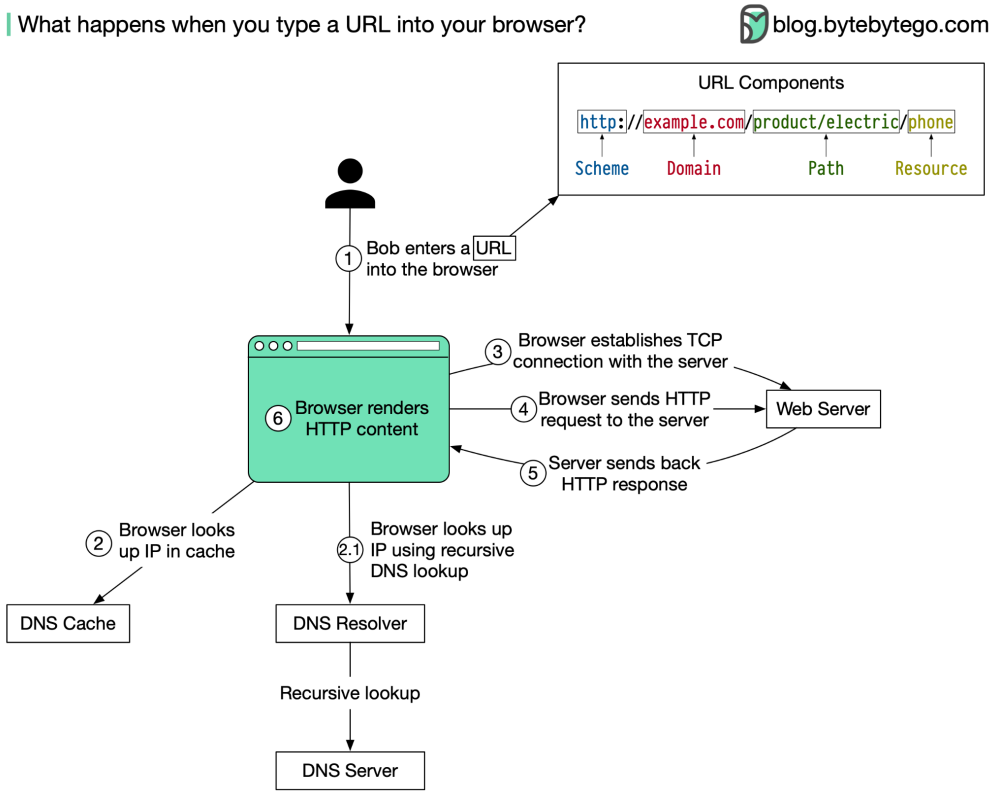
What happens when you type a URL into your browser?

The diagram below illustrates the steps.



1. Bob enters a URL into the browser and hits Enter. In this example, the URL is composed of 4 parts:

- scheme - ***https***://. This tells the browser to send a connection to the server using HTTPS.
- domain - ***example.com***. This is the domain name of the site.
- path - ***product/electric***. It is the path on the server to the requested resource: phone.
- resource - ***phone***. It is the name of the resource Bob wants to visit.

2. The browser looks up the IP address for the domain with a domain name system (DNS) lookup. To make the lookup process fast, data is cached at different layers: browser cache, OS cache, local network cache and ISP cache.

2.1 If the IP address cannot be found at any of the caches, the browser goes to DNS servers to do a recursive DNS lookup until the IP address is found (this will be covered in another post).

3. Now that we have the IP address of the server, the browser establishes a TCP connection with the server.

4. The browser sends a HTTP request to the server. The request looks like this:

```
GET /phone HTTP/1.1
Host: example.com
```

5. The server processes the request and sends back the response. For a successful response (the status code is 200). The HTML response might look like this:

```
HTTP/1.1 200 OK
Date: Sun, 30 Jan 2022 00:01:01 GMT
Server: Apache
Content-Type: text/html; charset=utf-8

<!DOCTYPE html>
<html lang="en">
Hello world
</html>
```

6. The browser renders the HTML content.

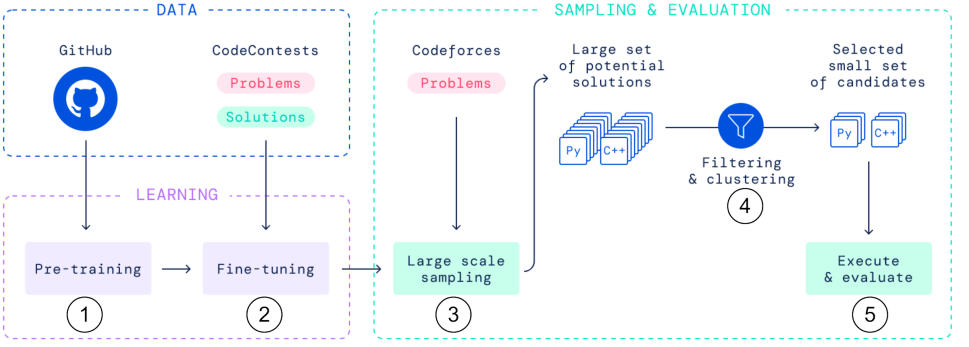
AI Coding engine

DeepMind says its new AI coding engine (AlphaCode) is as good as an average programmer.

The AI bot participated in the 10 Codeforces coding competitions and was ranked 54.3%. It means its score exceeded half of the human contestants. If we look at its score for the last 6 months, AlphaCode ranks at 28%.

The diagram below explains how the AI bot works:

How does AlphaCode Work?



Source: DeepMind Blog

1. Pre-train the transformer models on GitHub code.

2. Fine-tune the models on the relatively small competitive programming dataset.

3. At evaluation time, create a massive amount of solutions for each problem.

4. Filter, cluster and rerank the solutions to a small set of candidate programs (at most 10), and then submit for further assessments.

5. Run the candidate programs against the test cases, evaluate the performance, and choose the best one.

Do you think AI bot will be better at Leetcode or competitive programming than software engineers five years from now?

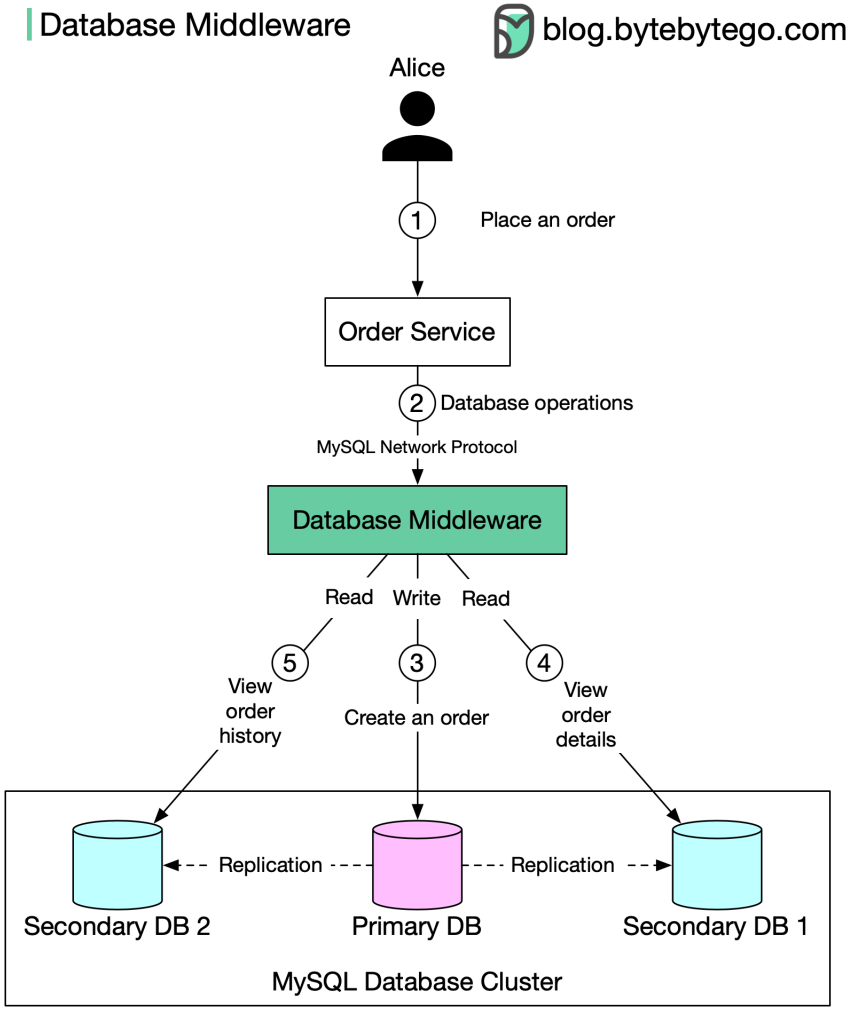
Read replica pattern

There are two common ways to implement the read replica pattern:

1. Embed the routing logic in the application code (explained in the last post).
2. Use database middleware.

We focus on option 2 here. The middleware provides transparent routing between the application and database servers. We can customize the routing logic based on difficult rules such as user, schema, statement, etc.

The diagram below illustrates the setup:



1. When Alice places an order on amazon, the request is sent to Order Service.

2. Order Service does not directly interact with the database. Instead, it sends database queries to the database middleware.

3. The database middleware routes writes to the primary database. Data is replicated to two replicas.

4. Alice views the order details (read). The request is sent through the middleware.

5. Alice views the recent order history (read). The request is sent through the middleware.

The database middleware acts as a proxy between the application and databases. It uses standard MySQL network protocol for communication.

Pros:
- Simplified application code. The application doesn't need to be aware of the database topology and manage access to the database directly.

- Better compatibility. The middleware uses the MySQL network protocol. Any MySQL compatible client can connect to the middleware easily. This makes database migration easier.

Cons:
- Increased system complexity. A database middleware is a complex system. Since all database queries go through the middleware, it usually requires a high availability setup to avoid a single point of failure.

- Additional middleware layer means additional network latency. Therefore, this layer requires excellent performance.

Read replica pattern

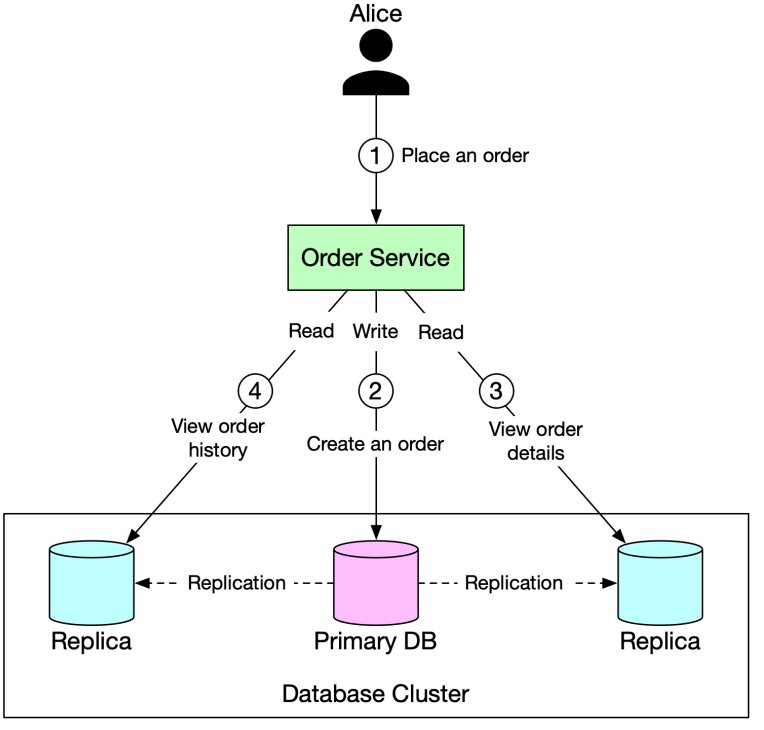
In this post, we talk about a simple yet commonly used database design pattern (setup): **Read replica pattern**.

