Lecture - Scalability in System Design

How?

- webhosts
  - Go Daddy, Dreamhost, etc
  - features
    - IP range blocking in geographic location
    - secure protocols (SFTP, SSH, HTTPS)
      - personal information
    - some offer unlimited resources
      - this is in the context of a shared machine
- alternatives
  - VPS (Virtual Private Server)
    - Dreamhost, Go Daddy,
    - own copy of OS inside VM
    - run VMs in hypervisors
      - VMWare
      - HyperV
      - Azure
    - hosting company has access to VM
      - may lead to
    - Amazon EC2 (Elastic Cloud Compute)

Types of scaling

- vertical scaling (scale up)
  - increase power of compute via resources
    - CPU
      - cores
      - cache
    - disks
      - PATA (Parallel ATA or IDE, older)
      - SATA (2.5 or 3.5, typically 7200 rpm)
      - SAS (Serial Attached SCSI - spin at 15000 rpm, higher cost)
      - RAID
      - SSDs (solid state drives, no moving parts, faster)

        - memory
    - increase power by going to high-performance computer
    - also allows chopping up
- horizontal scaling (scale out)
    - limited by ceiling of hardware performance
    - commodity hardware
    - increase number of machines to build out topology
    - often inside data centers
    - requires distribution and management of requests

Hardware Fault Tolerance

RAID

- RAID0
    - 2 identical HDs
    - stripes data across to allow multiple interleaved writes to avoid waiting for writes
- RAID1
    - mirror data across drives
- RAID5
    - 3 or more drives with 1 used for redundancy
- RAID6
    - 2 drives for redundancy
- RAID10
    - combines striping and mirroring across 4 drives

Load balancing

- distributes traffic of external requests through to backend servers
    - DNS returns public IP address which routes to load balancer which then services requests through to private IPs of servers that will handle
    - various methods for load balancing
        - round robin
            - may still lead to overloaded servers
        - based on load
        - can just be an internal DNS setup (BIND, etc)
        - layer 4 - based on transport layer information (DNS server information)
        - layer 7 - based on application layer information (HTTP header, etc)
    - need to consider shared global data
        - specifically sessions

- resolve with factoring out shared information into shared service, e.g. DB, fileserver, etc
    - still need to consider fault tolerance of this DB or fileserver
        - hardware fault tolerance
            - drives
            - RAM
            - power supplies
            - all hot swappable
        - replication
        - redundance with HA etc
        - distributed system
- sticky sessions
    - shared storage
        - FC (FibreChannel), iSCSI, DB (e.g. relational) with replication, NFS, etc
    - cookies
        - consider security
        - consider size
        - expiration
        - ID of server
            - possible IP changes
            - exploitation of IP configuration
        - better to store UUID of server which is translated in load balancer

- options
    - software
        - Amazon ELB (Elastic Load Balancer)
        - HAProxy
        - LVS (Linux Virtual Server)
        - Corosync/Pacemaker (Heartbeat)
        - more
    - hardware
        - very overpriced
        - Barracuda
        - Cisco
        - Citrix
        - F5
        - more

Interpreted Languages

- optimization
  - accelerators (specifically for PHP)
- Python .pyc compiled files (wordcode previously bytecode)

Caching

- HTML
  - webpages commonly store dynamic and serve dynamic data via languages like PHP and complex URL encodings
  - can generate static HTML which allows servers like Apache and Nginx to take advantage of static caching
    - drawbacks
      - drain on servers to generate
      - large amount of duplicate generated bits across pages
      - much more difficult to modify content, structure, and styling etc
- SQL (MySQL, etc)
  - MySQL query cache `query_cache_type = 1`
    - if records have not changed, then serves from cache
- memcached
  - software, server
  - stores whatever is needed in RAM
  - can be used in many languages
  - process
    - connect to memcached server
    - request and or store data
  - in some cases, can completely remove database in favor of memcached
  - RAM may run out
    - can use garbage collection
    - can start to write data to disk and reload on usage

