [Disadvantage\(s\): cache](#)
 - [Source\(s\) and further reading](#)
- [Asynchronism](#)
 - [Message queues](#)
 - [Task queues](#)
 - [Back pressure](#)
 - [Disadvantage\(s\): asynchronism](#)
 - [Source\(s\) and further reading](#)
- [Communication](#)
 - [Hypertext transfer protocol \(HTTP\)](#).
 - [Transmission control protocol \(TCP\)](#).
 - [User datagram protocol \(UDP\)](#).
 - [Remote procedure call \(RPC\)](#).
 - [Representational state transfer \(REST\)](#).
 - [RPC and REST calls comparison](#)
- [Security](#)
 - [Source\(s\) and further reading](#)
- [Appendix](#)
 - [Powers of two table](#)
 - [Latency numbers every programmer should know](#)
 - [Additional system design interview questions](#)
 - [Real world architectures](#)

- [Company architectures](#)
 - [Company engineering blogs](#)
 - [Under development](#)
 - [Credits](#)
 - [Contact info](#)
 - [License](#)
- [CONTRIBUTING](#)
- [Contributing](#)
 - [Bug Reports](#)
 - [Pull Requests](#)
 - [GitHub Pull Requests Docs](#)
 - [Translations](#)
 - [Changes to translations](#)
 - [Adding translations to new languages](#)
 - [Translation template credits](#)
- [System Design: Pastebin](#)
- [Design Pastebin.com \(or Bit.ly.\)](#)
 - [Step 1: Outline use cases and constraints](#)
 - [Use cases](#)
 - [Constraints and assumptions](#)
 - [Step 2: Create a high level design](#)
 - [Step 3: Design core components](#)
 - [Use case: User enters a block of text and gets a randomly generated link](#)
 - [Use case: User enters a paste's url and views the contents](#)
 - [Use case: Service tracks analytics of pages](#)
 - [Use case: Service deletes expired pastes](#)
 - [Step 4: Scale the design](#)
 - [Additional talking points](#)
 - [Caching](#)
 - [Asynchronism and microservices](#)
 - [Communications](#)
 - [Security](#)
 - [Latency numbers](#)
 - [Ongoing](#)
- [System Design: Twitter](#)
- [Design the Twitter timeline and search](#)

- [Step 1: Outline use cases and constraints](#)
 - [Use cases](#)
 - [Constraints and assumptions](#)
- [Step 2: Create a high level design](#)
- [Step 3: Design core components](#)
 - [Use case: User posts a tweet](#)
 - [Use case: User views the home timeline](#)
 - [Use case: User views the user timeline](#)
 - [Use case: User searches keywords](#)
- [Step 4: Scale the design](#)
- [Additional talking points](#)
 - [Caching](#)
 - [Asynchronism and microservices](#)
 - [Communications](#)
 - [Security](#)
 - [Latency numbers](#)
 - [Ongoing](#)
- [System Design: Web Crawler](#)
- [Design a web crawler](#)
 - [Step 1: Outline use cases and constraints](#)
 - [Use cases](#)
 - [Constraints and assumptions](#)
 - [Step 2: Create a high level design](#)
 - [Step 3: Design core components](#)
 - [Use case: Service crawls a list of urls](#)
 - [Handling duplicates](#)
 - [Determining when to update the crawl results](#)
 - [Use case: User inputs a search term and sees a list of relevant pages with titles and snippets](#)
 - [Step 4: Scale the design](#)
 - [Additional talking points](#)
 - [SQL scaling patterns](#)
 - [Caching](#)
 - [Asynchronism and microservices](#)
 - [Communications](#)
 - [Security](#)
 - [Latency numbers](#)

- [Ongoing](#)
- [System Design: Mint](#)
- [Design Mint.com](#)
 - [Step 1: Outline use cases and constraints](#)
 - [Use cases](#)
 - [Constraints and assumptions](#)
 - [Step 2: Create a high level design](#)
 - [Step 3: Design core components](#)
 - [Use case: User connects to a financial account](#)
 - [Use case: Service extracts transactions from the account](#)
 - [Use case: Service recommends a budget](#)
 - [Step 4: Scale the design](#)
 - [Additional talking points](#)
 - [Caching](#)
 - [Asynchronism and microservices](#)
 - [Communications](#)
 - [Security](#)
 - [Latency numbers](#)
 - [Ongoing](#)
- [System Design: Social Graph](#)
- [Design the data structures for a social network](#)
 - [Step 1: Outline use cases and constraints](#)
 - [Use cases](#)
 - [Constraints and assumptions](#)
 - [Step 2: Create a high level design](#)
 - [Step 3: Design core components](#)
 - [Use case: User searches for someone and sees the shortest path to the searched person](#)
 - [Step 4: Scale the design](#)
 - [Additional talking points](#)
 - [SQL scaling patterns](#)
 - [Caching](#)
 - [Asynchronism and microservices](#)
 - [Communications](#)
 - [Security](#)
 - [Latency numbers](#)
 - [Ongoing](#)

- [System Design: Query Cache](#)
- [Design a key-value cache to save the results of the most recent web server queries](#)
 - [Step 1: Outline use cases and constraints](#)
 - [Use cases](#)
 - [Constraints and assumptions](#)
 - [Step 2: Create a high level design](#)
 - [Step 3: Design core components](#)
 - [Use case: User sends a request resulting in a cache hit](#)
 - [Step 4: Scale the design](#)
 - [Expanding the Memory Cache to many machines](#)
 - [Additional talking points](#)
 - [SQL scaling patterns](#)
 - [Caching](#)
 - [Asynchronism and microservices](#)
 - [Communications](#)
 - [Security](#)
 - [Latency numbers](#)
 - [Ongoing](#)
- [System Design: Sales Rank](#)
- [Design Amazon's sales rank by category feature](#)
 - [Step 1: Outline use cases and constraints](#)
 - [Use cases](#)
 - [Constraints and assumptions](#)
 - [Step 2: Create a high level design](#)
 - [Step 3: Design core components](#)
 - [Use case: Service calculates the past week's most popular products by category](#)
 - [Use case: User views the past week's most popular products by category](#)
 - [Step 4: Scale the design](#)
 - [Additional talking points](#)
 - [Caching](#)
 - [Asynchronism and microservices](#)
 - [Communications](#)
 - [Security](#)
 - [Latency numbers](#)

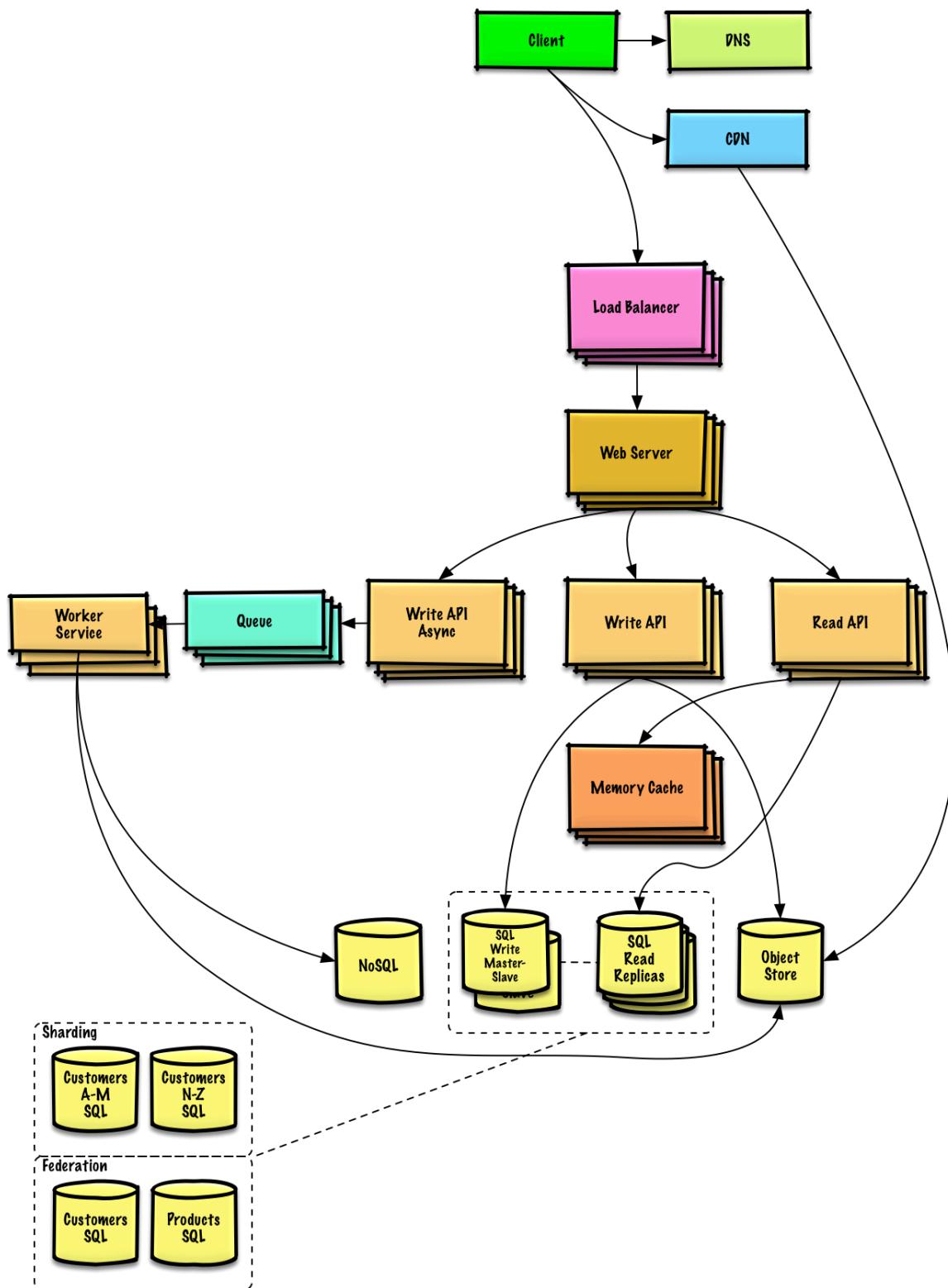
- [Ongoing](#)
- [System Design: Scaling Aws](#)
- [Design a system that scales to millions of users on AWS](#)
 - [Step 1: Outline use cases and constraints](#)
 - [Use cases](#)
 - [Constraints and assumptions](#)
 - [Step 2: Create a high level design](#)
 - [Step 3: Design core components](#)
 - [Use case: User makes a read or write request](#)
 - [Step 4: Scale the design](#)
 - [Users+](#)
 - [Users++](#)
 - [Users+++](#)
 - [Users++++](#)
 - [Users+++++](#)
 - [Additional talking points](#)
 - [SQL scaling patterns](#)
 - [Caching](#)
 - [Asynchronism and microservices](#)
 - [Communications](#)
 - [Security](#)
 - [Latency numbers](#)
 - [Ongoing](#)
- [Object Oriented Design: Hash Table](#)
- [Design a hash map](#)
- [Constraints and assumptions](#)
- [Solution](#)
- [Object Oriented Design: Lru Cache](#)
- [Design an LRU cache](#)
- [Constraints and assumptions](#)
- [Solution](#)
- [Object Oriented Design: Call Center](#)
- [Design a call center](#)
- [Constraints and assumptions](#)
- [Solution](#)
- [Object Oriented Design: Deck Of Cards](#)
- [Design a deck of cards](#)

- [Constraints and assumptions](#)
- [Solution](#)
- [Object Oriented Design: Parking Lot](#)
- [Design a parking lot](#)
- [Constraints and assumptions](#)
- [Solution](#)
- [Object Oriented Design: Online Chat](#)
- [Design an online chat](#)
- [Constraints and assumptions](#)
- [Solution](#)

[English](#) • [العربية](#) / [বাংলা](#) • [Português do Brasil](#) • [Deutsch](#) • [ελληνικά](#) • [תלְבִּינָה](#) • [Italiano](#) • [한국어](#) • [فارسی](#) • [Polski](#) • [русский язык](#) • [Español](#) • [ភាសាខ្មែរ](#) • [Türkçe](#) • [tiếng Việt](#) • [Français](#) | [Add Translation](#)

Help [translate](#) this guide!

The System Design Primer



Motivation

Learn how to design large-scale systems.

Prep for the system design interview.

Learn how to design large-scale systems

Learning how to design scalable systems will help you become a better engineer.

System design is a broad topic. There is a **vast amount of resources scattered throughout the web** on system design principles.

This repo is an **organized collection** of resources to help you learn how to build systems at scale.

Learn from the open source community

This is a continually updated, open source project.

[Contributions](#) are welcome!

Prep for the system design interview

In addition to coding interviews, system design is a **required component** of the **technical interview process** at many tech companies.

Practice common system design interview questions and **compare** your results with **sample solutions**: discussions, code, and diagrams.

Additional topics for interview prep:

- [Study guide](#)
- [How to approach a system design interview question](#)
- [System design interview questions, with solutions](#)
- [Object-oriented design interview questions, with solutions](#)
- [Additional system design interview questions](#)

Anki flashcards

The image shows two side-by-side screenshots of the Anki mobile application on an Android device. Both screens display a flashcard with the title "System Design".

Left Screen (Card 1):

- Title: System Design
- Time left: 11 minutes
- Progress: 1 / 0 / 3
- Card Type: User datagram protocol (UDP)
- Diagram: Shows two computers connected by a line with an envelope icon. Text below says: "Not all data is present. Do not resend."
- Source: [How to make a multiplayer game](#)
- Description: UDP is connectionless. Datagrams (analogous to packets) are guaranteed only at the datagram level. Datagrams might reach their destination out of order or not at all. UDP does not support congestion control. Without the guarantees that TCP supports, UDP is generally more efficient.
- Bottom navigation: Includes a red "AGAIN" button, a grey "HARD" button, a green "GOOD" button, and a blue "EASY" button.

Right Screen (Card 2):

- Title: System Design Exer...
- Time left: 15 minutes
- Progress: 0 / 0 / 7
- Card Type: Step 2: Create a high level design
- Text: Outline a high level design with all important components.
- Diagram: A network architecture diagram. A green "Client" box points down to a yellow "Web Server" box. The "Web Server" box has two outgoing arrows: one to a yellow "Search API" box and one to a yellow "User Graph Service" box. The "User Graph Service" box has two outgoing arrows: one to a yellow "Person Server" box and one to a yellow "Lookup Service" box. All boxes are interconnected with bidirectional arrows.
- Bottom navigation: Includes a red "AGAIN" button, a grey "HARD" button, a green "GOOD" button, and a blue "EASY" button.

The provided [Anki flashcard decks](#) use spaced repetition to help you retain key system design concepts.

- [System design deck](#)
- [System design exercises deck](#)
- [Object oriented design exercises deck](#)

Great for use while on-the-go.

Coding Resource: Interactive Coding Challenges

Looking for resources to help you prep for the [Coding Interview](#)?

Algorithm

A heap is a complete binary tree where each node is smaller than its children.

extract_min

```

      -5-
     /   \
    20   15
   / \   / \
  22 40 25
  
```

Save the root as the value to be returned: 5
Move the right most element to the root: 25

```

      -25-
     /   \
    20   15
   / \   / \
  22 40
  
```

Bubble down 25: Swap 25 and 15 (the smaller child)

```

      -15-
     /   \
    20   25
   / \   / \
  22 40
  
```

2 h AGAIN 9 d HARD 14 d GOOD 14 d EASY 2 h AGAIN 1 d GOOD

Check out the sister repo [Interactive Coding Challenges](#), which contains an additional Anki deck:

- [Coding deck](#)

Contributing

Learn from the community.

Feel free to submit pull requests to help:

- Fix errors
- Improve sections
- Add new sections

- [Translate](#)

Content that needs some polishing is placed [under development](#).

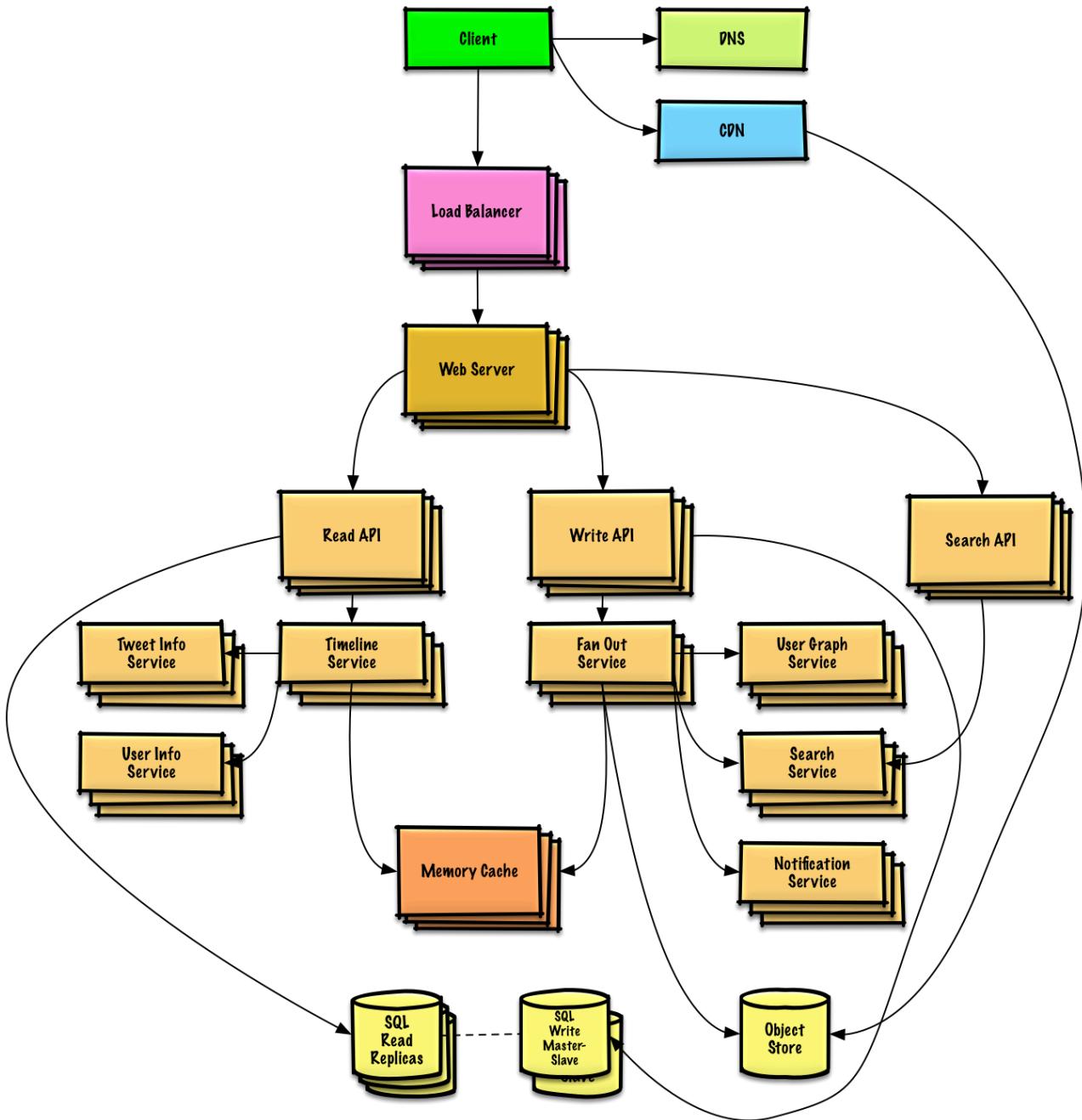
Review the [Contributing Guidelines](#).

Index of system design topics

Summaries of various system design topics, including pros and cons.

Everything is a trade-off.

Each section contains links to more in-depth resources.



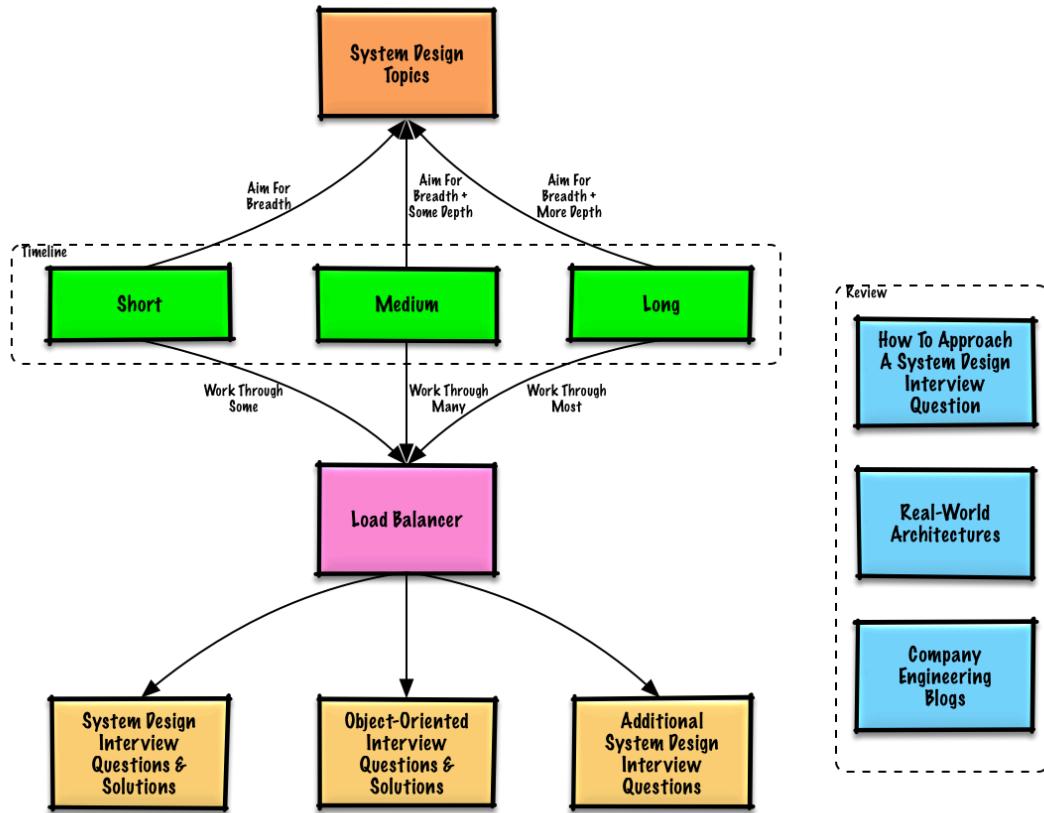
- [System design topics: start here](#)
 - [Step 1: Review the scalability video lecture](#)
 - [Step 2: Review the scalability article](#)
 - [Next steps](#)
- [Performance vs scalability](#)
- [Latency vs throughput](#)
- [Availability vs consistency](#)
 - [CAP theorem](#)
 - [CP - consistency and partition tolerance](#)

- [AP - availability and partition tolerance](#)
- [Consistency patterns](#)
 - [Weak consistency](#)
 - [Eventual consistency](#)
 - [Strong consistency](#)
- [Availability patterns](#)
 - [Fail-over](#)
 - [Replication](#)
 - [Availability in numbers](#)
- [Domain name system](#)
- [Content delivery network](#)
 - [Push CDNs](#)
 - [Pull CDNs](#)
- [Load balancer](#)
 - [Active-passive](#)
 - [Active-active](#)
 - [Layer 4 load balancing](#)
 - [Layer 7 load balancing](#)
 - [Horizontal scaling](#)
- [Reverse proxy \(web server\)](#)
 - [Load balancer vs reverse proxy](#)
- [Application layer](#)
 - [Microservices](#)
 - [Service discovery](#)
- [Database](#)
 - [Relational database management system \(RDBMS\)](#)
 - [Master-slave replication](#)
 - [Master-master replication](#)
 - [Federation](#)
 - [Sharding](#)
 - [Denormalization](#)
 - [SQL tuning](#)
 - [NoSQL](#)
 - [Key-value store](#)
 - [Document store](#)
 - [Wide column store](#)
 - [Graph Database](#)

- [SQL or NoSQL](#)
- [Cache](#)
 - [Client caching](#)
 - [CDN caching](#)
 - [Web server caching](#)
 - [Database caching](#)
 - [Application caching](#)
 - [Caching at the database query level](#)
 - [Caching at the object level](#)
 - [When to update the cache](#)
 - [Cache-aside](#)
 - [Write-through](#)
 - [Write-behind \(write-back\)](#).
 - [Refresh-ahead](#)
- [Asynchronism](#)
 - [Message queues](#)
 - [Task queues](#)
 - [Back pressure](#)
- [Communication](#)
 - [Transmission control protocol \(TCP\)](#).
 - [User datagram protocol \(UDP\)](#).
 - [Remote procedure call \(RPC\)](#).
 - [Representational state transfer \(REST\)](#).
- [Security](#)
- [Appendix](#)
 - [Powers of two table](#)
 - [Latency numbers every programmer should know](#)
 - [Additional system design interview questions](#)
 - [Real world architectures](#)
 - [Company architectures](#)
 - [Company engineering blogs](#)
- [Under development](#)
- [Credits](#)
- [Contact info](#)
- [License](#)

Study guide

Suggested topics to review based on your interview timeline (short, medium, long).



Imgur

Q: For interviews, do I need to know everything here?

A: No, you don't need to know everything here to prepare for the interview.

What you are asked in an interview depends on variables such as:

- How much experience you have
- What your technical background is
- What positions you are interviewing for
- Which companies you are interviewing with
- Luck

More experienced candidates are generally expected to know more about system design. Architects or team leads might be expected to know more than individual

contributors. Top tech companies are likely to have one or more design interview rounds.

Start broad and go deeper in a few areas. It helps to know a little about various key system design topics. Adjust the following guide based on your timeline, experience, what positions you are interviewing for, and which companies you are interviewing with.

- **Short timeline** - Aim for **breadth** with system design topics. Practice by solving **some** interview questions.
- **Medium timeline** - Aim for **breadth** and **some depth** with system design topics. Practice by solving **many** interview questions.
- **Long timeline** - Aim for **breadth** and **more depth** with system design topics. Practice by solving **most** interview questions.

	Short	Medium	Long
Read through the System design topics to get a broad understanding of how systems work	:+1:	:+1:	:+1:
Read through a few articles in the Company engineering blogs for the companies you are interviewing with	:+1:	:+1:	:+1:
Read through a few Real world architectures	:+1:	:+1:	:+1:
Review How to approach a system design interview question	:+1:	:+1:	:+1:
Work through System design interview questions with solutions	Some	Many	Most
Work through Object-oriented design interview questions with solutions	Some	Many	Most

	Short	Medium	Long
Review Additional system design interview questions	Some	Many	Most

How to approach a system design interview question

How to tackle a system design interview question.

The system design interview is an **open-ended conversation**. You are expected to lead it.

You can use the following steps to guide the discussion. To help solidify this process, work through the [System design interview questions with solutions](#) section using the following steps.

Step 1: Outline use cases, constraints, and assumptions

Gather requirements and scope the problem. Ask questions to clarify use cases and constraints. Discuss assumptions.

- Who is going to use it?
- How are they going to use it?
- How many users are there?
- What does the system do?
- What are the inputs and outputs of the system?
- How much data do we expect to handle?
- How many requests per second do we expect?
- What is the expected read to write ratio?

Step 2: Create a high level design

Outline a high level design with all important components.

- Sketch the main components and connections
- Justify your ideas

Step 3: Design core components

Dive into details for each core component. For example, if you were asked to [design a url shortening service](#), discuss:

- Generating and storing a hash of the full url
 - [MD5](#) and [Base62](#)
 - Hash collisions
 - SQL or NoSQL
 - Database schema
- Translating a hashed url to the full url
 - Database lookup
- API and object-oriented design

Step 4: Scale the design

Identify and address bottlenecks, given the constraints. For example, do you need the following to address scalability issues?

- Load balancer
- Horizontal scaling
- Caching
- Database sharding

Discuss potential solutions and trade-offs. Everything is a trade-off. Address bottlenecks using [principles of scalable system design](#).

Back-of-the-envelope calculations

You might be asked to do some estimates by hand. Refer to the [Appendix](#) for the following resources:

- [Use back of the envelope calculations](#)
- [Powers of two table](#)
- [Latency numbers every programmer should know](#)

Source(s) and further reading

Check out the following links to get a better idea of what to expect:

- [How to ace a systems design interview](#)

- [The system design interview](#)
- [Intro to Architecture and Systems Design Interviews](#)
- [System design template](#)

System design interview questions with solutions

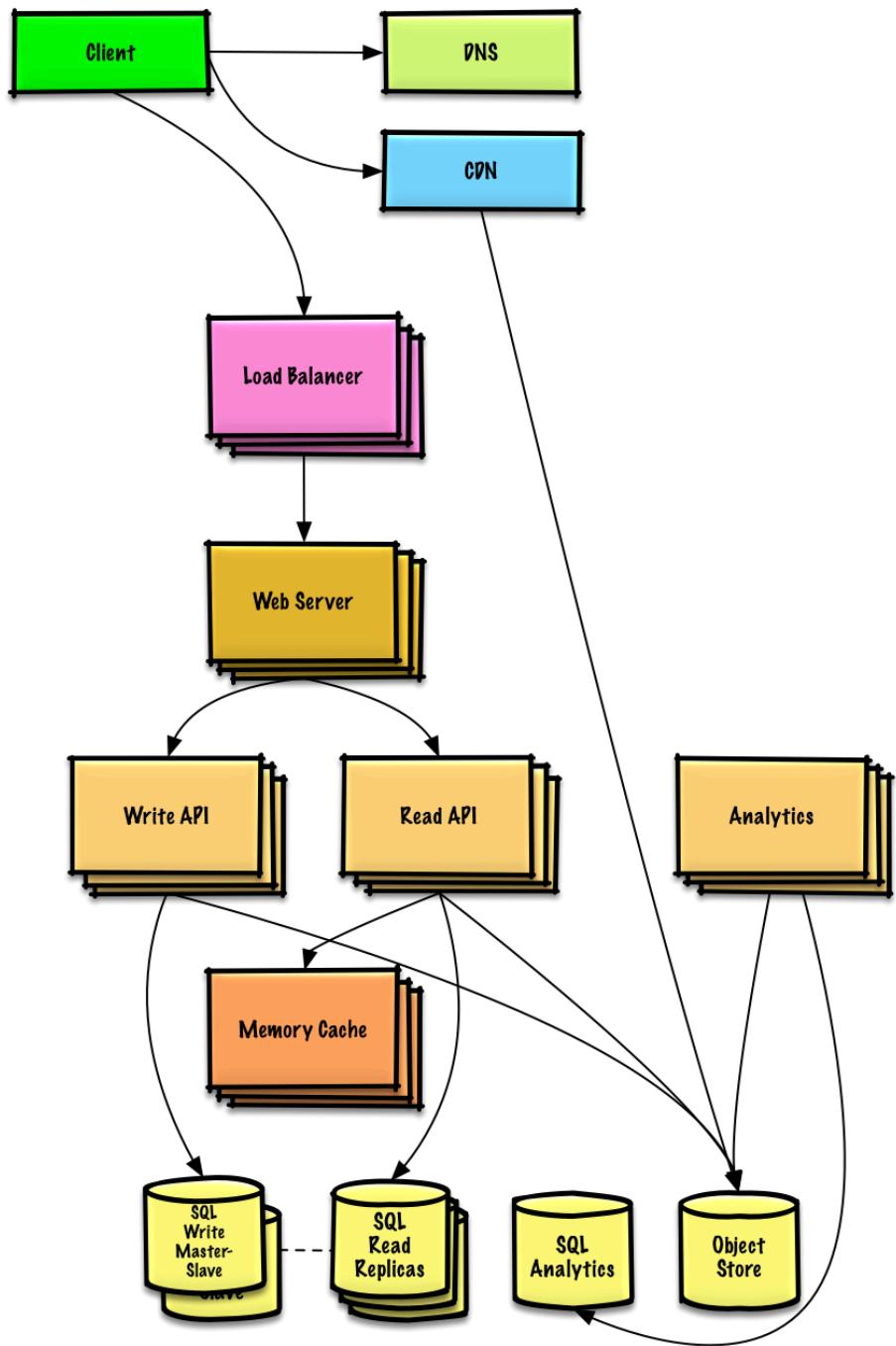
Common system design interview questions with sample discussions, code, and diagrams.

Solutions linked to content in the `solutions/` folder.

Question	
Design Pastebin.com (or Bit.ly)	Solution
Design the Twitter timeline and search (or Facebook feed and search)	Solution
Design a web crawler	Solution
Design Mint.com	Solution
Design the data structures for a social network	Solution
Design a key-value store for a search engine	Solution
Design Amazon's sales ranking by category feature	Solution
Design a system that scales to millions of users on AWS	Solution
Add a system design question	Contribute

Design Pastebin.com (or Bit.ly)

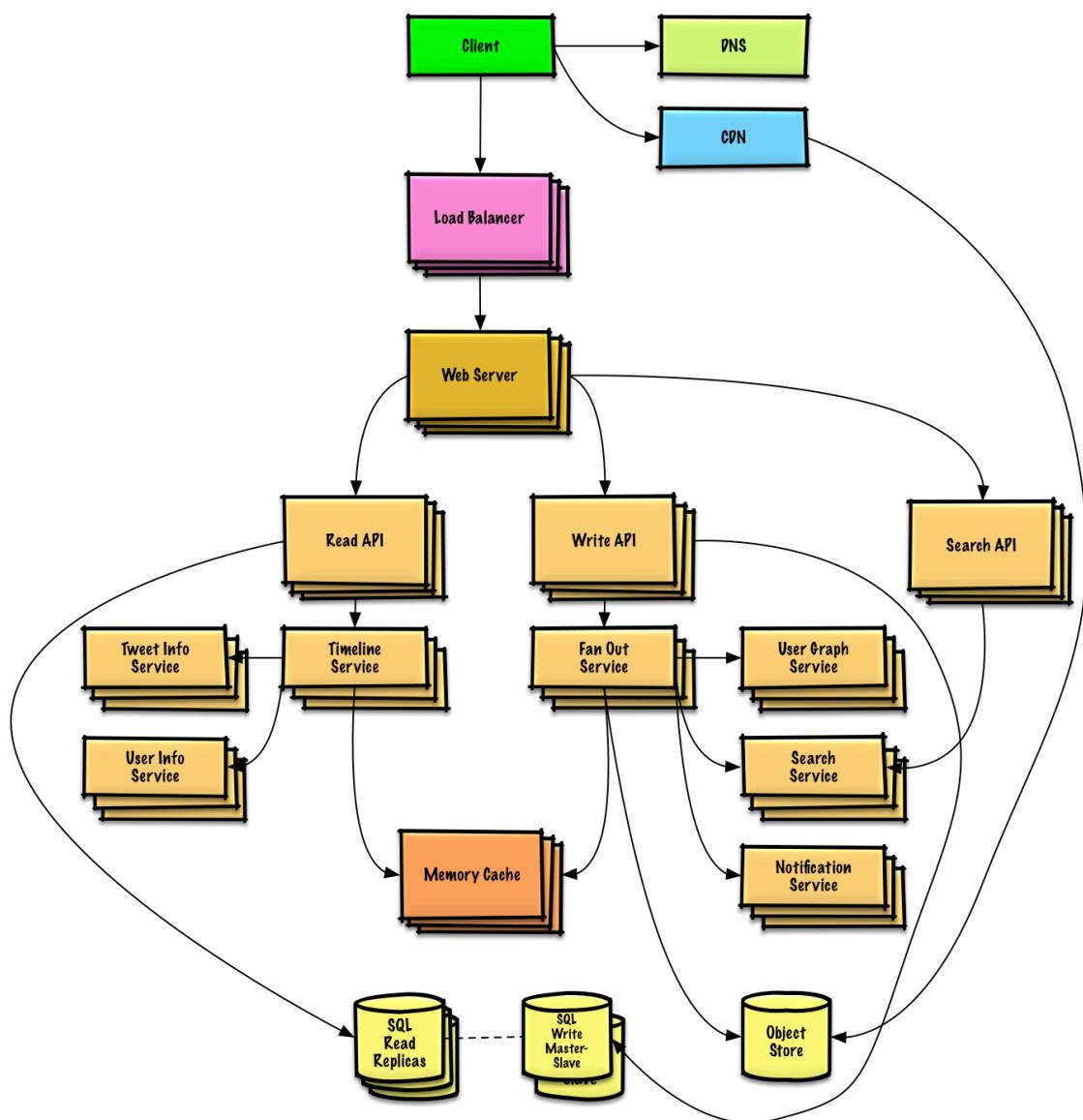
[View exercise and solution](#)



Imgur

Design the Twitter timeline and search (or Facebook feed and search)

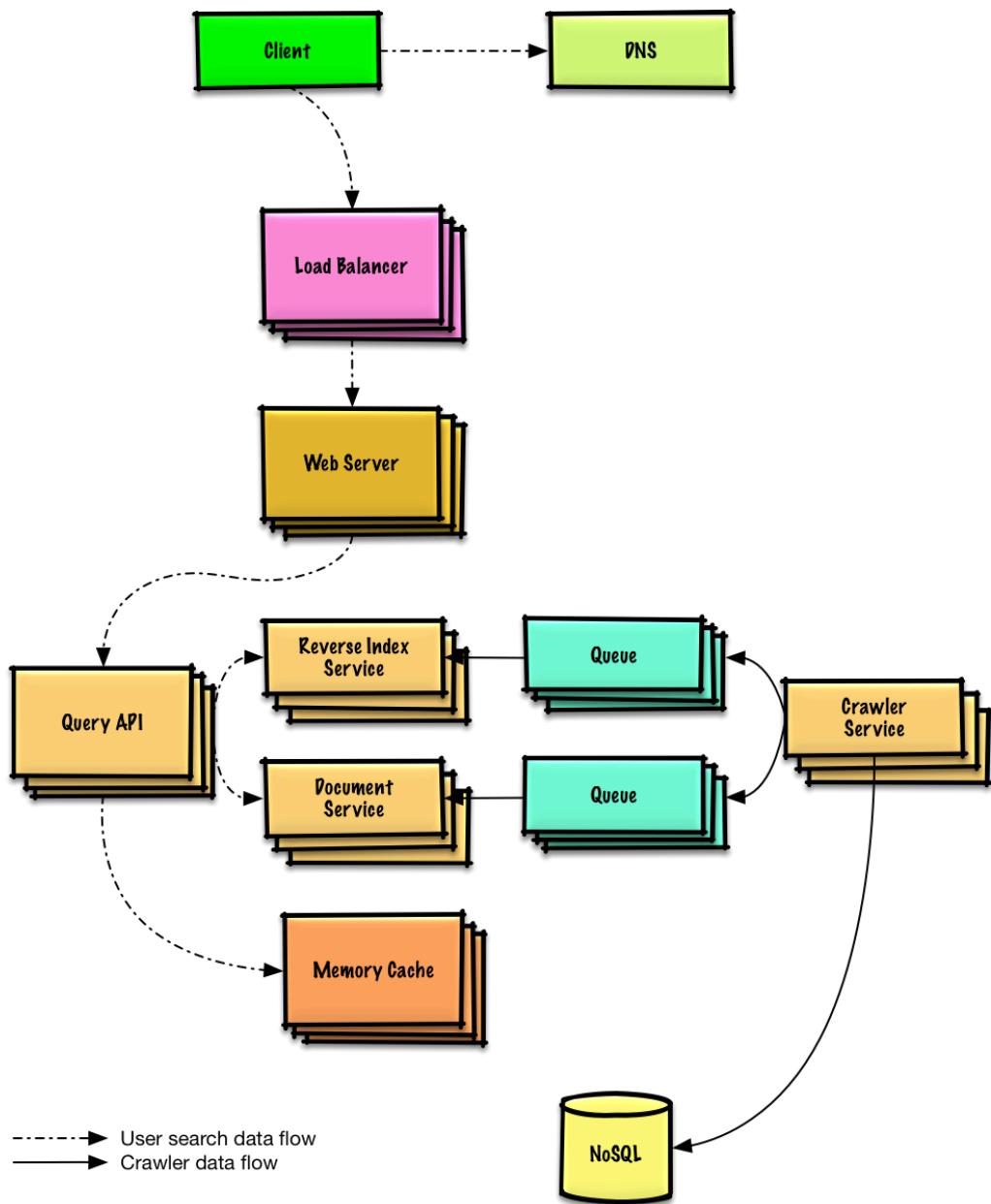
[View exercise and solution](#)