In this setup, all data-modifying commands like insert, delete, or update are sent to the primary DB, and reads are sent to read replicas.

The diagram below illustrates the setup:

1. When Alice places an order on amazon.com, the request is sent to Order Service.
2. Order Service creates a record about the order in the primary DB (write). Data is replicated to two replicas.
3. Alice views the order details. Data is served from a replica (read).
4. Alice views the recent order history. Data is served from a replica (read).

Read Replica Pattern



There is one major problem in this setup: **replication lag**.

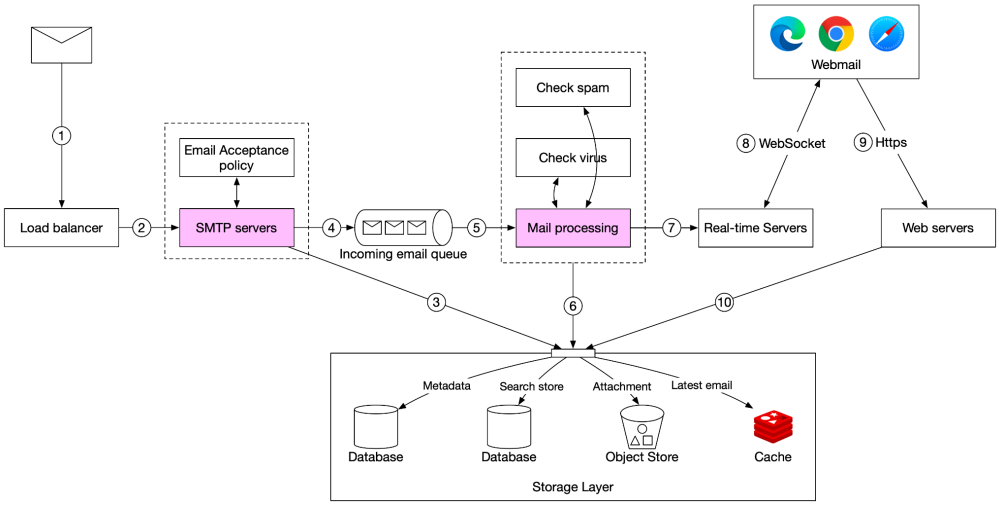
Under certain circumstances (network delay, server overload, etc.), data in replicas might be seconds or even minutes behind. In this case, if Alice immediately checks the order status (query is served by the replica) after the order is placed, she might not see the order at all. This leaves Alice confused. In this case, we need “read-after-write” consistency.

Possible solutions to mitigate this problem:

1. Latency sensitive reads are sent to the primary database.
2. Reads that immediately follow writes are routed to the primary database.
3. A relational DB generally provides a way to check if a replica is caught up with the primary. If data is up to date, query the replica. Otherwise, fail the read request or read from the primary.

Email receiving flow

The following diagram demonstrates the email receiving flow.



1. Incoming emails arrive at the SMTP load balancer.

2. The load balancer distributes traffic among SMTP servers. Email acceptance policy can be configured and applied at the SMTP-connection level. For example, invalid emails are bounced to avoid unnecessary email processing.

3. If the attachment of an email is too large to put into the queue, we can put it into the attachment store (s3).

4. Emails are put in the incoming email queue. The queue decouples mail processing workers from SMTP servers so they can be scaled independently. Moreover, the queue serves as a buffer in case the email volume surges.

5. Mail processing workers are responsible for a lot of tasks, including filtering out spam mails, stopping viruses, etc. The following steps assume an email passed the validation.

6. The email is stored in the mail storage, cache, and object data store.

7. If the receiver is currently online, the email is pushed to real-time servers.

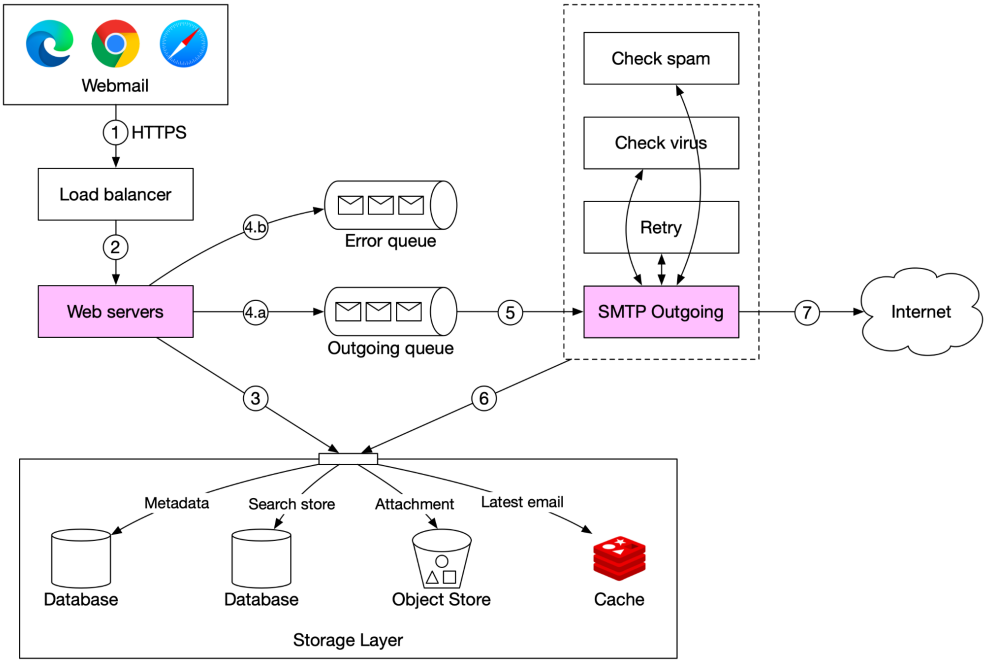
8. Real-time servers are WebSocket servers that allow clients to receive new emails in real-time.

9. For offline users, emails are stored in the storage layer. When a user comes back online, the webmail client connects to web servers via RESTful API.

10. Web servers pull new emails from the storage layer and return them to the client.

Email sending flow

In this post, we will take a closer look at the email sending flow.



1. A user writes an email on webmail and presses the “send” button. The request is sent to the load balancer.

2. The load balancer makes sure it doesn’t exceed the rate limit and routes traffic to web servers.

3. Web servers are responsible for:

- Basic email validation. Each incoming email is checked against pre-defined rules such as email size limit.

- Checking if the domain of the recipient’s email address is the same as the sender. If it is the same, email data is inserted to storage, cache, and object store directly. The recipient can fetch the email directly via the RESTful API. There is no need to go to step 4.

4. Message queues.

4.a. If basic email validation succeeds, the email data is passed to the outgoing queue.

4.b. If basic email validation fails, the email is put in the error queue.

5. SMTP outgoing workers pull events from the outgoing queue and make sure emails are spam and virus free.

6. The outgoing email is stored in the “Sent Folder” of the storage layer.

7. SMTP outgoing workers send the email to the recipient mail server.

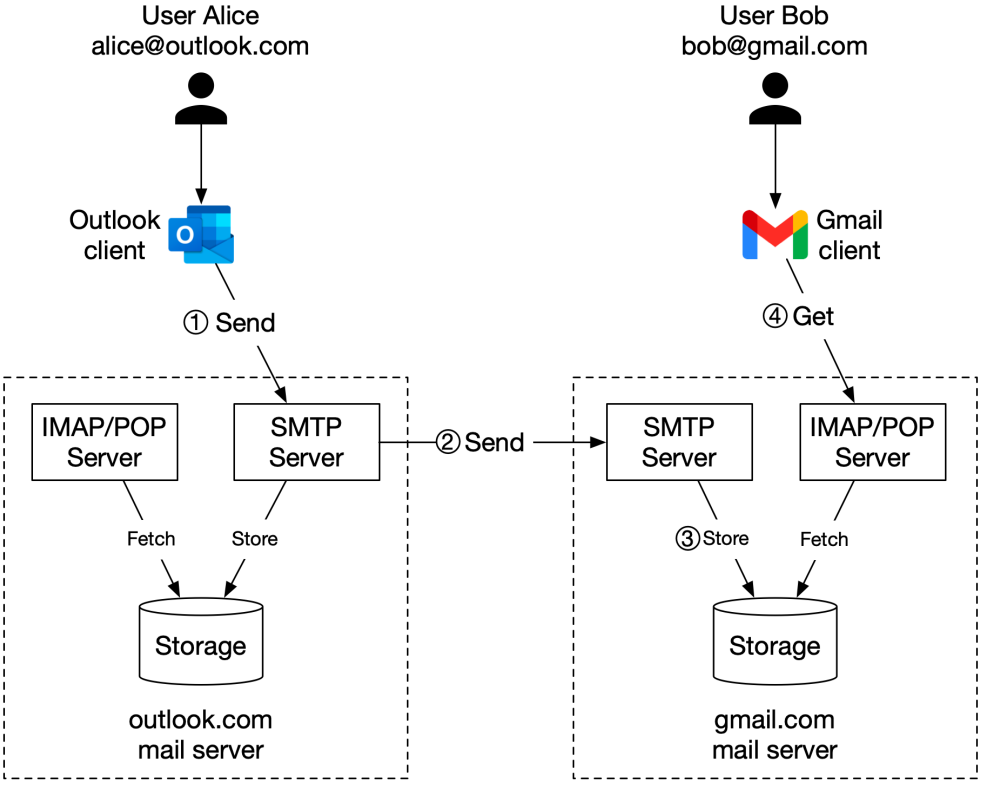
Each message in the outgoing queue contains all the metadata required to create an email. A distributed message queue is a critical component that allows asynchronous mail processing. By decoupling SMTP outgoing workers from the web servers, we can scale SMTP outgoing workers independently.

We monitor the size of the outgoing queue very closely. If there are many emails stuck in the queue, we need to analyze the cause of the issue. Here are some possibilities:
- The recipient’s mail server is unavailable. In this case, we need to retry sending the email at a later time. Exponential backoff might be a good retry strategy.

- Not enough consumers to send emails. In this case, we may need more consumers to reduce the processing time.

Interview Question: Design Gmail

One picture is worth more than a thousand words. In this post, we will take a look at what happens when Alice sends an email to Bob.



1. Alice logs in to her Outlook client, composes an email, and presses “send”. The email is sent to the Outlook mail server. The communication protocol between the Outlook client and mail server is SMTP.

2. Outlook mail server queries the DNS (not shown in the diagram) to find the address of the recipient’s SMTP server. In this case, it is Gmail’s SMTP server. Next, it transfers the email to the Gmail mail server. The communication protocol between the mail servers is SMTP.

3. The Gmail server stores the email and makes it available to Bob, the recipient.

4. Gmail client fetches new emails through the IMAP/POP server when Bob logs in to Gmail.

Please keep in mind this is a highly simplified design. Hope it sparks your interest and curiosity:) I'll explain each component in more depth in the future.

Map rendering

Google Maps Continued. Let's take a look at **Map Rendering** in this post.

Pre-Computed Tiles

One foundational concept in map rendering is tiling. Instead of rendering the entire map as one large custom image, the world is broken up into smaller tiles. The client only downloads the relevant tiles for the area the user is in and stitches them together like a mosaic for display. The tiles are pre-computed at different zoom levels. Google Maps uses 21 zoom levels.

