ITNEXT · Follow publication

Member-only story Featured

Solving Double Booking at Scale: System Design Patterns from Top Tech Companies

Learn how Airbnb, Ticketmaster, and booking platforms handle millions of concurrent reservations without conflicts

10 min read · Oct 8, 2025

Animesh Gaitonde Follow

Listen Share ••• More

Gone are the days when people used to stand in long queues to get tickets for concerts, flights, movies, matches and other events.

You can read the free version of the article here.

Tech companies like Ticketmaster, BookMyShow, Airbnb, Delta Airlines, etc have made reservations a one-click experience letting you book tickets from your home.

This simplicity comes from the tech platforms and services that hide and solve complex engineering problems behind the scenes. One such problem is preventing two or more users from booking the same seat.

Imagine the plight of two users who get assigned same seat for an event and realizing this just before the event's start. It leads to loss of customer trust and the users would think twice before booking their next event.

Hence, it's important to build a robust solution to solve the classic — *Double Booking Problem*.

In this article, we will learn how different tech companies solve this problem. Each company has a different use case and there's no one size fits all solution that solves this problem.

We will go over the different architectural patterns and understand their trade-offs. The article will help you gain depth and develop expertise in systems thinking.

With that, let's begin.

How does Double Booking occur?

Before diving deep into the solution, we will first understand how one seat can be booked by more than two users simultaneously.

Reservation System Architecture

Let's consider a simple reservation system that consists of:

- **Client** — Mobile App or Web Page that reserves a seat.
- **Booking Service** — Backend service that exposes APIs for seat reservation.
- **Database** — Relational database that manages the state of the seat.

Booking Service reserves a ticket through the following two steps:

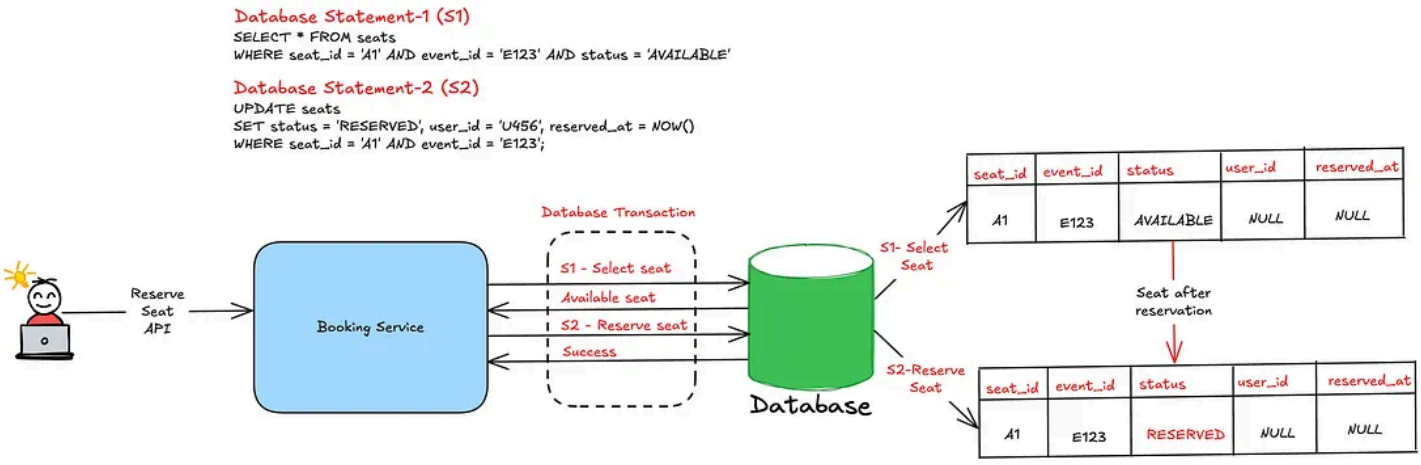
1. **Availability check** — It executes a `select` query to check for the seat's availability. (*S1*)
2. **Status update** — If the availability check succeeds, then it runs `update` to mark the seat's status as reserved. (*S2*)

```sql
BEGIN TRANSACTION;

# S1
SELECT * FROM seats
WHERE seat_id = 'A1' AND event_id = 'E123' AND status = 'AVAILABLE'

-- If row returned, update it

# S2
UPDATE seats
SET status = 'RESERVED', user_id = 'U456', reserved_at = NOW()
WHERE seat_id = 'A1' AND event_id = 'E123';
COMMIT;
```

SQL queries.sql hosted with ♥ by GitHub view raw

S1 and S2 SQL queries

Both *S1* and *S2* are executed within a database transaction. The below diagram illustrates the working along with the data model and SQL queries.



