I. FOUNDATIONS OF DATA SYSTEMS

M T W T F S S
Page No.:
Date:
YOUVA

1. RELIABLE, SCALABLE, AND MAINTAINABLE APPLICATIONS

I. Introduction :-

Data-Intensive application - Building Blocks

1. Store data - databases
2. Remember result - caches.
3. Search data by keyword - search indexes
4. Send message to be handled asynchronously - stream processing
5. Crunch a large amount of accumulated data - batch processing.

These are some of data systems which are a successful abstraction. Our purpose is to go through both the principles and the practicalities of data systems and we can use them to build data-intensive applications.

* Fundamentals of data systems - reliability, scalability, maintainability
* design decisions to be considered in data-intensive application

II. Thinking About Data Systems :-

Database and Message queue :-

* similarity is both store data for some time.
* difference is access patterns i.e., performance characteristics Thus both have different implementations.

But we call them (DBs, caches, queues etc) all under an umbrella term - "data system"

Why?

① New tools like Redis, Apache Kafka.
② Data processing and storage needs can't be handled by single tool.

Answers:-

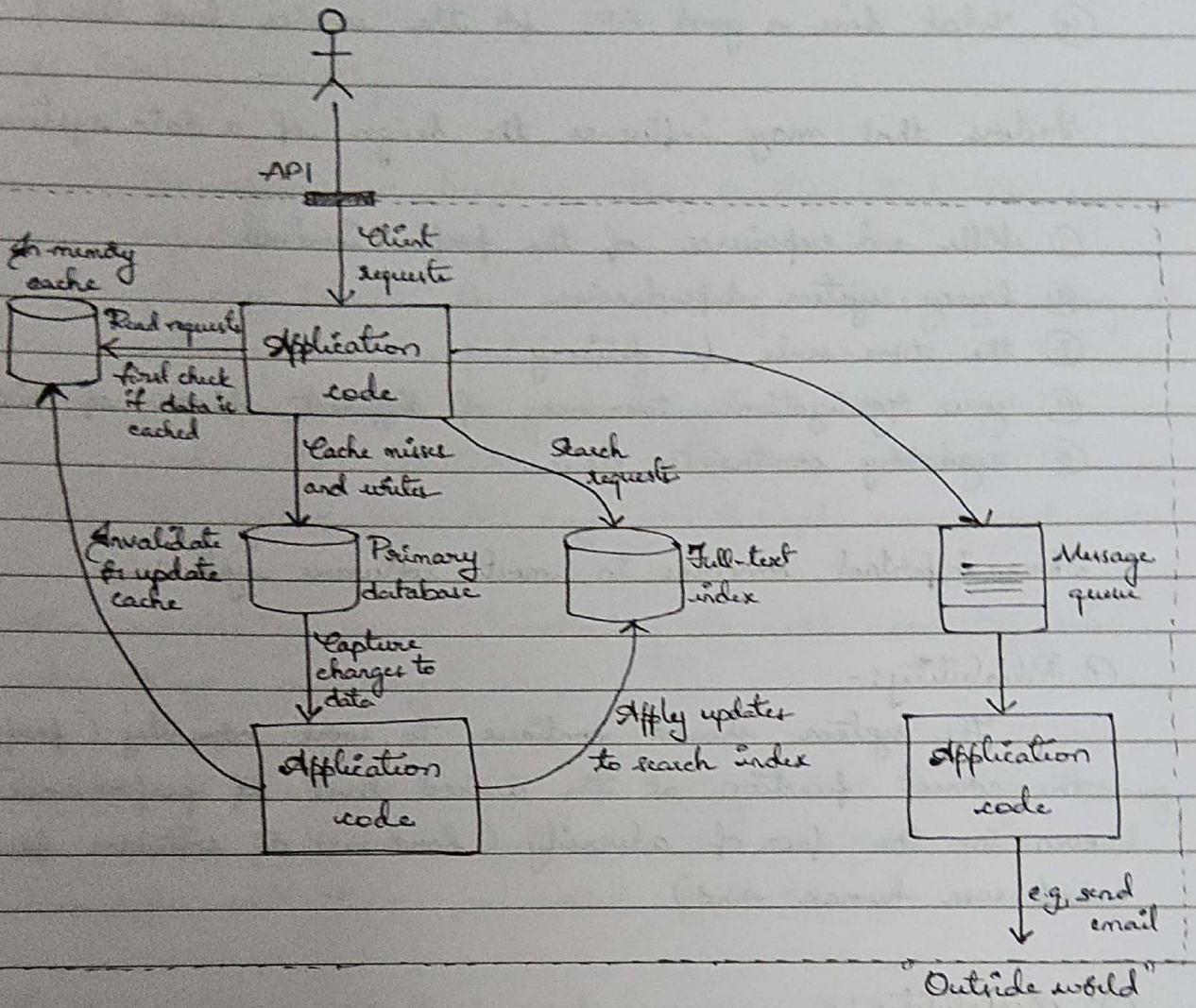
① Redis is not only a datastore but also used as message queue.
Apache Kafka is a message-queue but has a database-like durability guarantee.