Imgur

Design a web crawler

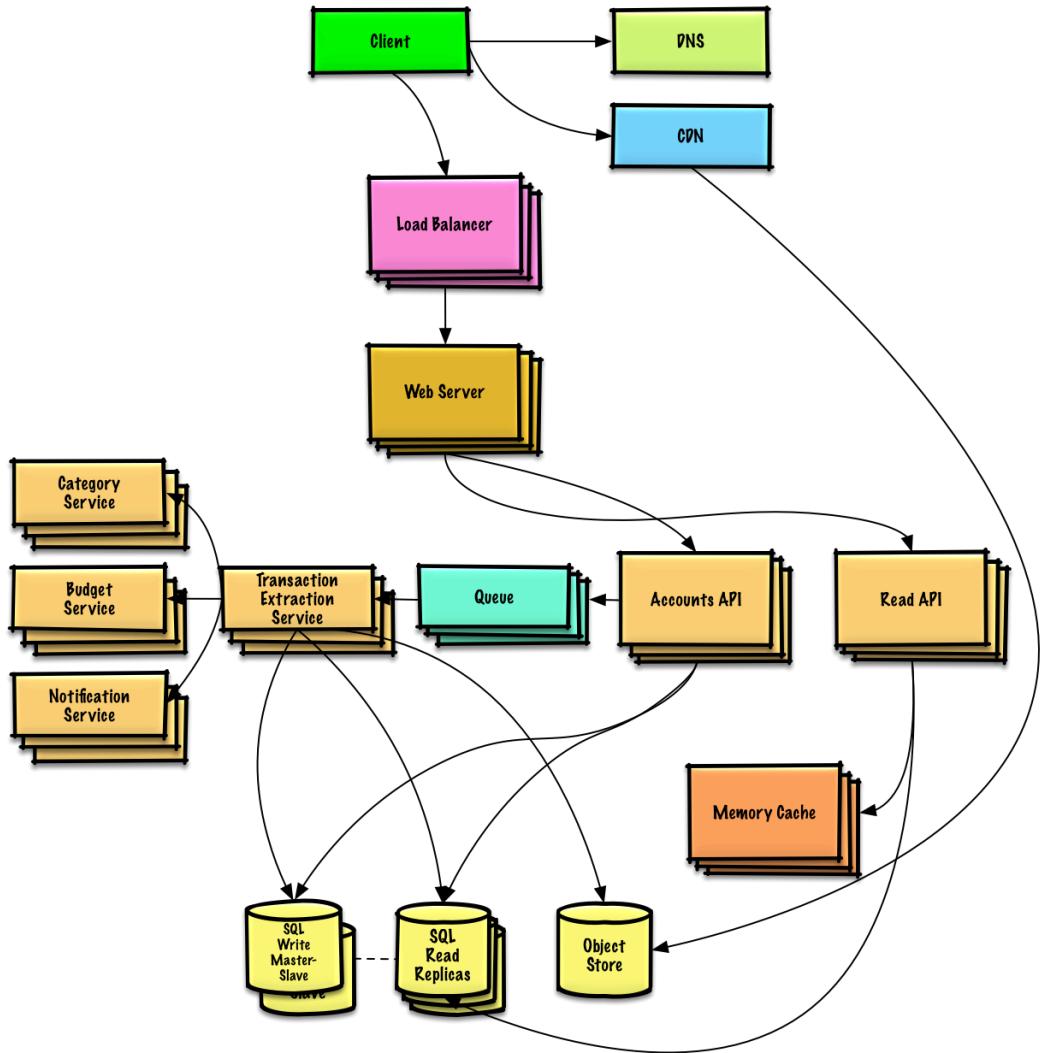
[View exercise and solution](#)



Imgur

Design Mint.com

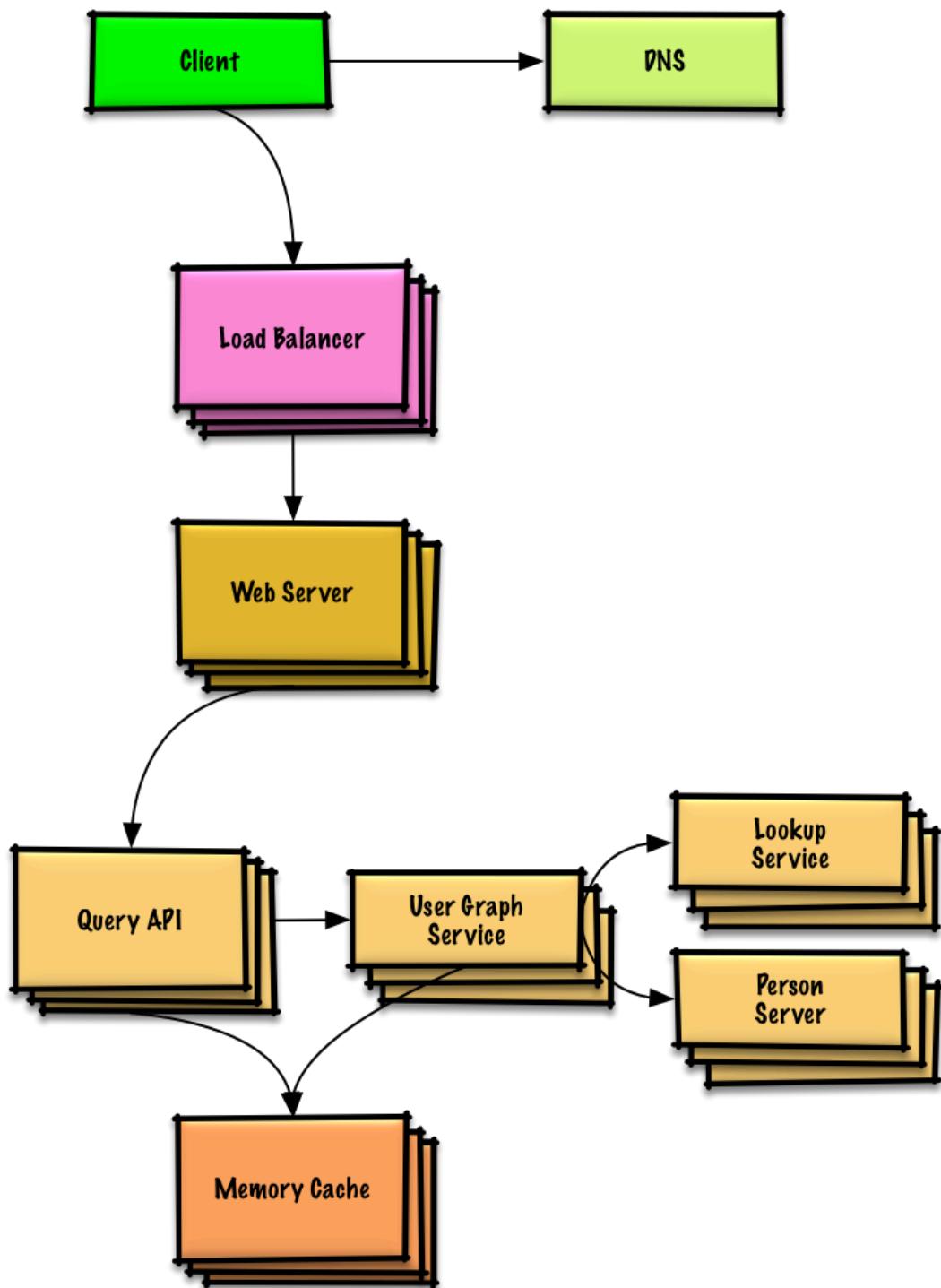
[View exercise and solution](#)



Imgur

Design the data structures for a social network

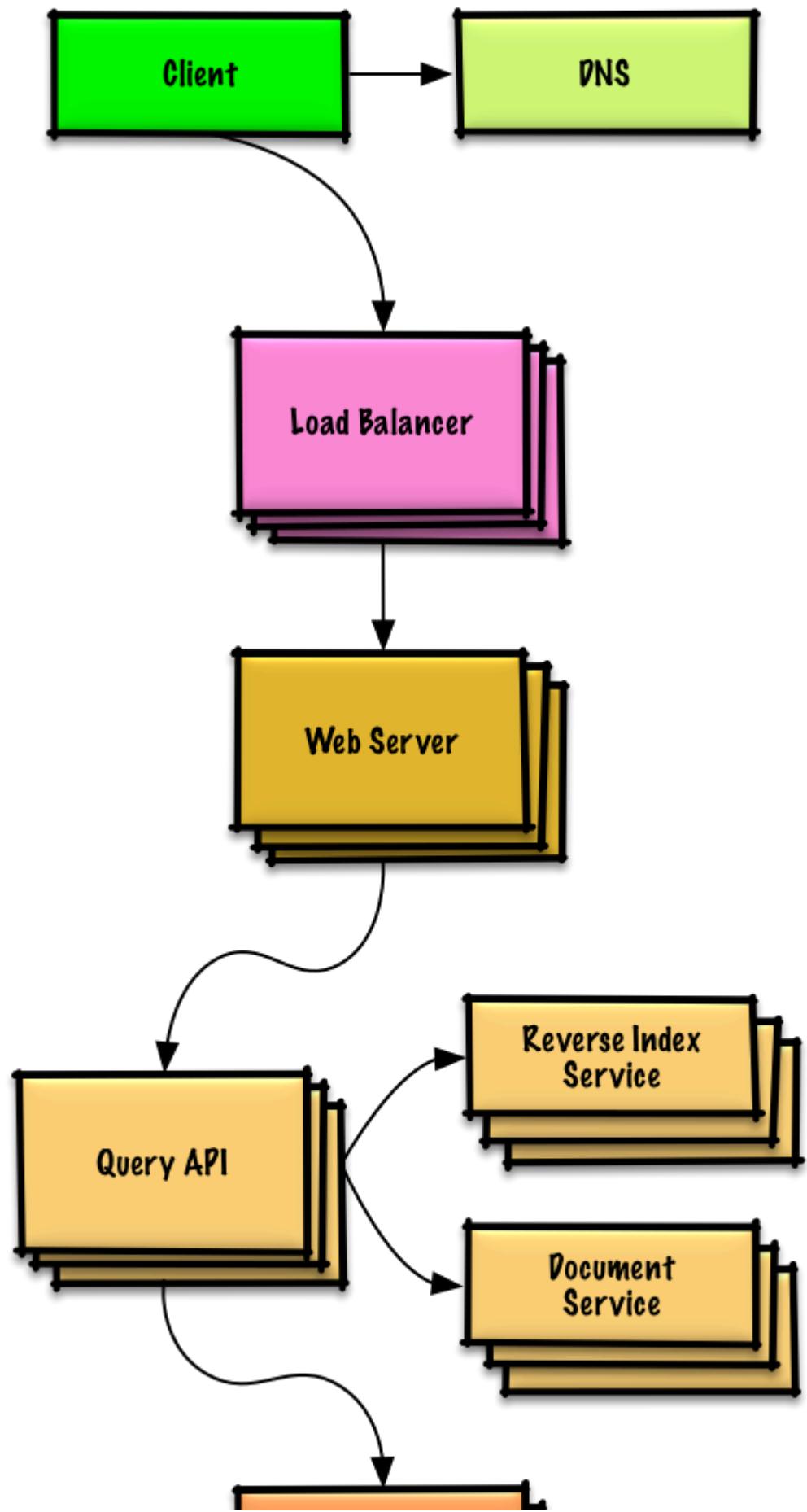
[View exercise and solution](#)



Imgur

Design a key-value store for a search engine

[View exercise and solution](#)

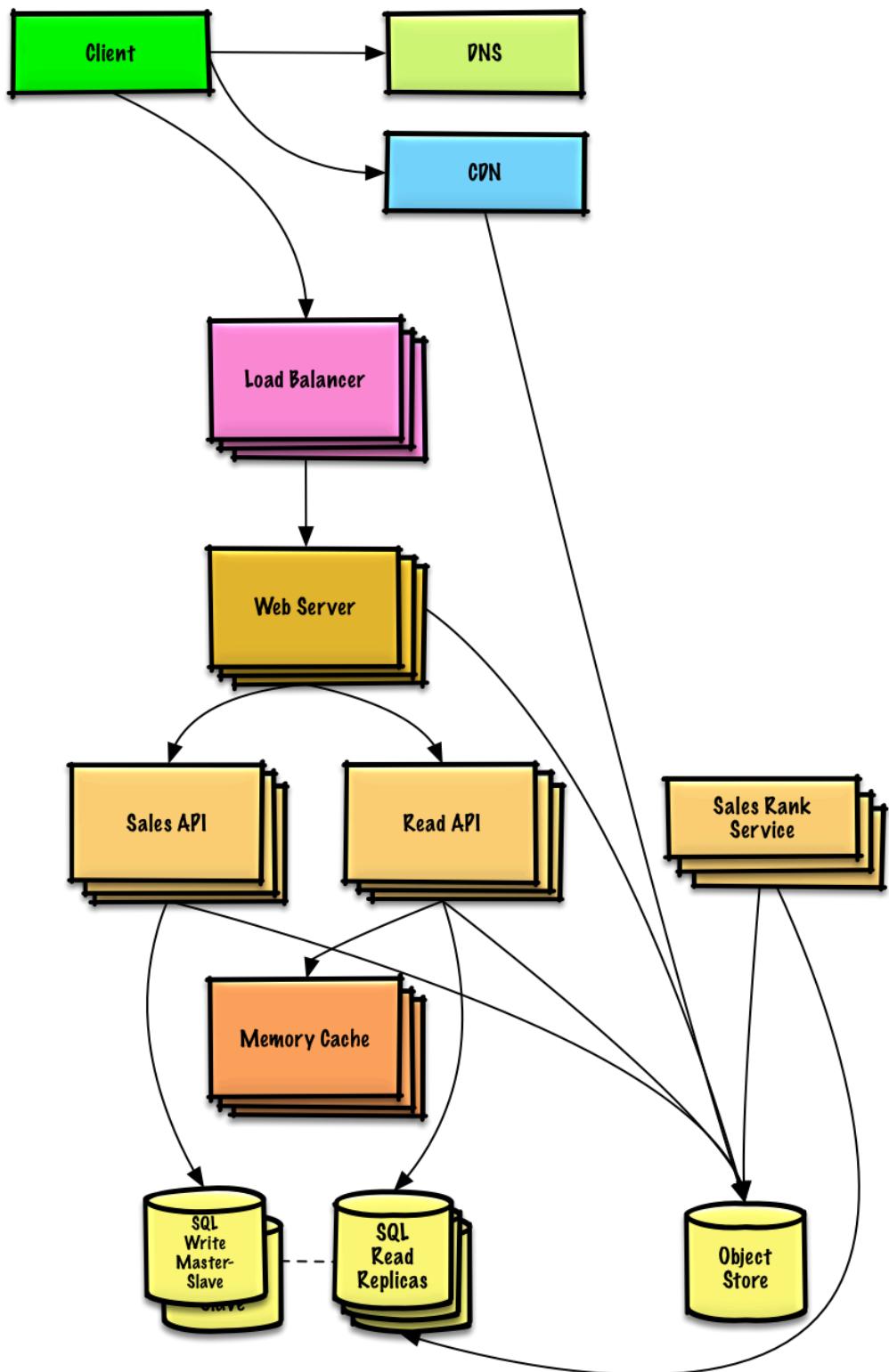




Imgur

Design Amazon's sales ranking by category feature

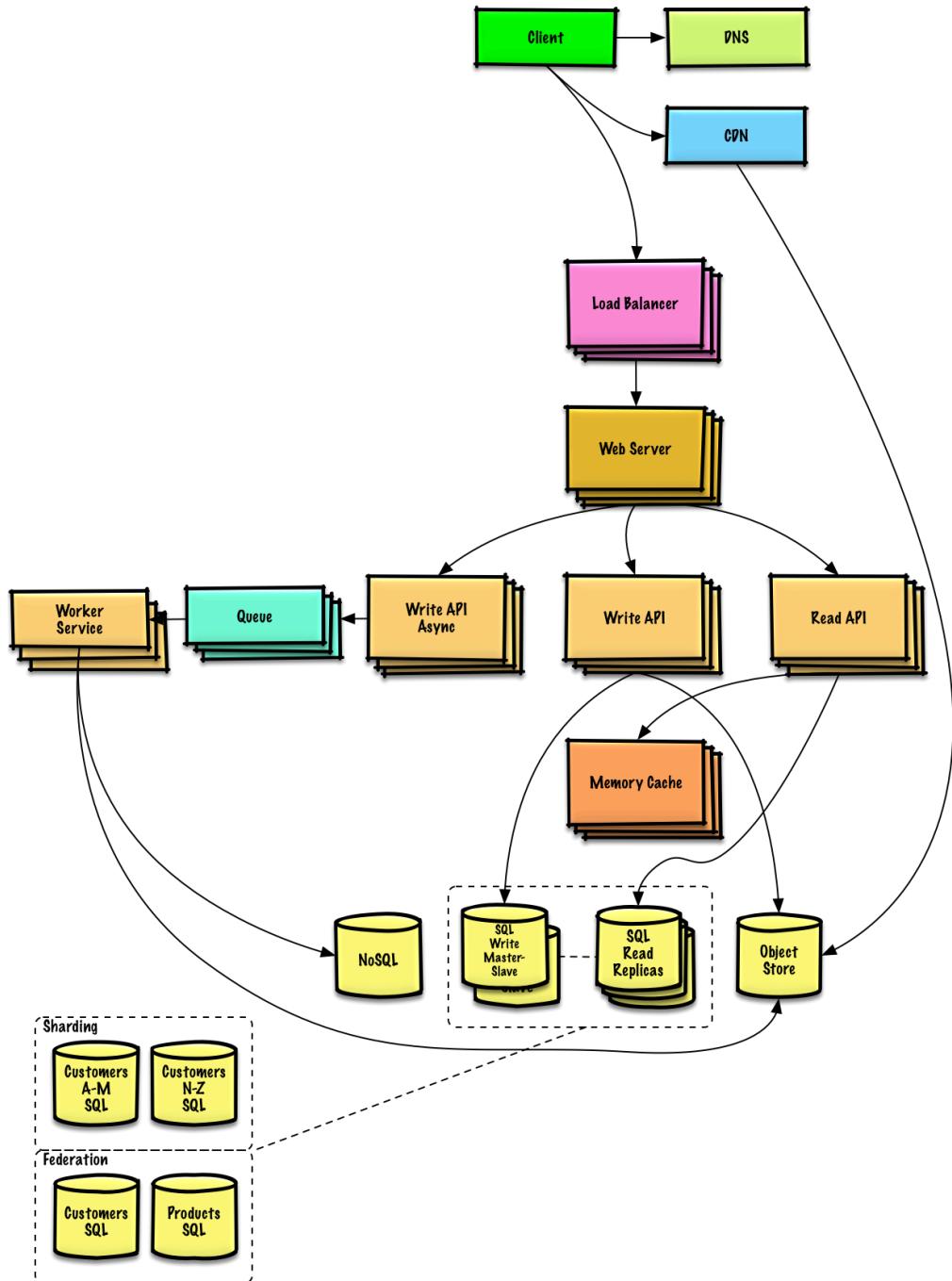
[View exercise and solution](#)



Imgur

Design a system that scales to millions of users on AWS

[View exercise and solution](#)



Imgur

Object-oriented design interview questions with solutions

Common object-oriented design interview questions with sample discussions, code, and diagrams.

Solutions linked to content in the `solutions/` folder.

Note: This section is under development

Question	
Design a hash map	Solution
Design a least recently used cache	Solution
Design a call center	Solution
Design a deck of cards	Solution
Design a parking lot	Solution
Design a chat server	Solution
Design a circular array	Contribute
Add an object-oriented design question	Contribute

System design topics: start here

New to system design?

First, you'll need a basic understanding of common principles, learning about what they are, how they are used, and their pros and cons.

Step 1: Review the scalability video lecture

[Scalability Lecture at Harvard](#)

- Topics covered:
 - Vertical scaling
 - Horizontal scaling
 - Caching
 - Load balancing
 - Database replication
 - Database partitioning

Step 2: Review the scalability article

[Scalability](#)

- Topics covered:
 - [Clones](#)
 - [Databases](#)
 - [Caches](#)
 - [Asynchronism](#)

Next steps

Next, we'll look at high-level trade-offs:

- **Performance vs scalability**
- **Latency vs throughput**
- **Availability vs consistency**

Keep in mind that **everything is a trade-off**.

Then we'll dive into more specific topics such as DNS, CDNs, and load balancers.

Performance vs scalability

A service is **scalable** if it results in increased **performance** in a manner proportional to resources added. Generally, increasing performance means serving more units of work, but it can also be to handle larger units of work, such as when datasets grow.¹

Another way to look at performance vs scalability:

- If you have a **performance** problem, your system is slow for a single user.
- If you have a **scalability** problem, your system is fast for a single user but slow under heavy load.

Source(s) and further reading

- [A word on scalability](#)
- [Scalability, availability, stability, patterns](#)

Latency vs throughput

Latency is the time to perform some action or to produce some result.

Throughput is the number of such actions or results per unit of time.

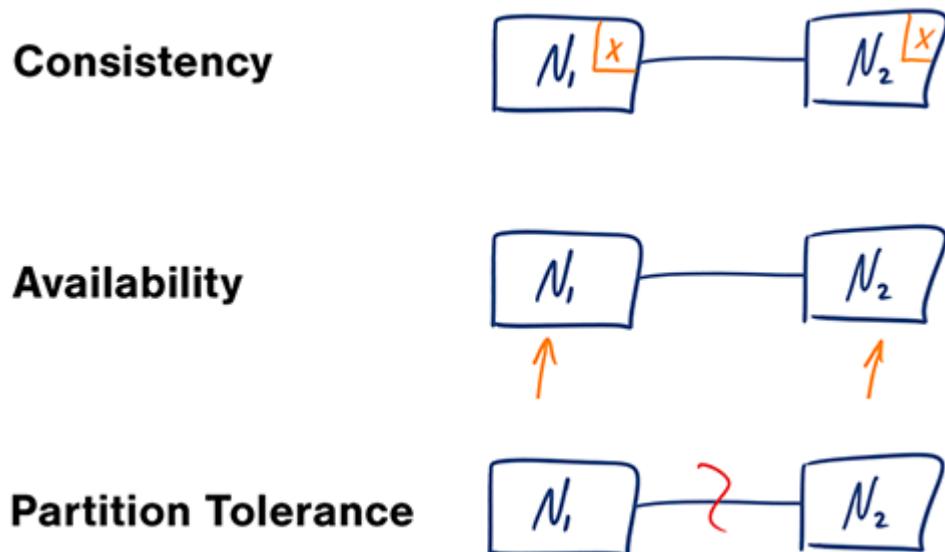
Generally, you should aim for **maximal throughput** with **acceptable latency**.

Source(s) and further reading

- [Understanding latency vs throughput](#)

Availability vs consistency

CAP theorem



Source: CAP theorem revisited

In a distributed computer system, you can only support two of the following guarantees:

- **Consistency** - Every read receives the most recent write or an error
- **Availability** - Every request receives a response, without guarantee that it contains the most recent version of the information
- **Partition Tolerance** - The system continues to operate despite arbitrary partitioning due to network failures

Networks aren't reliable, so you'll need to support partition tolerance. You'll need to make a software tradeoff between consistency and availability.

CP - consistency and partition tolerance

Waiting for a response from the partitioned node might result in a timeout error. CP is a good choice if your business needs require atomic reads and writes.

AP - availability and partition tolerance

Responses return the most readily available version of the data available on any node, which might not be the latest. Writes might take some time to propagate when the partition is resolved.

AP is a good choice if the business needs to allow for [eventual consistency](#) or when the system needs to continue working despite external errors.

Source(s) and further reading

- [CAP theorem revisited](#)
- [A plain english introduction to CAP theorem](#)
- [CAP FAQ](#)
- [The CAP theorem](#)

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.

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.

Source(s) and further reading

- [Transactions across data centers](#)

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

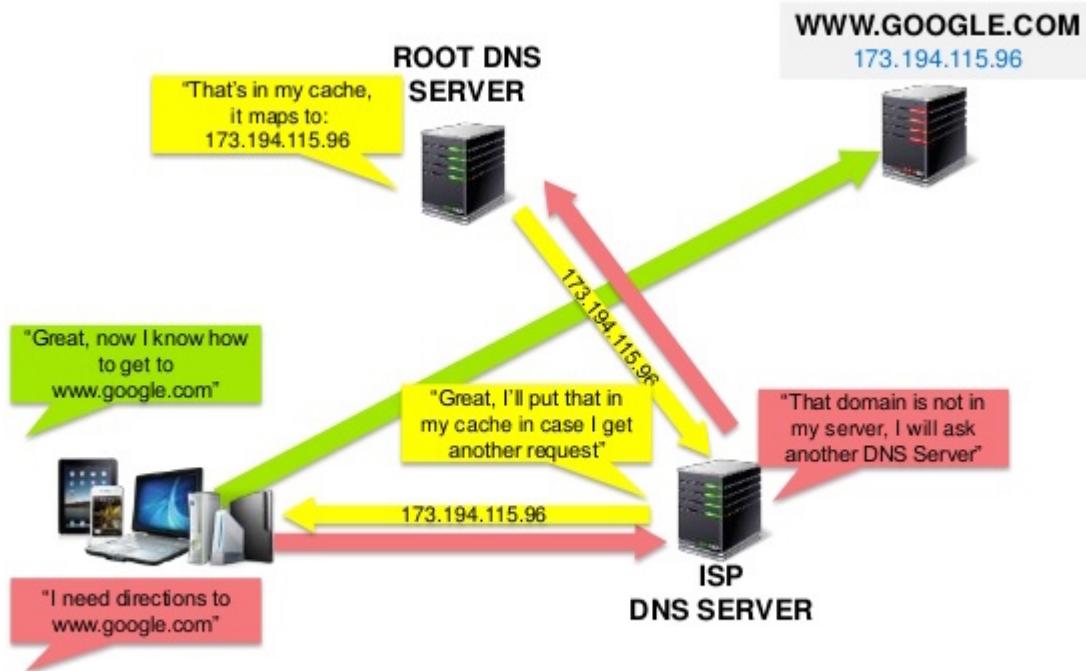
Overall availability increases when two components with availability < 100% are in parallel:

$$\text{Availability (Total)} = 1 - (1 - \text{Availability (Foo)}) * (1 - \text{Availability (Bar)})$$

If both Foo and Bar each had 99.9% availability, their total availability in parallel would be 99.9999%.

Domain name system

How Does DNS Work?



Source: DNS security presentation

A Domain Name System (DNS) translates a domain name such as www.example.com to an IP address.

DNS is hierarchical, with a few authoritative servers at the top level. Your router or ISP provides information about which DNS server(s) to contact when doing a lookup. Lower level DNS servers cache mappings, which could become stale due to DNS propagation delays. DNS results can also be cached by your browser or OS for a certain period of time, determined by the [time to live \(TTL\)](#).

- **NS record (name server)** - Specifies the DNS servers for your domain/subdomain.
- **MX record (mail exchange)** - Specifies the mail servers for accepting messages.
- **A record (address)** - Points a name to an IP address.
- **CNAME (canonical)** - Points a name to another name or CNAME (example.com to www.example.com) or to an A record.

Services such as [CloudFlare](#) and [Route 53](#) provide managed DNS services. Some DNS services can route traffic through various methods:

- [Weighted round robin](#)
 - Prevent traffic from going to servers under maintenance
 - Balance between varying cluster sizes
 - A/B testing
- [Latency-based](#)
- [Geolocation-based](#)

Disadvantage(s): DNS

- Accessing a DNS server introduces a slight delay, although mitigated by caching described above.
- DNS server management could be complex and is generally managed by [governments, ISPs, and large companies](#).
- DNS services have recently come under [DDoS attack](#), preventing users from accessing websites such as Twitter without knowing Twitter's IP address(es).

Source(s) and further reading

- [DNS architecture](#)
- [Wikipedia](#)
- [DNS articles](#)

Content delivery network



[Source: Why use a CDN](#)

A content delivery network (CDN) is a globally distributed network of proxy servers, serving content from locations closer to the user. Generally, static files such as HTML/CSS/JS, photos, and videos are served from CDN, although some CDNs such as Amazon's CloudFront support dynamic content. The site's DNS resolution will tell clients which server to contact.

Serving content from CDNs can significantly improve performance in two ways:

- Users receive content from data centers close to them
- Your servers do not have to serve requests that the CDN fulfills

Push CDNs

Push CDNs receive new content whenever changes occur on your server. You take full responsibility for providing content, uploading directly to the CDN and rewriting URLs to point to the CDN. You can configure when content expires and when it is updated. Content is uploaded only when it is new or changed, minimizing traffic, but maximizing storage.

Sites with a small amount of traffic or sites with content that isn't often updated work well with push CDNs. Content is placed on the CDNs once, instead of being re-pulled at regular intervals.

Pull CDNs

Pull CDNs grab new content from your server when the first user requests the content. You leave the content on your server and rewrite URLs to point to the CDN. This results in a slower request until the content is cached on the CDN.

A [time-to-live \(TTL\)](#) determines how long content is cached. Pull CDNs minimize storage space on the CDN, but can create redundant traffic if files expire and are pulled before they have actually changed.

Sites with heavy traffic work well with pull CDNs, as traffic is spread out more evenly with only recently-requested content remaining on the CDN.

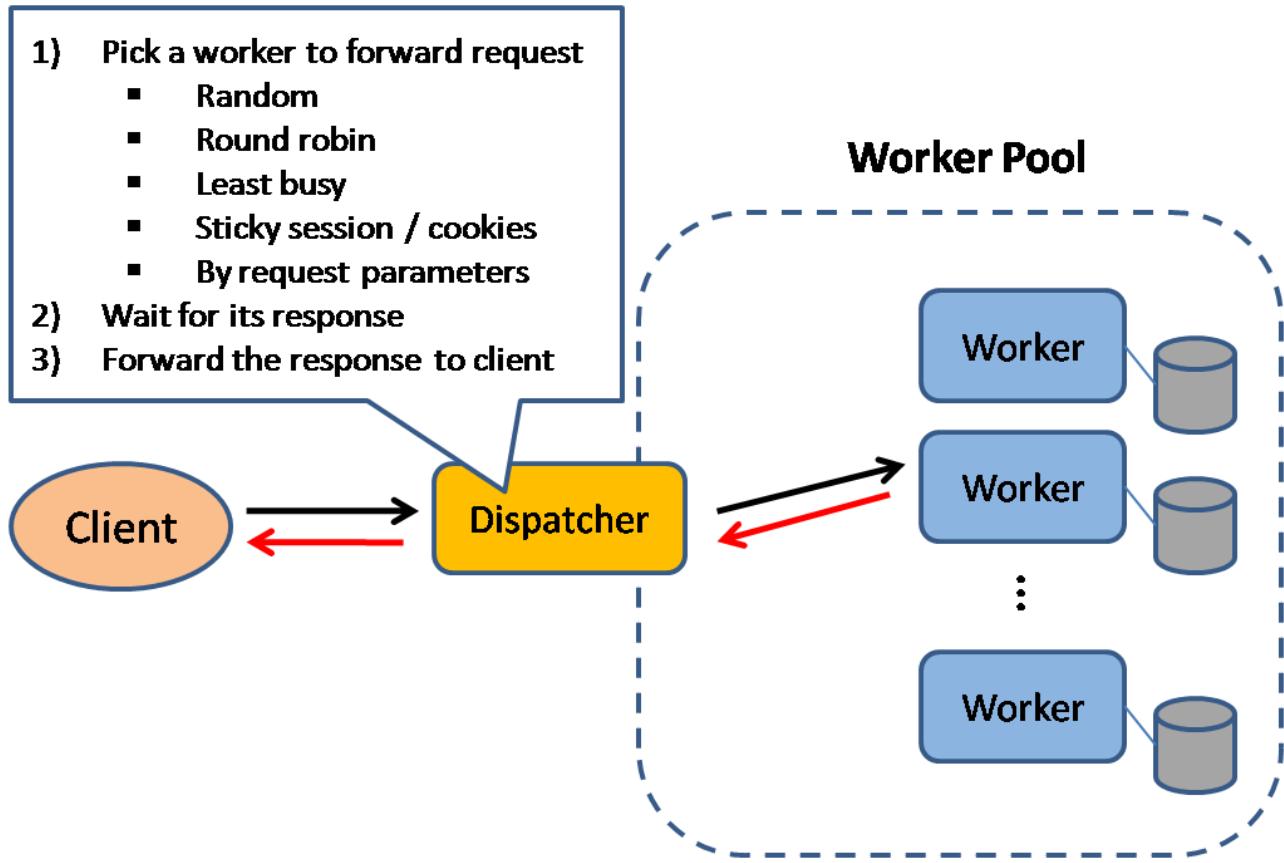
Disadvantage(s): CDN

- CDN costs could be significant depending on traffic, although this should be weighed with additional costs you would incur not using a CDN.
- Content might be stale if it is updated before the TTL expires it.
- CDNs require changing URLs for static content to point to the CDN.

Source(s) and further reading

- [Globally distributed content delivery](#)
- [The differences between push and pull CDNs](#)
- [Wikipedia](#)

Load balancer



Source: Scalable system design patterns

Load balancers distribute incoming client requests to computing resources such as application servers and databases. In each case, the load balancer returns the response from the computing resource to the appropriate client. Load balancers are effective at:

- Preventing requests from going to unhealthy servers
- Preventing overloading resources
- Helping to eliminate a single point of failure

Load balancers can be implemented with hardware (expensive) or with software such as HAProxy.

Additional benefits include:

- **SSL termination** – Decrypt incoming requests and encrypt server responses so backend servers do not have to perform these potentially expensive operations
 - Removes the need to install [X.509 certificates](#) on each server

- **Session persistence** - Issue cookies and route a specific client's requests to same instance if the web apps do not keep track of sessions

To protect against failures, it's common to set up multiple load balancers, either in [active-passive](#) or [active-active](#) mode.

Load balancers can route traffic based on various metrics, including:

- Random
- Least loaded
- Session/cookies
- [Round robin or weighted round robin](#)
- [Layer 4](#)
- [Layer 7](#)

Layer 4 load balancing

Layer 4 load balancers look at info at the [transport layer](#) to decide how to distribute requests. Generally, this involves the source, destination IP addresses, and ports in the header, but not the contents of the packet. Layer 4 load balancers forward network packets to and from the upstream server, performing [Network Address Translation \(NAT\)](#).

Layer 7 load balancing

Layer 7 load balancers look at the [application layer](#) to decide how to distribute requests. This can involve contents of the header, message, and cookies. Layer 7 load balancers terminate network traffic, reads the message, makes a load-balancing decision, then opens a connection to the selected server. For example, a layer 7 load balancer can direct video traffic to servers that host videos while directing more sensitive user billing traffic to security-hardened servers.

At the cost of flexibility, layer 4 load balancing requires less time and computing resources than Layer 7, although the performance impact can be minimal on modern commodity hardware.

Horizontal scaling

Load balancers can also help with horizontal scaling, improving performance and availability. Scaling out using commodity machines is more cost efficient and results in

higher availability than scaling up a single server on more expensive hardware, called **Vertical Scaling**. It is also easier to hire for talent working on commodity hardware than it is for specialized enterprise systems.

Disadvantage(s): horizontal scaling

- Scaling horizontally introduces complexity and involves cloning servers
 - Servers should be stateless: they should not contain any user-related data like sessions or profile pictures
 - Sessions can be stored in a centralized data store such as a [database](#) (SQL, NoSQL) or a persistent [cache](#) (Redis, Memcached)
- Downstream servers such as caches and databases need to handle more simultaneous connections as upstream servers scale out

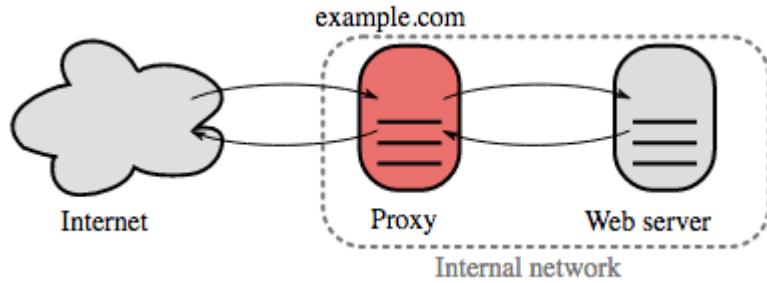
Disadvantage(s): load balancer

- The load balancer can become a performance bottleneck if it does not have enough resources or if it is not configured properly.
- Introducing a load balancer to help eliminate a single point of failure results in increased complexity.
- A single load balancer is a single point of failure, configuring multiple load balancers further increases complexity.

Source(s) and further reading

- [NGINX architecture](#)
- [HAProxy architecture guide](#)
- [Scalability](#)
- [Wikipedia](#)
- [Layer 4 load balancing](#)
- [Layer 7 load balancing](#)
- [ELB listener config](#)

Reverse proxy (web server)



[Source: Wikipedia](#)

A reverse proxy is a web server that centralizes internal services and provides unified interfaces to the public. Requests from clients are forwarded to a server that can fulfill it before the reverse proxy returns the server's response to the client.

Additional benefits include:

- **Increased security** - Hide information about backend servers, blacklist IPs, limit number of connections per client
- **Increased scalability and flexibility** - Clients only see the reverse proxy's IP, allowing you to scale servers or change their configuration
- **SSL termination** - Decrypt incoming requests and encrypt server responses so backend servers do not have to perform these potentially expensive operations
 - Removes the need to install [X.509 certificates](#) on each server
- **Compression** - Compress server responses
- **Caching** - Return the response for cached requests
- **Static content** - Serve static content directly
 - HTML/CSS/JS
 - Photos
 - Videos
 - Etc

Load balancer vs reverse proxy

- Deploying a load balancer is useful when you have multiple servers. Often, load balancers route traffic to a set of servers serving the same function.

- Reverse proxies can be useful even with just one web server or application server, opening up the benefits described in the previous section.
- Solutions such as NGINX and HAProxy can support both layer 7 reverse proxying and load balancing.

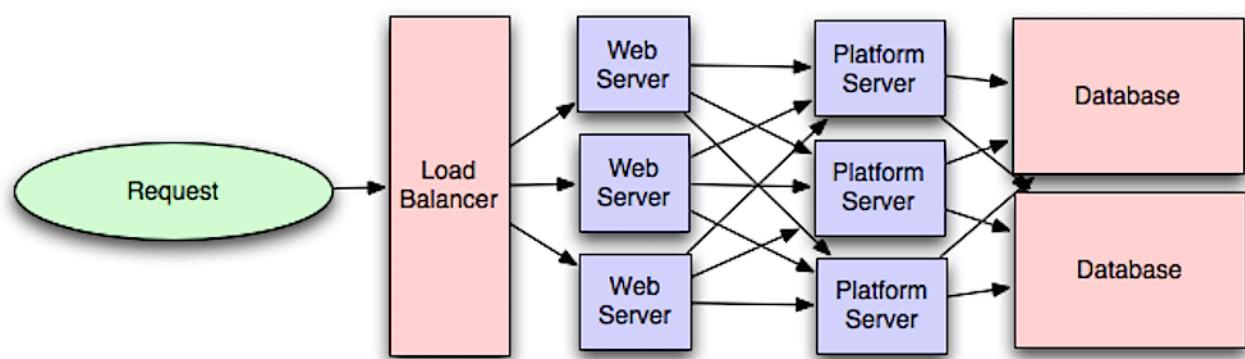
Disadvantage(s): reverse proxy

- Introducing a reverse proxy results in increased complexity.
- A single reverse proxy is a single point of failure, configuring multiple reverse proxies (ie a [failover](#)) further increases complexity.

Source(s) and further reading

- [Reverse proxy vs load balancer](#)
- [NGINX architecture](#)
- [HAProxy architecture guide](#)
- [Wikipedia](#)

Application layer



Source: [Intro to architecting systems for scale](#)

Separating out the web layer from the application layer (also known as platform layer) allows you to scale and configure both layers independently. Adding a new API results in adding application servers without necessarily adding additional web servers. The **single responsibility principle** advocates for small and autonomous services that work together. Small teams with small services can plan more aggressively for rapid growth.

Workers in the application layer also help enable [asynchronism](#).

Microservices

Related to this discussion are [microservices](#), which can be described as a suite of independently deployable, small, modular services. Each service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal.¹

Pinterest, for example, could have the following microservices: user profile, follower, feed, search, photo upload, etc.

Service Discovery

Systems such as [Consul](#), [Etcd](#), and [Zookeeper](#) can help services find each other by keeping track of registered names, addresses, and ports. [Health checks](#) help verify service integrity and are often done using an [HTTP](#) endpoint. Both Consul and Etcd have a built in [key-value store](#) that can be useful for storing config values and other shared data.

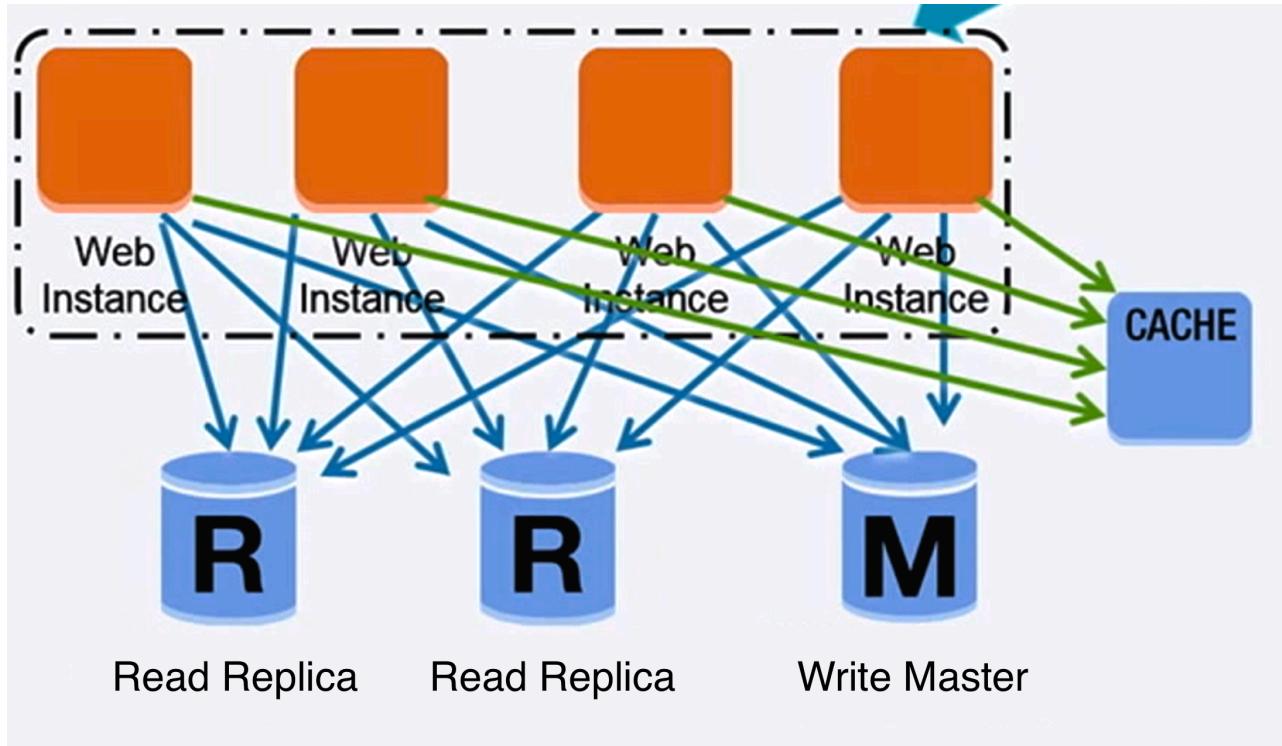
Disadvantage(s): application layer

- Adding an application layer with loosely coupled services requires a different approach from an architectural, operations, and process viewpoint (vs a monolithic system).
- Microservices can add complexity in terms of deployments and operations.