For example, at zoom level 0, The entire map is represented by a single tile of size 256 * 256 pixels. Then at zoom level 1, the number of map tiles doubles in both north-south and east-west directions, while each tile stays at 256 * 256 pixels. So we have 4 tiles at zoom level 1, and the whole image of zoom level 1 is 512 * 512 pixels. With each increment, the entire set of tiles has 4x as many pixels as the previous level. The increased pixel count provides an increasing level of detail to the user.

This allows the client to render the map at the best granularities depending on the client's zoom level without consuming excessive bandwidth to download tiles with too much detail. This is especially important when we are loading the images from mobile clients.

Road Segments

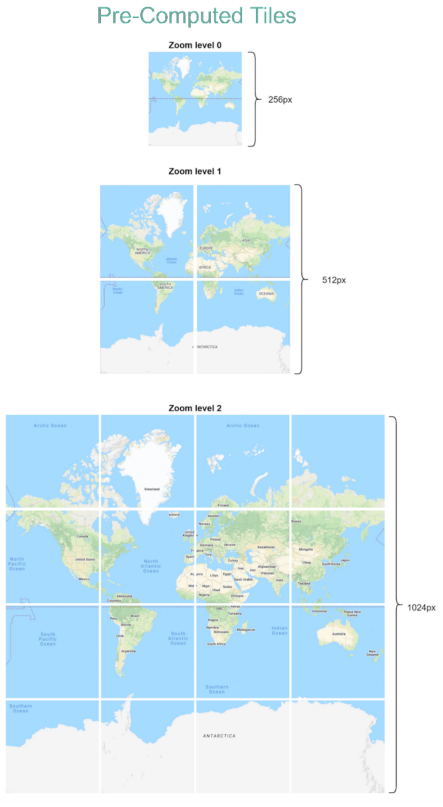
Now that we have transformed massive maps into tiles, we also need to define a data structure for the roads. We divide the world of roads into small blocks. We call these blocks road segments. Each road segment contains multiple roads, junctions, and other metadata.

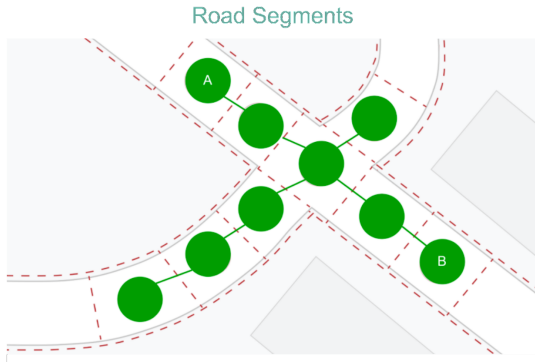
We group nearby segments into super segments. This process can be applied repeatedly to meet the level of coverage required.

We then transform the road segments into a data structure that the navigation algorithms can use. The typical approach is to convert the map into a ***graph***, where the nodes are road segments, and two nodes are connected if the corresponding road segments are reachable

neighbors. In this way, finding a path between two locations becomes a shortest-path problem, where we can leverage Dijkstra or A* algorithms.

Google Maps - Map Rendering



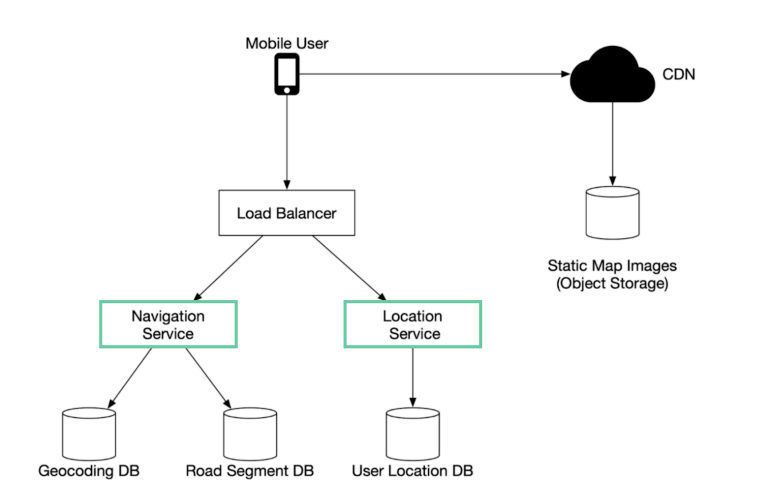


Interview Question: Design Google Maps

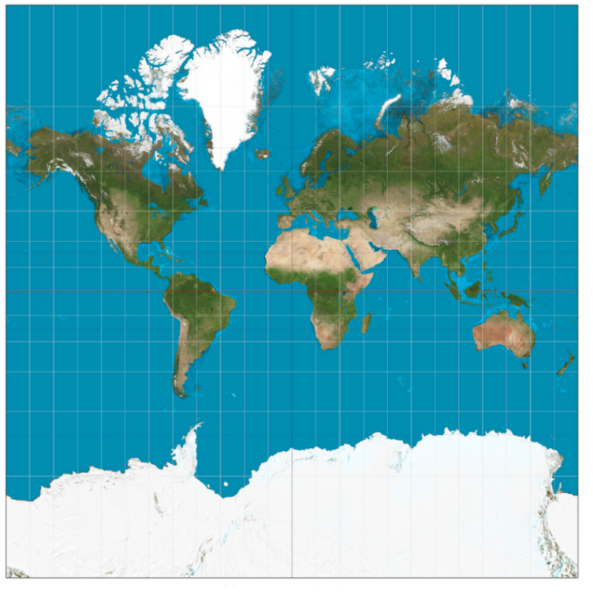
Google started project Google Maps in 2005. As of March 2021, Google Maps had one billion daily active users, 99% coverage of the world in 200 countries.

Although Google Maps is a very complex system, we can break it down into 3 high-level components. In this post, let's take a look at how to design a simplified Google Maps.

Google Maps



2D Map Projection



Location Service

The location service is responsible for recording a user's location update. The Google Map clients send location updates every few seconds. The user location data is used in many cases:

- detect new and recently closed roads
- improve the accuracy of the map over time
- used as an input for live traffic data.

Map Rendering

The world's map is projected into a huge 2D map image. It is broken down into small image blocks called "tiles" (see below). The tiles are static. They don't change very often. An efficient way to serve static tile files is with a CDN backed by cloud storage like S3. The users can load the necessary tiles to compose a map from nearby CDN.

What if a user is zooming and panning the map viewpoint on the client to explore their surroundings?

An efficient way is to pre-calculate the map blocks with different zoom levels and load the images when needed.

Navigation Service

This component is responsible for finding a reasonably fast route from point A to point B. It calls two services to help with the path calculation:

1. Geocoding Service: resolve the given address to a latitude/longitude pair

2. Route Planner Service: this service does three things in sequence:

    - Calculate the top-K shortest paths between A and B
    - Calculate the estimation of time for each path based on current traffic and historical data
    - Rank the paths by time predictions and user filtering. For example, the user doesn't want to avoid tolls.

Pull vs push models

There are two ways metrics data can be collected, pull or push. It is a routine debate as to which one is better and there is no clear answer. In this post, we will take a look at the pull model.

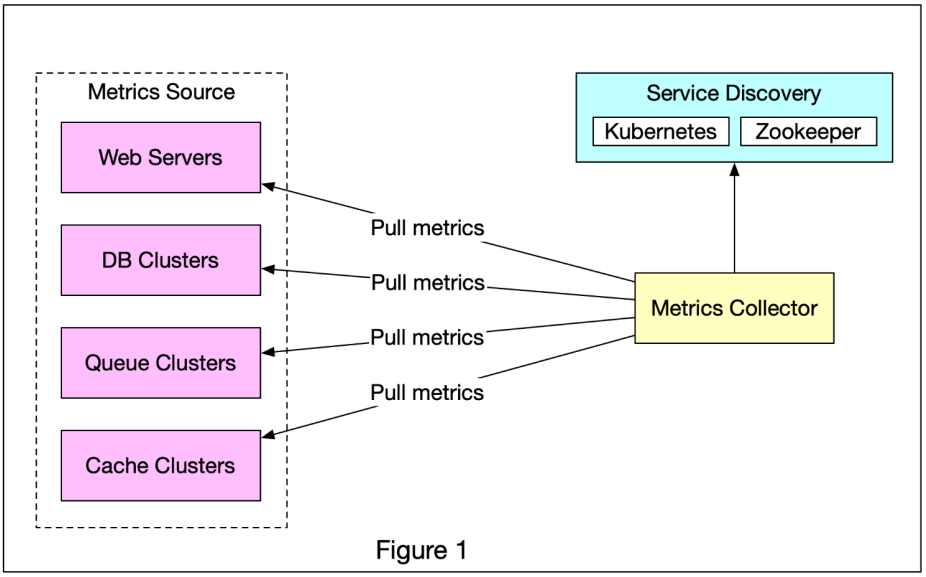


Figure 1

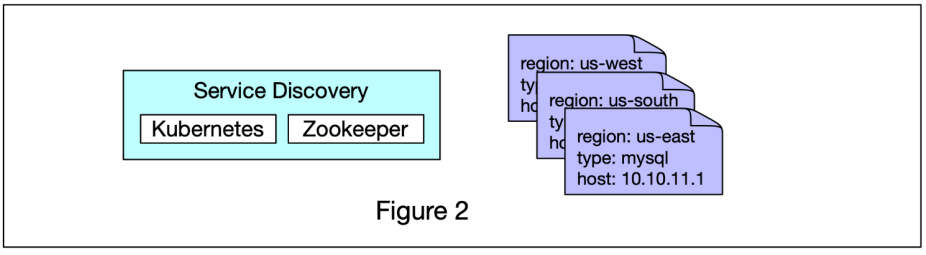


Figure 2

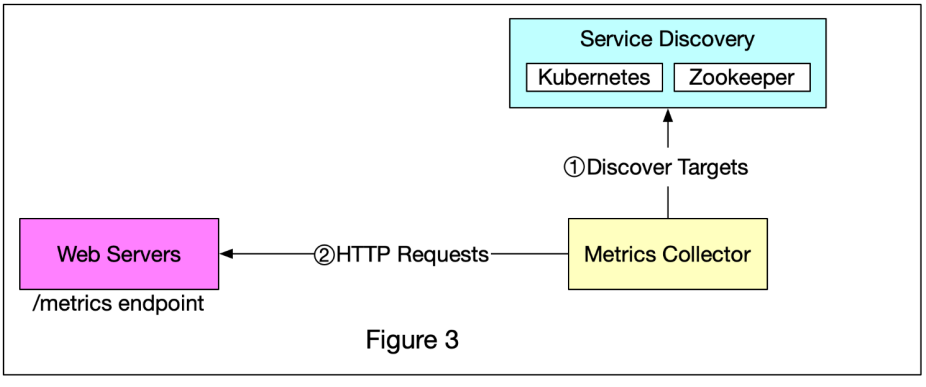


Figure 3

Figure 1 shows data collection with a pull model over HTTP. We have dedicated metric collectors which pull metrics values from the running applications periodically.

In this approach, the metrics collector needs to know the complete list of service endpoints to pull data from. One naive approach is to use a file to hold DNS/IP information for every service endpoint on the “metric collector” servers. While the idea is simple, this approach is hard to maintain in a large-scale environment where servers are added or removed frequently, and we want to ensure that metric collectors don’t miss out on collecting metrics from any new servers.