Relational DB Optimizations

- NOTE: may be outdated
- storage limits
- transactions
- locking granularity
- MVCC

- geospatial data type support
- geospatial indexing support
- B-tree indexes
- full-text search indexes
- clustered indexes
- data caches
- index caches
- compressed data
- encrypted data
- cluster database support
- replication support
- foreign key support
- backup / point-in-time recovery
- query cache support
- update statistics for data dictionary

Replication

- NOTE: limited scope of just standard relational DBs
- master-slave
  - 3 backups
  - may need to reconfigure to change master (can be automated)
  - can load balance reads across slaves and limit reads to master
  - still has fault tolerance issues as single point of failure of master
- master-master
  - 2 masters and each has slaves
  - can load balance writes and use consistency

Load Balancing + Replication

- use load balancer between backend servers to DB replicas
- load balancers use HA

Partitioning

- split data across various servers and route requests through load balancers based on location of data

High-Availability

- active-active
- active-passive
- uses heartbeats

Firewalls and Security

System Design Primer (https://github.com/donnemartin/system-design-primer)

Performance vs scalability

- scalable - increased performance for system proportional to resources added
- increasing performance
  - serving more units of work
  - handle larger units of work (dataset growth)
- performance problem - system is slow for a single user
- scalability problem - system is fast for a single user but slow under heavy load

Latency vs throughput

- latency - time to perform some action or to produce some result
- throughput - number of actions or results per unit of time
- aim for maximal throughput with acceptable latency

Availability vs consistency

- distributed system can only support two of the following.
  - consistency - every read receives the most recent write or an error
  - availability - every request receives a response
  - partition tolerance - the system continues to operate despite arbitrary partitioning due to network failures
- NOTE: availability is always sacrificed because networks aren't reliable
  - need to make trade-offs between partition tolerance, consistency and availability
  - CAP theorem is not adequate to describe the current understanding and is overused

Consistency patterns

- weak consistency
  - reads may or may not see the most recent write
  - a best effort approach is taken
  - used in memcached and more
  - works well in real time use cases
    - voip
    - video chat
    - realtime multiplayer games
- eventual consistency
  - reads will eventually see most recent write (typically within milliseconds)
  - data is replicated asynchronously
  - used in systems such as
    - dns
    - email
  - works well in highly available systems
- strong consistency
  - reads always see most recent write
  - data is replicated synchronously
  - used in
    - filesystems
    - many relational DBs
  - works well in systems that need transactions

Availability patterns

Fail-over

Active-passive

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

Active-active

- both servers manage traffic and spread the load between them
- if servers are public-facing, DNS needs to know about the public IPs of both servers
- if servers are internal, application logic needs to know about both servers
- also referred to as master-master failover

Disadvantages of 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
- many more replication strategies than listed here

Domain name system (DNS)

- translates a domain name such as www.example.com to an IP address
- hierarchical, with a few authoritative servers at the top level
- 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.
- many services (cloudfare dns and amazon route53)

- various methods for routing:
    - weighted round robin
        - prevent traffic from going to servers under maintenance
        - balance between varying cluster sizes
        - a/b testing
    - latency-based
    - geolocation-based

Disadvantages of DNS

- accessing a dns server introduces a slight delay, although mitigated by caching described above.
- dns server management could be complex, although they are generally managed by governments, isps, and large companies
- dns services have recently come under DDoS attack
    - prevented users from accessing websites such as twitter without knowing twitter's ip address(es)

Content delivery network

- globally distributed network of proxy servers, serving content from locations closer to the user
- static files such as HTML/CSS/JS, photos, and videos are served from CDN
- some CDNs such as Amazon's CloudFront support dynamic content
- the site's DNS resolution will tell clients which server to contact
- improves performance by:
    - users receive content at data centers close to them
    - your servers do not have to serve requests that the CDN fulfills

