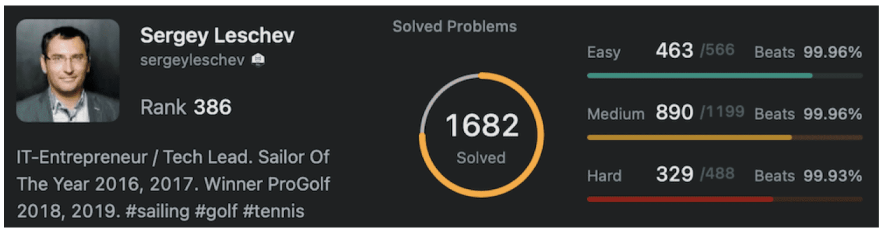
S.Leschev System Architect

🏆 Awards

Ranking #Dev: Global TOP 300 (Certificate)



Sergey Leschev
sergeyleschev

Rank 386

IT-Entrepreneur / Tech Lead. Sailor Of The Year 2016, 2017. Winner ProGolf 2018, 2019. #sailing #golf #tennis

Solved Problems

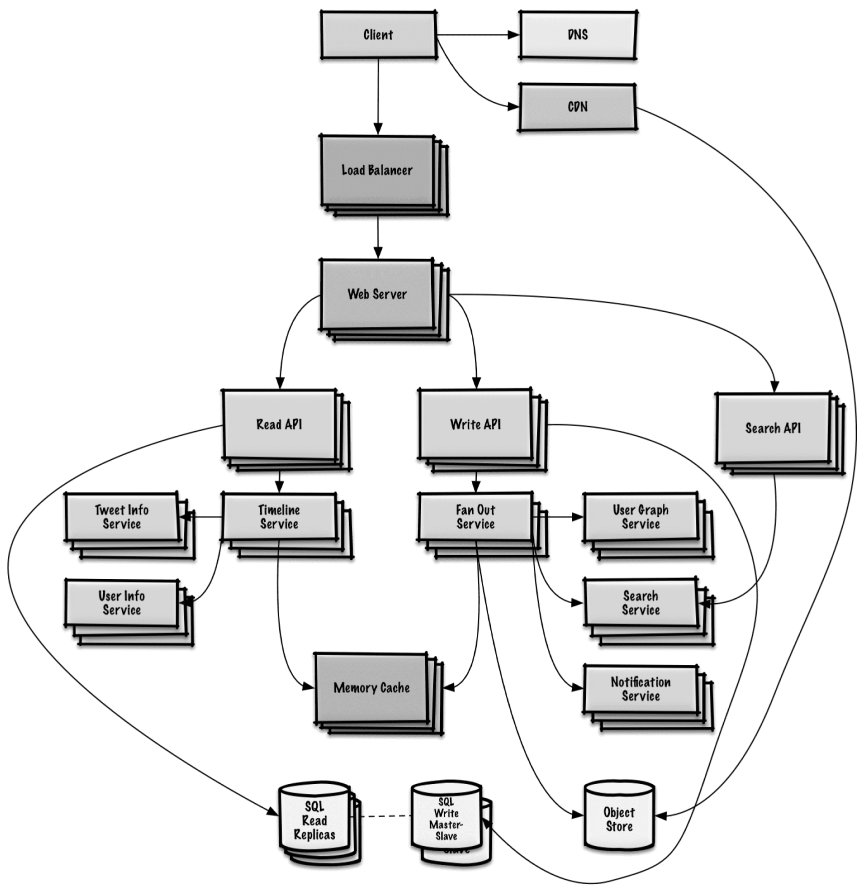
1682
Solved

```
Easy     463  /566    Beats  99.96%
Medium   890  /1199   Beats  99.96%
Hard     329  /488    Beats  99.93%
```

The System Design

Design large-scale systems

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



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.

How to approach a system design questions

> How to tackle a system design question.s

The system design 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 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 questions
- The system design questions
- Intro to Architecture and Systems Design questions
- System design template

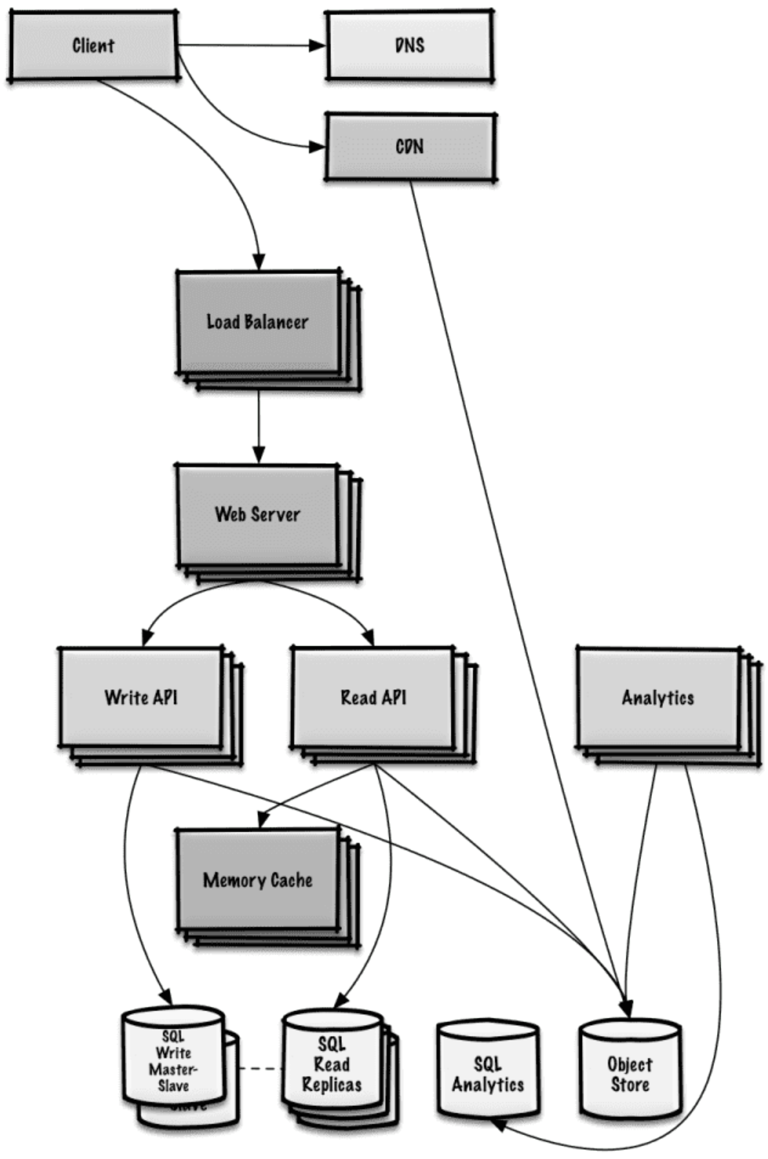
System design questions with solutions

> Common system design 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

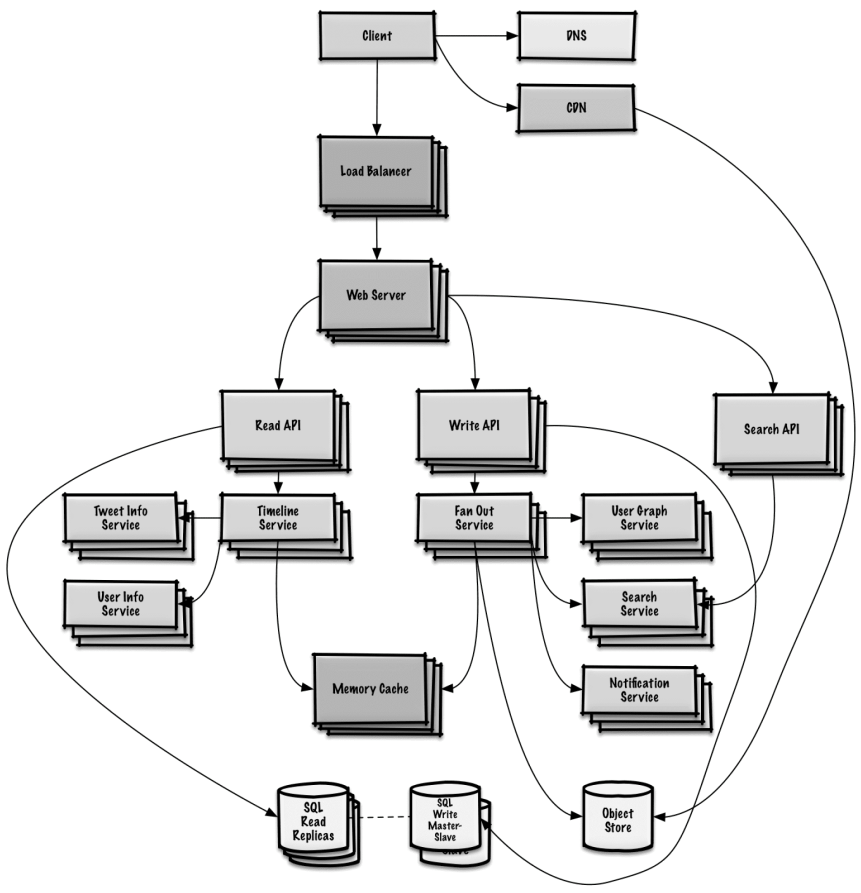
Design Pastebin.com (or Bit.ly)

View solution



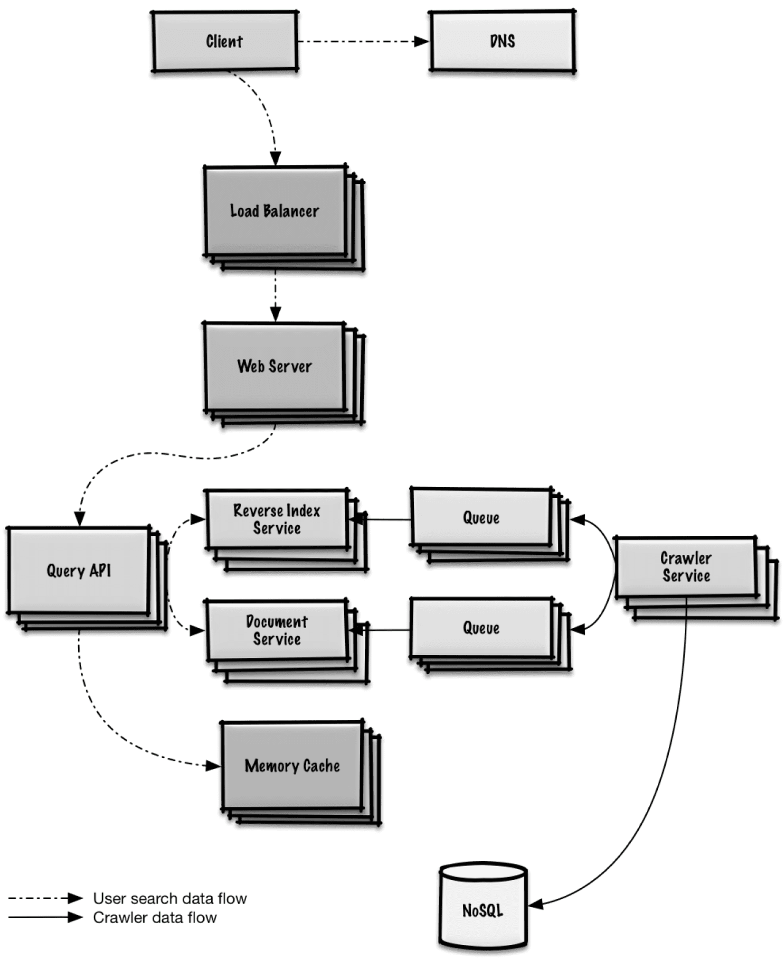
Design the Twitter timeline and search (or Facebook feed and search)

