**Questions**

1. How does server sent event work in sprint?

Server-Sent-Events, or SSE for short, is an HTTP standard that allows a web application to handle a unidirectional event stream and receive updates whenever server emits data.

Can be done usign Webstream flux

- @GetMapping(path = "/stream-flux", produces = MediaType.TEXT\_EVENT\_STREAM\_VALUE)

**public** Flux<String> **streamFlux**() { **return** Flux.interval(Duration.ofSeconds(1)) .map(sequence -> "Flux - " + LocalTime.now().toString()); }

2. **WebSockets vs Server-Sent Events:**

Server sent event advantgage

* **Built-in support for reconnection:**Server-Sent Event connections will reestablish a connection after it is lost, meaning less code to write to achieve an essential behavior.

Disadvantage:

* **Data format limitations.**Server-Sent Events are limited to transporting UTF-8 messages; binary data is not supported.
* **Limited concurrent connections.**You can only have six concurrent open SSE connections per browser at any one time. This can be especially painful when you want to open multiple tabs with SSE connections. See '[Server-Sent Events and browser limits](https://stackoverflow.com/questions/18584525/server-sent-events-and-browser-limits)' for more information and workaround suggestions.

The above is all that it meant

3.

The end point here creates an event

**Design google drive**

1.

**The first part here is just all the database stuff that we have**

1. Here the stuff has no difference and we just do it

2.

**Sql vs noql option here**

1. **Inventory Management:** SQL is vital for managing inventory where it's critical to know the relation between several tables like 'Product', 'Supplier' and 'Order'.
2. **Banking System:** Consider the case where you need to transfer money from one account to another. This requires a single transaction performing two updates - a debit from one account and a credit to another.

Nosql database here

How does nosql query data for multiple tables?

1. Use lookup to retrieve additonal data from 1 collection and document ehre and there
2. 2. and referencing to combine data from multiple documents or collections.

**Customers**

|  |  |  |
| --- | --- | --- |
| \_id | name | email |
| 1 | John Doe | johndoe@gmail.com |
| 2 | Jane Smith | janesmith@gmail.com |
| 3 | Bob Johnson | bobjohnson@gmail.com |

**Products**

|  |  |  |  |
| --- | --- | --- | --- |
| \_id | name | price | customer\_id |
| 1 | T-shirts | 10.99 | 1 |
| 2 | Sneakers | 29.99 | 1 |
| 3 | Jeans | 19.99 | 2 |
| 4 | Hoodie | 15.99 | 3 |
| 5 | Hat | 9.99 | 2 |
| 6 | Jacket | 39.99 | 1 |
| 7 | Sweatpants | 24.99 | 3 |

And then using lookup you can do the following here:

db.customers.aggregate([

{

$lookup: {

from: "products",

localField: "\_id",

foreignField: "customer\_id",

as: "products"

}

},

{

$project: {

\_id: 0,

name: 1,

products: "$products.name"

}

}

])

Then look up is used to perform a join like operation between the customers an dproducts here

**Approach 2 Embedded document here**

Embedded documents are documents within documents. In this approach, related data is stored together in a single document, creating a hierarchical structure. This method leverages MongoDB's support for complex document structures.

* **Atomic Operations:** Since all related data is in one document, updates to this document are atomic, ensuring data consistency.
* **Read Performance:** Fetching all related data in a single query is faster and more efficient, as it avoids the need for multiple queries and joins.
* **Data Locality:** Keeping related data together can improve performance, particularly when the data is frequently accessed together.

**This is based on the video**

**https://www.youtube.com/watch?v=cODCpXtPHbQ&t=585s**

**Choosing the database here**

1. What to choose here and why this is important?
2. Some caching is needed here

**File storage option**

**Usually blob storage** here, Amazon S3, can also use CDN with this as well, to serve static image, distrbiute image across the globe so it can be faster for storage

**Text search**

Using elastic search here

What to do when you want to store metric based database?

1. Like clicks and latency and all that, use time-series database, an extensino of relational database
2. You will be doing query based on a time range last 2 hours or sth
3. OpenTSDB

Choosing between relation and non-relation?

1. Structure data? Then relational makes sense
2. Do we need ACID like payment system account A, account B withdrawal  
     
   You need consistency and transaction then use this one right here
3. Unstructured data:

A person standing in front of a whiteboard

Description automatically generated

If you have complex queries

1. Then you need to use document db

Columnar DB

1. Have ever increasing data, Uber for example, drivers keep sending data here, and drivers keep increasing
2. Increasing in an exponential fashion here

A combination of databases?

**Another usecase of buying item in Amazon**

1. If you want to make sure that one item is bought on amazon and u want to make sure only 1 user buys an item at the time
2. This way we can use the relational
3. And then for metrics we can use Cassandra db

Case 2:

Give me all users who bought sugar in the last few days

1. The orders can be in cassandra here,
2. U can store a subset in mongodb (like orderid,

* Easily model in common table, user profile,

**What’s the pros and cons of using a job?**

Key disadvantages of cron jobs:

* Limited error handling:

Cron jobs do not inherently retry failed tasks, meaning if a scheduled job fails, it will not automatically be re-attempted on the next scheduled run.

* Lack of dependency management:

Cron jobs cannot easily manage dependencies between tasks, where one job needs to finish before another can start.

* Poor visibility into execution status:

It can be difficult to monitor if a cron job ran successfully or not without additional logging and monitoring mechanisms.

* Scalability issues in large systems:

Managing cron jobs across a large distributed system can become cumbersome due to the centralized nature of cron.