Reservation system architecture

Now that you understand the basic flow, let's see what would happen if two users decide to book a ticket simultaneously.

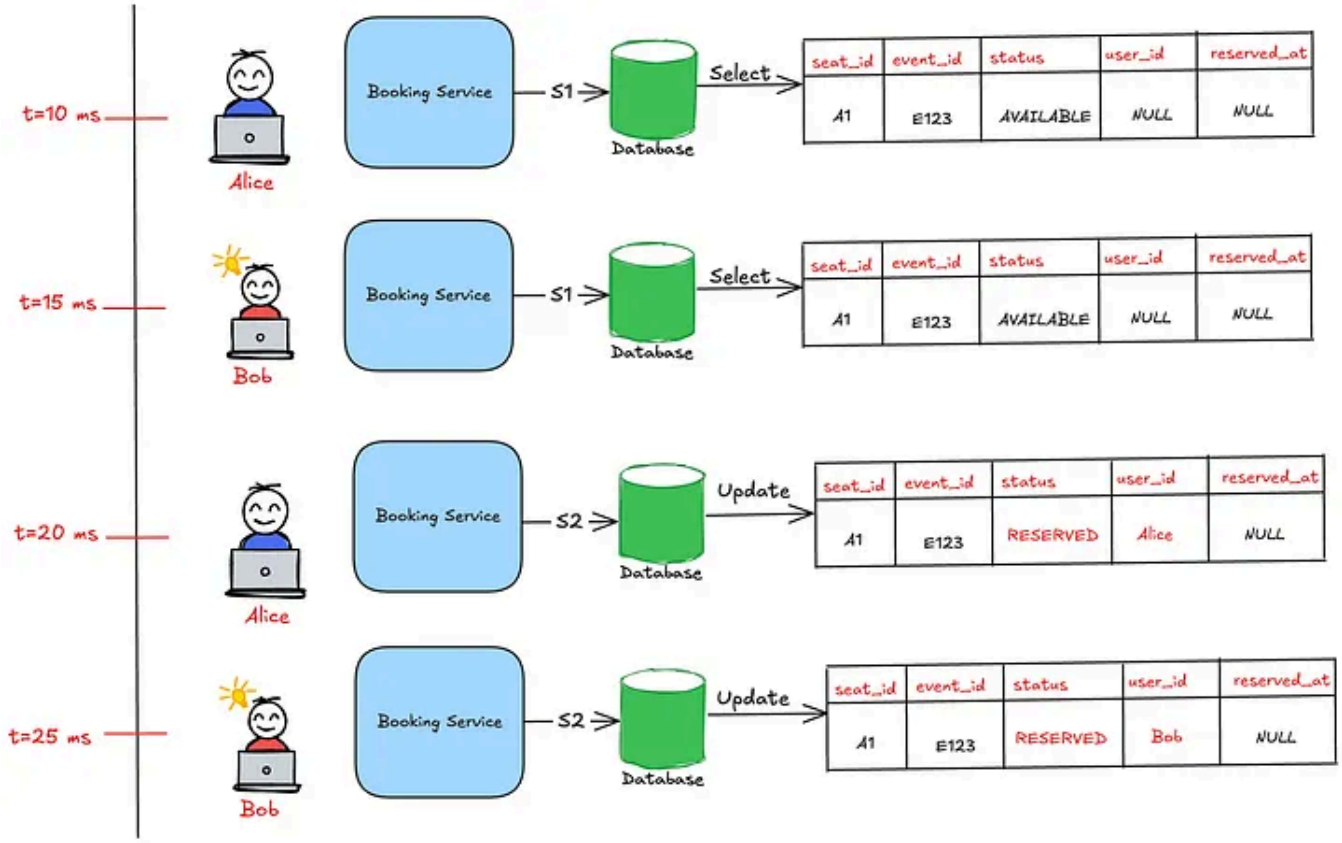
Double Booking Explained

Assume that Alice and Bob are two users who placed a request at time `t=0 sec`. Here's how the Booking Service would process the two requests concurrently:

1. **T=10 ms** — Alice's *S1* will succeed and find the seat available.
2. **T=15 ms** — Similarly, Bob's *S1* will find the seat available.

3. **T=20 ms** — Alice's S2 will update the ticket's state and assign it to Alice.
4. **T=25 ms** — Bob's S2 will overwrite Alice's ticket assignment.

Let's go through the visual representation to understand double booking.



Double booking in action

In the end, the service will give a successful response to both Alice and Bob. Both would think that the ticket is assigned to them. However, in the database, the ticket would be assigned to **Bob**.

If you have taken classes on multi-threading in Computer Science, the above example will remind you of classic case of race condition.