The good news is that we have a reliable, scalable, and maintainable solution available through Service Discovery, provided by Kubernetes, Zookeeper, etc., wherein services register their availability and the metrics collector can be notified by the Service Discovery component whenever the list of service endpoints changes. Service discovery contains configuration rules about when and where to collect metrics as shown in Figure 2.

Figure 3 explains the pull model in detail.

1 The metrics collector fetches configuration metadata of service endpoints from Service Discovery. Metadata include pulling interval, IP addresses, timeout and retries parameters, etc.

2 The metrics collector pulls metrics data via a pre-defined HTTP endpoint (for example, /metrics). To expose the endpoint, a client library usually needs to be added to the service. In Figure 3, the service is Web Servers.

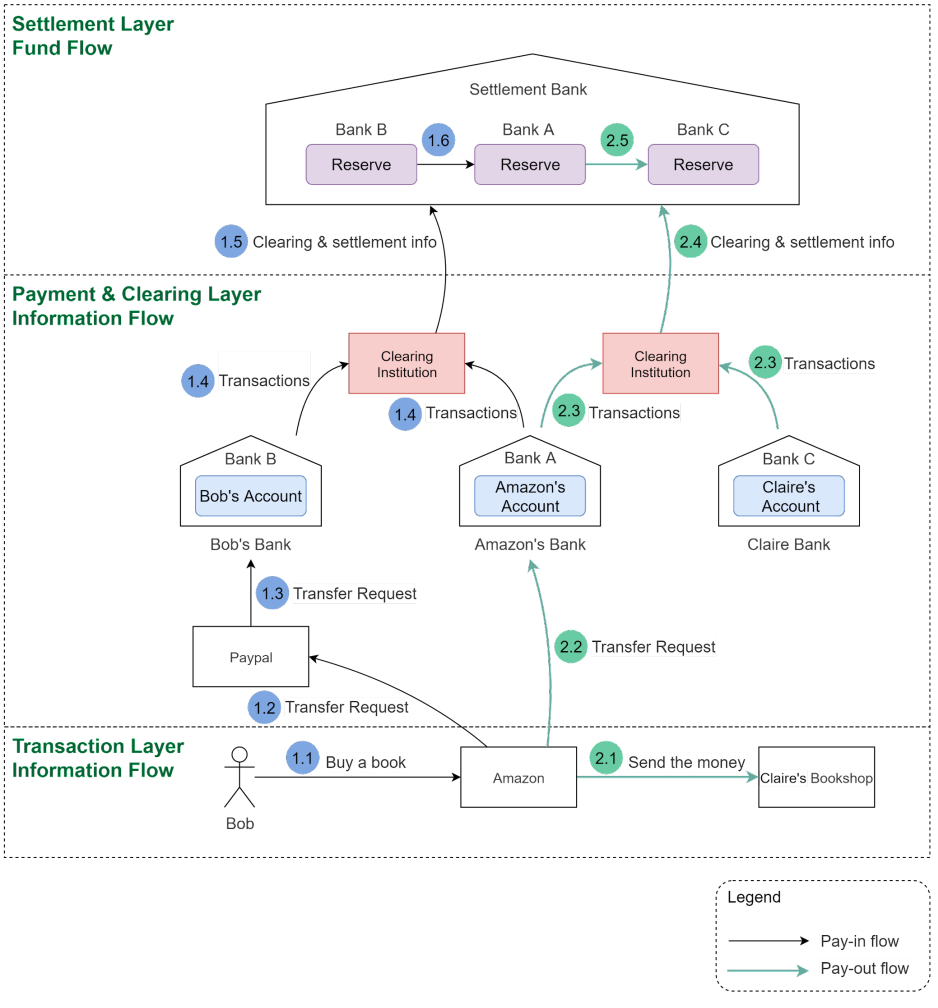
3 Optionally, the metrics collector registers a change event notification with Service Discovery to receive an update whenever the service endpoints change. Alternatively, the metrics collector can poll for endpoint changes periodically.

Money movement

One picture is worth more than a thousand words. This is what happens when you buy a product using Paypal/bank card under the hood.

To understand this, we need to digest two concepts: **clearing** & **settlement**. Clearing is a process that calculates who should pay whom with how much money; while settlement is a process where real money moves between reserves in the settlement bank.

Money Movement



Let’s say Bob wants to buy an SDI book from Claire’s shop on Amazon.

- Pay-in flow (Bob pays Amazon money):
1.1 Bob buys a book on Amazon using Paypal.
1.2 Amazon issues a money transfer request to Paypal.
1.3 Since the payment token of Bob’s debit card is stored in Paypal, Paypal can transfer money, on Bob’s behalf, to Amazon’s bank account in Bank A.
1.4 Both Bank A and Bank B send transaction statements to the clearing institution. It reduces the transactions that need to be settled. Let’s assume Bank A owns Bank B $100 and Bank B owns bank A $500 at the end of the day. When they settle, the net position is that Bank B pays Bank A $400.
1.5 & 1.6 The clearing institution sends clearing and settlement information to the settlement bank. Both Bank A and Bank B have pre-deposited funds in the settlement bank as money reserves, so actual money movement happens between two reserve accounts in the settlement bank.

- Pay-out flow (Amazon pays the money to the seller: Claire):
2.1 Amazon informs the seller (Claire) that she will get paid soon.
2.2 Amazon issues a money transfer request from its own bank (Bank A) to the seller bank (bank C). Here both banks record the transactions, but no real money is moved.
2.3 Both Bank A and Bank C send transaction statements to the clearing institution.
2.4 & 2.5 The clearing institution sends clearing and settlement information to the settlement bank. Money is transferred from Bank A’s reserve to Bank C’s reserve.

Notice that we have three layers:
- Transaction layer: where the online purchases happen
- Payment and clearing layer: where the payment instructions and transaction netting happen
- Settlement layer: where the actual money movement happen

The first two layers are called information flow, and the settlement layer is called fund flow.

You can see the **information flow and fund flow are separated**. In the info flow, the money seems to be deducted from one bank account and added to another bank account, but the actual money movement happens in the settlement bank at the end of the day.

Because of the asynchronous nature of the info flow and the fund flow, reconciliation is very important for data consistency in the systems along with the flow.

It makes things even more interesting when Bob wants to buy a book in the Indian market, where Bob pays USD but the seller can only receive INR.

Reconciliation

My previous post about painful payment reconciliation problems sparked lots of interesting discussions. One of the readers shared more problems we may face when working with intermediary payment processors in the trenches and a potential solution:

1. Foreign Currency Problem: When you operate a store globally, you will come across this problem quite frequently. To go back to the example from Paypal - if the transaction happens in a currency different from the standard currency of Paypal, this will create another layer, where the transaction is first received in that currency and exchanged to whatever currency your Paypal is using. There needs to be a reliable way to reconcile that currency exchange transaction. It certainly does not help that every payment provider handles this differently.

2. Payment providers are only that - intermediaries. Each purchase does not trigger two events for a company, but actually at least 4. The purchase via Paypal (where both the time and the currency dimension can come into play) trigger the debit/credit pair for the transaction and then, usually a few days later, another pair when the money is transferred from Paypal to a bank account (where there might be yet another FX discrepancy to reconcile if, for example, the initial purchase was in JPY, Paypal is set up in USD and your bank account is in EUR). There needs to be a way to reconcile all of these.

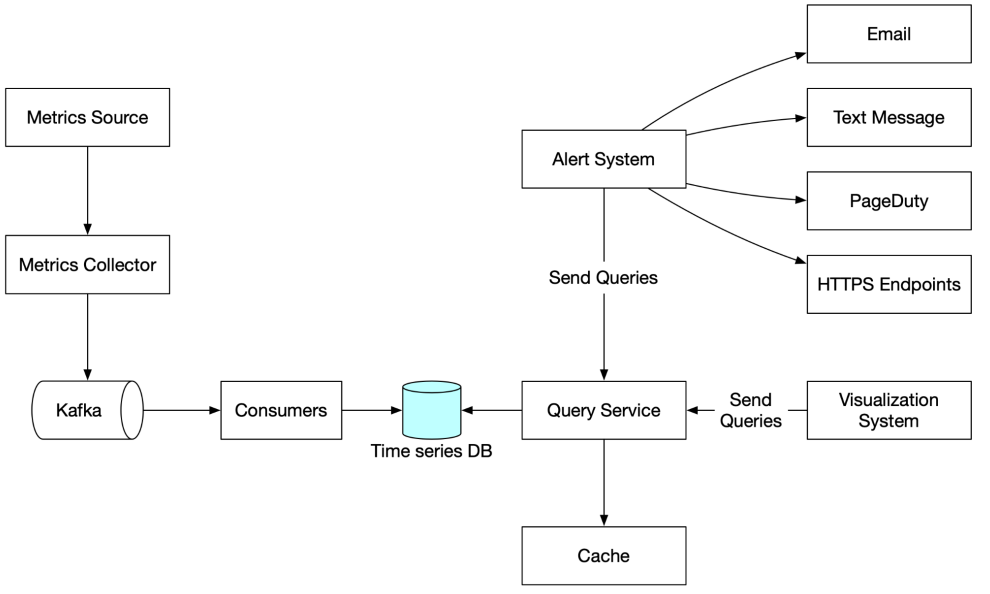
3. Some problems also pop up on the buyer side that is very platform-specific. One example is shadow transaction from Paypal: if you buy two items on Paypal with 1 week of time between the two transactions, Paypal will first debit money from your bank account for transaction A. If at the time of transaction B, transaction A has not gone through completely or is canceled, there might be a world where Paypal will use the money from transaction A to partially pay for transaction B, which leads to only a partial amount of transaction B being withdrawn from the bank account.

In practice, this usually looks something like this:
1) Your shop assigns an order number to the purchase

2) The order number is carried over to the payment provider
3) The payment provider creates another internal ID, which is carried over across transactions within the system
4) The payment ID is used when you get the payout on your bank account (or the payment provider bundles individual payments, which can be reconciled within the payment provider system)
5) Ideally, your payment provider and your shop have an integration/API with the tool you use to (hopefully automatically) create invoices. This usually carries over the order id from the shop (closing the loop) and sometimes even the payment id to match it with the invoice id, which you then can use to reconcile it with your accounts receivable/payable. :)

Credit: A knowledgeable reader who prefers to stay private. Thank you!

Continued: how to choose the right database for metrics collecting service?



There are many storage systems available that are optimized for time-series data. The optimization lets us use far fewer servers to handle the same volume of data. Many of these databases also have custom query interfaces specially designed for the analysis of time-series data that are much easier to use than SQL. Some even provide features to manage data retention and data aggregation. Here are a few examples of time-series databases.

OpenTSDB is a distributed time-series database, but since it is based on Hadoop and HBase, running a Hadoop/HBase cluster adds complexity. Twitter uses MetricsDB, and Amazon offers Timestream as a time-series database. According to DB-engines, the two most popular time-series databases are InfluxDB and Prometheus, which are designed to store large volumes of time-series data and quickly perform real-time analysis on that data. Both of them primarily rely on an in-memory cache and on-disk storage. And they both handle durability and performance quite well. According to the benchmark, an InfluxDB with 8 cores and 32GB RAM can handle over 250,000 writes per second.

Since a time-series database is a specialized database, you are not expected to understand the internals in an interview unless you explicitly mentioned it in your resume. For the purpose of an interview, it's important to understand the metrics data are time-series in nature and we can select time-series databases such as InfluxDB for storage to store them.

Another feature of a strong time-series database is efficient aggregation and analysis of a large amount of time-series data by labels, also known as tags in some databases. For example, InfluxDB builds indexes on labels to facilitate the fast lookup of time-series by labels. It provides clear best-practice guidelines on how to use labels, without overloading the database. The key is to make sure each label is of low cardinality (having a small set of possible values). This feature is critical for visualization, and it would take a lot of effort to build this with a general-purpose database.