② The work (or the entire needs/requirements) is broken down into tasks performed on each single tool, and those different tools are stitched together using application code.

Example:-

You have:
① Application-managed caching layer (Memcache or similar)
② Full-text search server (Elastic Search or Solr) separate from your main database
The application code's responsibility is to keep those caches and indexes in sync with the main database.

All these things clubbed together to look like a data system.
The services interface or API hides those implementation details from clients.
This new special-purpose data system is created from smaller, general-purpose components.



Fig, One possible architecture for a data system that combines several components

This system, for example, guarantees that the cache will be correctly invalidated or updated on writes so that outside clients see consistent results.

Tricky questions :-

① How do you ensure that the data remains correct and complete, even when things go wrong internally?
② How do you provide consistently good performance to clients, even when parts of your system are degraded?

③ How do you scale to handle an increase in load?
④ What does a good API for the service look like?

Factors that may influence the design of a data system

① skills and experience of the people involved
② legacy system dependencies
③ the time-scale for delivery
④ your organization's tolerance of different kinds of risk
⑤ regulatory constraints, etc.

Three important concerns in most software systems:-

① Reliability:-
The system should continue to work correctly (performing the correct function at the desired level of performance) even in the face of adversity (hardware or software faults, and even human error)

② Scalability:-
As the system grows (in data volume, traffic volume, or complexity), there should be reasonable ways of dealing with that growth.

③ Maintainability:-
Overtime, many different people will work on the system (engineering and operations, both maintaining current behavior and adapting the system to new use cases), and they should all be able to work on it productively.

1. RELIABILITY :-

If "Working correctly" means :

1. The application performs the function that the user expected.
2. It can tolerate the user making mistakes or using the software in unexpected ways.
3. Its performance is good enough for the required use case, under the expected load and data volume.
4. The system prevents any unauthorized access and abuse.

then reliability means "continuing to work correctly, even when things go wrong."

Faults and Fault-tolerance : The things that can go wrong are called faults and systems that anticipate faults and can cope with them are called fault-tolerant or resilient.

Fault vs Failure : Fault is not same as Failure.
A fault is defined as one component of the system deviating from its spec, whereas a failure is when the system as a whole stops providing the required service to the user.
It is best to design fault-tolerance mechanisms that prevent faults from causing failures.

Increase the rate of faults by triggering them deliberately. By doing so, you ensure that fault-tolerance machinery is continually exercised and tested, which can increase your confidence that faults will be handled correctly when they occur naturally. Eg :- Netflix Chaos Monkey

III. HARDWARE FAULTS

→ Hard disk crash.
→ RAM becomes faulty.
→ Power Grid has blackout
→ Unplug wrong network cable.

* MTTF - Mean time to failure.
Hard disks have MTTF of about 10 to 50 years.
Thus on a storage cluster with 10,000 disks, we should expect on average one disk to die per day.

* Add redundancy - to reduce failure rates of system.
It cannot completely prevent hardware problems from causing failures, but can often keep a machine running uninterrupted for years.

↳ Disks setup in a RAID configuration.
↳ servers may have dual power supplies and hot-swappable CPUs
↳ Datacenters may have batteries and diesel generators for backup power.

Redundancy means when one component dies, the redundant component can take its place while the broken component is replaced.

Redundancy of hardware component is sufficient when you can quickly restore a backup onto new machine fairly quickly. Thus, multi-machine redundancy was only required by a small number of applications for which high availability was absolutely essential.