Push CDNs

- receive new content whenever changes occur on your server
- client responsible for providing content, uploading directly to the CDN and rewriting URLs to point to the CDN
- 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

Pull CDNs

- grab new content from client server when the first user requests the content

- leave the content on your server and rewrite URLs to point to the CDN
- results in a slower request until the content is cached on the CDN
- pull CDNs minimize storage space on the CDN, but can create redundant traffic if files expire and are pulled before they have actually changed

Disadvantages of CDNs

- CDN costs can be significant depending on traffic
- content might be stale if it is updated before the TTL causes it to expire
- CDNs require changing URLs for static content to point to the CDN

Load balancer

- distribute incoming client requests to computing resources such as application servers and databases
- returns the response from the computing resource to the appropriate client
- effective at:
  - preventing requests from going to unhealthy servers
  - preventing overloading resources
  - helping eliminate single points of failure
- 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
- common to set up multiple load balancers to protect against failures
  - active-passive
  - active-active
- can route traffic based on various metrics
  - random
  - least loaded
  - session/cookies
  - round robin or weighted round robin
  - layer 4
    - monitor info at the transport layer to decide how to distribute requests
    - involves the source, destination IP addresses, and ports in the header, but not the contents of the packet

        - forward network packets to and from the upstream server, performing Network Address Translation (NAT)
    - layer 7
        - monitor the application layer to decide how to distribute requests
        - can involve contents of the header, message, and cookies
        - terminates network traffic, reads the message, makes a load-balancing decision, then opens a connection to the selected server
        - layer 4 load balancing requires less time and computing resources than Layer 7
            - sacrifices flexibility
            - the performance impact can be minimal on modern commodity hardware
- can also help with horizontal scaling
    - scaling out using commodity machines is more cost efficient and results in higher availability than scaling up a single server on more expensive hardware (**vertical scaling**)
    - easier to hire for talent working on commodity hardware than it is for specialized enterprise systems

Disadvantages of horizontal scaling

- scaling horizontally introduces complexity and involves cloning servers
    - servers should be stateless
        - 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

Disadvantages of load balancers

- 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 single points of failure results in increased complexity
- a single load balancer is a single point of failure
- configuring multiple load balancers further increases complexity

Reverse proxy (web server)

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

- load balancer - useful with multiple servers
  - often 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
- NGINX and HAProxy can support both layer 7 reverse proxying and load balancing.

Disadvantages of a reverse proxy

- results in increased complexity
- single point of failure
- configuring multiple reverse proxies (i.e. HA pair for failover) further increases complexity

Application layer

- separating out the web layer from the application layer (also known as platform layer) allows independent scaling and configuration
  - modifying APIs results in adding application servers without necessarily adding additional web servers

- **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

- a collection 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
- e.g. Pinterest
    - user profile
    - follower
    - feed
    - search
    - photo upload
    - etc.

Service Discovery

- systems/services that can help other services find each other by keeping track of registered names, addresses, and ports
    - Consul
    - etcd
    - Zookeeper
- use health checks to help verify service integrity
    - 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

Disadvantages of decoupling the 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

Database

Relational database management system (RDBMS)

- collection of data records 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
- many techniques to scale
    - **master-slave replication**
    - **master-master replication**
    - **federation**
    - **sharding**
    - **denormalization**
    - **SQL tuning**

Master-slave replication

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

Disadvantages of master-slave replication

- additional logic is needed to promote a slave to a master

Master-master replication

- both masters serve reads and writes and coordinate with each other on writes
- the system can continue to operate with both reads and writes if either master goes down

Disadvantages of master-master replication

- need a load balancer or need to make changes to 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

Disadvantages of 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
  - read replicas only support writing sequentially with a single thread.
- replication adds more hardware and additional complexity

Federation

- (functional partitioning) splits up databases by function
  - e.g. instead of a single, monolithic database could have three databases: **forums** , **users** , and **products**
  - results 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 can write in parallel, increasing throughput

Disadvantages of federation

- federation is not effective if schema requires huge functions or tables
- 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

Sharding

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.