View solution



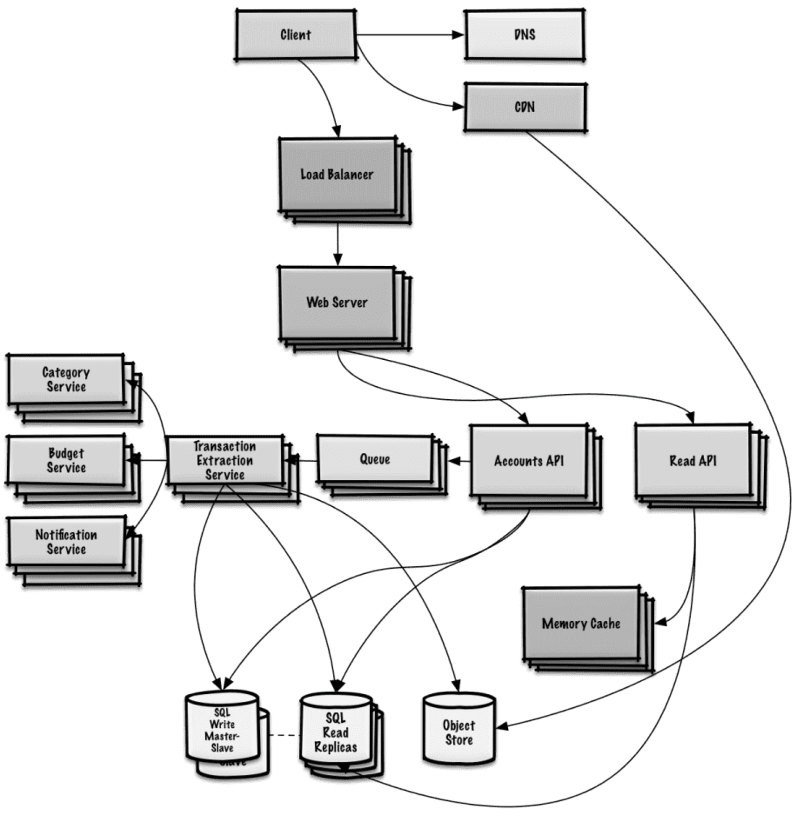
Design a web crawler

View solution



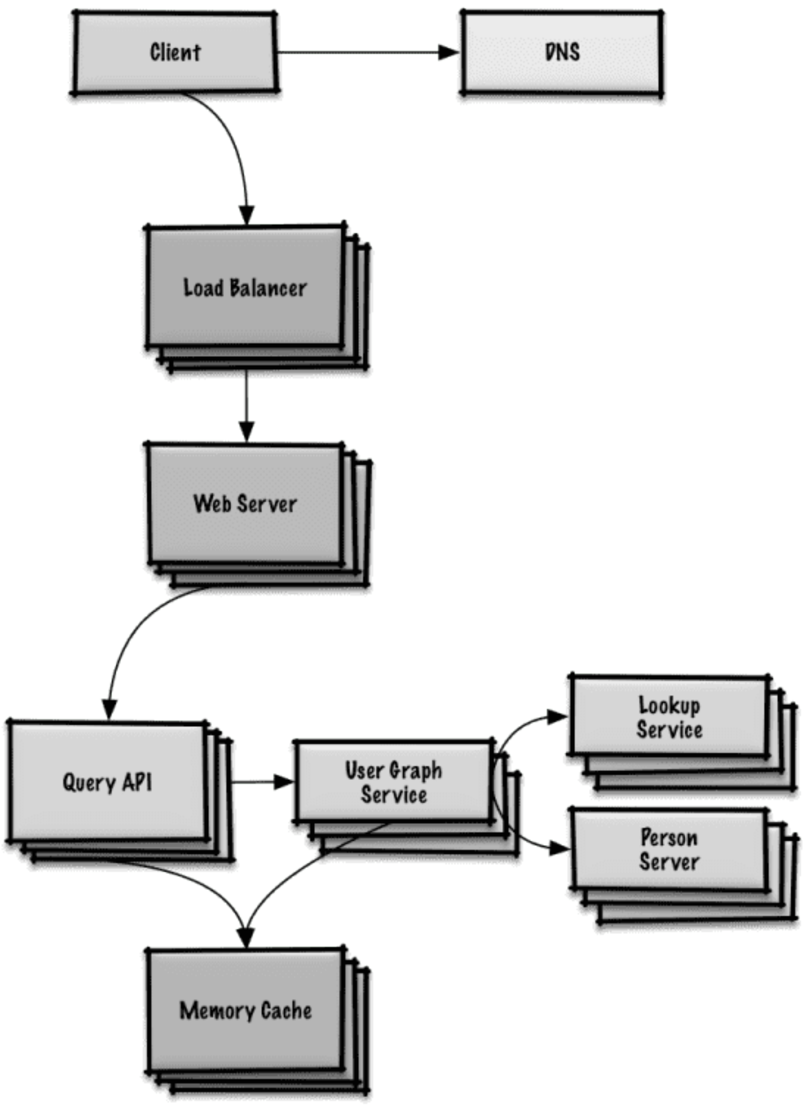
Design Mint.com

View solution



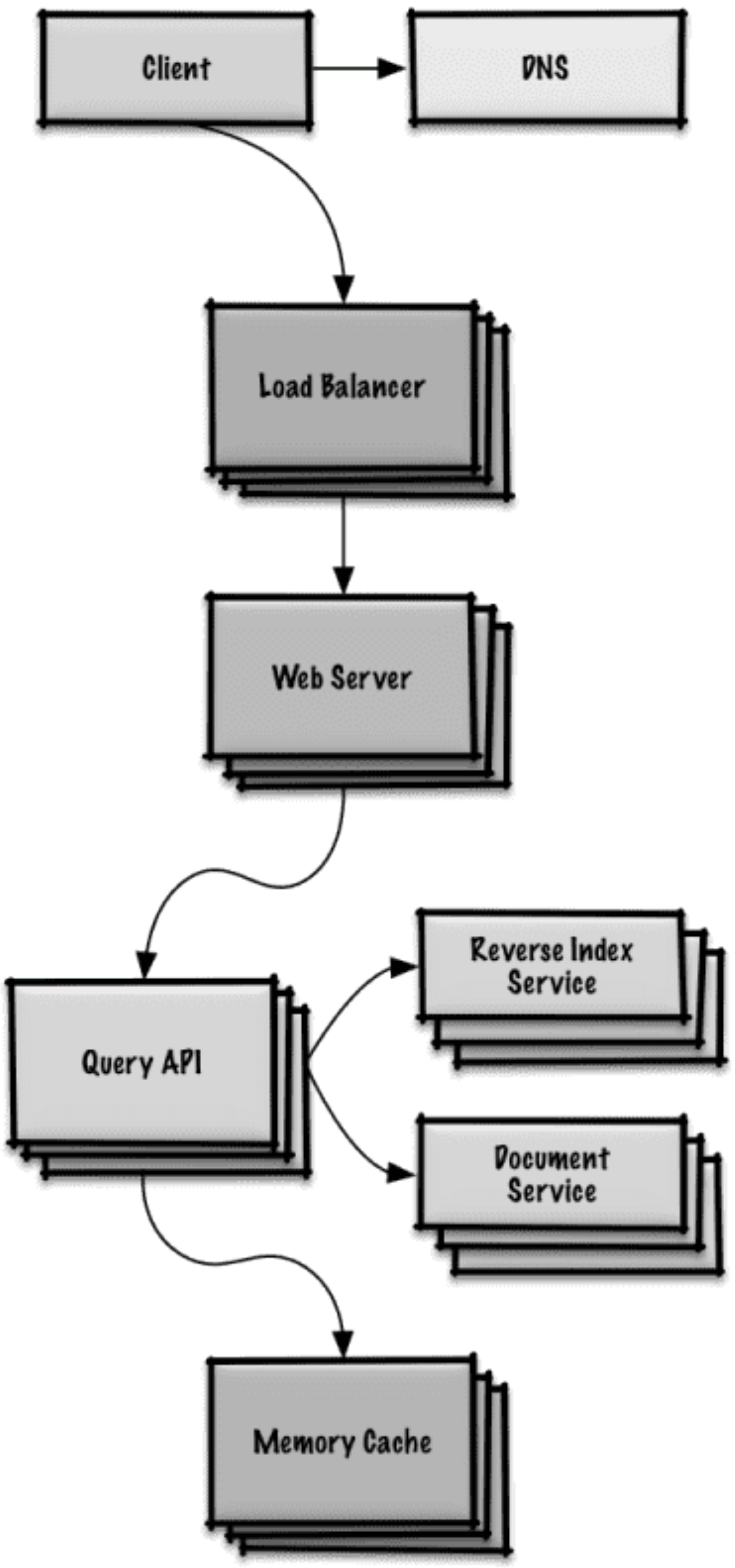
Design the data structures for a social network

View solution



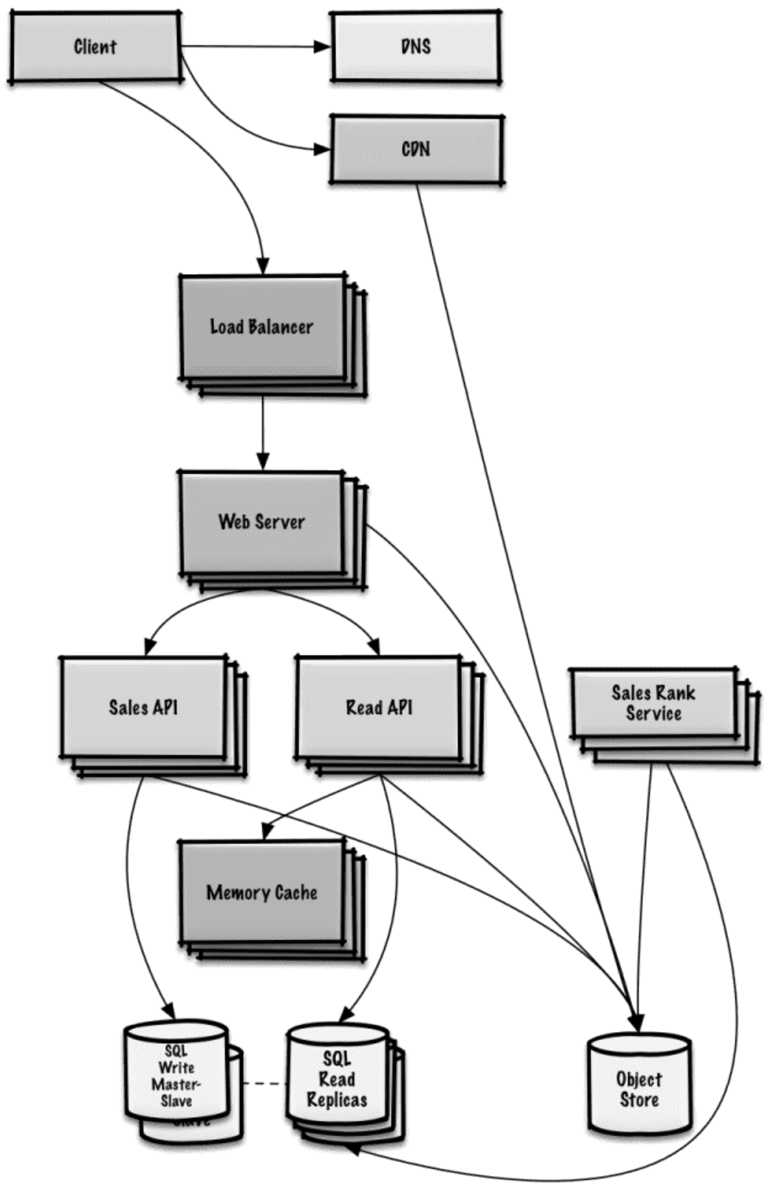
Design a key-value store for a search engine

View solution



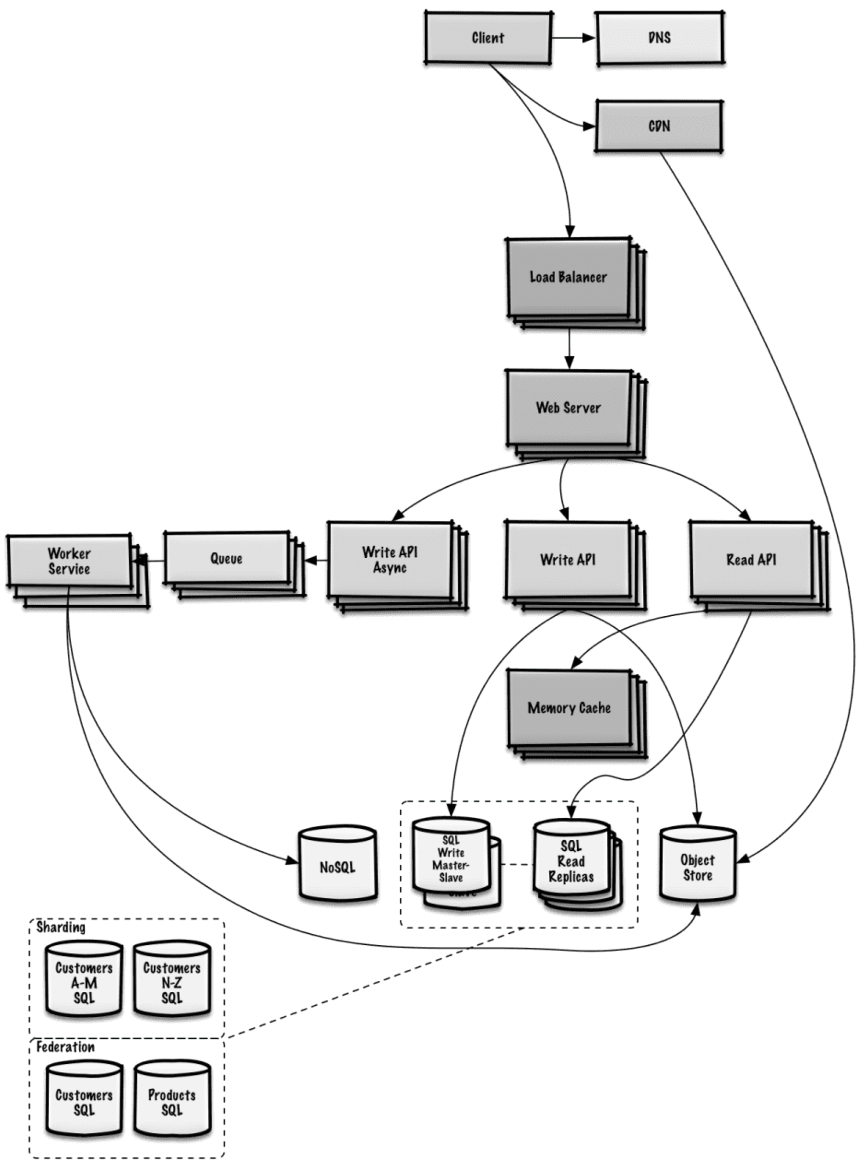
Design Amazon's sales ranking by category feature

View solution



Design a system that scales to millions of users on AWS

View solution



Object-oriented design questions with solutions

> Common object-oriented design 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

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.