As large data volumes and applications are emerging, there are large number of machines, hence the hardware faults. In AWS, multiple VM instances to become unavailable without warning, as the cloud platforms such as AWS are designed to prioritize flexibility and elasticity over single-machine reliability.

Hence, there is a move towards systems that can tolerate the loss of entire machines, by using software fault-tolerance techniques in preference or in addition to hardware redundancy.
Advantage over single-server: A single-server requires planned downtime if you need to reboot the machine (eg., to apply OS security patches), whereas,
a system that can tolerate machine failure can be patched one node at a time, without downtime of entire system (a rolling upgrade)

IV. SOFTWARE ERRORS:-

It is unlikely that a large number of hardware components will fail at the same time.

Another type/class of fault other than Hardware fault is a systematic error.

Harder to anticipate, correlated across nodes, hence they could cause many more system failures.

Examples:-

* Software bug that causes every instance of an application server to crash when given a particular bad input.
Eg. bug in linux kernel caused many applications to hang,

* A runaway process that uses up some shared resource - CPU time, memory, disk space, or network bandwidth.
* A service that the system depends slows down, becomes unresponsive, or starts returning corrupted responses.
* Cascading failures, where a small fault in one component triggers a fault in another component, which in turn triggers further faults.

There is no quick solution to the problem of systematic faults in software. Some steps can be taken:

→ carefully thinking about assumptions and interactions in the system.
→ thorough testing
→ process isolation
→ allowing processes to crash and restart
→ measuring, monitoring, and analyzing system behaviour in production.

If a system is expected to provide some guarantee (for example, in a message queue, that the no. of incoming messages equals the no. of outgoing messages), it can constantly check itself while it is running and raise an alert if a discrepancy is found.

HUMAN ERRORS

Configuration errors by operators were the leading cause of outages, whereas hardware faults (servers or network) caused only 10-25% of outages.

Several approaches to keep systems reliable, inspite of unreliable humans :-

① Design systems in a way that minimizes opportunities for error.

Eg:- well-designed abstractions,
APIs, and
admin interfaces.

② Decouple the places where people make the most mistakes from the places where they can cause failures. Means, providing fully featured non-production sandbox environments where people can explore and experiment safely, using real data, without affecting real users.

③ Test thoroughly at all levels, from unit tests to whole-system integration tests and manual tests. Automated testing is widely used, well understood, and especially valuable for covering corner cases that rarely arise in normal operation.

④ Allow quick and easy recovery from human errors, to minimize the impact in the case of a failure. For example, make it fast to roll back configuration changes, roll out new code gradually (so that any unexpected bugs affect only a small subset of users), and provide tools to recompute data (in case it turns out that the old computation was incorrect)

⑤ Set up detailed and clear monitoring, such as performance metrics and error rates. Monitoring can show us early warning signals and ~~for understanding failures~~ allow us to check whether any assumptions or constraints are being violated. Metrics will be invaluable when problems occur

⑥ Implement good management practices and training — a complex and important aspect.

VI. How IMPORTANT IS RELIABILITY?

Bugs in business applications cause lost productivity (and legal risks if figures are reported incorrectly), and outages of ecommerce sites can have huge costs in terms of lost revenue and damage to reputation.

There are situations in which we may choose to sacrifice reliability in order to reduce development cost (e.g. when developing a prototype product for an unproven market) or operational cost (e.g. for service with a very narrow profit margin) - but we should be very conscious of when we are cutting corners.

VII. SCALABILITY

Scalability is described as system's ability to cope with increased load.

Scalability is not a one-dimensional label. Scalability means considering questions like:

① If the system grows in a particular way, what are our options for coping with the growth?

② How can we add computing resources to handle the additional load?

VIII. DESCRIBING LOAD

Load can be described with a few numbers called load parameters. The best choice of parameters depends on the architecture of your system:

* requests/sec to a web server
* reads to writes ratio in a database
* no. of active users in a chat room.

* hit rate on a cache
etc.,

Example:-

Twitter's data published in 2012.

Two of Twitter's main operations:

① Post tweet: A user can publish a new message to their followers (4.6k requests/sec on average, over 12k requests/sec at peak).

② Home timeline: A user can view tweets posted by the people they follow (300k requests/sec).