**What are some alternatives to using a cron job?**

Can use the following here

:Many distributed systems offer dedicated job schedulers with advanced features for managing complex schedules, dependencies, and retries.

**How Slack Built a Distributed Cron Execution System for Scale**

**https://blog.bytebytego.com/p/how-slack-built-a-distributed-cron**

1. Best of cron job at slack

**Using apcahe spark**

**What’s apache spark?**

This is very good for real-time stuff, and can be used for ad-clicking example

Spark uses in-memory caching and optimized query execution to analyze data of any size. It's based on the Hadoop MapReduce model, but Spark processes and keeps data in memory for subsequent steps, which makes it faster

What are some real use cases for this?

* Data Processing and ETL. Data processing and ETL (extract, transform, load) are critical components in data engineering workflows. ...
* Stream Processing. ...
* Machine Learning and AI. ...
* Data Analytics. ...
* Log Processing. ...
* Fog Computing. ...
* Recommendation Systems. ...
* Real-time Advertising.

**Timebased window solution**

**Word and now this is connected as well as said previously as you have said multiple times before and I think thi is quite important**

**What are the 3 types of consistency? Can you give exmaples?**

**Consistency type 1: Strong consistency**

. It ensures that all clients see the same sequence of updates and that updates appear to be instantaneous. Achieving strong consistency often requires coordination and synchronization between distributed nodes, w

**Example**

A traditional SQL database system Achieved with a single master node and multiple replica

*When a client writes data to the master node, subsequent reads from any replica will immediately reflect the latest value written. All replicas are updated synchronously, ensuring that all clients see a consistent view of the data.*

And the above is problem 1 here:

**Consistency type 2: Eventual consistency**

While eventual consistency may lead to temporary inconsistencies, it guarantees that all updates will be eventually propagated and reconciled.

*Amazon's DynamoDB, a distributed NoSQL database, provides eventual consistency. When data is written to DynamoDB, it is initially stored locally on a single node and then asynchronously propagated to other nodes in the system. While clients may read slightly outdated values immediately after a write, all replicas eventually converge to the same value over time.*

**Type 3: Weak consistency here**

 only guarantees that updates will eventually propagate to all replicas. Unlike eventual consistency, which ensures convergence, weak consistency does not provide any guarantees about when or if replicas will converge. I

Allows for concurrent updates here

*A distributed caching system, such as* [*Redis*](https://www.geeksforgeeks.org/introduction-to-redis-server/) *or Memcached, often implements weak consistency. In such systems, data is stored and retrieved quickly from an in-memory cache, but updates may be asynchronously propagated to other nodes. This can lead to temporary inconsistencies where clients may observe old or divergent values until updates are fully propagated.*

Sharding section here (using the sharding here)

What are the different types of sharding?

**1. Key Based Sharding**

This technique is also known as **hash-based** sharding. Here, we take the value of an entity such as customer ID, customer email, IP address of a client, zip code, etc and we use this value as an input of the **hash function**. This process generates a **hash value** which is used to determine which shard we need to use to store the data.

* We need to keep in mind that the values entered into the hash function should all come from the **same column** (shard key) just to ensure that data is placed in the correct order and in a consistent manner.
* Basically, shard keys act like a primary key or a unique identifier for individual rows.

How to determine which server data should be placed on?

To determine which server data should be placed on, we perform a modulo operation on these applications id with the number 3. Then the remainder is used to identify the server to store our data.

This is the same modulus operation that was done before no difference

Advantage:

* Key-based sharding provides a predictable way to distribute data across shards.

Disadvatage

* **Uneven Data Distribution:** If the sharding key is not well-distributed it may result in uneven data distribution across shards
* **Limited Scalability with Specific Keys:** The scalability of key-based sharding may be limited if certain keys experience high traffic or if the dataset is heavily skewed toward specific key ranges

2. Horizontal or range-based sharding here

Using range-based sharding here

we divide the data by separating it into different parts based on the range of a specific value within each record. Let’s say you have a database of your online customers’ names and email information. You can split this information into two shards.

A diagram of a database

Description automatically generated with medium confidence

And below is the part that really matters here

1 **Advantages of Range Based Sharding:**

* **Scalability:** Horizontal or range-based sharding allows for seamless scalability by distributing data across multiple shards, accommodating growing datasets.
* **Improved Performance:** Data distribution among shards enhances query performance through parallelization, ensuring faster operations with smaller subsets of data handled by each shard.

**Disadvantages of Range Based Sharding:**

* **Can lead to hotspot as mentioned**

  For example, if a significant portion of the products in the database falls within a specific price range, the shard responsible for storing that range may experience a disproportionately high load, while other shards remain underutilized.

**Key or Hash-Based Sharding**

Key-based sharding, also known as hash-based sharding, is a technique that assigns a particular key to a shard using a hash function.

A well-designed hash function plays a crucial role in achieving a balanced distribution of keys. Instead of assigning a range of keys to each shard, hash-based sharding assigns a range of hashes to each shard. Consistent Hashing is one technique that’s often used to implement hash-based sharding.

The diagram below illustrates the basic concept of key or hash-based sharding:

[[A diagram of a key to a key

Description automatically generated](https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F3e1bfd20-ed30-4720-9f6d-a9914c650dd1_1600x1002.png)](https://substackcdn.com/image/fetch/f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F3e1bfd20-ed30-4720-9f6d-a9914c650dd1_1600x1002.png" \t "_blank)

Advantage here:

One of the main advantages of hash-based sharding is its ability to distribute keys fairly among the shards. By applying a hash function to the keys, the technique helps mitigate the risk of hot spots.