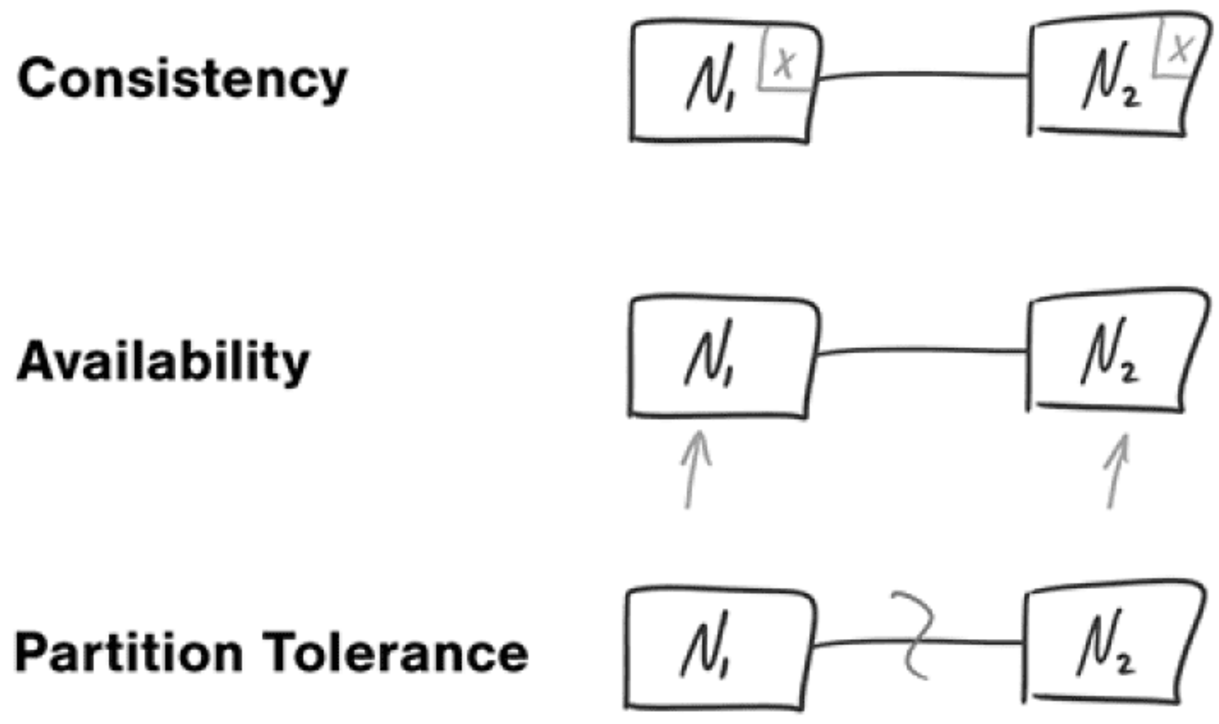
Generally, you should aim for **maximal throughput** with **acceptable latency**.

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 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:

```
Availability (Total) = Availability (Foo) * Availability (Bar)
```

If both `Foo` and `Bar` each had 99.9% availability, their total availability in sequence would be 99.8%.