Tweets scaling challenge is not primarily due to tweet volume but due to fan-out – each user follows many people and each user is followed by many people.

Two ways of implementing these two operations:

① Posting a tweet simply inserts the new tweet into a global collection of tweets.
When a user requests their home timeline, look up all the people they follow, find all the tweets for each of those users, and merge them (sorted by time).

currently logged in user: 17055506

follows table

follower_id	followee_id
17055506	12

tweets table

id	sender_id	text	timestamp
20	12	just setting up my twitter	1142 974 214

users table

id	screen_name	profile_image
12	jack	123456 7.jpg

Fig 1, Simple relational schema for implementing a Twitter home timeline.

```
SELECT tweets.*, users.* FROM tweets
    JOIN users ON tweets.sender_id = users.id
    JOIN follows ON follows.followee_id = users.id
    WHERE follows.follower_id = current_user.
```

(2) Maintain a cache for each user's home timeline – like a mailbox of tweets for each recipient user. When a user posts a tweet, look up all the people who follow that user, and insert the new tweet into each of their home timeline caches. The request to read the home timeline is then cheap, because its result has been computed ahead of time.

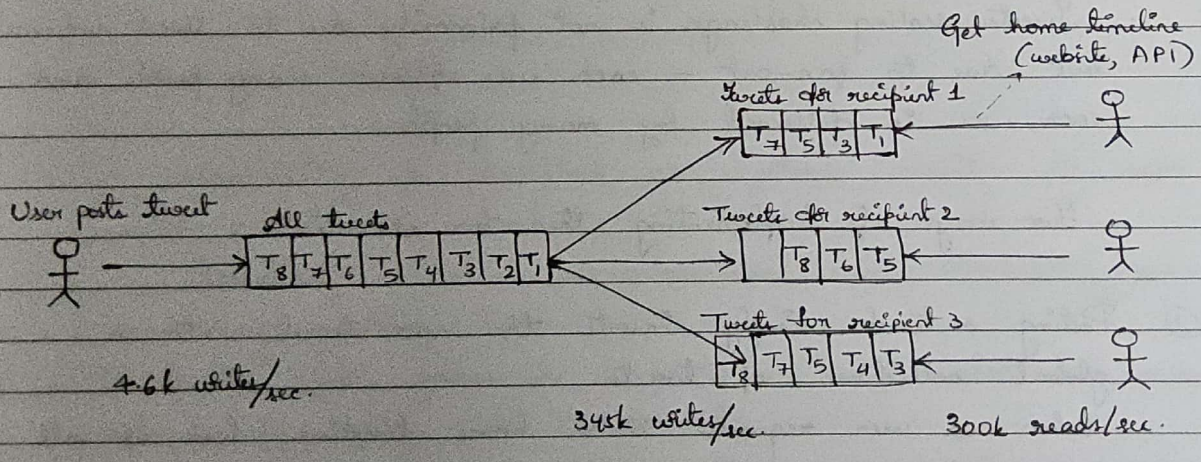


Fig 2, Twitter's data pipeline for delivering tweets to followers, with load parameters as of November 2012.

The first version of Twitter used approach 1, but the systems struggled to keep up with the load of home timeline queries, so the company switched to approach 2. This works better. Reason:

avg rate of published tweets < $\frac{\text{rate of home timeline reads}}{2}$

so its preferrable to do more work at write time and less at read time.

Downside of approach 2 :-

Posting a tweet (at the write side) requires extra work. On an average, a tweet is delivered to about 75 followers, so 4.6 k tweets/sec becomes 345 k writes/sec to home timeline caches.

But some users have over 30 million followers means a single tweet may result in over 30 million writes to home timelines.

Twitter tries to deliver tweets to followers within five seconds - is a significant challenge.

distribution of followers per user is a key load parameter for discussing scalability, since it determines the fan-out load.

Final twist :-

Now that approach 2 is robustly implemented, Twitter is moving to a hybrid approach.

Users like celebrity tweets — approach 1
since more followers are there, writing to follower cache is a lot of work. So the tweets are fetched from database when user queries

Normal users — approach 2
most users' tweets continue to be fanned out to home timelines at the time when posted

M T W T F S S
Page No.:
Date:
YOUVA

IX. Describing Performance

Two ways:

	1st way	2nd way
① increase load parameter	✓	✓
② keep system resources unchanged (CPU, memory, network B/w etc.,)	✓	✓
③ check	how performance affected?	how much to increase the resources to keep performance unchanged?