Which database shall I use for the metrics collecting system?

This is one of the most important questions we need to address in an interview.

Data access pattern
As shown in the diagram, each label on the y-axis represents a time series (uniquely identified by the names and labels) while the x-axis represents time.

The write load is heavy. As you can see, there can be many time-series data points written at any moment. There are millions of operational metrics written per day, and many metrics are collected at high frequency, so the traffic is undoubtedly write-heavy.

At the same time, the read load is spiky. Both visualization and alert services send queries to the database and depending on the access patterns of the graphs and alerts, the read volume could be bursty.

Choose the right database
The data storage system is the heart of the design. It's not recommended to build your own storage system or use a general-purpose storage system (MySQL) for this job.

A general-purpose database, in theory, could support time-series data, but it would require expert-level tuning to make it work at our scale. Specifically, a relational database is not optimized for operations you would commonly perform against time-series data. For example, computing the moving average in a rolling time window requires complicated SQL that is difficult to read (there is an example of this in the deep dive section). Besides, to support tagging/labeling data, we need to add an index for each tag. Moreover, a general-purpose relational database does not perform well under constant heavy write load. At our scale, we would need to expend significant effort in tuning the database, and even then, it might not perform well.

How about NoSQL? In theory, a few NoSQL databases on the market could handle time-series data effectively. For example, Cassandra and Bigtable can both be used for time series data. However, this would require deep knowledge of the internal workings of each NoSQL to devise a scalable schema for effectively storing and querying time-series data. With industrial-scale time-series databases readily available, using a general purpose NoSQL database is not appealing.

There are many storage systems available that are optimized for time-series data. The optimization lets us use far fewer servers to handle the same volume of data. Many of these databases also have custom query interfaces specially designed for the analysis of time-series data that are much easier to use than SQL. Some even provide features to manage data retention and data aggregation. Here are a few examples of time-series databases.

OpenTSDB is a distributed time-series database, but since it is based on Hadoop and HBase, running a Hadoop/HBase cluster adds complexity. Twitter uses MetricsDB, and Amazon offers Timestream as a time-series database. According to DB-engines, the two most popular time-series databases are InfluxDB and Prometheus, which are designed to store large volumes of time-series data and quickly perform real-time analysis on that data. Both of them primarily rely on an in-memory cache and on-disk storage. And they both handle durability and performance quite well. According to the benchmark listed on InfluxDB website, a DB server with 8 cores and 32GB RAM can handle over 250,000 writes per second.

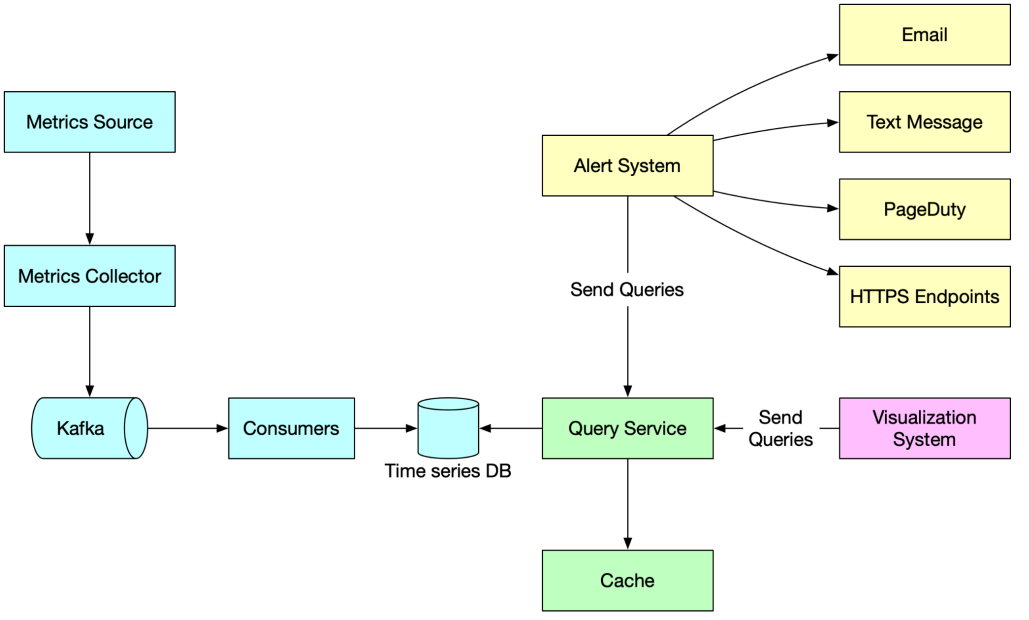
Since a time-series database is a specialized database, you are not expected to understand the internals in an interview unless you explicitly mentioned it in your resume. For the purpose of an interview, it’s important to understand the metrics data are time-series in nature and we can select time-series databases such as InfluxDB for storage to store them.

Another feature of a strong time-series database is efficient aggregation and analysis of a large amount of time-series data by labels, also known as tags in some databases. For example, InfluxDB builds indexes on labels to facilitate the fast lookup of time-series by

labels. It provides clear best-practice guidelines on how to use labels, without overloading the database. The key is to make sure each label is of low cardinality (having a small set of possible values). This feature is critical for visualization, and it would take a lot of effort to build this with a general-purpose database.

Metrics monitoring and altering system

A well-designed **metrics monitoring** and alerting system plays a key role in providing clear visibility into the health of the infrastructure to ensure high availability and reliability. The diagram below explains how it works at a high level.



Metrics source: This can be application servers, SQL databases, message queues, etc.

Metrics collector: It gathers metrics data and writes data into the time-series database.

Time-series database: This stores metrics data as time series. It usually provides a custom query interface for analyzing and summarizing a large amount of time-series data. It maintains indexes on labels to facilitate the fast lookup of time-series data by labels.

Kafka: Kafka is used as a highly reliable and scalable distributed messaging platform. It decouples the data collection and data processing services from each other.

Consumers: Consumers or streaming processing services such as Apache Storm, Flink and Spark, process and push data to the time-series database.

Query service: The query service makes it easy to query and retrieve data from the time-series database. This should be a very thin wrapper if we choose a good time-series database. It could also be entirely replaced by the time-series database's own query interface.

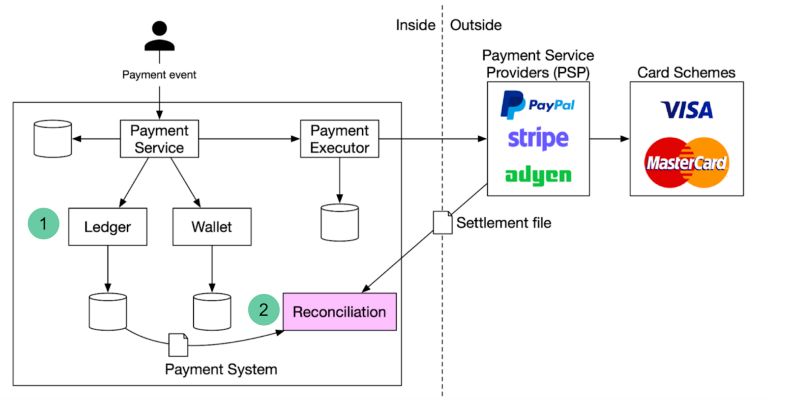
Alerting system: This sends alert notifications to various alerting destinations.

Visualization system: This shows metrics in the form of various graphs/charts.

Reconciliation

Reconciliation might be the most painful process in a payment system. It is the process of comparing records in different systems to make sure the amounts match each other.

Reconciliation in Payment System



Double-entry Bookkeeping in Ledger

Account	Debit	Credit
buyer	$200	
seller		$200

For example, if you pay $200 to buy a watch with Paypal:
- The eCommerce website should have a record about the purchase order of $200.
- There should be a transaction record of $200 in Paypal (marked with 2 in the diagram).
- The Ledger should record a debit of $200 dollars for the buyer, and a credit of $200 for the seller. This is called double-entry bookkeeping (see the table below).

Let's take a look at some pain points and how we can address them:

Problem 1: Data normalization. When comparing records in different systems, they come in different formats. For example, the timestamp can be “2022/01/01” in one system and “Jan 1, 2022” in another.
Possible solution: we can add a layer to transform different formats into the same format.

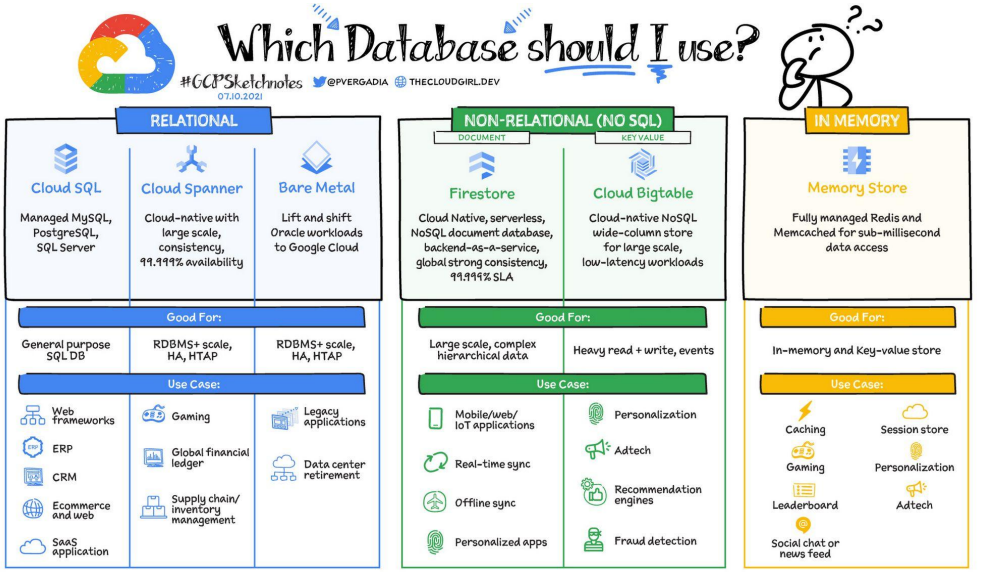
Problem 2: Massive data volume
Possible solution: we can use big data processing techniques to speed up data comparisons. If we need near real-time reconciliation, a streaming platform such as Flink is used; otherwise, end-of-day batch processing such as Hadoop is enough.

Problem 3: Cut-off time issue. For example, if we choose 00:00:00 as the daily cut-off time, one record is stamped with 23:59:55 in the internal system, but might be stamped 00:00:30 in the external system (Paypal), which is the next day. In this case, we couldn’t find this record in today’s Paypal records. It causes a discrepancy.
Possible solution: we need to categorize this break as a “temporary break” and run it later against the next day’s Paypal records. If we find a match in the next day’s Paypal records, the break is cleared, and no more action is needed.

You may argue that if we have exactly-once semantics in the system, there shouldn’t be any discrepancies. But the truth is, there are so many places that can go wrong. Having a reconciliation system is always necessary. It is like having a safety net to keep you sleeping well at night.

Which database shall I use? This is one of the most important questions we usually need to address in an interview.