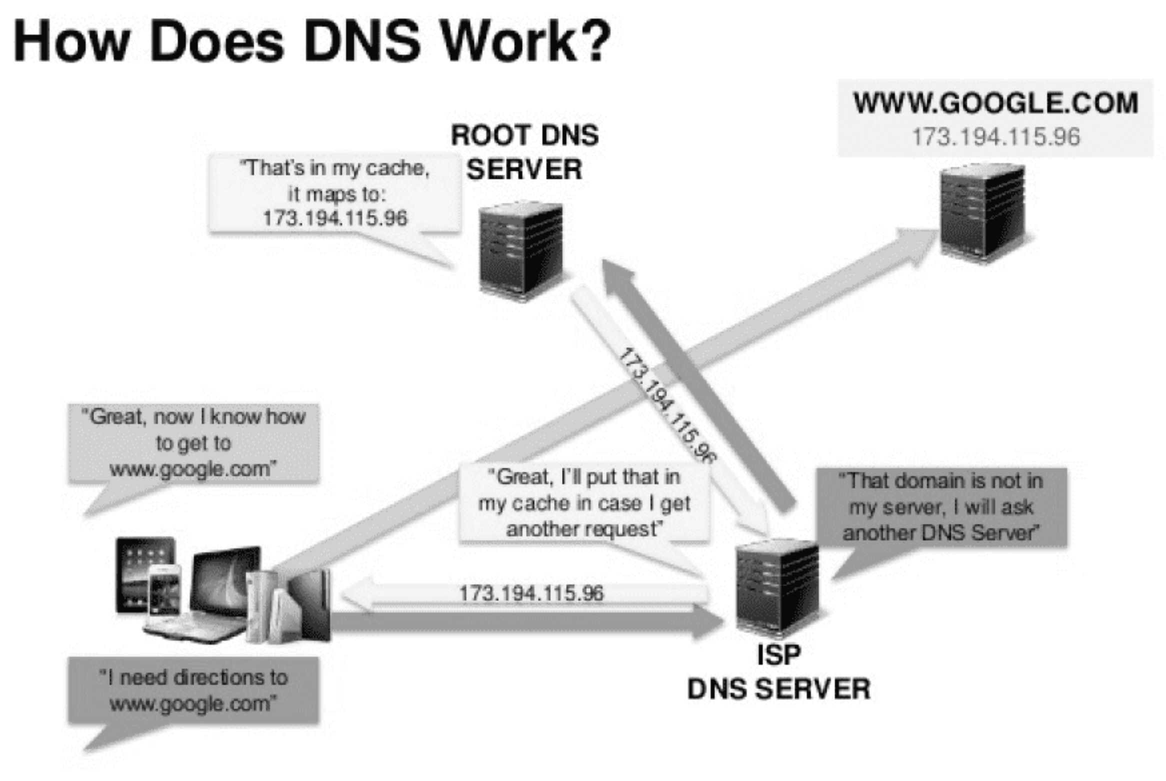
In parallel

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

```
Availability (Total) = 1 - (1 - Availability (Foo)) * (1 -
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?
ROOT DNS SERVER
WWW.GOOGLE.COM 173.194.115.96
"That's in my cache, it maps to: 173.194.115.96
173.194.115.96
"Great, now I know how to get to www.google.com"
"Great, I'll put that in my cache in case I get another request"
"That domain is not in my server, I will ask another DNS Server"
173.194.115.96
"I need directions to www.google.com"
ISP DNS SERVER

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

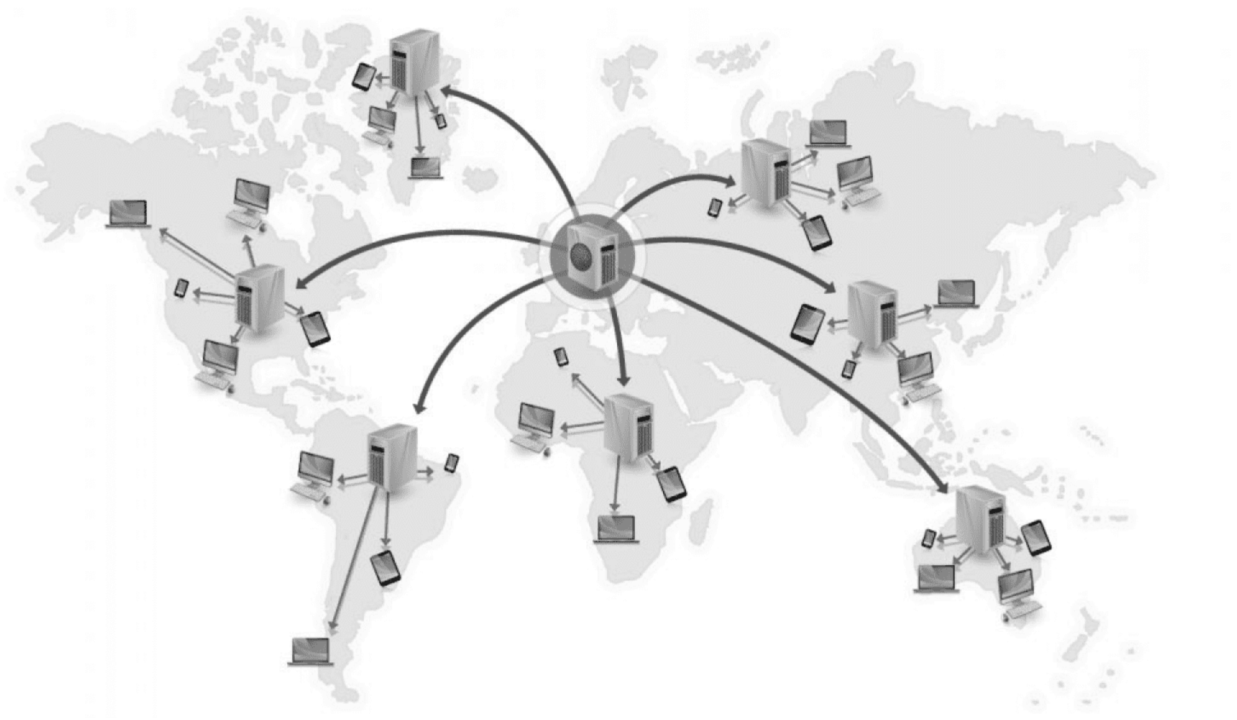
Disadvantage(s): DNS

- 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.

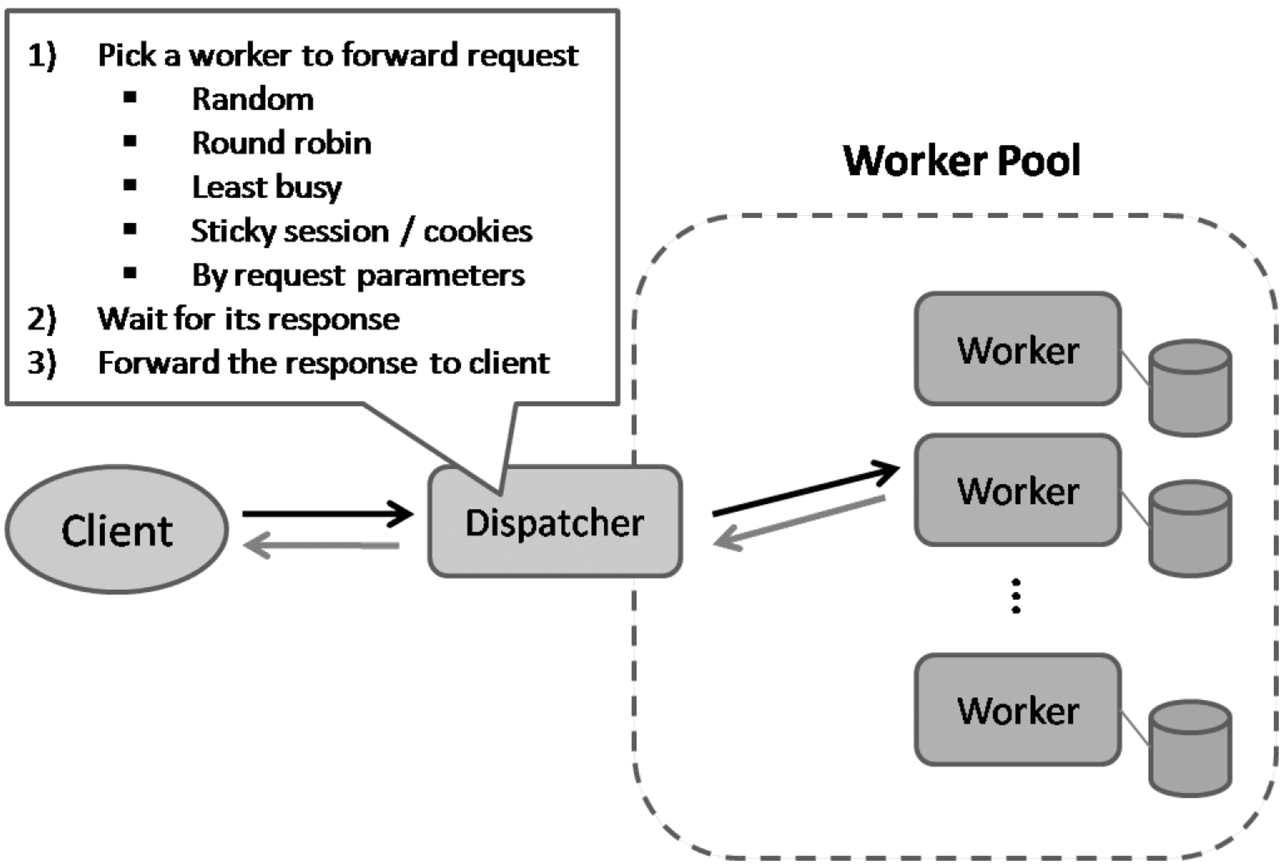
Disadvantage(s): CDN

- 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

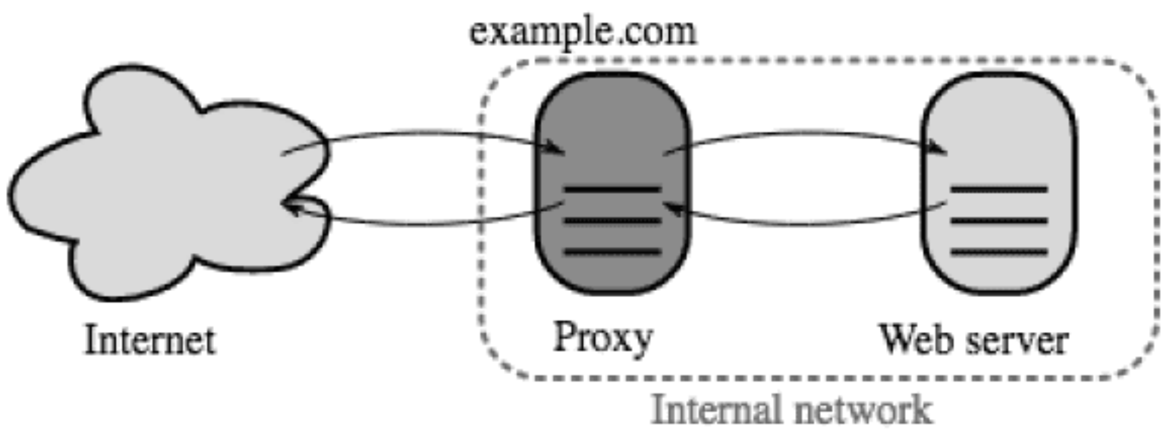
Disadvantage(s): load balancer

- 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.

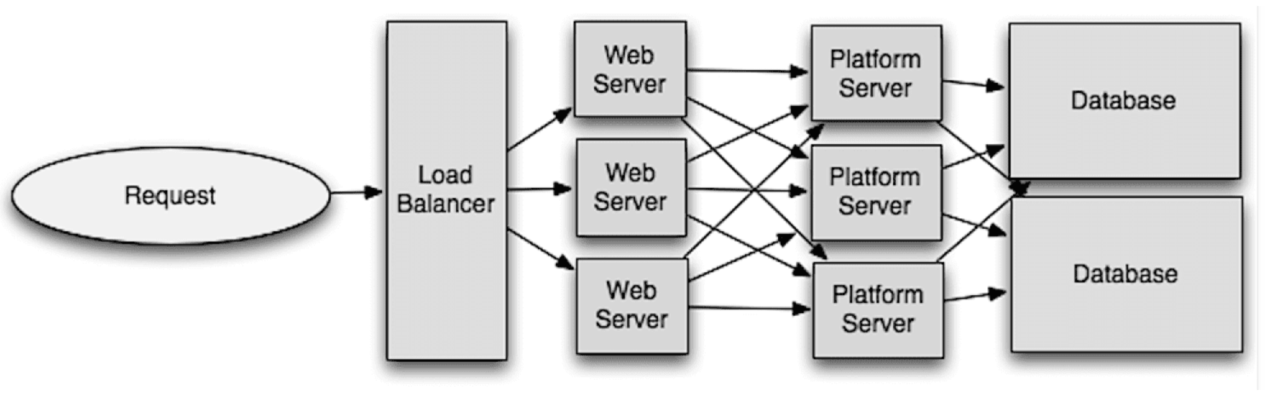
Disadvantage(s): reverse proxy

- 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.

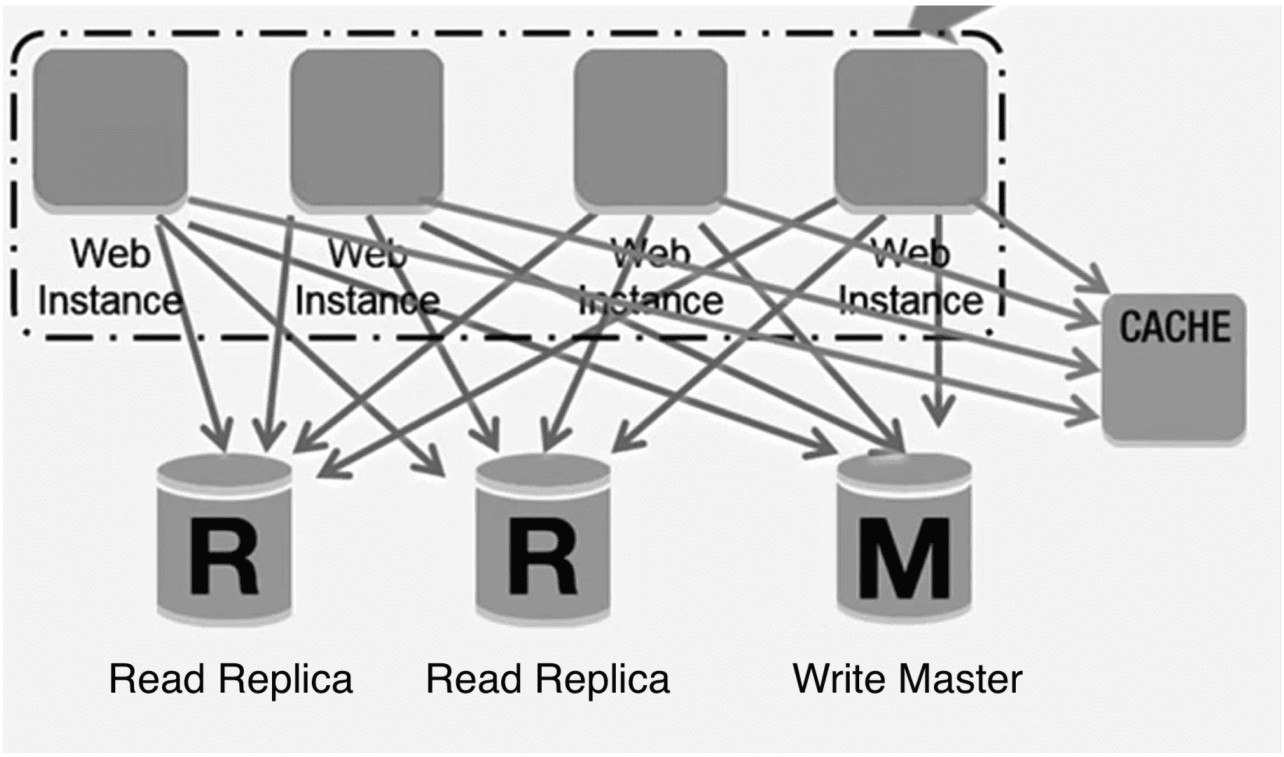
Disadvantage(s): application layer

- 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 questions
- 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.

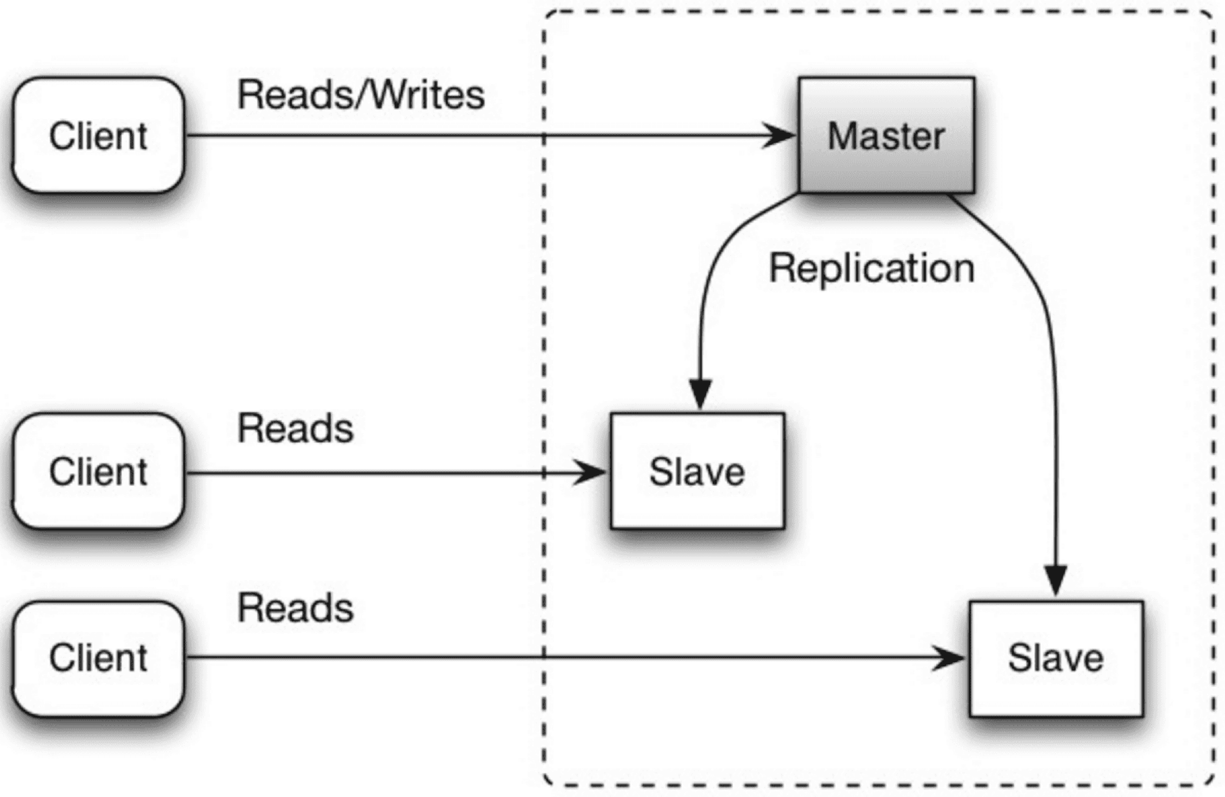
ACID is a set of properties of relational database transactions.

- **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.



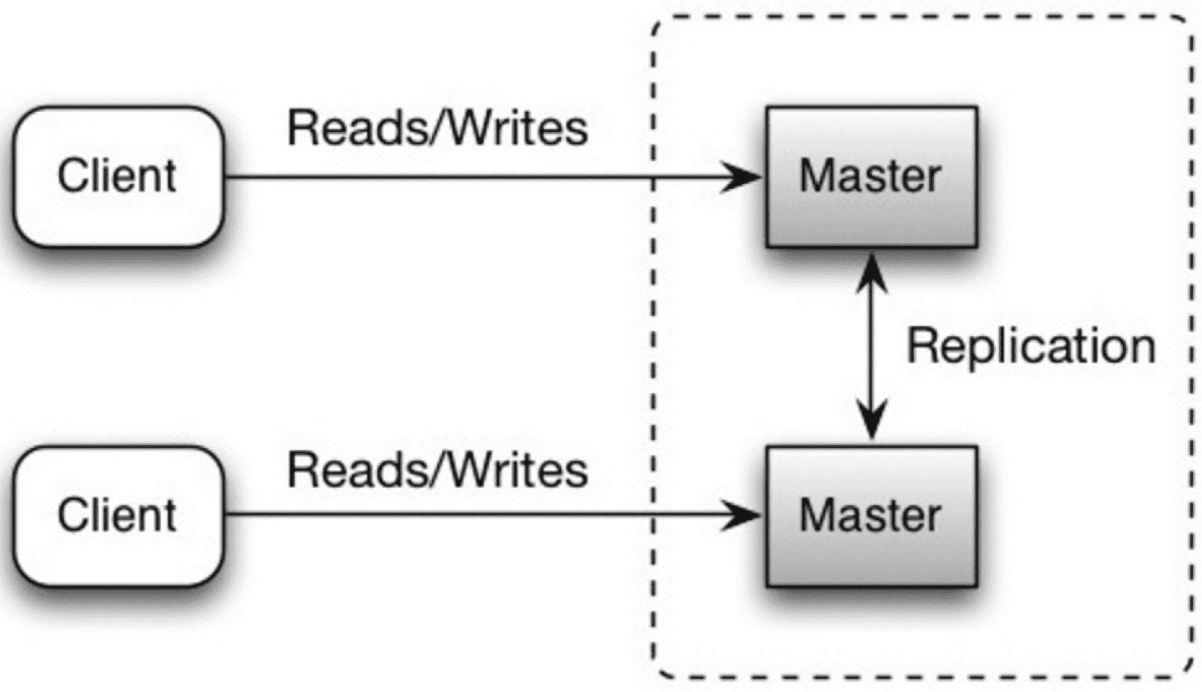
Source: Scalability, availability, stability, patterns

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

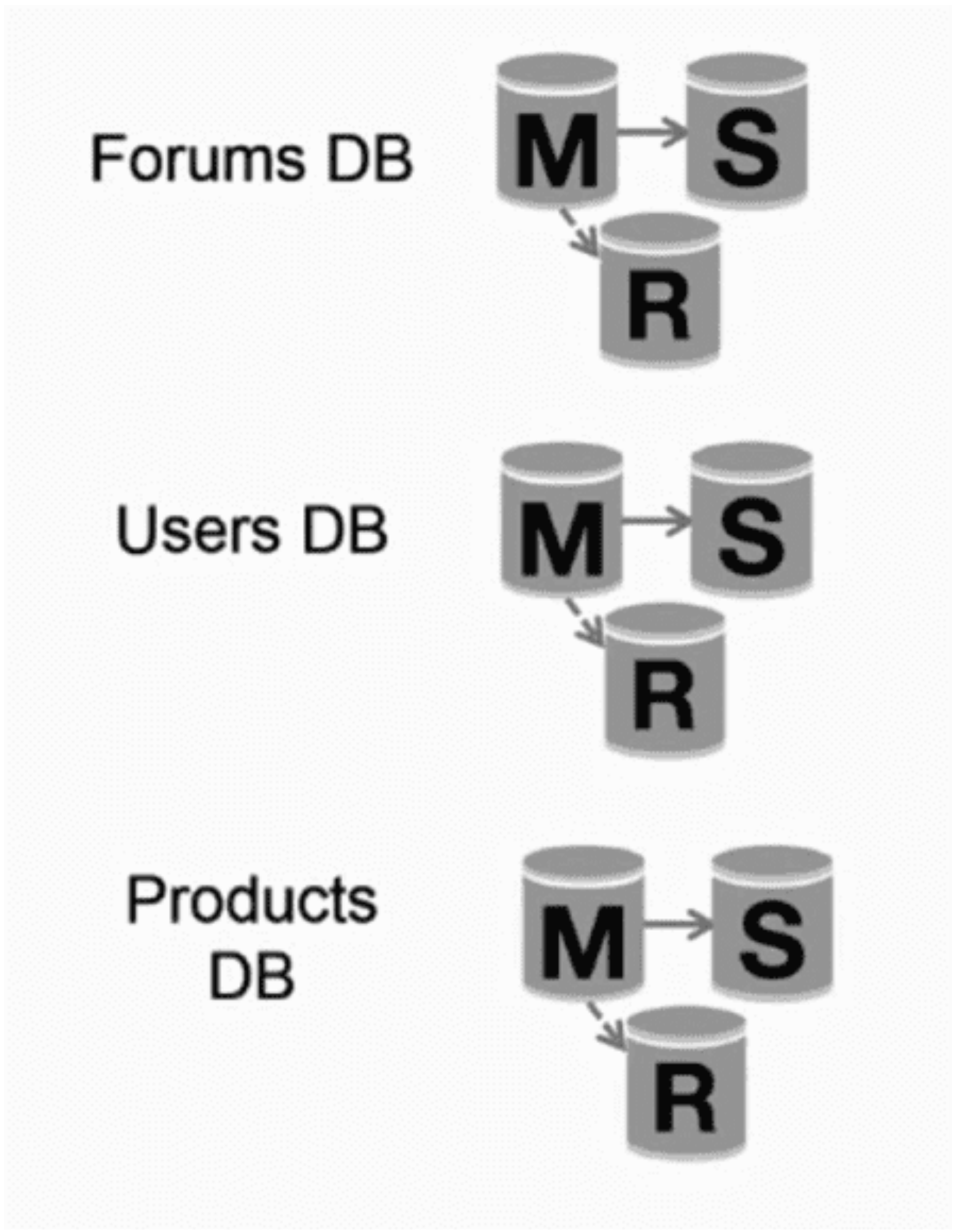
thread.

- Replication adds more hardware and additional complexity.

Source(s) and further reading: replication