You need performance numbers to check performance.

① Throughput

In a batch processing system such as Hadoop, we usually care about throughput - the no. of records we can process per second, or the total time it takes to run a job on a dataset of a certain size.

② Response time

In online systems, service's response time is important, i.e., the time between a client sending a request and receiving a response.

* Latency and response time:-

↳ Latency and response time are different.
↳ The response time is what the client sees: besides the actual time to process the request (the service time), it includes network delays and queuing delays.
↳ Latency is the duration that a request is waiting to be handled - during which it is latent - awaiting service.

99th
95th

* Response time is not a single number but a distribution of values that you can measure, since the response time keep changing for same request made over and over again.

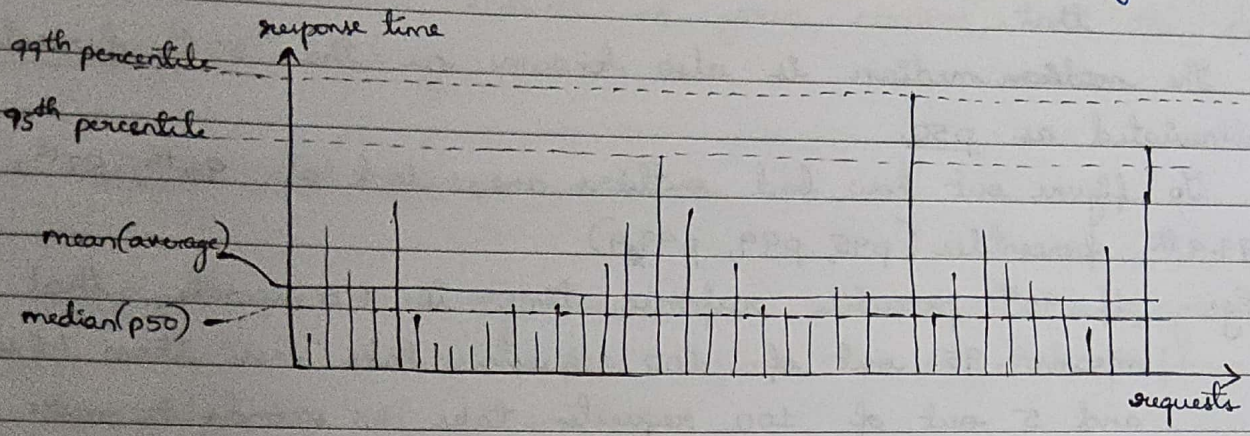


Fig., Illustrating mean and percentiles : response times for a sample of 100 requests to a service.

* In the above figure, there are occasional outliers that take much longer. The slow requests are more expensive, e.g., because they process more data.
* Even in a scenario where you'd think all requests should take same time, you get variation: random additional latency could be introduced ~~by~~ by:
  - → by a context switch to a background process
  - → the loss of a network packet and TCP retransmission
  - → a garbage collection pause
  - → a page fault forcing a read from disk
  - → mechanical vibrations in the server rack.
  - → many other causes.
* average response time - doesn't tell you how many users actually experienced that delay.
* percentiles - it is better to use percentiles. If you take your list of response times and sort it from fastest to slowest, & then the median is the halfway point.

Eg:- If your median response time is 200ms, that means half your requests return in less than 200ms, and half your requests take longer than that.

* The ~~median~~ median is also known as the 50th percentile, abbreviated as p50.
* To figure out how bad outliers are, look at 95th, 99th, 99.9th percentiles (p95, p99, p99.9)

Eg:- If 95th percentile response time is 1.5 seconds, that means 95 out of 100 requests take less than 1.5 seconds, and 5 out of 100 requests take 1.5 seconds or more.

* High percentiles of response times, also known as tail latencies, are important because they directly affect users' experience of the service.

Examples:-

Amazon describes response time requirements for internal services in terms of the 99.9th percentile, even though it only affects 1 in 1000 requests.

This is because the customers with the slowest requests are often those who have the most data on their accounts because they have made many purchases - that is, they're the most valuable customers.