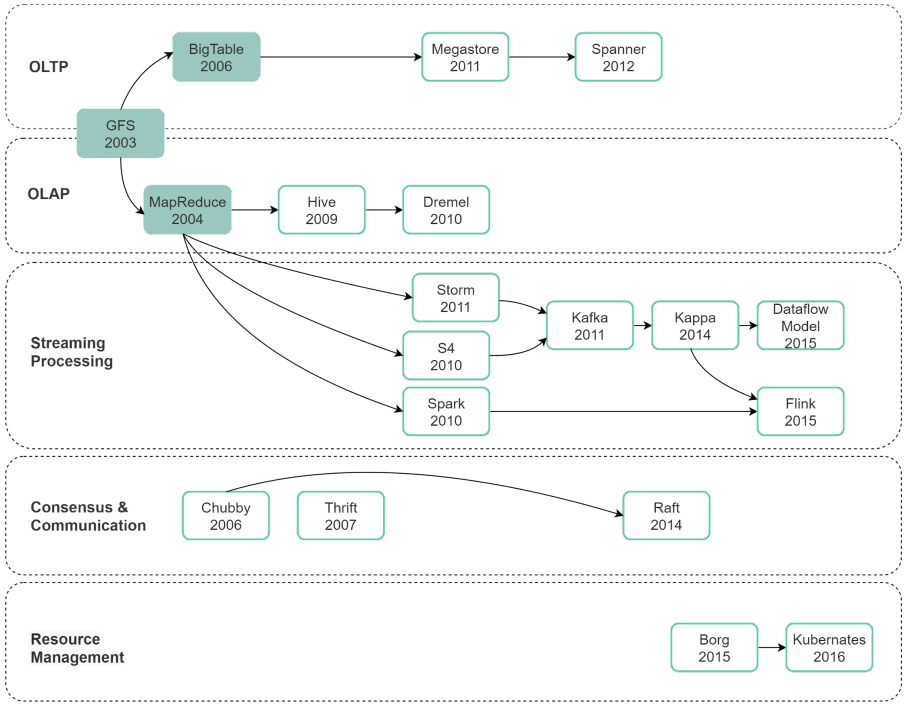
Choosing the right database is hard. Google Cloud recently posted a great article that summarized different database options available in Google Cloud and explained which use cases are best suited for each database option.



Big data papers

Below is a timeline of important big data papers and how the techniques evolved over time.

Big Data Theses Timeline & Relationship



The green highlighted boxes are the famous 3 Google papers, which established the foundation of the big data framework. At the high-level:

Big Data Techniques = *Massive data* + *Massive calculation*

Let's look at the **OLTP** evolution. BigTable provided a distributed storage system for structured data but dropped some characteristics of relational DB. Then Megastore brought back schema and simple transactions; Spanner brought back data consistency.

Now let's look at the **OLAP** evolution. MapReduce was not easy to program, so Hive solved this by introducing a SQL-like query

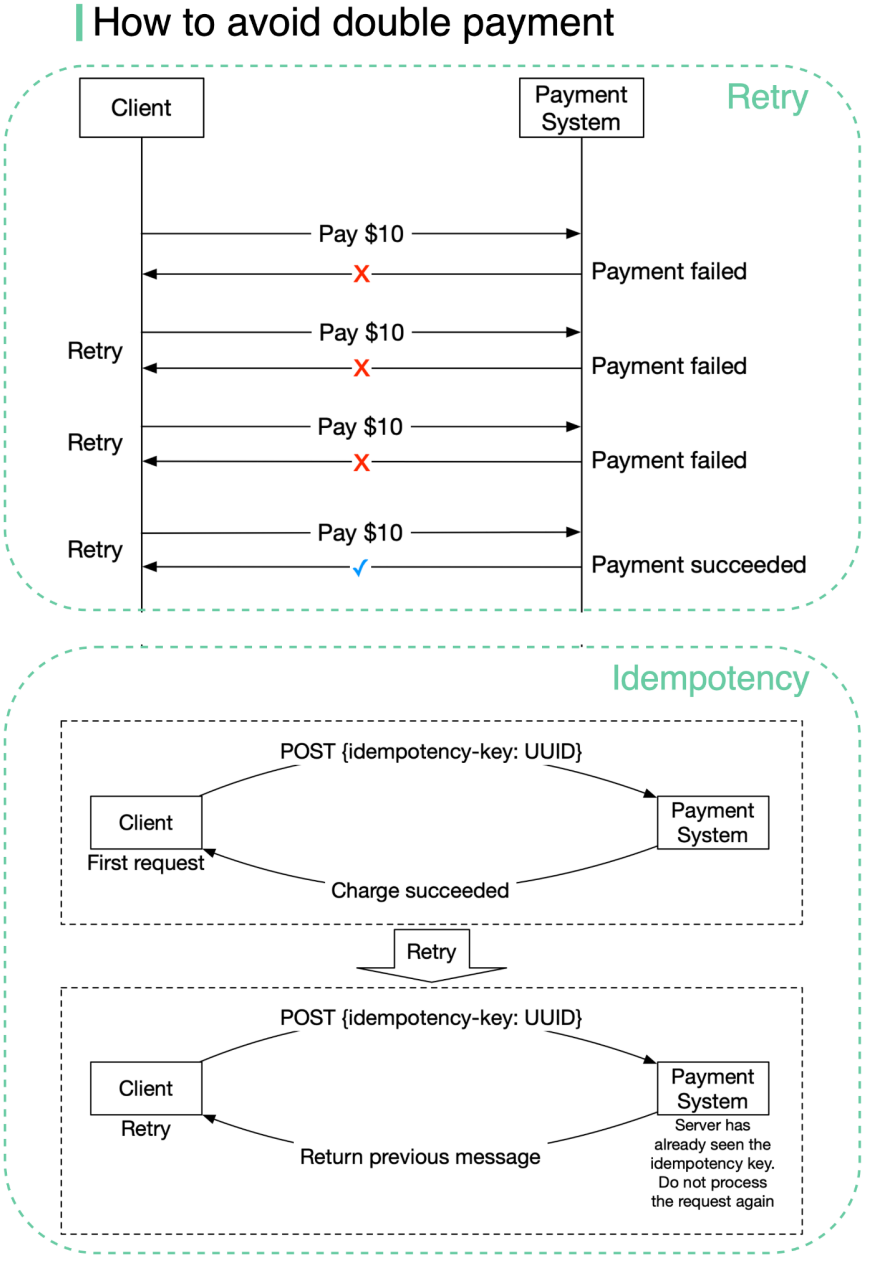
language. But Hive still used MapReduce under the hood, so it's not very responsive. In 2010, Dremel provided an interactive query engine.

Streaming processing was born to further solve the latency issue in OLAP. The famous ***lambda*** architecture was based on Storm and MapReduce, where streaming processing and batch processing have different processing flows. Then people started to build streaming processing with apache Kafka. ***Kappa*** architecture was proposed in 2014, where streaming and batching processings were merged into one flow. Google published The Dataflow Model in 2015, which was an abstraction standard for streaming processing, and Flink implemented this model.

To manage a big crowd of commodity server resources, we need resource management Kubernetes.

Avoid double charge

One of the most serious problems a payment system can have is to **double charge a customer**. When we design the payment system, it is important to guarantee that the payment system executes a payment order exactly-once.



At the first glance, exactly-once delivery seems very hard to tackle, but if we divide the problem into two parts, it is much easier to solve. Mathematically, an operation is executed exactly-once if:

1. It is executed at least once.

2. At the same time, it is executed at most once.

We now explain how to implement at least once using retry and at most once using idempotency check.

Retry

Occasionally, we need to retry a payment transaction due to network errors or timeout. Retry provides the at-least-once guarantee. For example, as shown in Figure 10, the client tries to make a $10 payment, but the payment keeps failing due to a poor network connection. Considering the network condition might get better, the client retries the request and this payment finally succeeds at the fourth attempt.

Idempotency

From an API standpoint, idempotency means clients can make the same call repeatedly and produce the same result.

For communication between clients (web and mobile applications) and servers, an idempotency key is usually a unique value that is generated by clients and expires after a certain period of time. A UUID is commonly used as an idempotency key and it is recommended by many tech companies such as Stripe and PayPal. To perform an idempotent payment request, an idempotency key is added to the HTTP header: *<idempotency-key: key_value>*.

Payment security

A few weeks ago, I posted the high-level design for the payment system. Today, I'll continue the discussion and focus on payment security.

The table below summarizes techniques that are commonly used in payment security. If you have any questions or I missed anything, please leave a comment.

Problem	Solution
Request/response eavesdropping	Use HTTPS
Data tampering	Enforce encryption and integrity monitoring
Man-in-the-middle attack	Use SSL and authentication certificates
Data loss	Database replication across multiple regions and take snapshot of data
Distributed denial-of-service attack (DDoS)	Rate limiting and firewall
Card theft	Tokenization. Instead of using real card numbers, tokens are stored and used for payment
PCI compliance	PCI DSS is an information security standard for organizations that handle branded credit cards
Fraud	Address verification, card verification value (CVV), user behavior analysis, etc.

System Design Interview Tip

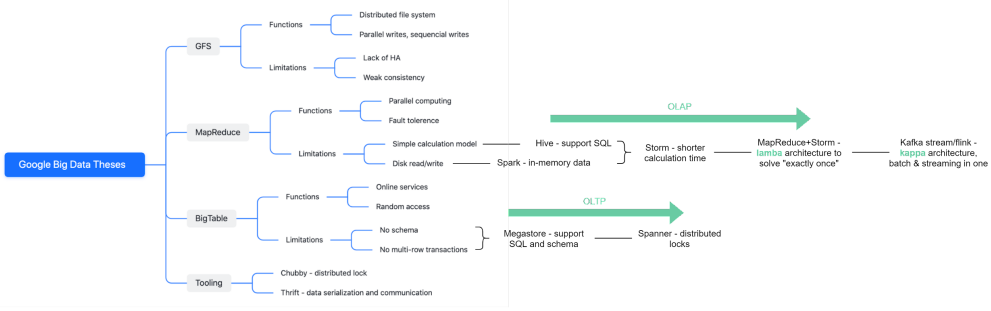
One pro tip for acing a system design interview is to read the engineering blog of the company you are interviewing with. You can get a good sense of what technology they use, why the technology was chosen over others, and learn what issues are important to engineers.

For example, here are 4 blog posts Twitter Engineering recommends:
1. The Infrastructure Behind Twitter: Scale
2. Discovery and Consumption of Analytics Data at Twitter
3. The what and why of product experimentation at Twitter
4. Twitter experimentation: technical overview

Big data evolvement

I hope everyone has a great time with friends and family during the holidays. If you are looking for some readings, classic engineering papers are a good start.

Big Data Evolvement



A lot of times when we are busy with work, we only focus on scattered information, telling us “**how**” and “what” to get our immediate needs to get things done.

However, reading the classics helps us know “**why**” behind the scenes, and teaches us how to solve problems, make better decisions, or even contribute to open source projects.

Let’s take **big data** as an example.