> ***Food for thought**: Do you think the same problem would occur if Alice's S1, S2 are executed first, followed by Bob's S1 and S2?*

Here's why the double booking problem occurs due to:

- **Shared resource** — Two or more services/threads compete for the same shared resource i.e ticket in this case that leads to race condition.

- **Non-atomic updates** — The overall process is split into two operations (*Select* and *Update*) that don't guarantee atomicity.

Let's now explore solutions to tackle the above problems.

Pessimistic Locking

We prevent race conditions in a system through locking on shared data structures. Locks guarantee mutual exclusion and allow only a single thread to update the data structure.

While locking works on in-memory data structures, does it work for persistent stores like database? The answer is **Yes.**

Databases like *PostgreSQL, MySQL, SQL Server,* etc provide constructs to acquire a lock on a database record. The lock ensures that only a single transaction can modify the record. The lock is released when the transaction is committed, and other transactions can then acquire a lock.

This approach is known as **Pessimistic locking** and often used to solve the double booking problem. Let's now apply this concept and understand its working for our use case.

We will now modify the query *S1* and append `FOR UPDATE` to explicitly acquire a lock on the ticket. Here's the modified version of *S1* and *S2* in the transaction.

```
BEGIN TRANSACTION;

# S1
SELECT * FROM seats
WHERE seat_id = 'A1' AND event_id = 'E123' AND status = 'AVAILABLE'
FOR UPDATE

-- If row returned, update it

# S2
UPDATE seats
SET status = 'RESERVED', user_id = 'U456', reserved_at = NOW()
WHERE seat_id = 'A1' AND event_id = 'E123';
COMMIT;
```

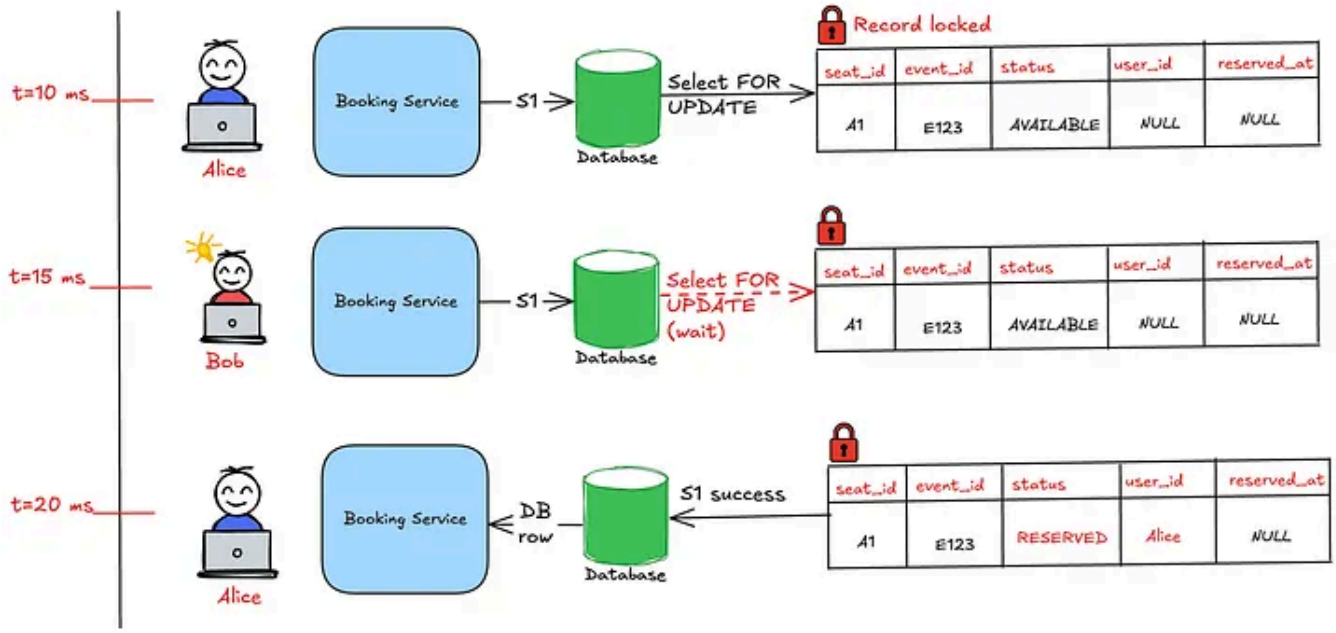
PessimisticConccurency.sql hosted with ♥ by GitHub view raw

Acquiring a lock on database row

Let's go through Alice and Bob's example and understand how the above approach solves the double booking problem. Here's how the Booking Service would now process the request:

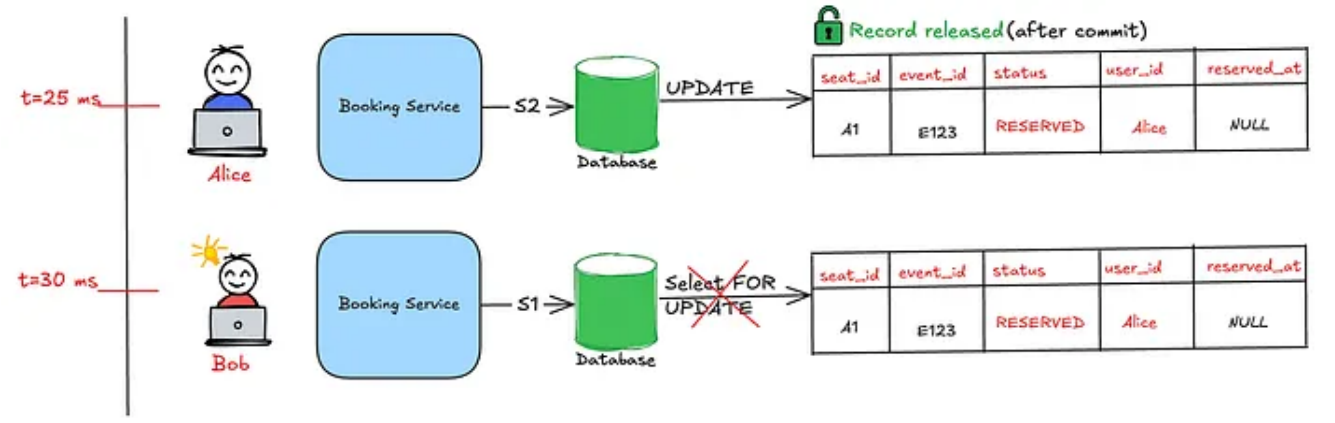
1. **T=10 ms** — Alice's *S1* will execute and **lock** the database row.
2. **T=15 ms** — Bob's *S1* will find the seat locked and wait for it.
3. **T=20 ms** — Alice's *S1* will succeed and find the available seat.

The following diagram illustrates the sequence of events.



Locking of the seat

The following diagram explains how the seat is booked followed by release of the lock.



Successful booking of Alice's seat

Once the seat is locked, Alice's *S2* will assign the ticket to Alice, update the state and release the **lock**. Further, Bob's *S1* will continue execution but find the ticket reserved.

The approach prevents double booking and guarantees consistency. It has the following advantages:

- *Simplicity* — Easy to reason about and implement.
- *Consistency* —Eliminates race condition.
- *High contention scenarios* — Suitable for use cases where many users compete for handful resources.

> ***Food for thought**: What would happen if S1 executes, but the database connection disconnects? Would the lock remain or get released?*

Do you see any disadvantages of this approach? Think for a moment before reading further.

While the above approach works for simple use cases, it struggles for:

- *High throughput use cases* — Lock becomes a bottleneck, slows the execution reducing the responsiveness.
- *Deadlock risk* — Chances of deadlock increase with more competing resources.
- *Scaling challenges* — Not suitable for popular events like concerts that have high traffic.

In the real-world, pessimistic locking finds applications in low-traffic use cases such as airline seat reservations during web-checkins.

Is there an alternative that addresses the limitations of pessimistic locking? Let's understand in the next section how we can improve the responsiveness of the system.

Optimistic Locking

Optimistic locking avoids locks and instead maintains a version attribute for every database record. Here's how the approach works:

1. **Database read** — Read the database record along with its version.

2. **Database update** — Adds a `where` clause in the `update` query to ensure the version doesn't change before the update.

3. **Version increment** — Increment the version every time the record is updated.

With this approach, transactions no longer need to wait on the locks. Also, it doesn't have to deal with deadlock related complexity.