Disadvantage here:

However, hash-based sharding comes with a trade-off. By using the hash of the key instead of the key itself, we lose the ability to perform efficient range queries. This is because adjacent keys may be scattered across different partitions, and their natural sort order is lost in the process.

Hash-based sharding helps with this, but does not eliminnte the problem here

What r some factors considered when choosing a shard key?

**Cardinality**

1. Cardinality refers to the number of possible values that a shard key can have. It determines the maximum number of shards that can be created.

2. For example, if a boolean data field is chosen as the shard key, the system will be limited to only two shards.

3. To maximize the benefit of horizontal scaling, it is generally recommended to select a shard key with high cardinality.

**Frequency**

1. The frequency of a shard key represents how often a particular shard key value appears in the dataset.

2. If a significant portion of the records contains only a subset of the possible shard key values, the shard responsible for storing that subset may become a hotspot.

3. For instance, if a fitness website’s database uses age as the shard key, most records may end up in the shard containing subscribers between the ages of 30 and 45, leading to an uneven distribution of data.

Why is consistent hashing so important?

It’s the technique to achieve even distrubtion of data in horizontal scaling across servers or database serverss, this is why this is important.

Consistent hashing is a special kind of hashing such that when a hash table is re-sized and consistent hashing is used, only k/n keys need to be remapped on average, where k is the number of keys, and n is the number of slots. In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped [1]”.

What is it basically?

* Map servers and keys on to the ring using a uniformly distributed hash function.
* To find out which server a key is mapped to, go clockwise from the key position until the first server on the ring is found.
* When new server is added, only one key needs to be remapped here

**We make use of sth called virtula nodes.**

**If we have a non-unifomr key distribution on the ring, how do we solve this problem?**

A virtual node refers to the real node, and each server is represented by multiple virtual nodes on the ring. In Figure 12, both *server 0* and *server 1* have 3 virtual nodes. The 3 is arbitrarily chosen; and in real-world systems,

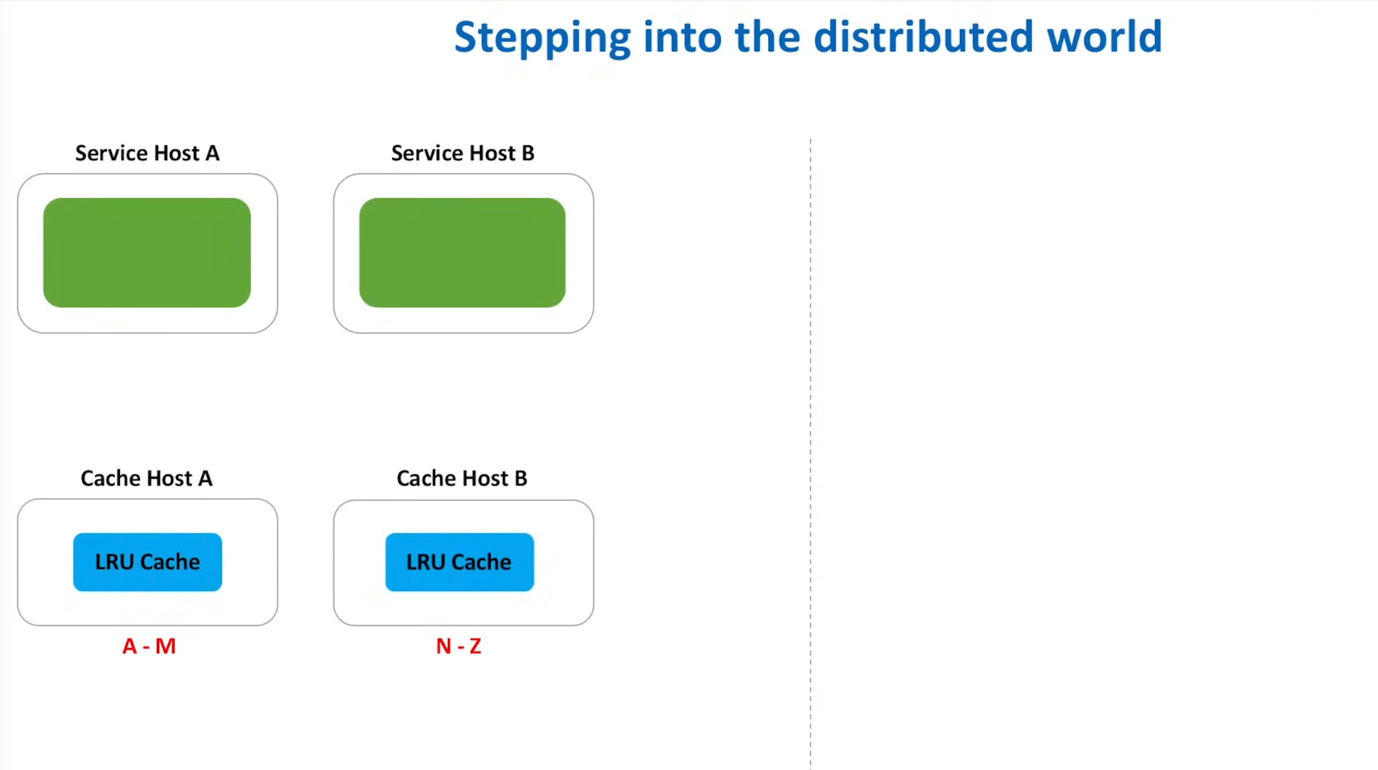
the number of virtual nodes is much larger. Instead of using *s0*, we have *s0\_0, s0\_1*, and s0*2 to represent \_server 0* on the ring. Similarly, *s1\_0, s1\_1*, and *s1\_2* represent server 1 on the ring. With virtual nodes, each server is responsible for multiple partitions. Partitions (edges) with label *s0* are managed by server 0. On the other hand, partitions with label *s1* are managed by *server 1*.

