Case study 7 (Design IRCTC)

Agenda

Why IRCTC?

- IRCTC deals with a very high level of concurrency.
  - It means you absolutely cannot have one train seat booked by two different users.
- It also needs to generate a lot of throughput.
  - A lot of people in a particular time slot try to book tickets. This happens especially during tatkal booking.
  - So, with so much high traffic volume, how do you still make sure that no seat is double booked?

These points make IRCTC a very interesting product.

Minimum Viable Product

- User Registration
- Given a source, destination and date, fetch the list of trains connecting the source to destination.
- Given a trainID, class and date, check for seat availability.
- For a given train, class, and date, book tickets if seats are available.
- Given a trainID, get all stops of the train (planned schedule).
- Payment gateway
- Notification of ticket booking (email, message)

Estimation of Scale

- Let's assume 10000 trains run per day.
  - Each bogie has an average of 72 seats.
  - Assuming a train has an average of 15 bogies, total number of seats in a train on average is equal to 15 * 72 = 1080 seats.
- Hence, total number of seats to be booked in all trains = 10000 * 1080 which is approximately 10 million seats.

Need for Sharding?

- **User** table which stores the credentials of IRCTC users may hold upto one billion entries equivalent to the population of India.
- The **Bookings** table will store the details of bookings such as userID, trainID, date, seat details, src and destination.
  - Historical bookings will be stored in a separate table.
  - The **Bookings** table stores ticket details of current and upcoming journeys only.
- Now, IRCTC allows booking of train seats upto 3 months in future. Hence, in the worst case, all train seats for the next 90 days are booked by users.
- This leads to a total 90 * 10 = 900 million seat bookings. Hence, the maximum number of records in the Bookings table can grow upto **900 million**.
- Assuming userID (8B), trainID(8B), date(8B), seat(4B), src(4B), destination(4B), the size of a single record is **36 Bytes,** let's approximate this to 50 Bytes.
- Total storage size needed for storing **Bookings** data= 50 * 900 million Bytes = 50 GB.
- Again, assuming the User table size to be **100 GB** (since it contains 1 Billion records at max), we have to store around 150 GB of data to get the system functioning.

Hence, as such there is **no need of sharding**. However, you might choose to shard if your design requires that. But based on the volume of data, there is no need to shard.

Read or Write Heavy System

System Type 1

Trains List and Schedules (A Static Microservice)

Mostly Read operations are carried on the tables containing details of:

- Trains running on specific dates from a source to destination
- Train schedules always handle read operations
- These details are almost static and there are mostly read operations.
- We can consider this system as a separate microservice built on heavy caching and replicas.

System Type 2

Seat Availability System

Given the trainID, src, destination, date and class, show the number of seats available for booking.

- This system should be eventually consistent.
- There is no concept of a highly consistent system in this regard as well. It is because the number of available seats are changing every second, hence the system should be eventually consistent.
- The exact number of seats available does not matter that much as the data is changing every second.
- The system can become consistent in the time gap between showing the availability of seats and booking the seats.
- Also, this system is a read heavy system. It derives its data from the **Bookings** Table.

System Type 3

Now, write operations are carried out on the **Bookings** table. In this regard:

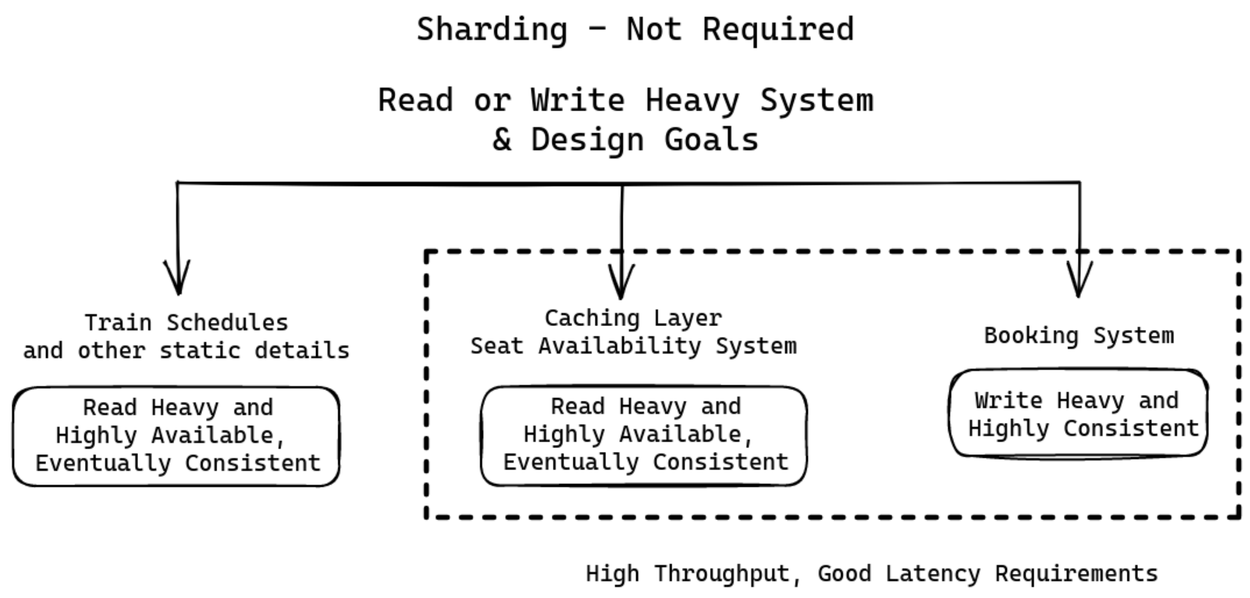
- Booking of seats needs to be highly consistent. Any seat cannot be booked twice in any situation.
- The Booking system is HC and it is both a read and write heavy system.

System **2** and **3** are dependent on each other. Whenever there are write operations on the **Bookings** table, the **Seat Availability System** has to update itself to become eventually consistent.

System **2** can be thought of as a Caching layer that keeps track of seat availability based on the Bookings system. The Booking layer/system in such a case is write heavy and Caching layer is ready heavy.

High Throughput is one of the design goals as during the peak time, for example in tatkal booking, the system would have to handle millions of requests in a small period of time.

Summary Till Now


Sharding - Not Required
Read or Write Heavy System & Design Goals
Train Schedules and other static details
Read Heavy and Highly Available, Eventually Consistent
Caching Layer Seat Availability System
Read Heavy and Highly Available, Eventually Consistent
Booking System
Write Heavy and Highly Consistent
High Throughput, Good Latency Requirements

API

getNumberOfAvailableSeats

It will have following arguments:

- trainID
- src
- dest
- date
- class
- class (or list of class)

bookSeats

It will have following major arguments:

- userID
- trainID
- src
- dest
- date
- class
- number_seats
- passenger_list

Problem #1 (Consistency):

Suppose a request comes to the load balancer trying to book a seat. Now the system has to make sure that once a seat is assigned to this request, no other request can claim that seat. In other words, the system has to be completely consistent. How do you do that?

There are essentially 3 steps:

- **Step 1:** Check for availability, let’s say 1 seat is available, **X**.
- **Step 2:** Book the seat X
- **Step 3:** Return X

When we transition from Step 1 and Step 2, there are chances that there are no seats left (even though step 1 indicated availability of a seat). How to ensure these three steps are **atomic** in nature? Either all of them happen or none.

Could we leverage the atomicity property of relational DBs?
Answer:

- Leverage the Atomicity property of Relational DBs to solve this problem.
- If you try to make the sequence of operations atomic on the application side (servers), it becomes extremely hard to make the entire operation atomic.
- Since the eventual source of data is your database, hence you should think of ways to make it atomic on your database itself.
- The advantage of this approach is that if you pass one large query, it provides you guarantee that either the entire query will succeed or nothing will succeed.
- Assume the Bookings table is prepopulated with the details of all seats of all trains and each seat is available for the next 90 days.