- Scalability, availability, stability, patterns
- Multi-master replication

Federation



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.

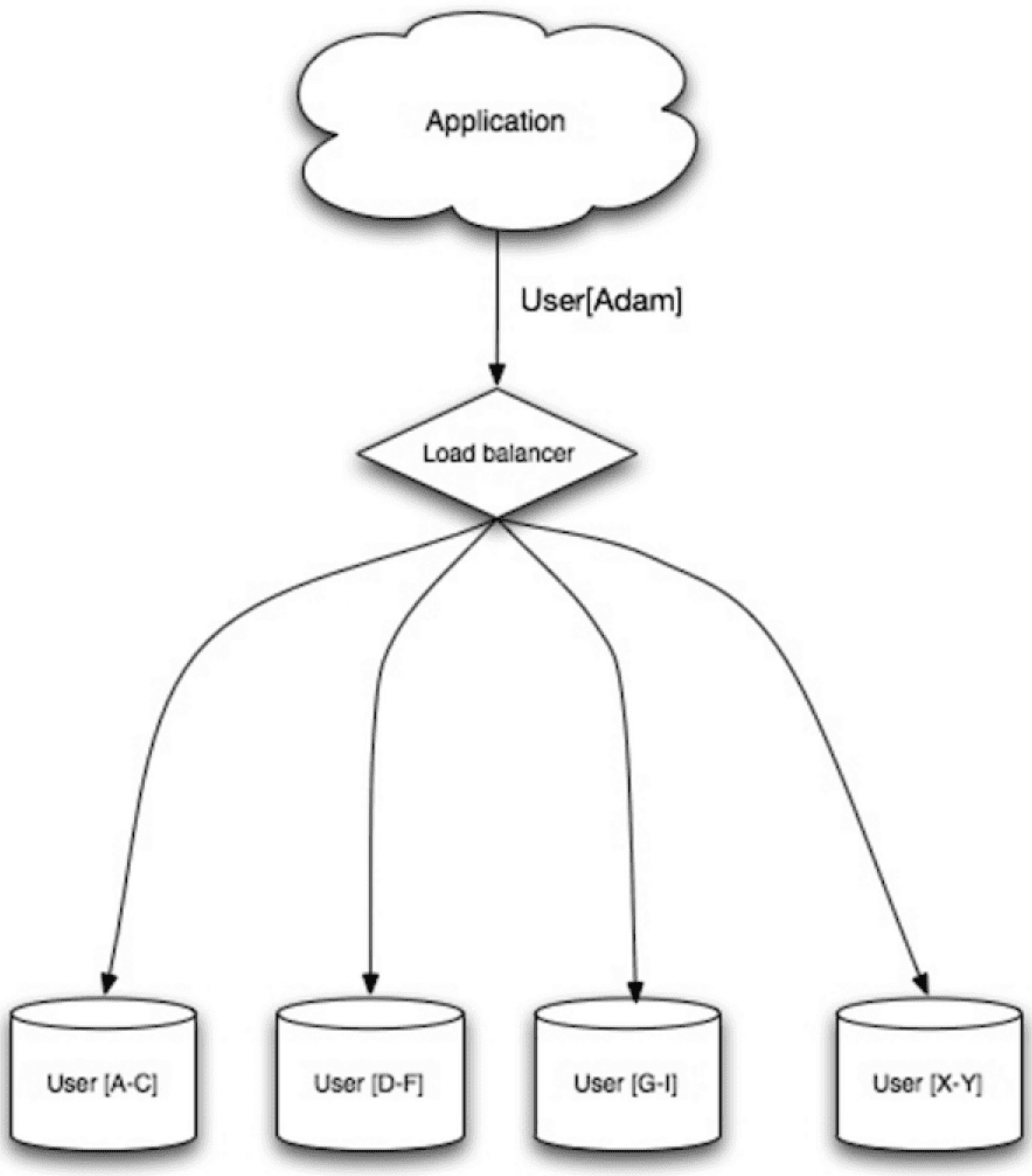
Disadvantage(s): federation

- 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.

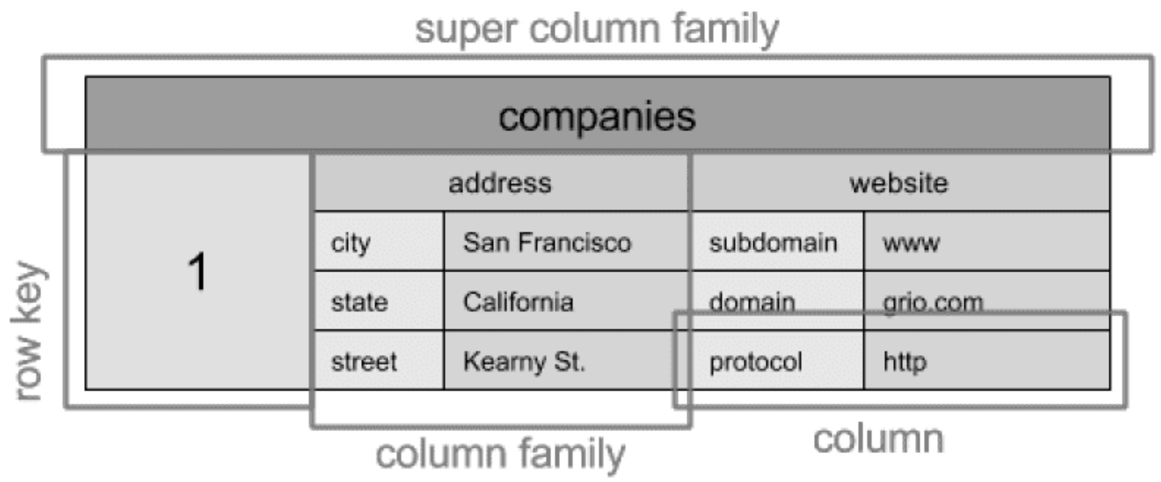
Some document stores like MongoDB and CouchDB also provide a SQL-like language to perform complex queries. DynamoDB supports both key-values and documents.

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



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.

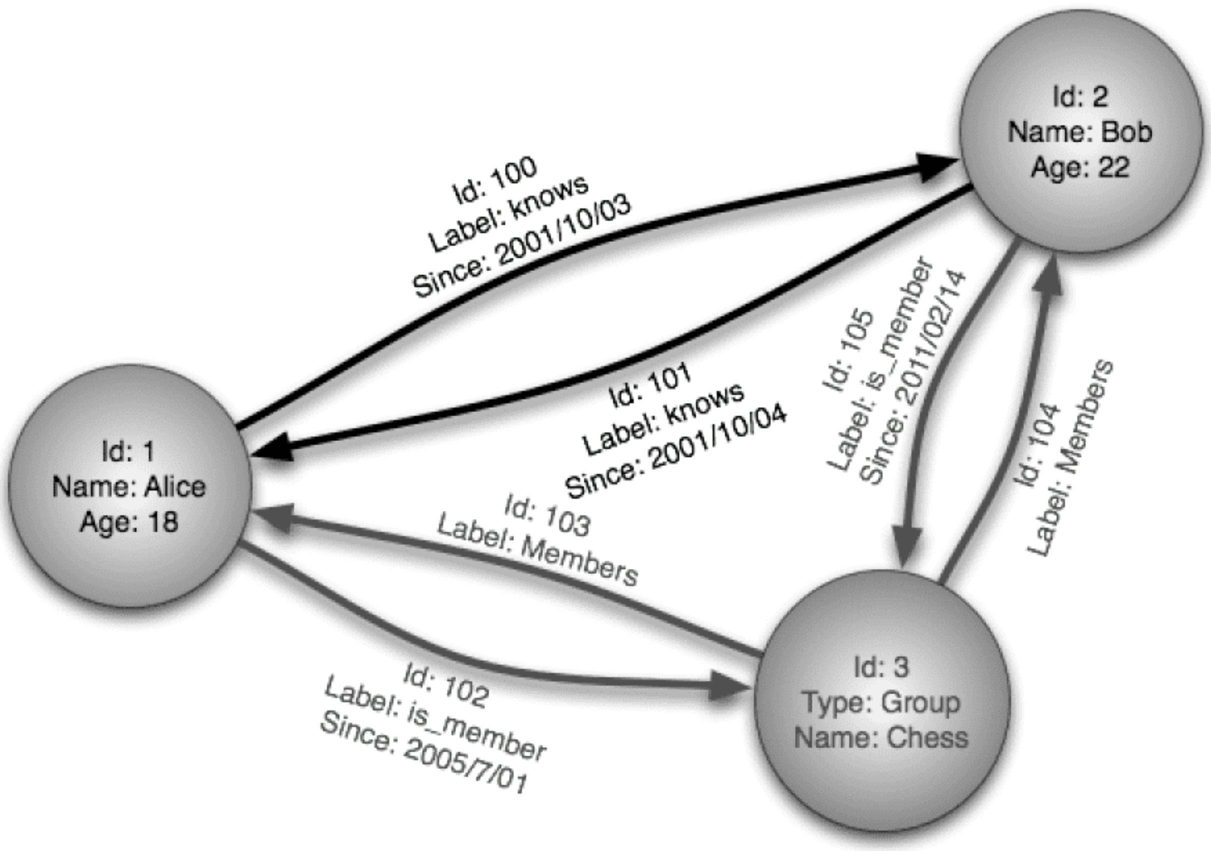
Google introduced Bigtable as the first wide column store, which influenced the open-source HBase often-used in the Hadoop ecosystem, and Cassandra from Facebook. Stores such as BigTable, HBase, and Cassandra maintain keys in lexicographic order, allowing efficient retrieval of selective key ranges.

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.

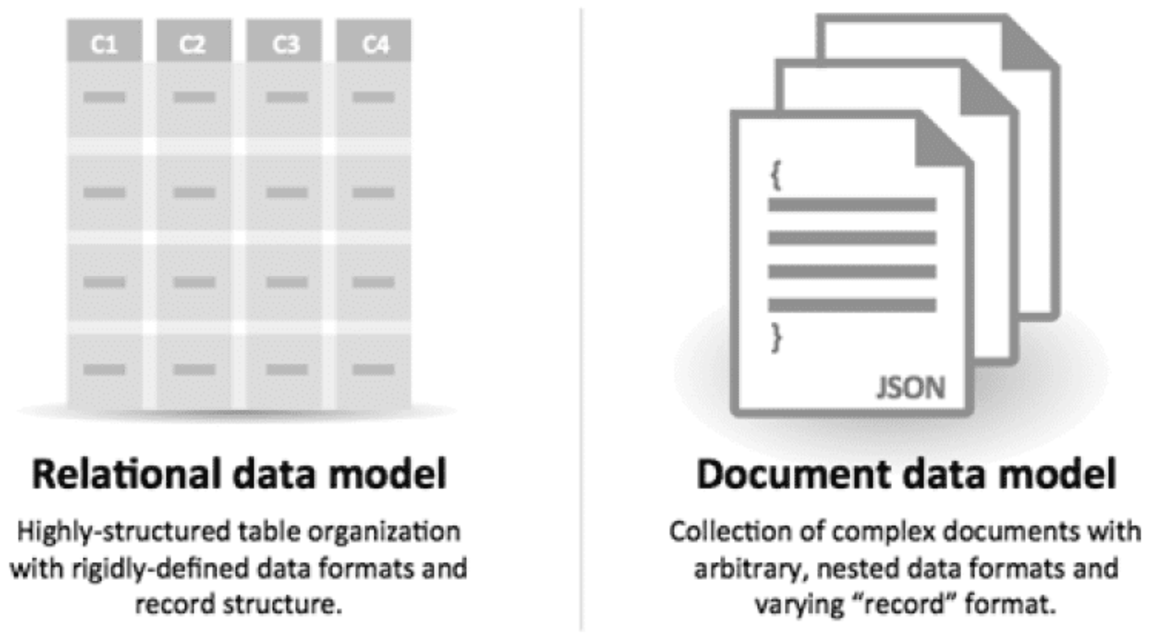
Source(s) and further reading: graph

- 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


C1
C2
C3
C4
Relational data model
Highly-structured table organization with rigidly-defined data formats and record structure.
{
}
JSON
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

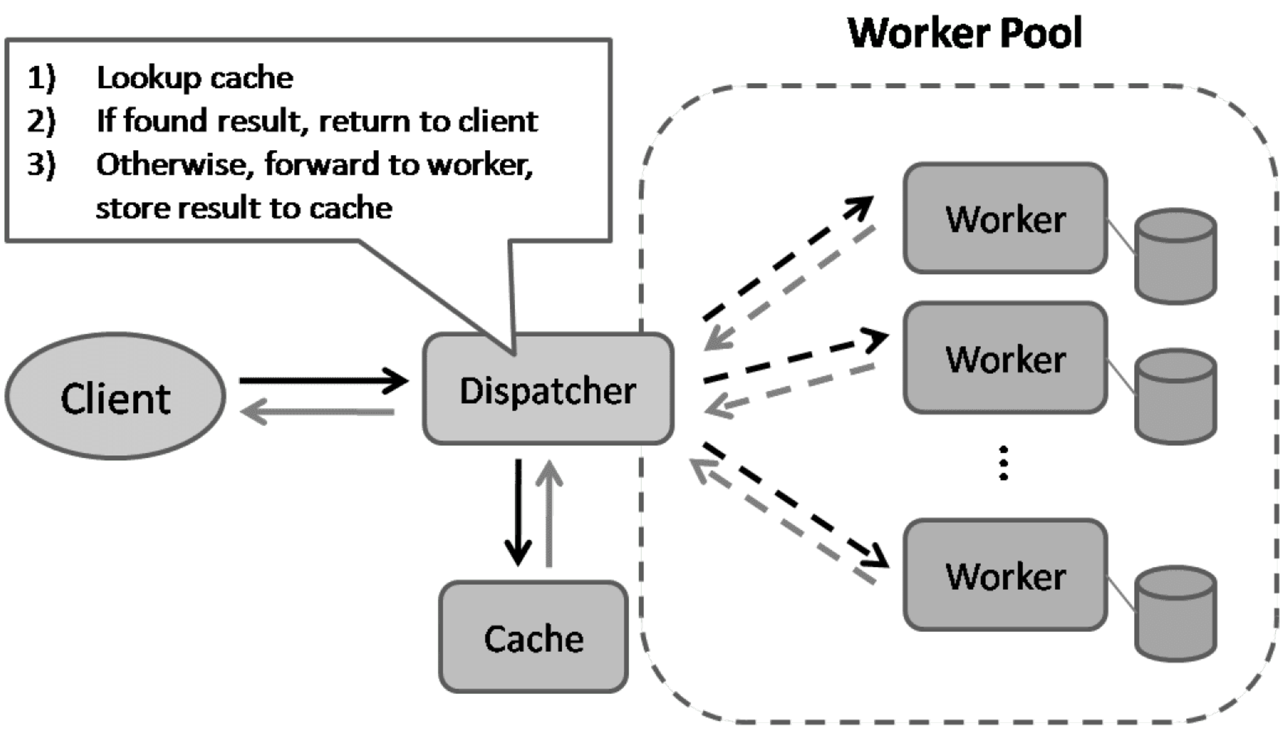
Sample data well-suited for NoSQL:

- 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


1) Lookup cache
2) If found result, return to client
3) Otherwise, forward to worker, store result to cache
Worker Pool
Client
Dispatcher
Cache
Worker
Worker
Worker

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

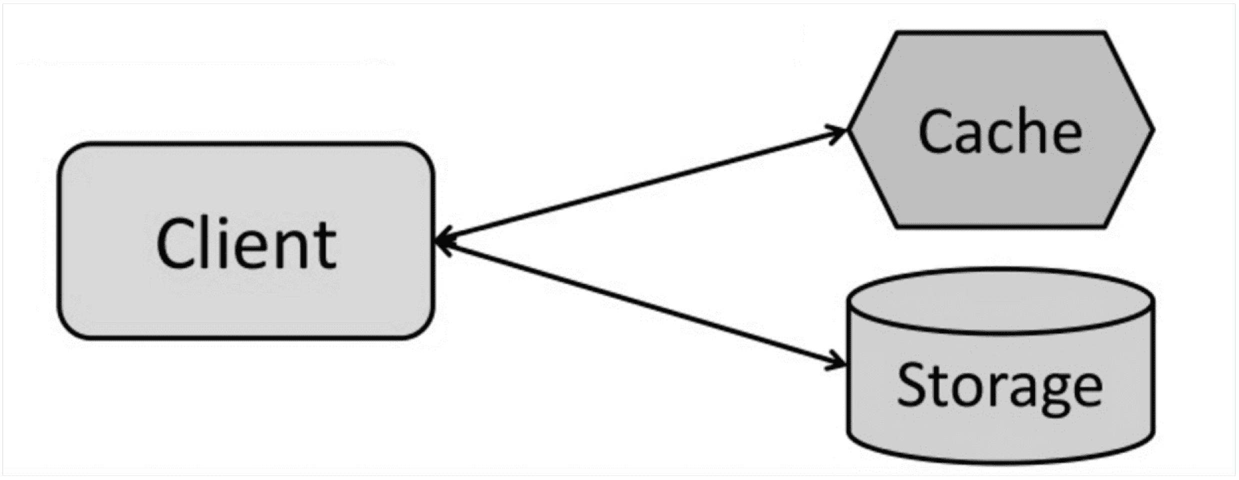
Suggestions of what to cache:

- 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
```