Source(s) and further reading

- [Intro to architecting systems for scale](#)
- [Crack the system design interview](#)
- [Service oriented architecture](#)
- [Introduction to Zookeeper](#)
- [Here's what you need to know about building microservices](#)

Database



Source: Scaling up to your first 10 million users

Relational database management system (RDBMS)

A relational database like SQL is a collection of data items organized in tables.

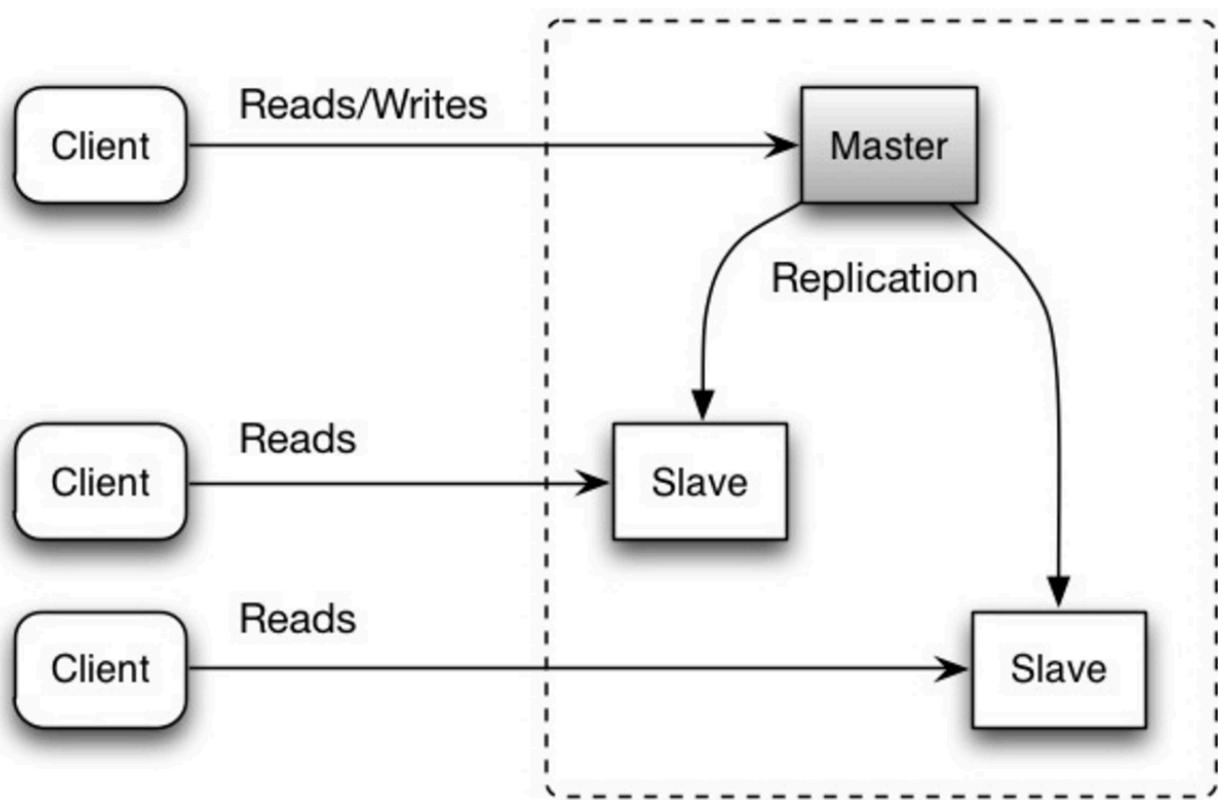
ACID is a set of properties of relational database [transactions](#).

- **Atomicity** - Each transaction is all or nothing
- **Consistency** - Any transaction will bring the database from one valid state to another
- **Isolation** - Executing transactions concurrently has the same results as if the transactions were executed serially
- **Durability** - Once a transaction has been committed, it will remain so

There are many techniques to scale a relational database: **master-slave replication**, **master-master replication**, **federation**, **sharding**, **denormalization**, and **SQL tuning**.

Master-slave replication

The master serves reads and writes, replicating writes to one or more slaves, which serve only reads. Slaves can also replicate to additional slaves in a tree-like fashion. If the master goes offline, the system can continue to operate in read-only mode until a slave is promoted to a master or a new master is provisioned.



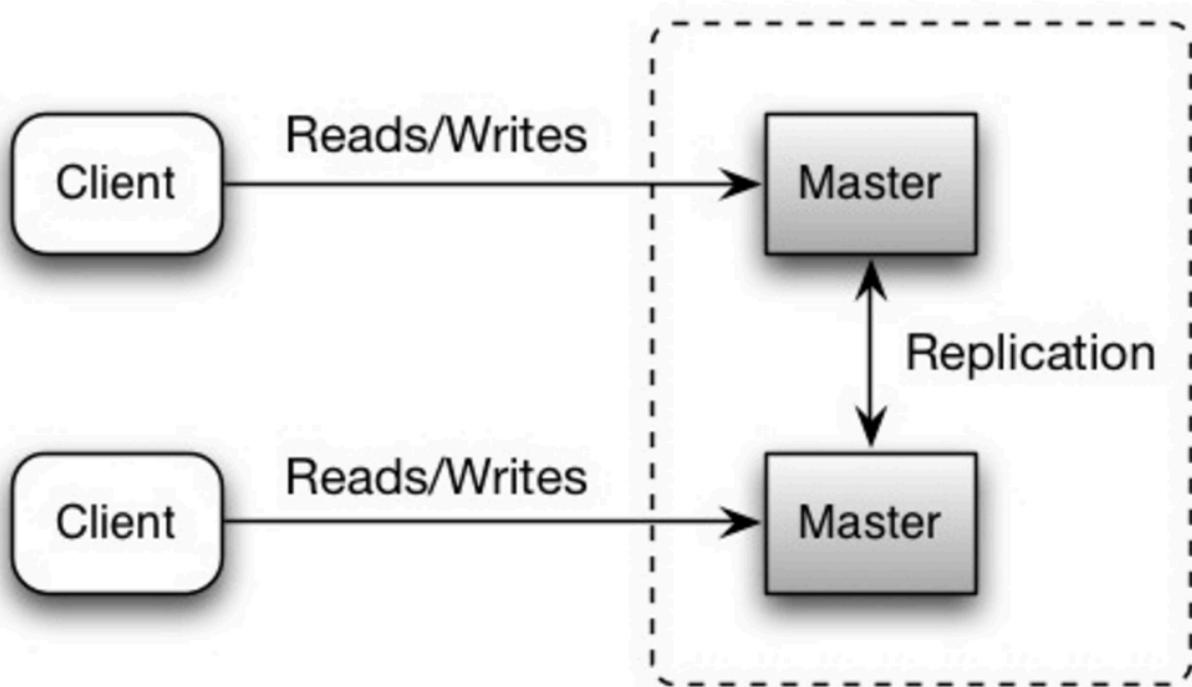
[Source: Scalability, availability, stability, patterns](#)

Disadvantage(s): master-slave replication

- Additional logic is needed to promote a slave to a master.
- See [Disadvantage\(s\): replication](#) for points related to **both** master-slave and master-master.

Master-master replication

Both masters serve reads and writes and coordinate with each other on writes. If either master goes down, the system can continue to operate with both reads and writes.



[Source: Scalability, availability, stability, patterns](#)

Disadvantage(s): master-master replication

- You'll need a load balancer or you'll need to make changes to your application logic to determine where to write.
- Most master-master systems are either loosely consistent (violating ACID) or have increased write latency due to synchronization.
- Conflict resolution comes more into play as more write nodes are added and as latency increases.
- See [Disadvantage\(s\): replication](#) for points related to **both** master-slave and master-master.

Disadvantage(s): replication

- There is a potential for loss of data if the master fails before any newly written data can be replicated to other nodes.
- Writes are replayed to the read replicas. If there are a lot of writes, the read replicas can get bogged down with replaying writes and can't do as many reads.
- The more read slaves, the more you have to replicate, which leads to greater replication lag.

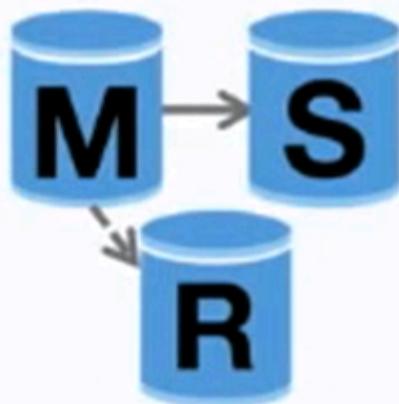
- On some systems, writing to the master can spawn multiple threads to write in parallel, whereas read replicas only support writing sequentially with a single thread.
- Replication adds more hardware and additional complexity.

Source(s) and further reading: replication

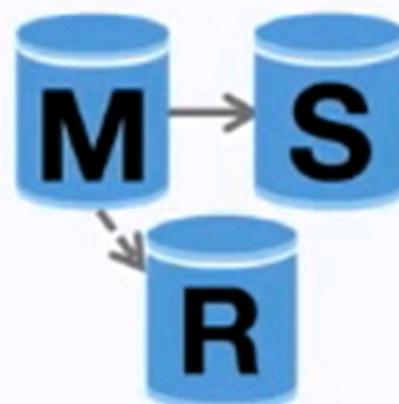
- [Scalability, availability, stability, patterns](#)
- [Multi-master replication](#)

Federation

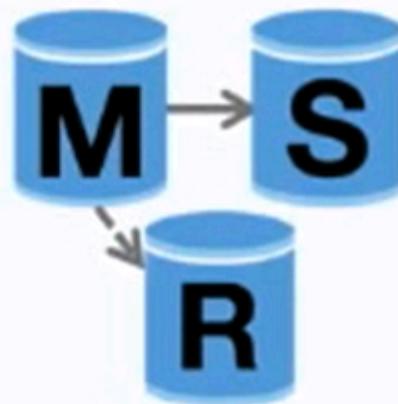
Forums DB



Users DB



Products
DB



[Source: Scaling up to your first 10 million users](#)

Federation (or functional partitioning) splits up databases by function. For example, instead of a single, monolithic database, you could have three databases: **forums**,

users, and **products**, resulting in less read and write traffic to each database and therefore less replication lag. Smaller databases result in more data that can fit in memory, which in turn results in more cache hits due to improved cache locality. With no single central master serializing writes you can write in parallel, increasing throughput.

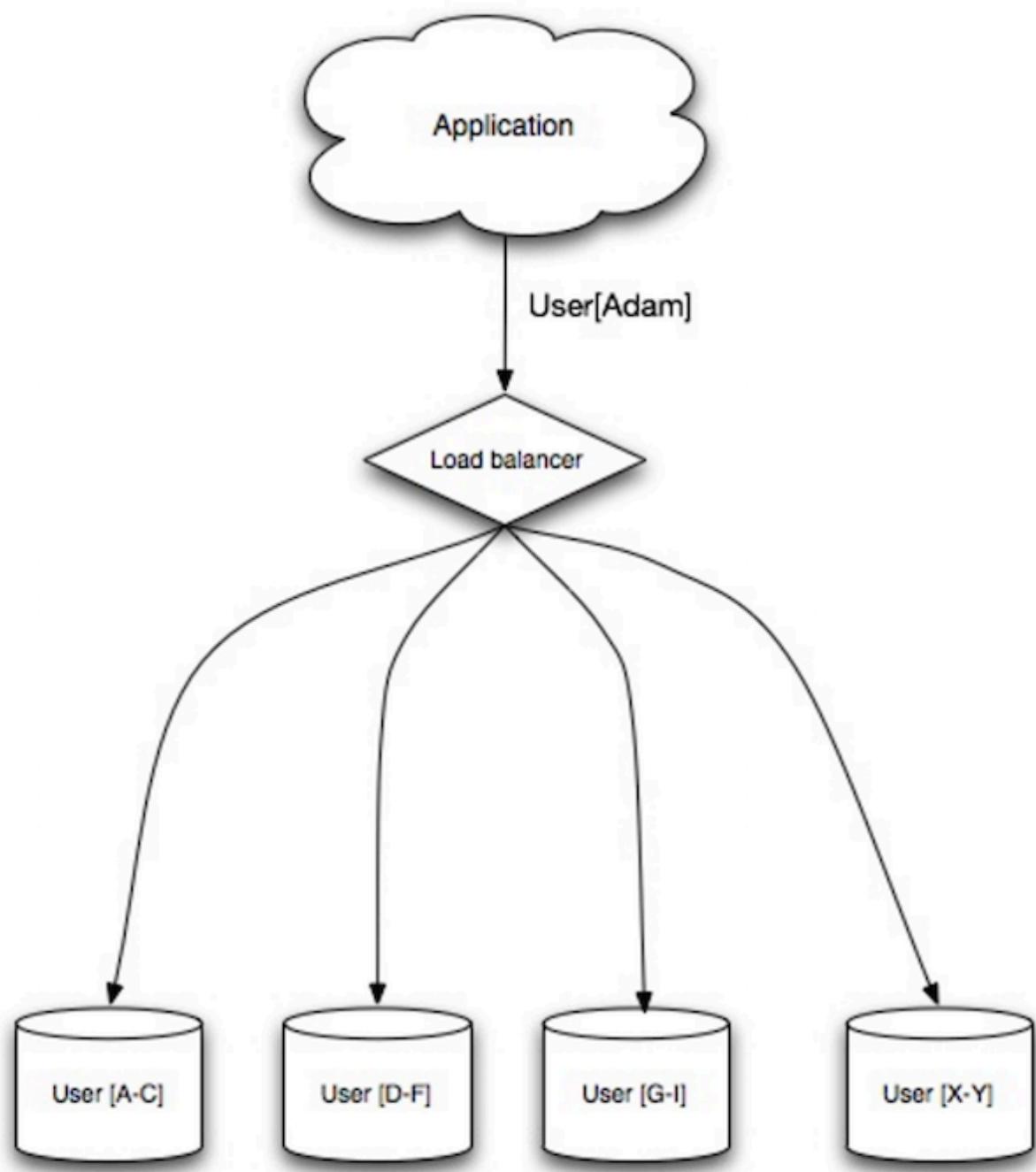
Disadvantage(s): federation

- Federation is not effective if your schema requires huge functions or tables.
- You'll need to update your application logic to determine which database to read and write.
- Joining data from two databases is more complex with a [server link](#).
- Federation adds more hardware and additional complexity.

Source(s) and further reading: federation

- [Scaling up to your first 10 million users](#)

Sharding



[Source: Scalability, availability, stability, patterns](#)

Sharding distributes data across different databases such that each database can only manage a subset of the data. Taking a users database as an example, as the number of users increases, more shards are added to the cluster.

Similar to the advantages of [federation](#), sharding results in less read and write traffic, less replication, and more cache hits. Index size is also reduced, which generally improves performance with faster queries. If one shard goes down, the other shards are still operational, although you'll want to add some form of replication to avoid data loss. Like federation, there is no single central master serializing writes, allowing you to write in parallel with increased throughput.

Common ways to shard a table of users is either through the user's last name initial or the user's geographic location.

Disadvantage(s): sharding

- You'll need to update your application logic to work with shards, which could result in complex SQL queries.
- Data distribution can become lopsided in a shard. For example, a set of power users on a shard could result in increased load to that shard compared to others.
 - Rebalancing adds additional complexity. A sharding function based on [consistent hashing](#) can reduce the amount of transferred data.
- Joining data from multiple shards is more complex.
- Sharding adds more hardware and additional complexity.

Source(s) and further reading: sharding

- [The coming of the shard](#)
- [Shard database architecture](#)
- [Consistent hashing](#)

Denormalization

Denormalization attempts to improve read performance at the expense of some write performance. Redundant copies of the data are written in multiple tables to avoid expensive joins. Some RDBMS such as [PostgreSQL](#) and Oracle support [materialized views](#) which handle the work of storing redundant information and keeping redundant copies consistent.

Once data becomes distributed with techniques such as [federation](#) and [sharding](#), managing joins across data centers further increases complexity. Denormalization might circumvent the need for such complex joins.

In most systems, reads can heavily outnumber writes 100:1 or even 1000:1. A read resulting in a complex database join can be very expensive, spending a significant amount of time on disk operations.

Disadvantage(s): denormalization

- Data is duplicated.
- Constraints can help redundant copies of information stay in sync, which increases complexity of the database design.
- A denormalized database under heavy write load might perform worse than its normalized counterpart.

Source(s) and further reading: denormalization

- [Denormalization](#)

SQL tuning

SQL tuning is a broad topic and many [books](#) have been written as reference.

It's important to **benchmark** and **profile** to simulate and uncover bottlenecks.

- **Benchmark** - Simulate high-load situations with tools such as [ab](#).
- **Profile** - Enable tools such as the [slow query log](#) to help track performance issues.

Benchmarking and profiling might point you to the following optimizations.

Tighten up the schema

- MySQL dumps to disk in contiguous blocks for fast access.
- Use CHAR instead of VARCHAR for fixed-length fields.
 - CHAR effectively allows for fast, random access, whereas with VARCHAR, you must find the end of a string before moving onto the next one.
- Use TEXT for large blocks of text such as blog posts. TEXT also allows for boolean searches. Using a TEXT field results in storing a pointer on disk that is used to locate the text block.
- Use INT for larger numbers up to 2^{32} or 4 billion.
- Use DECIMAL for currency to avoid floating point representation errors.
- Avoid storing large BLOBS, store the location of where to get the object instead.
- VARCHAR(255) is the largest number of characters that can be counted in an 8 bit number, often maximizing the use of a byte in some RDBMS.

- Set the NOT NULL constraint where applicable to [improve search performance](#).

Use good indices

- Columns that you are querying (SELECT, GROUP BY, ORDER BY, JOIN) could be faster with indices.
- Indices are usually represented as self-balancing [B-tree](#) that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
- Placing an index can keep the data in memory, requiring more space.
- Writes could also be slower since the index also needs to be updated.
- When loading large amounts of data, it might be faster to disable indices, load the data, then rebuild the indices.

Avoid expensive joins

- [Denormalize](#) where performance demands it.

Partition tables

- Break up a table by putting hot spots in a separate table to help keep it in memory.

Tune the query cache

- In some cases, the [query cache](#) could lead to [performance issues](#).

Source(s) and further reading: SQL tuning

- [Tips for optimizing MySQL queries](#)
- [Is there a good reason i see VARCHAR\(255\) used so often?](#)
- [How do null values affect performance?](#)
- [Slow query log](#)

NoSQL

NoSQL is a collection of data items represented in a **key-value store**, **document store**, **wide column store**, or a **graph database**. Data is denormalized, and joins are generally done in the application code. Most NoSQL stores lack true ACID transactions and favor [eventual consistency](#).

BASE is often used to describe the properties of NoSQL databases. In comparison with the [CAP Theorem](#), BASE chooses availability over consistency.

- **Basically available** - the system guarantees availability.
- **Soft state** - the state of the system may change over time, even without input.
- **Eventual consistency** - the system will become consistent over a period of time, given that the system doesn't receive input during that period.

In addition to choosing between [SQL or NoSQL](#), it is helpful to understand which type of NoSQL database best fits your use case(s). We'll review **key-value stores, document stores, wide column stores, and graph databases** in the next section.

Key-value store

Abstraction: hash table

A key-value store generally allows for O(1) reads and writes and is often backed by memory or SSD. Data stores can maintain keys in [lexicographic order](#), allowing efficient retrieval of key ranges. Key-value stores can allow for storing of metadata with a value.

Key-value stores provide high performance and are often used for simple data models or for rapidly-changing data, such as an in-memory cache layer. Since they offer only a limited set of operations, complexity is shifted to the application layer if additional operations are needed.

A key-value store is the basis for more complex systems such as a document store, and in some cases, a graph database.

Source(s) and further reading: key-value store

- [Key-value database](#)
- [Disadvantages of key-value stores](#)
- [Redis architecture](#)
- [Memcached architecture](#)

Document store

Abstraction: key-value store with documents stored as values

A document store is centered around documents (XML, JSON, binary, etc), where a document stores all information for a given object. Document stores provide APIs or a

query language to query based on the internal structure of the document itself. Note, many key-value stores include features for working with a value's metadata, blurring the lines between these two storage types.

Based on the underlying implementation, documents are organized by collections, tags, metadata, or directories. Although documents can be organized or grouped together, documents may have fields that are completely different from each other.

Some document stores like [MongoDB](#) and [CouchDB](#) also provide a SQL-like language to perform complex queries. [DynamoDB](#) supports both key-values and documents.

Document stores provide high flexibility and are often used for working with occasionally changing data.

Source(s) and further reading: document store

- [Document-oriented database](#)
- [MongoDB architecture](#)
- [CouchDB architecture](#)
- [Elasticsearch architecture](#)

Wide column store

super column family				
companies				
row key 1	address		website	
	city	San Francisco	subdomain	www
	state	California	domain	grio.com
	street	Kearny St.	protocol	http

column family column

[Source: SQL & NoSQL, a brief history](#)

Abstraction: nested map `ColumnFamily<RowKey, Columns<ColKey, Value, Timestamp>>`

A wide column store's basic unit of data is a column (name/value pair). A column can be grouped in column families (analogous to a SQL table). Super column families further

group column families. You can access each column independently with a row key, and columns with the same row key form a row. Each value contains a timestamp for versioning and for conflict resolution.

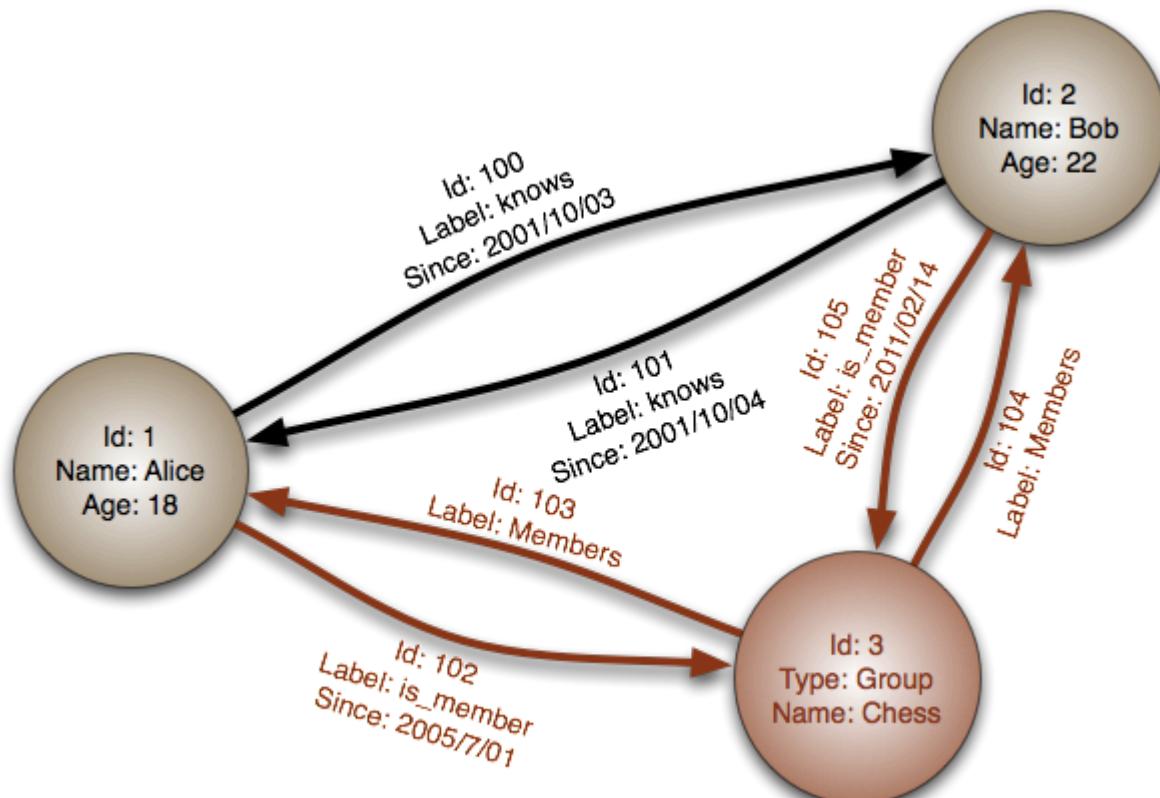
Google introduced [Bigtable](#) as the first wide column store, which influenced the open-source [HBase](#) often-used in the Hadoop ecosystem, and [Cassandra](#) from Facebook. Stores such as BigTable, HBase, and Cassandra maintain keys in lexicographic order, allowing efficient retrieval of selective key ranges.

Wide column stores offer high availability and high scalability. They are often used for very large data sets.

Source(s) and further reading: wide column store

- [SQL & NoSQL, a brief history](#)
- [Bigtable architecture](#)
- [HBase architecture](#)
- [Cassandra architecture](#)

Graph database



[Source: Graph database](#)

Abstraction: graph

In a graph database, each node is a record and each arc is a relationship between two nodes. Graph databases are optimized to represent complex relationships with many foreign keys or many-to-many relationships.

Graphs databases offer high performance for data models with complex relationships, such as a social network. They are relatively new and are not yet widely-used; it might be more difficult to find development tools and resources. Many graphs can only be accessed with [REST APIs](#).

Source(s) and further reading: graph

- [Graph database](#)
- [Neo4j](#)
- [FlockDB](#)

Source(s) and further reading: NoSQL

- [Explanation of base terminology](#)
- [NoSQL databases a survey and decision guidance](#)
- [Scalability](#)
- [Introduction to NoSQL](#)
- [NoSQL patterns](#)

SQL or NoSQL



Relational data model

Highly-structured table organization with rigidly-defined data formats and record structure.



Document data model

Collection of complex documents with arbitrary, nested data formats and varying "record" format.

Source: Transitioning from RDBMS to NoSQL

Reasons for SQL:

- Structured data
- Strict schema
- Relational data
- Need for complex joins
- Transactions
- Clear patterns for scaling
- More established: developers, community, code, tools, etc
- Lookups by index are very fast

Reasons for NoSQL:

- Semi-structured data
- Dynamic or flexible schema
- Non-relational data
- No need for complex joins
- Store many TB (or PB) of data
- Very data intensive workload
- Very high throughput for IOPS

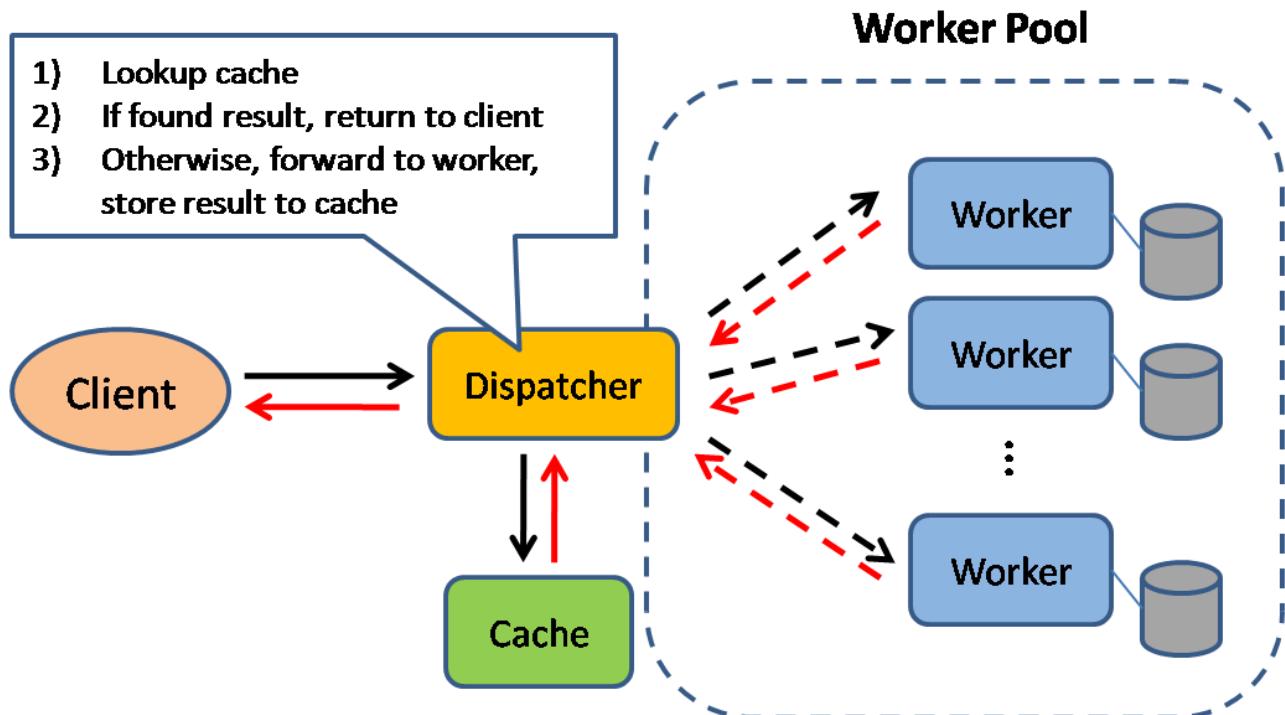
Sample data well-suited for NoSQL:

- Rapid ingest of clickstream and log data
- Leaderboard or scoring data
- Temporary data, such as a shopping cart
- Frequently accessed ('hot') tables
- Metadata/lookup tables

Source(s) and further reading: SQL or NoSQL

- [Scaling up to your first 10 million users](#)
- [SQL vs NoSQL differences](#)

Cache



[Source: Scalable system design patterns](#)

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.

Caching at the database query level

Whenever you query the database, hash the query as a key and store the result to the cache. This approach suffers from expiration issues:

- Hard to delete a cached result with complex queries
- If one piece of data changes such as a table cell, you need to delete all cached queries that might include the changed cell

Caching at the object level

See your data as an object, similar to what you do with your application code. Have your application assemble the dataset from the database into a class instance or a data structure(s):

- Remove the object from cache if its underlying data has changed
- Allows for asynchronous processing: workers assemble objects by consuming the latest cached object

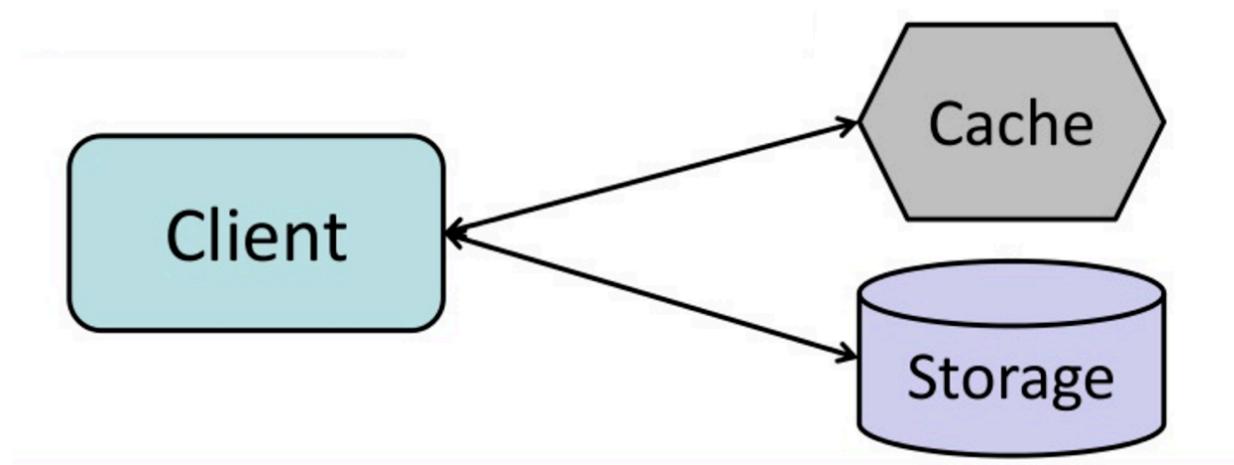
Suggestions of what to cache:

- User sessions
- Fully rendered web pages
- Activity streams
- User graph data

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



[Source: From cache to in-memory data grid](#)