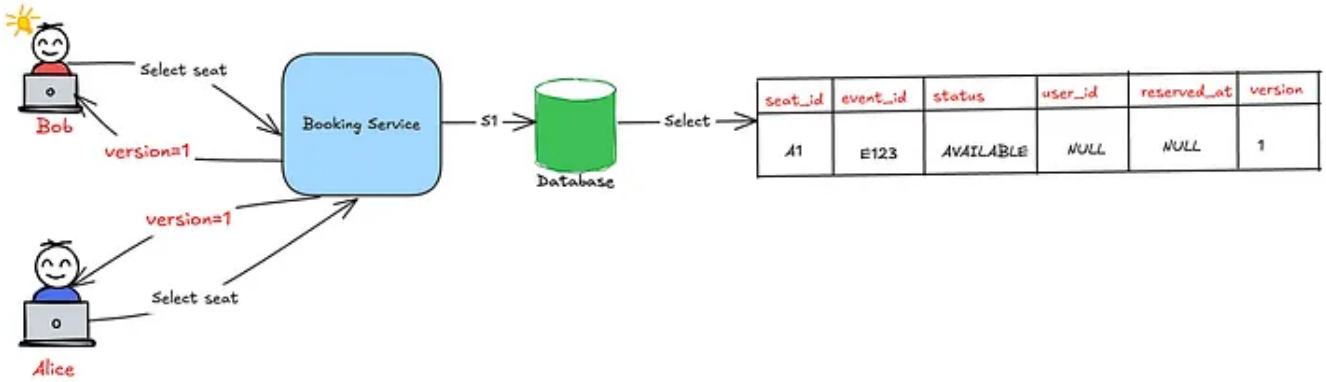
Here's the updated query for this approach:

```sql
-- Read
SELECT seat_id, status, version FROM seats
WHERE seat_id = 'A1' AND event_id = 'E123';

-- Attempt update
UPDATE seats
SET status = 'RESERVED', user_id = 'U456',
   reserved_at = NOW(), version = version + 1
WHERE seat_id = 'A1' AND event_id = 'E123'
  AND status = 'AVAILABLE'
  AND version = 42;

-- Check affected rows, retry if 0
```

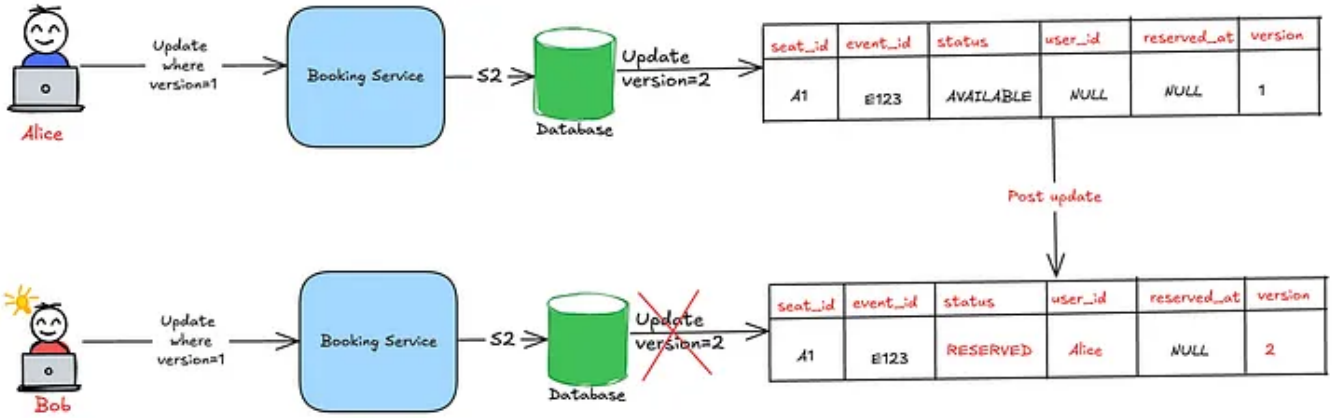
OptimisticConcurrency.sql hosted with ♥ by GitHub view raw

Optimistic locking

The following diagrams depicts the working of Optimistic locking.



Selecting the seat and reading the version



How the second update fails due to version mismatch

This technique offers the following advantages:

- *Improved throughput* — Without explicit locking, more queries can execute concurrently improving the overall throughput.
- *Scalability* — It can scale to handle moderate traffic and less popular events.
- *Improved read performance* —Unlike pessimistic locking, reads are performant.

The approach is suitable for traffic patterns that are steady and have low contention. Few examples are restaurant booking, hotel booking via Booking.com or Airbnb.

What if we use this approach for a popular event like a weekend blockbuster movie? For such events, there's a high chance that multiple people would try to book the same ticket in a time window.

While the approach would ensure only one person gets the ticket, the request for others would fail. Hence, others would have to retry booking their seat.

Optimistic locking has the following shortcomings:

- *Poor user experience* —Popular events may lead to conflicts while reservation leading to user retries and degraded user experience.
- *Application complexity* — Applications need to handle the version conflict errors gracefully.
- *Redundant compute* — High contention can lead to increased load on the database resulting in compute wastage.

> ***Food for thought**: Instead of checking the version, can the query rely solely on the `status` while updating the record's state? (Leave your thoughts in the comments below)*

So far, we have learnt techniques to solve the double booking problem in simple low-volume systems. They can't scale to handle high-traffic scenarios such as popular movie shows or sport events.

