OPERATING SYSTEMS CONCEPTS
TOP 80+ (Interview Questions)

Q1. What is the difference between Thread and Process?
Ans1.

Process	Thread
An executing instance of a program is called a process.	A thread is a subset of the process.
It has its own copy of the data segment of the parent process.	It has direct access to the data segment of its process.
Processes must use inter-process communication to communicate with sibling processes.	Threads can directly communicate with other threads of its process.
Processes have considerable overhead.	Threads have almost no overhead.
New processes require duplication of the parent process.	New threads are easily created.
Processes can only exercise control over child processes.	Threads can exercise considerable control over threads of the same process.
Any change in the parent process does not affect child processes.	Any change in the main thread may affect the behavior of the other threads of the process.

Run in separate memory spaces.	Run in shared memory spaces.
Most file descriptors are not shared.	It shares file descriptors.
There is no sharing of file system context.	It shares file system context.
It does not share signal handling.	It shares signal handling.
Process is controlled by the operating system.	Threads are controlled by programmer in a program.
Processes are independent.	Threads are dependent.

Q2. What is Caching?

Ans. A cache's primary purpose is to increase data retrieval performance by reducing the need to access the underlying slower storage layer. Trading off capacity for speed, a cache typically stores a subset of data transiently. It can be applied for low latency and high throughput through various technologies: -

Acting as an adjacent data access layer, In-Memory Cache is especially relevant for read-heavy application workloads, such as Q&A portals, gaming, media sharing, and social networking, and recommendation engines obtained from the results of database

queries, computationally intensive calculations, API requests/responses, HTML or image files.

For example,

- **Amazon CloudFront** is a global CDN service that accelerates delivery of websites, APIs, web assets by delivering a cached copy of web content such as videos, webpages, images to the client.

- Every domain request made on the internet essentially queries **DNS** cache servers in order to resolve the IP address associated with the domain name.

- Depending on the expected load on the API and the rate of change of underlying data, sometimes serving a cached result of the API will deliver the most optimal response time and cost-effective response. Also, it eliminates pressure to your infrastructure including the application servers and databases.

- Client side web caching can include browser based caching which retains a cached version of the previously visited web content.

Since **local caches** only benefit the local application consuming the data, hence, a **distributed caching environment** is preferred. In a distributed caching environment, the cache serves as a central layer that can be accessed from disparate systems while the data spans through multiple cache servers and be stored in a central location for the benefit of all the consumers of that data. A **global cache** is just as it sounds: all the nodes use the same single cache space, while in case of a cache miss, global cache itself is responsible for retrieval of data from database as well as eviction of outdated data.

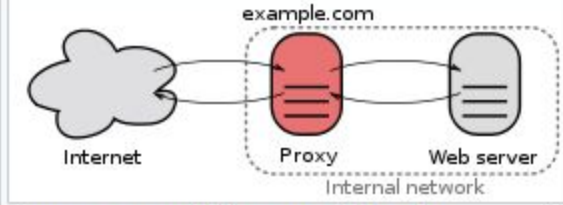
In many applications, it is likely that a small subset of data, such as a celebrity profile or popular product, will be accessed more frequently than the rest. This can result in **Database Hot Spots** in your database which is usually handled by overprovisioning of the database resources i.e., the inclusion of extra storage space for the most frequently used data.

Instead of overprovisioning of the data, storing common keys in an in-memory cache mitigates the need to overprovision while providing fast retrievals to the most commonly accessed data.

Q3. What is a proxy server?

NEXT

Ans.

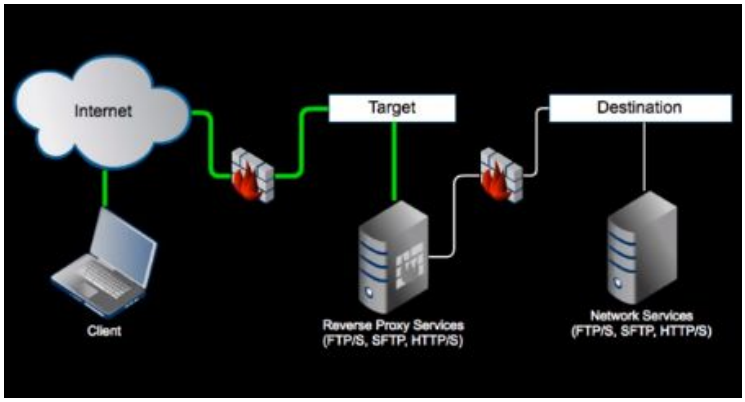


A reverse proxy taking requests from the Internet and forwarding them to servers in an internal network. Those making requests connect to the proxy and may not be aware of the internal network.

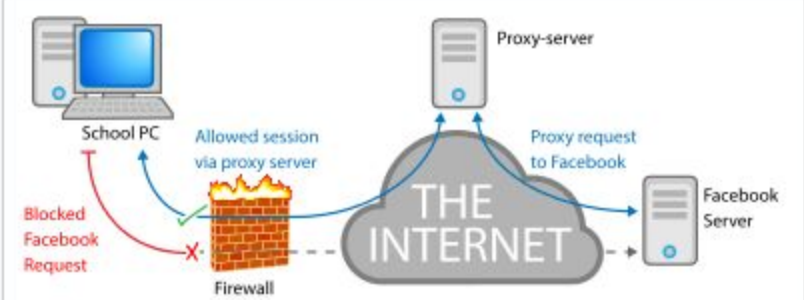
Figure 1 Reverse Proxy Server

When a proxy server receives a request for an Internet resource (such as a Web page), it looks in its local cache of previously pages. If it finds the page, it returns it to the user without needing to forward the request to the Internet. If the page is not in the cache, the proxy server, acting as a client on behalf of the user, uses one of its own IP addresses to request the page from the server out on the Internet. When the page is returned, the proxy server relates it to the original request and forwards it on to the user

An anonymous open proxy allows users to conceal their IP address while browsing the Web or using other Internet services. Users access forward proxies by directly surfing to a web proxy address or by configuring their Internet settings.



On the other hand, the reverse proxy sits closer to the web server and serves only a restricted set of websites. Requests are forwarded to one or more proxy servers which handle the request. The response from the proxy server is returned as if it came directly from the original server, leaving the client no knowledge of the origin servers. Reverse proxies are installed in the neighborhood of one or more web servers. All traffic coming from the Internet and with a destination of one of the neighborhood's web servers goes through the proxy server.



Many schools block access to popular websites such as Facebook. Students can use proxy servers to circumvent this security. However, by connecting to proxy servers, they might be opening themselves up to danger by passing sensitive information such as personal photos and passwords through the proxy server. Some content filters block proxy servers in order to keep users from using them to bypass the filter.

Use of Proxies –

1. **Content-filtering web proxy server** used by commercial and non-commercial organizations to control the content relayed in or out to ensure that Internet usage conforms to acceptable use policy.

2. **Content Filtering Proxy -** Governments censor undesirable content by restricting the web sites and online services that are accessible and available using Content filters.

3. **To circumvent filters** and access blocked content, often using a proxy, from which the user can then access the websites that the filter is trying to block.

4. **Blacklists** are often provided and maintained by web-filtering companies, often grouped into categories (pornography, gambling, shopping, torrents, social networks, etc.).

5. **Web Crawlers** - Some proxies scan outbound content, e.g., for data loss prevention; or scan content for malicious software.

Reasons for Using Reverse Proxies –

A. **Encryption –** when secure web sites are created, the **Secure Sockets Layer** (SSL) encryption is often not done by the web server itself, but by a reverse proxy that is equipped with SSL acceleration hardware.

B. **Load Balancing** – The reverse proxy can distribute the load to several web servers, each web server serving its own application area. In such a case, the reverse proxy may need to rewrite the URLs in each web page translating from externally known URLs to the internal locations.

C. **Cache Static Content –** A reverse proxy can offload the web servers by caching static content like pictures and other static graphical content.

D. **Compression –** The proxy server can optimize and compress the content to speed up the load time.

E. **Spoon-feeding –** reduces resource usage caused by slow clients on the web servers by caching the content the web server sent and slowly "spoon feeding" it to the client. This especially benefits dynamically generated pages.

F. **Security -** the proxy server is an additional layer of defense and can protect against some OS and Web Server specific attacks.

G. **Extranet Firewall –** a reverse proxy server facing the Internet can be used to communicate to a firewall server internal to an organization, providing extranet access to some functions while keeping the servers behind the firewalls. However, in case this server is compromised, since its web application is exposed to attack from the Internet, ensure the rest of the infrastructure behind remains safe.

How to Detect a Proxy- By comparing the client's external IP address to the address seen by an external web server. In case there is a mismatch, Proxy and owner of the IP address both can be detected and blocked.

Types of Proxies –

1. **Transparent Proxies -** These proxies centralize network traffic on corporate networks. Here, proxies separate the network from external network/Internet. It acts as a firewall that protects the network from outside intrusion and allows data to be scanned for security purposes before delivery to a client on the network.
   These proxies help with monitoring and administering network traffic that any data that enters or leave the computers from the corporate to ensure sender ownership in case of malicious content.

2. **Anonymous Proxies –** These proxies hide the IP address of the client using them which allows them to access materials that are blocked by firewalls or to circumvent IP address bans with enhanced privacy.

3. **Highly Anonymous Proxies –** Like TOR or the onion routing, which is a web browser designed for anonymous web surfing and protection against traffic analysis. These proxies not only hide the IP addresses of the client using them, but also the

fact that they are being used by clients and present a non-proxy public IP address

4. Caching Proxies – These proxy servers accelerate service requests by retrieving content saved from a previous request made by the same client or even other clients. Caching proxies keep local copies of frequently requested resources. This causes large organizations to significantly reduce their upstream bandwidth usage and costs, while significantly increasing performance.

Proxy Hacking - In proxy hacking, an attacker attempts to steal hits from an authentic web page in a search engine's index and search results pages. The proxy hacker would have either a fraudulent site emulating the original or whatever they felt like showing the clients requesting the page.

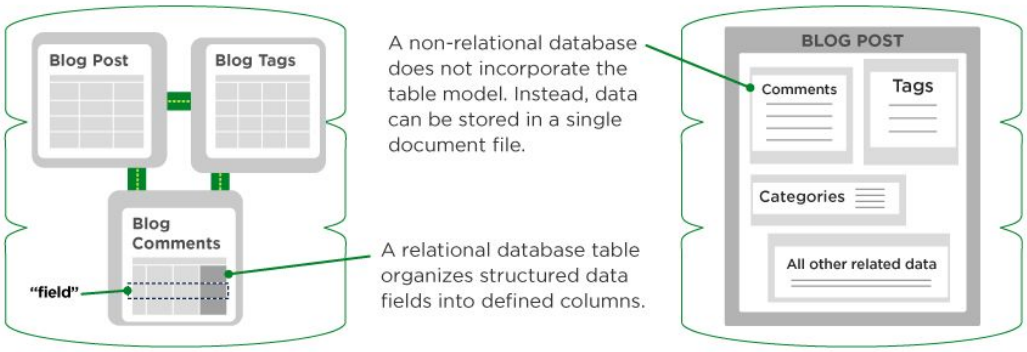
The attacker creates a copy of the targeted web page on a proxy server and uses methods such as keyword stuffing and linking to the copied page to artificially raise its search engine ranking. The authentic page will rank lower and may be seen as duplicated content, in which case a search engine may remove it from its index.

Proxy hacking can direct users to fake banking site, for example, to steal account info which can then be sold or used to steal funds from the account. The attacker can also use the hack to direct users to a malware-infected site to compromise their machines for a variety of nefarious purposes

To check whether a website has been proxy hacked, check the SERP or search engine results page with a keyword uniquely identifying the website, if a second site with the same content shows up, it may be a proxy page.

Q4. What is the difference between SQL (Relational) VS No-SQL (Non-Relational) databases?

Ans.



- **SQL is the more rigid, organized and structured way of storing data. E.g., MS-SQL, Oracle, MySql, PostgreSQL**

- **It consists of two or more tables with columns and rows.**
- **Each row represents an entry, and each column sorts a very specific type of information, like a name, address, and phone number.**
- **The relationship between tables and field types is called a schema.**
- **In a relational database, the schema must be clearly defined before any information can be added.**
- **A well-designed schema minimizes data redundancy.**

Reasons to Choose Relation Databases: -

- **You need to ensure ACID compliancy (Atomicity, Consistency, Isolation, Durability).**
- **ACID compliancy reduces anomalies and protects the integrity of your database by prescribing exactly how transactions interact with the database, which is usually compromised for flexibility and speed in No-SQL.**
- **In case your data is structured and unchanging, and the data you work remains consistent then you may not need to switch to a fast and flexible means.**

No-SQL Database: -

- Non-relational databases are more like file folders, assembling related information of all types. For example, each file could store data for a blog post: social likes, photos, text, metrics, links, and more.
- No-SQL is helpful, if dealing with massive amounts of unstructured data, or when the requirements aren't clear.
- No-SQL is a schema less alternative that allows you to treat data more flexibly.
- It is specifically used to store, process, and analyze unstructured data, from the web including photos, location-based information, online activity, usage metrics etc.

- Instead of tables, NoSQL databases are document-oriented. This way, non-structured data (such as articles, photos, social media data, videos, or content within a blog post) can be stored in a single document that can be easily found but isn't necessarily categorized into fields like a relational database does.

- No-SQL databases provide ease of access, through APIs, which allow developers to execute queries without having to learn SQL or understand the underlying architecture of their database system.

Popular NoSQL Databases: -

- Riak, BigTable, Azure, Oracle NoSQL etc.

- MongoDB - The most popular open source NoSQL system, especially among startups. A document-oriented database with JSON-like documents in dynamic schemas instead of relational tables that's used on the back end of sites like Craigslist, eBay, Foursquare.

- HBase – An open source Apache project, developed as a part of Hadoop, NoSQL DB, written in Java, with BigTable-like capabilities.

- **Cassandra DB -** Cassandra is a distributed database that's great at handling massive amounts of structured data. Cassandra is great at scaling up. Instagram, Comcast, Apple, and Spotify.

Reasons to Use No-SQL

- **No-SQL is accessed through APIs,** which allow developers to execute queries without having to learn SQL or understand the underlying architecture of their database system.

- **Storing large volumes of data that often have little to no structure**, with document-based databases, No-SQL allows to store data in one place without having to define what "types" of data those are in advance.

- **Making the most of cloud computing and storage.** Cloud-based storage is an excellent cost-saving solution, but requires data to be easily spread across multiple servers to scale up. NoSQL databases like Cassandra are designed to be scaled across multiple data centers out of the box without a lot of headaches.

- **Rapid development** and frequent updates to the data structure becomes seamless and that too without a lot of downtime between versions using a No-SQL database while a relational database slows down.

Q 5. What is Data Partitioning?

Ans.

Data partitioning aka sharding is a technique to break up a big database (DB) into many smaller parts across multiple machines, in order to improve

the manageability, performance, availability and load balancing of an application. Also, it is much cheaper and simpler to scale horizontally than vertically by bulkier servers.

Types of Partitioning Methods: -

A. **Horizontal Partitioning –** or Range based sharding, entails storing different ranges of data in separate locations. For example, zip codes with less than 250000 are stored at one location while records/rows above 250000 is stored at another location. The key problem with this approach is that this may lead to unbalanced serves if the range-value chosen for partitioning is not chosen carefully. That is, a range may encounter way more number of records that another range due to another factor.

B. **Vertical Partitioning –** In this scheme, we divide our data to store tables related to a specific feature to their own server. For example, we can place user profile information on one DB server, friend lists on another and photos uploaded by users on a third server. The key problem with this approach is that if the application experiences additional growth, then it may entail further partitioning a feature specific DB across various servers.

C. **Directory Based Partitioning –** In this partitioning approach, we query our directory server that holds the mapping between each tuple key to its DB server. Here, we create A lookup service which knows your current partitioning scheme to find out where does a particular data entity resides. This creates an abstraction from the rest of the DB Access code, which allows us to add servers to the DB pool or change the partitioning scheme without having to impact the application. The key problem with this approach is it creates a new single point of failure.

PARTITIONING CRITERIA: -

I. **List Partitioning** – Each partition is assigned a list of values say, name of countries, so whenever we want to insert a new record, we will see which partition contains our key and then store it there. For example, all users living in Iceland, Norway, Sweden, Finland or Denmark gets stored in a partition for the Nordic countries.

II. **Key/Hash Based partitioning –** Under this scheme, we apply a hash function to some key attribute of the entity we are storing, that yields the partition number. For example, if we have 100 DB servers and our ID is a numeric value that gets incremented by one, each time a new record is inserted, the hash function could be 'ID % 100', which will give us the server number where we can store/read that record.

Advantage - uniform allocation of data among servers.
Disadvantage - Fixes the total number of DB servers, thus adding new servers means changing the hash function, which requires redistribution of data and downtime for the service.

III. **Round-Robin Partitioning -** This is a very simple strategy that ensures uniform data distribution. With 'n' partitions, the 'i' tuple is assigned to partition (i mod n).

IV. Composite Partitioning - Under this scheme, we combine any of above partitioning schemes to devise a new scheme. For example, composite of hash and list partitioning where the hash reduces the key space to a size that can be listed.