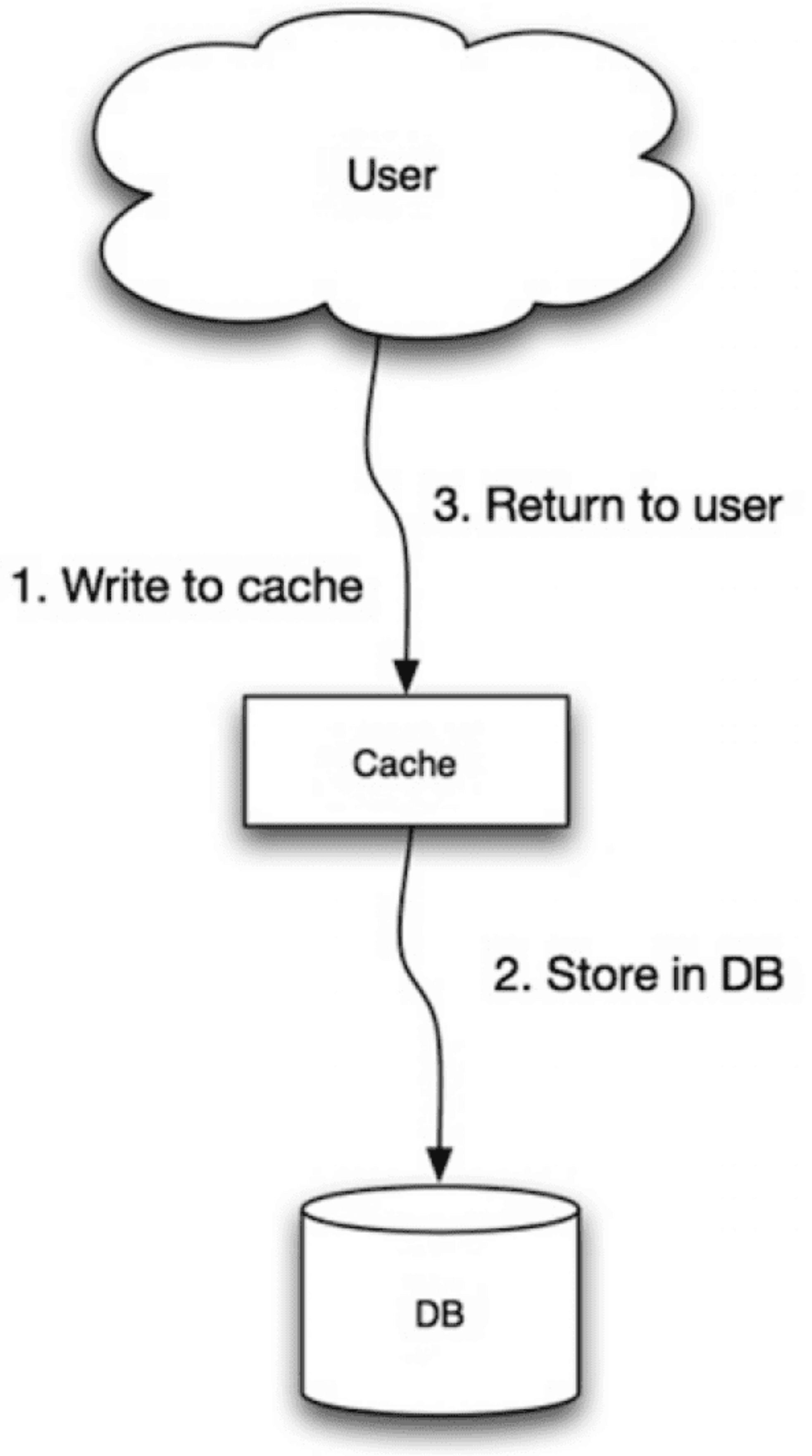
Memcached is generally used in this manner.