Big data area has progressed a lot over the past 20 years. It started from 3 Google papers (see the links in the comment), which tackled real engineering challenges at Google scale:

- GFS (2003) - big data storage
- MapReduce (2004) - calculation model
- BigTable (2006) - online services

The diagram below shows the functionalities and limitations of the 3 techniques, and how they evolve over time into two streams: OLTP and OLAP. Each evolved product was trying to solve the limitations of the

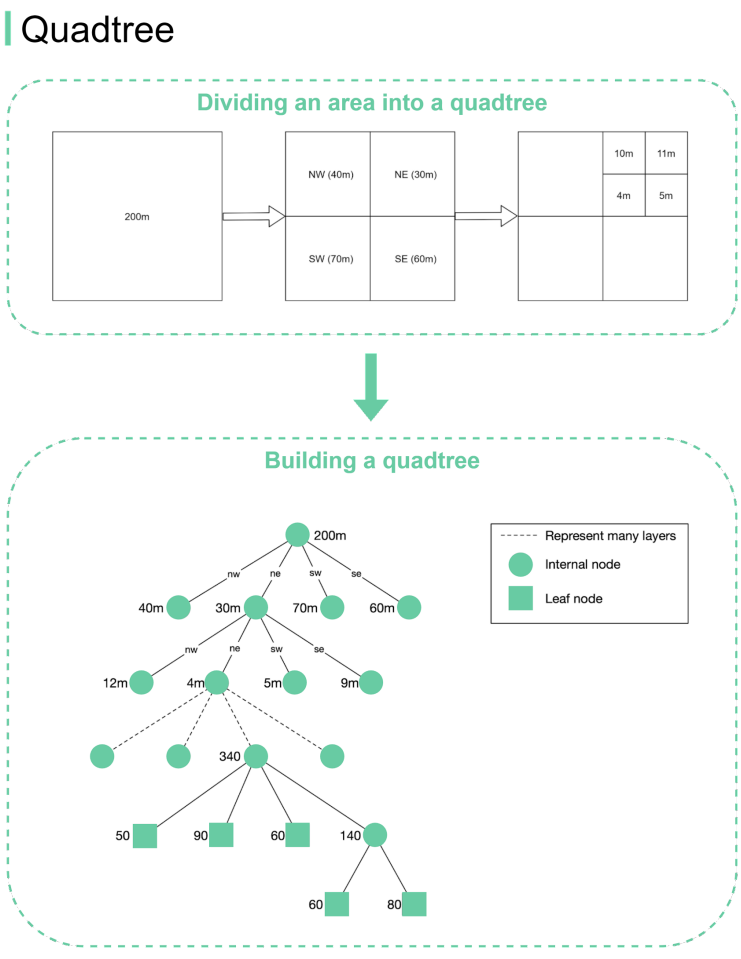
last generation. For example, “Hive - support SQL” means Hive was trying to solve the lack of SQL in MapReduce.

If you want to learn more, you can refer to the papers for details. What other classics would you recommend?

Quadtree

In this post, let's explore another data structure to find nearby restaurants on Yelp or Google Maps.

A quadtree is a data structure that is commonly used to partition a two-dimensional space by recursively subdividing it into four quadrants (grids) until the contents of the grids meet certain criteria (see the first diagram).



Quadtree is an **in-memory data structure** and it is not a database solution. It runs on each LBS (Location-Based Service, see last week's post) server, and the data structure is built at server start-up time.

The second diagram explains the quadtree building process in more detail. The root node represents the whole world map. The root node is **recursively** broken down into 4 quadrants until no nodes are left with more than 100 businesses.

How to get nearby businesses with quadtree?
- Build the quadtree in memory.

- After the quadtree is built, start searching from the root and traverse the tree, until we find the leaf node where the search origin is.

- If that leaf node has 100 businesses, return the node. Otherwise, add businesses from its neighbors until enough businesses are returned.

Update LBS server and rebuild quadtree
- It may take a few minutes to build a quadtree in memory with 200 million businesses at the server start-up time.

- While the quadtree is being built, the server cannot serve traffic.

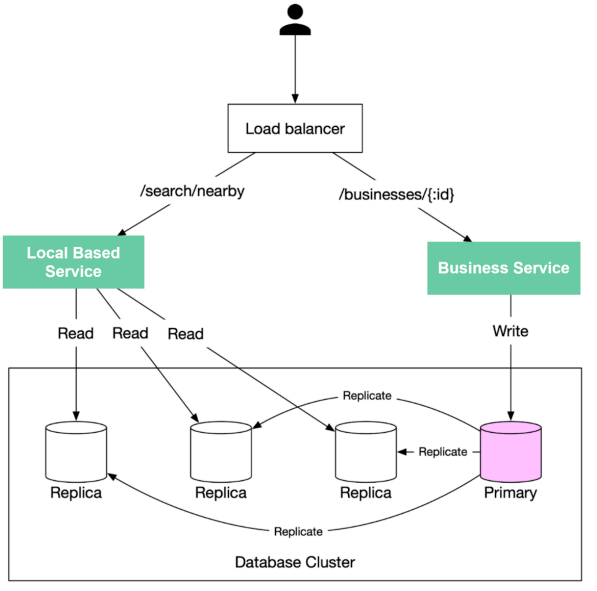
- Therefore, we should roll out a new release of the server incrementally to **a small subset** of servers at a time. This avoids taking a large swathe of the server cluster offline and causes service brownout.

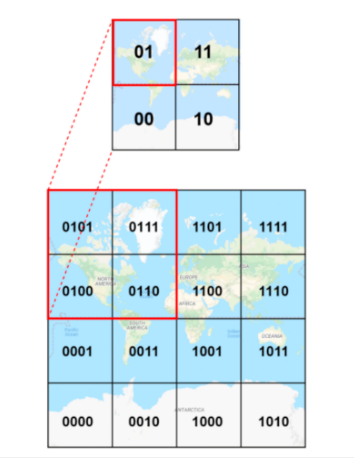
How do we find nearby restaurants on Yelp?

Here are some design details behind the scenes.

There are two key services (see the diagram below):

Proximity Service Design





- Business Service

- Add/delete/update restaurant information
- Customers view restaurant details

- Local-based Service (LBS)

- Given a radius and location, return a list of nearby restaurants

How are the restaurant locations stored in the database so that LBS can return nearby restaurants efficiently?

Store the latitude and longitude of restaurants in the database? The query will be very inefficient when you need to calculate the distance between you and every restaurant.

One way to speed up the search is using the **geohash algorithm**.

First, divide the planet into four quadrants along with the prime meridian and equator：

- Latitude range [-90, 0] is represented by 0

- Latitude range [0, 90] is represented by 1

- Longitude range [-180, 0] is represented by 0

- Longitude range [0, 180] is represented by 1

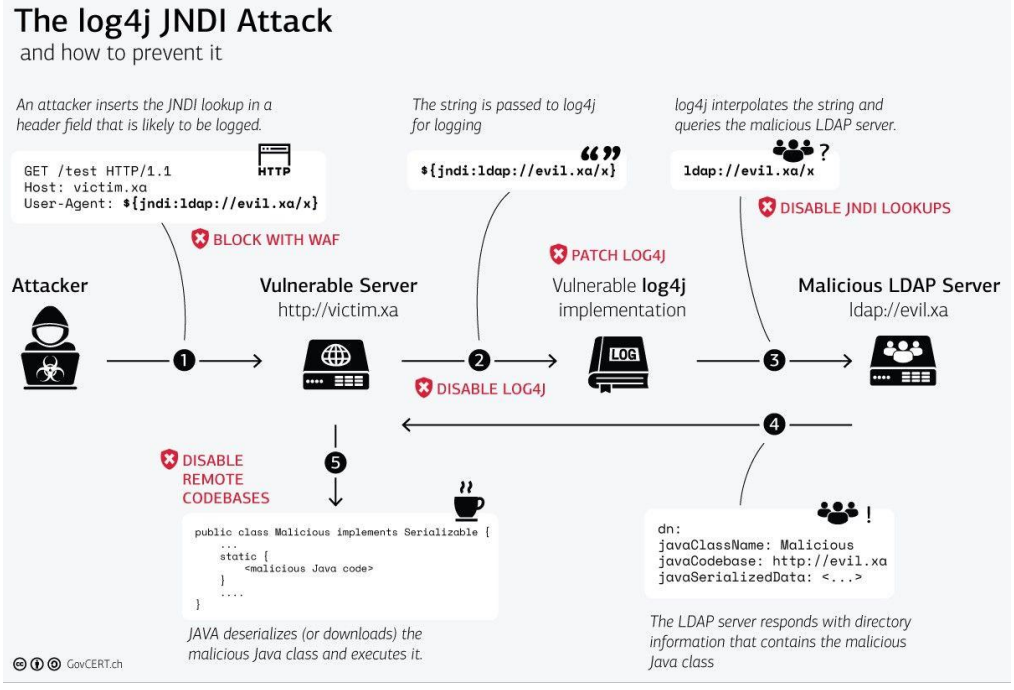
Second, divide each grid into four smaller grids. Each grid can be represented by alternating between longitude bit and latitude bit.

So when you want to search for the nearby restaurants in the red-highlighted block, you can write SQL like:

```
SELECT * FROM geohash_index WHERE geohash LIKE `01%`
```

Geohash has some limitations. There can be a lot of restaurants in one block (downtown New York), but none in another block (ocean). So there are other more complicated algorithms to optimize the process. Let me know if you are interested in the details.

One picture is worth more than a thousand words. Log4j from attack to prevention in one illustration.



Credit GovCERT

Link:
https://www.govcert.ch/blog/zero-day-exploit-targeting-popular-java-library-log4j/

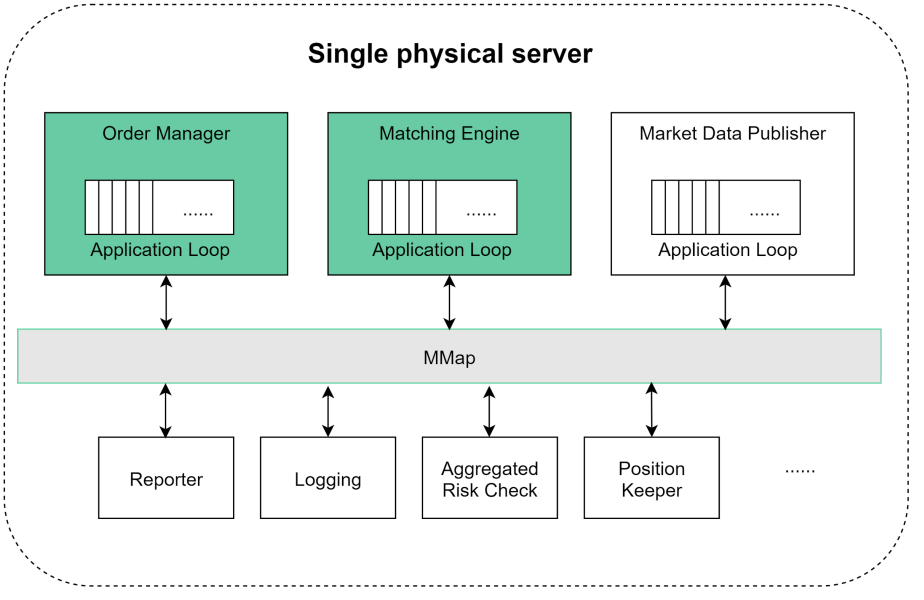
How does a modern stock exchange achieve microsecond latency?

The principal is:

Do less on the critical path !

- Fewer tasks on the critical path
- Less time on each task
- Fewer network hops
- Less disk usage

Low Latency Stock Exchange Design



For the stock exchange, the critical path is:
- **start**: an order comes into the order manager
- mandatory risk checks
- the order gets matched and the execution is sent back
- **end**: the execution comes out of the order manager

Other non-critical tasks should be removed from the critical path.

We put together a design as shown in the diagram:

- deploy all the components in a single giant server (no containers)

- use shared memory as an event bus to communicate among the components, no hard disk

- key components like Order Manager and Matching Engine are single-threaded on the critical path, and each pinned to a CPU so that there is **no context switch** and **no locks**

- the single-threaded application loop executes tasks one by one in sequence

- other components listen on the event bus and react accordingly

Match buy and sell orders

Stocks go up and down. Do you know what data structure is used to efficiently match buy and sell orders?