Common Constraints and Additional Complexities on a Sharded Database: -

a. **Joins and Denormalization –** Performing joins on a sharded databases is not feasible. Plus, such joins are not performance efficient since data has to be compiled from multiple servers. A common workaround for this problem is to denormalize the database so that queries that previously required joins can be performed from a single table. However, this incurs an additional problem of data inconsistency due to Denormalization.

b. **Referential Integrity –** Referential Integrity means that the foreign key in any referencing table must always refer to a valid row in the referenced table and ensures that the relationship between two such tables remain consistent and synchronized during updates and deletes. enforce such integrity constraints is extremely difficult on a sharded database and often has to be regularly enforced in application code.

Rebalancing –It can be of two types, (1) The data distribution is not uniform, meaning there may be too many records for a particular partition to store while other may have vast amount of empty space. (2) There may be a lot of load on a specific shard handling user photo requests.

To handle these problems, either we can create more DB shards or have to rebalance existing shards, which means the partitioning scheme has to be changed and all the existing data has to be moved to new locations. Doing this without incurring downtime is extremely difficult.

Q6. Describe Database Indexing?

Ans. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

Indexing technology enables sub-linear time lookup to improve performance, as linear search is inefficient for large databases. Many index designs exhibit logarithmic (O(log(N))) lookup performance and in some applications it is possible to achieve flat (O(1)) performance.

Indices can be implemented using a variety of data structures including balanced trees, B+ trees and hashes.

The order that the index definition defines the columns in is important. In case of composite indexes created on the columns, say {city, first name, last name}, to improve the performance and keep sequential lookups to the minimal, one must ensure that the index is created on the order of search columns.

With an index the database simply follows the B-tree data structure until the entry has been found; this is much less computationally expensive than a full table scan.

COVERING INDEX - the index is only used to locate data records in the table and not to return data. However, a covering index is a special case where the

index itself contains the required data field(s) and can answer the required data. E.g., to find the Name for ID 13, an index on (ID) is useful, but the record must still be read to get the Name. However, an index on (ID, Name) contains the required data field and eliminates the need to look up the record.

TYPES OF INDEXES –

A. **BITMAP INDEX** – A bitmap index is a special kind of indexing that stores the bulk of its data as bit arrays or bitmaps and answers most queries by performing bitwise logical operations on these bitmaps. the bitmap index is designed for cases where the values of a variable repeat very frequently or have modest number of distinct values. For example, the gender field in a customer database contain utmost three values Male, Female, or Unknown. For such variables, the bitmap index can have a significant performance advantage over the commonly used trees.

B. **DENSE INDEX –** A dense index in databases is a file with pairs of keys and pointers for every record in the data file. Every key in this file is associated with a particular pointer to *a record* in the sorted data file.

C. **SPARSE INDEX –** A sparse index in databases is a file with pairs of keys and pointers for every block in the data file. Every key in this file is associated with a particular pointer *to the block* in the sorted data file.

D. **REVERSE INDEX –** A reverse key index reverses the key value before entering it in the index. E.g., the value 24538 becomes 83542 in the index. Reversing the key value is particularly useful for indexing data such as sequence numbers, where new key values monotonically increase.

METHODS OF INDEXING –

i. **Unique Index -** Database systems usually implicitly create an index on a set of columns declared as PRIMARY KEY.

ii. **Non-clustered Index Architecture -** The data is present in arbitrary order, but the **logical ordering** is specified by the index. The non-clustered index tree contains the index keys in sorted order, with the leaf level of the index containing the pointer to the record like page or row number. In a Non-Clustered index, (1) The physical order of the rows is not the same as the index order. (2) The indexed columns are typically non-primary key columns. (3) There can be more than one non-clustered index on a database table.

iii. **Clustered Index Architecture -** Clustering alters the data block into a certain distinct order to match the index, resulting in the row data being stored in order. Therefore, only one clustered index can be created on a given database table. The primary feature of a clustered index is therefore the ordering of the physical data rows in accordance with the index blocks that point to them. Since the physical records are in this sort order on disk, the next row item in the sequence is immediately before or after the last one, and so fewer data block reads are required.

iv. **Cluster Index -** When multiple databases and multiple tables are joined, it's referred to as a **cluster.** The records for the tables sharing the value of a cluster key shall be stored together in the same or nearby data blocks. This may improve the joins of these tables on the cluster key, since the matching records are stored together and less I/O is required to locate them. A cluster can be keyed with a B-Tree index or a hash table.

Q7. What is Search Engine Indexing?

Ans.

The search engine index provides the results for search queries, and pages that are stored within the search engine index that appear on the search engine results page. Without a search engine index, the search engine would take considerable amounts of time and effort each time a search query was initiated, as the search engine would have to search not only every web page or piece of data that has to do with the particular keyword used in the search query, but every other piece of information it has access to, to ensure that it is not missing something that has something to do with the particular keyword.

Search engine spiders, also called search engine crawlers, are how the search engine index gets its information, as well as keeping it up to date and free of spam.

Before you search, web crawlers discover publicly available webpages and gather information from across hundreds of billions of webpages and organize it in the Search index. Crawlers look at webpages and follow links on those pages, much like you would if you were browsing content on the web. They

go from link to link and bring data about those webpages back to Google's servers.
Google's Search Engine Index is like the index in the back of a book — with an entry for every word seen on every web page. When Google indexes a web page, they add it to the entries for all of the words it contains.

Q8. Describe CAP Theorem?

Ans.

The CAP Theorem states that, in a distributed system (a collection of interconnected nodes that share data.), you can only have two out of the following three guarantees across a write/read pair: Consistency, Availability, and Partition Tolerance - one of them must be sacrificed.

- **Consistency** - A read is guaranteed to return the most recent write for a given client.
- **Availability** - A non-failing node will return a reasonable response within a reasonable amount of time with no error or timeouts.
- **Partition Tolerance** - The system will continue to function when network partitions occur.

Given that networks aren't completely reliable; you must tolerate partitions in a distributed system. Since, we cannot control network outages and errors as they are outside the scope of our software, but we do have control on when to carry partitions.
Thus, we are left with choosing between Consistency or Availability. The decision between Consistency and Availability is a *software trade off*.

- **Consistency/Partition Tolerance –** Choose Consistency over Availability when your business requirements dictate atomic reads and writes. However, waiting for a response from the partitioned node could result in timeout error.

- **Availability/Partition Tolerance –** Choose Availability over Consistency when your business requirements allow for some flexibility, around, when the data in the system synchronizes. Use this trade off, when the system needs to continue to function in spite of any external errors. This state also accepts writes that can be processed later when the partition is

resolved. And may result stale data, while returning the most recent version of the data it has.

Q9. Explain Consistent Hashing?

Ans.

Consistent Hashing is an algorithm that is employed by distributed caches that power many high-traffic dynamic web applications. When using distributed caching to optimize performance, it may happen that the number of caching servers changes (reasons for this may be a server crashing, or the need to add or remove a server to increase or decrease overall capacity). By using consistent hashing to distribute keys between the servers, we can rest assured that should that happen, the number of keys being rehashed—and therefore, the impact on original servers be minimized, preventing potential downtime or performance issues.

A hash function is a function that maps one piece of data—typically describing some kind of object, often of arbitrary size—to another piece of data, typically an integer, known as *hash code*, or simply *hash*. Good hash functions should somehow hash the input data in such a way that the outputs for different input values are spread as evenly as possible over the output range.

So, we have an array of size N, with each entry pointing to an object bucket. To add a new object, we need to calculate its hash modulo N, and check the bucket at the resulting index, adding the object if it's not already there. To search for an object, we do the same, just looking into the bucket to check if the object is there. Such a structure is called a *hash table*, and although the searches within buckets are linear, a properly sized hash table should have a reasonably small number of objects per bucket, resulting in *almost* constant time access (an average complexity of O(N/k), where k is the number of buckets).

In distributed caching, pool of caching servers that host many key/value pairs and are used to provide fast access to data originally stored (or computed) elsewhere. For example, to reduce the load on a database server and at the same time improve performance, an application can be designed to first fetch data from the cache servers, and only if it's not present there—a situation known as *cache miss*—resort to the database, running the relevant query and caching the results with an appropriate key, so that it can be found next time it's needed.

The simplest way to implement Distributed hashing is to take the hash *modulo* of the number of servers. That is, server = hash(key) mod N, where N is the size of the pool. To store or retrieve a key, the client first computes the hash, applies a modulo N operation, and uses the resulting index to contact the appropriate server (probably by using a lookup table of IP addresses). Note that the hash function used for key distribution must be the same one across all clients.

Thus placing a heavy load on the original database servers resulting in sever performance degradation or server crashes.

Say we have three servers, A, B and C, and we have some string keys with their hashes:

KEY	HASH	HASH mod 3
"john"	1633428562	2
"bill"	7594634739	0
"jane"	5000799124	1
"steve"	9787173343	0
"kate"	3421657995	2

A client wants to retrieve the value for key john. Its hash modulo 3 is 2, so it must contact server C. If the key is not found there, the client fetches the data from the source and adds it to the pool. The pool looks like this:

A	B	C
"bill"	"jane"	"john"
"steve"		"kate"

if one of the server crashes or becomes unavailable? Keys need to be redistributed to account for the missing server, of course. if one or more new servers are added to the pool; keys need to be redistributed to include the new servers. In case of a simple distribution hash like Modulo which directly depend

on the number of server, most keys will need to be rehashed and moved to a new server, even if one server is added or removed. This would mean that, all of a sudden, distributed caching won't work as the keys won't be found because they won't yet be present at their new location resulting in multiple misses.

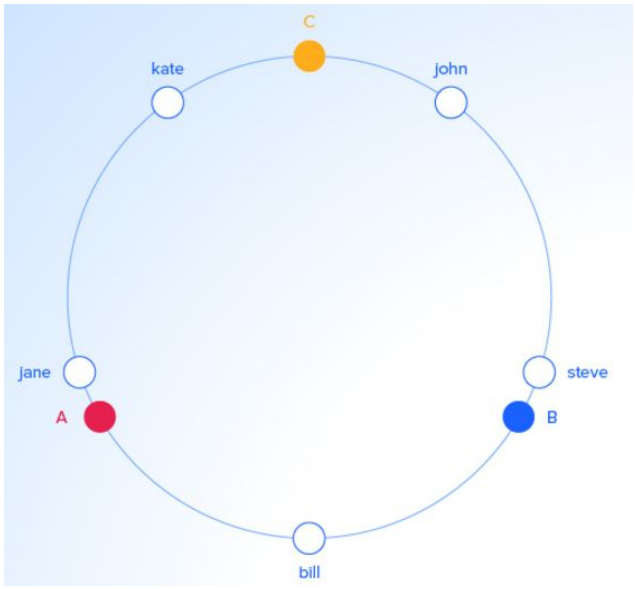
Consistent Hashing minimizes key relocation. Consistent Hashing is a distributed hashing scheme that operates independently of the number of servers or objects in a distributed *hash table* by assigning them a position on a *hash ring*. This allows servers and objects to scale without affecting the overall system.

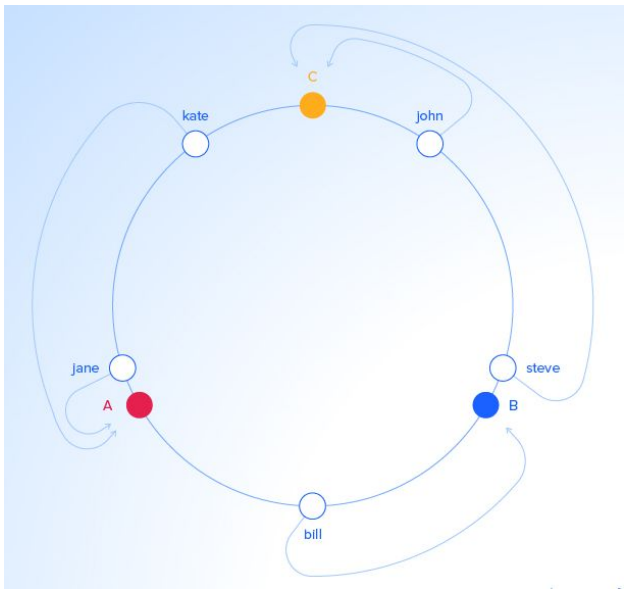
Imagine we mapped the hash output range onto the edge of a circle. That means that the minimum possible hash value, zero, would correspond to an angle of zero, and the maximum possible value (INT_MAX) would correspond to an angle of 2π radians (or 360 degrees), and all other hash values would linearly fit somewhere in between. So, we could take a key, compute its hash, and find out where it lies on the circle's edge.

Now, we also placed the servers on the edge of the circle, by pseudo-randomly assigning them angles too. A convenient way of doing this is by hashing the server name (or IP address, or some ID).

we have the keys for both the objects and the servers on the same circle, we may define a simple rule to associate the former with the latter: Each object key will belong in the server whose key is closest, in a either clockwise or counterclockwise direction.

to find out which server to ask for a given key, we need to locate the key on the circle and move in the ascending angle direction until we find a server.





The benefit of this approach is that removing a server results in its object keys being randomly reassigned to the rest of the servers, **leaving all other keys untouched**. For example, the absence of server C labels does not affect the object keys belonging to server A and B. Or, say, if we wanted to add server D, the result would be that roughly one-third of the existing keys belonging to A and B will be reassigned to D and rest of the keys would stay same. This is how consistent hashing solves the rehashing problem. In general, only k/n keys need to be remapped where k is the number of keys and N is the number of servers.

NEXT

Q10. What is the Difference Between Data Replication and Data Redundancy?

Ans.

Data Replication refers to the scenario when same data is stored on multiple storage devices for high availability even in case of system failures.

It can be used to prevent the loss of a single server from causing your directory service to become unavailable. The amount of replication required for fault tolerance depends on the quality of the hardware and networks used by your directory. Unreliable hardware requires more backup servers.

The two types of Data Replication include –

- **Active replication -** performed by processing the same request at every replica.
- **Passive replication -** involves processing each single request on a single replica and then transferring its resultant state to the other replicas.

Data Redundancy, however refers to unnecessary data duplication. It leads to data anomalies and corruption and generally should be avoided by design.
Applying database normalization and proper user of foreign keys prevent redundancy and makes the best pc NEXT ge of storage

Q11. How does a Content Delivery Network work?

Ans. CDNs are a kind of cache that comes into play for sites serving large amounts of static media.

In a typical CDN setup, a request will first ask the CDN for a piece of static media; the CDN will serve that content if it has it locally available. If it isn't available, the CDN will query the back-end servers for the file and then cache it locally and serve it to the requesting user.

NEXT

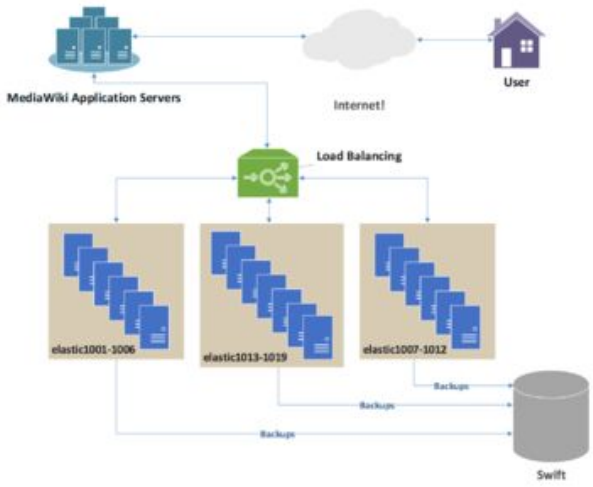
Q12. What is Load Balancing?

Ans.. Load balancing improves the distribution of workloads across multiple computing resources. It is based on the idea that using multiple components with load balancing instead of a single component may increase reliability and availability through redundancy.
Load balancing helps to achieve higher levels of fault tolerance for the applications by automatically routing traffic across multiple server instances using a scheduling algorithm. In absence of Load Balancing, if many users try to

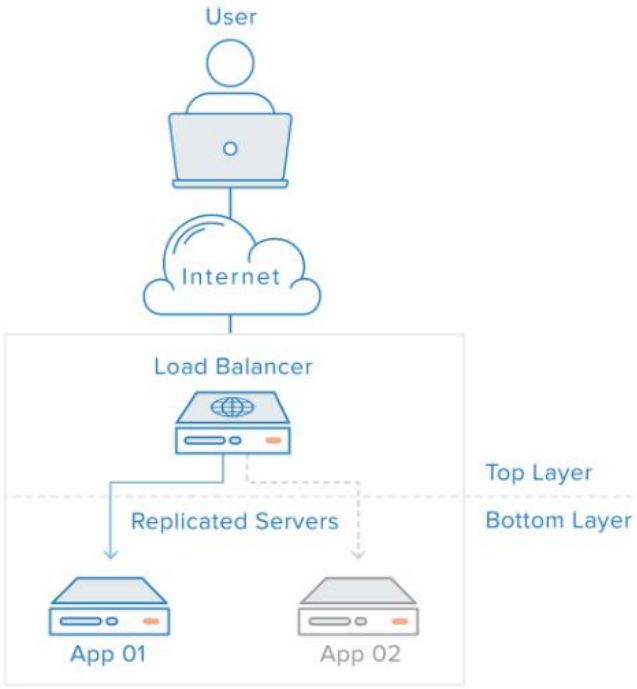
access the server simultaneously and it is unable to handle the load, they may experience slow load times or may be unable to connect at all.

- **Classic Load Balancer** – routes traffic based on either application or network level information.

- **Application Load Balancer** – routes traffic based on advanced application level information that includes the content of the request.