What’s the best strategy to avoid hot spot in sharding?

- Can use hash-based sharding but does not eliminiate the problem entirely as said

-

1. In the real world here

****

**This is the ad click aggregator example here**

Advertisers can query ad click metrics over time at 1 minute intervals

**How do we make sure user can query ad click metrics over time at 1 minute intervals?**

Basically we want to know when a click comes in and how many counts here.

Bad approach 1: Store and query from the same database

When a click comes in from a user, we can simply store that click event in our database. The schema would look something like this:

| EventId | Ad Id | User Id | Timestamp |
| --- | --- | --- | --- |
| 1 | 123 | 456 | 1640000000 |

Then, when an advertiser wants to query metrics, we would query the database for all events that match the ad ID and time range running a GROUP BY query to get the metrics that they need. For example,

SELECT

COUNT(\*) as total\_clicks,

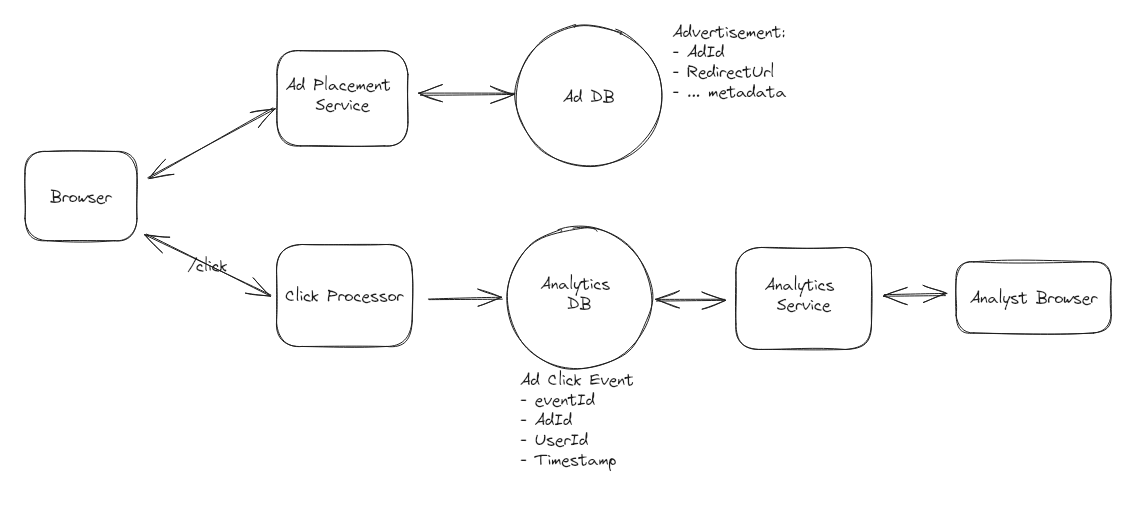
COUNT(DISTINCT UserId) as unique\_users

FROM ClickEvents

WHERE AdId = 123

AND Timestamp BETWEEN 1640000000 AND 1640000001

GROUP BY AdId



The database would become a bottleneck if scale to 10000 k per clicks.

Option 2: Using batch processing

en, in batches, we can process the raw events and aggregate them into a separate database that is special optimized for querying.

The schema for our analytics database could look something like this:

| Ad Id | Minute Timestamp | Unique Clicks |
| --- | --- | --- |
| 123 | 1640000000 | 100 |

We store the above in a separate database and process this over there.

For a lot of writes:

1. Use Cassandra to write to the event table

For batch processing:

1. Use Apache spark

**How would this work exactly?**

Let's take a step back and recap the system we've designed so far. When a user clicks on an ad:

1. The ad click is sent to the click processor service.
2. The click processor service writes the event to our event store.
3. Every N minutes, a cron job kicks off a spark job that reads the events from the event store and aggregates them.
4. The aggregated data is stored in an OLAP database for querying.
5. Advertisers can query the OLAP database to get metrics on their ads.

Problems with this approach:

This solution works, and it works pretty well, but there are several key challenges that prevent it from being a "great" solution. First, the batched processing job introduces a significant delay in the data pipeline. Advertisers will always be querying data that is at a couple minutes old. This is not ideal and something we can improve upon. Second, the system is not scalable. If we have a sudden spike in clicks, the system will not be able to handle the load.

**Option 3: use a stream processing design here**

To address the latency and scalability issues, let's introduce a stream for real-time processing. This system allows us to process events as they come in, rather than waiting for a batch job to run.

When a click comes in our click processing service will immediately write the event to a stream like [**Kafka**](https://kafka.apache.org/) or [**Kinesis**](https://aws.amazon.com/kinesis/). We then need a stream processor like [**Flink**](https://flink.apache.org/) or [**Spark Streaming**](https://spark.apache.org/streaming/) to read the events from the stream and aggregate them in real-time.

Now, when a click comes in:

1. The click processor service writes the event to a stream like Kafka or Kinesis.
2. A stream processor like Flink or Spark Streaming reads the events from the stream and aggregates them in real-time.
3. The aggregated data is stored in our OLAP database for querying.
4. Advertisers can query the OLAP database to get metrics on their ads in near real-time.