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.

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.

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.

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 in either 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.

Wide column store

> 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.

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 .

SQL or 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

Cache

Caching improves page load times and can reduce the load on your servers and databases. In this model, the dispatcher will first lookup if the request has been made before and try to find the previous result to return, in order to save the actual execution.

Databases often benefit from a uniform distribution of reads and writes across its partitions. Popular items can skew the distribution, causing bottlenecks. Putting a cache in front of a database can help absorb uneven loads and spikes in traffic.

Client caching

Caches can be located on the client side (OS or browser), server side , or in a distinct cache layer.

CDN caching

CDNs are considered a type of cache.

Web server caching

Reverse proxies and caches such as Varnish can serve static and dynamic content directly. Web servers can also cache requests, returning responses without having to contact application servers.

Database caching

Your database usually includes some level of caching in a default configuration, optimized for a generic use case. Tweaking these settings for specific usage patterns can further boost performance.

Application caching

In-memory caches such as Memcached and Redis are key-value stores between your application and your data storage. Since the data is held in RAM, it is much faster than typical databases where data is stored on disk. RAM is more limited than disk, so cache invalidation alg orithms 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** an d **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

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

- Look for entry in cache, resulting in a cache miss
- Load entry from the database
- Add entry to cache
- Return entry

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

Memcached is generally used in this manner.

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

Disadvantage(s): cache-aside

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

Write-through

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

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

- Return

Application code:

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

Cache code:

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

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

Disadvantage(s): write through

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

Write-behind (write-back)

In write-behind, the application does the following:

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

Disadvantage(s): write-behind

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

Refresh-ahead

You can configure the cache to automatically refresh any recently accessed cache entry prior to its expiration.

Refresh-ahead can result in reduced latency vs read-through if the cache can accurately predict which items are likely to be needed in the future.

Disadvantage(s): refresh-ahead

- Not accurately predicting which items are likely to be needed in the future can result in reduced performance than without refresh-ahead.

Disadvantage(s): cache

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

Asynchronism

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.

Communication

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** .

Transmission control protocol (TCP)

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)

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 b ecause 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

Remote procedure call (RPC)

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

Security

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 .