User requests to the Wikimedia Elasticsearch server cluster are routed via load balancing



2. Load Balancing

In the example illustrated above, the user accesses the load balancer, which forwards the user's request to a backend server, which then responds directly to the user's request. In case of Load Balancer failure, we can add a second load balancer as a passive unit to mitigate single point of failure when needed. Load balancers support load balancing HTTP traffic, HTTPS traffic, TCP traffic, as well as DNS and UDP traffic.

LOAD BALANCING ALGORITHMS: -

- **Round Robin —** Round Robin means servers will be selected sequentially. The load balancer will select the first server on its list for the first request, then move down the list in order, starting over at the top when it reaches the end.

- **Least Connections —** Least Connections means the load balancer will select the server with the least number of connections and is recommended when traffic results in longer sessions.

- **Source —** With the Source algorithm, the load balancer will select which server to use based on a hash of the source IP of the request, such as the visitor's IP address. This method ensures that a particular user will

consistently connect to the same server. This helps in establishing sticky sessions.

- **Health Checks -** If a server stops listening and fails a health check, and therefore is unable to serve requests, it is automatically removed from the pool, and traffic is not forwarded to it until it responds to the health checks again.

Q13. How to build a bot using Sentiment Analysis on E-mails in NLP?

Ans.

Natural language processing is the use of algorithms to analyze and understand ordinary human speech to determine metrics such as sentiment.

Sentiment Analysis consists of techniques that determine the tone of a text or speech. To determine if an email needs your attention, the bot will parse it and determine if there is a strong negative tone. It will then send out a text alert if needed using *twilio*. To avoid misinterpretation due to the problem of negation and mixed sentiments, we pursue sentiment analysis using a Recursive Neural Tensor Network. Recursive Neural Tensor Networks are groups of neural networks organized in a tree structure where each node is a neural network. These are particularly useful in natural-language processing where the algorithm processes words and their interaction in a sentence.

The RNTN algorithm first splits a sentence up into individual words. It then constructs a neural network where the nodes are the individual words, finally adding a Tensor layer. We make use of the open source *Stanford NLP library* written in Java. It is a set of natural language processing software. A web server is built for python using *flask* and *Thrift* is used to resolve communication between python and java codes. Thrift is a code generator and a protocol used to enable two applications, often written in different languages, to be able to communicate with one another using a defined protocol.

Finally, we need a logic score to determine the sentiment of the text and produce a string containing the sentiment annotations for each sentence in the text. For example, use a simple scoring system and if an email contains more than 75% negative sentiment sentences, we can mark that as a potential alarm email that may require an immediate response. The logic for sentiment score may appear as below –

```
urgent = len(negative_sentences) / len(sentences) > 0.75
```

```
Received: Here is a test for the system. This is supposed to be a non-urgent request.
It's very good! For the most part this is positive or neutral. Great things
are happening!
urgent = False

Received: This is an urgent request. Everything is truly awful. This is a disaster.
People hate this tasteless mail.
urgent = True
```

3. Sample Output

This is how we can build an email bot that is able to receive emails, perform sentiment analysis, and determine if an email requires immediate attention.

Q14. Discuss underlying concepts & algorithms of Neural Networks & Deep Learning?

Ans.

Consider a supervised, binary classification problem, whose algorithm is given a handful of labeled examples, say 10 images of dogs with the label 1 (“Dog”) and 10 images of other things with the label 0 (“Not dog”). Consider your data as symptoms and your labels illnesses. The algorithm “learns” to identify images of dogs and, when fed a new image, hopes to produce the correct label for that image i.e., 1 if it’s an image of a dog, and 0 otherwise.

Perceptron is an example of supervised training algorithm, which uses basic neural network building blocks to create a classifier, linear or otherwise, that best separates the labelled data into categories.

Training the Perceptron –

The training of the perceptron consists of feeding it multiple training samples and calculating the output for each of them. After each sample, the weights *w* is adjusted in such a way so as to minimize the *output error*, defined as the difference between the *desired* (target) and the *actual* outputs.

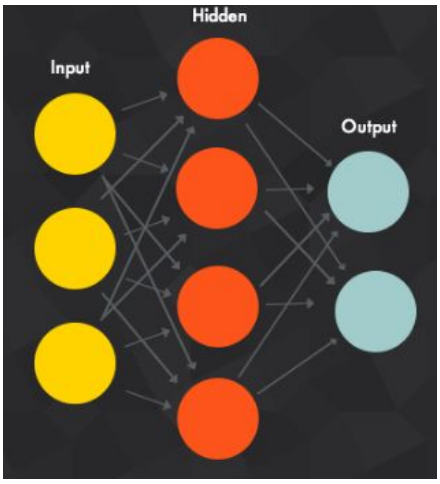


fig. 1 A Neural Network with a 3 neuron input layer, 4-neuron hidden layer and 2-unit output layer.

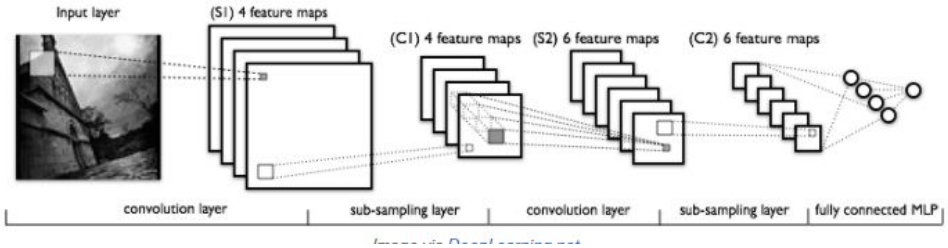
- Multilayer perceptron i.e. one with more than one neurons in input-output layer are capable of coming with classifiers beyond linearity.
- The units of the input layer serve as inputs for the units of the hidden layer, while the hidden layer units are inputs to the output layer.
- Each connection between two neurons has a weight *w*
- Most neural networks use non-linear activation functions like the logistic, tan h, binary or rectifier instead of linear activation functions.
- Input values are propagated forward to the hidden units using the weighted sum transfer function for each hidden unit(neuron), which in turn calculate their outputs (activation function).
- The hidden layer acts as effective feature detector as middle layer is usually lesser in number than i/o layer, which result in dimensality reduction at the output layer focusing only on the major features.

CONVOLUTIONAL NETWORKS –

A common deep learning architecture for image processing. **Convolutional layers** apply a number of *filters* to the input. An image *filter* is a square region with associated weights. A filter is applied across an entire input image, and you will often apply multiple filters. The result of one filter applied across the image is

called *feature map* (FM) and the number feature maps is equal to the number of filters.
If the previous layer is also convolutional, the filters are applied across all of its FMs with different weights, so each input FM is connected to each output FM. The intuition behind the shared weights across the image is that the features will be detected regardless of their location, while the multiplicity of filters allows each of them to detect different set of features. A loss function can be defined in many different ways but a common one is MSE (mean squared error), which is ½ times (actual - predicted) squared.

Subsampling layers reduce the size of the input. For example, if the input consists of a 32x32 image and the layer has a subsampling region of 2x2, the output value would be a 16x16 image, which means that 4 pixels (each 2x2 square) of the input image are combined into a single output pixel. The last subsampling (or convolutional) layer is usually connected to one or more fully connected layers, the last of which represents the target data.



Training is performed using modified backpropagation that takes the subsampling layers into account and updates the convolutional filter weights based on all values to which that filter is applied. Propagating data through the network is a two-step process:

Determine the order of the layers –

To get the results from a multilayer perceptron, the data is “clamped” to the input layer (hence, this is the first layer to be calculated) and propagated all the way to the output layer. In order to update the weights during backpropagation, the output error has to be propagated through every layer employing different graph traversal methods like breadth first strategy. The order is actually determined by the connections between the layers, so the strategies return an ordered list of connections.

Calculate the activation value –

Each layer has an associated connection calculator. which takes its list of connections (from the previous step) and input values (from other layers) and calculates the resulting activation.

Facebook uses neural nets for their automatic tagging algorithms, Google for photo search, Amazon for product recommendations. Facebook can use all the photos of the billion users it currently has, Google can use search data, and Amazon can use data from the millions of products that are bought every day. The more training data that you can give to a network, the more training iterations you can make, the more weight updates you can make, and the better tuned the network.

Q15. Discuss Clustering Algorithms in depth?

Ans. Clustering Algorithms are part of Unsupervised Learning, where the data we want to describe is not labeled. clustering – separating data into groups (clusters) that contain similar data points, while the dissimilarity between groups is as high as possible. This is essentially useful when we want to group similar data points say, users and run specific operations on each cluster.

K-means Clustering

The algorithm begins by selecting k points as starting centroids ('centres' of clusters). We can just select any k random points, or we can use some other approach, but picking random points is a good start. Then, we iteratively repeat two steps:

1. Assignment step: each of m points from our dataset is assigned to a cluster that is represented by the closest of the k centroids. For each point, we calculate distances to each centroid, and simply pick the least distant one.

2. Update step: for each cluster, a new centroid is calculated as the mean of all points in the cluster. From the previous step, we have a set of points which are assigned to a cluster. Now, for each such set, we calculate a mean that we declare a new centroid of the cluster.

    ⌘ After each iteration, the centroids are slowly moving, and the total distance from each point to its assigned centroid gets lower and lower. The two steps are alternated until there are no more changes in cluster assignment.

- ⌘ After a number of iterations, the same set of points will be assigned to each centroid, therefore leading to the same centroids again. K-Means is guaranteed to converge to a local optimum.
- ⌘ Selection initial points is an important problem as the final clustering result can depend on the selection of initial centroids.

- ⌘ One solution is to run K-Means a couple of times with random initial assignments then select the best result by taking the one with the minimal sum of distances (error value) from each point to its cluster.

- ⌘ Another way is make selection using distant points. That is, pick random points but with probability proportional to square distance from the previously assigned centroids. Points that are further away will have higher probability to be selected as starting centroids.

- ⌘ Hence, with K-Means clustering each point is assigned to just a single cluster, and a cluster described only by its centroid. This is not too flexible, as we may have problems with clusters that are overlapping, or ones that are not of circular shape.

is

E M Clustering

EM Clustering, describes each cluster by its centroid (mean), covariance (so that we can have elliptical clusters), and weight (the size of the cluster). The probability that a point belongs to a cluster is now given by a multivariable Gaussian probability distribution. That also means that we can calculate the probability of a point being under a Gaussian 'bell', i.e. the probability of a point belonging to a cluster.

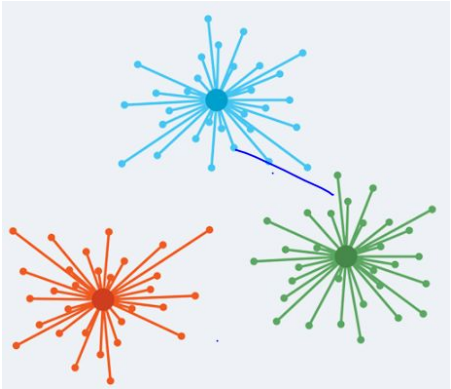
For each point, the probabilities of it belonging to each of the current clusters (randomly created at the beginning) is calculated. If one cluster is a really good candidate for a point, it will have a probability close to one. However, two or more clusters can be acceptable candidates (called soft clustering), so the point has a distribution of probabilities over clusters. If we want to determine a single cluster for each point, we may simply choose the most probable one.

To calculate the new mean, covariance and weight of a cluster, each point data is weighted by its probability of belonging to the cluster, as calculated in the previous step.

Alternating these two steps will increase the total log-likelihood until it converges. Again, the maximum may be local, so we can run the algorithm several times to get better clusters.

However, the main drawbacks of K-Means and similar algorithms are to select the number of clusters, and choose the initial set of points.

Affinity Propagation –



As an input, the algorithm requires us to provide two sets of data:

- Similarities between data points
- Preferences

- Similarities between data points represent how well-suited a point is to be another one's exemplar. If there's no similarity between two points, as in they cannot belong to the same cluster, this similarity can be omitted or set to -Infinity depending on implementation.

- Preferences represent each data point's suitability to be an exemplar. We may have some priori information about which points could be favored for this role, and so we can represent it through preferences.

The algorithm then runs through a number of iterations, until it converges. Each iteration has two message-passing steps:

- **Calculating responsibilities:** Responsibility r(i, k) reflects the accumulated evidence for how well-suited point k is to serve as the exemplar for point i, taking into account other potential exemplars for point i. Responsibility is sent from data point i to candidate exemplar point k.

⌘ **Calculating availabilities:** Availability a(i, k) reflects the accumulated evidence for how appropriate it would be for point i to choose point k as its exemplar, taking into account the support from other points that point k should be an exemplar. Availability is sent from candidate exemplar point k to point i.

Q16. Enumerate theory behind machine learning and its applications?
Ans.

The highly complex nature of real world problems that cannot be solved by numerical methods alone require the use of ML algorithms.

Some Example of Machine Learning Problems

- **Supervised machine learning:** The program is "trained" on a pre-defined set of "training examples", which then facilitate its ability to reach an accurate conclusion when given new data.

- **Unsupervised machine learning:** The program is given a bunch of data and must find patterns and correlations therein. E.g., identifying close-knit groups of friends in social network data. K-means Clustering is a popular example of Unsupervised learning algorithm.

⌘ Supervised Learning –

Supervised Learning" consists of using sophisticated mathematical algorithms to optimize the predictor function so that, given input data x about a certain domain (say, square footage of a house), will accurately predict some interesting value h(x) (say, market price for said house).

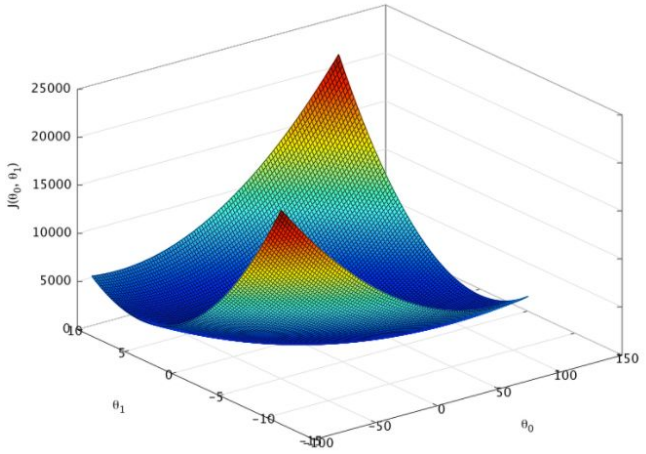
The goal is to devise an optimal predictor. Optimizing the predictor h(x) is done using **training examples**. For each training example, we have an input value. for which a corresponding output, y, is known in advance. For each example, we find the difference between the known, correct value y, and our predicted value, this process is repeated over and over until the system has converged on the best values for the predictor function, its coefficients (based on number of data inputs) and constants. In this way, the predictor becomes trained, and is ready to do some real-world predicting.

But how do we make sure that the constants and coefficiencts are getting better with each step, and not worse? This can be achieved through Gradient Descent which measures the degree of wrongness during the optimization process. This is known as **loss or cost function.** With a least squares function, the penalty for a bad guess goes up quadratically with the difference between the guess and the correct answer. The cost function computes an average penalty over all of the training examples.

Remember our goal here is to find θ_0 and θ_1 for our predictor h(x) such that our cost function $J(\theta_0, \theta_1)$ is as small as possible, where –

$$H(x) = \theta_0 + \theta_1 x$$

In the diagram below, the bottom of the bowl represents the lowest cost our predictor can give us based on the given training data. The goal is to "roll down the hill", and find θ_0 and θ_1 corresponding to this point.



This is where calculus comes in machine learning. Essentially what we do is take the gradient of $J(\theta_0, \theta_1)$, which is the pair of derivatives of $J(\theta_0, \theta_1)$ (one over θ_0 and one over θ_1). The gradient will be different for every different value of θ_0 and θ_1, and tells us what the "slope of the hill is" and, in particular, "which way is down", for these particular θs. For example, when we plug our current values of θ into the gradient, it may tell us that adding a little to θ_0 and subtracting a little from θ_1 will take us in the direction of the cost function-valley floor. Therefore, we add a little to θ_0, and subtract a little from θ_1, and voilà! We have completed one round of our learning algorithm. Our updated predictor, h(x) = θ_0 + θ_1x, will return better predictions than before. Our machine is now a little bit smarter.

This process of alternating between calculating the current gradient, and updating the θs from the results, is known as gradient descent.

TYPES OF SUPERVISED ML

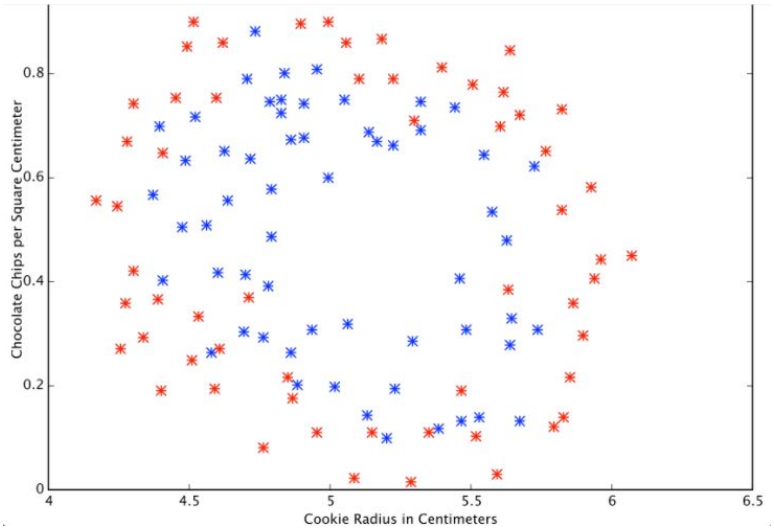
- **Regression machine learning systems:** Systems where the value being predicted falls somewhere on a continuous spectrum. These

systems help us with questions of “**How much** say cost?” or “How many?”.