Query to allocate seat:

```
UPDATE Bookings
SET available = userID
WHERE
        trainID = T and
        date = D and
        (src = S ….) and
        available = 1
        LIMIT 1;
```

- The where clause checks for seat availability, and only from available seats it will book one seat if available. **LIMIT 1** implies only one of the available seats will be impacted randomly.
- If the above query is executed atomically, either both Update and where clause succeed or none of them succeed.

- It cannot happen that the where clause gives you an available seat but before updating the bookings table, that seat is booked. This is due to the way atomic operations are handled in RDBMS.
- The replicas of **Bookings** database do not need to run the queries. They can get transaction **logs** from the main database and update themselves accordingly. It is similar to the master slave system. All the slaves receive the transaction logs and update themselves.
- A highly consistent system means that the master should return success only after some number of slaves have also written the update through the logs.

Problem #2: How to handle berth preference?

It's the art of writing SQL queries. You can use ORDER BY to sort the records according to the berth preference. So the WHERE clause filters records and ORDER BY clause is for handling preferences.

Problem #3: Increase the Throughput

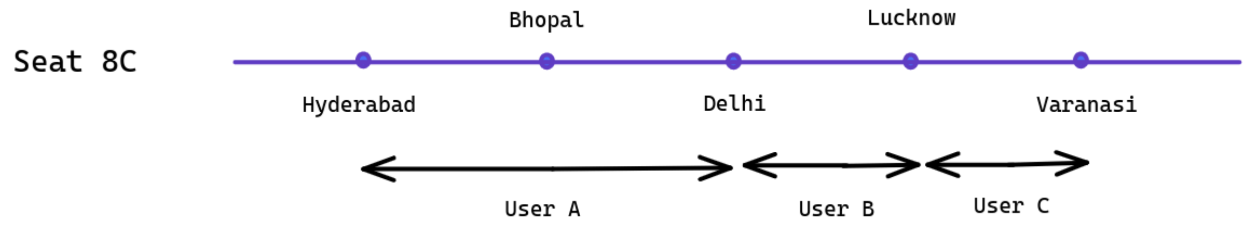
All writes have to go to the master DB in master slave. If there is only one master, it will become the bottleneck. So, how do we increase throughput?
What if we shard? Not because we need to for storage, but because it helps us have different independent masters to increase write throughput.

- Best solution is to shard based on trainID.
    - A single shard can contain multiple trains but a single train is on one shard only.
- The trainID is a good sharding key because:
    - Ticket booking happens on only one train at one time.
- Now the throughput is the summation of the throughputs of all shards.

You can now horizontally scale by adding more and more shards to distribute load between more storage machines. You can still stay consistent, because for a given train, you only go to a single master where you can utilise the atomicity of RDBMS update.

Segmentation of Journey

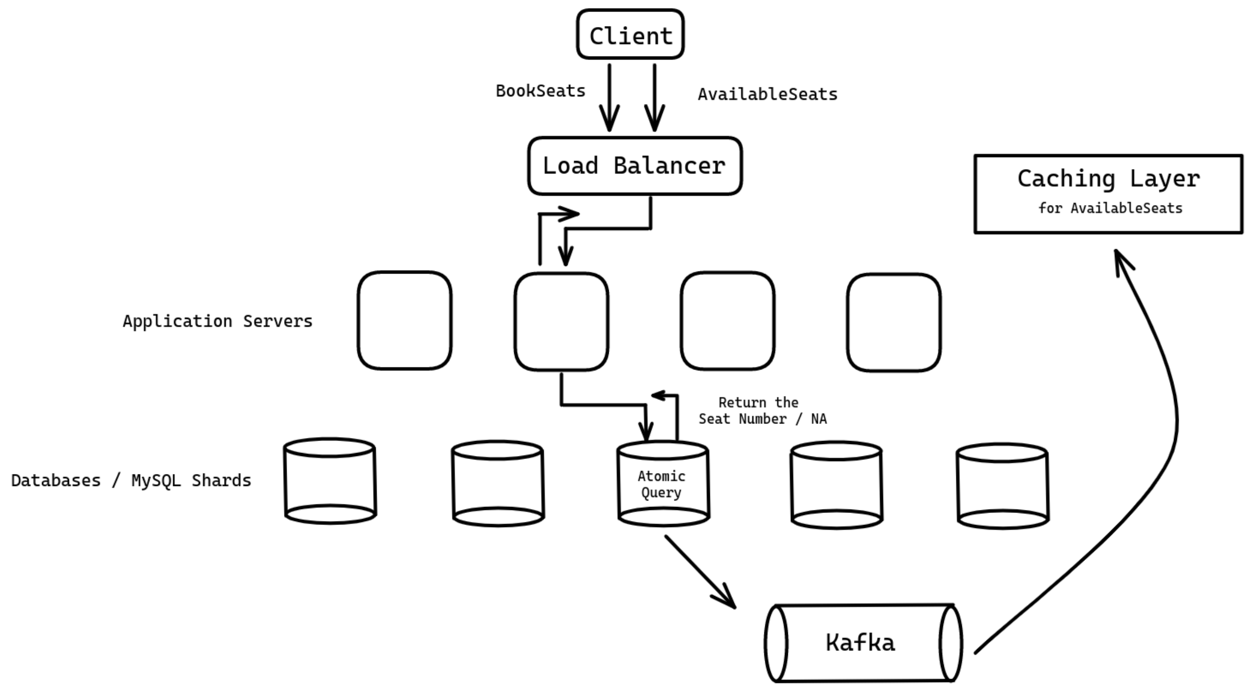
The IRCTC ticket booking system is a little more complex than other ticket booking sites like makemytrip, etc. Here, a single seat can be booked several times during the length of a journey.