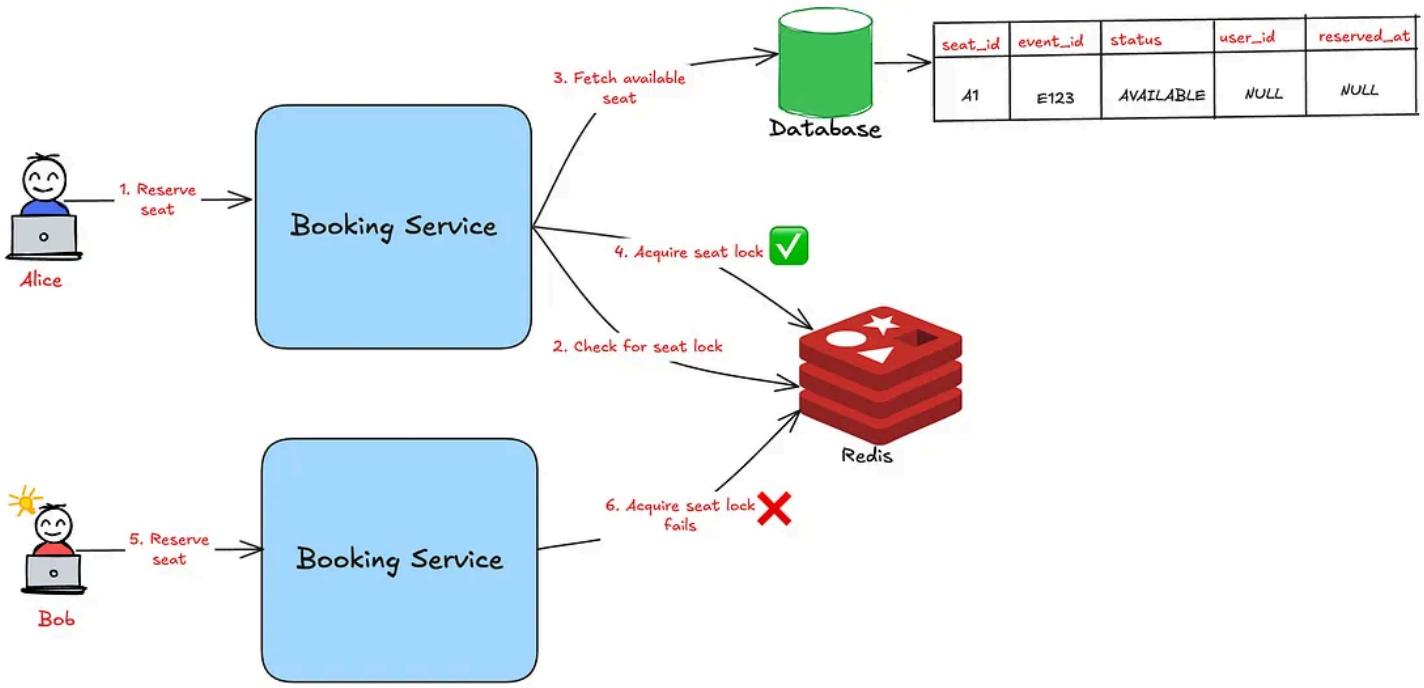
Let's explore in the following section how to solve double booking problem for high-traffic events.

In-memory Distributed Locking

We can reduce the database load by using in-memory cache for distributed lock. Applications can acquire a seat lock before seat reservation.

All the requests would first check for the presence of lock in the cache. The request acquiring the lock would update the database and then release the lock.

The following diagram illustrates how in-memory distributed locking prevents the double booking problem.



Double booking prevented by In-Memory distributed locking

Here's how the approach addresses the challenges of previous two solutions:

- *High Performance* — It is performant since in-memory operations are fast and database load is reduced.

- *High Concurrency* — It can handle large number of concurrent bookings as database is no longer the bottleneck.

While this solution scales, an additional in-memory cache does introduce complexity. We now need to address cases such as:

- *Data loss* — Cache crashing and losing all the data
- *Cache unavailability* — Unavailability of the cache can result in a database load spike.
- *Unreleased locks* — When a lock isn't released, it blocks others from booking the available seat.

These downsides need to be handled to guarantee system correctness. For eg:- Cache replication can prevent unavailability. Locks can be set to expire to avoid indefinite locking.

> ***Food for thought****: What if the cache crashes just after a seat lock is acquired? Would another request override the seat booking leading to double booking? (Leave your thoughts in the comments)*

The solution is suitable for use cases that have high contention such as popular sport matches and movie shows.It can handle the **1K-10K** requests/sec traffic seamlessly.