- **Classification machine learning systems:** Systems where we seek a yes-or-no prediction, such as “Is this tumour cancerous?”, “Does this cookie meet our quality standards?”, and so on. The major differences between both systems are the design of the predictor h(x) and the design of the cost function $\mathbf{J}(\theta)$.

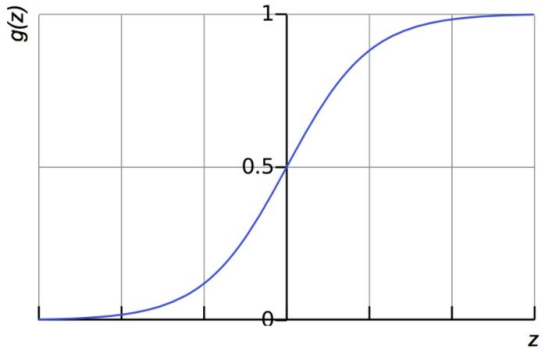
Classification problem

Below are the results of a cookie quality testing study, where the training examples have all been labeled as either “good cookie” (y = 1) in blue or “bad cookie” (y = 0) in red.



Unlike a regression predictor, a classification predictor makes a guess somewhere between 0 and 1. Say, a prediction of 1 would represent a very confident guess that the cookie is delicious. While A prediction of 0 represents high confidence that the cookie is close to inedible. While a value exactly in the middle, at 0.5, might represent complete uncertainty or a tough call. This isn’t always how confidence is distributed in a classifier but it’s a very common design and works for purposes of our illustration.

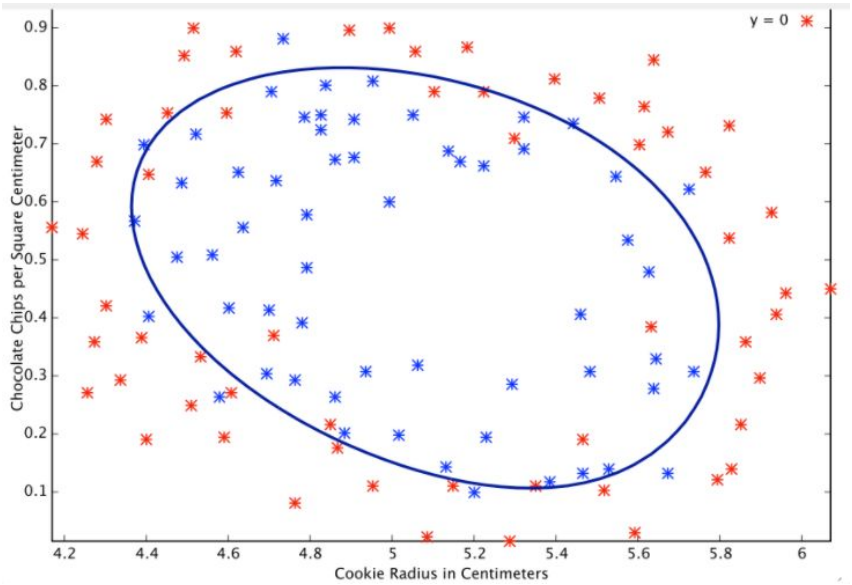
A function that captures this behavior very well is called a sigmoid function, illustrated below. The sigmoid function transforms our output into the range between 0 and 1.



Sigmoid function g(z) distribution over z as input

The logic behind the design of the cost function is that, if the correct guess was 0 and we guessed 1, or vice versa, the penalty will be maximum. Alternatively, if the correct guess was 0 and we guessed 0, then no cost would be added. Or, If the guess was right, but we weren't completely confident (e.g. y = 1, but h(x) = 0.8), this should come with a small cost, and if our guess was wrong but we weren't completely confident (e.g. y = 1 but h(x) = 0.3), this should come with some significant cost. The cost function $\mathbf{J}(\theta)$ gives us the average cost over all of our training examples.

A classification predictor can be visualized by drawing the boundary line; i.e., the barrier where the prediction changes from a "yes" (a prediction greater than 0.5) to a "no" (a prediction less than 0.5). The classification predictor boundary would be drawn as shown below –



Classification Predictor Boundary

NEXT

Q17. What is Big Data?
Ans.

Big Data as the name suggest means lots of data. The first step to big data analytics is gathering the data itself. This is known as "data mining." Data mining is the task of pulling a huge amount of data from a source and storing it. The result of this is "big data," which is just a large amount of data in one place.

For an organization, big data analytics can provide insights that surpass human capability. Being able to run large amounts of data through computation-heavy analysis.

Data can come from anywhere. You can extract quite a bit from a user by analyzing their tweets and trends. You can also see how people are talking specific topics using keywords and business names. Most businesses deal with gigabytes of user, product, or location data. For example, Messenger data provides data on users' conversations. In this case, the big data are conversations between users. If you were to individually read the conversations of each user, you would be able to get a good sense of what they like, and be

able to recommend products to them accordingly. The whole data gathering process can be automated using NLP techniques.

If you're trying to get a large amount of data to run analytics on to analyze how your company is received in the general public. You could collect the last 2,000 tweets that mention your company and run a sentiment analysis algorithm over it. Here are some practical ways you can use this information:

- Create a spatial graph on where your company is mentioned the most around the world
- Run sentiment analysis on tweets to see if the overall opinion of your company is positive or negative
- Create a social graphs of the most popular users that tweet about your company or product

Q18. Describe Virtual Reality in Automotive Industry?

Ans.

Google's self-driving Waymo car drove an average of 5,000 miles on its own before requiring human intervention. Tesla's Autopilot conveniently drives you across a highway, but after it reaches your exit ramp, you are expected to take control again. You may sell taxi rides in driverless cars. Alternatively, Toyota used a driving simulator to demonstrate the effects of distracted driving using their TeenDrive 365. The same approach could be used for pedestrians and cyclists, not just car drivers. They can learn the basics, and they can also be confronted with numerous scenarios that would be impractical or impossible to reproduce in real life.

VR could be viewed as a learning aid. VR can also be used to train people how to behave in various emergencies, which could make for much more enjoyable fire drills. Enter AR and you can interactively guide panicking crowds to safety.

Q19. Describe types of Process Schedul

Ans. Two Main types –

- Long Term Scheduler – Long Term Scheduler is responsible for loading processes from the secondary memory to the main memory, so it decided the degree of multiprogramming i.e., the number of processes that can be present in the main memory at a time. Once LTS brings processes to the main memory, then duty is delegated to Short Term Scheduler.

- Short Term Scheduler – STS is only responsible for dispatching a process from the running to ready state and reverse depending upon the scheduling algorithm. Its decision lasts for a shorter duration than LTS.

Q20. Explain the concept of inode in file system?

Ans. Inode is a data structure in a UNIX style file system used to store a file's metadata like owner, time of last change, block location etc.

Q21. Describe the steps involved in context switching (switching from on thread to another in a multithreaded system)?

Ans.

Context Switching is a computationally intensive process which involves switching registers, stack pointers and program counters.

Steps: -

1) In a context switch, the state of the process currently executing must be saved, so that when it is rescheduled, this state can be restored.

2) The state of process includes all the registers that the process may be using, the program counter, other OS specific data. This is stored in PCB or Process Control Block.

3) A handle to the PCB is added to the queue of processes that are ready to run known as ready queue.

4) Now that the execution of current process is suspended by OS, it then switches context by choosing a process from the ready queue and restoring its PCB.

5) During this task, the Program Counter from the PCB is loaded and thus execution can continue in the chosen process. If the ready queue is a priority queue, then thread priority may influence which process is eventually chosen from the queue.

Q22. List the difference between System Call and Interrupt?

Ans. System Calls – lets the user access the kernel in a controlled manner. Because User programs cannot be trusted to invoke some functionality in kernel directly. So, to access any system functionality, they make use of system calls.

Interrupt – An interrupt is a request made by an external device for a response from the processor. This may be hardware interrupts like from keyboard, mouse etc. or software interrupts generated via a program.

Q23. How can you implement x=x+a for a concurrent system?

Ans. To implement the code safely, we need to acquire a write lock before writing our value. Because for concurrent systems, race condition can come and result in multiple processes or threads reading the old value simultaneously and updating independently. This would result in overwriting results on each other.

A mutually exclusive write lock on x would result in updating of value sequentially which would avoid the race condition.

Q24. What is Preemptive Multi-Tasking Vs Time-Sharing Multitasking?

Ans.

In Time Sharing, OS decides the time that should be given for each process and after the time elapse the resources are allocated to the next process in the queue. However, depending on task's priority, additional time might be allocated to some processes eg., some OS background.

In Preemptive multitasking, to prevent a process from taking indefinite/full control over the resources, the preemptive multitasking restricts the time allocated for each process to use the resources.

Q25. In Linux OS, can two files have the same Inode number?

Ans. Yes, In Linux kernel, Inode numbers are created to index files locally and are file-system specific. That is, a file is identified by first identifying the device and an Inode within the device. Therefore, two files on different devices or partitions can have same Inode number.

Q26. Difference between Mutex and Semaphore?

Ans.

Mutex as its name suggests provides complete mutual exclusion. At any point of time only one thread can lock a particular mutex. A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section.

Semaphore is a non-negative integer used when we want more fine grained parallelism. A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore).

Semaphores are useful when we do not need complete mutual exclusion but want only concurrency control. For example, in Readers-Writers problem where multiple readers can read at any point of time but only one writer can change the data.

Q27. What is a Spin lock in Operating Systems?

Ans. Spin lock is a lock that will cause a thread enter a loop to repeatedly check if the lock is available. When the calling thread requests a spin lock that is already held by another thread, the second thread spins in a loop to test if the lock has become available.

It is different from regular lock like mutex which will put a thread into BLOCK state and likely to pre-empt the thread, which has more overhead when the lock is only unavailable for a short period of time. If the thread could acquire the lock within its own CPU quota, then it would be more efficient to use spin lock. However, if the waiting time is unknown, or too long, mutex will be definitely better.

Q28. If you want to create a software which consists of a video and its audio and subtitle file running in the background, what would you choose to create? Process or Thread?

Ans. Threads are easy to maintain synchronize status between each other. Process is more heavy weight. Since all three different actions - video, audio and subtitle run over the same background and are in absolute synchronization, thread is the best option.

Q29. What are some of the ways that processes can communicate with each other?

Ans. Interprocess Communication or IPC refers to the mechanisms an operating system provides to allow the processes to manage shared data. Common

approaches include Sockets, message passing, shared memory, or using files to store data.

Q30. Explain virtual memory and page faults and LRU algo?

Ans.

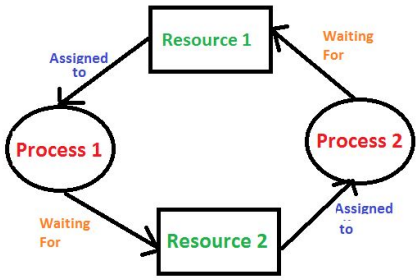
Virtual Memory refers to mapping of virtual addresses to physical memory addresses done by the CPUs MMU (memory management unit) based on information from the operating system. Virtual Memory is a way of using hard drive to provide a memory for the computer. Elements of virtual memory are called pages. When a needed memory that is not in the real memory is requested a memory from virtual memory moves to real memory address. Computers have a finite amount of RAM so when many programs run at the same time memory can run out. Using virtual memory, it can load larger programs at the same time and operate like it has infinite memory. However, using virtual memory can slow computers down because data must be mapped between real memory (physical) and virtual memory which requires extra capabilities for address translations .

Page fault occurs when a virtual address or swapped out address is accessed that doesn't exist at all. In such case, interrupt is raised by hardware and OS terminates the program assuming an error.

LRU or least recently used algorithm is a common algorithm used to manage caches in a way that elements that have been accessed (read/write) recently are kept in the cache while elements not being used lately are purged from cache if the memory is needed. This algorithm is applied for swapping out pages, to secondary storage, like disc.

Q31. *Describe Deadlock?

Ans.



Deadlock is a situation where a set of processes are blocked because each

process is holding a resource while waiting for another resource acquired by some other process. A relatable example, you need experience to get a job and a job to get experience.

Necessary Conditions for Deadlock

- Mutual Exclusion: There is s resource that cannot be shared.
- Hold and Wait: A process is holding at least one resource and waiting for another resource which is with some other process.
- No Preemption: The operating system is not allowed to take a resource back from a process until process gives it back.
- Circular Wait: A set of processes are waiting for each other in circular form.

How to break the deadlock?

The above are necessary but not sufficient conditions, even if all the conditions exist the system may or may not be in a deadlock, but if it's missing then deadlock won't happen.

Q32. Enumerate the differences between user level thread and kernel level thread?

Ans.

USER LEVEL THREAD	KERNEL LEVEL THREAD
User thread are implemented by users.	kernel threads are implemented by OS.
OS doesn't recognized user level threads.	Kernel threads are recognized by OS.
Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
Context switch time is less.	Context switch time is more.
Context switch requires no hardware support.	Hardware support is needed.
If one user level thread perform blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.
Example : Java thread, POSIX threads.	Example : Window Solaris.

Q33. What is Paging?

Ans. Paging is a memory management scheme that eliminates the need for contiguous allocation of physical memory. This scheme permits the physical address space of a process to be non – contiguous by providing mapping technique to translate virtual addresses to physical addresses.

Q34. Will Shortest Job First suffer from Convoy effect?

Ans. Yes, the non-preemptive version of SJF will suffer from Convoy effect. If the very first process which came in Ready state is having a large burst time. But it will be scheduled on CPU as it is the only process. Then soon after other processes begin to come having small burst time but the 1st process will keep on executing as SJF is Non Preemptive Mode. This will affect the Avg. waiting time of the processes which will led to Convoy Effect.

Q35. Difference between Paging and Segmentation?

Ans.

Paging	Segmentation
A page is a physical unit of information.	A segment is a logical unit of information.
A page is invisible to the user's program.	A segment is visible to the user's program.
A page is of fixed size e.g. 4Kbytes.	A segment is of varying size.
The page size is determined by the machine architecture.	A segment size is determined by the user.
Fragmentation may occur.	Segmentation eliminates fragmentation.
Page frames on main memory are required.	No frames are required.

Q36. Explain the working of copying garbage collector?

Ans. The copying garbage collector basically works by going through live objects and copying them into a specific region in the memory. This collector traces through all the live objects one by one. This entire process is performed in a single pass. Any object that is not copied in memory is garbage.

Q37. What is a translation look aside buffer?

Ans. In a cached system, the base addresses of the last few referenced pages is maintained in registers called the TLB that aids in faster lookup. TLB contains those page-table entries that have been most recently used. Normally, each virtual memory reference causes 2 physical memory accesses- one to fetch appropriate page-table entry, and one to fetch the desired data. Using TLB in-between, this is reduced to just one physical memory access in cases of TLB-hit.

Q38. What is Demand Paging?

Ans. Demand paging is a system wherein area of memory that are not currently being used are swapped to disk to make room for an application's need.

Q39. What is Reentrancy?

Ans. It is a useful, memory-saving technique for multiprogrammed timesharing systems. A Reentrant Procedure is one in which multiple users can share a single copy of a program during the same period. Reentrancy has 2 key aspects: The program code cannot modify itself, and the local data for each user process must be stored separately.

Q40. Difference between shared memory and message queue?

Ans. It's an area of storage that can be read and written by more than one process. It provides no inherent synchronization; in other words, it's up to the programmer to ensure that one process doesn't clobber another's data. But it's efficient in terms of throughput: reading and writing are relatively fast operations

A message queue is a one-way pipe: one process writes to the queue, and another reads the data in the order it was written until an end-of-data condition occurs. When the queue is created, the message size and queue length (maximum number of pending messages) are set. Access is slower than shared memory because each read/write operation is typically a single message. But the queue guarantees that each operation will either processes an entire message successfully or fail without altering the queue. So the writer can never fail after writing only a partial message, and the reader will either retrieve a complete message or nothing at all.

Q41. What happens if non recursive mutex is locked more than once?

Ans. Any attempt to perform the "lock" operation on an ordinary mutex (lock) would either fail or block (deadlock) when the mutex is already locked. Although,

Recursive (or, Reentrant) mutex, could be locked multiple times by a thread, also requires same amount of unlock operations.

Q42. Which is better? Shortest Job First or First Come First Serve algorithm for process scheduling?

Ans. Shortest Job First is the optimal algorithm for process scheduling but it can be beaten by First come First serve in the following scenarios –

1. When all processes have equal burst times. In this case unnecessary time will be wasted by SJF sorting algorithm. Hence, it will lead to more overhead.
2. When all inputs are already sorted in ascending order. This again follows from the complexity of sorting algorithm.

Q43. What is Priority Inversion? How to fix it?

Ans.

PRIORITY INVERSION

Shared resource needed by high priority processes are held by low priority processes, while low priority processes fail to get CPU usage. This makes the priority inversed (High priority waits for low priority.), and results in the situation that neither high priority process cannot execute normally (without resources) nor low priority cannot execute normally (without CPU).

SOLUTION

Use "Priority Inheritance" to solve this situation. First, we allow low priority processes to inherit the priority of high priority processes, which makes low priority processes execute normally and exit and then release the resources. Thus, high priority process can get the resources they need and execute successfully.

Q44. How do you write code that is sure not to run into deadlock condition?

Ans.