And then next here

And then we have the code

**How to decide which shard to call?**

1. We can use either consistent hashing or the ot

1. cacheHostNumber = hash\_function(key ) % numberOfCacheHosts

Choosing a cache host here

What are the top patterns for high availability?

1. Using load balancer:

By dividing up incoming traffic among several servers or instances, load balancers provide fault tolerance, scalability, and optimal resource use

2. Database replication

By replicating data among several nodes, database replication technologies guarantee data availability, consistency, and resilience to outages.

3. Cluster management:

To ensure high availability and fault tolerance, containerized applications are deployed, scaled, and managed automatically by cluster management frameworks like Kubernetes and Apache Mesos.

What database is good for high writes?

Ad click aggregator solution here

**This document has been broken down into different sections here**

**How to handle high write database here**

**What**

1. Batching: Group multiple write operations into batches to reduce the overhead of individual requests. This reduces the number of network round-trips and improves throughput by processing multiple operations in a single transaction.
2. Asynchronous Processing: Offload non-critical or time-consuming write operations to background tasks or worker queues. This allows the system to continue processing other requests without waiting for the completion of the write operations, improving overall responsiveness and throughput.
3. Indexing: Use indexes strategically to speed up write operations, especially for frequently queried fields or columns. However, be mindful of the overhead of maintaining indexes, especially in high-update scenarios, and consider trade-offs between read and write performance.
4. Partitioning and Sharding: Partition data across multiple nodes or shards to distribute the write load and improve scalability. This allows the system to handle a higher volume of write operations by   
   parallelizing the processing across multiple resources.
5. Write Ahead Logging (WAL): Use write-ahead logging techniques to optimize write durability and recovery. Write operations are first logged to a durable log before being applied to the main data store, ensuring data durability while minimizing disk I/O overhead.

Caching: Cache frequently accessed data in memory to reduce the number of write operations to the underlying storage layer. This improves write performance by reducing latency and I/O overhead for commonly accessed data.

**How do we prevent multiple ride requests from being sent to the same driver simultaneously? (Uber)**

We defined consistency in ride matching as a key non-functional requirment. This means that we only request one driver at a time for a given ride request AND that each driver only receives one ride request at a time.

That driver would then have 10 seconds to accept or deny the request before we move on to the next driver if necessary.   
  
 If you've solved Ticketmaster before, you know this problem well -- as it's almost exactly the same as ensuring that a ticket is only sold once while being reserved for a specific amount of time at checkout.

**Option 1: Lock with ttl**

To solve the timeout issue, **we can use a distributed lock implemented with an in-memory data store like Redis.**   
  
 When a ride request is sent to a driver, a lock is created with a unique identifier (e.g., driverId) and a TTL set to the acceptance window duration of 10 seconds.  
  
 The Ride Matching Service attempts to acquire a lock on the driverId in Redis. If the lock is successfully acquired, it means no other service instance can send a ride request to the same driver until the lock expires or is released.  
  
 If the driver accepts the ride within the TTL window, the Ride Matching Service updates the ride status to "accepted" in the database, and the lock is released in Redis. If the driver does not accept the ride within the TTL window, the lock in Redis expires automatically. This expiration allows the Ride Matching Service to consider the driver for new ride requests again.

**Challenges**

The main challenge with this approach is the system's reliance on the availability and performance of the in-memory data store for locking. This requires robust monitoring and failover strategies to ensure that the system can recover quickly from failures and that locks are not lost or corrupted. Given locks are only held for 10 seconds, this is a reasonable tradeoff as the emphemorality of the data makes it easier to recover from failures.

**Solution: Distributed Locking with Redis**

To address this problem, we can use distributed locking with Redis. Here's how it would work:

1. Before starting any transaction, the respective microservice tries to acquire a lock for the specific account in Redis witth the account id.
2. If the lock is acquired, the microservice performs the transaction and then releases the lock.
3. If the lock can't be acquired, the microservice waits and retries.

**Advantages and Disadvantages**

**Advantages:**

1. **Consistency**: Ensures that only one transaction is processed at a time for a single account.
2. **Scalability**: Using Redis allows us to handle a large number of concurrent requests.
3. **High Availability**: Using Redis Sentinel or Cluster increases system reliability.

**What are the disadvantages?**

**Disadvantages:**

1. **Latency**: Acquiring and releasing locks may introduce small delays.
2. **Complexity**: The system becomes more complex, making debugging and monitoring more challenging.
3. **Resource Usage**: Requires additional infrastructure for Redis.

**How do we ensure system resilience?**

**System Resilience**

In a microservices architecture, ensuring system resilience is crucial. Here are some techniques we can employ:

1. **Circuit Breaker**: Use the Circuit Breaker pattern to prevent cascading failures.
2. **Retry Mechanism**: Implement a smart retry mechanism to handle transient failures.
3. **Rate Limiting**: Introduce rate limiting to protect the system from excessive load

How do we make sure that advertisers can query metrics at low latency? (Ad click aggregator example)

This can be done using a Cron job,

We can pre-aggregate the data in the OLAP database,

This can be done by creating a new table that stores the aggregated data at a higher level of granularity, like daily or weekly.

Example:   
 This can be via a nighly cron job that runs a query to aggregate the data and store it in the new table.

When an advertiser queries the data, they can query the pre-aggregated table for the higher level of granularity and then drill down to the lower level of granularity if needed.

**How can we ensure that the redirect (redirects to the original url) is fast in url shortener?**

Solutino 1: Add an index or use redis for caching  
  
 Avoid a full scan,

What’s the disadvantage of caching? What’s the caching invalidation startegy that should be used in this case.