But, would it scale for very popular events like Coldplay concert? In such concerts, more than **100K** people try to book a seat at the same time.

Any failure like cache crashes can't be tolerated in such cases. We will see in the next section how to tackle this challenge.

Virtual Waiting Queue

Booking popular concert tickets like Coldplay is challenging for users. Often, users end up not getting a seat due to the demand.

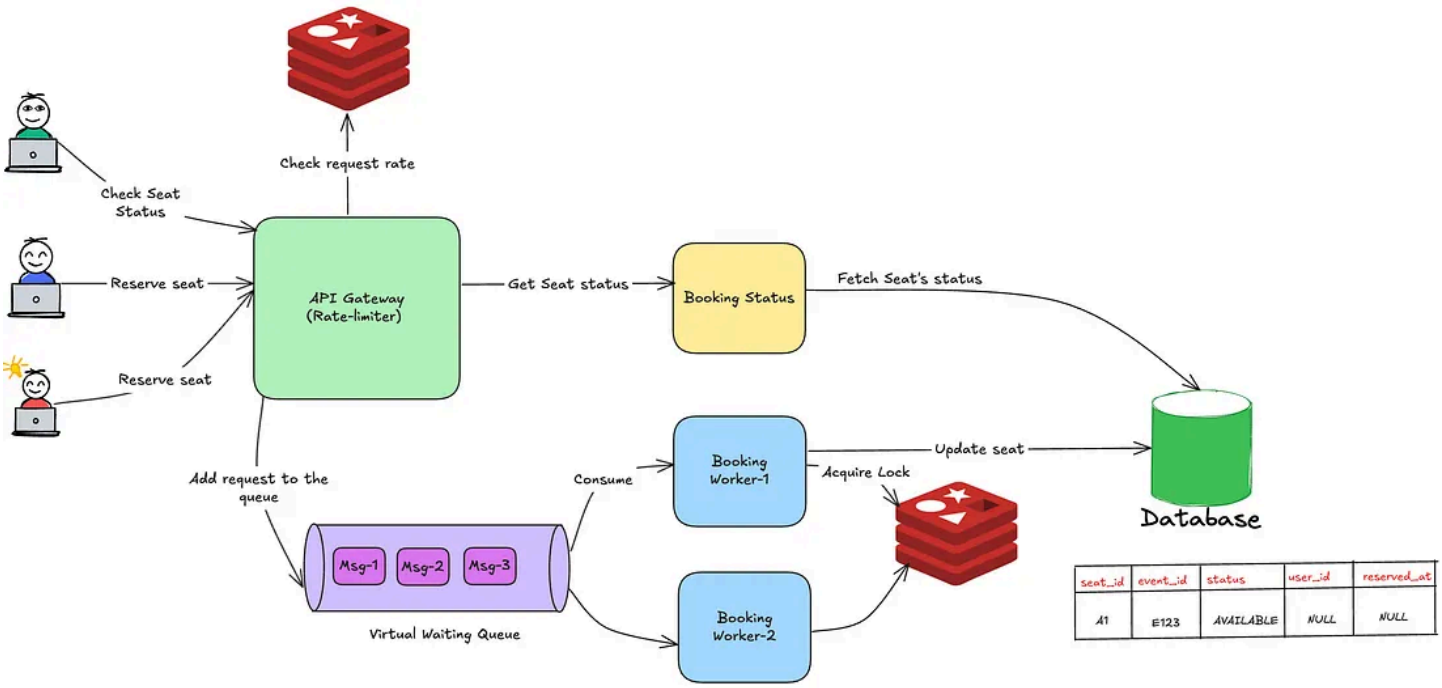
However, the tech systems ensure robustness and provide the best user experience. They tackle the high demand by introducing a virtual queue for the users booking the ticket.

The virtual queue makes the booking process asynchronous. It acts as a buffer and prevents the system from getting overwhelmed with requests.

Here's how the process works:-

1. The system detects surge in the traffic and directs the requests to a waiting queue.
2. The requests in the waiting queue are processed asynchronously by the application and the seats are booked gradually.
3. The users can check the status of the seats and can't book another seat until they are in the waiting queue.
4. Once the seat is booked, the user would be notified (through Server-Sent Events or any other mechanism).

The following diagram shows the architecture and the working of the system.



Seat reservation using Virtual Waiting Queue

The approach offers the following advantages:

- *Scalability* — It protects the database, cache and other components from becoming a bottleneck. The async approach reduces the load and improves the scalability.

- *User experience* — Users no longer see multiple error messages while booking seats. This greatly improves the user experience.
- *Fairness* — The FIFO queue ensures that the system prioritises the users who entered the queue first.