- **By not using any semaphores or mutexes (synchronization primitives) if not really required.**
- **By not using multiple synchronization primitives.**
- **By not acquiring synchronization primitives in a nested manner.**
- **By nesting synchronization primitives but only acquiring them in the same order.**
- **By some algorithm that detects dead locks and sets one party back.**

Q45. How do you detect Thrashing in OS?

Ans. Thrashing usually refers to a phenomenon in memory and storage. It occurs when OS spends most of time on handling page faults (paging in or paging out) rather than getting anything done. When thrashing is detected, we could either kill a process or swap a process completely into secondary storage. Additionally, we could also add more RAM to the machine.

Q46. Why Sleep Inside Interrupt Service Routines discouraged?

Ans. Because, if an interrupt (thread) goes to sleep. No other thread can wake that thread except for an (interrupt)thread with a greater Interrupt priority level (IPL) than the sleeping interrupt's IPL. And/or, until its sleep time expires. So, it may have to wait for a longer time or forever until this happens.

Q47. If multiple cores are executing at different frequencies, how to implement synchronization?

Ans. For example if 1 core is working at x speed and other working at 100x speed, the faster core will always hog on the common resource and slow core will starve. solution to this problem is token based mutex. The mutex should assign a token to the incoming request irrespective of which core generates the request. And it should serialize the service to request using these tokens. This way both cores will get the shared variable irrespective of their clock speeds.

Q48. What are the differences between Exception and Interrupts?

Ans. Exception condition refers to unexpected events caused by a process, such as addressing illegal memory, executing privileged instructions, divide by zero etc. While interrupts are caused by the events that are external to process. Interrupts are Hardware interrupts, while exceptions are software interrupts.

Q49. What happens when we type "ls" in command prompt?

Ans. First of all, whenever we press a key on keyboard, the keyboard controller will emit an interrupt to processor(CPU) indicating there is an event that needs immediate attention. As interrupts usually have high priority, the processor will be suspending its current execution, save its state, and call an interrupt handler

(should be the one that handles keyboard interrupt). Suppose we type 'l' then this character will be written the file that fd stdout points to, while shell's stdout usually points to screen (a device, in *nix family, everything "looks" like a file), then "l" will be shown on the screen. After the interrupt handler finishes its job, the process will resume its original work.

We type 'ls' and hit enter, then shell will first check out $PATH environment variable to see if there is a program 'ls' under each given path. Suppose we find /usr/bin/ls. Shell will call fork (), followed by execve("/usr/bin/ls"). Fork () will create an identical child process and return twice. In parent(shell), it will typically call wait () to wait child process to complete. In child, it will execute execve () and a successful execve () will replace original data (including text, data, heap and stack, etc.) in the child process's address space with new data in order to run the new executable. Note that file descriptors opened by parent will be kept (that is why output from ls will be displayed on screen like shell).

Then the child process will be one that runs "/usr/bin/ls" code, it will make system calls (open (2), printf(3c) etc.) to list directory entries in the current working directory. After the child process finishes its job, it will call exit ()
The parent process (in this case the shell) will be notified of child's termination, so wait () will be return and child exit code could be read from wait(). Then parent process (the shell) can proceed, waiting for next command to run.

Q50. Difference between a fork and a thread?

Ans.

- ⌘ A fork gives us a brand new process which is a copy of the current process with the same code segment. It looks exactly like the parent process with different process id having its own memory.

- ⌘ Parent process creates a separate address space for the child with same code segments but executes independently of each other. Because the system issues a new memory space and environment for the child process, it is known as heavy-weight process.

- ⌘ While threads can execute in parallel with same context. Also, memory and other resources are shared between the threads causing less overhead.

- ⌘ A thread process is considered a sibling while a forked process is considered a child. Also, threads are known as light-weight processes as they don't have any overhead as compared to processes (as it doesn't

issue any separate command for creating completely new virtual address space).

- ⌘ A single process can have multiple threads. For all threads of any process, communication between them is direct. While process needs some interprocess communication mechanism to talk to other processes.

Q51. List some OS Scheduling algorithms?

Ans.

- ⌘ First Come First Serve(FCFS) Scheduling.
- ⌘ Shortest-Job-First(SJF) Scheduling.
- ⌘ Priority Scheduling.
- ⌘ Round Robin(RR) Scheduling (time slicing)
- ⌘ Multilevel Queue Scheduling.

Q52. What is Kernel and an Operating System?

Ans. Kernel is a part of an operating system. For example, Linux is a kernel as it does not include any extra applications with it. That's why companies like Ubuntu, RedHat have added some extra utilities and interfaces and made that as an OS. The kernel is "brain" of the operating system, which controls everything from access to the hard disk to memory management.

It provides low level services like device management, process management, memory management i.e it provides all the core system calls to accomplish any task.

OS acts as an interface between the application programs and the system hardware. It is responsible for resource allocation, assigning CPU to processes running, managing hardware etc. Operating Systems include GUI components like shells, command interpreter, browser, compilers, text editors, windowing systems etc. via which user asks the kernel to perform a certain job.

Q53. Give a brief introduction of the terminology of Race Condition in OS?

Ans. A race condition arises when concurrently executing code attempts to access a shared resource simultaneously, due to an error in the code / assumption about sequence of events / assumption about timing of events / assumption of exclusive access.

Most common example is simultaneous access to a file / shared memory (read/write, write/write) - a process reads/writes a file assuming that another process already finished writing into a file.

Another common example is reading from database with improper transaction isolation level.

Using proper IPC techniques (mutex/locks/semaphores) helps to reduce the risks of race conditions.

Q54. How does a Segmentation fault work internally?

Ans. When a processor is given a virtual address, it passes the address to MMU which will check if there is a mapping from the virtual address to physical address by looking up the page table (MMU will check TLB first though).

If there is no such mapping, the processor will take it as a page fault and subsequently check the page fault is a valid or not by checking if the address belongs to any segments the process currently has.

When the page fault is valid, which means the page resides in the swap space and needs to be swapped into memory, the processor will proceed after the page gets swapped in; otherwise, the page fault is invalid, the processor will send a segmentation fault signal to the process and kills the process by default if there is no signal handler that catches the SIGSEGV signal.

Q55. Describe the Role of ACID and BASE in Database transactions?

Ans. The two most common consistency models are known by the acronyms ACID and BASE. Both have their advantages or disadvantages.

The ACID Consistency Model guarantees that it provides a safe environment of data so data can be considered reliable.

The ACID model says that database transactions should be:

- **A**tomic: Everything in a transaction succeeds or the entire transaction is rolled back.
- **C**onsistent: A transaction cannot leave the database in an inconsistent state.
- **I**solated: Transactions cannot interfere with each other.

- **D**urable: Completed transactions persist, even when servers restart etc.

THE BASE CONSISTENCY MODEL

BASE model works when databases have loosened the requirements for immediate consistency, data freshness and accuracy in order to gain other benefits, like scale and resilience.

The Base Acronym says: -

- **B**asic **A**vailability
  - The database appears to work most of the time.
- **S**oft-state
  - Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.
- **E**ventual consistency
  - Stores exhibit consistency at some later point (e.g., lazily at read time).

A BASE data store values availability (since that's important for scale), but it doesn't offer guaranteed consistency of replicated data at write time.
Rather than requiring consistency after every transaction, it is enough for the database to eventually be in a consistent state.

Q56. Explain different levels of RAID?

Ans. ….

Q57. Explain Lazy Loading?

Ans. …

Q58. Explain Asynchronous flows? How it affects Scaling?

Ans. …

Q59. What is a Content Delivery Network? How does it work?

Ans. …

Q60. How is Routing Performed through Distributed Hash Table?
Ans. ....

Q61. What are MicroServices?
Ans. ...

Q62. Do you understand how to parallelize algorithms?

Ans. ….

Q63. Do you know what is coherence and starvation?
Ans. …

Q64. Are you familiar with IPC and TCP/IP?
Ans. ……

Q65. What is the difference between throughput and latency and when each is relevant factor?

Ans. …….

Q66. What is the best way to provide distributed storage to blob data?

Ans. Blob data refers to small and large objects such as pdf, image, video, gif etc. One alternative is Facebook's Haystack which is not open source (cons). Another better open source alternative is HDFS cluster to store images and videos and files. And its scalable and highly fault tolerant (too many data nodes) (pros). A typical file in HDFS can range from GBs to TBs, spawning into multiple blocks.

Q67. What is Back of the envelope analysis and Microbenchmarking?

Ans. The

1. *Back-of-the-envelope analysis*. This essentially means developing an intuition for the performance of different alternate designs, so that you can reject possible designs out-of-hand, or choose which alternatives to consider more carefully.

2. *Microbenchmarking*. If you can identify the bottleneck operation for a given resource, then you can construct a micro-benchmark that compares the performance of different implementations of that operation. This works in tandem with your intuition: the more microbenchmarking you do, the better your intuition for system performance becomes.

Q68. What do we mean by version vectors? How does it help in casuality tracking solution? Mention different approaches of casuality tracking?

A **version vector** is a mechanism for tracking changes to data in a distributed system, where multiple agents might update the data at different times. The version vector allows the participants to determine if one update preceded another (happened-before), followed it, or if the two updates happened concurrently (and therefore might conflict with each other). In this way, version vectors enable causality tracking among data replicas and are a basic mechanism for optimistic replication. In mathematical terms, the version vector generates a preorder that tracks the events that precede, and may therefore influence, later updates.

This last writer wins (lww) policy may lead to lost updates. An approach which avoids this, must be able to represent and maintain causally concurrent updates until they can be reconciled. Accurate tracking of concurrent data updates can be achieved by a careful use of well established causality tracking mechanisms [11,14,20,19,2]. In particular, for data storage systems, version vectors (vv) [14] enable the system to compare any pair of replica versions and detect if they are equivalent, concurrent or if one makes the other obsolete

Approaches for causality tracking

a correct tracking of concurrent updates is essential to allow all updates to be considered for conflict resolution.

Last Writer Wins (lww) In systems that enforce a lww policy, such as Cassandra, concurrent updates are not represented in the stored state and only the last update prevails.

Q69. Asynchronous Processing in a web application

Taken from

http://blog.codepath.com/2012/11/15/asynchronous-processing-in-web-applications-part-1-a-database-is-not-a-queue/

Asynchronous Processing in Web Applications, Part 1: A Database Is Not a Queue

November 15, 2012 by Nathan Esquenazi

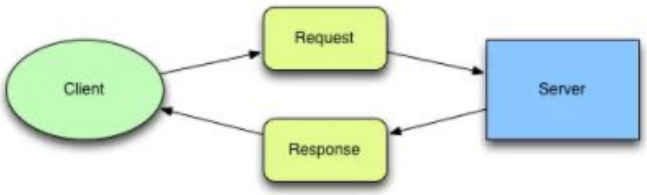
When hacking on web applications, you will inevitably find certain actions that are taking too long and as a result must be pulled out of the http request / response cycle. In other cases, applications will need an easy way to reliably communicate with other services in your system architecture.

The specific reasons will vary; perhaps the website has a real-time element, there's a live chat feature, we need to resize and process images, we need to slice up and transcode video, do analysis of our logs, or perhaps just send emails at a high volume. In all the cases though, asynchronous processing becomes important to your operations.

Fortunately, there are a variety of libraries across all platforms intended to provide asynchronous processing. In this series, I want to explore this landscape, understand the solutions available, how they compare and more importantly how to pick the right one for your needs.

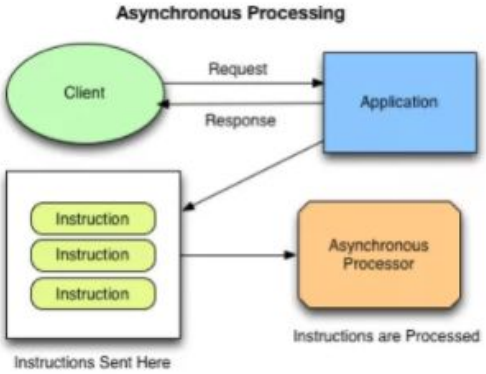
Asynchronous Processing

Let's start by understanding asynchronous processing a bit better. Web applications undoubtedly have a great deal of code that executes as part of the HTTP request/response cycle. This is suitable for faster tasks that can be done within hundreds of milliseconds or less. However, any processing that would take more then a second or two will ultimately be far too slow for synchronous execution. In addition, there is often processing that needs to be scheduled in the future and/or processing that needs to affect an external service.



HTTP Request Response Cycle

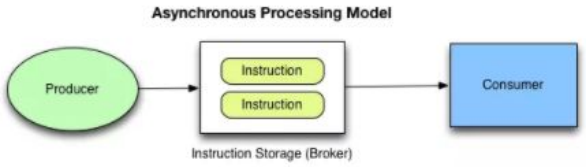
In these cases when we have a task that needs to execute but that is not a candidate for synchronous processing, the best course of action is to move the execution outside the request/response cycle. Specifically, we can have the synchronous web app simply notify another separate program that certain processing needs to be done at a later time.



Now, instead of the task running as a part of the actual web response, the processing runs separately so that the web application can respond quickly to the request. In order to achieve asynchronous processing, we need a way to allow multiple separate processes to pass information to one another. One naive approach might be to persist these notifications into a traditional database and then retrieve them for processing using another service.

Why not a database?

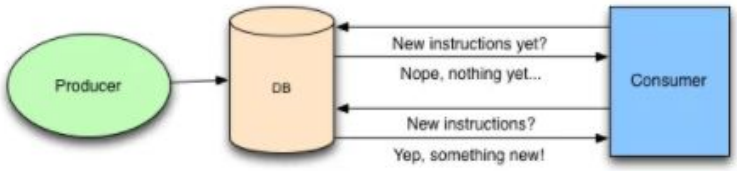
There are many good reasons why a traditional database is not well-suited for cases of asynchronous processing. Often you might be tempted to use a database because ther can be an understandable reluctance to introduce new technologies into a web stack. This leads people to try to use their RDBMS as a way to do background processing or service communication. While this can often appear to 'get the job done', there are a number of limitations and concerns that should not be overlooked.



There are two aspects to any asynchronous processing: the service(s) that **creates** processing tasks and the service(s) that **consume** and process these tasks accordingly. In the case of using a database as a method for this, typically there would be a database table which has records that represent notified tasks and then a flag to represent which state the task is in and whether the task is completed.

Polling Can Hurt

The first issue revolves around how to best consume and process these tasks stored within a table. With a traditional database this typically means a service that is constantly querying for new processing tasks or messages. The service may have a database poll interval of every second or every minute, but the service is constantly asking the database if there has been an update. Polling the database for processing has several downsides. You might have a short interval for polling and be hammering your database with constant queries. Alternatively, you could perhaps set a long interval in which case there will be many unnecessary processing delays. In the event of having multiple processing steps, the fastest route through your system has now been delayed to the sum of all the different polling intervals.



Polling requires very fast and frequent queries to the table to be most effective, which adds a significant load to the database even at a medium volume. Even worse, a given database table is typically prepared to be fast at adding data, updating data or querying data, but almost never all three on the same table. If you are constantly inserting, updating and querying, race conditions and deadlocks become inevitable. These 'locks' occur because multiple consumers will all be "fighting" with each other over the same table and with the 'producers' constantly adding new items. You will see load increase for the database, performance decrease and pile-ups become increasingly common as the volume scales up.

Manual Cleanup

In addition to that, clearing old jobs can be troublesome as well because at even a medium volume, there will be many tasks in the database table. At certain intervals, the old completed tasks need to be removed otherwise the table will grow quite large. Unfortunately, this has to be performed manually and deletes are even often not particularly efficient for tables especially when being removed so frequently in conjunction with updates and queries all happening at the same time.

Scaling Won't Be Easy

In the end, while a database will appear to work at first as a simple way to send background instructions to be processed, this approach will come back to bite you. The many limitations such as constant polling, manual cleanups, deadlocks, race conditions and manual handling of failed processing should make it fairly clear that a database table is not a scalable strategy for this type of processing.

Takeaways

The takeaway here is that asynchronous processing is useful whether you need to send an sms, validate emails, approve posts, process orders, or just pass information between services. This type of background processing is simply not a task that a traditional RDBMS is best-suited to solve. While there are exceptions to this rule with PostgreSQL and its excellent notify support, there is a different class of software that was built from scratch for asynchronous processing use cases.

This type of software is called a message queue and you should strongly consider using this instead for a far more reliable, efficient and scalable way to perform asynchronous processing tasks. In the next part of this series, we will explore several different message queues in detail to understand how they work as well as how to contrast and compare the various options. Hope that this introduction was helpful and please let me know what you'd like to have covered in future parts of this series.

Continue to Part 2: Developers Need To Understand Message Queues

Q70. Developers need to understand Message Queues.

Taken from --

http://blog.codepath.com/2013/01/06/asynchronous-processing-in-web-applications-part-2-developers-need-to-understand-message-queues/

Asynchronous Processing in Web Applications, Part 2: Developers Need to Understand Message Queues

January 6, 2013 by Nathan Esquenazi

In the first part of this series, we explained asynchronous processing, when you might need to use it and why leveraging a database for that purpose is not necessarily the best option. In this post, we will explore a smarter approach to asynchronous processing using "message queues".

Message Queues

Given the reasons that a traditional database is not the right approach for asynchronous processing, let's take a look at the why a message queue might be a smarter choice. With a message queue, we can efficiently support a significantly higher volume of concurrent messages, messages are pushed in real-time instead of periodically polled, messages are automatically cleaned up after being received and we don't need to worry about any pesky deadlocks or race conditions. In short, message queues present a fundamentally sound strategy for handling high-volume asynchronous processing.