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:

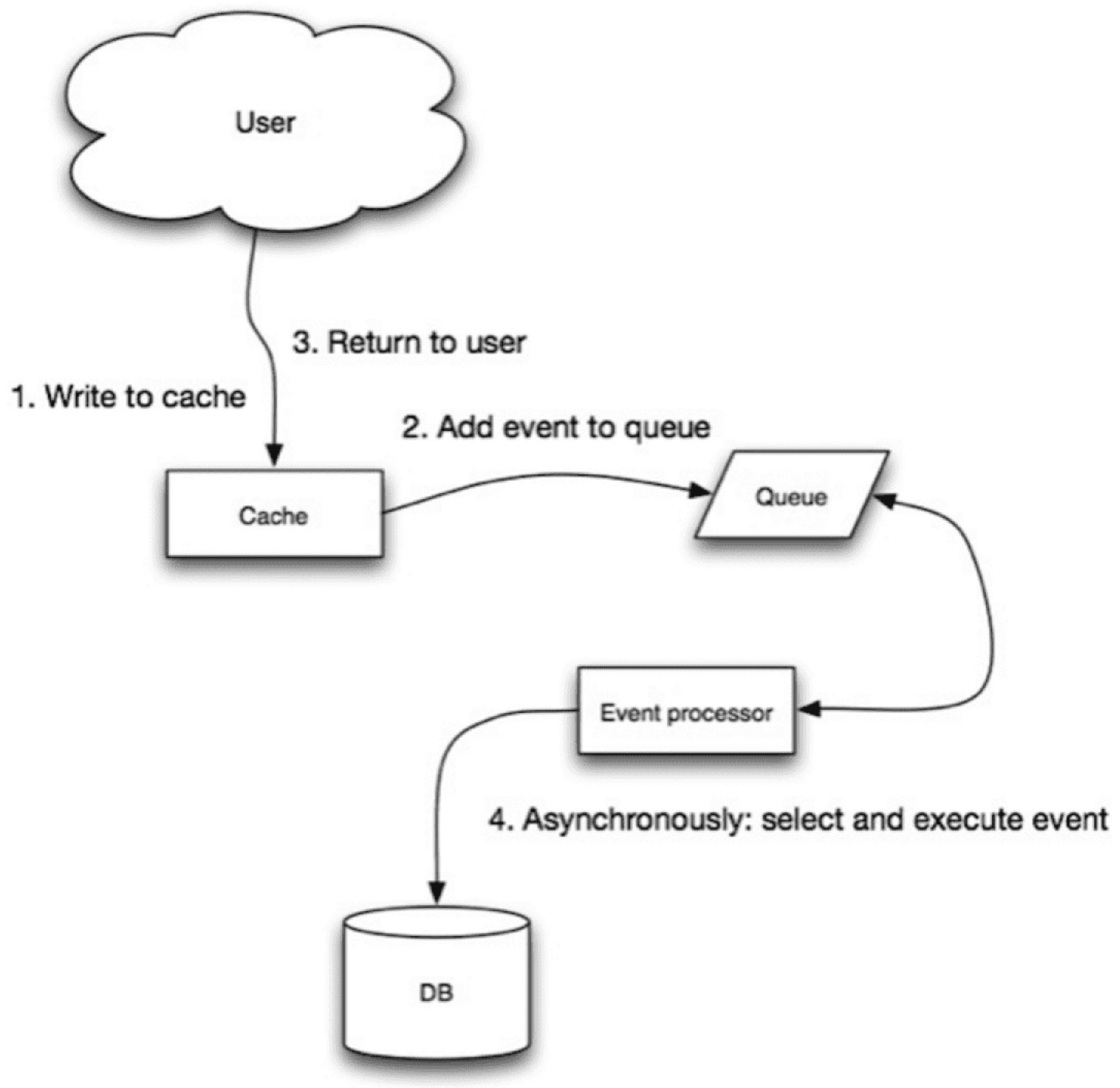
```
def set_user(user_id, values):
    user = db.query("UPDATE Users WHERE id = {0}", user_id, values)
    cache.set(user_id, user)
```

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

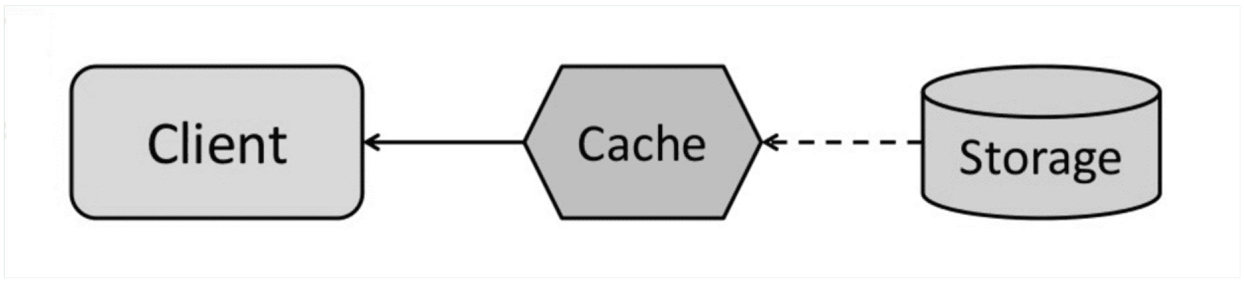
In write-behind, the application does the following:

- 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.

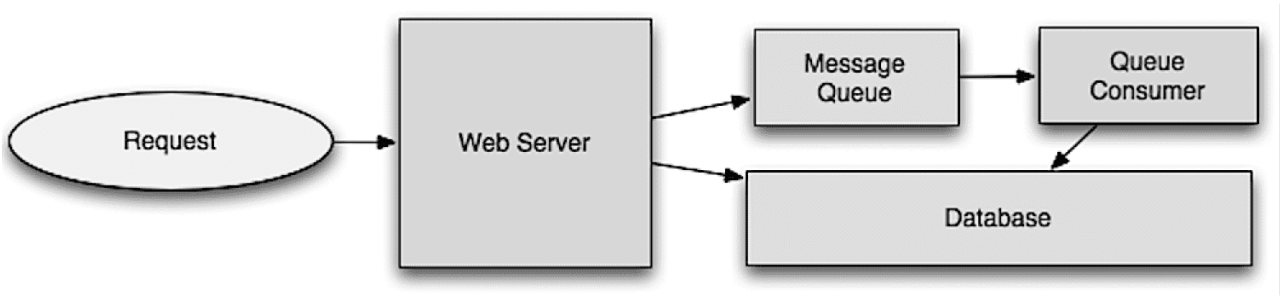
Disadvantage(s): cache

- 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.

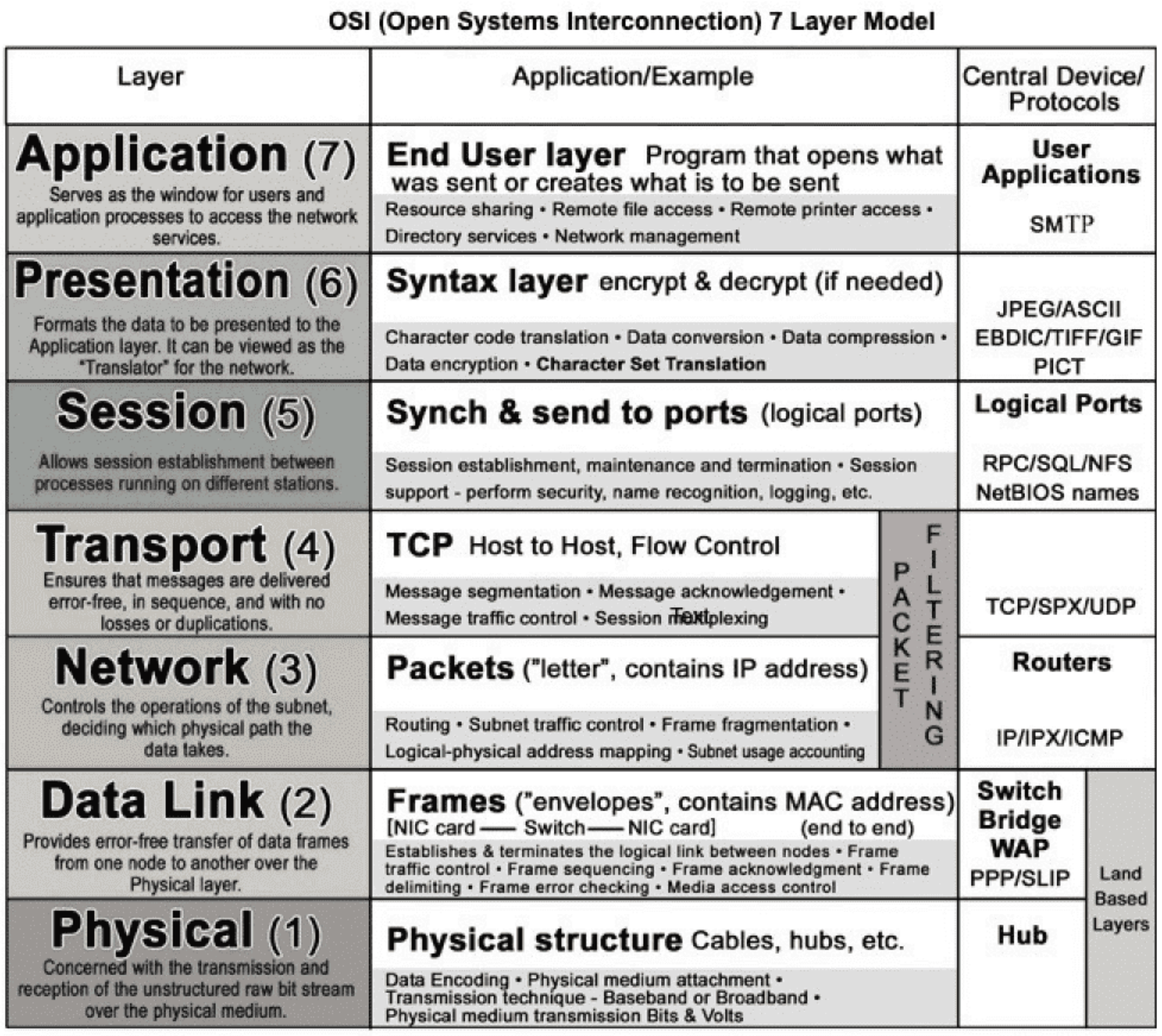
Disadvantage(s): asynchronism

- 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 PACKET FILTERING	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 PACKET FILTERING	**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 — Land Based Layers
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

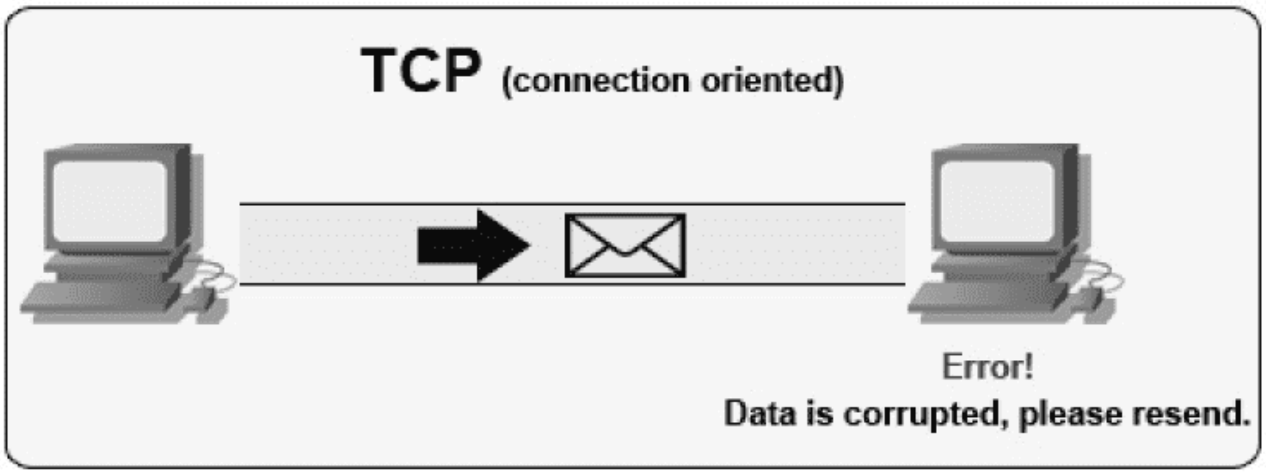
*Can be called many times without different outcomes.

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.