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
```

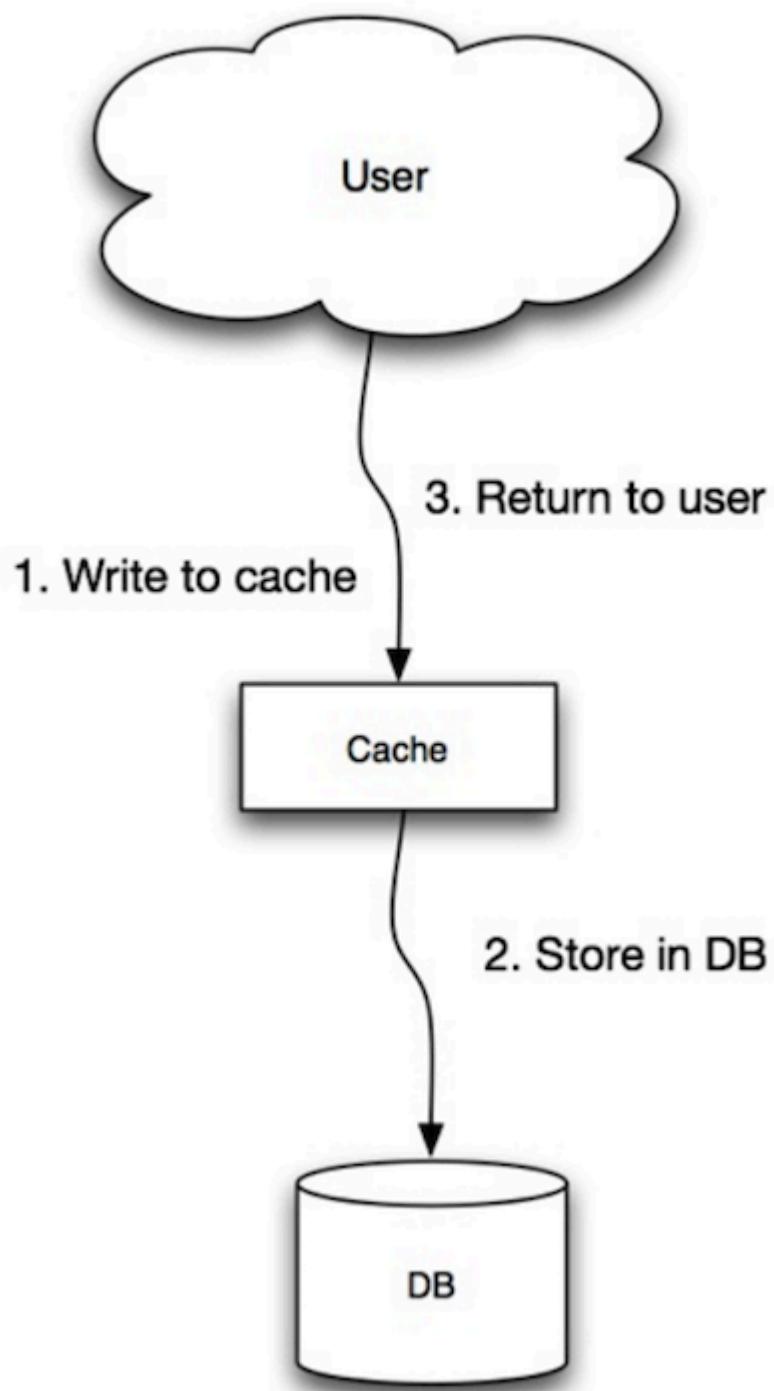
[Memcached](#) is generally used in this manner.

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



[Source: Scalability, availability, stability, patterns](#)

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:

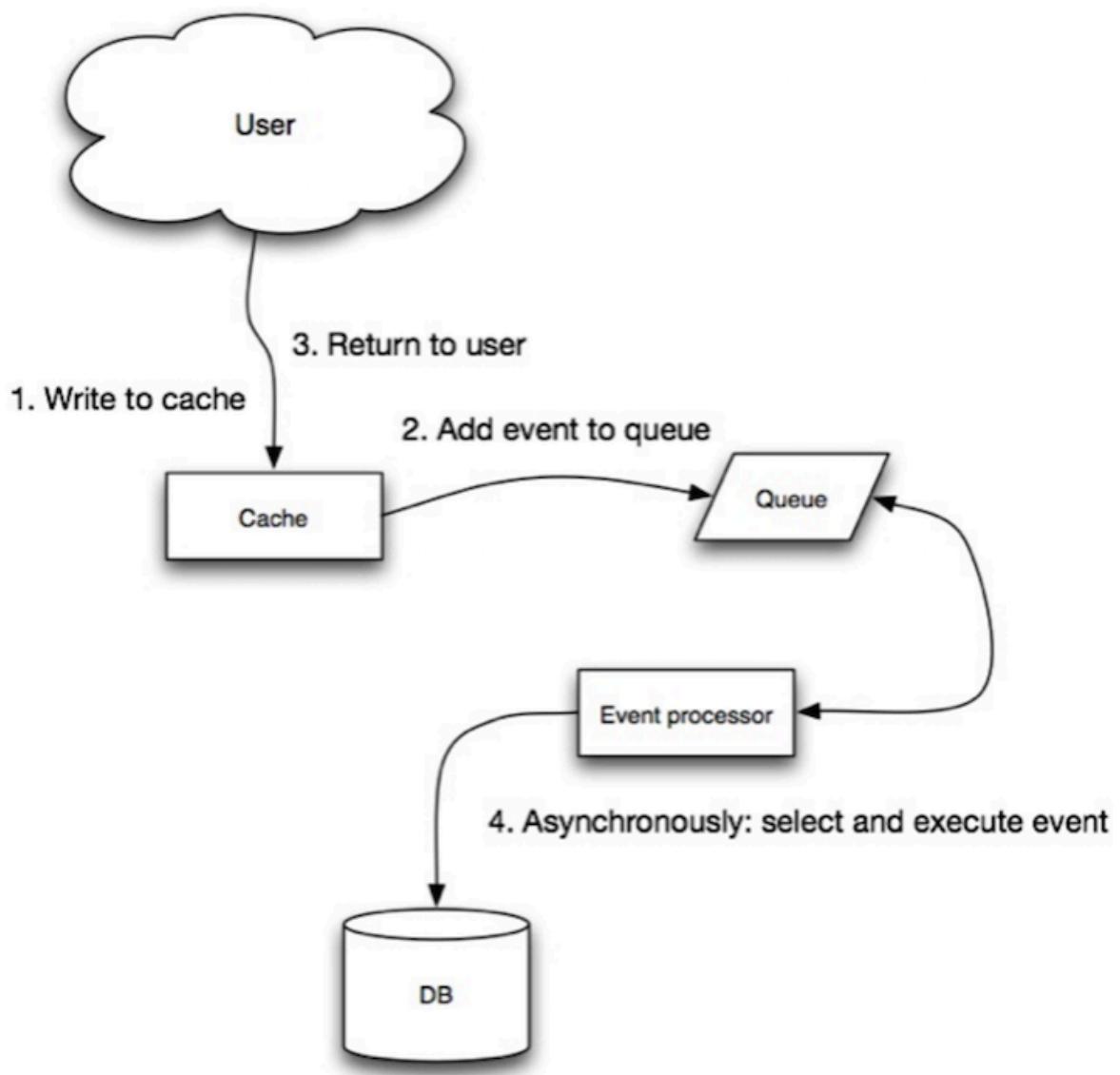
```
def set_user(user_id, values):
    user = db.query("UPDATE Users WHERE id = {0}", user_id, values)
    cache.set(user_id, user)
```

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)



[Source: Scalability, availability, stability, patterns](#)

In write-behind, the application does the following:

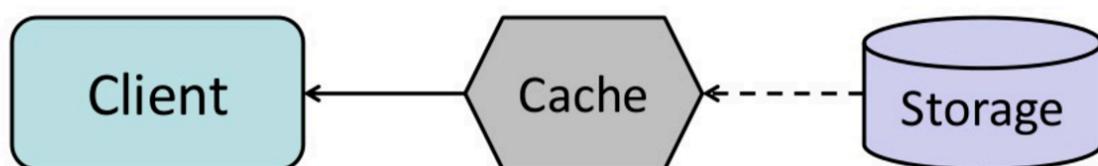
- Add/update entry in cache
- Asynchronously write entry to the data store, improving write performance

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



[Source: From cache to in-memory data grid](#)

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

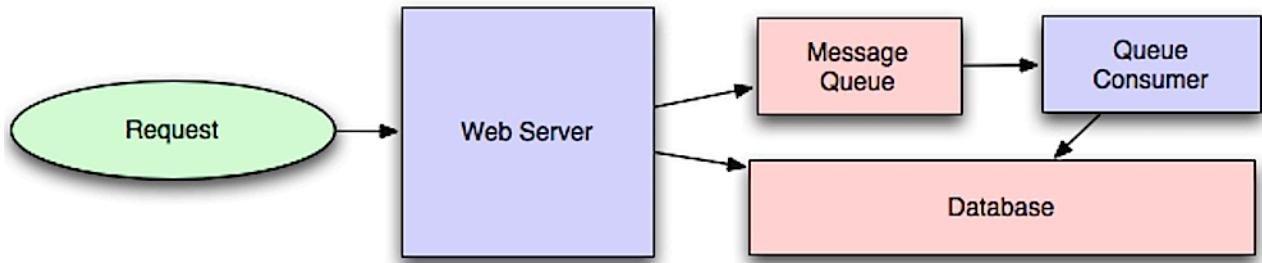
- Need to maintain consistency between caches and the source of truth such as the database through [cache invalidation](#).
- Cache invalidation is a difficult problem, there is additional complexity associated with when to update the cache.
- Need to make application changes such as adding Redis or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

Asynchronism



Source: [Intro to architecting systems for scale](#)

Asynchronous workflows help reduce request times for expensive operations that would otherwise be performed in-line. They can also help by doing time-consuming work in advance, such as periodic aggregation of data.

Message queues

Message queues receive, hold, and deliver messages. If an operation is too slow to perform inline, you can use a message queue with the following workflow:

- An application publishes a job to the queue, then notifies the user of job status
- A worker picks up the job from the queue, processes it, then signals the job is complete

The user is not blocked and the job is processed in the background. During this time, the client might optionally do a small amount of processing to make it seem like the task has completed. For example, if posting a tweet, the tweet could be instantly posted to your timeline, but it could take some time before your tweet is actually delivered to all of your followers.

[**Redis**](#) is useful as a simple message broker but messages can be lost.

[**RabbitMQ**](#) is popular but requires you to adapt to the 'AMQP' protocol and manage your own nodes.

[Amazon SQS](#) is hosted but can have high latency and has the possibility of messages being delivered twice.

Task queues

Tasks queues receive tasks and their related data, runs them, then delivers their results. They can support scheduling and can be used to run computationally-intensive jobs in the background.

[Celery](#) has support for scheduling and primarily has python support.

Back pressure

If queues start to grow significantly, the queue size can become larger than memory, resulting in cache misses, disk reads, and even slower performance. [Back pressure](#) can help by limiting the queue size, thereby maintaining a high throughput rate and good response times for jobs already in the queue. Once the queue fills up, clients get a server busy or HTTP 503 status code to try again later. Clients can retry the request at a later time, perhaps with [exponential backoff](#).

Disadvantage(s): asynchronism

- Use cases such as inexpensive calculations and realtime workflows might be better suited for synchronous operations, as introducing queues can add delays and complexity.

Source(s) and further reading

- [It's all a numbers game](#)
- [Applying back pressure when overloaded](#)
- [Little's law](#)
- [What is the difference between a message queue and a task queue?](#)

Communication

OSI (Open Systems Interconnection) 7 Layer Model

Layer	Application/Example	Central Device/Protocols
Application (7) Serves as the window for users and application processes to access the network services.	End User layer Program that opens what was sent or creates what is to be sent Resource sharing • Remote file access • Remote printer access • Directory services • Network management	User Applications SMTP
Presentation (6) Formats the data to be presented to the Application layer. It can be viewed as the "Translator" for the network.	Syntax layer encrypt & decrypt (if needed) Character code translation • Data conversion • Data compression • Data encryption • Character Set Translation	JPEG/ASCII EBDIC/TIFF/GIF PICT
Session (5) Allows session establishment between processes running on different stations.	Synch & send to ports (logical ports) Session establishment, maintenance and termination • Session support - perform security, name recognition, logging, etc.	Logical Ports RPC/SQL/NFS NetBIOS names
Transport (4) Ensures that messages are delivered error-free, in sequence, and with no losses or duplications.	TCP Host to Host, Flow Control Message segmentation • Message acknowledgement • Message traffic control • Session multiplexing	F I L T E R E P A C K E T R O U B L E TCP/SPX/UDP
Network (3) Controls the operations of the subnet, deciding which physical path the data takes.	Packets ("letter", contains IP address) Routing • Subnet traffic control • Frame fragmentation • Logical-physical address mapping • Subnet usage accounting	Routers IP/IPX/ICMP
Data Link (2) Provides error-free transfer of data frames from one node to another over the Physical layer.	Frames ("envelopes", contains MAC address [NIC card — Switch — NIC card] (end to end)) Establishes & terminates the logical link between nodes • Frame traffic control • Frame sequencing • Frame acknowledgment • Frame delimiting • Frame error checking • Media access control	Switch Bridge WAP PPP/SLIP
Physical (1) Concerned with the transmission and reception of the unstructured raw bit stream over the physical medium.	Physical structure Cables, hubs, etc. Data Encoding • Physical medium attachment • Transmission technique - Baseband or Broadband • Physical medium transmission Bits & Volts	Hub Land Based Layers

[Source: OSI 7 layer model](#)

Hypertext transfer protocol (HTTP)

HTTP is a method for encoding and transporting data between a client and a server. It is a request/response protocol: clients issue requests and servers issue responses with relevant content and completion status info about the request. HTTP is self-contained, allowing requests and responses to flow through many intermediate routers and servers that perform load balancing, caching, encryption, and compression.

A basic HTTP request consists of a verb (method) and a resource (endpoint). Below are common HTTP verbs:

Verb	Description	Idempotent*	Safe	Cacheable
GET	Reads a resource	Yes	Yes	Yes
POST	Creates a resource or trigger a process that handles data	No	No	Yes if response contains freshness info
PUT	Creates or replace a resource	Yes	No	No
PATCH	Partially updates a resource	No	No	Yes if response contains freshness info
DELETE	Deletes a resource	Yes	No	No

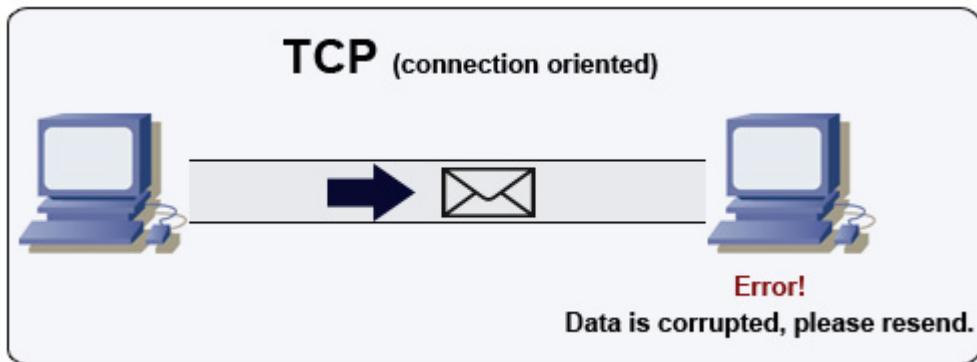
*Can be called many times without different outcomes.

HTTP is an application layer protocol relying on lower-level protocols such as **TCP** and **UDP**.

Source(s) and further reading: HTTP

- [What is HTTP?](#)
- [Difference between HTTP and TCP](#)
- [Difference between PUT and PATCH](#)

Transmission control protocol (TCP)



[Source: How to make a multiplayer game](#)

TCP is a connection-oriented protocol over an [IP network](#). Connection is established and terminated using a [handshake](#). All packets sent are guaranteed to reach the destination in the original order and without corruption through:

- Sequence numbers and [checksum fields](#) for each packet
- [Acknowledgement](#) packets and automatic retransmission

If the sender does not receive a correct response, it will resend the packets. If there are multiple timeouts, the connection is dropped. TCP also implements [flow control](#) and [congestion control](#). These guarantees cause delays and generally result in less efficient transmission than UDP.

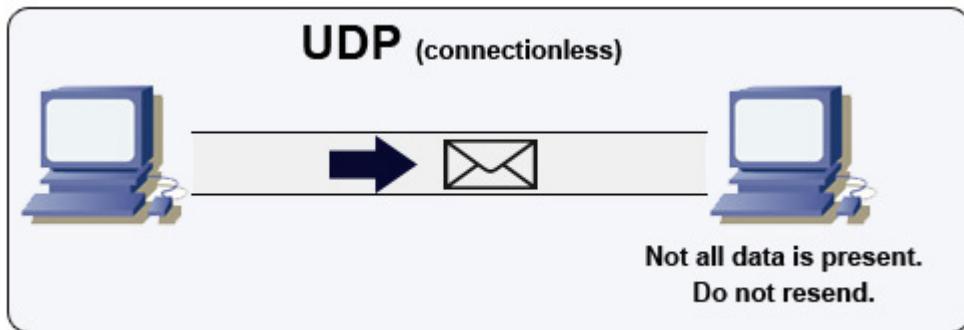
To ensure high throughput, web servers can keep a large number of TCP connections open, resulting in high memory usage. It can be expensive to have a large number of open connections between web server threads and say, a [memcached](#) server. [Connection pooling](#) can help in addition to switching to UDP where applicable.

TCP is useful for applications that require high reliability but are less time critical. Some examples include web servers, database info, SMTP, FTP, and SSH.

Use TCP over UDP when:

- You need all of the data to arrive intact
- You want to automatically make a best estimate use of the network throughput

User datagram protocol (UDP)



[Source: How to make a multiplayer game](#)

UDP is connectionless. Datagrams (analogous to packets) are guaranteed only at the datagram level. Datagrams might reach their destination out of order or not at all. UDP does not support congestion control. Without the guarantees that TCP supports, UDP is generally more efficient.

UDP can broadcast, sending datagrams to all devices on the subnet. This is useful with [DHCP](#) because the client has not yet received an IP address, thus preventing a way for TCP to stream without the IP address.

UDP is less reliable but works well in real time use cases such as VoIP, video chat, streaming, and realtime multiplayer games.

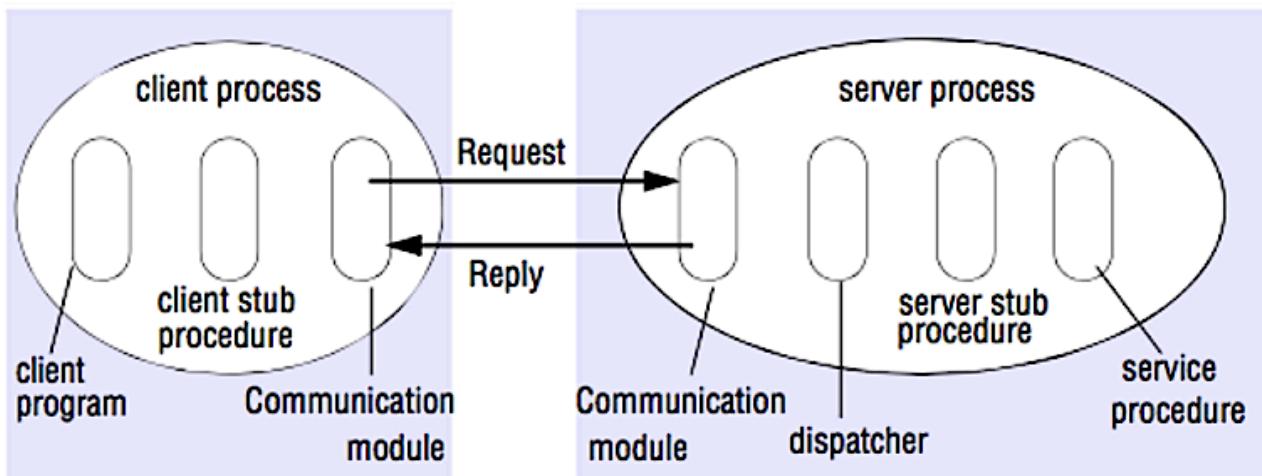
Use UDP over TCP when:

- You need the lowest latency
- Late data is worse than loss of data
- You want to implement your own error correction

Source(s) and further reading: TCP and UDP

- [Networking for game programming](#)
- [Key differences between TCP and UDP protocols](#)
- [Difference between TCP and UDP](#)
- [Transmission control protocol](#)
- [User datagram protocol](#)
- [Scaling memcache at Facebook](#)

Remote procedure call (RPC)



Source: Crack the system design interview

In an RPC, a client causes a procedure to execute on a different address space, usually a remote server. The procedure is coded as if it were a local procedure call, abstracting away the details of how to communicate with the server from the client program. Remote calls are usually slower and less reliable than local calls so it is helpful to distinguish RPC calls from local calls. Popular RPC frameworks include [Protobuf](#), [Thrift](#), and [Avro](#).

RPC is a request-response protocol:

- **Client program** - Calls the client stub procedure. The parameters are pushed onto the stack like a local procedure call.
- **Client stub procedure** - Marshals (packs) procedure id and arguments into a request message.
- **Client communication module** - OS sends the message from the client to the server.
- **Server communication module** - OS passes the incoming packets to the server stub procedure.
- **Server stub procedure** - Unmarshalls the results, calls the server procedure matching the procedure id and passes the given arguments.
- The server response repeats the steps above in reverse order.

Sample RPC calls:

```
GET /someoperation?data=anId

POST /anotheroperation
{
  "data": "anId";
  "anotherdata": "another value"
}
```

RPC is focused on exposing behaviors. RPCs are often used for performance reasons with internal communications, as you can hand-craft native calls to better fit your use cases.

Choose a native library (aka SDK) when:

- You know your target platform.
- You want to control how your “logic” is accessed.
- You want to control how error control happens off your library.
- Performance and end user experience is your primary concern.

HTTP APIs following **REST** tend to be used more often for public APIs.

Disadvantage(s): RPC

- RPC clients become tightly coupled to the service implementation.
- A new API must be defined for every new operation or use case.
- It can be difficult to debug RPC.
- You might not be able to leverage existing technologies out of the box. For example, it might require additional effort to ensure [RPC calls are properly cached](#) on caching servers such as [Squid](#).

Representational state transfer (REST)

REST is an architectural style enforcing a client/server model where the client acts on a set of resources managed by the server. The server provides a representation of resources and actions that can either manipulate or get a new representation of resources. All communication must be stateless and cacheable.

There are four qualities of a RESTful interface:

- **Identify resources (URI in HTTP)** - use the same URI regardless of any operation.

- **Change with representations (Verbs in HTTP)** - use verbs, headers, and body.
- **Self-descriptive error message (status response in HTTP)** - Use status codes, don't reinvent the wheel.
- **HATEOAS (HTML interface for HTTP)** - your web service should be fully accessible in a browser.

Sample REST calls:

```
GET /someresources/anId

PUT /someresources/anId
{"anotherdata": "another value"}
```

REST is focused on exposing data. It minimizes the coupling between client/server and is often used for public HTTP APIs. REST uses a more generic and uniform method of exposing resources through URIs, [representation through headers](#), and actions through verbs such as GET, POST, PUT, DELETE, and PATCH. Being stateless, REST is great for horizontal scaling and partitioning.

Disadvantage(s): REST

- With REST being focused on exposing data, it might not be a good fit if resources are not naturally organized or accessed in a simple hierarchy. For example, returning all updated records from the past hour matching a particular set of events is not easily expressed as a path. With REST, it is likely to be implemented with a combination of URI path, query parameters, and possibly the request body.
- REST typically relies on a few verbs (GET, POST, PUT, DELETE, and PATCH) which sometimes doesn't fit your use case. For example, moving expired documents to the archive folder might not cleanly fit within these verbs.
- Fetching complicated resources with nested hierarchies requires multiple round trips between the client and server to render single views, e.g. fetching content of a blog entry and the comments on that entry. For mobile applications operating in variable network conditions, these multiple roundtrips are highly undesirable.
- Over time, more fields might be added to an API response and older clients will receive all new data fields, even those that they do not need, as a result, it bloats the payload size and leads to larger latencies.

RPC and REST calls comparison

Operation	RPC	REST
Signup	POST /signup	POST /persons
Resign	POST /resign { "personid": "1234" }	DELETE /persons/1234
Read a person	GET /readPerson? personid=1234	GET /persons/1234
Read a person's items list	GET /readUsersItemsList? personid=1234	GET /persons/1234/items
Add an item to a person's items	POST /addItemToUsersItemsList { "personid": "1234"; "itemid": "456" }	POST /persons/1234/items { "itemid": "456" }
Update an item	POST /modifyItem { "itemid": "456"; "key": "value" }	PUT /items/456 { "key": "value" }
Delete an item	POST /removeItem { "itemid": "456" }	DELETE /items/456

[Source: Do you really know why you prefer REST over RPC](#)

Source(s) and further reading: REST and RPC

- [Do you really know why you prefer REST over RPC](#)

- [When are RPC-ish approaches more appropriate than REST?](#)
- [REST vs JSON-RPC](#)
- [Debunking the myths of RPC and REST](#)
- [What are the drawbacks of using REST](#)
- [Crack the system design interview](#)
- [Thrift](#)
- [Why REST for internal use and not RPC](#)

Security

This section could use some updates. Consider [contributing!](#)

Security is a broad topic. Unless you have considerable experience, a security background, or are applying for a position that requires knowledge of security, you probably won't need to know more than the basics:

- Encrypt in transit and at rest.
- Sanitize all user inputs or any input parameters exposed to user to prevent [XSS](#) and [SQL injection](#).
- Use parameterized queries to prevent SQL injection.
- Use the principle of [least privilege](#).

Source(s) and further reading

- [API security checklist](#)
- [Security guide for developers](#)
- [OWASP top ten](#)

Appendix

You'll sometimes be asked to do 'back-of-the-envelope' estimates. For example, you might need to determine how long it will take to generate 100 image thumbnails from disk or how much memory a data structure will take. The **Powers of two table** and **Latency numbers every programmer should know** are handy references.

Powers of two table

Power	Exact Value	Approx Value	Bytes
7	128		
8	256		
10	1024	1 thousand	1 KB
16	65,536		64 KB
20	1,048,576	1 million	1 MB
30	1,073,741,824	1 billion	1 GB
32	4,294,967,296		4 GB
40	1,099,511,627,776	1 trillion	1 TB

Source(s) and further reading

- [Powers of two](#)

Latency numbers every programmer should know

Latency Comparison Numbers

L1 cache reference	0.5	ns				
Branch mispredict	5	ns				
L2 cache reference	7	ns				14x
L1 cache						
Mutex lock/unlock	25	ns				
Main memory reference	100	ns				20x
L2 cache, 200x L1 cache						
Compress 1K bytes with Zippy	10,000	ns	10	us		
Send 1 KB bytes over 1 Gbps network	10,000	ns	10	us		
Read 4 KB randomly from SSD*	150,000	ns	150	us		
~1GB/sec SSD						
Read 1 MB sequentially from memory	250,000	ns	250	us		
Round trip within same datacenter	500,000	ns	500	us		
Read 1 MB sequentially from SSD*	1,000,000	ns	1,000	us	1	ms
~1GB/sec SSD, 4X memory						
HDD seek	10,000,000	ns	10,000	us	10	ms
datacenter roundtrip						20x
Read 1 MB sequentially from 1 Gbps memory, 10X SSD	10,000,000	ns	10,000	us	10	ms
Read 1 MB sequentially from HDD memory, 30X SSD	30,000,000	ns	30,000	us	30	ms
Send packet CA->Netherlands->CA	150,000,000	ns	150,000	us	150	ms

Notes

1 ns = 10^{-9} seconds

1 us = 10^{-6} seconds = 1,000 ns

1 ms = 10^{-3} seconds = 1,000 us = 1,000,000 ns

Handy metrics based on numbers above:

- Read sequentially from HDD at 30 MB/s
- Read sequentially from 1 Gbps Ethernet at 100 MB/s
- Read sequentially from SSD at 1 GB/s
- Read sequentially from main memory at 4 GB/s
- 6-7 world-wide round trips per second
- 2,000 round trips per second within a data center

Latency numbers visualized



Source(s) and further reading

- [Latency numbers every programmer should know - 1](#)
- [Latency numbers every programmer should know - 2](#)
- [Designs, lessons, and advice from building large distributed systems](#)
- [Software Engineering Advice from Building Large-Scale Distributed Systems](#)

Additional system design interview questions

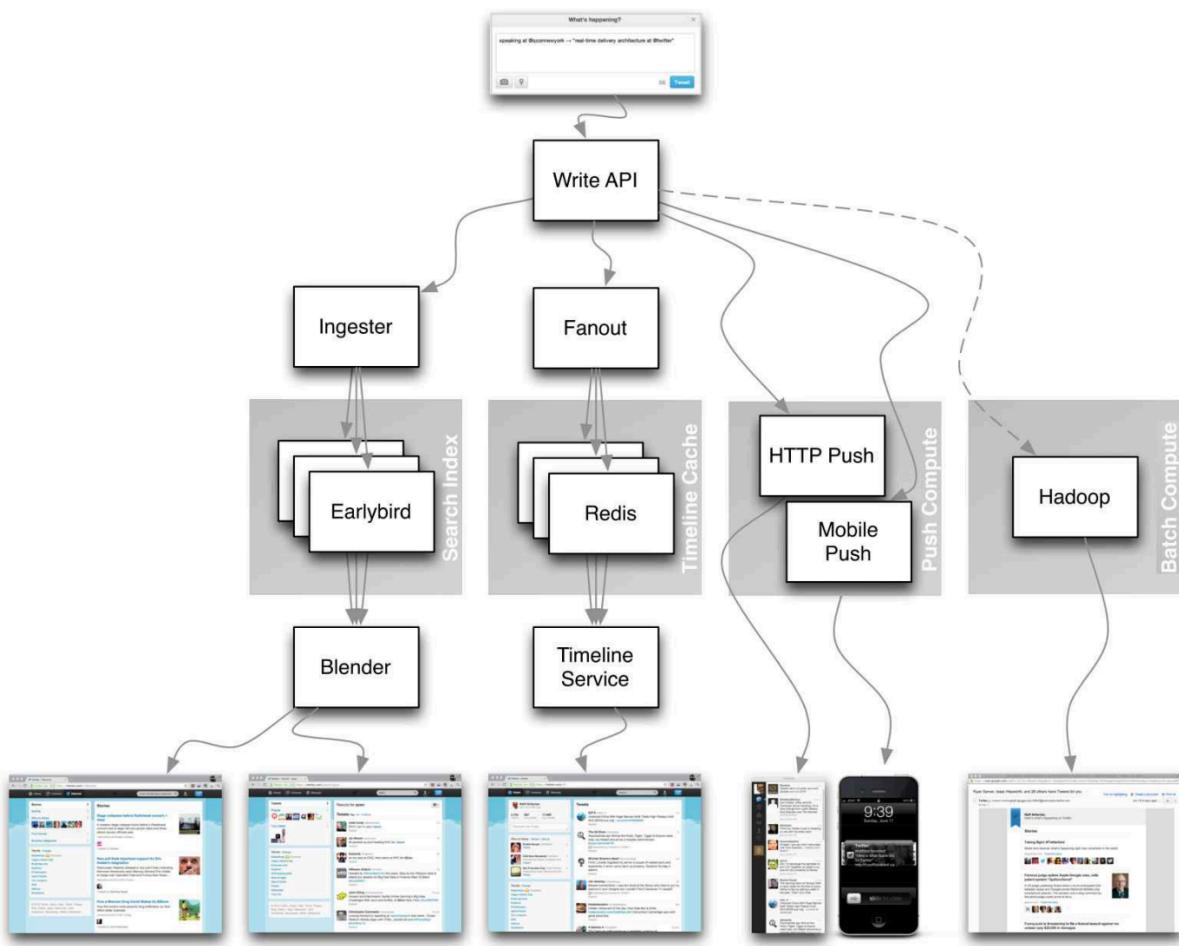
Common system design interview questions, with links to resources on how to solve each.

Question	Reference(s)
Design a file sync service like Dropbox	youtube.com
Design a search engine like Google	queue.acm.org stackexchange.com ardendertat.com stanford.edu
Design a scalable web crawler like Google	quora.com
Design Google docs	code.google.com neil.fraser.name
Design a key-value store like Redis	slideshare.net
Design a cache system like Memcached	slideshare.net
Design a recommendation system like Amazon's	hulu.com ijcai13.org
Design a tinyurl system like Bitly	n00tc0d3r.blogspot.com
Design a chat app like WhatsApp	highscalability.com
Design a picture sharing system like Instagram	highscalability.com highscalability.com
Design the Facebook news feed function	quora.com quora.com slideshare.net
Design the Facebook timeline function	facebook.com highscalability.com
Design the Facebook chat function	erlang-factory.com facebook.com
Design a graph search function like Facebook's	facebook.com facebook.com

Question	Reference(s)
	facebook.com
Design a content delivery network like CloudFlare	figshare.com
Design a trending topic system like Twitter's	michael-noll.com snikolov.wordpress.com
Design a random ID generation system	blog.twitter.com github.com
Return the top k requests during a time interval	cs.ucsb.edu wpi.edu
Design a system that serves data from multiple data centers	highscalability.com
Design an online multiplayer card game	indieflashblog.com buildnewgames.com
Design a garbage collection system	stuffwithstuff.com washington.edu
Design an API rate limiter	https://stripe.com/blog/
Design a Stock Exchange (like NASDAQ or Binance)	Jane Street Golang Implementation Go Implementation
Add a system design question	Contribute

Real world architectures

Articles on how real world systems are designed.



[Source: Twitter timelines at scale](#)

Don't focus on nitty gritty details for the following articles, instead:

- Identify shared principles, common technologies, and patterns within these articles
- Study what problems are solved by each component, where it works, where it doesn't
- Review the lessons learned

Type	System	Reference(s)
Data processing	MapReduce - Distributed data processing from Google	research.google.com
Data processing	Spark - Distributed data processing from Databricks	slideshare.net
Data processing	Storm - Distributed data processing from Twitter	slideshare.net
Data store	Bigtable - Distributed column-oriented database from Google	harvard.edu
Data store	HBase - Open source implementation of Bigtable	slideshare.net
Data store	Cassandra - Distributed column-oriented database from Facebook	slideshare.net
Data store	DynamoDB - Document-oriented database from Amazon	harvard.edu
Data store	MongoDB - Document-oriented database	slideshare.net
Data store	Spanner - Globally-distributed database from Google	research.google.com
Data store	Memcached - Distributed memory caching system	slideshare.net

Type	System	Reference(s)
Data store	Redis - Distributed memory caching system with persistence and value types	slideshare.net
File system	Google File System (GFS) - Distributed file system	research.google.com
File system	Hadoop File System (HDFS) - Open source implementation of GFS	apache.org
Misc	Chubby - Lock service for loosely-coupled distributed systems from Google	research.google.com
Misc	Dapper - Distributed systems tracing infrastructure	research.google.com
Misc	Kafka - Pub/sub message queue from LinkedIn	slideshare.net
Misc	Zookeeper - Centralized infrastructure and services enabling synchronization	slideshare.net
	Add an architecture	Contribute

Company architectures

Company	Reference(s)
Amazon	Amazon architecture
Cinchcast	Producing 1,500 hours of audio every day
DataSift	Realtime datamining At 120,000 tweets per second
Dropbox	How we've scaled Dropbox
ESPN	Operating At 100,000 duh nuh nuhs per second
Google	Google architecture
Instagram	14 million users, terabytes of photos What powers Instagram
Justin.tv	Justin.Tv's live video broadcasting architecture
Facebook	Scaling memcached at Facebook TAO: Facebook's distributed data store for the social graph Facebook's photo storage How Facebook Live Streams To 800,000 Simultaneous Viewers
Flickr	Flickr architecture
Mailbox	From 0 to one million users in 6 weeks
Netflix	A 360 Degree View Of The Entire Netflix Stack Netflix: What Happens When You Press Play?

Company	Reference(s)
Pinterest	From 0 To 10s of billions of page views a month 18 million visitors, 10x growth, 12 employees
Playfish	50 million monthly users and growing
PlentyOfFish	PlentyOfFish architecture
Salesforce	How they handle 1.3 billion transactions a day
Stack Overflow	Stack Overflow architecture
TripAdvisor	40M visitors, 200M dynamic page views, 30TB data
Tumblr	15 billion page views a month
Twitter	Making Twitter 10000 percent faster Storing 250 million tweets a day using MySQL 150M active users, 300K QPS, a 22 MB/S firehose Timelines at scale Big and small data at Twitter Operations at Twitter: scaling beyond 100 million users How Twitter Handles 3,000 Images Per Second
Uber	How Uber scales their real-time market platform Lessons Learned From Scaling Uber To 2000 Engineers, 1000 Services, And 8000 Git Repositories

Company	Reference(s)
WhatsApp	The WhatsApp architecture Facebook bought for \$19 billion
YouTube	YouTube scalability YouTube architecture

Company engineering blogs

Architectures for companies you are interviewing with.

Questions you encounter might be from the same domain.

- [Airbnb Engineering](#)
- [Atlassian Developers](#)
- [AWS Blog](#)
- [Bitly Engineering Blog](#)
- [Box Blogs](#)
- [Cloudera Developer Blog](#)
- [Dropbox Tech Blog](#)
- [Engineering at Quora](#)
- [Ebay Tech Blog](#)
- [Evernote Tech Blog](#)
- [Etsy Code as Craft](#)
- [Facebook Engineering](#)
- [Flickr Code](#)
- [Foursquare Engineering Blog](#)
- [GitHub Engineering Blog](#)
- [Google Research Blog](#)
- [Groupon Engineering Blog](#)
- [Heroku Engineering Blog](#)
- [Hubspot Engineering Blog](#)
- [High Scalability](#)
- [Instagram Engineering](#)
- [Intel Software Blog](#)
- [Jane Street Tech Blog](#)
- [LinkedIn Engineering](#)

- [Microsoft Engineering](#)
- [Microsoft Python Engineering](#)
- [Netflix Tech Blog](#)
- [Paypal Developer Blog](#)
- [Pinterest Engineering Blog](#)
- [Reddit Blog](#)
- [Salesforce Engineering Blog](#)
- [Slack Engineering Blog](#)
- [Spotify Labs](#)
- [Stripe Engineering Blog](#)
- [Twilio Engineering Blog](#)
- [Twitter Engineering](#)
- [Uber Engineering Blog](#)
- [Yahoo Engineering Blog](#)
- [Yelp Engineering Blog](#)
- [Zynga Engineering Blog](#)

Source(s) and further reading

Looking to add a blog? To avoid duplicating work, consider adding your company blog to the following repo:

- [kilimchoi/engineering-blogs](#)

Under development

Interested in adding a section or helping complete one in-progress? [Contribute!](#)

- Distributed computing with MapReduce
- Consistent hashing
- Scatter gather
- [Contribute](#)

Credits

Credits and sources are provided throughout this repo.

Special thanks to:

- [Hired in tech](#)
- [Cracking the coding interview](#)
- [High scalability](#)
- [checkcheckzz/system-design-interview](#)
- [shashank88/system_design](#)
- [mmcgrana/services-engineering](#)
- [System design cheat sheet](#)
- [A distributed systems reading list](#)
- [Cracking the system design interview](#)

Contact info

Feel free to contact me to discuss any issues, questions, or comments.

My contact info can be found on my [GitHub page](#).

License

I am providing code and resources in this repository to you under an open source license. Because this is my personal repository, the license you receive to my code and resources is from me and not my employer (Facebook).

Copyright 2017 Donne Martin

Creative Commons Attribution 4.0 International License (CC BY 4.0)

<http://creativecommons.org/licenses/by/4.0/>

CONTRIBUTING

Contributing

Contributions are welcome!

Please carefully read this page to make the code review process go as smoothly as possible and to maximize the likelihood of your contribution being merged.

Bug Reports

For bug reports or requests [submit an issue](#).

Pull Requests

The preferred way to contribute is to fork the [main repository](#) on GitHub.

1. Fork the [main repository](#). Click on the 'Fork' button near the top of the page. This creates a copy of the code under your account on the GitHub server.
2. Clone this copy to your local disk:

```
$ git clone git@github.com:YourLogin/system-design-primer.git  
$ cd system-design-primer
```

3. Create a branch to hold your changes and start making changes. Don't work in the master branch!

```
$ git checkout -b my-feature
```

4. Work on this copy on your computer using Git to do the version control. When you're done editing, run the following to record your changes in Git:

```
$ git add modified_files  
$ git commit
```

5. Push your changes to GitHub with:

```
$ git push -u origin my-feature
```

6. Finally, go to the web page of your fork of the system-design-primer repo and click 'Pull Request' to send your changes for review.

GitHub Pull Requests Docs

If you are not familiar with pull requests, review the [pull request docs](#).

Translations

We'd like for the guide to be available in many languages. Here is the process for maintaining translations:

- This original version and content of the guide is maintained in English.
- Translations follow the content of the original. Contributors must speak at least some English, so that translations do not diverge.
- Each translation has a maintainer to update the translation as the original evolves and to review others' changes. This doesn't require a lot of time, but a review by the maintainer is important to maintain quality.

See [Translations](#).

Changes to translations

- Changes to content should be made to the English version first, and then translated to each other language.
- Changes that improve translations should be made directly on the file for that language. Pull requests should only modify one language at a time.
- Submit a pull request with changes to the file in that language. Each language has a maintainer, who reviews changes in that language. Then the primary maintainer [@donnemartin](#) merges it in.
- Prefix pull requests and issues with language codes if they are for that translation only, e.g. "es: Improve grammar", so maintainers can find them easily.
- Tag the translation maintainer for a code review, see the list of [translation maintainers](#).
 - You will need to get a review from a native speaker (preferably the language maintainer) before your pull request is merged.

Adding translations to new languages

Translations to new languages are always welcome! Keep in mind a translation must be maintained.

- Do you have time to be a maintainer for a new language? Please see the list of [translations](#) and tell us so we know we can count on you in the future.
- Check the [translations](#), issues, and pull requests to see if a translation is in progress or stalled. If it's in progress, offer to help. If it's stalled, consider

becoming the maintainer if you can commit to it.

- If a translation has not yet been started, file an issue for your language so people know you are working on it and we'll coordinate. Confirm you are native level in the language and are willing to maintain the translation, so it's not orphaned.
- To get started, fork the repo, then submit a pull request to the main repo with the single file README-xx.md added, where xx is the language code. Use standard [IETF language tags](#), i.e. the same as is used by Wikipedia, *not* the code for a single country. These are usually just the two-letter lowercase code, for example, fr for French and uk for Ukrainian (not ua, which is for the country). For languages that have variations, use the shortest tag, such as zh-Hant.
- Feel free to invite friends to help your original translation by having them fork your repo, then merging their pull requests to your forked repo. Translations are difficult and usually have errors that others need to find.
- Add links to your translation at the top of every README-XX.md file. For consistency, the link should be added in alphabetical order by ISO code, and the anchor text should be in the native language.
- When you've fully translated the English README.md, comment on the pull request in the main repo that it's ready to be merged.
 - You'll need to have a complete and reviewed translation of the English README.md before your translation will be merged into the master branch.
 - Once accepted, your pull request will be squashed into a single commit into the master branch.

Translation template credits

Thanks to [The Art of Command Line](#) for the translation template.

System Design: Pastebin

Design Pastebin.com (or Bit.ly)

Note: This document links directly to relevant areas found in the [system design topics](#) to avoid duplication. Refer to the linked content for general talking points, tradeoffs, and alternatives.

Design Bit.ly - is a similar question, except pastebin requires storing the paste contents instead of the original unshortened url.

Step 1: Outline use cases and constraints

Gather requirements and scope the problem. Ask questions to clarify use cases and constraints. Discuss assumptions.

Without an interviewer to address clarifying questions, we'll define some use cases and constraints.

Use cases

We'll scope the problem to handle only the following use cases

- **User** enters a block of text and gets a randomly generated link
 - Expiration
 - Default setting does not expire
 - Can optionally set a timed expiration
- **User** enters a paste's url and views the contents
- **User** is anonymous
- **Service** tracks analytics of pages
 - Monthly visit stats
- **Service** deletes expired pastes
- **Service** has high availability

Out of scope

- **User** registers for an account
 - **User** verifies email
- **User** logs into a registered account
 - **User** edits the document
- **User** can set visibility
- **User** can set the shortlink

Constraints and assumptions

State assumptions

- Traffic is not evenly distributed
- Following a short link should be fast
- Pastes are text only
- Page view analytics do not need to be realtime
- 10 million users
- 10 million paste writes per month
- 100 million paste reads per month
- 10:1 read to write ratio

Calculate usage

Clarify with your interviewer if you should run back-of-the-envelope usage calculations.

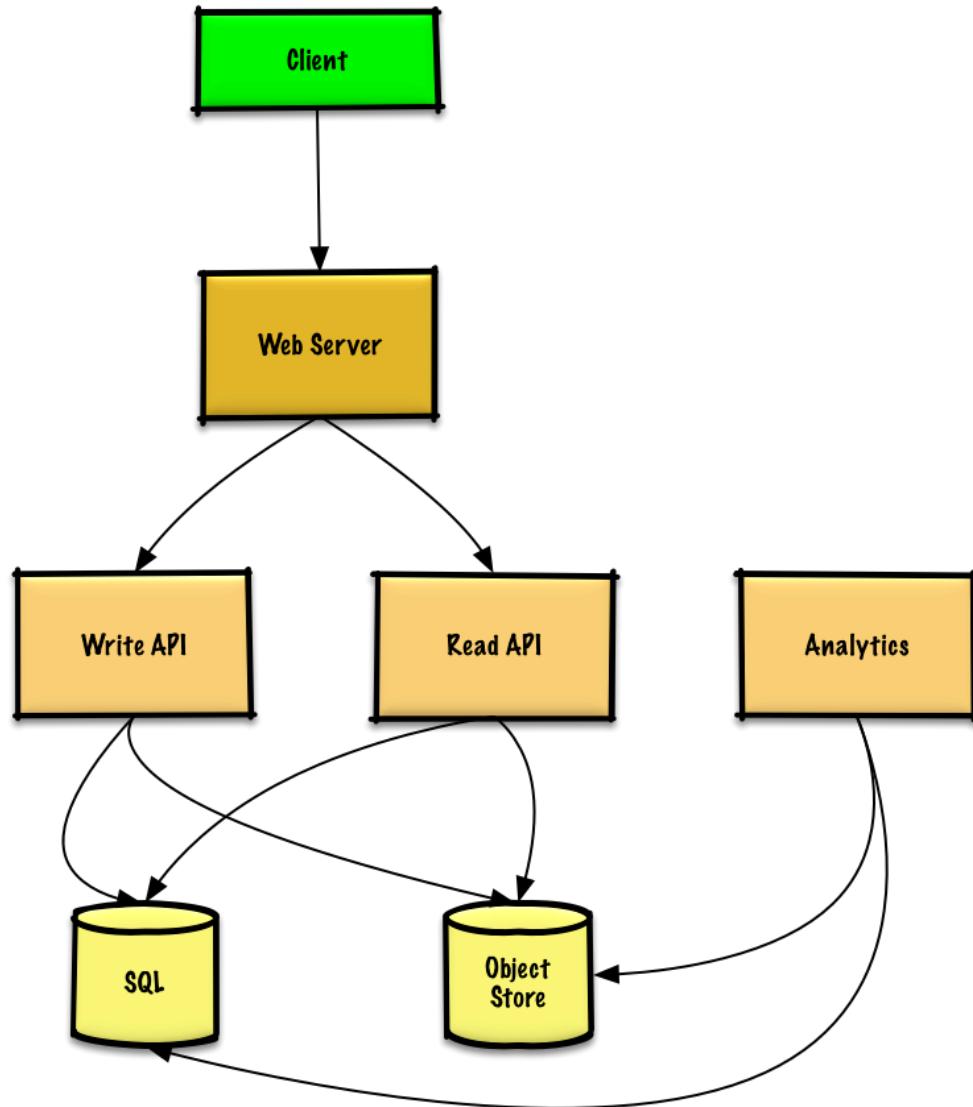
- Size per paste
 - 1 KB content per paste
 - shortlink - 7 bytes
 - expiration_length_in_minutes - 4 bytes
 - created_at - 5 bytes
 - paste_path - 255 bytes
 - total = ~1.27 KB
- 12.7 GB of new paste content per month
 - 1.27 KB per paste * 10 million pastes per month
 - ~450 GB of new paste content in 3 years
 - 360 million shortlinks in 3 years
 - Assume most are new pastes instead of updates to existing ones
- 4 paste writes per second on average
- 40 read requests per second on average

Handy conversion guide:

- 2.5 million seconds per month
- 1 request per second = 2.5 million requests per month
- 40 requests per second = 100 million requests per month
- 400 requests per second = 1 billion requests per month

Step 2: Create a high level design

Outline a high level design with all important components.



Imgur

Step 3: Design core components

Dive into details for each core component.

Use case: User enters a block of text and gets a randomly generated link

We could use a [relational database](#) as a large hash table, mapping the generated url to a file server and path containing the paste file.

Instead of managing a file server, we could use a managed **Object Store** such as Amazon S3 or a [NoSQL document store](#).

An alternative to a relational database acting as a large hash table, we could use a [NoSQL key-value store](#). We should discuss the [tradeoffs between choosing SQL or NoSQL](#). The following discussion uses the relational database approach.

- The **Client** sends a create paste request to the **Web Server**, running as a [reverse proxy](#)
- The **Web Server** forwards the request to the **Write API** server
- The **Write API** server does the following:
 - Generates a unique url
 - Checks if the url is unique by looking at the **SQL Database** for a duplicate
 - If the url is not unique, it generates another url
 - If we supported a custom url, we could use the user-supplied (also check for a duplicate)
 - Saves to the **SQL Database** pastes table
 - Saves the paste data to the **Object Store**
 - Returns the url

Clarify with your interviewer how much code you are expected to write.

The pastes table could have the following structure:

```
shortlink char(7) NOT NULL
expiration_length_in_minutes int NOT NULL
created_at datetime NOT NULL
paste_path varchar(255) NOT NULL
PRIMARY KEY(shortlink)
```

Setting the primary key to be based on the shortlink column creates an [index](#) that the database uses to enforce uniqueness. We'll create an additional index on created_at to speed up lookups (log-time instead of scanning the entire table) and to keep the data in

memory. Reading 1 MB sequentially from memory takes about 250 microseconds, while reading from SSD takes 4x and from disk takes 80x longer.¹

To generate the unique url, we could:

- Take the [MD5](#) hash of the user's ip_address + timestamp
 - MD5 is a widely used hashing function that produces a 128-bit hash value
 - MD5 is uniformly distributed
 - Alternatively, we could also take the MD5 hash of randomly-generated data
- [Base 62](#) encode the MD5 hash
 - Base 62 encodes to [a-zA-Z0-9] which works well for urls, eliminating the need for escaping special characters
 - There is only one hash result for the original input and Base 62 is deterministic (no randomness involved)
 - Base 64 is another popular encoding but provides issues for urls because of the additional + and / characters
 - The following [Base 62 pseudocode](#) runs in O(k) time where k is the number of digits = 7:

```
def base_encode(num, base=62):  
    digits = []  
    while num > 0:  
        remainder = modulo(num, base)  
        digits.push(remainder)  
        num = divide(num, base)  
    digits = digits.reverse
```

- Take the first 7 characters of the output, which results in 62^7 possible values and should be sufficient to handle our constraint of 360 million shortlinks in 3 years:

```
url = base_encode(md5(ip_address+timestamp))[:URL_LENGTH]
```

We'll use a public [REST API](#):

```
$ curl -X POST --data '{ "expiration_length_in_minutes": "60", \  
    "paste_contents": "Hello World!" }' https://pastebin.com/api/v1/paste
```

Response:

```
{  
    "shortlink": "foobar"  
}
```

For internal communications, we could use [Remote Procedure Calls](#).

Use case: User enters a paste's url and views the contents

- The **Client** sends a get paste request to the **Web Server**
- The **Web Server** forwards the request to the **Read API** server
- The **Read API** server does the following:
 - Checks the **SQL Database** for the generated url
 - If the url is in the **SQL Database**, fetch the paste contents from the **Object Store**
 - Else, return an error message for the user

REST API:

```
$ curl https://pastebin.com/api/v1/paste?shortlink=foobar
```

Response:

```
{  
    "paste_contents": "Hello World"  
    "created_at": "YYYY-MM-DD HH:MM:SS"  
    "expiration_length_in_minutes": "60"  
}
```

Use case: Service tracks analytics of pages

Since realtime analytics are not a requirement, we could simply **MapReduce** the **Web Server** logs to generate hit counts.

Clarify with your interviewer how much code you are expected to write.