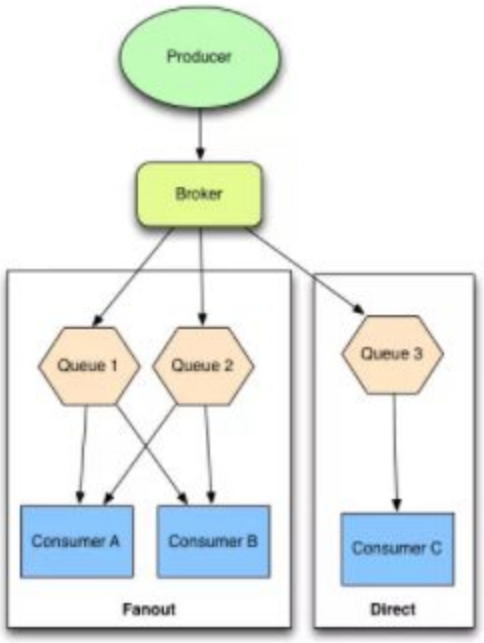
Before we continue, it is important to mention that I am **not** claiming that a database can never be used as a queue. In fact, every database can be made to work at a low or medium volume as a queue given enough time and effort of a clever developer. Moreover, there are actually databases such as PostgreSQL that have additional support for queueing and solid libraries that can make this a manageable solution. If you are processing a low to medium volume of messages largely intended to be used as a job queue and you already use postgreSQL to store your data, there are undoubtedly cases where avoiding a separate message queue is the best choice at least until you feel pain. That said a message queue can be quite a bit more versatile, powerful and flexible depending on your needs. With that said, let's dive into the world of message queues.

What is a message queue?

At the simplest level, a message queue is a way for applications and discrete components to send messages between one another in order to reliably communicate. Message queues are typically (but not always) 'brokers' that facilitate message passing by providing a protocol or interface which other services can access. This interface connects **producers** which create messages and the **consumers** which then process them. Within the context of a web application, one common case is that the **producer** is a client application (i.e Rails or Sinatra) that creates messages based on interactions from the user (i.e user signing up). The consumer in that case is typically daemon processes (i.e rake tasks) that can then process the arriving messages. In other cases, various subsystems will need to communicate with each other via these messages.

Features

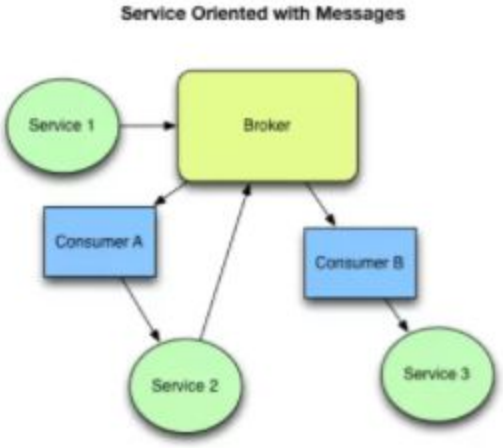
On top of the fundamental ability to pass messages quickly and reliably, most message queues offer additional complementary features. For instance, **multiple separate queues** can allow different messages to be passed to different queues. This allows the messages of different types to be consumed by different services. The consumer processing email sending requests can be on a totally different queue (and/or server) then the one that resizes uploaded images. Message delivery behavior will often vary depending on your need. Certain messages will go from one producer to a single consumer (direct) while other times the message is sent to multiple different listening consumers (fanout).



Another critical feature of message queues is robustness and reliability through **persistence strategies**. In order to keep reliability of the messages high, most message queues offer the ability to persist all messages to disk until they have been received and completed by the consumer(s). Even if the applications or the message queue itself happens to crash, the messages are safe and will be accessible to consumers as soon as the system is operational. In contrast, transient tasks performed synchronously in a web app or on a thread in memory will be lost if anything goes awry. This is especially relevant when dealing with deployments and updating code since restarting components no longer put your messages or tasks at risk.

Another critical feature of message queues is robustness and reliability through **persistence strategies**. In order to keep reliability of the messages high, most message queues offer the ability to persist all messages to disk until they have been received and completed by the consumer(s). Even if the applications or the message queue itself happens to crash, the messages are safe and will be accessible to consumers as soon as the system is operational. In contrast, transient tasks performed synchronously in a web app or on a thread in memory will be lost if anything goes awry. This is especially relevant when dealing with deployments and updating code since restarting components no longer put your messages or tasks at risk.

In addition, message queues can give you a **greater visibility** into the volume of messages. At the minimum, since there's typically a broker for messages, inspecting the tasks being processed at any given time is much easier then with synchronous processing. Similarly, handling a high volume of tasks to process is simpler since you can just **horizontally scale** your message consumers to handle higher loads with minimal fuss. As an added bonus, the application itself now doesn't have to deal with these tasks, so higher volumes of tasks won't slow down the response times of the front-end web app.



Arguably though the most powerful reason to introduce a message queue is the architectural benefits that their use can afford. In particular, when you have message queues that enable lightweight communication between any number of disparate services, the ability to separate your application into many small subsystems is much easier which will often improve your architecture in several ways including making the individual pieces easier to maintain, test, debug and scale. In addition, with a services oriented approach made possible with message queues, you can easily have multiple teams working along well-defined boundary points and even using different tools or stacks when necessary.

Considerations

All that said, message queues are not a panacea and like all tools have their downsides. Setting up and configuring message queues, especially more complicated ones can add a lot of moving parts to your application. Often in small apps, you don't actually need to introduce that overhead early on and can instead slowly offload tasks to message queues as the volume of traffic increases over time. Also, with a traditional message queue, error handling is sometimes a very manual effort if a message or task fails and communicating with message queues adds certain complexity into your application logic. In addition, for more general queues, you often have to define your own state machine for messages. That is, a message that needs to be passed to three different services for processing requires you to manually architect a queue workflow that supports that. Nonetheless, for asynchronous processing of many types, a message queue is often the best tool for the job depending on your needs.

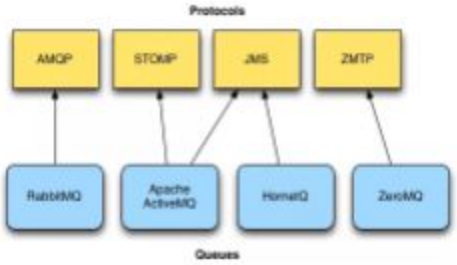
Still, which message queue should we use then? What options and alternatives exist? Why would we pick one over another? What are the differences?

Comparing Message Queues

Overview

The good news and the bad news is that there are a lot of message queues to choose from. In fact, there are dozens of message queues with all sorts of names, features, and different pros and cons such as sparrow, starling, kestrel, kafka, Amazon SQS, and many more that will not be discussed here at length.

The easiest place to start is to explain the emerging open standard protocols for message queues which are AMQP and STOMP. These are the most popular message queue standards around today and many of the message queues implement one or both of these protocols. In addition, there is also JMS (Java Message Service) which is widely used on-top of the JVM. For completeness, there is also MSMQ which is not covered in this post but is the defacto queue for most .NET applications.

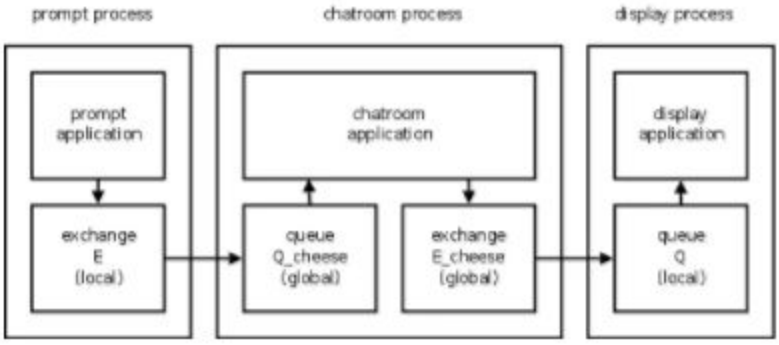


There are certain generic message queue implementations that have emerged as well, many of which built on top of the aforementioned protocols. The most popular and well-supported general purpose message queues available today are RabbitMQ, Apache ActiveMQ and ZeroMQ.

How do these options compare? Well, fortunately all three tools are being actively developed and maintained, have a large user base, good documentation and decent client libraries across many languages. So in a way, you can't really go wrong with any of these options. However, let's spend a moment understanding how they compare with one another. In the end, the one to use depends on the needs of your application and the required flexibility.

ZeroMQ

The interesting thing to understand is that ZeroMQ is actually not so much a pre-packaged message queue like the others but instead acts as a framework for building message queues. ZeroMQ focuses mostly on just passing the messages very efficiently over the wire while RabbitMQ acts as a full-fledged 'broker' which handles persisting, filtering and monitoring messages. ZeroMQ has no broker built in which means that it does not have a central dispatcher to manage your messages and is really not a "full service" message queue.



Think of ZeroMQ as it's own toolbox or framework for creating message queues tailored to your own needs. An example is given above for using ZeroMQ to create a basic instant messaging service. RabbitMQ on the other hand tries to be a more complete queue implementation. RabbitMQ is much more packaged and as such requires less configuration and setup overhead for typical use cases.

Of course, with RabbitMQ or ActiveMQ, the broker and persistence built in adds quite a bit of overhead but those libraries choose to sacrifice raw speed to provide a much richer feature set with less manual tinkering. ZeroMQ is an excellent solution when you want more control or just want to do it yourself. In other cases where you just want to use a queue for typical use cases and you are willing to accept the higher overhead, you should consider RabbitMQ or ActiveMQ.

RabbitMQ and ActiveMQ

Comparing RabbitMQ with ActiveMQ is closer to a head-on comparison because they are solving similar problems. However, the differences here are mostly in the details. ActiveMQ is built in Java on the JMS (Java Message Service) and is very frequently used within applications on the JVM (Java, Scala, Clojure, et al). ActiveMQ also supports STOMP which provides support for Ruby, PHP and Python. RabbitMQ is built on Erlang, powered by AMQP and is used frequently with applications within Erlang, Python, PHP, Ruby, et al.

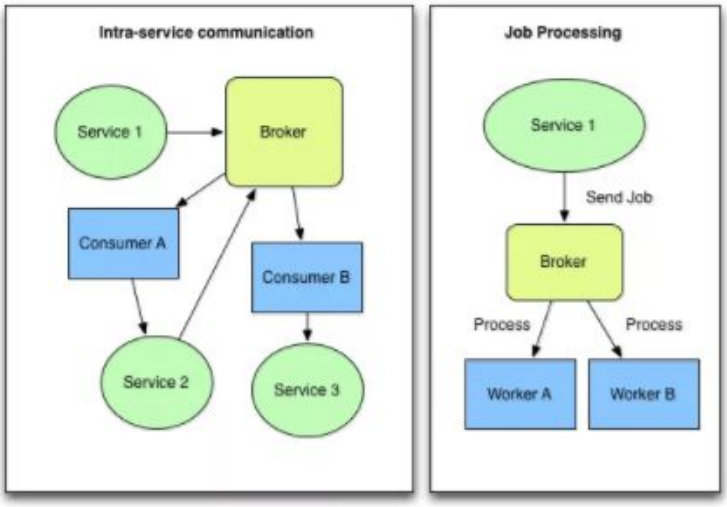
As you might expect, the developers preferences influenced the choices they make throughout the queues. For example, the configuration for ActiveMQ is in XML and the routing of messages is handled with custom rules defined by ActiveMQ. In contrast, the configuration of RabbitMQ is through an Erlang syntax and the advanced routing and configuration follows standard AMQP specifications. The protocols (AMQP vs JMS) used by each queue have certain underlying differences as well. One key difference is that in AMQP a producer sends a message to the broker without knowing the intended distribution strategy while in JMS the producer is aware of the strategy to be used explicitly.

The key takeaway here is that for a general purpose messaging queue to handle all your messaging needs and supporting most advanced requirements, you could use any of the three options listed above. However, my preference and recommendation in *most* cases for a Ruby or Python web application is to select **RabbitMQ** on AMQP. Conversely, if you are building on the JVM stack with Scala or Java, then my recommendation would be to use **ActiveMQ**. In either case, if you are looking for a customizable general message queue, then you won't be disappointed. Of course, don't be too quick to dismiss **ZeroMQ** if you are looking for raw speed and are interested in a light-weight, do-it-yourself protocol for delivering messages.

Specialized Queues

Incidentally, this is not the end of the story though. While message queues are incredibly helpful here, the major ones listed were built to be generic and general purpose. That is, they don't solve one particular use case but instead support a wide plethora of use cases requiring varying levels of configuration. These 'industrial strength' message queues can power chat rooms, instant messaging services, inter-service communication, or even multiplayer online games.

For the average web application though, the requirements are very different. To understand your requirements, ask yourself if you are sending messages to communicate between different services in your application or if you just want to process simple background jobs. In the latter case, we certainly don't need the most powerful or flexible message queue nor the expensive associated setup costs.



Most popular web applications really only need a way to do background job processing and offload tasks to an asynchronous queue. These more specific and constrained requirements open up the possibility for a *lighter-weight* message queue that is easier to use and focused on doing one thing well.

Takeaways

In this article we have covered the features of a message queue, how they work, popular message queue libraries and how to understand if you need a general message queue or a more specialized one. Hopefully you now understand the potential benefits introducing a message queue into your system architecture but also the tradeoffs and overhead of adding it prematurely. In addition, you should be able to see how a message queue works complementary to a traditional database and how queues used appropriately can help improve the modularity, maintainability and scalability of your applications.

In the next part of this series, we will explore a *specialized* message queue specifically built to process background jobs called a **work queue**. We will explore how a work queue operates, what features they typically have, how to scale them and which are most popular today. Hope that this introduction was helpful and please let me know what topics or issues you'd like to have covered in future parts of this series.

Q71. What is Message Queuing?

Taken from –

https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html

---

Message queuing allow applications to communicate by sending messages to each other. The message queue provide a temporary message storage when the destination program is busy or not connected.

In this blog article I will explain message queueing. I will explain what it is, how it can be used and benefits achieved when using message queues.

A **queue** is a line of things waiting to be handled - in sequential order starting at the beginning of the line. A message queue is a queue of messages sent between applications. It includes a sequence of work objects that are waiting to be processed.

A **message** is the data transported between the sender and and the receiver application, it's essentially a byte array with some headers on top. A message example could be anything that tells one system to start processing a task, information about a finished task or a plain message.

The basic architecture of a **message queue** is simple, there are client applications called producers that create messages and deliver them to the message queue. An other application, called consumer, connect to the queue and get the messages to be processed. Messages placed onto the queue are stored until the consumer retrieves them.

Message queues

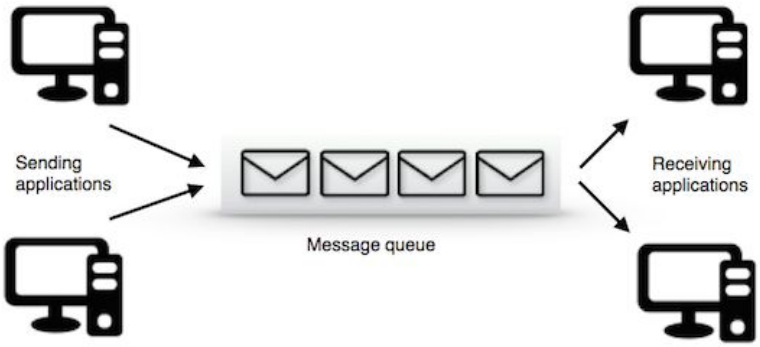
A message queue provide an **asynchronous communications protocol,** a system that puts a message onto a message queue does not require an immediate response to continue processing. Email is probably the best example of asynchronous messaging. When an email is sent can the sender continue processing other things without an immediate response from the receiver. This way of handling messages **decouple** the producer from the consumer. The producer and the consumer of the message do not need to interact with the message queue at the same time.

DECOUPLING AND SCALABILITY

Decoupling is used to describe how much one piece of a system relies on another piece of the system. Decoupling is the process of separating them so that their functionality will be more self contained.

A decoupled system are achieved when two or more systems are able to communicate without being connected. The systems can remain completely autonomous and unaware of each other. Decoupling is often a sign of a computer system that is well structured. It is usually easier to maintain, extend and debug.

If one process in a decoupled system fails processing messages from the queue, other messages can still be added to the queue and be processed when the system has recovered. You can also use a message queue to delay processing; A producer post messages to a queue. At the appointed time, the receivers are started up and process the messages in the queue. Messages in queue can be stored-and-forwarded and the message be redelivered until the message is processed.



Instead of building one large application, is it beneficial to decouple different parts of your application and only communicate between them asynchronously with messages. That way different parts of your application can evolve independently, be written in different languages and/or maintained by separated developer teams.

A message queue will keep the processes in your application separated and independent of each other. The first process will never need to invoke another process, or post notifications to another process, or follow the process flow of the other processes. It can just put the message on the queue and then continue processing. The other processes can also handle their work independently. They can take the messages from the queue when they are able to process them. This way of handling messages creates a system that is easy to maintain and easy to scale.

Message queuing - a simple use case

Imagine that you have a web service that receives many requests every second, where no request is afford to get lost and all requests needs to be processed by a process that is time consuming.