**Solution 2: Leveraging CDN and edging computer**

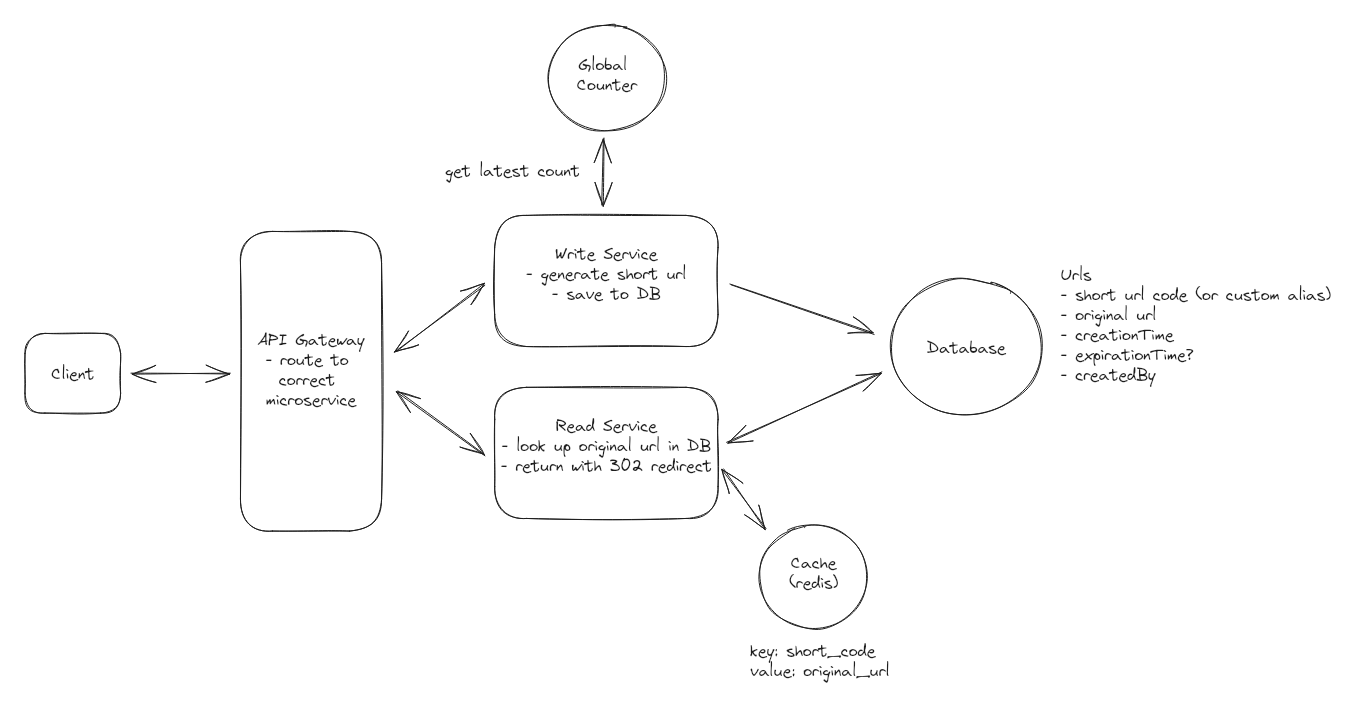
Another thing we can do to reduce latency is to utilize Content Delivery Networks (CDNs) and edge computing. In this approach, the short URL domain is served through a CDN with Points of Presence (PoPs) geographically distributed around the world.

The CDN nodes cache the mappings of short codes to long URLs, allowing redirect requests to be handled close to the user's location. Furthermore, by deploying the redirect logic to the edge using platforms like Cloudflare Workers or AWS Lambda@Edge, the redirection can happen directly at the CDN level without reaching the origin server.

**How to deal with having a centralized counter? (Url shortener)**

Using a single source of truth here

We could solve this by using a centralized Redis instance to store the counter. This Redis instance can be used to store the counter and any other metadata that needs to be shared across all instances of the Write Service. Redis is single-threaded and is very fast for this use case.  
  
 It also supports atomic increment operations which allows us to increment the counter without any issues. Now, when a user requests to shorten a url, the Write Service will get the next counter value from the Redis instance, compute the short code, and store the mapping in the database.



**Ticket master section**

**How to ensure that the ticket is locked for the user whilie they are checking out?**

Now, when a user wants to book a ticket:

1. A user will select a seat from the interactive seat map. This will trigger a POST /bookings with the ticketId associated with that seat.
2. The request will be forwarded from our API gateway onto the Booking Service.
3. The Booking Service will lock that ticket by adding it to our Redis Distributed Lock with a TTL of 10 minutes (this is how long we will hold the ticket for).
4. The Booking Service will also write a new booking entry in the DB with a status of in-progress.
5. We will then respond to the user with their newly created bookingId and route the client to a the payment page.
   1. If the user stops here, then after 10 minutes the lock is auto-released and the ticket is available for another user to purchase.
6. The user will fill out their payment details and click “Purchase.” In doing so, the payment (along with the bookingId) gets sent to Stripe for processing and Stripe responds via webhook that the payment was successful.
7. Upon successful payment confirmation from Stripe, our system's webhook retrieves the bookingId embedded within the Stripe metadata. With this bookingId, the webhook initiates a database transaction to concurrently update the Ticket and Booking tables. Specifically, the status of the ticket linked to the booking is changed to "sold" in the Ticket table. Simultaneously, the corresponding booking entry in the Booking table is marked as "confirmed."
8. Now the ticket is booked!