Buy 2700 shares: 2700 - 200 - 400 - 1100 - 100 - 900 = 0

Stock exchanges use the data structure called **order books**. An order book is an electronic list of buy and sell orders, organized by price levels. It has a buy book and a sell book, where each side of the book contains a bunch of price levels, and each price level contains a list of orders (first in first out).

The image is an example of price levels and the queued quantity in each price level.

So what happens when you place a market order to buy 2700 shares in the diagram?

- The buy order is matched with all the sell onrders at price 100.10, and the first order at price 100.11 (illustrated in light red).

- Now because of the big buy order which “eats up” the first price level on the sell book, the best ask price goes up from 100.10 to 100.11.

- So when the market is bullish, people tend to buy stocks, and the price goes up and up.

An efficient data structure for an order book must satisfy:

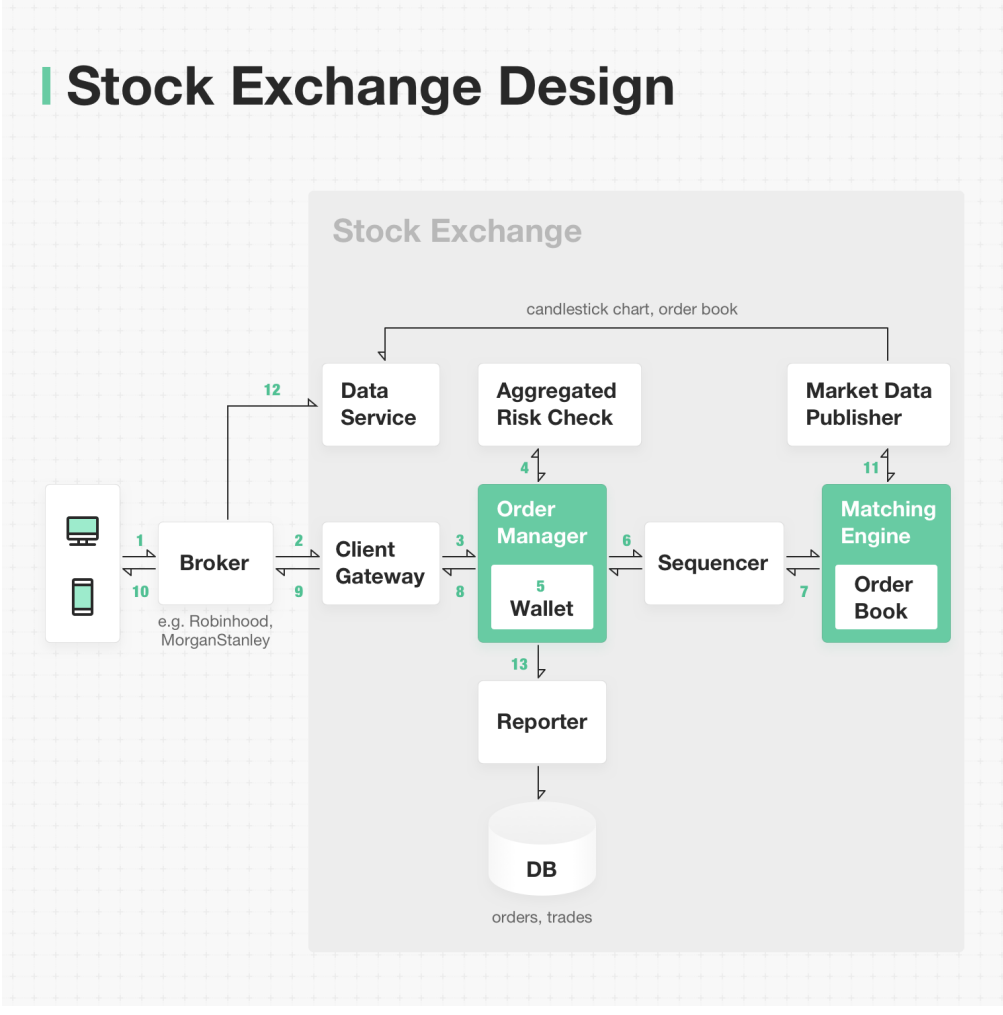
- Constant lookup time. Operations include: get volume at a price level or between price levels, query best bid/ask.

- Fast add/cancel/execute/update operations, preferably O(1) time complexity. Operations include: place a new order, cancel an order, and match an order.

Stock exchange design

The stock market has been volatile recently.

Coincidentally, we just finished a new chapter “Design a stock exchange”. I’ll use plain English to explain what happens when you place a stock buying order. The focus is on the exchange side.



Step 1: client places an order via the broker’s web or mobile app.

Step 2: broker sends the order to the exchange.

Step 3: the exchange client gateway performs operations such as validation, rate limiting, authentication, normalization, etc, and sends the order to the order manager.

Step 4: the order manager performs risk checks based on rules set by the risk manager.

Step 5: once risk checks pass, the order manager checks if there is enough balance in the wallet.

Step 6-7: the order is sent to the matching engine. The matching engine sends back the execution result if a match is found. Both order and execution results need to be sequenced first in the sequencer so that matching determinism is guaranteed.

Step 8 - 10: execution result is passed all the way back to the client.

Step 11-12: market data (including the candlestick chart and order book) are sent to the data service for consolidation. Brokers query the data service to get the market data.

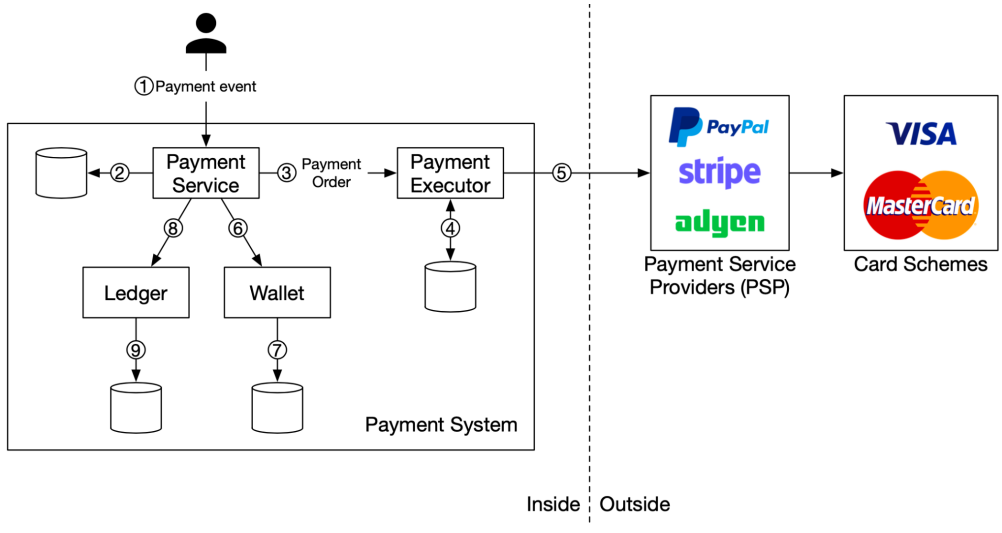
Step 13: the reporter composes all the necessary reporting fields (e.g. client_id, price, quantity, order_type, filled_quantity, remaining_quantity) and writes the data to the database for persistence

A stock exchange requires **extremely low latency**. While most web applications are ok with hundreds of milliseconds latency, a stock exchange requires **micro-second level latency**. I'll leave the latency discussion for a separate post since the post is already long.

Design a payment system

Today is Cyber Monday. Here is how money moves when you click the Buy button on Amazon or any of your favorite shopping websites.

I posted the same diagram last week for an overview and a few people asked me about the detailed steps, so here you go:



1. When a user clicks the “Buy” button, a payment event is generated and sent to the payment service.

2. The payment service stores the payment event in the database.

3. Sometimes a single payment event may contain several payment orders. For example, you may select products from multiple sellers in a single checkout process. The payment service will call the payment executor for each payment order.

4. The payment executor stores the payment order in the database.

5. The payment executor calls an external PSP to finish the credit card payment.

6. After the payment executor has successfully executed the payment, the payment service will update the wallet to record how much money a given seller has.

7. The wallet server stores the updated balance information in the database.

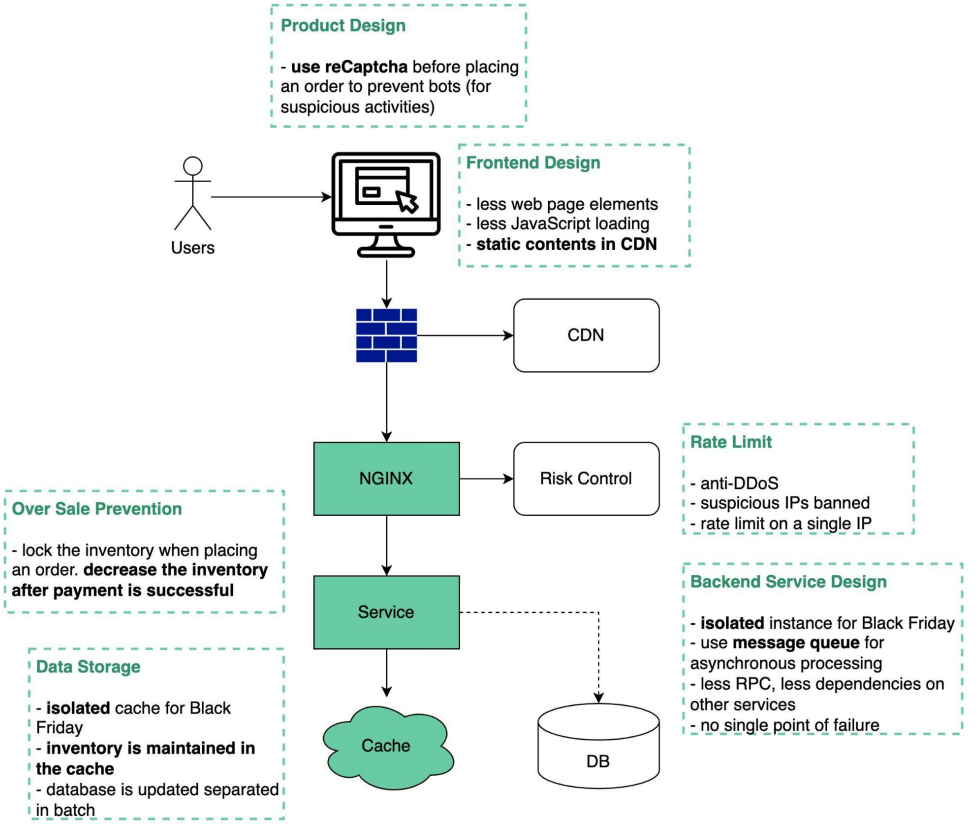
8. After the wallet service has successfully updated the seller's balance information, the payment service will call the ledger to update it.

9. The ledger service appends the new ledger information to the database.

10. Every night the PSP or banks send settlement files to their clients. The settlement file contains the balance of the bank account, together with all the transactions that took place on this bank account during the day.

Design a flash sale system

Black Friday is coming. Designing a system with extremely high concurrency, high availability and quick responsiveness needs to consider many aspects **all the way from frontend to backend.** See the below picture for details:



Design principles:

1. Less is more - less element on the web page, fewer data queries to the database, fewer web requests, fewer system dependencies
2. Short critical path - fewer hops among services or merge into one service
3. Async - use message queues to handle high TPS
4. Isolation - isolate static and dynamic contents, isolate processes and databases for rare items
5. Overselling is bad. When Decreasing the inventory is important

6. User experience is important. We definitely don’t want to inform users that they have successfully placed orders but later tell them no items are actually available