Imagine that your web service always has to be highly available and ready to receive new request instead of being locked by the processing of previous received requests. In this case it is ideal to put a queue between the web service and the processing service. The web service can put the "start processing"-message on a queue and the other process can take and handle messages in order. The two processes will be decoupled from each other and does not need to wait for each other. If you have a lot of requests coming in a short amount of time, the processing system will be able to process them all anyway. The queue will persist requests if their number becomes really huge.

You can then imagine that your business and your workload is growing and you need to scale up your system. All that is needed to be done now is to add more workers, receivers, to work off the queues faster.

RabbitMQ

If you do start to consider a queue-based solution, CloudAMQP offers hosting of the message queue RabbitMQ. RabbitMQ is open source message-oriented middleware that implements the Advanced Message Queuing Protocol (AMQP). AMQP have features like queuing, routing, reliability and security. You can read more about CloudAMQP here.

Q72. RabbitMQ?

Taken from -- https://www.rabbitmq.com/getstarted.html

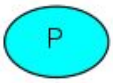
Introduction

RabbitMQ is a message broker: it accepts and forwards messages. You can think about it as a post office: when you put the mail that you want posting in a post box, you can be sure that Mr. Postman will eventually deliver the mail to your recipient. In this analogy, RabbitMQ is a post box, a post office and a postman.

The major difference between RabbitMQ and the post office is that it doesn't deal with paper, instead it accepts, stores and forwards binary blobs of data – *messages*.

RabbitMQ, and messaging in general, uses some jargon.

Producing means nothing more than sending. A program that sends messages is a *producer*:



A queue is the name for a post box which lives inside RabbitMQ. Although messages flow through RabbitMQ and your applications, they can only be stored inside a *queue*. A *queue* is only bound by the host's memory & disk limits, it's essentially a large message buffer. Many *producers* can send messages that go to one queue, and many *consumers* can try to receive

Prerequisites

This tutorial assumes RabbitMQ is installed and running on `localhost` on standard port (`5672`). In case you use a different host, port or credentials, connections settings would require adjusting.

Where to get help

If you're having trouble going through this tutorial you can contact us through the mailing list.

Q73. Scalability For Dummies ?

Taken from ....

http://www.lecloud.net/post/7295452622/scalability-for-dummies-part-1-clones

Part -1 : Clones

Scalability for Dummies - Part 1: Clones

Just recently I was asked what it would take to make a web service massively scalable. My answer was lengthy and maybe it is also for other people interesting. So I share it with you here in my blog and split it into parts to make it easier to read. New parts are released on a regular basis. Have fun and your comments are always welcomed!

The other parts of the series "Scalability for Dummies" you can (soon) find here.

Part 1 - Clones

Public servers of a scalable web service are hidden behind a load balancer. This load balancer evenly distributes load (requests from your users) onto your group/cluster of application servers. That means that if, for example, user Steve interacts with your service, he may be served at his first request by server 2, then with his second request by server 9 and then maybe again by server 2 on his third request.

Steve should always get the same results of his request back, independent what server he "landed on". That leads to the first golden rule for scalability: every server contains exactly the same codebase and does not store any user-related data, like sessions or profile pictures, on local disc or memory.

Sessions need to be stored in a centralized data store which is accessible to all your application servers. It can be an external database or an external persistent cache, like Redis. An external persistent cache will have better performance than an external database. By external I mean that the data store does not reside on the application servers. Instead, it is somewhere in or near the data center of your application servers.

But what about deployment? How can you make sure that a code change is sent to all your servers without one server still serving old code? This tricky problem is fortunately already solved by the great tool Capistrano. It requires some learning, especially if you are not into Ruby on Rails, but it is definitely both the effort.

After "outsourcing" your sessions and serving the same codebase from all your servers, you can now create an image file from one of these servers (AWS calls this AMI - Amazon Machine Image.) Use this AMI as a "super-clone" that all your new instances are based upon. Whenever you start a new instance/clone, just do an initial deployment of your latest code and you are ready!

Q74. Scalability for Dummies part -2 Database.

Taken from ---

http://www.lecloud.net/post/7994751381/scalability-for-dummies-part-2-database

Scalability for Dummies - Part 2: Database

After following Part 1 of this series, your servers can now horizontally scale and you can already serve thousands of concurrent requests. But somewhere down the road your application gets slower and slower and finally breaks down. The reason: your database. It's MySQL, isn't it?

Now the required changes are more radical than just adding more cloned servers and may even require some boldness. In the end, you can choose from 2 paths:

Path #1 is to stick with MySQL and keep the "beast" running. Hire a database administrator (DBA,) tell him to do master-slave replication (read from slaves, write to master) and upgrade your master server by adding RAM, RAM and more RAM. In some months, your DBA will come up with words like "sharding", "denormalization" and "SQL tuning" and will look worried about the necessary overtime during the next weeks. At that point every new action to keep your database running will be more expensive and time consuming than the previous one. You might have been better off if you had chosen Path #2 while your dataset was still small and easy to migrate.

Path #2 means to denormalize right from the beginning and include no more Joins in any database query. You can stay with MySQL, and use it like a NoSQL database, or you can switch to a better and easier to scale NoSQL database like MongoDB or CouchDB. Joins will now need to be done in your application code. The sooner you do this step the less code you will have to change in the future. But even if you successfully switch to the latest and greatest NoSQL database and let your app do the dataset-joins, soon your database requests will again be slower and slower. You will need to introduce a cache.

Q75. Scalability for Dummies Part 3 – Cache.

Taken from ---

http://www.lecloud.net/post/9246290032/scalability-for-dummies-part-3-cache

Scalability for Dummies - Part 3: Cache

After following Part 2 of this series, you now have a scalable database solution. You have no fear of storing terabytes anymore and the world is looking fine. But just for you. Your users still have to suffer slow page requests when a lot of data is fetched from the database. The solution is the implementation of a cache.

With "cache" I always mean in-memory caches like Memcached or Redis. Please **never do file-based caching**, it makes cloning and auto-scaling of your servers just a pain.

But back to in-memory caches. A cache is a simple key-value store and it should reside as a buffering layer between your application and your data storage. Whenever your application has to read data it should at first try to retrieve the data from your cache. Only if it's not in the cache should it then try to get the data from the main data source. Why should you do that? Because **a cache is lightning-fast**. It holds every dataset in RAM and requests are handled as fast as technically possible. For example, Redis can do several hundreds of thousands of read operations per second when being hosted on a standard server. Also writes, especially increments, are very, very fast. Try that with a database!

There are 2 patterns of caching your data. An old one and a new one:

#1 - Cached Database Queries

That's still the most commonly used caching pattern. Whenever you do a query to your database, you store the result dataset in cache. A hashed version of your query is the cache key. The next time you run the query, you first check if it is already in the cache. The next time you run the query, you check at first the cache if there is already a result. This pattern has several issues. The main issue is the expiration. It is hard to delete a cached result when you cache a complex query (who has not?). When one piece of data changes (for example a table cell) you need to delete all cached queries who may include that table cell. You get the point?

#2 - Cached Objects

That's my strong recommendation and I always prefer this pattern. In general, see your data as an object like you already do in your code (classes, instances, etc.). Let your class assemble a dataset from your database and then store the complete instance of the class or the assembed dataset in the cache. Sounds theoretical, I know, but just look how you normally code. You have, for example, a class called "Product" which has a property called "data". It is an array containing prices, texts, pictures, and customer reviews of your product. The property "data" is filled by several methods in the class doing several database requests which are hard to cache, since many things relate to each other. Now, do the following: when your class has finished the "assembling" of the data array, directly store the data array, or better yet the complete instance of the class, in the cache! This allows you to easily get rid of the object whenever something did change and makes the overall operation of your code faster and more logical.

And the best part: it makes asynchronous processing possible! Just imagine an army of worker servers who assemble your objects for you! The application just consumes the latest cached object and nearly never touches the databases anymore!

Some ideas of objects to cache:

- user sessions (never use the database!)
- fully rendered blog articles
- activity streams
- user<->friend relationships

As you maybe already realized, I am a huge fan of caching. It is easy to understand, very simple to implement and the result is always breathtaking. In general, I am more a friend of Redis than Memcached, because I love the extra database-features of Redis like persistence and the built-in data structures like lists and sets. With Redis and a clever key'ing there may be a chance that you even can get completly rid of a database. But if you just need to cache, take Memcached, because it scales like a charm.

Happy caching!

Q76. Scalability for dummies part 4 – Asynchronism.

Taken from --

http://www.lecloud.net/post/9699762917/scalability-for-dummies-part-4-asynchronism

Scalability for Dummies - Part 4: Asynchronism

This 4th part of the series starts with a picture: please imagine that you want to buy bread at your favorite bakery. So you go into the bakery, ask for a loaf of bread, but there is no bread there! Instead, you are asked to come back in 2 hours when your ordered bread is ready. That's annoying, isn't it?

To avoid such a "please wait a while" - situation, asynchronism needs to be done. And what's good for a bakery, is maybe also good for your web service or web app.

In general, there are two ways / paradigms asynchronism can be done.

Async #1
Let's stay in the former bakery picture. The first way of async processing is the "bake the breads at night and sell them in the morning" way. No waiting time at the cash register and a happy customer. Referring to a web app this means doing the time-consuming work in advance and serving the finished work with a low request time.

Very often this paradigm is used to turn dynamic content into static content. Pages of a website, maybe built with a massive framework or CMS, are pre-rendered and locally stored as static HTML files on every change. Often these computing tasks are done on a regular basis, maybe by a script which is called every hour by a cronjob. This pre-computing of overall general data can extremely improve websites and web apps and makes them very scalable and performant. Just imagine the scalability of your website if the script would upload these pre-rendered HTML pages to AWS S3 or Cloudfront or another Content Delivery Network! Your website would be super responsive and could handle millions of visitors per hour!

Async #2

Back to the bakery. Unfortunately, sometimes customers has special requests like a birthday cake with "Happy Birthday, Steve!" on it. The bakery can not foresee these kind of customer wishes, so it must start the task when the customer is in the bakery and tell him to come back at the next day. Refering to a web service that means to handle tasks asynchronously.

Here is a typical workflow:

A user comes to your website and starts a very computing intensive task which would take several minutes to finish. So the frontend of your website sends a job onto a job queue and immediately signals back to the user: your job is in work, please continue to the browse the page. The job queue is constantly checked by a bunch of workers for new jobs. If there is a new job then the worker does the job and after some minutes sends a signal that the job was done. The frontend, which constantly checks for new "job is done" - signals, sees that the job was done and informs the user about it. I know, that was a very simplified example.

If you now want to dive more into the details and actual technical design, I recommend you take a look at the first 3 tutorials on the RabbitMQ website. RabbitMQ is one of many systems which help to implement async processing. You could also use ActiveMQ or a simple Redis list. The basic idea is to have a queue of tasks or jobs that a worker can process. Asynchronism seems complicated, but it is definitely worth your time to learn about it and implement it yourself. Backends become nearly infinitely scalable and frontends become snappy which is good for the overall user experience.

If you do something time-consuming, try to do it always asynchronously.

The already released parts of the series "Scalability for Dummies" you can find here.

Q77. Scalable Architectures.

Taken from ---

http://tutorials.jenkov.com/software-architecture/scalable-architectures.html

A scalable architecture is an architecture that can scale up to meet increased work loads. In other words, if the work load all of a sudden exceeds the capacity of your existing software + hardware combination, you can scale up the system (software + hardware) to meet the increased work load.

Scalability Factor

When you scale up your system's hardware capacity, you want the workload it is able to handle to scale up to the same degree. If you double hardware capacity of your system, you would like your system to be able to handle double the workload too. This situation is called "linear scalability".

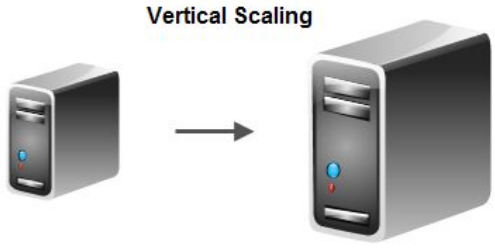
Linear scalability is often not the case though. Very often there is an overhead associated with scaling up, which means that when you double hardware capacity, your system can handle less than double the workload.

The extra workload your system can handle when you scale up your hardware capacity is your system's scalability factor. The scalability factor may vary depending on what part of your system you scale up.

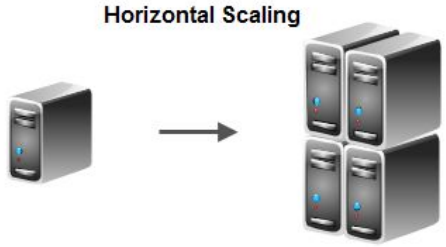
Vertical and Horizontal Scalability

There are two primary ways to scale up a system: Vertical scaling and horizontal scaling.

Vertical scaling means that you scale up the system by deploying the software on a computer with higher capacity than the computer it is currently deployed on. The new computer may have a faster CPU, more memory, faster and larger hard disk, faster memory bus etc. than the current computer.



Horizontal scaling means that you scale up the system by adding more computers with your software deployed on. The added computers typically have about the same capacity as the current computers the system is running on, or whatever capacity you would get for the same money at the time of purchase (computers tend to get more powerful for the same money over time).



Architecture Scalability Requirements

The easiest way to scale up your software from a developer perspective is vertical scalability. You just deploy on a bigger machine, and the software performs better. However, once you get past standard hardware requirements, buying faster CPUs, bigger and faster RAM modules, bigger and faster hard disks, faster and bigger motherboards etc. gets really, really expensive compared to the extra performance you get. Also, if you add more CPUs to a computer, and your software is not explicitly implemented to take advantage of them, you will not get any increased performance out of the extra CPUs.

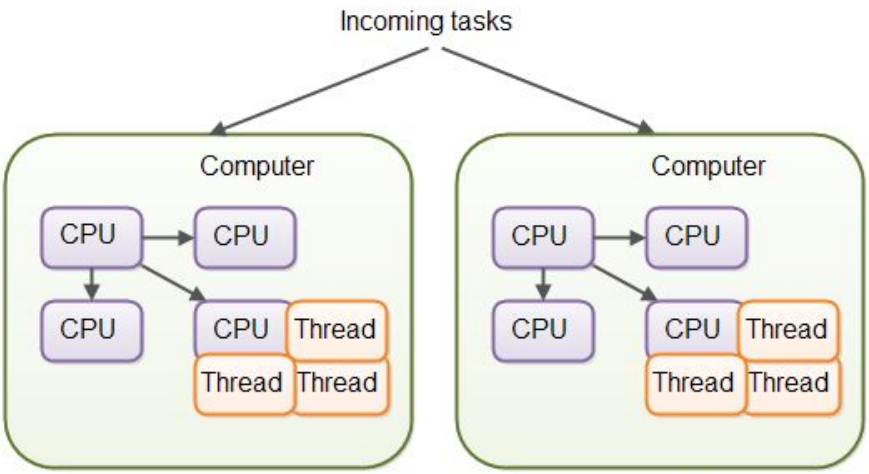
Scaling horizontally is not as easy seen from a software developer's perspective. In order to make your software take advantage of multiple computers (or even multiple CPUs within the same computer), your software need to be able to parallelize its tasks. In fact, the better your software is at parallelizing its tasks, the better your software scales horizontally.

Task Parallelization

Parallelization of tasks can be done at several levels:

- Distributing separate tasks onto separate computers.
- Distributing separate tasks onto separate CPUs on the same computer.
- Distributing separate tasks onto separate threads on the same CPU.

You may also take advantage of special hardware the computers might have, like graphics cards with lots of CPU cores, or InfiniBand network interface cards etc.



Parallizing tasks onto multiple computers, CPUs and threads.

Distributing separate tasks to separate computers is often referred to as "load balancing". Load balancing will be covered in more detail in a separate text.

Executing multiple different applications on the same computer, possible using the same CPU or using different CPUs is referred to as "multitasking". Multitasking is typically done by the operating system, so this is not something software developers need to think too much about. What you need to think about is how to break your application into smaller, independent but collaborating processes, which can also be distributed onto different CPUs or even computers if needed.

Distributing tasks inside the same application to different threads is referred to as "multithreading". I have a separate tutorial on **Java Multithreading** so I will not get deeper into multithreading here.

To be fully parallelizable, a task must be independent of other tasks executing in parallel with it.

To be fully parallelizable tasks must be indedepent of other tasks executing in parallel.

To be fully distributable onto any computer, the task must contain, or be able to access, any data needed to execute the task, regardless of what computer executes the task. Exactly what that means depends on the kind of application you are developing, so I cannot get into deeper detail here.

Q78. An Unorthodox approach to database design – Shards.

http://highscalability.com/blog/2009/8/6/an-unorthodox-approach-to-database-design-the-coming-of-the.html

Once upon a time we scaled databases by buying ever bigger, faster, and more expensive machines. While this arrangement is great for big iron profit margins, it doesn't work so well for the bank accounts of our heroic system builders who need to scale well past what they can afford to spend on giant database servers. In a extraordinary two article series, Dathan Pattishall, explains his motivation for a revolutionary new database architecture--sharding--that he began thinking about even before he worked at Friendster, and fully implemented at Flickr. Flickr now handles more than 1 billion transactions per day, responding in less then a few seconds and can scale linearly at a low cost.

What is sharding and how has it come to be the answer to large website scaling problems?

Information Sources

- Unorthodox approach to database design Part1:History
- Unorthodox approach to database design Part 2:Friendster

What Is Sharding?

While working at Auction Watch, Dathan got the idea to solve their scaling problems by creating a database server for a group of users and running those servers on cheap Linux boxes. In this scheme the data for User A is stored on one server and the data for User B is stored on another server. It's a federated model. Groups of 500K users are stored together in what are called *shards*.