```

class HitCounts(MRJob):

    def extract_url(self, line):
        """Extract the generated url from the log line."""
        ...

    def extract_year_month(self, line):
        """Return the year and month portions of the timestamp."""
        ...

    def mapper(self, _, line):
        """Parse each log line, extract and transform relevant lines.

        Emit key value pairs of the form:

        (2016-01, url0), 1
        (2016-01, url0), 1
        (2016-01, url1), 1
        """
        url = self.extract_url(line)
        period = self.extract_year_month(line)
        yield (period, url), 1

    def reducer(self, key, values):
        """Sum values for each key.

        (2016-01, url0), 2
        (2016-01, url1), 1
        """
        yield key, sum(values)

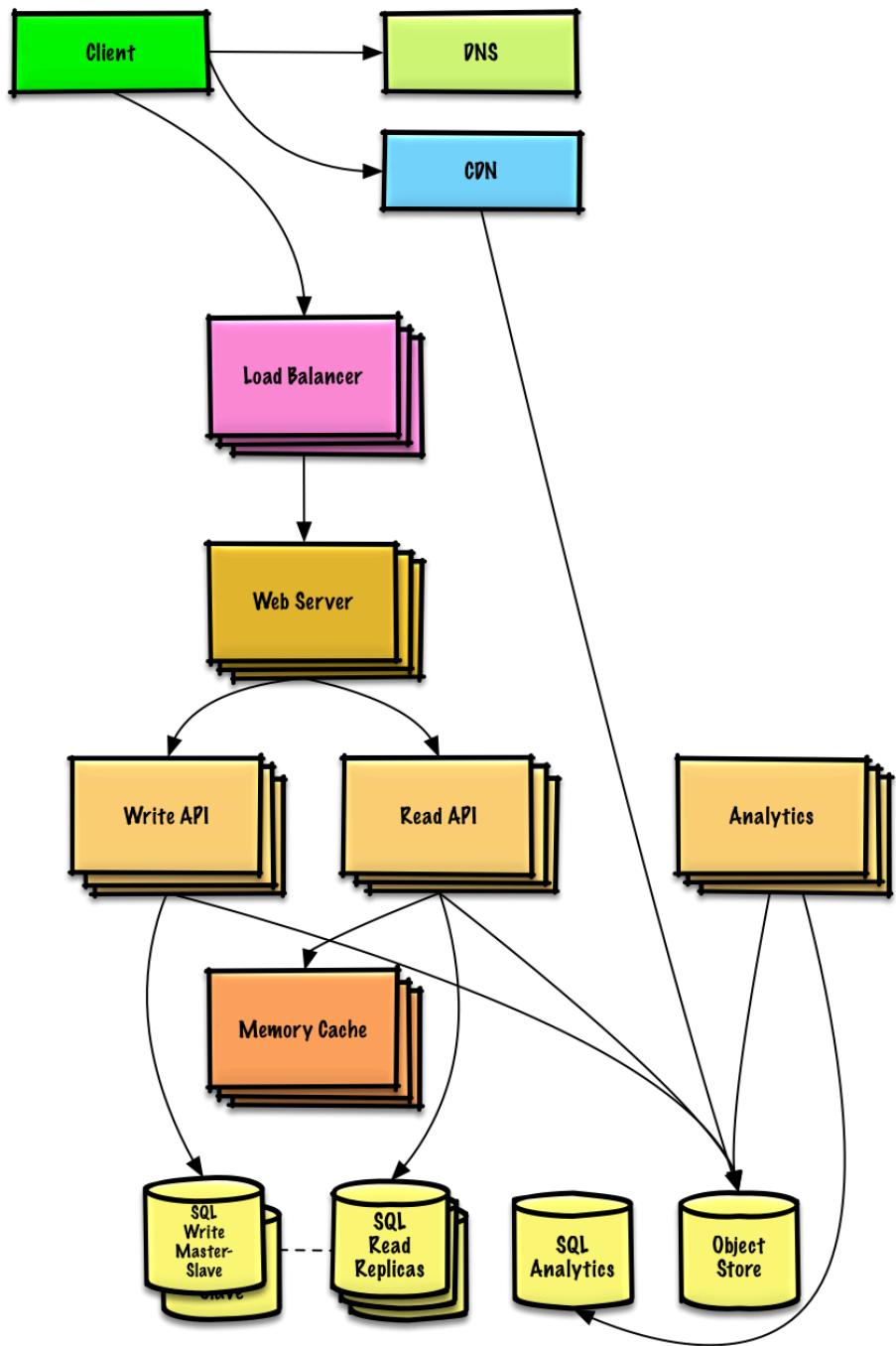
```

Use case: Service deletes expired pastes

To delete expired pastes, we could just scan the **SQL Database** for all entries whose expiration timestamp are older than the current timestamp. All expired entries would then be deleted (or marked as expired) from the table.

Step 4: Scale the design

Identify and address bottlenecks, given the constraints.



Imgur

Important: Do not simply jump right into the final design from the initial design!

State you would do this iteratively: 1) **Benchmark/Load Test**, 2) **Profile** for bottlenecks 3) address bottlenecks while evaluating alternatives and trade-offs, and 4) repeat. See [Design a system that scales to millions of users on AWS](#) as a sample on how to iteratively scale the initial design.

It's important to discuss what bottlenecks you might encounter with the initial design and how you might address each of them. For example, what issues are addressed by adding a **Load Balancer** with multiple **Web Servers?** **CDN?** **Master-Slave Replicas?** What are the alternatives and **Trade-Offs** for each?

We'll introduce some components to complete the design and to address scalability issues. Internal load balancers are not shown to reduce clutter.

To avoid repeating discussions, refer to the following [system design topics](#) for main talking points, tradeoffs, and alternatives:

- [DNS](#)
- [CDN](#)
- [Load balancer](#)
- [Horizontal scaling](#)
- [Web server \(reverse proxy\)](#)
- [API server \(application layer\)](#)
- [Cache](#)
- [Relational database management system \(RDBMS\)](#)
- [SQL write master-slave failover](#)
- [Master-slave replication](#)
- [Consistency patterns](#)
- [Availability patterns](#)

The **Analytics Database** could use a data warehousing solution such as Amazon Redshift or Google BigQuery.

An **Object Store** such as Amazon S3 can comfortably handle the constraint of 12.7 GB of new content per month.

To address the 40 average read requests per second (higher at peak), traffic for popular content should be handled by the **Memory Cache** instead of the database. The **Memory Cache** is also useful for handling the unevenly distributed traffic and traffic spikes. The **SQL Read Replicas** should be able to handle the cache misses, as long as the replicas are not bogged down with replicating writes.

4 average paste writes per second (with higher at peak) should be do-able for a single **SQL Write Master-Slave**. Otherwise, we'll need to employ additional SQL scaling patterns:

- [Federation](#)
- [Sharding](#)
- [Denormalization](#)
- [SQL Tuning](#)

We should also consider moving some data to a **NoSQL Database**.

Additional talking points

Additional topics to dive into, depending on the problem scope and time remaining.

NoSQL

- [Key-value store](#)
- [Document store](#)
- [Wide column store](#)
- [Graph database](#)
- [SQL vs NoSQL](#)

Caching

- Where to cache
 - [Client caching](#)
 - [CDN caching](#)
 - [Web server caching](#)
 - [Database caching](#)
 - [Application caching](#)
- What to cache
 - [Caching at the database query level](#)
 - [Caching at the object level](#)
- When to update the cache
 - [Cache-aside](#)
 - [Write-through](#)
 - [Write-behind \(write-back\)](#)
 - [Refresh ahead](#)

Asynchronism and microservices

- [Message queues](#)
- [Task queues](#)
- [Back pressure](#)
- [Microservices](#)

Communications

- Discuss tradeoffs:
 - External communication with clients - [HTTP APIs following REST](#)
 - Internal communications - [RPC](#)
- [Service discovery](#)

Security

Refer to the [security section](#).

Latency numbers

See [Latency numbers every programmer should know](#).

Ongoing

- Continue benchmarking and monitoring your system to address bottlenecks as they come up
- Scaling is an iterative process

System Design: Twitter

Design the Twitter timeline and search

Note: This document links directly to relevant areas found in the [system design topics](#) to avoid duplication. Refer to the linked content for general talking points, tradeoffs, and alternatives.

Design the Facebook feed and **Design Facebook search** are similar questions.

Step 1: Outline use cases and constraints

Gather requirements and scope the problem. Ask questions to clarify use cases and constraints. Discuss assumptions.

Without an interviewer to address clarifying questions, we'll define some use cases and constraints.

Use cases

We'll scope the problem to handle only the following use cases

- **User** posts a tweet
 - **Service** pushes tweets to followers, sending push notifications and emails
- **User** views the user timeline (activity from the user)
- **User** views the home timeline (activity from people the user is following)
- **User** searches keywords
- **Service** has high availability

Out of scope

- **Service** pushes tweets to the Twitter Firehose and other streams
- **Service** strips out tweets based on users' visibility settings
 - Hide @reply if the user is not also following the person being replied to
 - Respect 'hide retweets' setting
- Analytics

Constraints and assumptions

State assumptions

General

- Traffic is not evenly distributed
- Posting a tweet should be fast
 - Fanning out a tweet to all of your followers should be fast, unless you have millions of followers
- 100 million active users
- 500 million tweets per day or 15 billion tweets per month

- Each tweet averages a fanout of 10 deliveries
- 5 billion total tweets delivered on fanout per day
- 150 billion tweets delivered on fanout per month
- 250 billion read requests per month
- 10 billion searches per month

Timeline

- Viewing the timeline should be fast
- Twitter is more read heavy than write heavy
 - Optimize for fast reads of tweets
- Ingesting tweets is write heavy

Search

- Searching should be fast
- Search is read-heavy

Calculate usage

Clarify with your interviewer if you should run back-of-the-envelope usage calculations.

- Size per tweet:
 - tweet_id - 8 bytes
 - user_id - 32 bytes
 - text - 140 bytes
 - media - 10 KB average
 - Total: ~10 KB
- 150 TB of new tweet content per month
 - 10 KB per tweet * 500 million tweets per day * 30 days per month
 - 5.4 PB of new tweet content in 3 years
- 100 thousand read requests per second
 - 250 billion read requests per month * (400 requests per second / 1 billion requests per month)
- 6,000 tweets per second
 - 15 billion tweets per month * (400 requests per second / 1 billion requests per month)
- 60 thousand tweets delivered on fanout per second

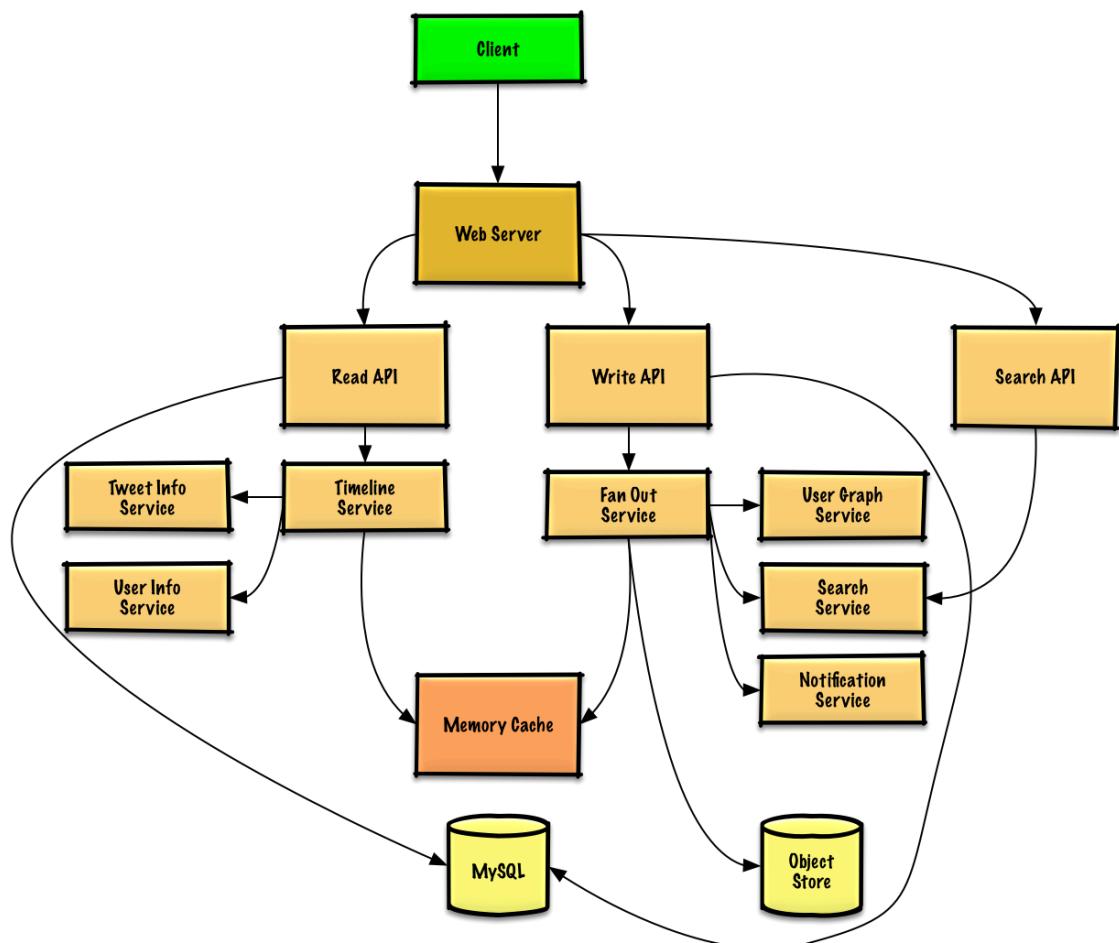
- 150 billion tweets delivered on fanout per month * (400 requests per second / 1 billion requests per month)
- 4,000 search requests per second
 - 10 billion searches per month * (400 requests per second / 1 billion requests per month)

Handy conversion guide:

- 2.5 million seconds per month
- 1 request per second = 2.5 million requests per month
- 40 requests per second = 100 million requests per month
- 400 requests per second = 1 billion requests per month

Step 2: Create a high level design

Outline a high level design with all important components.



Imgur

Step 3: Design core components

Dive into details for each core component.

Use case: User posts a tweet

We could store the user's own tweets to populate the user timeline (activity from the user) in a [relational database](#). We should discuss the [use cases and tradeoffs between choosing SQL or NoSQL](#).

Delivering tweets and building the home timeline (activity from people the user is following) is trickier. Fanning out tweets to all followers (60 thousand tweets delivered on fanout per second) will overload a traditional [relational database](#). We'll probably want to choose a data store with fast writes such as a **NoSQL database or Memory Cache**. Reading 1 MB sequentially from memory takes about 250 microseconds, while reading from SSD takes 4x and from disk takes 80x longer.¹

We could store media such as photos or videos on an **Object Store**.

- The **Client** posts a tweet to the **Web Server**, running as a [reverse proxy](#)
- The **Web Server** forwards the request to the **Write API** server
- The **Write API** stores the tweet in the user's timeline on a **SQL database**
- The **Write API** contacts the **Fan Out Service**, which does the following:
 - Queries the **User Graph Service** to find the user's followers stored in the **Memory Cache**
 - Stores the tweet in the *home timeline of the user's followers* in a **Memory Cache**
 - O(n) operation: 1,000 followers = 1,000 lookups and inserts
 - Stores the tweet in the **Search Index Service** to enable fast searching
 - Stores media in the **Object Store**
 - Uses the **Notification Service** to send out push notifications to followers:
 - Uses a **Queue** (not pictured) to asynchronously send out notifications

Clarify with your interviewer how much code you are expected to write.

If our **Memory Cache** is Redis, we could use a native Redis list with the following structure:

```

        tweet n+2           tweet n+1           tweet n
| 8 bytes  8 bytes  1 byte | 8 bytes  8 bytes  1 byte | 8 bytes  8 bytes
1 byte |
| tweet_id  user_id  meta   | tweet_id  user_id  meta   | tweet_id  user_id
meta   |

```

The new tweet would be placed in the **Memory Cache**, which populates the user's home timeline (activity from people the user is following).

We'll use a public [REST API](#):

```
$ curl -X POST --data '{ "user_id": "123", "auth_token": "ABC123", \
    "status": "hello world!", "media_ids": "ABC987" }' \
    https://twitter.com/api/v1/tweet
```

Response:

```
{
  "created_at": "Wed Sep 05 00:37:15 +0000 2012",
  "status": "hello world!",
  "tweet_id": "987",
  "user_id": "123",
  ...
}
```

For internal communications, we could use [Remote Procedure Calls](#).

Use case: User views the home timeline

- The **Client** posts a home timeline request to the **Web Server**
- The **Web Server** forwards the request to the **Read API** server
- The **Read API** server contacts the **Timeline Service**, which does the following:
 - Gets the timeline data stored in the **Memory Cache**, containing tweet ids and user ids - $O(1)$
 - Queries the **Tweet Info Service** with a [multiget](#) to obtain additional info about the tweet ids - $O(n)$
 - Queries the **User Info Service** with a multiget to obtain additional info about the user ids - $O(n)$

REST API:

```
$ curl https://twitter.com/api/v1/home_timeline?user_id=123
```

Response:

```
{
  "user_id": "456",
  "tweet_id": "123",
  "status": "foo"
},
{
  "user_id": "789",
  "tweet_id": "456",
  "status": "bar"
},
{
  "user_id": "789",
  "tweet_id": "579",
  "status": "baz"
},
```

Use case: User views the user timeline

- The **Client** posts a user timeline request to the **Web Server**
- The **Web Server** forwards the request to the **Read API** server
- The **Read API** retrieves the user timeline from the **SQL Database**

The REST API would be similar to the home timeline, except all tweets would come from the user as opposed to the people the user is following.

Use case: User searches keywords

- The **Client** sends a search request to the **Web Server**
- The **Web Server** forwards the request to the **Search API** server
- The **Search API** contacts the **Search Service**, which does the following:
 - Parses/tokenizes the input query, determining what needs to be searched
 - Removes markup
 - Breaks up the text into terms
 - Fixes typos

- Normalizes capitalization
- Converts the query to use boolean operations
- Queries the **Search Cluster** (ie [Lucene](#)) for the results:
 - [Scatter gathers](#) each server in the cluster to determine if there are any results for the query
 - Merges, ranks, sorts, and returns the results

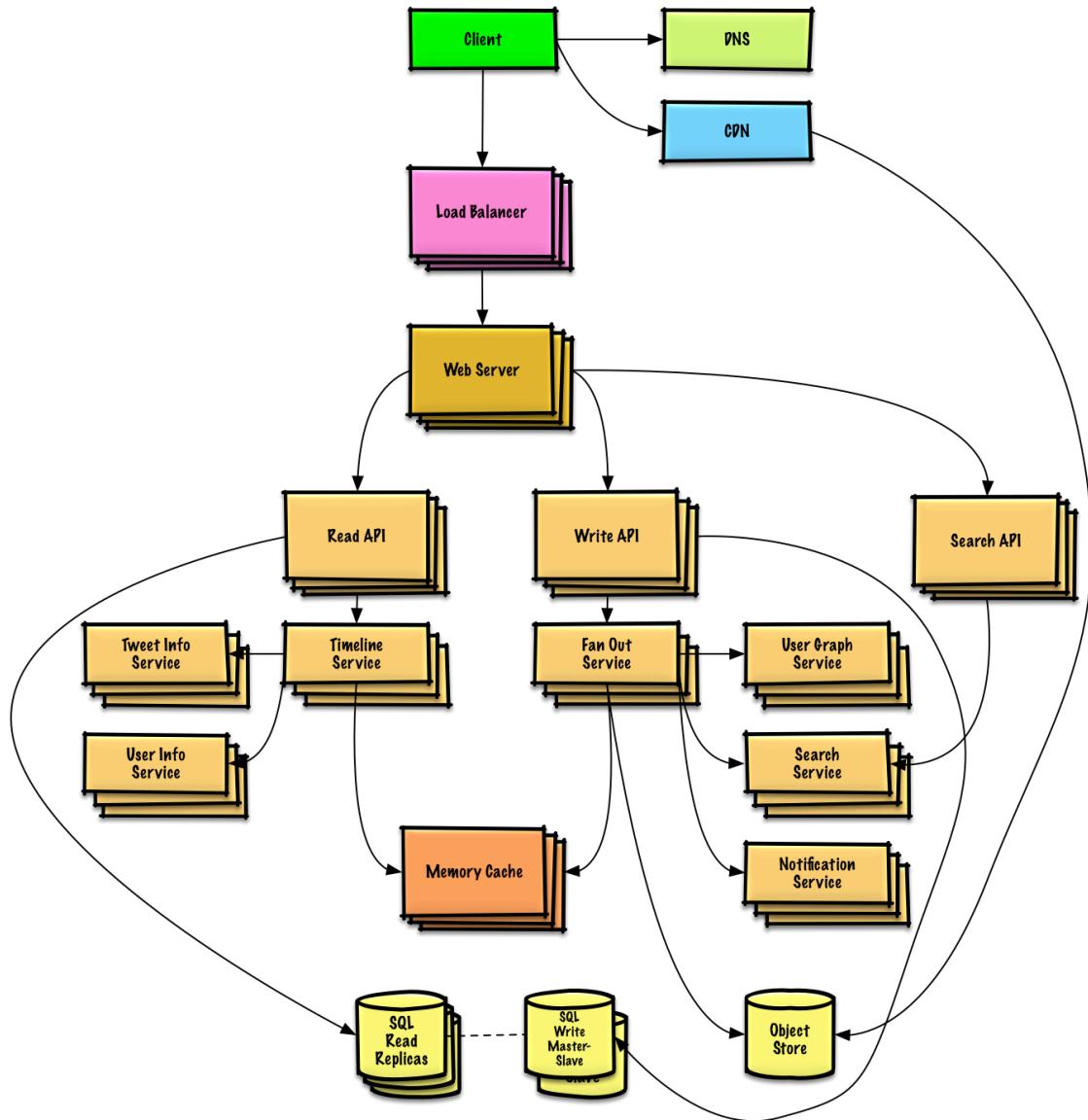
REST API:

```
$ curl https://twitter.com/api/v1/search?query=hello+world
```

The response would be similar to that of the home timeline, except for tweets matching the given query.

Step 4: Scale the design

Identify and address bottlenecks, given the constraints.



Imgur

Important: Do not simply jump right into the final design from the initial design!

State you would 1) **Benchmark/Load Test**, 2) **Profile** for bottlenecks 3) address bottlenecks while evaluating alternatives and trade-offs, and 4) repeat. See [Design a system that scales to millions of users on AWS](#) as a sample on how to iteratively scale the initial design.

It's important to discuss what bottlenecks you might encounter with the initial design and how you might address each of them. For example, what issues are addressed by adding a **Load Balancer** with multiple **Web Servers**? **CDN**? **Master-Slave Replicas**? What are the alternatives and **Trade-Offs** for each?

We'll introduce some components to complete the design and to address scalability issues. Internal load balancers are not shown to reduce clutter.

To avoid repeating discussions, refer to the following [system design topics](#) for main talking points, tradeoffs, and alternatives:

- [DNS](#)
- [CDN](#)
- [Load balancer](#)
- [Horizontal scaling](#)
- [Web server \(reverse proxy\)](#)
- [API server \(application layer\)](#)
- [Cache](#)
- [Relational database management system \(RDBMS\)](#)
- [SQL write master-slave failover](#)
- [Master-slave replication](#)
- [Consistency patterns](#)
- [Availability patterns](#)

The **Fanout Service** is a potential bottleneck. Twitter users with millions of followers could take several minutes to have their tweets go through the fanout process. This could lead to race conditions with @replies to the tweet, which we could mitigate by re-ordering the tweets at serve time.

We could also avoid fanning out tweets from highly-followed users. Instead, we could search to find tweets for highly-followed users, merge the search results with the user's home timeline results, then re-order the tweets at serve time.

Additional optimizations include:

- Keep only several hundred tweets for each home timeline in the **Memory Cache**
- Keep only active users' home timeline info in the **Memory Cache**
 - If a user was not previously active in the past 30 days, we could rebuild the timeline from the **SQL Database**
 - Query the **User Graph Service** to determine who the user is following
 - Get the tweets from the **SQL Database** and add them to the **Memory Cache**
- Store only a month of tweets in the **Tweet Info Service**
- Store only active users in the **User Info Service**

- The **Search Cluster** would likely need to keep the tweets in memory to keep latency low

We'll also want to address the bottleneck with the **SQL Database**.

Although the **Memory Cache** should reduce the load on the database, it is unlikely the **SQL Read Replicas** alone would be enough to handle the cache misses. We'll probably need to employ additional SQL scaling patterns.

The high volume of writes would overwhelm a single **SQL Write Master-Slave**, also pointing to a need for additional scaling techniques.

- [Federation](#)
- [Sharding](#)
- [Denormalization](#)
- [SQL Tuning](#)

We should also consider moving some data to a **NoSQL Database**.

Additional talking points

Additional topics to dive into, depending on the problem scope and time remaining.

NoSQL

- [Key-value store](#)
- [Document store](#)
- [Wide column store](#)
- [Graph database](#)
- [SQL vs NoSQL](#)

Caching

- Where to cache
 - [Client caching](#)
 - [CDN caching](#)
 - [Web server caching](#)
 - [Database caching](#)
 - [Application caching](#)

- What to cache
 - [Caching at the database query level](#)
 - [Caching at the object level](#)
- When to update the cache
 - [Cache-aside](#)
 - [Write-through](#)
 - [Write-behind \(write-back\)](#)
 - [Refresh ahead](#)

Asynchronism and microservices

- [Message queues](#)
- [Task queues](#)
- [Back pressure](#)
- [Microservices](#)

Communications

- Discuss tradeoffs:
 - External communication with clients - [HTTP APIs following REST](#)
 - Internal communications - [RPC](#)
- [Service discovery](#)

Security

Refer to the [security section](#).

Latency numbers

See [Latency numbers every programmer should know](#).

Ongoing

- Continue benchmarking and monitoring your system to address bottlenecks as they come up
- Scaling is an iterative process

System Design: Web Crawler

Design a web crawler

Note: This document links directly to relevant areas found in the [system design topics](#) to avoid duplication. Refer to the linked content for general talking points, tradeoffs, and alternatives.

Step 1: Outline use cases and constraints

Gather requirements and scope the problem. Ask questions to clarify use cases and constraints. Discuss assumptions.

Without an interviewer to address clarifying questions, we'll define some use cases and constraints.

Use cases

We'll scope the problem to handle only the following use cases

- **Service** crawls a list of urls:
 - Generates reverse index of words to pages containing the search terms
 - Generates titles and snippets for pages
 - Title and snippets are static, they do not change based on search query
- **User** inputs a search term and sees a list of relevant pages with titles and snippets the crawler generated
 - Only sketch high level components and interactions for this use case, no need to go into depth
- **Service** has high availability

Out of scope

- Search analytics
- Personalized search results
- Page rank

Constraints and assumptions

State assumptions

- Traffic is not evenly distributed
 - Some searches are very popular, while others are only executed once
- Support only anonymous users
- Generating search results should be fast
- The web crawler should not get stuck in an infinite loop
 - We get stuck in an infinite loop if the graph contains a cycle
- 1 billion links to crawl
 - Pages need to be crawled regularly to ensure freshness
 - Average refresh rate of about once per week, more frequent for popular sites
 - 4 billion links crawled each month
 - Average stored size per web page: 500 KB
 - For simplicity, count changes the same as new pages
- 100 billion searches per month

Exercise the use of more traditional systems - don't use existing systems such as [solr](#) or [nutch](#).

Calculate usage

Clarify with your interviewer if you should run back-of-the-envelope usage calculations.

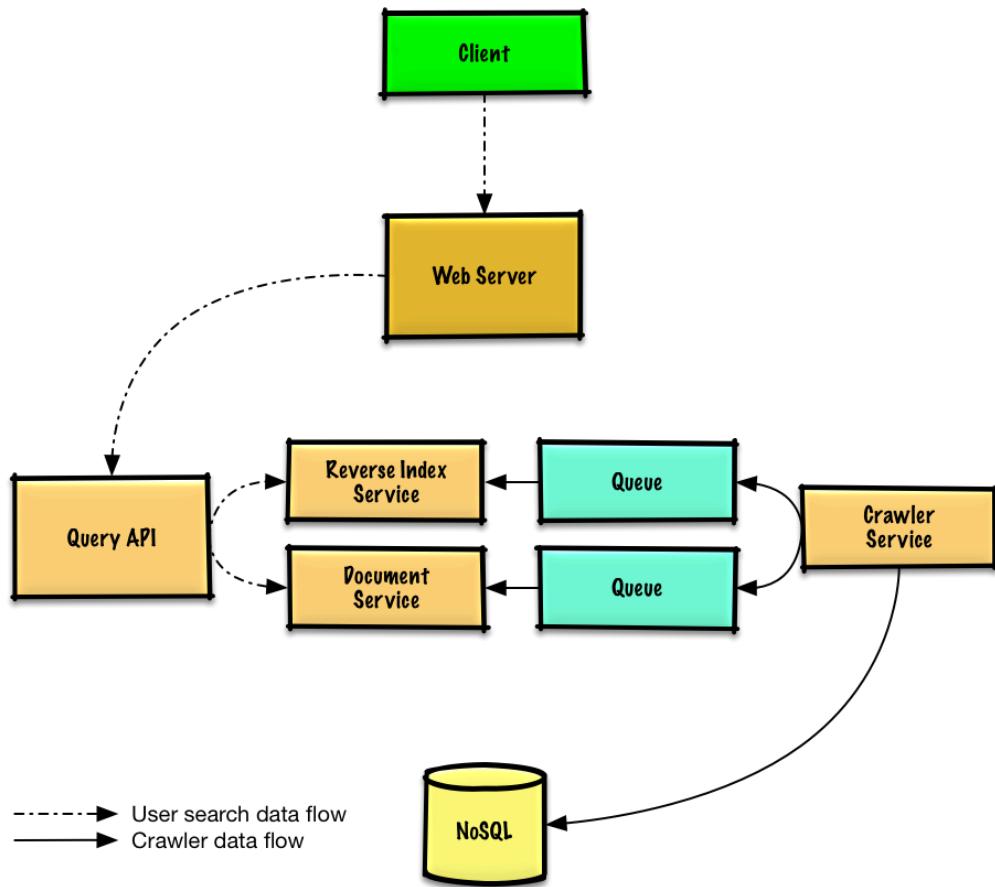
- 2 PB of stored page content per month
 - 500 KB per page * 4 billion links crawled per month
 - 72 PB of stored page content in 3 years
- 1,600 write requests per second
- 40,000 search requests per second

Handy conversion guide:

- 2.5 million seconds per month
- 1 request per second = 2.5 million requests per month
- 40 requests per second = 100 million requests per month
- 400 requests per second = 1 billion requests per month

Step 2: Create a high level design

Outline a high level design with all important components.



Imgur

Step 3: Design core components

Dive into details for each core component.

Use case: Service crawls a list of urls

We'll assume we have an initial list of `links_to_crawl` ranked initially based on overall site popularity. If this is not a reasonable assumption, we can seed the crawler with popular sites that link to outside content such as [Yahoo](#), [DMOZ](#), etc.

We'll use a table `crawled_links` to store processed links and their page signatures.

We could store `links_to_crawl` and `crawled_links` in a key-value **NoSQL Database**. For the ranked links in `links_to_crawl`, we could use [Redis](#) with sorted sets to maintain a ranking of page links. We should discuss the [use cases and tradeoffs between choosing SQL or NoSQL](#).

- The **Crawler Service** processes each page link by doing the following in a loop:
 - Takes the top ranked page link to crawl
 - Checks `crawled_links` in the **NoSQL Database** for an entry with a similar page signature
 - If we have a similar page, reduces the priority of the page link
 - This prevents us from getting into a cycle
 - Continue
 - Else, crawls the link
 - Adds a job to the **Reverse Index Service** queue to generate a [reverse index](#)
 - Adds a job to the **Document Service** queue to generate a static title and snippet
 - Generates the page signature
 - Removes the link from `links_to_crawl` in the **NoSQL Database**
 - Inserts the page link and signature to `crawled_links` in the **NoSQL Database**

Clarify with your interviewer how much code you are expected to write.

PagesDataStore is an abstraction within the **Crawler Service** that uses the **NoSQL Database**:

```

class PagesDataStore(object):

    def __init__(self, db);
        self.db = db
        ...

    def add_link_to_crawl(self, url):
        """Add the given link to `links_to_crawl`."""
        ...

    def remove_link_to_crawl(self, url):
        """Remove the given link from `links_to_crawl`."""
        ...

    def reduce_priority_link_to_crawl(self, url)
        """Reduce the priority of a link in `links_to_crawl` to avoid
        cycles."""
        ...

    def extract_max_priority_page(self):
        """Return the highest priority link in `links_to_crawl`."""
        ...

    def insert_crawled_link(self, url, signature):
        """Add the given link to `crawled_links`."""
        ...

    def crawled_similar(self, signature):
        """Determine if we've already crawled a page matching the given
        signature"""
        ...

```

Page is an abstraction within the **Crawler Service** that encapsulates a page, its contents, child urls, and signature:

```

class Page(object):

    def __init__(self, url, contents, child_urls, signature):
        self.url = url
        self.contents = contents
        self.child_urls = child_urls
        self.signature = signature

```

Crawler is the main class within **Crawler Service**, composed of Page and PagesDataStore.