- Break down the entire journey into segments (collection of contiguous stations). In this situation, when you book a seat, book it for one or more segments.

Example, when A says to book a seat 8C from Hyd to Delhi. Do the following things:
- Figure out the segments of Hyd and Delhi let's say X and Y respectively.
    - Hyd is the starting point of segment X.
    - Delhi is the ending point of segment Y.
- In that case book 8C for all segments between X to Y (both inclusive).
- Now, the query checks seat availability in all segments from X to Y and then updates the status for all such rows/segments.

Block Diagram



There are two cases regarding how IRCTC can function:

Case 1:

- Fill passenger details
- Complete Payment
- Seat Booking Happens
- If the seat is booked, return the PNR and seat details otherwise refund.

Case 2

- Fill passenger details

- Temporarily block seats for the passenger if available. Change the status of blocked seats to *temporarily_blocked*.
- If no seats, game over.
- Otherwise, ask for payment within 60 seconds(this time can be decided).
- If payment is done within the stipulated time, return the PNR and other details and change status to *permanently_booked*.

We have been designing for **Case 1** till now. For **Case 2**, only one thing changes, every 60 seconds run a **Cron job** making all the *temporarily_blocked* seats *available* again.

CRON JOB

```
UPDATE Bookings
SET available = 1
WHERE
        available = -1 and
        updated_at = NOW() - 60 seconds
```

Assumption: Let **available** be a column with three possible values:

- 1 if seat is free or available
- 0 if it is permanently booked
- -1 if it is temporarily blocked