The advantages are:

- **High availability**. If one box goes down the others still operate.
- **Faster queries**. Smaller amounts of data in each user group mean faster querying.
- **More write bandwidth**. With no master database serializing writes you can write in parallel which increases your write throughput. Writing is major bottleneck for many websites.
- **You can do more work**. A parallel backend means you can do more work simultaneously. You can handle higher user loads, especially when writing data, because there are parallel paths through your system. You can load balance web servers, which access shards over different network paths, which are processed by separate CPUs, which use separate caches of RAM and separate disk IO paths to process work. Very few bottlenecks limit your work.

How Is Sharding Different Than Traditional Architectures?

Sharding is different than traditional database architecture in several important ways:

- **Data are denormalized**. Traditionally we normalize data. Data are splayed out into anomaly-less tables and then joined back together again when they need to be used. In sharding the data are denormalized. You store together data that are used together.

This doesn't mean you don't also segregate data by type. You can keep a user's profile data separate from their comments, blogs, email, media, etc, but the user profile data would be stored and retrieved as a whole. This is a very fast approach. You just get a blob and store a blob. No joins are needed and it can be written with one disk write.

- **Data are parallelized across many physical instances**. Historically database servers are scaled up. You buy bigger machines to get more power. With sharding the data are parallelized and you scale by scaling out. Using this approach you can get massively more work done because it can be done in parallel.

- **Data are kept small**. The larger a set of data a server handles the harder it is to cash intelligently because you have such a wide diversity of data being accessed. You need huge gobs of RAM that may not even be enough to cache the data when you need it. By isolating data into smaller shards the data you are accessing is more likely to stay in cache.

Smaller sets of data are also easier to backup, restore, and manage.

- **Data are more highly available**. Since the shards are independent a failure in one doesn't cause a failure in another. And if you make each shard operate at 50% capacity it's much easier to upgrade a shard in place. Keeping multiple data copies within a shard also helps with redundancy and making the data more parallelized so more work can be done on the data. You can also setup a shard to have a master-slave or dual master relationship within the shard to avoid a single point of failure within the shard. If one server goes down the other can take over.

- **It doesn't use replication**. Replicating data from a master server to slave servers is a traditional approach to scaling. Data is written to a master server and then replicated to one or more slave servers. At that point read operations can be handled by the slaves, but all writes happen on the master.

Obviously the master becomes the write bottleneck and a single point of failure. And as load increases the cost of replication increases. Replication costs in CPU, network bandwidth, and disk IO. The slaves fall behind and have stale data. The folks at YouTube had a big problem with replication overhead as they scaled.

Sharding cleanly and elegantly solves the problems with replication.

Some Problems With Sharding

Sharding isn't perfect. It does have a few problems.

- **Rebalancing data**. What happens when a shard outgrows your storage and needs to be split? Let's say some user has a particularly large friends list that blows your storage capacity for the shard. You need to move the user to a different shard.

On some platforms I've worked on this is a killer problem. You had to build out the data center correctly from the start because moving data from shard to shard required a lot of downtime.

Rebalancing has to be built in from the start. Google's shards automatically rebalance. For this to work data references must go through some sort of naming service so they can be relocated. This is what Flickr does. And your references must be invalidateable so the underlying data can be moved while you are using it.

- **Joining data from multiple shards**. To create a complex friends page, or a user profile page, or a thread discussion page, you usually must pull together lots of different data from many different sources. With sharding you can't just issue a query and get back all the data. You have to make individual requests to your data sources, get all the responses, and the build the page. Thankfully, because of caching and fast networks this process is usually fast enough that your page load times can be excellent.

- **How do you partition your data in shards?** What data do you put in which shard? Where do comments go? Should all user data really go together, or just their profile data? Should a user's media, IMs, friends lists, etc go somewhere else? Unfortunately there are no easy answer to these questions.

- **Less leverage**. People have experience with traditional RDBMS tools so there is a lot of help out there. You have books, experts, tool chains, and discussion forums when something goes wrong or you are wondering how to implement a new feature. Eclipse won't have a shard view and you won't find any automated backup and restore programs for your shard. With sharding you are on your own.

- **Implementing shards is not well supported**. Sharding is currently mostly a roll your own approach. LiveJournal makes their tool chain available. Hibernate has a library under development. MySQL has added support for partioning. But in general it's still something you must implement yourself.

Q79. Architecture timelines that power the company wide application?

Google Architecture

http://highscalability.com/google-architecture

Twitter Architecture

https://www.infoq.com/presentations/Twitter-Timeline-Scalability

LiveJournal Architecture

http://highscalability.com/livejournal-architecture

Q80. Facebook-shares-some-secrets-on-making-mysql-scale.

https://gigaom.com/2011/12/06/facebook-shares-some-secrets-on-making-mysql-scale/

When you're storing every transaction for 800 million users and handling more than 60 million queries per second, your database environment had better be something special. Many readers might see these numbers and think NoSQL, but Facebook held a Tech Talk on Monday night explaining how it built a MySQL (s orcl) environment capable of handling everything the company needs in terms of scale, performance and availability.

Over the summer, I reported on Michael Stonebraker's stance that Facebook is trapped in a MySQL "fate worse than death" because of its reliance on an outdated database paired with a complex sharding and caching strategy (read the comments and this follow-up post for a bevy of opinions on the validity of Stonebraker's stance on SQL). Facebook declined an official comment at the time, but last night's night talk proved to me that Stonebraker (and I) might have been wrong.

Keeping up with performance

Kicking off the event, Facebook's Domas Mituzas shared some stats that illustrate the importance of its MySQL user database:

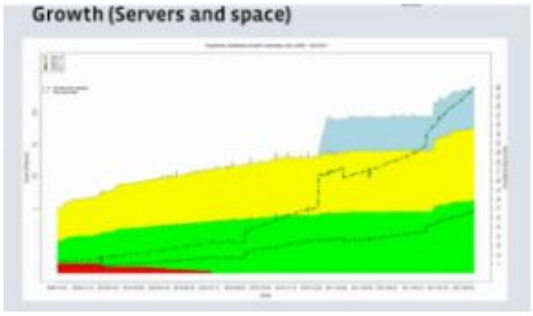
- MySQL handles pretty much every user interaction: likes, shares, status updates, alerts, requests, etc.
- Facebook has 800 million users; 500 million of them visit the site daily.
- 350 million mobile users are constantly pushing and pulling status updates

- 7 million applications and web sites are integrated into the Facebook platform
- User data sets are made even larger by taking into account both scope and time

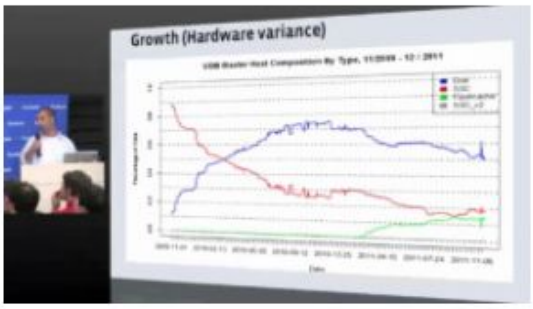
And, as Mituzas pointed out, everything on Facebook is social, so every action has a ripple effect that spreads beyond that specific user. “It’s not just about me accessing some object,” he said. “It’s also about analyzing and ranking through that include all my friends’ activities.” The result (although Mituzas noted these numbers are somewhat outdated) is 60 million queries per second, and nearly 4 million row changes per second.

Facebook shards, or splits its database into numerous distinct sections, because of the sheer volume of the data it stores (a number it doesn’t share), but it caches extensively in order to write all these transactions in a hurry. In fact, most queries (more than 90 percent) never hit the database at all but only touch the cache layer. Facebook relies heavily on the open-source memcached MySQL caching tool, as well as it custom-built Flashcache module for caching data on solid-state drives.

Keeping up with scale



Speaking of drives, and hardware generally, Facebook's Mark Konetchy took the stage after Mituzas to share some data points on the growth of Facebook's MySQL infrastructure. Although he made sure to point out that the "buzzkills at legal" won't let him share actual numbers, he was able to point to 3x server growth across all data centers over the past two years, 7x growth in raw user data, and 20x growth in all user data (which includes replicated data). The median data-set size per physical host has increased almost 5x since Jan. 2010, and maximum data-set size per host has increased 10x.



Konetchy credits the ability to store so much more data per host on software-performance improvements made by Facebook's MySQL team, as well as on better server technology. Facebook's MySQL user database is composed of approximately 60 percent hard disk drives, 20 percent SSDs and 10 percent hybrid HDD-plus-SSD servers running Flashcache.

However, Facebook wants to buy fewer servers while still improving MySQL performance. Looking forward, Konetchy said some primary objectives are to automate the splitting of large data sets onto underutilized hardware, to improve MySQL compression and to move more data to the Hadoop-based HBase data store when appropriate. NoSQL databases such as HBase (which powers Facebook Messages) weren't really around when Facebook built its MySQL environment, so there likely are unstructured or semistructured data currently in MySQL that are better suited for HBase.

With all this growth, why MySQL?

The logical question when one sees rampant growth and performance requirements like this is "Why stick with MySQL?". As Stonebraker pointed out over the summer, both NoSQL and NewSQL are arguably better suited to large-scale web applications than is MySQL. Perhaps, but Facebook begs to differ.

Facebook's Mark Callaghan, who spent eight years as a "principal member of the technical staff" at Oracle (s orcl) , explained that using open-source software lets Facebook operate with "orders of magnitude" more machines than people, which means lots of money saved on software licenses and lots of time put into working on new features (many of which, including the rather-cool Online Schema Change, are discussed in the talk).

Additionally, he said, the patch and update cycles at companies like Oracle are far slower than what Facebook can get by working on issues internally and with an open-source community. The same holds true for general support issues, which Facebook can resolve itself in hours instead of waiting days for commercial support.

On the performance front, Callaghan noted, Facebook might find some interesting things if large vendors allowed it to benchmark their products. But they won't, and they won't let Facebook publish the results, so MySQL it is. Plus, he said, you actually can tune MySQL to perform very fast per node if you know what you're doing — and Facebook has the best MySQL team around. That also helps keep costs down because it requires fewer servers.

Callaghan was more open to using NoSQL databases, but said they're still not quite ready for primetime, especially for mission-critical workloads such as Facebook's user database. The implementations just aren't as mature, he said, and there are no published cases of NoSQL databases operating at the scale o Facebook's MySQL database. And, Callaghan noted, the HBase engineering team at Facebook is quite a bit larger than the MySQL engineering team, suggesting that tuning HBase to meet Facebook's needs is more resource-intensive process than is tuning MySQL at this point.

The whole debate about Facebook and MySQL was never really about whether it should be using it, but rather about how much work it has put into MySQL to make it work at Facebook scale. The answer, clearly, is *a lot*, but Facebook seems to have it down to an art at this point, and everyone appears pretty content with what they have in place and how they plan to improve it. It doesn't seem like a fate worse than death, and if it had to start from scratch, I don't get the impression Facebook would do too much differently, even with the new database offerings available today.

Q81. What is web scale SQL?

http://webscalesql.org/

What is WebScaleSQL?

WebScaleSQL is a collaboration among engineers from several companies that face similar challenges in running MySQL at scale, and seek greater performance from a database technology tailored for their needs.

Our goal in launching WebScaleSQL is to enable the scale-oriented members of the MySQL community to work more closely together in order to prioritize the aspects that are most important to us. We aim to create a more integrated system of knowledge-sharing to help companies leverage the great features already found in MySQL 5.6, while building and adding more features that are specific to deployments in large scale environments. In the last few months, engineers from all four companies have contributed code and provided feedback to each other to develop a new, more unified, and more collaborative branch of MySQL.

But as effective as this collaboration has been so far, we know we're not the only ones who are trying to solve these particular challenges. So we will keep WebScaleSQL open as we go, to encourage others who have the scale and resources to customize MySQL to join in our efforts. And of course we will welcome input from anyone who wants to contribute, regardless of what they are currently working on.

Who is behind WebScaleSQL?

WebScaleSQL currently includes contributions from MySQL engineering teams at Alibaba, Facebook, Google, LinkedIn, and Twitter. Together, we are working to share a common base of code changes to the upstream MySQL branch that we can all use and that will be made available via open source. This collaboration will expand on existing work by the MySQL community, and we will continue to track the upstream branch that is the latest, production-ready release (currently MySQL 5.6).

Q82. queues and message passing

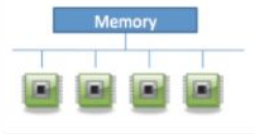
Taken from - **http://web.mit.edu/6.005/www/fa15/**

Objectives

After reading the notes and examining the code for this class, you should be able to use message passing (with synchronous queues) instead of shared memory for communication between threads.

Two models for concurrency

In our introduction to concurrency, we saw two models for concurrent programming: *shared memory* and *message passing*.



- In the **shared memory** model, concurrent modules interact by reading and writing shared mutable objects in memory. Creating multiple threads inside a single Java process is our primary example of shared-memory concurrency.
- In the **message passing** model, concurrent modules interact by sending immutable messages to one another over a communication channel. We've had one example of message passing so far: the client/server pattern, in which clients and servers are concurrent processes, often on different machines, and the communication channel is a network socket.

The message passing model has several advantages over the shared memory model, which boil down to greater safety from bugs. In message-passing, concurrent modules interact *explicitly*, by passing messages through the communication channel, rather than *implicitly* through mutation of shared data. The implicit interaction of shared memory can too easily lead to *inadvertent* interaction, sharing and manipulating data in parts of the program that don't know they're concurrent and aren't cooperating properly in the thread safety strategy. Message passing also shares only immutable objects (the messages) between modules, whereas shared memory *requires* sharing mutable objects, which we have already seen can be a source of bugs.

We'll discuss in this reading how to implement message passing within a single process, as opposed to between processes over the network. We'll use **blocking queues** (an existing threadsafe type) to implement message passing between threads within a process.

Message passing with threads

We've previously talked about message passing between processes: clients and servers communicating over network sockets. We can also use message passing between threads within the same process, and this design is often preferable to a shared memory design with locks.

Use a synchronized queue for message passing between threads. The queue serves the same function as the buffered network communication channel in client/server message passing. Java provides the `BlockingQueue` interface for queues with blocking operations:

In an ordinary `Queue`:

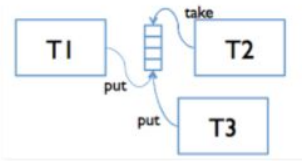
- `add(e)` adds element `e` to the end of the queue.
- `remove()` removes and returns the element at the head of the queue, or throws an exception if the queue is empty.

A `BlockingQueue` extends this interface:

> additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

- `put(e)` *blocks* until it can add element `e` to the end of the queue (if the queue does not have a size bound, `put` will not block).
- `take()` *blocks* until it can remove and return the element at the head of the queue, waiting until the queue is non-empty.

When you are using a `BlockingQueue` for message passing between threads, make sure to use the `put()` and `take()` operations, not ~~`add()`~~ ~~and~~ ~~`remove()`~~.



Analogous to the client/server pattern for message passing over a network is the **producer-consumer design pattern** for message passing between threads. Producer threads and consumer threads share a synchronized queue. Producers put data or requests onto the queue, and consumers remove and process them. One or more producers and one or more consumers might all be adding and removing items from the same queue. This queue must be safe for concurrency.

Java provides two implementations of `BlockingQueue`:

- `ArrayBlockingQueue` is a fixed-size queue that uses an array representation. `put`ting a new item on the queue will block if the queue is full.
- `LinkedBlockingQueue` is a growable queue using a linked-list representation. If no maximum capacity is specified, the queue will never fill up, so `put` will never block.

Unlike the streams of bytes sent and received by sockets, these synchronized queues (like normal collections classes in Java) can hold objects of an arbitrary type. Instead of designing a wire protocol, we must choose or design a type for messages in the queue. **It must be an immutable type.** And just as we did with operations on a threadsafe ADT or messages in a wire protocol, we must design our messages here to prevent race conditions and enable clients to perform the atomic operations they need.

Bank account example

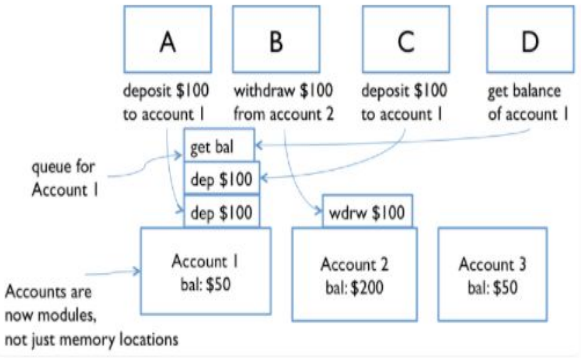
Our first example of message passing was the bank account example.

Each cash machine and each account is its own module, and modules interact by sending messages to one another. Incoming messages arrive on a queue.

We designed messages for `get-balance` and `withdraw`, and said that each cash machine checks the account balance before withdrawing to prevent overdrafts:

```
get-balance
if balance >= 1 then withdraw 1
```

But it is still possible to interleave messages from two cash machines so they are both fooled into thinking they can safely withdraw the last dollar from an account with only $1 in it.



We need to choose a better atomic operation: `withdraw-if-sufficient-funds` would be a better operation than just `withdraw`.

Summary

- Rather than synchronize with locks, message passing systems synchronize on a shared communication channel, e.g. a stream or a queue.
- Threads communicating with blocking queues is a useful pattern for message passing within a single process.