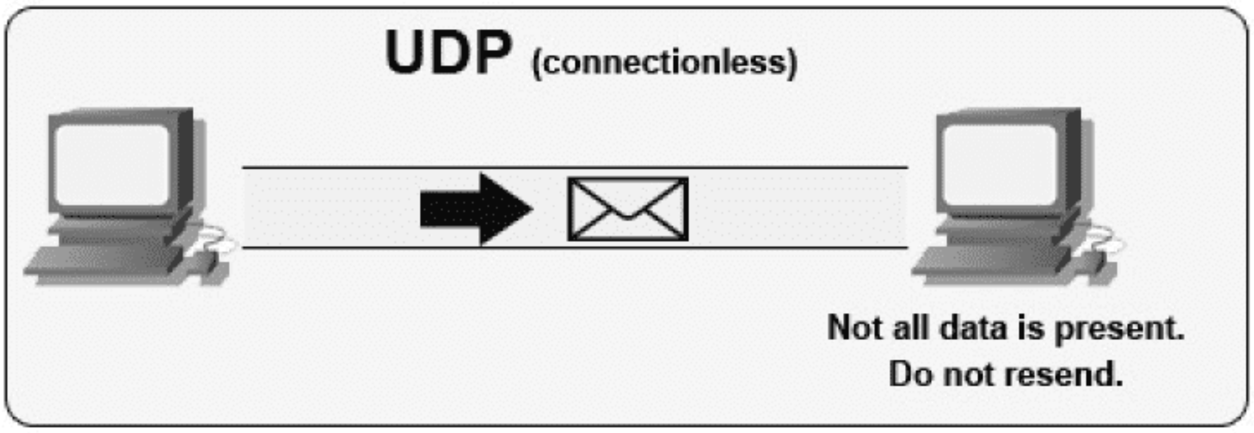
To ensure high throughput, web servers can keep a large number of TCP connections open, resulting in high memory usage. It can be expensive to have a large number of open connections between web server threads and say, a memcached server. Connection pooling can help in addition to switching to UDP where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

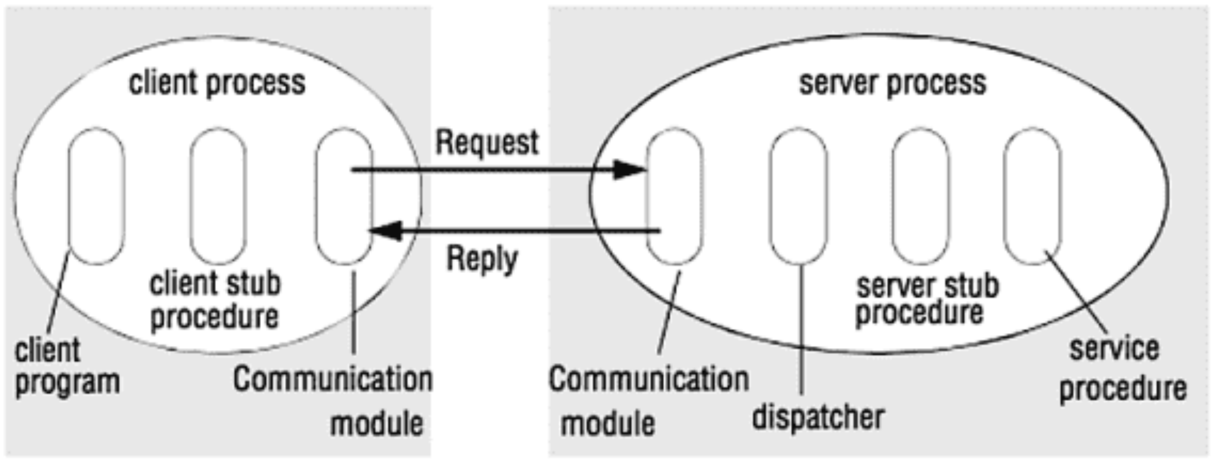
Use UDP over TCP when:

- 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 questions

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 questions
- Thrift
- Why REST for internal use and not RPC

Security

This section could use some updates.

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

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
```

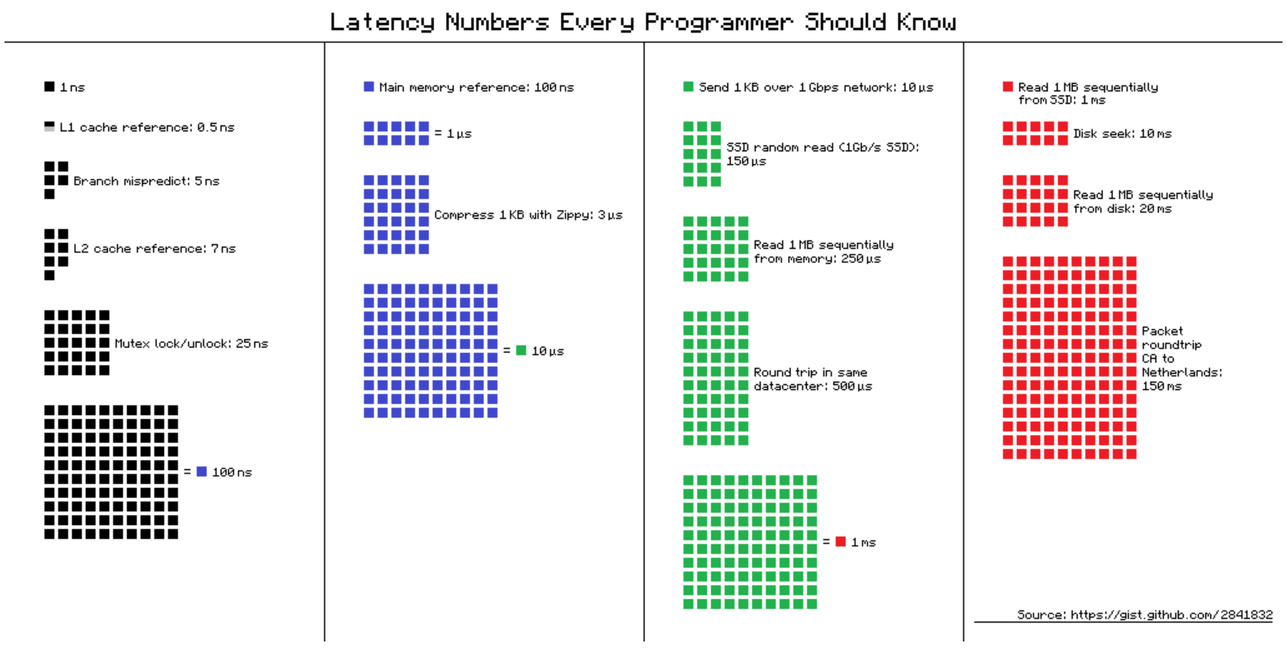
```
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us
1 ms  ~1GB/sec SSD, 4X memory
HDD seek                            10,000,000   ns   10,000 us
10 ms  20x datacenter roundtrip
Read 1 MB sequentially from 1 Gbps  10,000,000   ns   10,000 us
10 ms  40x memory, 10X SSD
Read 1 MB sequentially from HDD     30,000,000   ns   30,000 us
30 ms 120x memory, 30X SSD
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

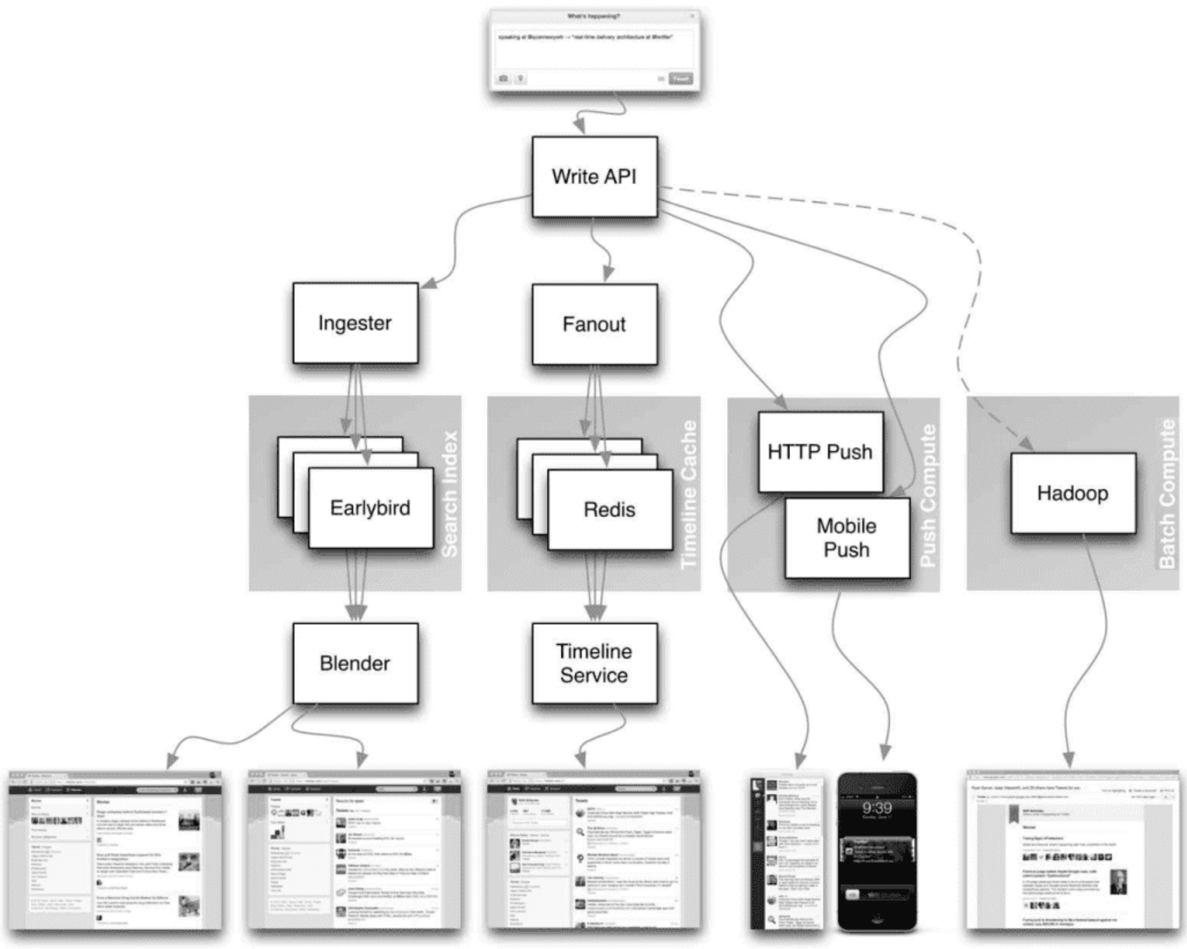
> Common system design questions, with links to resources on how to solve each.

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 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 Implemenation

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

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	**Storm** - Distributed data processing	

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
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

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?
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
WhatsApp	The WhatsApp architecture Facebook bought for $19 billion
YouTube	YouTube scalability YouTube architecture

Company engineering blogs

> Architectures for companies.

- 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
- Twilio Engineering Blog
- Twitter Engineering
- Uber Engineering Blog
- Yahoo Engineering Blog
- Yelp Engineering Blog
- Zynga Engineering Blog

Licenses & certifications

- 🏆 LeetCode Global TOP 300 (Swift: Certificate, Sources: Swift).
- 🏆 Golden Award for the Year of the Tiger Challenge (TypeScript: Certificate, Sources: Codility).
- 🏆 Golden Award Muad'Dib's Challenge (Swift: Certificate, Sources: Swift).
- LinkedIn Skill Asessment (Mobile): Swift (Programming Language), Object-Oriented Programming (OOP), Objective-C, C++, Ionic, JSON, XML, Android, Kotlin, Maven, Java, REST APIs.
- LinkedIn Skill Asessment (Front-End): Front-end Development, Angular, React, Javascript, HTML, CSS, jQuery.
- LinkedIn Skill Asessment (Back-End): Node.js, Java, Spring Framework, Scala, C#, .NET Framework, Unity, Python (Programming Language), Django, PHP, C (Programming Language).
- LinkedIn Skill Asessment (Databases): MongoDB, NoSQL, Transact-SQL (T-SQL), MySQL.
- LinkedIn Skill Asessment (Infra/DevOps): Bash, Git, Amazon Web Services (AWS), AWS Lambda, Google Cloud Platform (GCP), Microsoft Azure, Hadoop, IT Operations.

Contacts

I have a clear focus on time-to-market and don't prioritize technical debt.

🛩 #startups #management #cto #swift #typescript #database

📧 Email: sergey.leschev@gmail.com

👋 LinkedIn: https://linkedin.com/in/sergeyleschev

👋 Twitter: https://twitter.com/sergeyleschev

👋 Github: https://github.com/sergeyleschev

🌎 Website: https://sergeyleschev.github.io

🖨 PDF: Download

ALT: SIARHEI LIASHCHOU