It's important for Amazon to keep them happy by ensuring the website is fast for them:

Amazon has also observed that a 100ms increase in response time reduces sales by 1% and others report that a 1-second slowdown reduces the customer satisfaction metric by 16%.

On the other hand, optimizing the 99.99th percentile was deemed too expensive and to not yield enough benefit for Amazon's purposes. Reducing response times at very high percentiles is difficult because they are easily affected by random events outside of your control, and the benefits are diminishing.

SLA's and SLO's :-

* Service level objectives (SLOs) and Service level agreements (SLAs) are the contracts that define the expected performance and availability of a service.
* An SLA may state that:
  the service is considered to be up
  1) if it has a median response time < 200ms.
  2) 99th percentile under 1s
  3) the service may be required to be up at least 99.9% of the time.
* These metrics set expectations for clients of the service and allow customers to demand a refund if the SLA is not met.

Queueing delays - head-of-line blocking :-

* Queueing delays often account for a large part of the response time at high percentiles.
* As a server can only process a small number of things in parallel (limited, for example, by its number of CPU cores), it only takes a small number of slow requests to hold up the processing of subsequent requests - an effect sometimes known as head-of-line blocking.
* Even if those subsequent requests are fast to process on the server, the client will see a slow overall response time due to the time waiting for the prior request to complete.

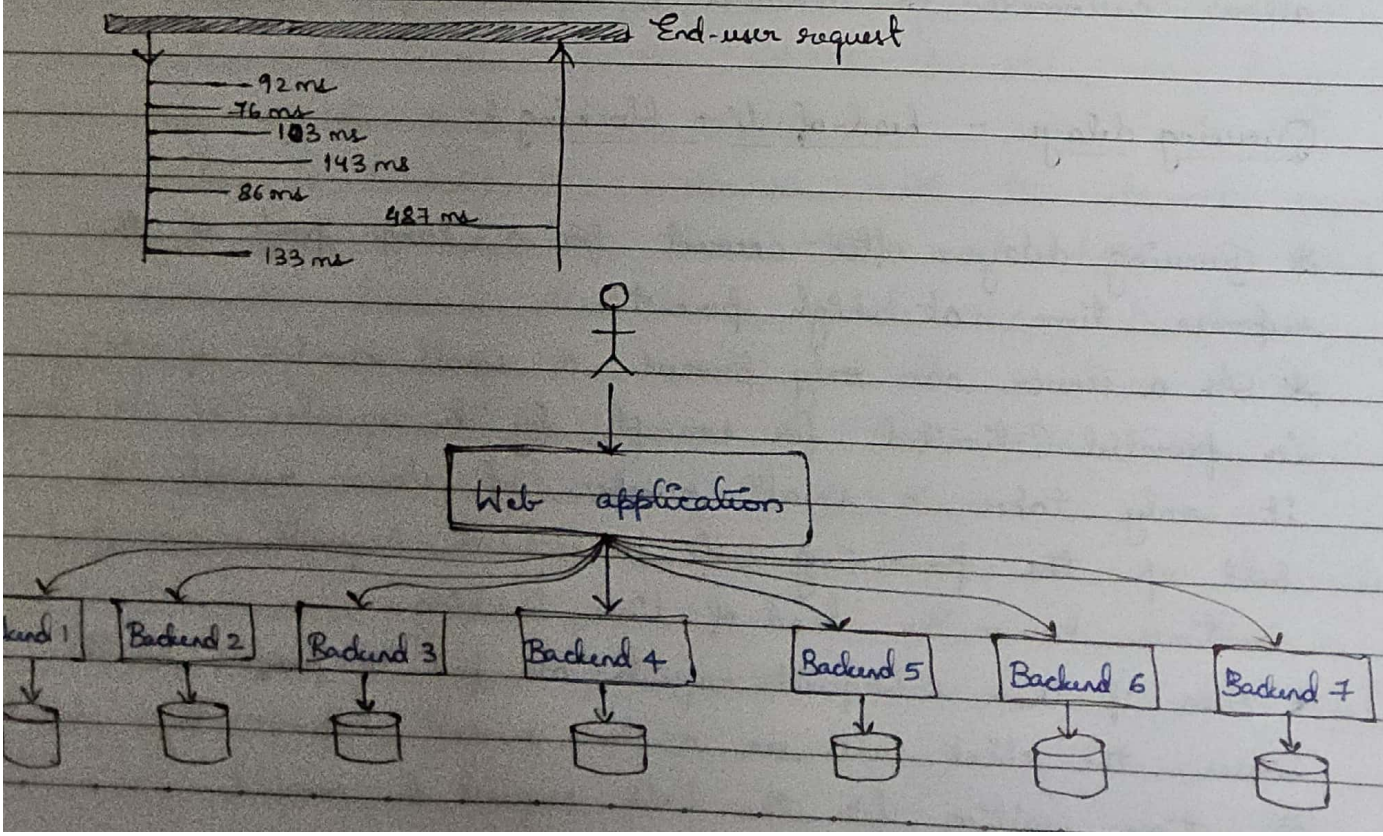
* Due to this effect, it is important to measure response times on the client side.