> ***Food for thought**: At what RPS, should one pivot to a Virtual Waiting Queue based system? (10K RPS, 50 KRPS or 100K RPS?)*

While the approach scales and improves the user experience, it does so at the cost of:

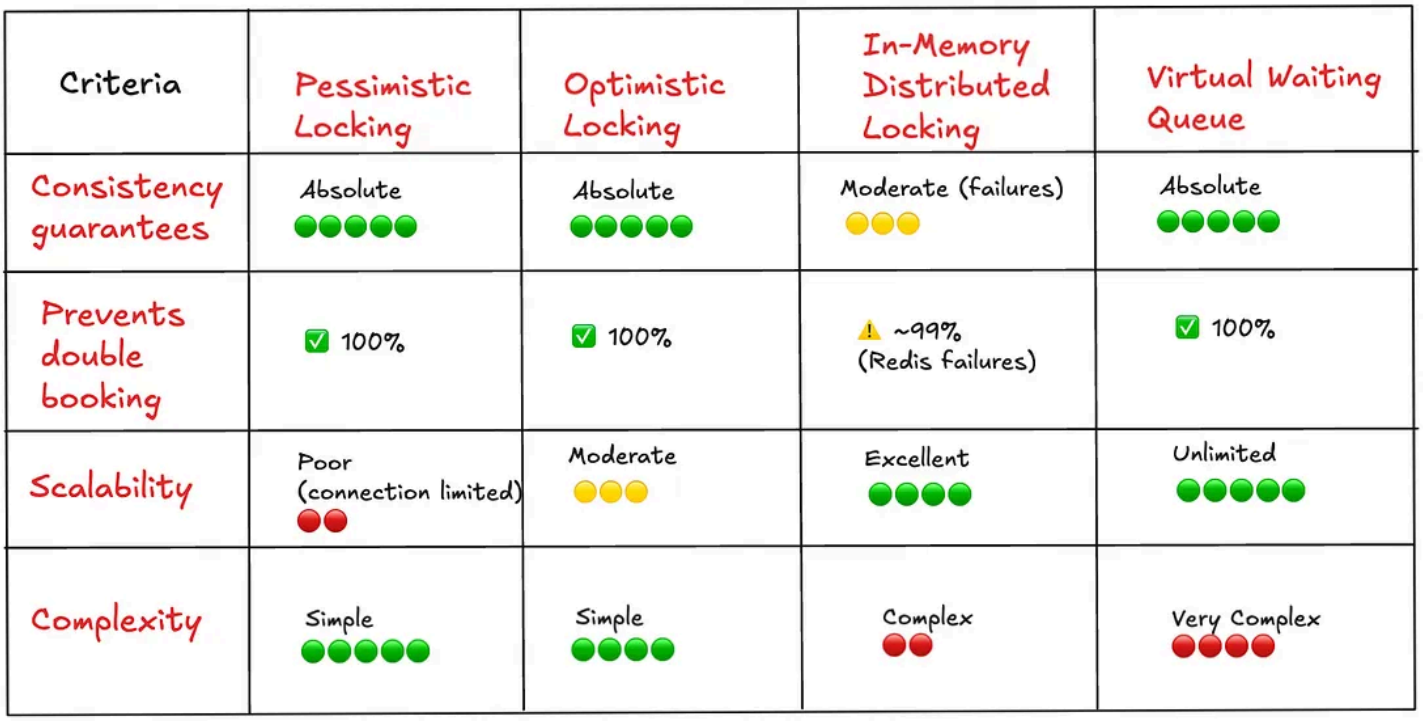
- *Complexity* — Developers have to deal with the queueing layer, manage and operate it. Similarly, the system must deliver real-time updates via SSE(Server-Sent Events). This adds to the infrastructure complexity and increases the costs.

By now, you would have understood the different techniques used to solve the double booking problem. Let's now summarize what we have learnt so far.

Conclusion

All reservation systems such as TicketMaster, Airbnb, BookMyShow, etc face a challenge to prevent double booking. Accidental double bookings break customer trust and hence it becomes a critical business problem.

In this article, we discussed the different ways to tackle the double booking problem. The following table summaries the pros/cons of the different approaches that we discussed.



Pros/Cons of Double Booking Solutions

Every reservation use case is different and has its own constraints. There's no universal solution that solves for all the use cases.

Do you think we can combine the different solutions and have a single system that caters to use cases from reserving a flight seat to booking a flash concert ticket? If yes, what technical challenges do you foresee in building it?

Before you go:

- 👏 for the story and follow me for more such articles
- Subscribe to my free engineering newsletter here
- 🔔 Follow me: LinkedIn, Twitter, Medium

Software Development | Software Engineering | System Design Interview | Distributed Systems | Programming