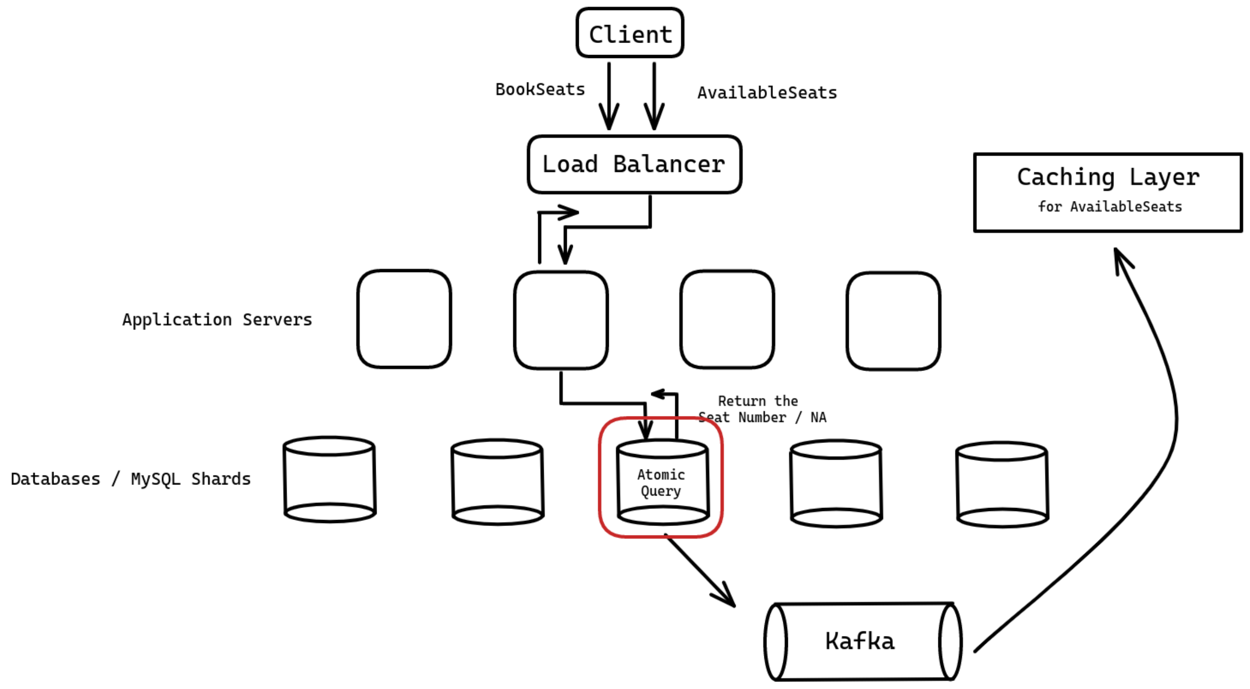
In Case 1, when a user books more than one seat for multiple users:

- The BookSeats API is called multiple times.
- Each time the request goes to the **same shard** because the trainID in all the cases are the same.
    - Also, if somehow magically, all the requests arrive at the same time with difference in nanoseconds, since the shard guarantees atomicity the queries corresponding to these requests will run in parallel independently.
    - This is the property of the database system.
    - If only one seat gets booked out of 5, the remaining seats can get a refund.
- So, the booking of seats happens one by one once the payment successful message is received.
- Till the second step in Case 1 (Complete Payment), the details of the passengers are not stored in the databases.
- Once the payment is completed, the details are fetched from the client and booking is done.

System 2 Design - Seat Availability System

- Each time a query comes to find the number of available seats, the actual answer is stored in the database in the shard as shown below.

- But if each time we query the database, it will become both a read and write heavy system. In earlier classes, you have learnt it is impossible to design a database that is both read and write heavy.
- To avoid this scenario, you can build a cache on top of it. If this cache needs to be highly consistent, you can build a write-through cache. You can build a bunch of caching machines which cache data from the database.
    - Whenever there is an update, they can propagate upwards.
- Another solution is to use a master-slave system. The read requests will be directed towards the slaves. However, if you want the AvailableSeats API to be very fast, it would be better to use cache.
- Regarding the buffer time, it is not there. You are trying to propagate the updates as soon as possible. The kafka would have some delay depending on the throughput of kafka.



Generic Question

- You need to be one driving the discussion. Nobody should be poking you giving you scenarios to think of. Because in that case it means you are doing the bare minimum and walking off.
- How deep the discussion should be depends on the interviewer. It is your responsibility to probe the interviewer regarding this and get the job done. It is because we can go as deep as possible like we can go till the code and start discussing the functions.
- There are types of HLD interviews as well. Most of the MAANG and US-based startups prioritize discussion on the high-level components and not the solutioning. The Indian-based startups however may prioritize the solutioning part.

- Solutioning means talking about the specifics and configurations part rather than the problem solving part.
- In the above solution, the Caching layer is a black box. Questions regarding the caching part can be these:
    - Do you have one or multiple machines?
    - If multiple, are you splitting the information among these machines? If yes, what's the logic behind that?
    - The storage required on these machines?
    - If you want to access some information, do you go to any of these machines or some kind of algorithm there?