```
class Crawler(object):

    def __init__(self, data_store, reverse_index_queue, doc_index_queue):
        self.data_store = data_store
        self.reverse_index_queue = reverse_index_queue
        self.doc_index_queue = doc_index_queue

    def create_signature(self, page):
        """Create signature based on url and contents."""
        ...

    def crawl_page(self, page):
        for url in page.child_urls:
            self.data_store.add_link_to_crawl(url)
        page.signature = self.create_signature(page)
        self.data_store.remove_link_to_crawl(page.url)
        self.data_store.insert_crawled_link(page.url, page.signature)

    def crawl(self):
        while True:
            page = self.data_store.extract_max_priority_page()
            if page is None:
                break
            if self.data_store.crawled_similar(page.signature):
                self.data_store.reduce_priority_link_to_crawl(page.url)
            else:
                self.crawl_page(page)
```

Handling duplicates

We need to be careful the web crawler doesn't get stuck in an infinite loop, which happens when the graph contains a cycle.

Clarify with your interviewer how much code you are expected to write.

We'll want to remove duplicate urls:

- For smaller lists we could use something like `sort | unique`

- With 1 billion links to crawl, we could use **MapReduce** to output only entries that have a frequency of 1

```
class RemoveDuplicateUrls(MRJob):

    def mapper(self, _, line):
        yield line, 1

    def reducer(self, key, values):
        total = sum(values)
        if total == 1:
            yield key, total
```

Detecting duplicate content is more complex. We could generate a signature based on the contents of the page and compare those two signatures for similarity. Some potential algorithms are [Jaccard index](#) and [cosine similarity](#).

Determining when to update the crawl results

Pages need to be crawled regularly to ensure freshness. Crawl results could have a timestamp field that indicates the last time a page was crawled. After a default time period, say one week, all pages should be refreshed. Frequently updated or more popular sites could be refreshed in shorter intervals.

Although we won't dive into details on analytics, we could do some data mining to determine the mean time before a particular page is updated, and use that statistic to determine how often to re-crawl the page.

We might also choose to support a Robots.txt file that gives webmasters control of crawl frequency.

Use case: User inputs a search term and sees a list of relevant pages with titles and snippets

- The **Client** sends a request to the **Web Server**, running as a [reverse proxy](#)
- The **Web Server** forwards the request to the **Query API** server
- The **Query API** server does the following:
 - Parses the query
 - Removes markup
 - Breaks up the text into terms

- Fixes typos
- Normalizes capitalization
- Converts the query to use boolean operations
- Uses the **Reverse Index Service** to find documents matching the query
 - The **Reverse Index Service** ranks the matching results and returns the top ones
- Uses the **Document Service** to return titles and snippets

We'll use a public [REST API](#):

```
$ curl https://search.com/api/v1/search?query=hello+world
```

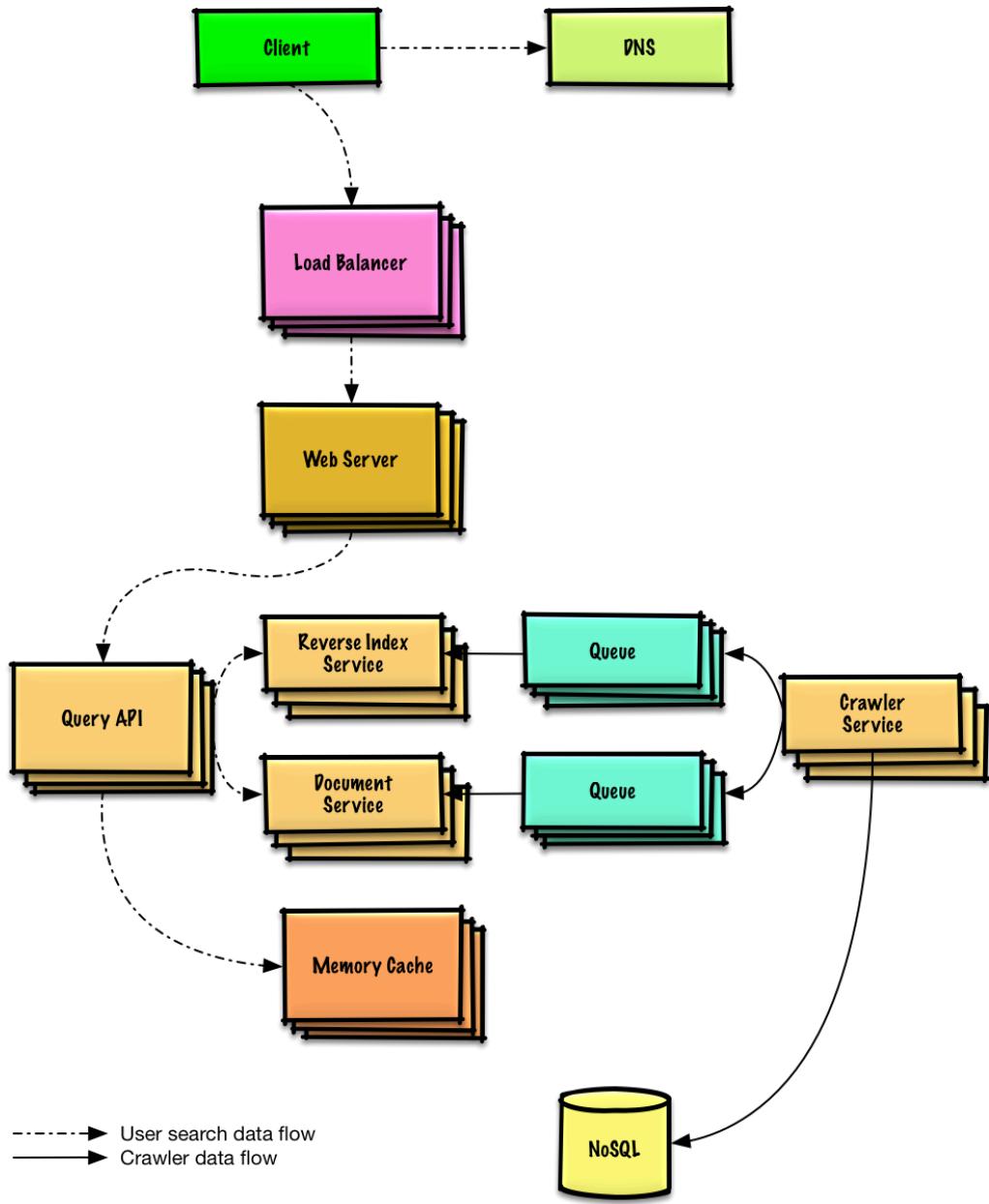
Response:

```
{
  "title": "foo's title",
  "snippet": "foo's snippet",
  "link": "https://foo.com",
},
{
  "title": "bar's title",
  "snippet": "bar's snippet",
  "link": "https://bar.com",
},
{
  "title": "baz's title",
  "snippet": "baz's snippet",
  "link": "https://baz.com",
},
```

For internal communications, we could use [Remote Procedure Calls](#).

Step 4: Scale the design

Identify and address bottlenecks, given the constraints.



Imgur

Important: Do not simply jump right into the final design from the initial design!

State you would 1) **Benchmark/Load Test**, 2) **Profile** for bottlenecks 3) address bottlenecks while evaluating alternatives and trade-offs, and 4) repeat. See [Design a system that scales to millions of users on AWS](#) as a sample on how to iteratively scale the initial design.

It's important to discuss what bottlenecks you might encounter with the initial design and how you might address each of them. For example, what issues are addressed by

adding a **Load Balancer** with multiple **Web Servers?** **CDN?** **Master-Slave Replicas?** What are the alternatives and **Trade-Offs** for each?

We'll introduce some components to complete the design and to address scalability issues. Internal load balancers are not shown to reduce clutter.

To avoid repeating discussions, refer to the following [system design topics](#) for main talking points, tradeoffs, and alternatives:

- [DNS](#)
- [Load balancer](#)
- [Horizontal scaling](#)
- [Web server \(reverse proxy\)](#)
- [API server \(application layer\)](#)
- [Cache](#)
- [NoSQL](#)
- [Consistency patterns](#)
- [Availability patterns](#)

Some searches are very popular, while others are only executed once. Popular queries can be served from a **Memory Cache** such as Redis or Memcached to reduce response times and to avoid overloading the **Reverse Index Service** and **Document Service**. The **Memory Cache** is also useful for handling the unevenly distributed traffic and traffic spikes. Reading 1 MB sequentially from memory takes about 250 microseconds, while reading from SSD takes 4x and from disk takes 80x longer.¹

Below are a few other optimizations to the **Crawling Service**:

- To handle the data size and request load, the **Reverse Index Service** and **Document Service** will likely need to make heavy use sharding and federation.
- DNS lookup can be a bottleneck, the **Crawler Service** can keep its own DNS lookup that is refreshed periodically
- The **Crawler Service** can improve performance and reduce memory usage by keeping many open connections at a time, referred to as [connection pooling](#)
 - Switching to [UDP](#) could also boost performance
- Web crawling is bandwidth intensive, ensure there is enough bandwidth to sustain high throughput

Additional talking points

Additional topics to dive into, depending on the problem scope and time remaining.

SQL scaling patterns

- [Read replicas](#)
- [Federation](#)
- [Sharding](#)
- [Denormalization](#)
- [SQL Tuning](#)

NoSQL

- [Key-value store](#)
- [Document store](#)
- [Wide column store](#)
- [Graph database](#)
- [SQL vs NoSQL](#)

Caching

- Where to cache
 - [Client caching](#)
 - [CDN caching](#)
 - [Web server caching](#)
 - [Database caching](#)
 - [Application caching](#)
- What to cache
 - [Caching at the database query level](#)
 - [Caching at the object level](#)
- When to update the cache
 - [Cache-aside](#)
 - [Write-through](#)
 - [Write-behind \(write-back\)](#)
 - [Refresh ahead](#)

Asynchronism and microservices

- [Message queues](#)
- [Task queues](#)
- [Back pressure](#)
- [Microservices](#)

Communications

- Discuss tradeoffs:
 - External communication with clients - [HTTP APIs following REST](#)
 - Internal communications - [RPC](#)
- [Service discovery](#)

Security

Refer to the [security section](#).

Latency numbers

See [Latency numbers every programmer should know](#).

Ongoing

- Continue benchmarking and monitoring your system to address bottlenecks as they come up
- Scaling is an iterative process

System Design: Mint

Design Mint.com

Note: This document links directly to relevant areas found in the [system design topics](#) to avoid duplication. Refer to the linked content for general talking points, tradeoffs, and alternatives.

Step 1: Outline use cases and constraints

Gather requirements and scope the problem. Ask questions to clarify use cases and constraints. Discuss assumptions.

Without an interviewer to address clarifying questions, we'll define some use cases and constraints.

Use cases

We'll scope the problem to handle only the following use cases

- **User** connects to a financial account
- **Service** extracts transactions from the account
 - Updates daily
 - Categorizes transactions
 - Allows manual category override by the user
 - No automatic re-categorization
 - Analyzes monthly spending, by category
- **Service** recommends a budget
 - Allows users to manually set a budget
 - Sends notifications when approaching or exceeding budget
- **Service** has high availability

Out of scope

- **Service** performs additional logging and analytics

Constraints and assumptions

State assumptions

- Traffic is not evenly distributed
- Automatic daily update of accounts applies only to users active in the past 30 days
- Adding or removing financial accounts is relatively rare
- Budget notifications don't need to be instant
- 10 million users
 - 10 budget categories per user = 100 million budget items
 - Example categories:

- Housing = \$1,000
 - Food = \$200
 - Gas = \$100
- Sellers are used to determine transaction category
 - 50,000 sellers
- 30 million financial accounts
- 5 billion transactions per month
- 500 million read requests per month
- 10:1 write to read ratio
 - Write-heavy, users make transactions daily, but few visit the site daily

Calculate usage

Clarify with your interviewer if you should run back-of-the-envelope usage calculations.

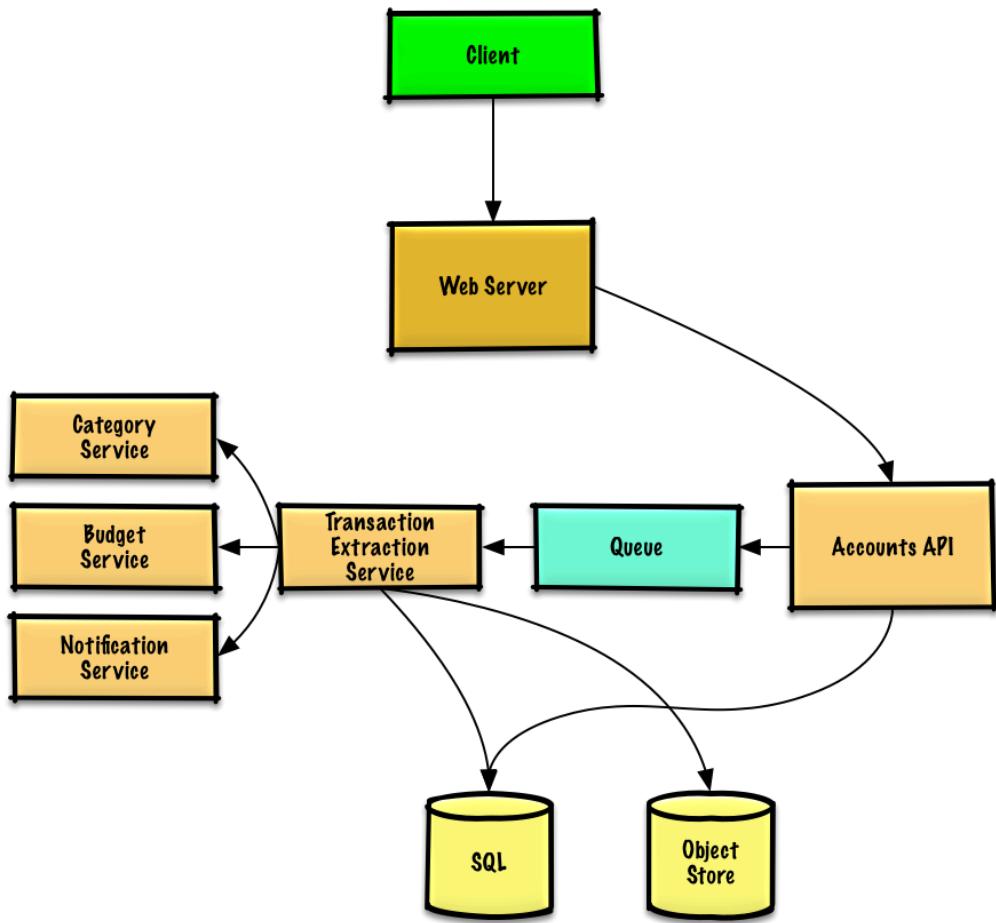
- Size per transaction:
 - user_id - 8 bytes
 - created_at - 5 bytes
 - seller - 32 bytes
 - amount - 5 bytes
 - Total: ~50 bytes
- 250 GB of new transaction content per month
 - 50 bytes per transaction * 5 billion transactions per month
 - 9 TB of new transaction content in 3 years
 - Assume most are new transactions instead of updates to existing ones
- 2,000 transactions per second on average
- 200 read requests per second on average

Handy conversion guide:

- 2.5 million seconds per month
- 1 request per second = 2.5 million requests per month
- 40 requests per second = 100 million requests per month
- 400 requests per second = 1 billion requests per month

Step 2: Create a high level design

Outline a high level design with all important components.



Imgur

Step 3: Design core components

Dive into details for each core component.

Use case: User connects to a financial account

We could store info on the 10 million users in a [relational database](#). We should discuss the [use cases and tradeoffs between choosing SQL or NoSQL](#).

- The **Client** sends a request to the **Web Server**, running as a [reverse proxy](#).
- The **Web Server** forwards the request to the **Accounts API** server

- The **Accounts API** server updates the **SQL Database** accounts table with the newly entered account info

Clarify with your interviewer how much code you are expected to write.

The accounts table could have the following structure:

```
id int NOT NULL AUTO_INCREMENT
created_at datetime NOT NULL
last_update datetime NOT NULL
account_url varchar(255) NOT NULL
account_login varchar(32) NOT NULL
account_password_hash char(64) NOT NULL
user_id int NOT NULL
PRIMARY KEY(id)
FOREIGN KEY(user_id) REFERENCES users(id)
```

We'll create an [index](#) on id, user_id, and created_at to speed up lookups (log-time instead of scanning the entire table) and to keep the data in memory. Reading 1 MB sequentially from memory takes about 250 microseconds, while reading from SSD takes 4x and from disk takes 80x longer.¹

We'll use a public [REST API](#):

```
$ curl -X POST --data '{ "user_id": "foo", "account_url": "bar", \
"account_login": "baz", "account_password": "qux" }' \
https://mint.com/api/v1/account
```

For internal communications, we could use [Remote Procedure Calls](#).

Next, the service extracts transactions from the account.

Use case: Service extracts transactions from the account

We'll want to extract information from an account in these cases:

- The user first links the account
- The user manually refreshes the account
- Automatically each day for users who have been active in the past 30 days

Data flow:

- The **Client** sends a request to the **Web Server**
- The **Web Server** forwards the request to the **Accounts API** server
- The **Accounts API** server places a job on a **Queue** such as [Amazon SQS](#) or [RabbitMQ](#)
 - Extracting transactions could take awhile, we'd probably want to do this [asynchronously with a queue](#), although this introduces additional complexity
- The **Transaction Extraction Service** does the following:
 - Pulls from the **Queue** and extracts transactions for the given account from the financial institution, storing the results as raw log files in the **Object Store**
 - Uses the **Category Service** to categorize each transaction
 - Uses the **Budget Service** to calculate aggregate monthly spending by category
 - The **Budget Service** uses the **Notification Service** to let users know if they are nearing or have exceeded their budget
 - Updates the **SQL Database** `transactions` table with categorized transactions
 - Updates the **SQL Database** `monthly_spending` table with aggregate monthly spending by category
 - Notifies the user the transactions have completed through the **Notification Service**:
 - Uses a **Queue** (not pictured) to asynchronously send out notifications

The `transactions` table could have the following structure:

```

id int NOT NULL AUTO_INCREMENT
created_at datetime NOT NULL
seller varchar(32) NOT NULL
amount decimal NOT NULL
user_id int NOT NULL
PRIMARY KEY(id)
FOREIGN KEY(user_id) REFERENCES users(id)
  
```

We'll create an [index](#) on `id`, `user_id`, and `created_at`.

The `monthly_spending` table could have the following structure:

```
id int NOT NULL AUTO_INCREMENT
month_year date NOT NULL
category varchar(32)
amount decimal NOT NULL
user_id int NOT NULL
PRIMARY KEY(id)
FOREIGN KEY(user_id) REFERENCES users(id)
```

We'll create an [index](#) on id and user_id.

Category service

For the **Category Service**, we can seed a seller-to-category dictionary with the most popular sellers. If we estimate 50,000 sellers and estimate each entry to take less than 255 bytes, the dictionary would only take about 12 MB of memory.

Clarify with your interviewer how much code you are expected to write.

```
class DefaultCategories(Enum):

    HOUSING = 0
    FOOD = 1
    GAS = 2
    SHOPPING = 3
    ...

    seller_category_map = {}
    seller_category_map['Exxon'] = DefaultCategories.GAS
    seller_category_map['Target'] = DefaultCategories.SHOPPING
    ...
```

For sellers not initially seeded in the map, we could use a crowdsourcing effort by evaluating the manual category overrides our users provide. We could use a heap to quickly lookup the top manual override per seller in O(1) time.

```

class Categorizer(object):

    def __init__(self, seller_category_map,
                seller_category_crowd_overrides_map):
        self.seller_category_map = seller_category_map
        self.seller_category_crowd_overrides_map = \
            seller_category_crowd_overrides_map

    def categorize(self, transaction):
        if transaction.seller in self.seller_category_map:
            return self.seller_category_map[transaction.seller]
        elif transaction.seller in self.seller_category_crowd_overrides_map:
            self.seller_category_map[transaction.seller] = \
                self.seller_category_crowd_overrides_map[transaction.seller].peek_min()
            return self.seller_category_map[transaction.seller]
        return None

```

Transaction implementation:

```

class Transaction(object):

    def __init__(self, created_at, seller, amount):
        self.created_at = created_at
        self.seller = seller
        self.amount = amount

```

Use case: Service recommends a budget

To start, we could use a generic budget template that allocates category amounts based on income tiers. Using this approach, we would not have to store the 100 million budget items identified in the constraints, only those that the user overrides. If a user overrides a budget category, which we could store the override in the TABLE budget_overrides.

```

class Budget(object):

    def __init__(self, income):
        self.income = income
        self.categories_to_budget_map = self.create_budget_template()

    def create_budget_template(self):
        return {
            DefaultCategories.HOUSING: self.income * .4,
            DefaultCategories.FOOD: self.income * .2,
            DefaultCategories.GAS: self.income * .1,
            DefaultCategories.SHOPPING: self.income * .2,
            ...
        }

    def override_category_budget(self, category, amount):
        self.categories_to_budget_map[category] = amount

```

For the **Budget Service**, we can potentially run SQL queries on the transactions table to generate the `monthly_spending` aggregate table. The `monthly_spending` table would likely have much fewer rows than the total 5 billion transactions, since users typically have many transactions per month.

As an alternative, we can run **MapReduce** jobs on the raw transaction files to:

- Categorize each transaction
- Generate aggregate monthly spending by category

Running analyses on the transaction files could significantly reduce the load on the database.

We could call the **Budget Service** to re-run the analysis if the user updates a category.

Clarify with your interviewer how much code you are expected to write.

Sample log file format, tab delimited:

```
user_id    timestamp    seller    amount
```

MapReduce implementation:

```

class SpendingByCategory(MRJob):

    def __init__(self, categorizer):
        self.categorizer = categorizer
        self.current_year_month = calc_current_year_month()
        ...

    def calc_current_year_month(self):
        """Return the current year and month."""
        ...

    def extract_year_month(self, timestamp):
        """Return the year and month portions of the timestamp."""
        ...

    def handle_budget_notifications(self, key, total):
        """Call notification API if nearing or exceeded budget."""
        ...

    def mapper(self, _, line):
        """Parse each log line, extract and transform relevant lines.

        Argument line will be of the form:

        user_id    timestamp    seller    amount

        Using the categorizer to convert seller to category,
        emit key value pairs of the form:

        (user_id, 2016-01, shopping), 25
        (user_id, 2016-01, shopping), 100
        (user_id, 2016-01, gas), 50
        ...
        user_id, timestamp, seller, amount = line.split('\t')
        category = self.categorizer.categorize(seller)
        period = self.extract_year_month(timestamp)
        if period == self.current_year_month:
            yield (user_id, period, category), amount

    def reducer(self, key, value):
        """Sum values for each key.

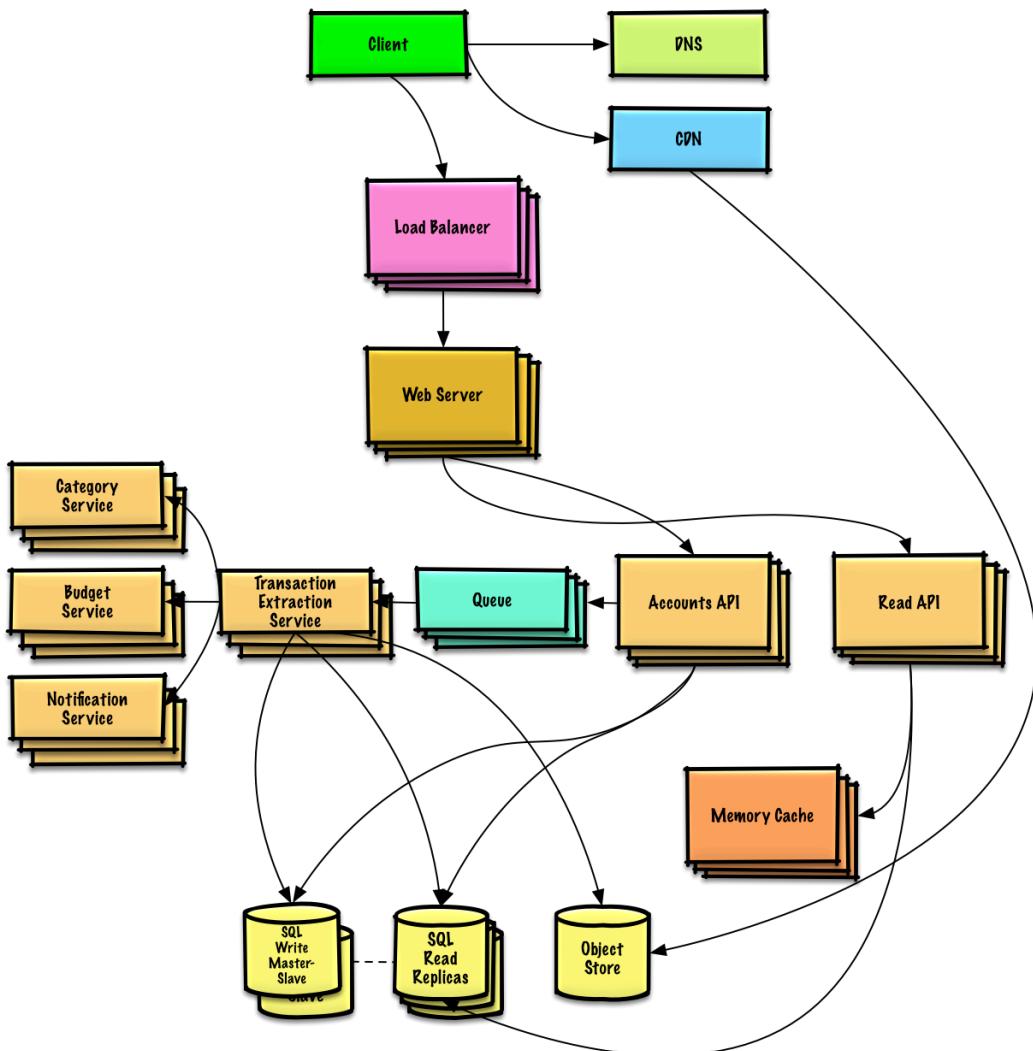
        (user_id, 2016-01, shopping), 125
        (user_id, 2016-01, gas), 50
    """

```

```
....  
total = sum(values)  
yield key, sum(values)
```

Step 4: Scale the design

Identify and address bottlenecks, given the constraints.



Imgur

Important: Do not simply jump right into the final design from the initial design!

State you would 1) **Benchmark/Load Test**, 2) **Profile** for bottlenecks 3) address bottlenecks while evaluating alternatives and trade-offs, and 4) repeat. See [Design a](#)

[system that scales to millions of users on AWS](#) as a sample on how to iteratively scale the initial design.

It's important to discuss what bottlenecks you might encounter with the initial design and how you might address each of them. For example, what issues are addressed by adding a **Load Balancer** with multiple **Web Servers?** **CDN?** **Master-Slave Replicas?** What are the alternatives and **Trade-Offs** for each?

We'll introduce some components to complete the design and to address scalability issues. Internal load balancers are not shown to reduce clutter.

To avoid repeating discussions, refer to the following [system design topics](#) for main talking points, tradeoffs, and alternatives:

- [DNS](#)
- [CDN](#)
- [Load balancer](#)
- [Horizontal scaling](#)
- [Web server \(reverse proxy\)](#)
- [API server \(application layer\)](#)
- [Cache](#)
- [Relational database management system \(RDBMS\)](#)
- [SQL write master-slave failover](#)
- [Master-slave replication](#)
- [Asynchronism](#)
- [Consistency patterns](#)
- [Availability patterns](#)

We'll add an additional use case: **User** accesses summaries and transactions.

User sessions, aggregate stats by category, and recent transactions could be placed in a **Memory Cache** such as Redis or Memcached.

- The **Client** sends a read request to the **Web Server**
- The **Web Server** forwards the request to the **Read API** server
 - Static content can be served from the **Object Store** such as S3, which is cached on the **CDN**
- The **Read API** server does the following:
 - Checks the **Memory Cache** for the content

- If the url is in the **Memory Cache**, returns the cached contents
- Else
 - If the url is in the **SQL Database**, fetches the contents
 - Updates the **Memory Cache** with the contents

Refer to [When to update the cache](#) for tradeoffs and alternatives. The approach above describes [cache-aside](#).

Instead of keeping the `monthly_spending` aggregate table in the **SQL Database**, we could create a separate **Analytics Database** using a data warehousing solution such as Amazon Redshift or Google BigQuery.

We might only want to store a month of transactions data in the database, while storing the rest in a data warehouse or in an **Object Store**. An **Object Store** such as Amazon S3 can comfortably handle the constraint of 250 GB of new content per month.

To address the 200 average read requests per second (higher at peak), traffic for popular content should be handled by the **Memory Cache** instead of the database. The **Memory Cache** is also useful for handling the unevenly distributed traffic and traffic spikes. The **SQL Read Replicas** should be able to handle the cache misses, as long as the replicas are not bogged down with replicating writes.

2,000 average transaction writes per second (higher at peak) might be tough for a single **SQL Write Master-Slave**. We might need to employ additional SQL scaling patterns:

- [Federation](#)
- [Sharding](#)
- [Denormalization](#)
- [SQL Tuning](#)

We should also consider moving some data to a **NoSQL Database**.

Additional talking points

Additional topics to dive into, depending on the problem scope and time remaining.

NoSQL

- [Key-value store](#)
- [Document store](#)
- [Wide column store](#)
- [Graph database](#)
- [SQL vs NoSQL](#)

Caching

- Where to cache
 - [Client caching](#)
 - [CDN caching](#)
 - [Web server caching](#)
 - [Database caching](#)
 - [Application caching](#)
- What to cache
 - [Caching at the database query level](#)
 - [Caching at the object level](#)
- When to update the cache
 - [Cache-aside](#)
 - [Write-through](#)
 - [Write-behind \(write-back\)](#)
 - [Refresh ahead](#)

Asynchronism and microservices

- [Message queues](#)
- [Task queues](#)
- [Back pressure](#)
- [Microservices](#)

Communications

- Discuss tradeoffs:
 - External communication with clients - [HTTP APIs following REST](#)
 - Internal communications - [RPC](#)
- [Service discovery](#)

Security

Refer to the [security section](#).

Latency numbers

See [Latency numbers every programmer should know](#).

Ongoing

- Continue benchmarking and monitoring your system to address bottlenecks as they come up
- Scaling is an iterative process

System Design: Social Graph

Design the data structures for a social network

Note: This document links directly to relevant areas found in the [system design topics](#) to avoid duplication. Refer to the linked content for general talking points, tradeoffs, and alternatives.

Step 1: Outline use cases and constraints

Gather requirements and scope the problem. Ask questions to clarify use cases and constraints. Discuss assumptions.

Without an interviewer to address clarifying questions, we'll define some use cases and constraints.

Use cases

We'll scope the problem to handle only the following use cases

- **User** searches for someone and sees the shortest path to the searched person
- **Service** has high availability

Constraints and assumptions

State assumptions

- Traffic is not evenly distributed
 - Some searches are more popular than others, while others are only executed once
- Graph data won't fit on a single machine
- Graph edges are unweighted
- 100 million users
- 50 friends per user average
- 1 billion friend searches per month

Exercise the use of more traditional systems - don't use graph-specific solutions such as [GraphQL](#) or a graph database like [Neo4j](#)

Calculate usage

Clarify with your interviewer if you should run back-of-the-envelope usage calculations.

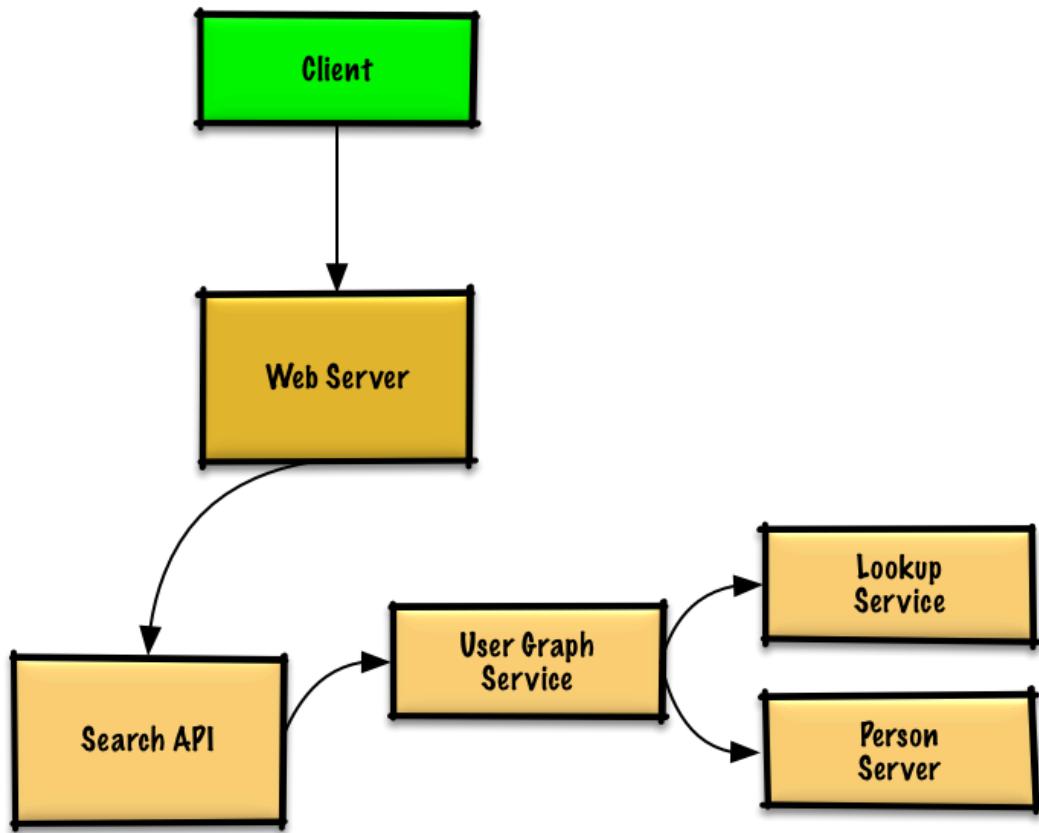
- 5 billion friend relationships
 - 100 million users * 50 friends per user average
- 400 search requests per second

Handy conversion guide:

- 2.5 million seconds per month
- 1 request per second = 2.5 million requests per month
- 40 requests per second = 100 million requests per month
- 400 requests per second = 1 billion requests per month

Step 2: Create a high level design

Outline a high level design with all important components.



Imgur

Step 3: Design core components

Dive into details for each core component.

Use case: User searches for someone and sees the shortest path to the searched person

Clarify with your interviewer how much code you are expected to write.

Without the constraint of millions of users (vertices) and billions of friend relationships (edges), we could solve this unweighted shortest path task with a general BFS approach:

```

class Graph(Graph):

    def shortest_path(self, source, dest):
        if source is None or dest is None:
            return None
        if source is dest:
            return [source.key]
        prev_node_keys = self._shortest_path(source, dest)
        if prev_node_keys is None:
            return None
        else:
            path_ids = [dest.key]
            prev_node_key = prev_node_keys[dest.key]
            while prev_node_key is not None:
                path_ids.append(prev_node_key)
                prev_node_key = prev_node_keys[prev_node_key]
            return path_ids[::-1]

    def _shortest_path(self, source, dest):
        queue = deque()
        queue.append(source)
        prev_node_keys = {source.key: None}
        source.visit_state = State.visited
        while queue:
            node = queue.popleft()
            if node is dest:
                return prev_node_keys
            prev_node = node
            for adj_node in node.adj_nodes.values():
                if adj_node.visit_state == State.unvisited:
                    queue.append(adj_node)
                    prev_node_keys[adj_node.key] = prev_node.key
                    adj_node.visit_state = State.visited
        return None

```

We won't be able to fit all users on the same machine, we'll need to [shard](#) users across **Person Servers** and access them with a **Lookup Service**.

- The **Client** sends a request to the **Web Server**, running as a [reverse proxy](#)
- The **Web Server** forwards the request to the **Search API** server
- The **Search API** server forwards the request to the **User Graph Service**
- The **User Graph Service** does the following:

- Uses the **Lookup Service** to find the **Person Server** where the current user's info is stored
- Finds the appropriate **Person Server** to retrieve the current user's list of friend_ids
- Runs a BFS search using the current user as the source and the current user's friend_ids as the ids for each adjacent_node
- To get the adjacent_node from a given id:
 - The **User Graph Service** will *again* need to communicate with the **Lookup Service** to determine which **Person Server** stores the adjacent_node matching the given id (potential for optimization)

Clarify with your interviewer how much code you should be writing.

Note: Error handling is excluded below for simplicity. Ask if you should code proper error handing.

Lookup Service implementation:

```
class LookupService(object):

    def __init__(self):
        self.lookup = self._init_lookup() # key: person_id, value:
                                         person_server

    def _init_lookup(self):
        ...

    def lookup_person_server(self, person_id):
        return self.lookup[person_id]
```

Person Server implementation:

```
class PersonServer(object):

    def __init__(self):
        self.people = {} # key: person_id, value: person

    def add_person(self, person):
        ...

    def people(self, ids):
        results = []
        for id in ids:
            if id in self.people:
                results.append(self.people[id])
        return results
```

Person implementation:

```
class Person(object):

    def __init__(self, id, name, friend_ids):
        self.id = id
        self.name = name
        self.friend_ids = friend_ids
```

User Graph Service implementation:

```

class UserGraphService(object):

    def __init__(self, lookup_service):
        self.lookup_service = lookup_service

    def person(self, person_id):
        person_server = self.lookup_service.lookup_person_server(person_id)
        return person_server.people([person_id])

    def shortest_path(self, source_key, dest_key):
        if source_key is None or dest_key is None:
            return None
        if source_key is dest_key:
            return [source_key]
        prev_node_keys = self._shortest_path(source_key, dest_key)
        if prev_node_keys is None:
            return None
        else:
            # Iterate through the path_ids backwards, starting at dest_key
            path_ids = [dest_key]
            prev_node_key = prev_node_keys[dest_key]
            while prev_node_key is not None:
                path_ids.append(prev_node_key)
                prev_node_key = prev_node_keys[prev_node_key]
            # Reverse the list since we iterated backwards
            return path_ids[::-1]

    def _shortest_path(self, source_key, dest_key, path):
        # Use the id to get the Person
        source = self.person(source_key)
        # Update our bfs queue
        queue = deque()
        queue.append(source)
        # prev_node_keys keeps track of each hop from
        # the source_key to the dest_key
        prev_node_keys = {source_key: None}
        # We'll use visited_ids to keep track of which nodes we've
        # visited, which can be different from a typical bfs where
        # this can be stored in the node itself
        visited_ids = set()
        visited_ids.add(source.id)
        while queue:
            node = queue.popleft()
            if node.key is dest_key:

```

```

        return prev_node_keys
prev_node = node
for friend_id in node.friend_ids:
    if friend_id not in visited_ids:
        friend_node = self.person(friend_id)
        queue.append(friend_node)
        prev_node_keys[friend_id] = prev_node.key
        visited_ids.add(friend_id)
return None

```

We'll use a public [REST API](#):

```
$ curl https://social.com/api/v1/friend_search?person_id=1234
```

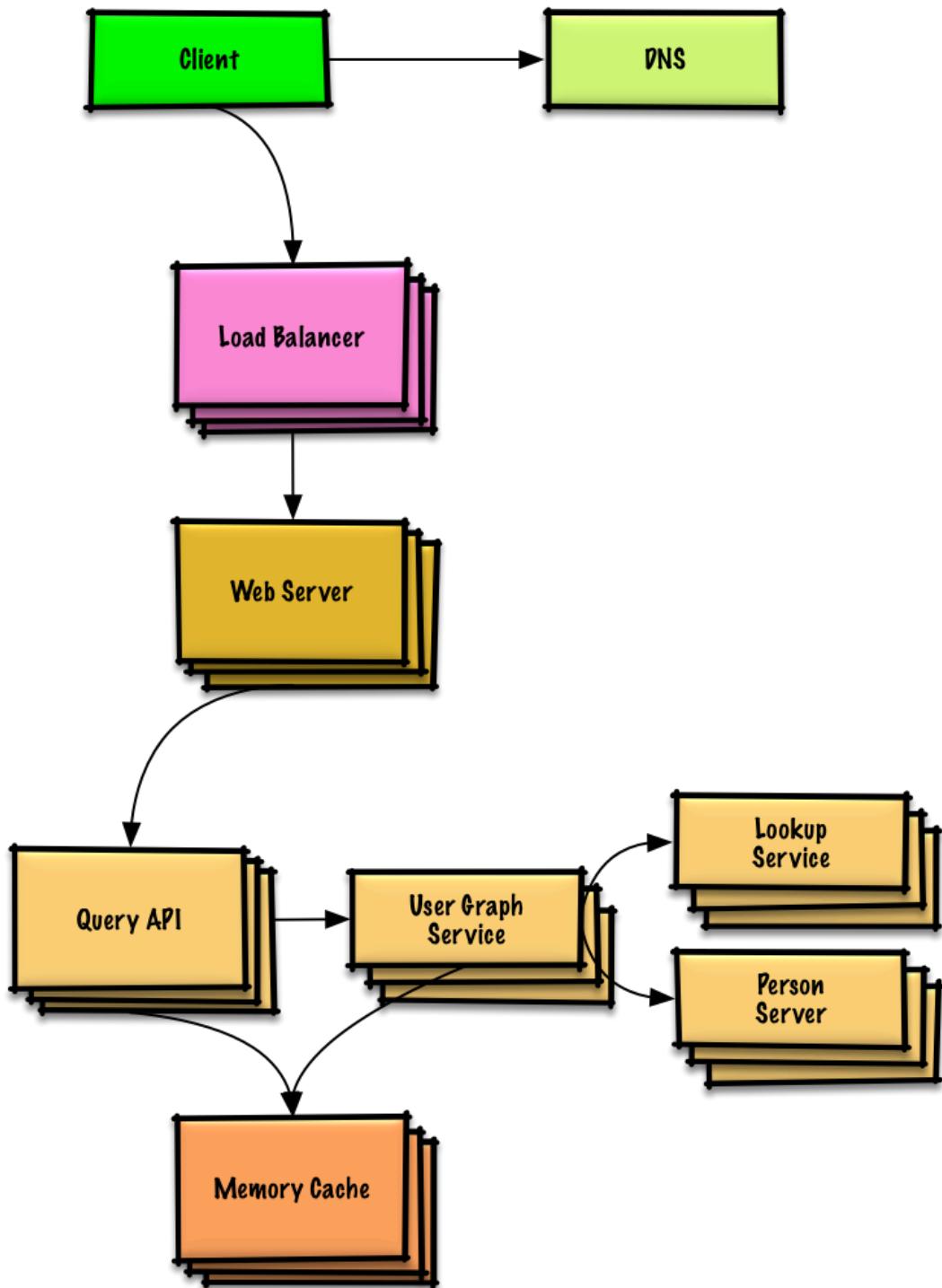
Response:

```
{
  "person_id": "100",
  "name": "foo",
  "link": "https://social.com/foo",
},
{
  "person_id": "53",
  "name": "bar",
  "link": "https://social.com/bar",
},
{
  "person_id": "1234",
  "name": "baz",
  "link": "https://social.com/baz",
},
```

For internal communications, we could use [Remote Procedure Calls](#).

Step 4: Scale the design

Identify and address bottlenecks, given the constraints.



Imgur

Important: Do not simply jump right into the final design from the initial design!

State you would 1) **Benchmark/Load Test**, 2) **Profile** for bottlenecks 3) address bottlenecks while evaluating alternatives and trade-offs, and 4) repeat. See [Design a system that scales to millions of users on AWS](#) as a sample on how to iteratively scale the initial design.

It's important to discuss what bottlenecks you might encounter with the initial design and how you might address each of them. For example, what issues are addressed by adding a **Load Balancer** with multiple **Web Servers?** **CDN?** **Master-Slave Replicas?** What are the alternatives and **Trade-Offs** for each?

We'll introduce some components to complete the design and to address scalability issues. Internal load balancers are not shown to reduce clutter.

To avoid repeating discussions, refer to the following [system design topics](#) for main talking points, tradeoffs, and alternatives:

- [DNS](#)
- [Load balancer](#)
- [Horizontal scaling](#)
- [Web server \(reverse proxy\)](#).
- [API server \(application layer\)](#).
- [Cache](#)
- [Consistency patterns](#)
- [Availability patterns](#)

To address the constraint of 400 average read requests per second (higher at peak), person data can be served from a **Memory Cache** such as Redis or Memcached to reduce response times and to reduce traffic to downstream services. This could be especially useful for people who do multiple searches in succession and for people who are well-connected. Reading 1 MB sequentially from memory takes about 250 microseconds, while reading from SSD takes 4x and from disk takes 80x longer.¹

Below are further optimizations:

- Store complete or partial BFS traversals to speed up subsequent lookups in the **Memory Cache**
- Batch compute offline then store complete or partial BFS traversals to speed up subsequent lookups in a **NoSQL Database**
- Reduce machine jumps by batching together friend lookups hosted on the same **Person Server**
 - [Shard Person Servers](#) by location to further improve this, as friends generally live closer to each other
- Do two BFS searches at the same time, one starting from the source, and one from the destination, then merge the two paths

- Start the BFS search from people with large numbers of friends, as they are more likely to reduce the number of [degrees of separation](#) between the current user and the search target
- Set a limit based on time or number of hops before asking the user if they want to continue searching, as searching could take a considerable amount of time in some cases
- Use a **Graph Database** such as [Neo4j](#) or a graph-specific query language such as [GraphQL](#) (if there were no constraint preventing the use of **Graph Databases**)

Additional talking points

Additional topics to dive into, depending on the problem scope and time remaining.

SQL scaling patterns

- [Read replicas](#)
- [Federation](#)
- [Sharding](#)
- [Denormalization](#)
- [SQL Tuning](#)

NoSQL

- [Key-value store](#)
- [Document store](#)
- [Wide column store](#)
- [Graph database](#)
- [SQL vs NoSQL](#)

Caching

- Where to cache
 - [Client caching](#)
 - [CDN caching](#)
 - [Web server caching](#)
 - [Database caching](#)
 - [Application caching](#)

- What to cache
 - [Caching at the database query level](#)
 - [Caching at the object level](#)
- When to update the cache
 - [Cache-aside](#)
 - [Write-through](#)
 - [Write-behind \(write-back\)](#)
 - [Refresh ahead](#)

Asynchronism and microservices

- [Message queues](#)
- [Task queues](#)
- [Back pressure](#)
- [Microservices](#)

Communications

- Discuss tradeoffs:
 - External communication with clients - [HTTP APIs following REST](#)
 - Internal communications - [RPC](#)
- [Service discovery](#)

Security

Refer to the [security section](#).

Latency numbers

See [Latency numbers every programmer should know](#).

Ongoing

- Continue benchmarking and monitoring your system to address bottlenecks as they come up
- Scaling is an iterative process

System Design: Query Cache

Design a key-value cache to save the results of the most recent web server queries

Note: This document links directly to relevant areas found in the [system design topics](#) to avoid duplication. Refer to the linked content for general talking points, tradeoffs, and alternatives.

Step 1: Outline use cases and constraints

Gather requirements and scope the problem. Ask questions to clarify use cases and constraints. Discuss assumptions.

Without an interviewer to address clarifying questions, we'll define some use cases and constraints.

Use cases

We'll scope the problem to handle only the following use cases

- **User** sends a search request resulting in a cache hit
- **User** sends a search request resulting in a cache miss
- **Service** has high availability

Constraints and assumptions

State assumptions

- Traffic is not evenly distributed
 - Popular queries should almost always be in the cache
 - Need to determine how to expire/refresh
- Serving from cache requires fast lookups
- Low latency between machines
- Limited memory in cache

- Need to determine what to keep/remove
- Need to cache millions of queries
- 10 million users
- 10 billion queries per month

Calculate usage

Clarify with your interviewer if you should run back-of-the-envelope usage calculations.

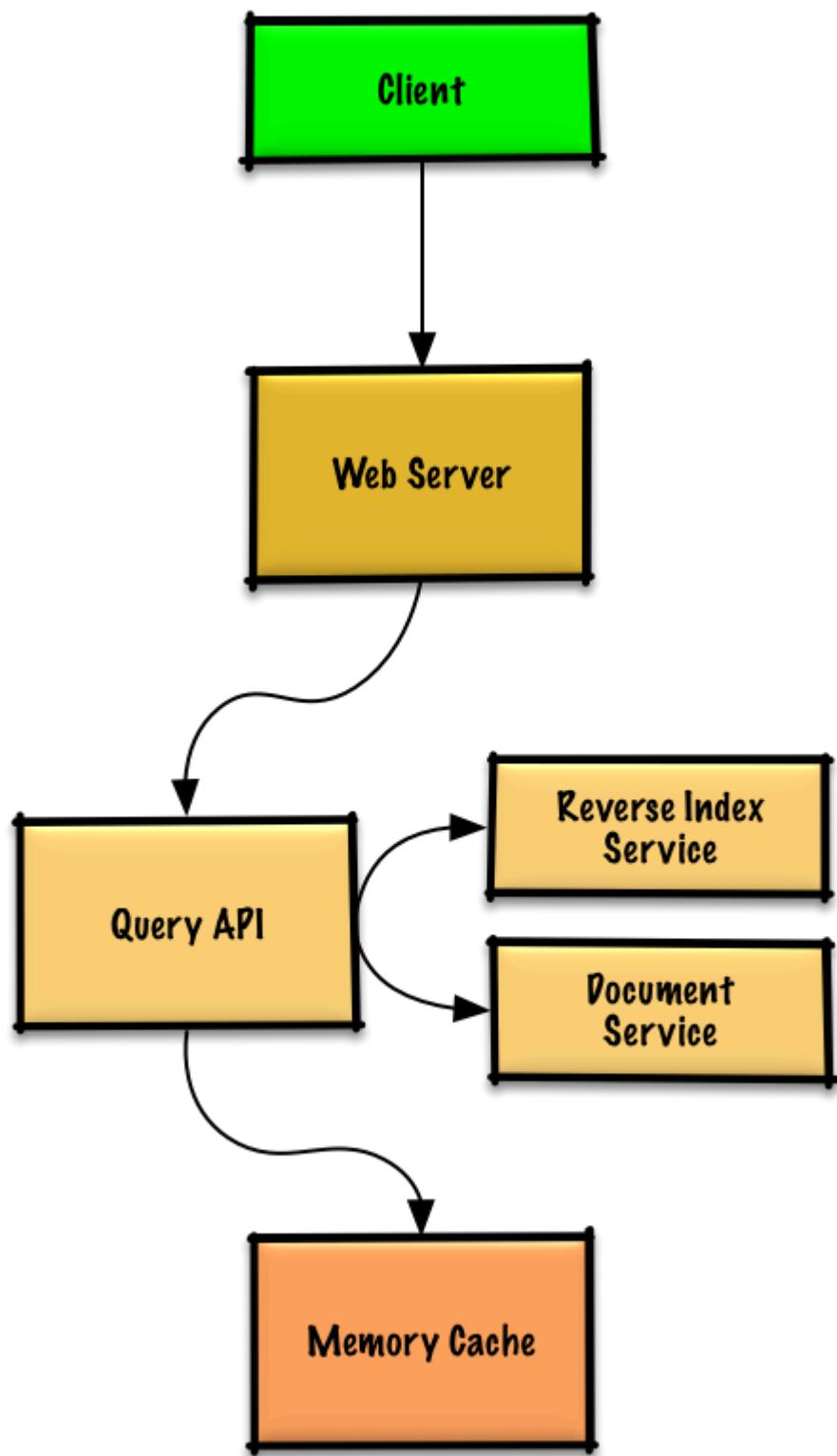
- Cache stores ordered list of key: query, value: results
 - query - 50 bytes
 - title - 20 bytes
 - snippet - 200 bytes
 - Total: 270 bytes
- 2.7 TB of cache data per month if all 10 billion queries are unique and all are stored
 - 270 bytes per search * 10 billion searches per month
 - Assumptions state limited memory, need to determine how to expire contents
- 4,000 requests per second

Handy conversion guide:

- 2.5 million seconds per month
- 1 request per second = 2.5 million requests per month
- 40 requests per second = 100 million requests per month
- 400 requests per second = 1 billion requests per month

Step 2: Create a high level design

Outline a high level design with all important components.



Imgur

Step 3: Design core components

Dive into details for each core component.

Use case: User sends a request resulting in a cache hit

Popular queries can be served from a **Memory Cache** such as Redis or Memcached to reduce read latency and to avoid overloading the **Reverse Index Service** and **Document Service**. Reading 1 MB sequentially from memory takes about 250 microseconds, while reading from SSD takes 4x and from disk takes 80x longer.¹

Since the cache has limited capacity, we'll use a least recently used (LRU) approach to expire older entries.

- The **Client** sends a request to the **Web Server**, running as a [reverse proxy](#).
- The **Web Server** forwards the request to the **Query API** server
- The **Query API** server does the following:
 - Parses the query
 - Removes markup
 - Breaks up the text into terms
 - Fixes typos
 - Normalizes capitalization
 - Converts the query to use boolean operations
 - Checks the **Memory Cache** for the content matching the query
 - If there's a hit in the **Memory Cache**, the **Memory Cache** does the following:
 - Updates the cached entry's position to the front of the LRU list
 - Returns the cached contents
 - Else, the **Query API** does the following:
 - Uses the **Reverse Index Service** to find documents matching the query
 - The **Reverse Index Service** ranks the matching results and returns the top ones
 - Uses the **Document Service** to return titles and snippets
 - Updates the **Memory Cache** with the contents, placing the entry at the front of the LRU list

Cache implementation

The cache can use a doubly-linked list: new items will be added to the head while items to expire will be removed from the tail. We'll use a hash table for fast lookups to each linked list node.

Clarify with your interviewer how much code you are expected to write.

Query API Server implementation:

```
class QueryApi(object):

    def __init__(self, memory_cache, reverse_index_service):
        self.memory_cache = memory_cache
        self.reverse_index_service = reverse_index_service

    def parse_query(self, query):
        """Remove markup, break text into terms, deal with typos,
        normalize capitalization, convert to use boolean operations.
        """
        ...

    def process_query(self, query):
        query = self.parse_query(query)
        results = self.memory_cache.get(query)
        if results is None:
            results = self.reverse_index_service.process_search(query)
            self.memory_cache.set(query, results)
        return results
```

Node implementation:

```
class Node(object):

    def __init__(self, query, results):
        self.query = query
        self.results = results
```

LinkedList implementation:

```
class LinkedList(object):

    def __init__(self):
        self.head = None
        self.tail = None

    def move_to_front(self, node):
        ...

    def append_to_front(self, node):
        ...

    def remove_from_tail(self):
        ...
```

Cache implementation:

```

class Cache(object):

    def __init__(self, MAX_SIZE):
        self.MAX_SIZE = MAX_SIZE
        self.size = 0
        self.lookup = {} # key: query, value: node
        self.linked_list = LinkedList()

    def get(self, query)
        """Get the stored query result from the cache.

        Accessing a node updates its position to the front of the LRU list.
        """
        node = self.lookup[query]
        if node is None:
            return None
        self.linked_list.move_to_front(node)
        return node.results

    def set(self, results, query):
        """Set the result for the given query key in the cache.

        When updating an entry, updates its position to the front of the LRU
        list.
        If the entry is new and the cache is at capacity, removes the oldest
        entry
        before the new entry is added.
        """
        node = self.lookup[query]
        if node is not None:
            # Key exists in cache, update the value
            node.results = results
            self.linked_list.move_to_front(node)
        else:
            # Key does not exist in cache
            if self.size == self.MAX_SIZE:
                # Remove the oldest entry from the linked list and lookup
                self.lookup.pop(self.linked_list.tail.query, None)
                self.linked_list.remove_from_tail()
            else:
                self.size += 1
            # Add the new key and value
            new_node = Node(query, results)

```

```
self.linked_list.append_to_front(new_node)
self.lookup[query] = new_node
```

When to update the cache

The cache should be updated when:

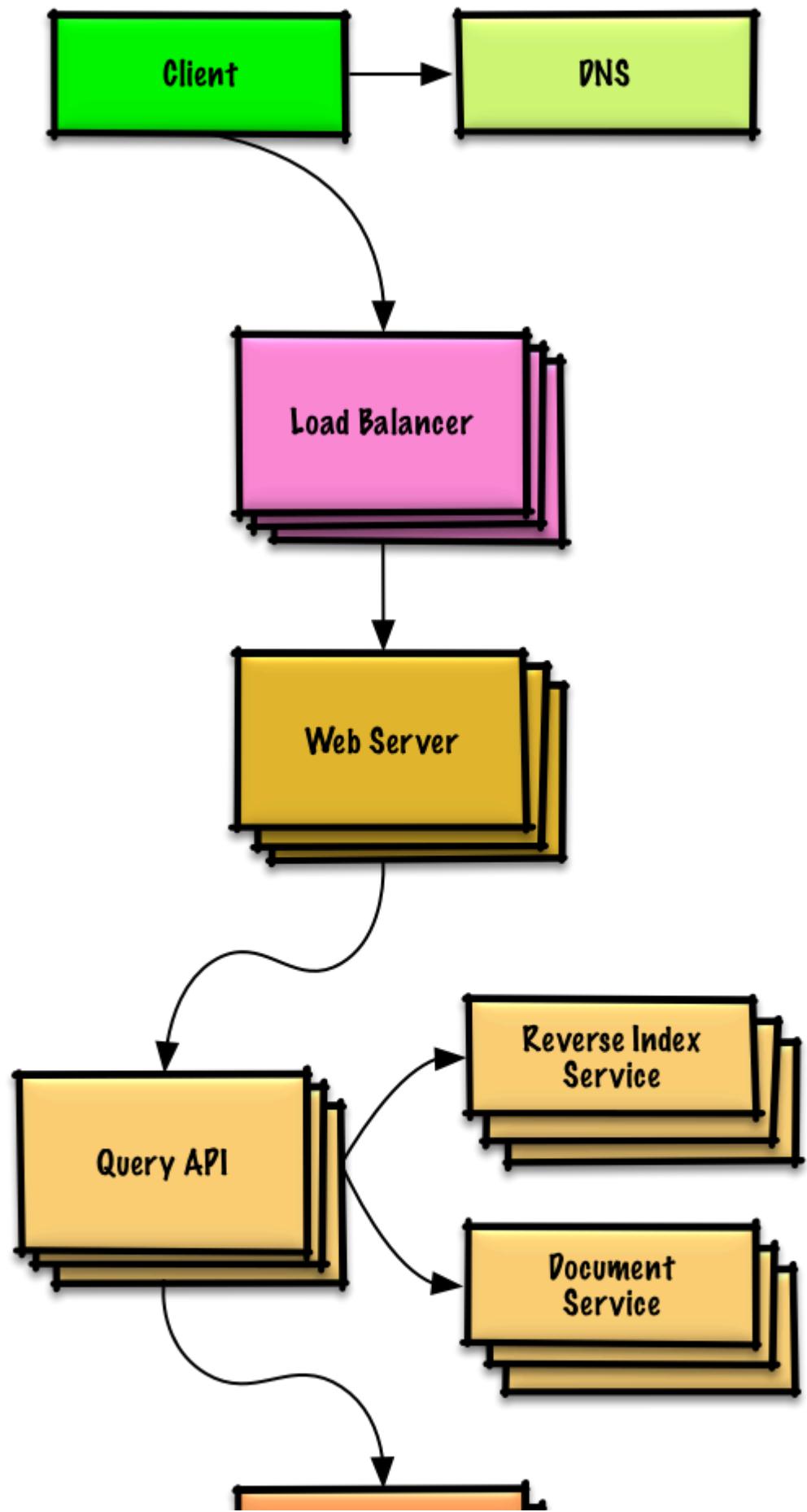
- The page contents change
- The page is removed or a new page is added
- The page rank changes

The most straightforward way to handle these cases is to simply set a max time that a cached entry can stay in the cache before it is updated, usually referred to as time to live (TTL).

Refer to [When to update the cache](#) for tradeoffs and alternatives. The approach above describes [cache-aside](#).

Step 4: Scale the design

Identify and address bottlenecks, given the constraints.





Imgur

Important: Do not simply jump right into the final design from the initial design!

State you would 1) **Benchmark/Load Test**, 2) **Profile** for bottlenecks 3) address bottlenecks while evaluating alternatives and trade-offs, and 4) repeat. See [Design a system that scales to millions of users on AWS](#) as a sample on how to iteratively scale the initial design.

It's important to discuss what bottlenecks you might encounter with the initial design and how you might address each of them. For example, what issues are addressed by adding a **Load Balancer** with multiple **Web Servers?** **CDN?** **Master-Slave Replicas?** What are the alternatives and **Trade-Offs** for each?

We'll introduce some components to complete the design and to address scalability issues. Internal load balancers are not shown to reduce clutter.

To avoid repeating discussions, refer to the following [system design topics](#) for main talking points, tradeoffs, and alternatives:

- [DNS](#)
- [Load balancer](#)
- [Horizontal scaling](#)
- [Web server \(reverse proxy\)](#).
- [API server \(application layer\)](#).
- [Cache](#)
- [Consistency patterns](#)
- [Availability patterns](#)

Expanding the Memory Cache to many machines

To handle the heavy request load and the large amount of memory needed, we'll scale horizontally. We have three main options on how to store the data on our **Memory Cache** cluster:

- **Each machine in the cache cluster has its own cache** - Simple, although it will likely result in a low cache hit rate.
- **Each machine in the cache cluster has a copy of the cache** - Simple, although it is an inefficient use of memory.
- **The cache is sharded across all machines in the cache cluster** - More complex, although it is likely the best option. We could use hashing to determine which machine could have the cached results of a query using `machine = hash(query)`. We'll likely want to use [consistent hashing](#).

Additional talking points

Additional topics to dive into, depending on the problem scope and time remaining.

SQL scaling patterns

- [Read replicas](#)
- [Federation](#)
- [Sharding](#)
- [Denormalization](#)
- [SQL Tuning](#)

NoSQL

- [Key-value store](#)
- [Document store](#)
- [Wide column store](#)
- [Graph database](#)
- [SQL vs NoSQL](#)

Caching

- Where to cache
 - [Client caching](#)
 - [CDN caching](#)
 - [Web server caching](#)
 - [Database caching](#)
 - [Application caching](#)

- What to cache
 - [Caching at the database query level](#)
 - [Caching at the object level](#)
- When to update the cache
 - [Cache-aside](#)
 - [Write-through](#)
 - [Write-behind \(write-back\)](#)
 - [Refresh ahead](#)

Asynchronism and microservices

- [Message queues](#)
- [Task queues](#)
- [Back pressure](#)
- [Microservices](#)

Communications

- Discuss tradeoffs:
 - External communication with clients - [HTTP APIs following REST](#)
 - Internal communications - [RPC](#)
- [Service discovery](#)

Security

Refer to the [security section](#).

Latency numbers

See [Latency numbers every programmer should know](#).

Ongoing

- Continue benchmarking and monitoring your system to address bottlenecks as they come up
- Scaling is an iterative process

System Design: Sales Rank

Design Amazon's sales rank by category feature

Note: This document links directly to relevant areas found in the [system design topics](#) to avoid duplication. Refer to the linked content for general talking points, tradeoffs, and alternatives.

Step 1: Outline use cases and constraints

Gather requirements and scope the problem. Ask questions to clarify use cases and constraints. Discuss assumptions.

Without an interviewer to address clarifying questions, we'll define some use cases and constraints.

Use cases

We'll scope the problem to handle only the following use case

- **Service** calculates the past week's most popular products by category
- **User** views the past week's most popular products by category
- **Service** has high availability

Out of scope

- The general e-commerce site
 - Design components only for calculating sales rank

Constraints and assumptions

State assumptions

- Traffic is not evenly distributed
- Items can be in multiple categories
- Items cannot change categories

- There are no subcategories ie foo/bar/baz
- Results must be updated hourly
 - More popular products might need to be updated more frequently
- 10 million products
- 1000 categories
- 1 billion transactions per month
- 100 billion read requests per month
- 100:1 read to write ratio

Calculate usage

Clarify with your interviewer if you should run back-of-the-envelope usage calculations.

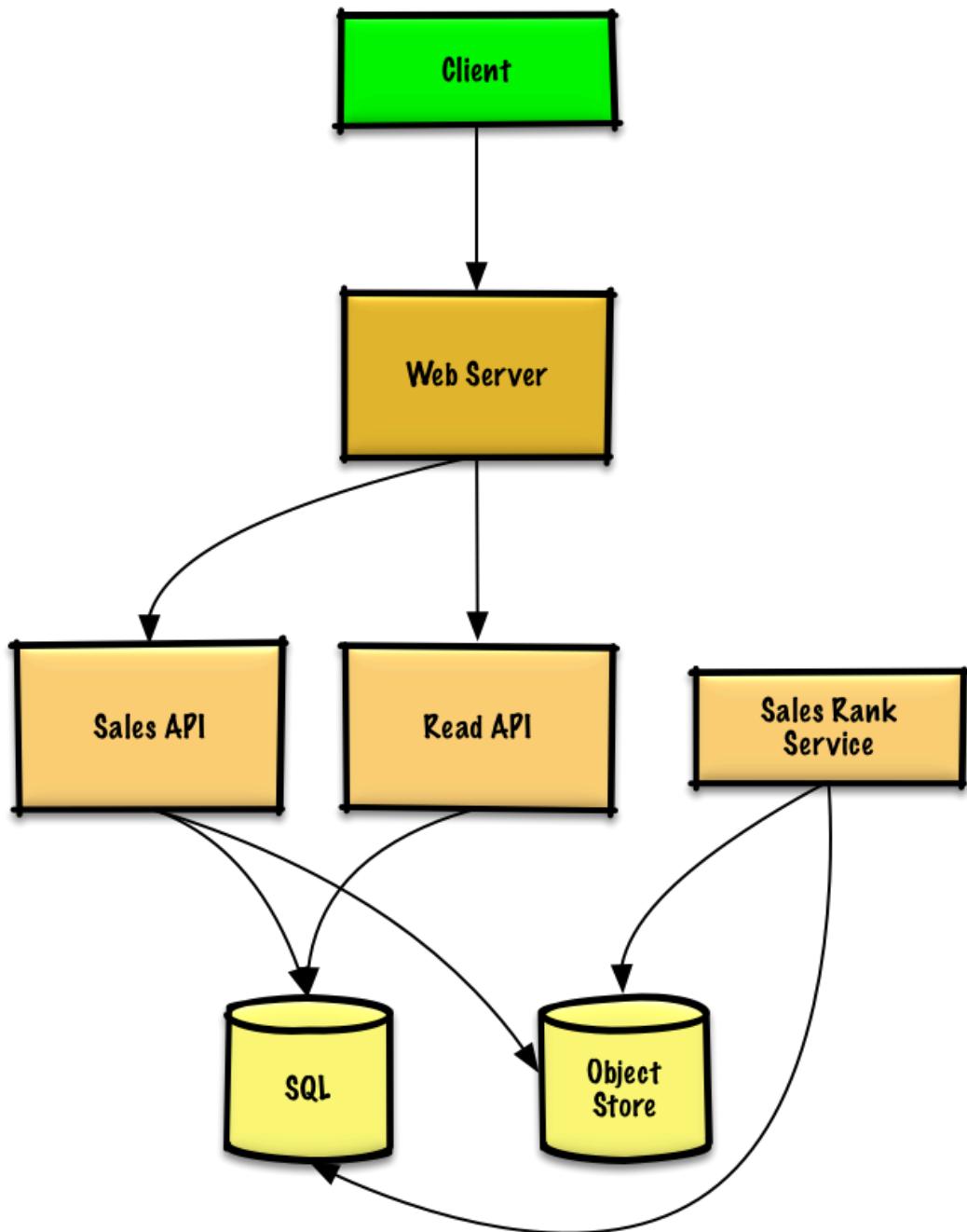
- Size per transaction:
 - created_at - 5 bytes
 - product_id - 8 bytes
 - category_id - 4 bytes
 - seller_id - 8 bytes
 - buyer_id - 8 bytes
 - quantity - 4 bytes
 - total_price - 5 bytes
 - Total: ~40 bytes
- 40 GB of new transaction content per month
 - 40 bytes per transaction * 1 billion transactions per month
 - 1.44 TB of new transaction content in 3 years
 - Assume most are new transactions instead of updates to existing ones
- 400 transactions per second on average
- 40,000 read requests per second on average

Handy conversion guide:

- 2.5 million seconds per month
- 1 request per second = 2.5 million requests per month
- 40 requests per second = 100 million requests per month
- 400 requests per second = 1 billion requests per month

Step 2: Create a high level design

Outline a high level design with all important components.



Imgur

Step 3: Design core components

Dive into details for each core component.

Use case: Service calculates the past week's most popular products by category

We could store the raw **Sales API** server log files on a managed **Object Store** such as Amazon S3, rather than managing our own distributed file system.

Clarify with your interviewer how much code you are expected to write.

We'll assume this is a sample log entry, tab delimited:

timestamp	product_id	category_id	qty	total_price	seller_id
buyer_id					
t1	product1	category1	2	20.00	1
t2	product1	category2	2	20.00	2
t2	product1	category2	1	10.00	2
t3	product2	category1	3	7.00	3
t4	product3	category2	7	2.00	4
t5	product4	category1	1	5.00	5
...					

The **Sales Rank Service** could use **MapReduce**, using the **Sales API** server log files as input and writing the results to an aggregate table `sales_rank` in a **SQL Database**. We should discuss the [use cases and tradeoffs between choosing SQL or NoSQL](#).

We'll use a multi-step **MapReduce**:

- **Step 1** - Transform the data to (category, product_id), sum(quantity)
- **Step 2** - Perform a distributed sort

```

class SalesRanker(MRJob):

    def within_past_week(self, timestamp):
        """Return True if timestamp is within past week, False otherwise."""
        ...

    def mapper(self, _ line):
        """Parse each log line, extract and transform relevant lines.

        Emit key value pairs of the form:

        (category1, product1), 2
        (category2, product1), 2
        (category2, product1), 1
        (category1, product2), 3
        (category2, product3), 7
        (category1, product4), 1
        """
        timestamp, product_id, category_id, quantity, total_price, seller_id,
        \
        buyer_id = line.split('\t')
        if self.within_past_week(timestamp):
            yield (category_id, product_id), quantity

    def reducer(self, key, value):
        """Sum values for each key.

        (category1, product1), 2
        (category2, product1), 3
        (category1, product2), 3
        (category2, product3), 7
        (category1, product4), 1
        """
        yield key, sum(values)

    def mapper_sort(self, key, value):
        """Construct key to ensure proper sorting.

        Transform key and value to the form:

        (category1, 2), product1
        (category2, 3), product1
        (category1, 3), product2
        (category2, 7), product3
        (category1, 1), product4
        """

```

The shuffle/sort step of MapReduce will then do a distributed sort on the keys, resulting in:

```
(category1, 1), product4
(category1, 2), product1
(category1, 3), product2
(category2, 3), product1
(category2, 7), product3
"""
category_id, product_id = key
quantity = value
yield (category_id, quantity), product_id

def reducer_identity(self, key, value):
    yield key, value

def steps(self):
    """Run the map and reduce steps."""
    return [
        self.mr(mapper=self.mapper,
                reducer=self.reducer),
        self.mr(mapper=self.mapper_sort,
                reducer=self.reducer_identity),
    ]
```

The result would be the following sorted list, which we could insert into the sales_rank table:

```
(category1, 1), product4
(category1, 2), product1
(category1, 3), product2
(category2, 3), product1
(category2, 7), product3
```

The sales_rank table could have the following structure:

```
id int NOT NULL AUTO_INCREMENT
category_id int NOT NULL
total_sold int NOT NULL
product_id int NOT NULL
PRIMARY KEY(id)
FOREIGN KEY(category_id) REFERENCES Categories(id)
FOREIGN KEY(product_id) REFERENCES Products(id)
```

We'll create an [index](#) on id, category_id, and product_id to speed up lookups (log-time instead of scanning the entire table) and to keep the data in memory. Reading 1 MB sequentially from memory takes about 250 microseconds, while reading from SSD takes 4x and from disk takes 80x longer.¹

Use case: User views the past week's most popular products by category

- The **Client** sends a request to the **Web Server**, running as a [reverse proxy](#).
- The **Web Server** forwards the request to the **Read API** server
- The **Read API** server reads from the **SQL Database** sales_rank table

We'll use a public [REST API](#):

```
$ curl https://amazon.com/api/v1/popular?category_id=1234
```

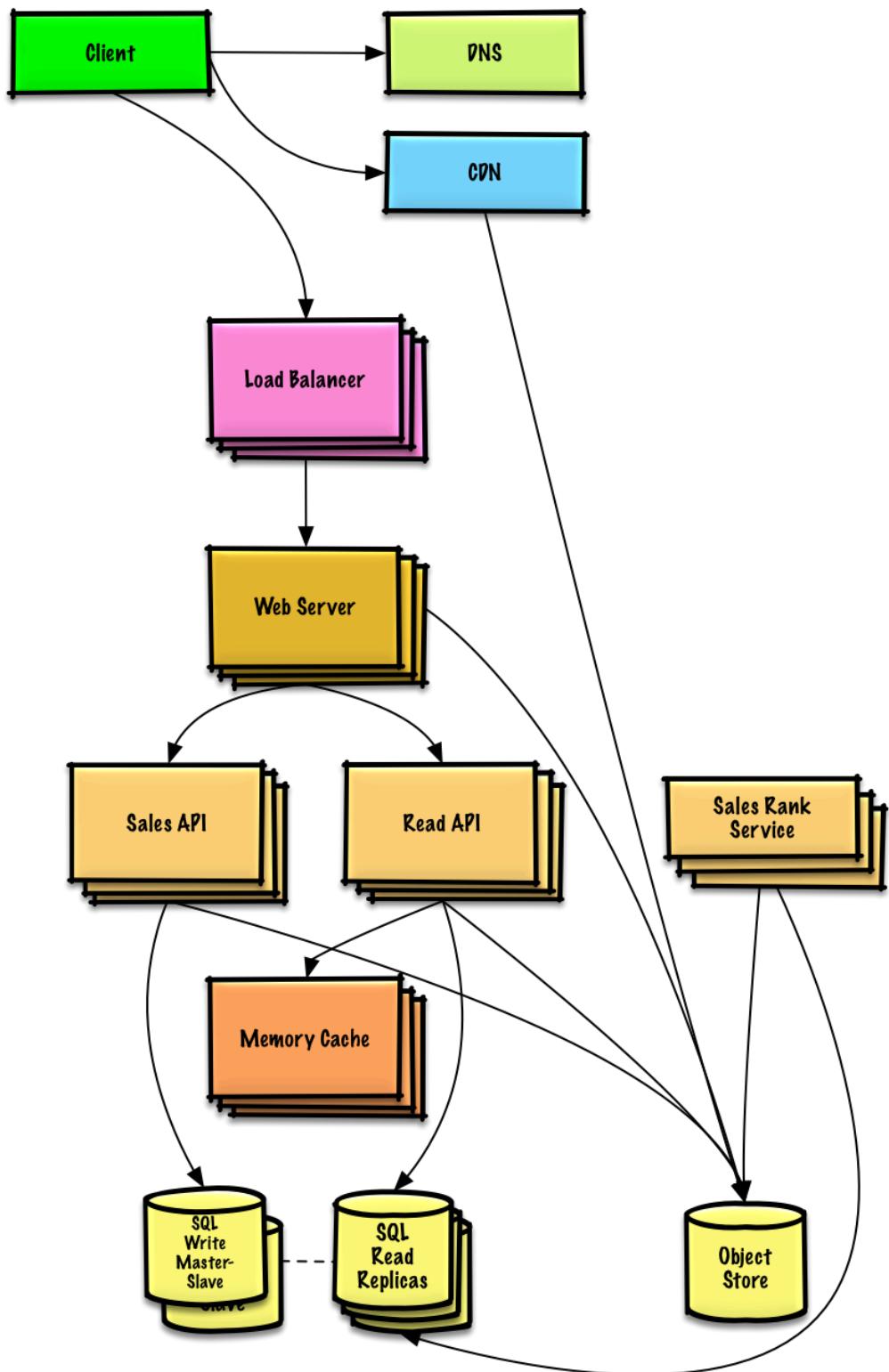
Response:

```
{  
    "id": "100",  
    "category_id": "1234",  
    "total_sold": "100000",  
    "product_id": "50",  
,  
{  
    "id": "53",  
    "category_id": "1234",  
    "total_sold": "90000",  
    "product_id": "200",  
,  
{  
    "id": "75",  
    "category_id": "1234",  
    "total_sold": "80000",  
    "product_id": "3",  
,
```

For internal communications, we could use [Remote Procedure Calls](#).

Step 4: Scale the design

Identify and address bottlenecks, given the constraints.



Imgur

Important: Do not simply jump right into the final design from the initial design!

State you would 1) **Benchmark/Load Test**, 2) **Profile** for bottlenecks 3) address bottlenecks while evaluating alternatives and trade-offs, and 4) repeat. See [Design a system that scales to millions of users on AWS](#) as a sample on how to iteratively scale the initial design.

It's important to discuss what bottlenecks you might encounter with the initial design and how you might address each of them. For example, what issues are addressed by adding a **Load Balancer** with multiple **Web Servers?** **CDN?** **Master-Slave Replicas?** What are the alternatives and **Trade-Offs** for each?

We'll introduce some components to complete the design and to address scalability issues. Internal load balancers are not shown to reduce clutter.

To avoid repeating discussions, refer to the following [system design topics](#) for main talking points, tradeoffs, and alternatives:

- [DNS](#)
- [CDN](#)
- [Load balancer](#)
- [Horizontal scaling](#)
- [Web server \(reverse proxy\)](#).
- [API server \(application layer\)](#).
- [Cache](#)
- [Relational database management system \(RDBMS\)](#).
- [SQL write master-slave failover](#)
- [Master-slave replication](#)
- [Consistency patterns](#)
- [Availability patterns](#)

The **Analytics Database** could use a data warehousing solution such as Amazon Redshift or Google BigQuery.

We might only want to store a limited time period of data in the database, while storing the rest in a data warehouse or in an **Object Store**. An **Object Store** such as Amazon S3 can comfortably handle the constraint of 40 GB of new content per month.

To address the 40,000 average read requests per second (higher at peak), traffic for popular content (and their sales rank) should be handled by the **Memory Cache** instead of the database. The **Memory Cache** is also useful for handling the unevenly distributed

traffic and traffic spikes. With the large volume of reads, the **SQL Read Replicas** might not be able to handle the cache misses. We'll probably need to employ additional SQL scaling patterns.

400 average writes per second (higher at peak) might be tough for a single **SQL Write Master-Slave**, also pointing to a need for additional scaling techniques.

SQL scaling patterns include:

- [Federation](#)
- [Sharding](#)
- [Denormalization](#)
- [SQL Tuning](#)

We should also consider moving some data to a **NoSQL Database**.

Additional talking points

Additional topics to dive into, depending on the problem scope and time remaining.

NoSQL

- [Key-value store](#)
- [Document store](#)
- [Wide column store](#)
- [Graph database](#)
- [SQL vs NoSQL](#)

Caching

- Where to cache
 - [Client caching](#)
 - [CDN caching](#)
 - [Web server caching](#)
 - [Database caching](#)
 - [Application caching](#)
- What to cache
 - [Caching at the database query level](#)

- [Caching at the object level](#)
- When to update the cache
 - [Cache-aside](#)
 - [Write-through](#)
 - [Write-behind \(write-back\)](#).
 - [Refresh ahead](#)

Asynchronism and microservices

- [Message queues](#)
- [Task queues](#)
- [Back pressure](#)
- [Microservices](#)

Communications

- Discuss tradeoffs:
 - External communication with clients - [HTTP APIs following REST](#)
 - Internal communications - [RPC](#)
- [Service discovery](#).

Security

Refer to the [security section](#).

Latency numbers

See [Latency numbers every programmer should know](#).

Ongoing

- Continue benchmarking and monitoring your system to address bottlenecks as they come up
- Scaling is an iterative process

System Design: Scaling Aws

Design a system that scales to millions of users on AWS

Note: This document links directly to relevant areas found in the [system design topics](#) to avoid duplication. Refer to the linked content for general talking points, tradeoffs, and alternatives.

Step 1: Outline use cases and constraints

Gather requirements and scope the problem. Ask questions to clarify use cases and constraints. Discuss assumptions.

Without an interviewer to address clarifying questions, we'll define some use cases and constraints.

Use cases

Solving this problem takes an iterative approach of: 1) **Benchmark/Load Test**, 2) **Profile** for bottlenecks 3) address bottlenecks while evaluating alternatives and trade-offs, and 4) repeat, which is good pattern for evolving basic designs to scalable designs.

Unless you have a background in AWS or are applying for a position that requires AWS knowledge, AWS-specific details are not a requirement. However, **much of the principles discussed in this exercise can apply more generally outside of the AWS ecosystem.**

We'll scope the problem to handle only the following use cases

- **User** makes a read or write request
 - **Service** does processing, stores user data, then returns the results
- **Service** needs to evolve from serving a small amount of users to millions of users
 - Discuss general scaling patterns as we evolve an architecture to handle a large number of users and requests
- **Service** has high availability

Constraints and assumptions

State assumptions

- Traffic is not evenly distributed
- Need for relational data
- Scale from 1 user to tens of millions of users
 - Denote increase of users as:
 - Users+
 - Users++
 - Users+++
 - ...
 - 10 million users
 - 1 billion writes per month
 - 100 billion reads per month
 - 100:1 read to write ratio
 - 1 KB content per write

Calculate usage

Clarify with your interviewer if you should run back-of-the-envelope usage calculations.

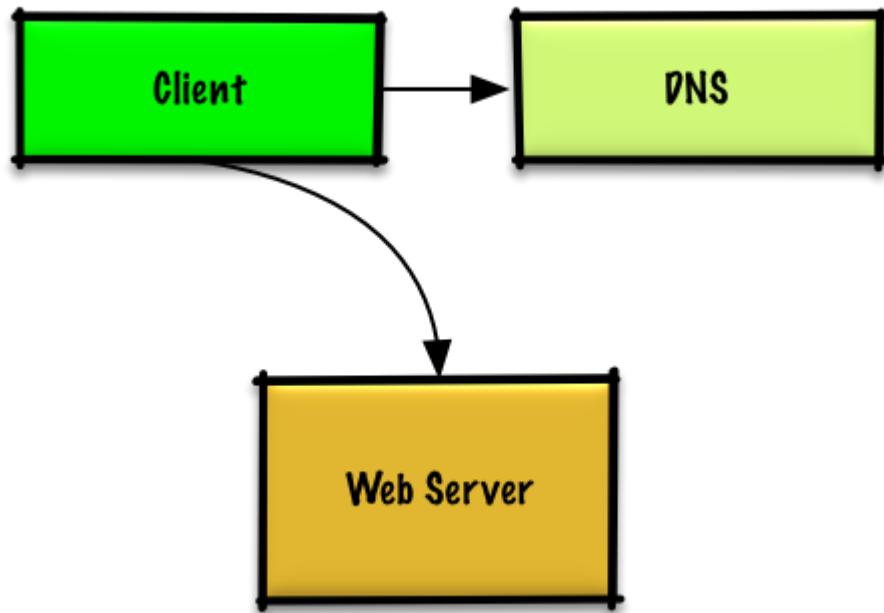
- 1 TB of new content per month
 - 1 KB per write * 1 billion writes per month
 - 36 TB of new content in 3 years
 - Assume most writes are from new content instead of updates to existing ones
- 400 writes per second on average
- 40,000 reads per second on average

Handy conversion guide:

- 2.5 million seconds per month
- 1 request per second = 2.5 million requests per month
- 40 requests per second = 100 million requests per month
- 400 requests per second = 1 billion requests per month

Step 2: Create a high level design

Outline a high level design with all important components.



Imgur

Step 3: Design core components

Dive into details for each core component.

Use case: User makes a read or write request

Goals

- With only 1-2 users, you only need a basic setup
 - Single box for simplicity
 - Vertical scaling when needed
 - Monitor to determine bottlenecks

Start with a single box

- **Web server** on EC2
 - Storage for user data
 - [MySQL Database](#)

Use Vertical Scaling:

- Simply choose a bigger box
- Keep an eye on metrics to determine how to scale up
 - Use basic monitoring to determine bottlenecks: CPU, memory, IO, network, etc
 - CloudWatch, top, nagios, statsd, graphite, etc
- Scaling vertically can get very expensive
- No redundancy/failover

Trade-offs, alternatives, and additional details:

- The alternative to **Vertical Scaling** is [Horizontal scaling](#)

Start with SQL, consider NoSQL

The constraints assume there is a need for relational data. We can start off using a **MySQL Database** on the single box.

Trade-offs, alternatives, and additional details:

- See the [Relational database management system \(RDBMS\)](#) section
- Discuss reasons to use [SQL or NoSQL](#)

Assign a public static IP

- Elastic IPs provide a public endpoint whose IP doesn't change on reboot
- Helps with failover, just point the domain to a new IP

Use a DNS

Add a **DNS** such as Route 53 to map the domain to the instance's public IP.

Trade-offs, alternatives, and additional details:

- See the [Domain name system](#) section

Secure the web server

- Open up only necessary ports
 - Allow the web server to respond to incoming requests from:
 - 80 for HTTP

- 443 for HTTPS
- 22 for SSH to only whitelisted IPs
- Prevent the web server from initiating outbound connections

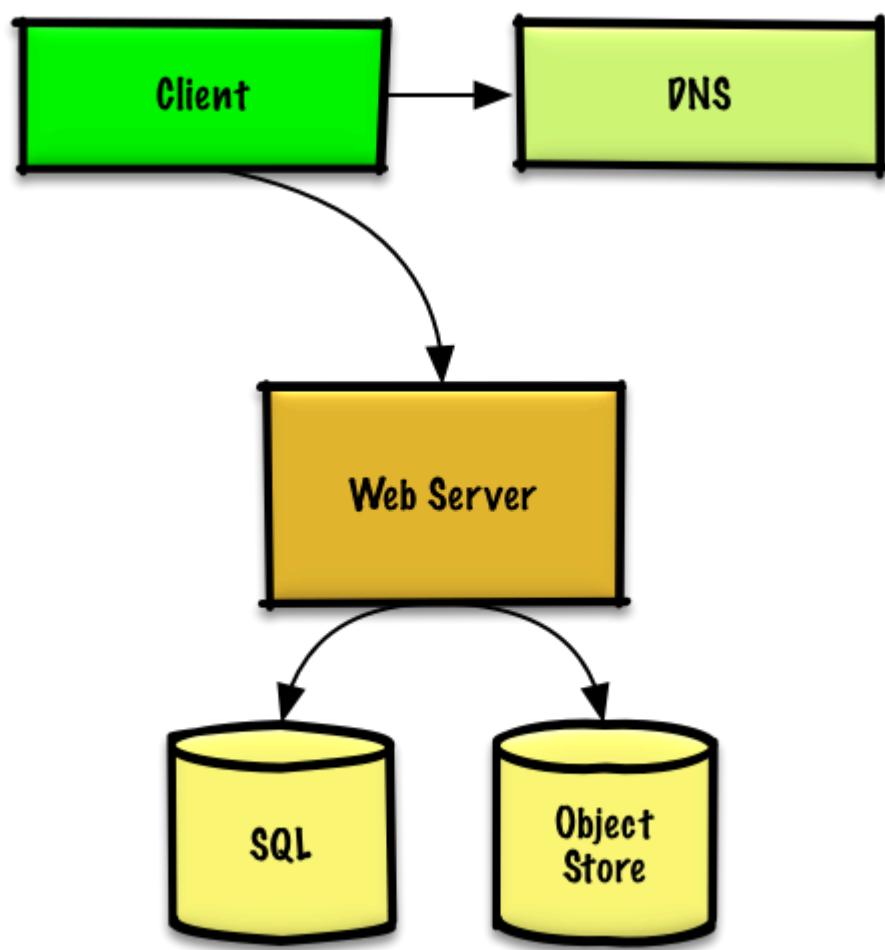
Trade-offs, alternatives, and additional details:

- See the [Security](#) section

Step 4: Scale the design

Identify and address bottlenecks, given the constraints.

Users+



Imgur

Assumptions

Our user count is starting to pick up and the load is increasing on our single box. Our **Benchmarks/Load Tests** and **Profiling** are pointing to the **MySQL Database** taking up more and more memory and CPU resources, while the user content is filling up disk space.

We've been able to address these issues with **Vertical Scaling** so far. Unfortunately, this has become quite expensive and it doesn't allow for independent scaling of the **MySQL Database** and **Web Server**.

Goals

- Lighten load on the single box and allow for independent scaling
 - Store static content separately in an **Object Store**
 - Move the **MySQL Database** to a separate box
- Disadvantages
 - These changes would increase complexity and would require changes to the **Web Server** to point to the **Object Store** and the **MySQL Database**
 - Additional security measures must be taken to secure the new components
 - AWS costs could also increase, but should be weighed with the costs of managing similar systems on your own

Store static content separately

- Consider using a managed **Object Store** like S3 to store static content
 - Highly scalable and reliable
 - Server side encryption
- Move static content to S3
 - User files
 - JS
 - CSS
 - Images
 - Videos

Move the MySQL database to a separate box

- Consider using a service like RDS to manage the **MySQL Database**
 - Simple to administer, scale
 - Multiple availability zones

- Encryption at rest

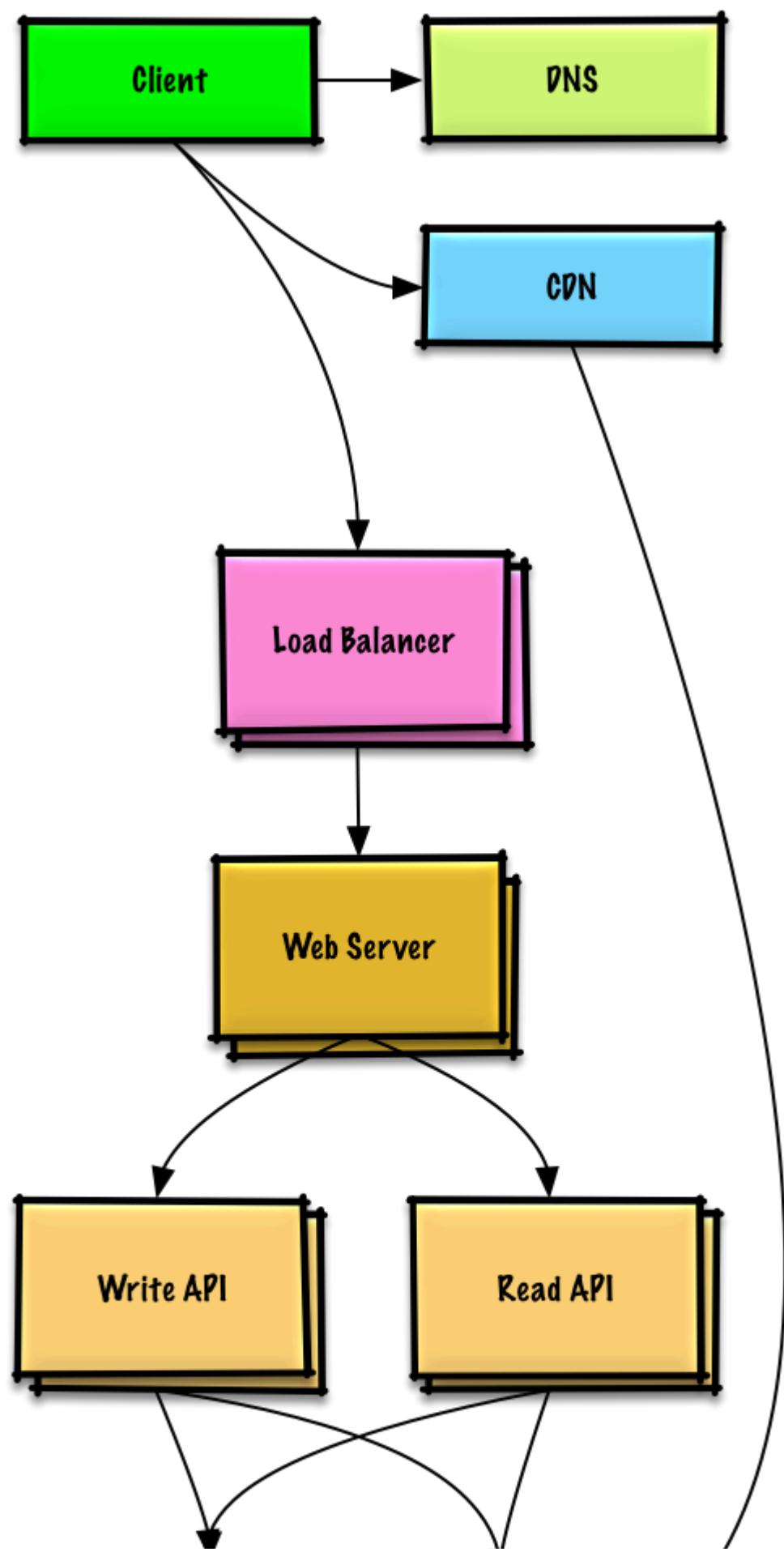
Secure the system

- Encrypt data in transit and at rest
- Use a Virtual Private Cloud
 - Create a public subnet for the single **Web Server** so it can send and receive traffic from the internet
 - Create a private subnet for everything else, preventing outside access
 - Only open ports from whitelisted IPs for each component
- These same patterns should be implemented for new components in the remainder of the exercise

Trade-offs, alternatives, and additional details:

- See the [Security](#) section

Users++





Imgur

Assumptions

Our **Benchmarks/Load Tests** and **Profiling** show that our single **Web Server** bottlenecks during peak hours, resulting in slow responses and in some cases, downtime. As the service matures, we'd also like to move towards higher availability and redundancy.

Goals

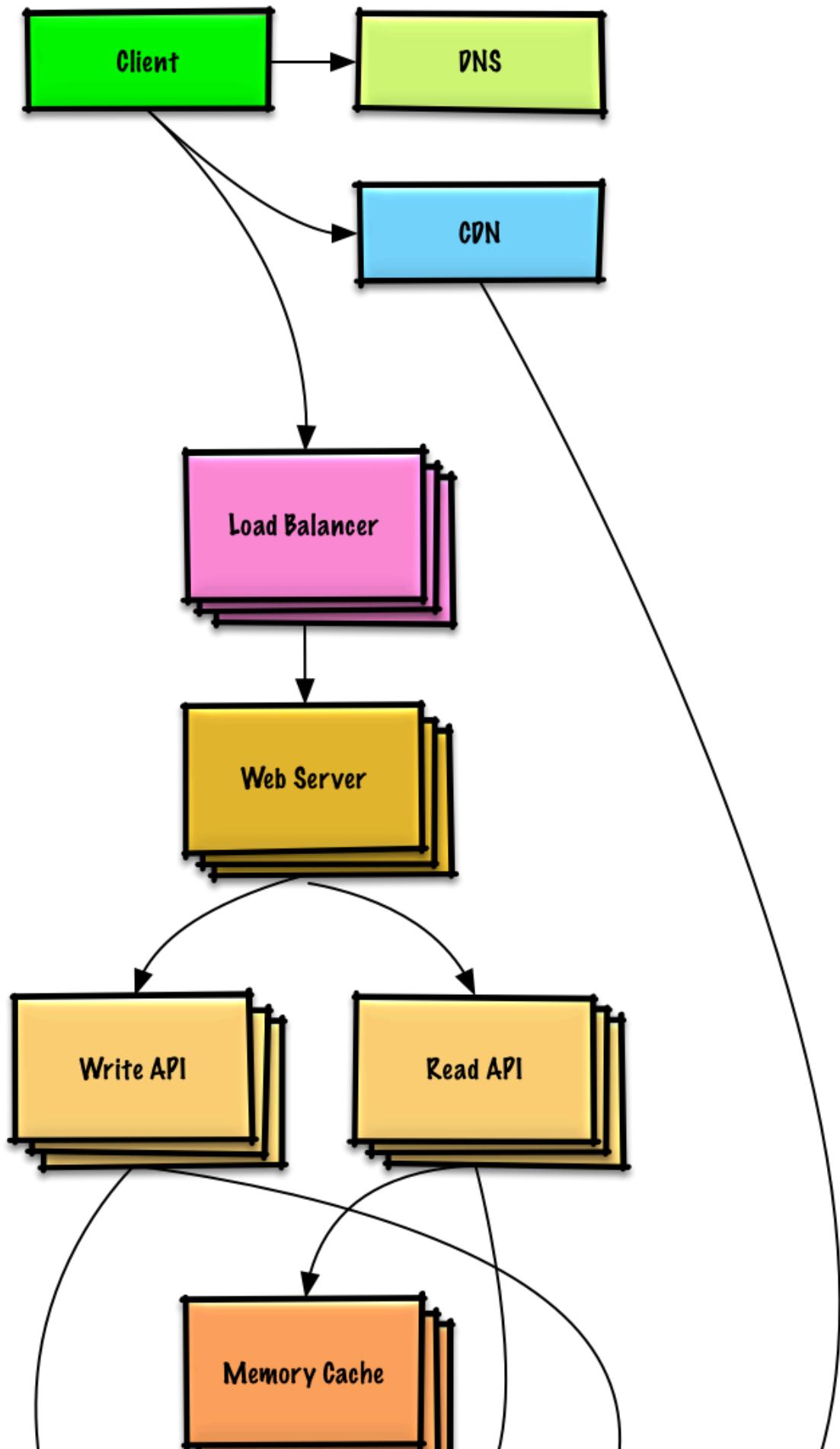
- The following goals attempt to address the scaling issues with the **Web Server**
 - Based on the **Benchmarks/Load Tests** and **Profiling**, you might only need to implement one or two of these techniques
- Use **Horizontal Scaling** to handle increasing loads and to address single points of failure
 - Add a **Load Balancer** such as Amazon's ELB or HAProxy
 - ELB is highly available
 - If you are configuring your own **Load Balancer**, setting up multiple servers in [active-active](#) or [active-passive](#) in multiple availability zones will improve availability
 - Terminate SSL on the **Load Balancer** to reduce computational load on backend servers and to simplify certificate administration
 - Use multiple **Web Servers** spread out over multiple availability zones
 - Use multiple **MySQL** instances in [Master-Slave Failover](#) mode across multiple availability zones to improve redundancy
- Separate out the **Web Servers** from the [Application Servers](#)
 - Scale and configure both layers independently
 - **Web Servers** can run as a [Reverse Proxy](#)
 - For example, you can add **Application Servers** handling **Read APIs** while others handle **Write APIs**

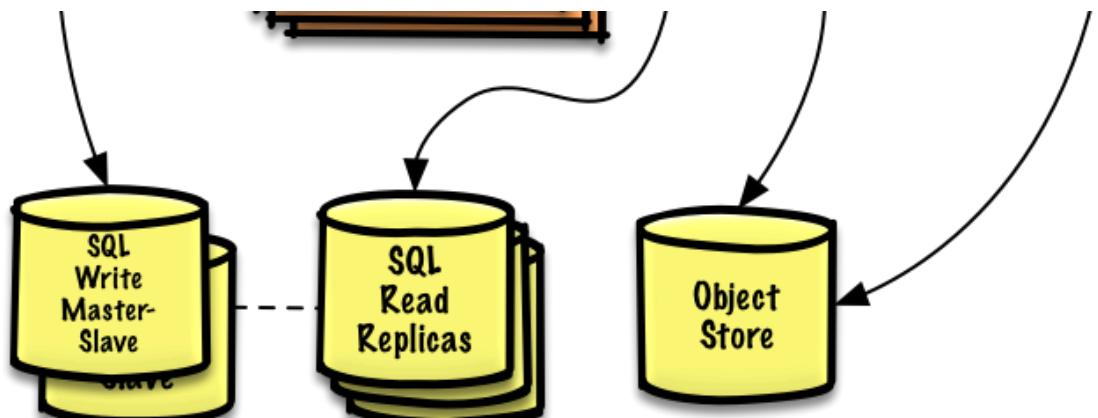
- Move static (and some dynamic) content to a [**Content Delivery Network \(CDN\)**](#) such as CloudFront to reduce load and latency

Trade-offs, alternatives, and additional details:

- See the linked content above for details

Users+++





Imgur

Note: Internal Load Balancers not shown to reduce clutter

Assumptions

Our **Benchmarks/Load Tests** and **Profiling** show that we are read-heavy (100:1 with writes) and our database is suffering from poor performance from the high read requests.

Goals

- The following goals attempt to address the scaling issues with the **MySQL Database**
 - Based on the **Benchmarks/Load Tests** and **Profiling**, you might only need to implement one or two of these techniques
- Move the following data to a **Memory Cache** such as ElastiCache to reduce load and latency:
 - Frequently accessed content from **MySQL**
 - First, try to configure the **MySQL Database** cache to see if that is sufficient to relieve the bottleneck before implementing a **Memory Cache**
 - Session data from the **Web Servers**
 - The **Web Servers** become stateless, allowing for **Autoscaling**
 - Reading 1 MB sequentially from memory takes about 250 microseconds, while reading from SSD takes 4x and from disk takes 80x longer.¹
- Add **MySQL Read Replicas** to reduce load on the write master
- Add more **Web Servers** and **Application Servers** to improve responsiveness

Trade-offs, alternatives, and additional details:

- See the linked content above for details

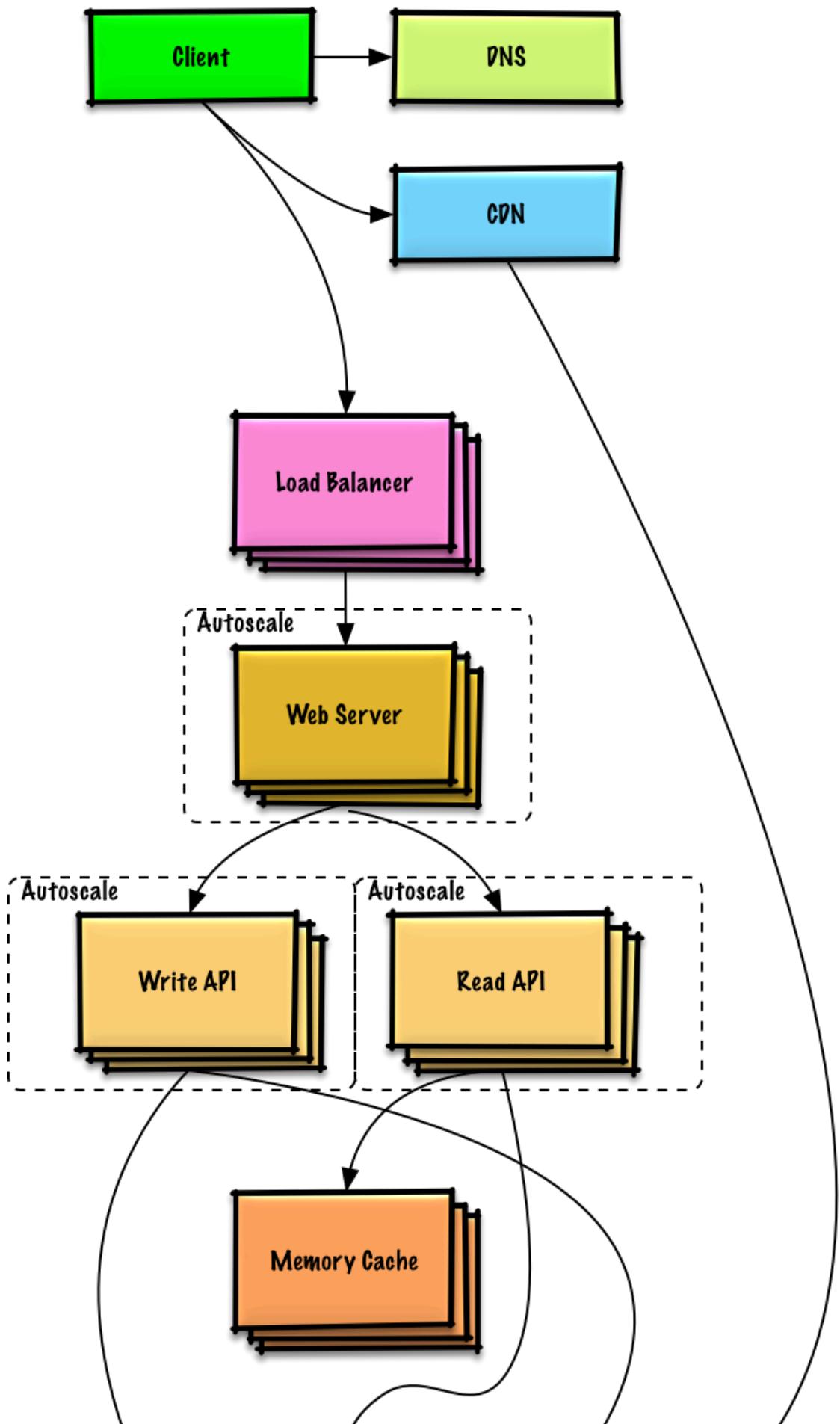
Add MySQL read replicas

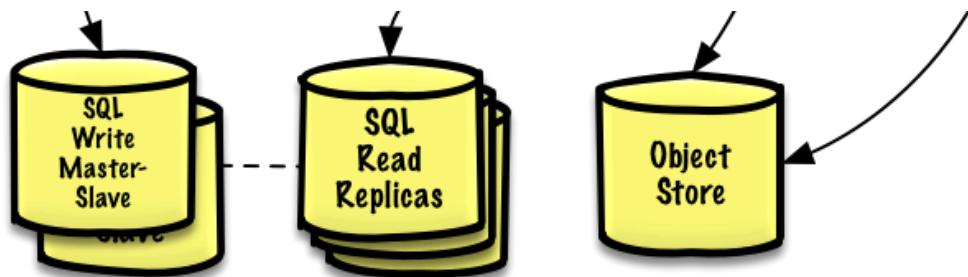
- In addition to adding and scaling a **Memory Cache**, **MySQL Read Replicas** can also help relieve load on the **MySQL Write Master**
- Add logic to **Web Server** to separate out writes and reads
- Add **Load Balancers** in front of **MySQL Read Replicas** (not pictured to reduce clutter)
- Most services are read-heavy vs write-heavy

Trade-offs, alternatives, and additional details:

- See the [Relational database management system \(RDBMS\)](#) section

Users++++





Imgur

Assumptions

Our **Benchmarks/Load Tests** and **Profiling** show that our traffic spikes during regular business hours in the U.S. and drop significantly when users leave the office. We think we can cut costs by automatically spinning up and down servers based on actual load. We're a small shop so we'd like to automate as much of the DevOps as possible for **Autoscaling** and for the general operations.

Goals

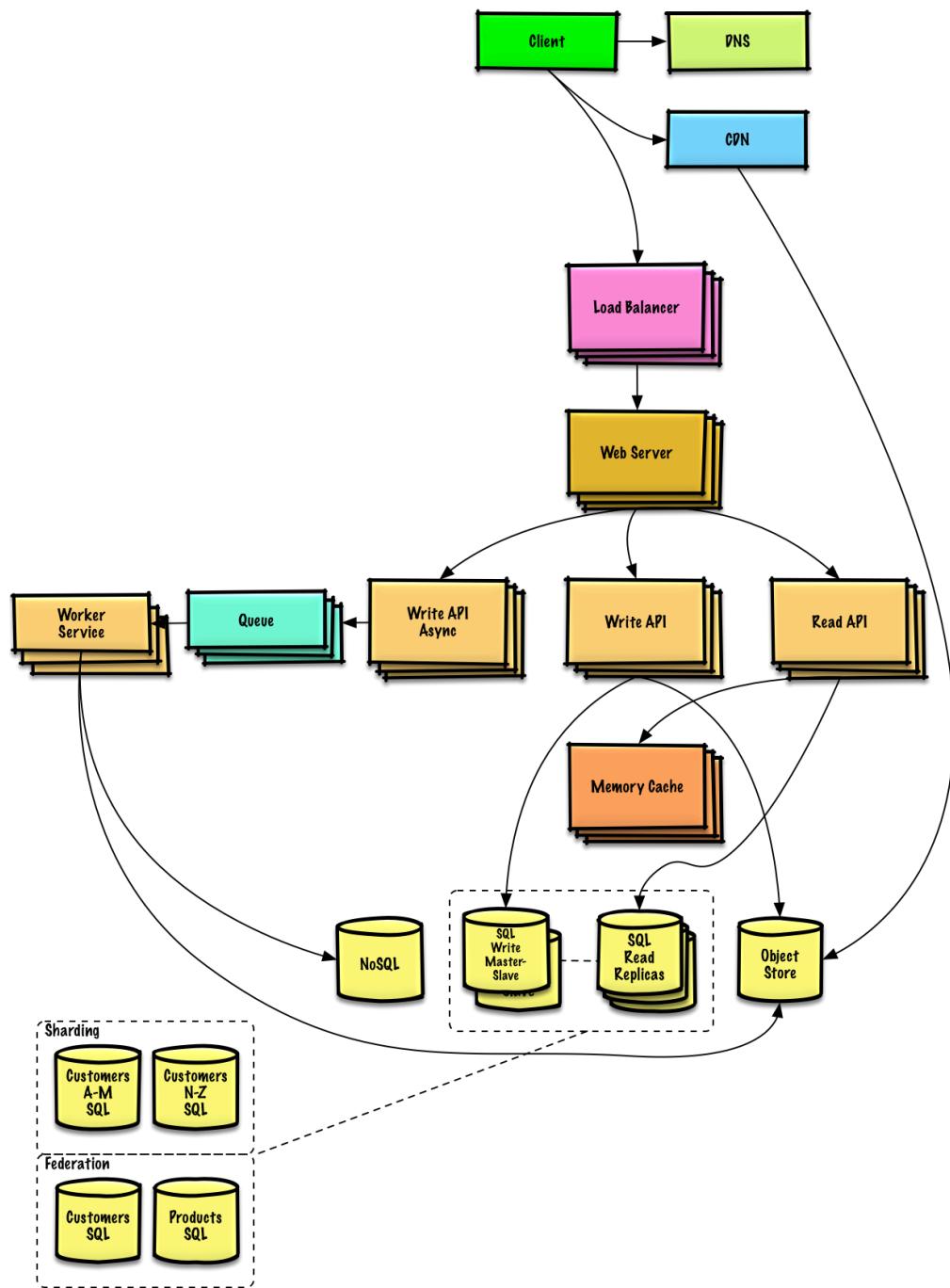
- Add **Autoscaling** to provision capacity as needed
 - Keep up with traffic spikes
 - Reduce costs by powering down unused instances
- Automate DevOps
 - Chef, Puppet, Ansible, etc
- Continue monitoring metrics to address bottlenecks
 - **Host level** - Review a single EC2 instance
 - **Aggregate level** - Review load balancer stats
 - **Log analysis** - CloudWatch, CloudTrail, Loggly, Splunk, Sumo
 - **External site performance** - Pingdom or New Relic
 - **Handle notifications and incidents** - PagerDuty
 - **Error Reporting** - Sentry

Add autoscaling

- Consider a managed service such as AWS **Autoscaling**
 - Create one group for each **Web Server** and one for each **Application Server** type, place each group in multiple availability zones
 - Set a min and max number of instances
 - Trigger to scale up and down through CloudWatch
 - Simple time of day metric for predictable loads or

- Metrics over a time period:
 - CPU load
 - Latency
 - Network traffic
 - Custom metric
- Disadvantages
 - Autoscaling can introduce complexity
 - It could take some time before a system appropriately scales up to meet increased demand, or to scale down when demand drops

Users+++++



Imgur

Note: Autoscaling groups not shown to reduce clutter

Assumptions

As the service continues to grow towards the figures outlined in the constraints, we iteratively run **Benchmarks/Load Tests** and **Profiling** to uncover and address new bottlenecks.

Goals

We'll continue to address scaling issues due to the problem's constraints:

- If our **MySQL Database** starts to grow too large, we might consider only storing a limited time period of data in the database, while storing the rest in a data warehouse such as Redshift
 - A data warehouse such as Redshift can comfortably handle the constraint of 1 TB of new content per month
- With 40,000 average read requests per second, read traffic for popular content can be addressed by scaling the **Memory Cache**, which is also useful for handling the unevenly distributed traffic and traffic spikes
 - The **SQL Read Replicas** might have trouble handling the cache misses, we'll probably need to employ additional SQL scaling patterns
- 400 average writes per second (with presumably significantly higher peaks) might be tough for a single **SQL Write Master-Slave**, also pointing to a need for additional scaling techniques

SQL scaling patterns include:

- [Federation](#)
- [Sharding](#)
- [Denormalization](#)
- [SQL Tuning](#)

To further address the high read and write requests, we should also consider moving appropriate data to a [NoSQL Database](#) such as DynamoDB.

We can further separate out our [Application Servers](#) to allow for independent scaling. Batch processes or computations that do not need to be done in real-time can be done [Asynchronously](#) with **Queues** and **Workers**:

- For example, in a photo service, the photo upload and the thumbnail creation can be separated:
 - **Client** uploads photo
 - **Application Server** puts a job in a **Queue** such as SQS
 - The **Worker Service** on EC2 or Lambda pulls work off the **Queue** then:
 - Creates a thumbnail
 - Updates a **Database**

- Stores the thumbnail in the **Object Store**

Trade-offs, alternatives, and additional details:

- See the linked content above for details

Additional talking points

Additional topics to dive into, depending on the problem scope and time remaining.

SQL scaling patterns

- [Read replicas](#)
- [Federation](#)
- [Sharding](#)
- [Denormalization](#)
- [SQL Tuning](#)

NoSQL

- [Key-value store](#)
- [Document store](#)
- [Wide column store](#)
- [Graph database](#)
- [SQL vs NoSQL](#)

Caching

- Where to cache
 - [Client caching](#)
 - [CDN caching](#)
 - [Web server caching](#)
 - [Database caching](#)
 - [Application caching](#)
- What to cache
 - [Caching at the database query level](#)
 - [Caching at the object level](#)
- When to update the cache

- [Cache-aside](#)
- [Write-through](#)
- [Write-behind \(write-back\)](#)
- [Refresh ahead](#)

Asynchronism and microservices

- [Message queues](#)
- [Task queues](#)
- [Back pressure](#)
- [Microservices](#)

Communications

- Discuss tradeoffs:
 - External communication with clients - [HTTP APIs following REST](#)
 - Internal communications - [RPC](#)
- [Service discovery](#)

Security

Refer to the [security section](#).

Latency numbers

See [Latency numbers every programmer should know](#).

Ongoing

- Continue benchmarking and monitoring your system to address bottlenecks as they come up
- Scaling is an iterative process

Object Oriented Design: Hash Table

This notebook was prepared by [Donne Martin](#). Source and license info is on [GitHub](#).

Design a hash map

Constraints and assumptions

- For simplicity, are the keys integers only?
 - Yes
- For collision resolution, can we use chaining?
 - Yes
- Do we have to worry about load factors?
 - No
- Can we assume inputs are valid or do we have to validate them?
 - Assume they're valid
- Can we assume this fits memory?
 - Yes

Solution

```
%%writefile hash_map.py
class Item(object):

    def __init__(self, key, value):
        self.key = key
        self.value = value


class HashTable(object):

    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(self.size)]

    def _hash_function(self, key):
        return key % self.size

    def set(self, key, value):
        hash_index = self._hash_function(key)
        for item in self.table[hash_index]:
            if item.key == key:
                item.value = value
                return
        self.table[hash_index].append(Item(key, value))

    def get(self, key):
        hash_index = self._hash_function(key)
        for item in self.table[hash_index]:
            if item.key == key:
                return item.value
        raise KeyError('Key not found')

    def remove(self, key):
        hash_index = self._hash_function(key)
        for index, item in enumerate(self.table[hash_index]):
            if item.key == key:
                del self.table[hash_index][index]
                return
        raise KeyError('Key not found')
```

Overwriting `hash_map.py`

Object Oriented Design: Lru Cache

This notebook was prepared by [Donne Martin](#). Source and license info is on [GitHub](#).

Design an LRU cache

Constraints and assumptions

- What are we caching?
 - We are caching the results of web queries
- Can we assume inputs are valid or do we have to validate them?
 - Assume they're valid
- Can we assume this fits memory?
 - Yes

Solution

```
%%writefile lru_cache.py
class Node(object):

    def __init__(self, results):
        self.results = results
        self.prev = None
        self.next = None


class LinkedList(object):

    def __init__(self):
        self.head = None
        self.tail = None

    def move_to_front(self, node): # ...
    def append_to_front(self, node): # ...
    def remove_from_tail(self): # ...


class Cache(object):

    def __init__(self, MAX_SIZE):
        self.MAX_SIZE = MAX_SIZE
        self.size = 0
        self.lookup = {} # key: query, value: node
        self.linked_list = LinkedList()

    def get(self, query)
        """Get the stored query result from the cache.

        Accessing a node updates its position to the front of the LRU list.
        """
        node = self.lookup.get(query)
        if node is None:
            return None
        self.linked_list.move_to_front(node)
        return node.results

    def set(self, results, query):
        """Set the result for the given query key in the cache.
```

```

When updating an entry, updates its position to the front of the LRU
list.
If the entry is new and the cache is at capacity, removes the oldest
entry
before the new entry is added.
.....
node = self.lookup.get(query)
if node is not None:
    # Key exists in cache, update the value
    node.results = results
    self.linked_list.move_to_front(node)
else:
    # Key does not exist in cache
    if self.size == self.MAX_SIZE:
        # Remove the oldest entry from the linked list and lookup
        self.lookup.pop(self.linked_list.tail.query, None)
        self.linked_list.remove_from_tail()
    else:
        self.size += 1
    # Add the new key and value
    new_node = Node(results)
    self.linked_list.append_to_front(new_node)
    self.lookup[query] = new_node

```

Overwriting `lru_cache.py`

Object Oriented Design: Call Center

This notebook was prepared by [Donne Martin](#). Source and license info is on [GitHub](#).

Design a call center

Constraints and assumptions

- What levels of employees are in the call center?
 - Operator, supervisor, director
- Can we assume operators always get the initial calls?
 - Yes

- If there is no available operators or the operator can't handle the call, does the call go to the supervisors?
 - Yes
- If there is no available supervisors or the supervisor can't handle the call, does the call go to the directors?
 - Yes
- Can we assume the directors can handle all calls?
 - Yes
- What happens if nobody can answer the call?
 - It gets queued
- Do we need to handle 'VIP' calls where we put someone to the front of the line?
 - No
- Can we assume inputs are valid or do we have to validate them?
 - Assume they're valid

Solution

```
%%writefile call_center.py
from abc import ABCMeta, abstractmethod
from collections import deque
from enum import Enum

class Rank(Enum):
    OPERATOR = 0
    SUPERVISOR = 1
    DIRECTOR = 2

class Employee(metaclass=ABCMeta):

    def __init__(self, employee_id, name, rank, call_center):
        self.employee_id = employee_id
        self.name = name
        self.rank = rank
        self.call = None
        self.call_center = call_center

    def take_call(self, call):
        """Assume the employee will always successfully take the call."""
        self.call = call
        self.call.employee = self
        self.call.state = CallState.IN_PROGRESS

    def complete_call(self):
        self.call.state = CallState.COMPLETE
        self.call_center.notify_call_completed(self.call)

    @abstractmethod
    def escalate_call(self):
        pass

    def _escalate_call(self):
        self.call.state = CallState.READY
        call = self.call
        self.call = None
        self.call_center.notify_call_escalated(call)
```

```

class Operator(Employee):

    def __init__(self, employee_id, name):
        super(Operator, self).__init__(employee_id, name, Rank.OPERATOR)

    def escalate_call(self):
        self.call.level = Rank.SUPERVISOR
        self._escalate_call()

class Supervisor(Employee):

    def __init__(self, employee_id, name):
        super(Operator, self).__init__(employee_id, name, Rank.SUPERVISOR)

    def escalate_call(self):
        self.call.level = Rank.DIRECTOR
        self._escalate_call()

class Director(Employee):

    def __init__(self, employee_id, name):
        super(Operator, self).__init__(employee_id, name, Rank.DIRECTOR)

    def escalate_call(self):
        raise NotImplemented('Directors must be able to handle any call')

class CallState(Enum):

    READY = 0
    IN_PROGRESS = 1
    COMPLETE = 2

class Call(object):

    def __init__(self, rank):
        self.state = CallState.READY
        self.rank = rank
        self.employee = None

```

```

class CallCenter(object):

    def __init__(self, operators, supervisors, directors):
        self.operators = operators
        self.supervisors = supervisors
        self.directors = directors
        self.queued_calls = deque()

    def dispatch_call(self, call):
        if call.rank not in (Rank.OPERATOR, Rank.SUPERVISOR, Rank.DIRECTOR):
            raise ValueError('Invalid call rank: {}'.format(call.rank))
        employee = None
        if call.rank == Rank.OPERATOR:
            employee = self._dispatch_call(call, self.operators)
        if call.rank == Rank.SUPERVISOR or employee is None:
            employee = self._dispatch_call(call, self.supervisors)
        if call.rank == Rank.DIRECTOR or employee is None:
            employee = self._dispatch_call(call, self.directors)
        if employee is None:
            self.queued_calls.append(call)

    def _dispatch_call(self, call, employees):
        for employee in employees:
            if employee.call is None:
                employee.take_call(call)
                return employee
        return None

    def notify_call_escalated(self, call): # ...
    def notify_call_completed(self, call): # ...
    def dispatch_queued_call_to_newly_freed_employee(self, call, employee):
        # ...

```

Overwriting call_center.py

Object Oriented Design: Deck Of Cards

This notebook was prepared by [Donne Martin](#). Source and license info is on [GitHub](#).

Design a deck of cards

Constraints and assumptions

- Is this a generic deck of cards for games like poker and black jack?
 - Yes, design a generic deck then extend it to black jack
- Can we assume the deck has 52 cards (2-10, Jack, Queen, King, Ace) and 4 suits?
 - Yes
- Can we assume inputs are valid or do we have to validate them?
 - Assume they're valid

Solution

```
%%writefile deck_of_cards.py
from abc import ABCMeta, abstractmethod
from enum import Enum
import sys

class Suit(Enum):
    HEART = 0
    DIAMOND = 1
    CLUBS = 2
    SPADE = 3

class Card(metaclass=ABCMeta):

    def __init__(self, value, suit):
        self.value = value
        self.suit = suit
        self.is_available = True

    @property
    @abstractmethod
    def value(self):
        pass

    @value.setter
    @abstractmethod
    def value(self, other):
        pass

class BlackJackCard(Card):

    def __init__(self, value, suit):
        super(BlackJackCard, self).__init__(value, suit)

    def is_ace(self):
        return self._value == 1

    def is_face_card(self):
        """Jack = 11, Queen = 12, King = 13"""

```

```

        return 10 < self._value <= 13

@property
def value(self):
    if self.is_ace() == 1:
        return 1
    elif self.is_face_card():
        return 10
    else:
        return self._value

@value.setter
def value(self, new_value):
    if 1 <= new_value <= 13:
        self._value = new_value
    else:
        raise ValueError('Invalid card value: {}'.format(new_value))

class Hand(object):

    def __init__(self, cards):
        self.cards = cards

    def add_card(self, card):
        self.cards.append(card)

    def score(self):
        total_value = 0
        for card in self.cards:
            total_value += card.value
        return total_value

class BlackJackHand(Hand):

    BLACKJACK = 21

    def __init__(self, cards):
        super(BlackJackHand, self).__init__(cards)

    def score(self):
        min_over = sys.MAXSIZE
        max_under = -sys.MAXSIZE
        for score in self.possible_scores():

```

```

        if self.BLACKJACK < score < min_over:
            min_over = score
        elif max_under < score <= self.BLACKJACK:
            max_under = score
    return max_under if max_under != -sys.MAXSIZE else min_over

def possible_scores(self):
    """Return a list of possible scores, taking Aces into account."""
    # ...


class Deck(object):

    def __init__(self, cards):
        self.cards = cards
        self.deal_index = 0

    def remaining_cards(self):
        return len(self.cards) - deal_index

    def deal_card():
        try:
            card = self.cards[self.deal_index]
            card.is_available = False
            self.deal_index += 1
        except IndexError:
            return None
        return card

    def shuffle(self): # ...

```

Overwriting `deck_of_cards.py`

Object Oriented Design: Parking Lot

This notebook was prepared by [Donne Martin](#). Source and license info is on [GitHub](#).

Design a parking lot

Constraints and assumptions

- What types of vehicles should we support?
 - Motorcycle, Car, Bus
- Does each vehicle type take up a different amount of parking spots?
 - Yes
 - Motorcycle spot -> Motorcycle
 - Compact spot -> Motorcycle, Car
 - Large spot -> Motorcycle, Car
 - Bus can park if we have 5 consecutive "large" spots
- Does the parking lot have multiple levels?
 - Yes

Solution

```
%%writefile parking_lot.py
from abc import ABCMeta, abstractmethod

class VehicleSize(Enum):
    MOTORCYCLE = 0
    COMPACT = 1
    LARGE = 2

class Vehicle(metaclass=ABCMeta):

    def __init__(self, vehicle_size, license_plate, spot_size):
        self.vehicle_size = vehicle_size
        self.license_plate = license_plate
        self.spot_size = spot_size
        self.spots_taken = []

    def clear_spots(self):
        for spot in self.spots_taken:
            spot.remove_vehicle(self)
        self.spots_taken = []

    def take_spot(self, spot):
        self.spots_taken.append(spot)

    @abstractmethod
    def can_fit_in_spot(self, spot):
        pass

class Motorcycle(Vehicle):

    def __init__(self, license_plate):
        super(Motorcycle, self).__init__(VehicleSize.MOTORCYCLE,
                                         license_plate, spot_size=1)

    def can_fit_in_spot(self, spot):
        return True
```

```

class Car(Vehicle):

    def __init__(self, license_plate):
        super(Car, self).__init__(VehicleSize.COMPACT, license_plate,
                                 spot_size=1)

    def can_fit_in_spot(self, spot):
        return True if (spot.size == LARGE or spot.size == COMPACT) else
        False


class Bus(Vehicle):

    def __init__(self, license_plate):
        super(Bus, self).__init__(VehicleSize.LARGE, license_plate,
                                 spot_size=5)

    def can_fit_in_spot(self, spot):
        return True if spot.size == LARGE else False


class ParkingLot(object):

    def __init__(self, num_levels):
        self.num_levels = num_levels
        self.levels = []

    def park_vehicle(self, vehicle):
        for level in levels:
            if level.park_vehicle(vehicle):
                return True
        return False


class Level(object):

    SPOTS_PER_ROW = 10

    def __init__(self, floor, total_spots):
        self.floor = floor
        self.num_spots = total_spots
        self.available_spots = 0
        self.parking_spots = []

    def spot_freed(self):
        self.available_spots += 1

```

```

def park_vehicle(self, vehicle):
    spot = self._find_available_spot(vehicle)
    if spot is None:
        return None
    else:
        spot.park_vehicle(vehicle)
        return spot

def _find_available_spot(self, vehicle):
    """Find an available spot where vehicle can fit, or return None"""
    # ...

def _park_starting_at_spot(self, spot, vehicle):
    """Occupy starting at spot.spot_number to vehicle.spot_size."""
    # ...


class ParkingSpot(object):

    def __init__(self, level, row, spot_number, spot_size, vehicle_size):
        self.level = level
        self.row = row
        self.spot_number = spot_number
        self.spot_size = spot_size
        self.vehicle_size = vehicle_size
        self.vehicle = None

    def is_available(self):
        return True if self.vehicle is None else False

    def can_fit_vehicle(self, vehicle):
        if self.vehicle is not None:
            return False
        return vehicle.can_fit_in_spot(self)

    def park_vehicle(self, vehicle): # ...
    def remove_vehicle(self): # ...

```

Overwriting parking_lot.py

Object Oriented Design: Online Chat

This notebook was prepared by [Donne Martin](#). Source and license info is on [GitHub](#).

Design an online chat

Constraints and assumptions

- Assume we'll focus on the following workflows:
 - Text conversations only
 - Users
 - Add a user
 - Remove a user
 - Update a user
 - Add to a user's friends list
 - Add friend request
 - Approve friend request
 - Reject friend request
 - Remove from a user's friends list
 - Create a group chat
 - Invite friends to a group chat
 - Post a message to a group chat
 - Private 1-1 chat
 - Invite a friend to a private chat
 - Post a message to a private chat
- No need to worry about scaling initially

Solution

```
%%writefile online_chat.py
from abc import ABCMeta

class UserService(object):

    def __init__(self):
        self.users_by_id = {} # key: user id, value: User

    def add_user(self, user_id, name, pass_hash): # ...
    def remove_user(self, user_id): # ...
    def add_friend_request(self, from_user_id, to_user_id): # ...
    def approve_friend_request(self, from_user_id, to_user_id): # ...
    def reject_friend_request(self, from_user_id, to_user_id): # ...


class User(object):

    def __init__(self, user_id, name, pass_hash):
        self.user_id = user_id
        self.name = name
        self.pass_hash = pass_hash
        self.friends_by_id = {} # key: friend id, value: User
        self.friend_ids_to_private_chats = {} # key: friend id, value:
                                             private chats
        self.group_chats_by_id = {} # key: chat id, value: GroupChat
        self.received_friend_requests_by_friend_id = {} # key: friend id,
                                                       value: AddRequest
        self.sent_friend_requests_by_friend_id = {} # key: friend id, value:
                                                       AddRequest

    def message_user(self, friend_id, message): # ...
    def message_group(self, group_id, message): # ...
    def send_friend_request(self, friend_id): # ...
    def receive_friend_request(self, friend_id): # ...
    def approve_friend_request(self, friend_id): # ...
    def reject_friend_request(self, friend_id): # ...


class Chat(metaclass=ABCMeta):

    def __init__(self, chat_id):
        self.chat_id = chat_id
```

```

        self.users = []
        self.messages = []

class PrivateChat(Chat):

    def __init__(self, first_user, second_user):
        super(PrivateChat, self).__init__()
        self.users.append(first_user)
        self.users.append(second_user)

class GroupChat(Chat):

    def add_user(self, user): # ...
    def remove_user(self, user): # ...

class Message(object):

    def __init__(self, message_id, message, timestamp):
        self.message_id = message_id
        self.message = message
        self.timestamp = timestamp

class AddRequest(object):

    def __init__(self, from_user_id, to_user_id, request_status, timestamp):
        self.from_user_id = from_user_id
        self.to_user_id = to_user_id
        self.request_status = request_status
        self.timestamp = timestamp

class RequestStatus(Enum):

    UNREAD = 0
    READ = 1
    ACCEPTED = 2
    REJECTED = 3

```

Overwriting `online_chat.py`