Important note :-

* Client has to send requests independently of response time during scalability test.
* If client waits on previous requests to complete before sending another requests, this behavior has the effect of artificially keeping the queues shorter in the test than they would be in ~~not~~ reality, which skews the measurements.

* Percentiles in Practice :-

High percentiles are important in backend services that are called multiple times as part of serving a single end-user request.



Even if you make parallel calls, it takes just one slow call to make the entire end-user request slow, as shown in above figure.

→ Tail latency amplification :-

Even if only a small percentage of backend calls are slow, the chance of getting a slow call increases if an end-user request requires multiple backend calls, and so a higher proportion of end-user requests end up being slow (an effect known as tail latency amplification.

→ Implementation :-

To add response time percentiles to the monitoring dashboards for your services, you need to efficiently calculate them on an ongoing basis.

For example, you may want to keep a rolling window of response times of requests in the last 10 minutes and every minute, calculate the median and various percentiles over the values in that window and plot those metrics on a graph.

The naive implementation is to keep a list of response times for all requests within the time window and to sort that list every minute. There are better efficient algorithms such as forward decay, t-digest, or HdrHistogram.

Averaging percentiles is mathematically meaningless. The right way of aggregating response time data is to add the histograms.

* Approaches for Coping with Load :-

scaling up - vertical scaling - moving to a more powerful machine.

scaling out - horizontal scaling - distributing the load across multiple smaller machin (also called shared-nothing architecture)

Choose scaling up until scaling cost or high availability requirements forced you to make it distributed.

The architecture of systems that operate at large scale is usually highly specific to the application - there is no one-size-fits-all scalable architecture (informally known as magic scaling sauce).

The problem may be
- ↳ the volume of reads
- ↳ the volume of writes
- ↳ the volume of data to store
- ↳ the complexity of the data
- ↳ the response time requirements
- ↳ the access patterns
- ↳ (or) (usually) some mixture of all of these + many more issue

Example :-

100000 requests/sec.	3 requests/minute
1 request size = 1 kB	1 request size = 2 GB.
$\Rightarrow \frac{100000 \times 10^3 \text{ Bytes}}{\text{sec}} = 10^8/\text{sec}$	$\frac{2 \times 10^9 \times 3 \text{ Bytes}}{\text{min}}$
$= 60 \times 10^8/\text{min}$	$= 6 \times 10^9/\text{min}$
$= 6 \times 10^9/\text{min}$	

In the above example, the two systems design will be different although the throughput is same. The difference is because the handling of requests are different in both systems (100000 req/sec of 1 kB size) and (3 req/min of 2 GB size).

An architecture that scales well for a particular application is built around
- which operations will be common } load parameters
- which will be rare

In an early-stage startup or an unproven product it's usually more important to be able to iterate quickly on product features than it is to scale to some hypothetical future load.

Even though scalable architectures are specific to a particular application, they are usually built from general-purpose building blocks, arranged in familiar patterns.

MAINTAINABILITY :-

* Majority of the cost of software is in its ongoing maintenance like
  - fixing bugs.
  - keeping its systems operational
  - investigating failures
  - adapting it to new platforms
  - modifying it for new use cases
  - repaying technical debt
  - adding new features.