**How to allow users to scale to support 10 million of concurrent requests per day?**

* Prioritize caching for data with high read rates and low update frequency, such as event details (names, dates, venue information), performer bios, and static venue details like location and capacity. Because this data does not change frequently, we can cache it like crazy to heavily minimize the load of our SQL DB and meet our high availability requirements.
* [**Cache**](https://www.hellointerview.com/learn/system-design/in-a-hurry/key-technologies#distributed-cache) key-value pairs like eventId:eventObject to efficiently serve frequently accessed data.
* Utilize Redis or Memcached as in-memory data stores, leveraging their speed for handling large volumes of read operations. A read-through cache strategy ensures data availability, with cache misses triggering a database read and subsequent cache update.
* Cache Invalidation and Consistency:
  + Set up database triggers to notify the caching system of data changes, such as updates in event dates or performer lineups, to invalidate relevant cache entries.
  + Implement a Time-to-Live policy for cache entries, ensuring periodic refreshes. These TTLs can be long for static data like venue information and short for frequently updated data like event availability.

- Also using horizontal scaling is important here

**How do we make sure that the booking map is always up to date?**

Option 1: SSE for real time seat updates

1. To ensure that the seat map is always up to date, we can use [**Server-Sent Events (SSE)**](https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events) to push updates to the client in real-time. This will allow us to update the

Challenge

2. This might not scale well for “taylor swift” user

Option 2: using virtual waiting q for popular events for websocket connection

For extremely popular events, we can implement an admin enabled virtual waiting queue system to manage user access during times of exceptionally high demand. Users are placed in this queue before even being able to see the booking page (seat map selected).

1. When a user requests to view the booking page, they are placed in a virtual queue. We establish a WebSocket connection with their client and add the user to the queue using their unique WebSocket connection.
2. Periodically or based on certain criteria (like tickets booked), dequeue users from the front of the queue. Notify these users via their WebSocket connection that they can proceed to purchase tickets.
3. At the same time, update the database to reflect that this user is now allowed to access the ticket purchasing system.

**Challenges**

Long wait times in the queue might lead to user frustration, especially if the estimated wait times are not accurate or if the queue moves slower than expected. By pushing updates to the client in real-time, we can mitigate this risk by providing users with constant feedback on their queue position and estimated wait time.

**How can you improve text search to ensure we meet our low latency requirements?**

SELECT \*

FROM Events

WHERE name LIKE '%Taylor%'

OR description LIKE '%Taylor%'

1. Create indexes on event table,

2 Use full text indexes in the database such as mysql or postgresql here:

We can extend the basic indexing strategy above to utilize full-text indexes in our database, if available. For popular SQL databases like MySQL or Postgres, full text extensions are available which utilize search engines like Lucene under the covers. These make queries for specific strings like "Taylor" or "Swift" much faster than doing a full table scan using LIKE.

**Challenges**

* Full text indexes require additional storage space and can be slower to query than standard indexes.
* Full text indexes can be more difficult to maintain, as they require special handling in both queries and in maintaining the database.

**How does elastic search sync with Postgresql?**

**Approach**

Add [**Elasticsearch**](https://www.hellointerview.com/learn/system-design/deep-dives/elasticsearch) or a similar [**full-text search engine**](https://www.hellointerview.com/learn/system-design/in-a-hurry/key-technologies#search-optimized-database). Elasticsearch is a powerful search engine that excels in full-text search, complex query execution, and handling high-volume traffic efficiently.   
  
 At its core, Elasticsearch operates using inverted indexes, a key feature that makes it highly efficient for search operations. Inverted indexes allow Elasticsearch to quickly locate and retrieve data by mapping each unique word to the documents or records it appears in, significantly speeding up search queries.

* To make sure the data in Elasticsearch is always in sync with the data in our SQL DB, we can use change data capture (CDC) for real-time or near-real-time data synchronization from PostgreSQL to Elasticsearch. This setup captures changes in the PostgreSQL database, such as inserts, updates, and deletes, and replicates them to the Elasticsearch index.
* We can enable fuzzy search functionality with Elasticsearch, which allows for error tolerance in search queries. This is way we can handle typos and slight variations in spellings such as "Taylor Swift" vs "Tayler Swift". This is something that would be very difficult to do with SQL alone.

**Challenges**

* Keeping the Elasticsearch index synchronized with PostgreSQL can be complex and requires a reliable mechanism to ensure data consistency.
* Maintaining an Elasticsearch cluster adds additional infrastructure complexity and cost.

**What’s the role of a coordinator node?**

Coordinator same as a master/slave architecture,

There is *usually a single coordinator*(or master) node and there are many worker nodes. The overall goal is to divide big requests, which would require lots of CPU or RAM on a single machine, into smaller tasks so that they can be done faster and more reliably in a distributed fashion.

What are the job duties of each node?

**Coordinator node is responsible for:**

* Handling client’s requests
* Dividing up requests into smaller tasks(if necessary)
* Sending small tasks to worker nodes
* Keeping track of which tasks are done
* Retrying failed tasks
* Keeping track of current running worker nodes

**Worker nodes are responsible for:**

* Handling a defined task(e.g. running a query over S3 files, or computing sum of primes over a small range)
* Sending the result back to the coordinator nodes
* Updating their status(number of current tasks running, available CPU, RAM etc.) with the coordinator node

The main goal of this episode is to introduce this architecture and to demonstrate the implementation details of it using our *Sum of Primes* example in K8s.