Additional Notes

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
```

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 ca
Mutex lock/unlock                          100   ns
Main memory reference                      100   ns                      20x L2 ca
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
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
Disk seek                           10,000,000   ns   10,000 us   10 ms  20x datac
Read 1 MB sequentially from 1 Gbps  10,000,000   ns   10,000 us   10 ms  40x memor
Read 1 MB sequentially from disk    30,000,000   ns   30,000 us   30 ms 120x memor
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 disk 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
- Istio - service mesh for managing microservices
- alternatives to Kubernetes
    - Prometheus
    - other products inherited from Borg

NAS LVM (Logical Volume Manager)

NOTE: general overview with more to do here

Replication

- leader-based replication (active/passive or master-slave)
- multi-leader (master-master or active/active)
- leaderless
- see also:
    - conflict-free replicated data types (CRDTs)

Architecture

- client/server
  - client/server with multiple servers or load-balanced servers (proxy)
- peer-to-peer
- client-queue-client

HA cluster modes

- active/active - always active on all nodes, service passed to any when using load balancing, with no load balancing just used to reduce failover time
- active/passive - 2 or more, only one master at a time
- n + 1 - 2 or more, when 2 degenerates to active/passive, n active, 1 passive for failover
- n + m - > 2, n active, m passive, used for many members to manage active services with two or more failover nodes
- n-to-1 - 2 or more, degenerates to active/passive with 2, uses the single standby only during recover of a failed active node
- n-to-n - > 2, similar to n-to-1 with one recovery node for each active node

OSI Layers

1. physical
2. data-link
3. network
4. transport
5. session
6. presentation
7. application

Architectural Models

- layered - abstractions sit on top of each other
- tiered - presentation (user interaction), application (business logic), data logic (storage)
- two tiered - user interaction layer communicates with server that handles app and data
- three tiered - user interacts with application which interacts with a database
- thin client
- proxy

- brokerage
- reflection
- (middleware)
- monolith
- microservices

Communication

- socket -> (tcp or udp)
- marshalling -> convert a collection of data into a suitable form for message passing (serialization)
- unmarshalling -> disassembling
- corba, xml, java serialization, json, protocol buffers
- multicast
- network virtualization - overlay networks

Remote Invocation

- request-response - structured messages
- rpc - requires explicit interfaces
- rmi (remote method invocation) - like rpc but uses remote objects with unique object references

Indirect Communication

- group communication - message is sent to a group and the message is delivered to all members of the group (abstraction over multicast) (one-to-many)
- publish-subscribe - publish structured events to an event service which delivers (one-to-many)
- message queues - point-to-point to achieve space and time uncoupling
- shared memory - distributed shared memory
- tuple space communication

EIP

Integration Criteria

- coupling

- simplicity
- technology
- data format
- data timeliness
- data/functionality
- asynchronicity

Integration Options

- file transfer
- shared database
- RPC
- messaging

Patterns

Message Construct

- Message
- Command Message
- Document Message
- Event Message
- Request-Reply
- Return Address
- Correlation Identifier
- Message Sequence
- Message Expiration
- Format Indicator

Message Routing

- Pipes-and-Filters
- Message Router
- Content-based Router
- Message Filter
- Dynamic Router
- Recipient List
- Splitter
- Aggregator
- Resequencer

- Composed Msg. Processor
- Scatter-Gather
- Routing Slip
- Process Manager
- Message Broker

Message Transformation

- Message Translator
- Envelope Wrapper
- Content Enricher
- Content Filter
- Claim Check
- Normalizer
- Canonical Data Model

Messaging Endpoints

- Message Endpoint
- Messaging Gateway
- Messaging Mapper
- Transactional Client
- Polling Consumer
- Event-driven Consumer
- Competing Consumers
- Message Dispatcher
- Selective Consumer
- Durable Subscriber
- Idempotent Receiver
- Service Activator

Messaging Channels

- Message Channel
- Point-to-Point Channel
- Publish-Subscr. Channel
- Datatype Channel
- Invalid Message Channel
- Dead Letter Channel
- Guaranteed Delivery
- Channel Adapter

- Messaging Bridge
- Message Bus

Systems Mgmt

- Control Bus
- Detour
- Wire Tap
- Message History
- Message Store
- Smart Proxy
- Test Message
- Channel Purger

Consensus (Replication protocols)

- Paxos (also cheap Paxos and fast Paxos)
- Raft
- Chandra-Toueg
- Lockstep protocol
- MSR-type algorithms
- hashgraph
- Zab (ZooKeeper Atomic Broadcast)
- viewstamped replication
- see also:
  - gossip
- bitcoin-related consensus
  - general goals
    - ensures next value (blockchain) is the single source of truth
    - keeps competing agents from modifying the set value in the context
  - proof of work*
    - proven by solving a cryptographic problem
  - proof of stake
    - proven by investing in unit of system
  - proof of activity
    - combines proof of work and proof of stake
  - proof of burn
    - commit units of system to an unusable state
  - proof of capacity
    - given priority by agent's memory space

    - proof of elapsed time

Message Brokers

- Kafka
- Google Cloud Pub/Sub
- Amazon Kinesis
- Simple Queueing Service
- NOTE: when considering Thrift vs. ZeroMQ
    - Thrift -> designed for use with streaming protocol (TCP)
    - ZeroMQ -> designed for messaging
    - consider which applications are better suited to which