* Three design principles :-

→ Operability:
Make it easy for operations teams to keep the system running smoothly

→ Simplicity:
Make it easy for new engineers to understand the system, by removing as much complexity as possible from the system. (This is not same as simplicity of user interface)

→ Evolvability: also known as extensibility, modifiability, or plasticity.
Make it easy for engineers to make changes to the system in the future, adapting it for unanticipated use cases as requirements change.

* Operability :- Making Life Easy for Operations :-

good operations can often work around the limitations of bad (or incomplete) software, but good software cannot run reliably with bad operations.

Operations team is responsible for the following:
① Monitoring the health of the system and quickly restoring service if it goes into a bad state.
② Tracking down the cause of problems, such as system failures or degraded performance.
③ Keeping software and platforms up to date, including security patches.
④ Keeping tabs on how different systems affect each other, so

that a problematic change can be avoided before it causes damage.

(5) Anticipating future problems and solving them before they occur (e.g., capacity planning).

(6) Establishing good practices and tools for deployment, configuration management and more.

(7) Performing complex maintenance tasks, such as moving an application from one platform to another.

(8) Maintaining the security of the system as configuration changes are made.

(9) Defining processes that make operations predictable and help keep the production environment stable.

(10) Preserving the organization's knowledge about the system, even as individual people come and go.

Good operability means making routine tasks easy, allowing the operations team to focus their efforts on high-value activities.

Data systems can do various things to make routine tasks easy, including:

(1) Providing visibility into the runtime behavior and internals of the system, with good monitoring.

(2) Providing good support for automation and integration with standard tools.

(3) Avoiding dependency on individual machines (allowing machines to be taken down for maintenance while the system as a whole continues running uninterrupted).

(4) Providing good documentation and an easy-to-understand operational model ("If I do X, Y will happen").

(5) Providing good default behavior, but also giving administrators the freedom to override defaults when needed.

(6) Self-healing but also giving administrators manual control over the state

(7) Exhibiting predictable behavior, minimizing surprises.

* Simplicity - Managing Complexity :-

A software project mired in complexity is sometimes described as a big ball of mud.

Symptoms :-
- → explosion of state space.
- → tight-coupling of modules.
- → tangled dependencies
- → inconsistent naming and terminology
- → hacks aimed at solving performance problems.
- → special-casing to work around issues

One of the best tools for removing complexity is abstraction.

Example :-

1) High-level programming language: are abstractions that hide machine code, CPU registers, and syscalls.

2) SQL is an abstraction that hides on-disk and in-memory data structures, concurrent requests from other clients and inconsistencies after crashes.

We will look for good abstractions that allow us to extract parts of a large system into well-defined, reusable components

* Evolvability : Making Change Easy :-

→ System requirements will not remain unchanged.
- you learn new facts
- previously unanticipated use cases emerge.
- business priorities change.
- users request new features.
- new platforms replace old platforms.

- legal or regulatory requirements change.
- growth of the system forces architectural changes etc.

Agile working patterns and tools help with adapting to change.

Agility on a data system level : evolvability.

Summary :-

We learnt some fundamental principles of data intensive applications. There are functional requirements and non functional requirements.

Functional requirements -
what it should do such as:
① allowing data to be stored, retrieved, searched and processed in various ways.

non functional requirements -
general properties like
① security, reliability, compliance, scalability, compatibility and maintainability.

Reliability

means making systems work correctly, even when faults occur. Faults can be in hardware (typically random and uncorrelated), software (bugs are typically systematic and hard to deal with) and humans (who inevitably make mistakes from time to time). Fault-tolerance techniques can hide certain types of faults from the end user.

Scalability:

means having strategies for keeping performance good, even when load increases. In order to discuss scalability, we first need ways of describing load and performance quantitatively.

We briefly looked at Twitter's home timelines as an example of describing load, and response time percentiles as a way of measuring performance.

In a scalable system, you can add processing capacity in order to remain reliable under high load.

Maintainability:

has many facets, but in essence it's about making life better for the engineering and operations teams who need to work with the system.

Good abstractions can help reduce complexity and make the system easier to modify and adapt for new use cases.

Good operability means having good visibility into the system's health and having effective ways of managing it.

There is no easy fix for making applications reliable, scalable or maintainable.

However, there are certain patterns and techniques that keep reappearing in different kinds of applications.

There are certain data systems that work toward these goals.

There are certain patterns for systems that consist of several components working together.