**What happens if a coordinator node fails?**

We are using a **coordinator node** in both the **Scheduling** and **Execution** service.

To prevent the **coordinator** from becoming a **single point of failure**, deploy multiple **coordinator nodes** with a **leader-election mechanism**.

This ensures that one node is the active leader, while others are on standby. If the leader fails, a new leader is elected, and the system continues to function without disruption.

* **Leader Election**: Use a consensus algorithm like **Raft** or **Paxos** to elect a leader from the pool of coordinators. Tools like **Zookeeper** or **etcd** are commonly used for managing distributed leader elections.
* **Failover**: If the leader coordinator fails, the other coordinators detect the failure and elect a new leader. The new leader takes over responsibilities immediately, ensuring continuity in job scheduling, worker management, and health monitoring.
* **Data Synchronization**: All coordinators should have access to the same **shared state** (e.g., job scheduling data and worker health information). This can be stored in a **distributed database** (e.g., Cassandra, DynamoDB). This ensures that when a new leader takes over, it has the latest data to work with.

**How is batching done in spring boot?**

This is an interesting questino here, how is this done here usinb spring batch.

to its fullest potential, ensuring robust communication in your distributed applications.

f

**How do we scale kafka event-driven microservice here?**

1. Can be achieved by increasing # of instance of the respective services

 Kafka partitions are the unit of parallelism, and each partition is consumed by only one consumer in a consumer group. Therefore, we should have enough partitions to allow for efficient parallel processing.

3. Add more consumers to a consumer group here

To scale consumers, we need to set a unique instanceId for each consumer instance and add multiple instances of the same consumer to the consumer group.

@Service  
public class KafkaConsumerService {  
  
 @Value("${kafka.instanceId}")  
 private String instanceId;  
  
 @KafkaListener(  
 topicPartitions = @TopicPartition(  
 topic = "example\_topic",  
 partitionOffsets = {  
 @PartitionOffset(partition = "0", initialOffset = "0"),  
 @PartitionOffset(partition = "1", initialOffset = "0"),  
 // Add more partitions and offsets as needed  
 }  
 ),  
 groupId = "example\_group"  
 )  
 public void receiveMessage(String message) {  
 System.out.println("[" + instanceId + "] Received message: " + message);  
 }  
}

In this example, we used the @KafkaListener annotation

with topicPartitions to specify the partitions and their initial offsets for this consumer instance. Each consumer instance will process events from different partitions, increasing parallelism.

groupId: This attribute specifies the consumer group to which this consumer belongs. Consumers in the same group share the load of consuming messages from the topic partitions.

To create multiple instances of the consumer service, we need to run the application with different instanceId values. We can use Spring profiles to achieve this.

Then we use the following for this

@SpringBootApplication  
public class KafkaApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.run(KafkaApplication.class, "--spring.profiles.active=instance1");  
 SpringApplication.run(KafkaApplication.class, "--spring.profiles.active=instance2");  
 // Add more instances as needed  
 }  
}

In this example, we started two instances of the application with profiles “instance1” and “instance2.” Each instance will have a unique instanceId, and they will consume events from different partitions, providing better parallelism.

**How to increase # of partition in a group?**

@Configuration  
public class KafkaTopicConfiguration {  
  
 @Bean  
 public NewTopic exampleTopic() {  
 return new NewTopic("example\_topic", 4, (short) 1);  
 }  
  
 @Bean  
 public NewTopic deadLetterTopic() {  
 return new NewTopic("example\_topic\_dead\_letter", 1, (short) 1);  
 }  
}

In this example the partition has gone up to 4 here.

**The top questoins here**

**How to achieve a low latency here**

**An example: How do we make sure user can query ad click metrics over time at 1 minute intervals?**

- can use an index,

- use caching

- use hash table or tree for fast retrieval and look up here

- Use load balancer : Distribute incoming traffic evenly across multiple servers using load balancers to prevent overloading any single component.

- using asynchornous processing, this needs to be non-blockign here

**How to achieve a high availability?**

High availability is often measured in terms of uptime, which is the ratio of time that a system is operational to the total time it is supposed to be operational.

Have to ensure high availability here, and that’s number 1 here.

Strategies are as below:

**1. Using redundancy and replication**

By duplicating critical components or entire systems, organizations can ensure that if one fails, the redundant system takes over seamlessly, avoiding any interruption in service.

Replication involves creating multiple copies of data here

2. Using load balancing

1. Load balancing involves [distributing workloads across multiple servers](https://www.designgurus.io/course-play/grokking-system-design-fundamentals/doc/64240a259fe8f8ee37ced12e), ensuring that no single server is overwhelmed.

2. No server is overloaded here if using this apporach here

3. Failover clusttering for database

1. If 1 db fails and then others will come up to save this here this is also important here

4. Health monitoring and alerts

Implementing robust health monitoring systems ensures that organizations can proactively identify and address potential issues before they impact system availability.

Real-time monitoring and automated alerts enable timely response and rapid resolution of problems, minimizing downtime.

6. Once websocket connection is built where do we store the mapping? Each server will map to a single post? Where is the mapping stored?

7. When do we need to use a zookeeper? How do we maintain a good websocket connection

**Chat system design**

1. How are websocket connections handled with websocket server? What’s the max it can handle?

2. How do user receive message in a group chat here

Using the code here

1.

A diagram of a computer flow

Description automatically generated with medium confidence

**Hotel reservation section**

**How to prevent the 2 users from taking 1 transaction?(Can check the hotel reservation system)**

1. Pessimistic locking
2. Optimistic locking (using version)
3. Constraint