IRCTC TATKAL BOOKING: THE SYSTEM DESIGN

How do you sell a train's last few seats to a huge crowd that arrives in the same second, without ever selling one seat twice or charging anyone twice?

SIMPLIFIED EXAMPLE

This is a teaching design. It is not IRCTC's actual architecture, and it does not claim to describe how IRCTC built its system. It contains no real traffic or revenue figures. Every technique here is a standard one that you can use in an interview or in your own project.

WHAT IS INSIDE

1. The situation
2. Requirements
3. Why the simple design breaks
4. The full picture
5. Idea 1: waiting room
6. Idea 2: cache for seat counts
7. Idea 3: sell a seat in segments
8. Idea 4: lock, plus a rule the database enforces
9. Idea 5: seat hold during payment
10. Idea 6: idempotency key
11. One booking, start to finish
12. What this design leaves out
13. Answer: who gets a cancelled seat?

1. THE SITUATION

Tatkal tickets open at a fixed time in the morning, one day before the journey. Very few seats are on offer, and everybody who wants one arrives at the same moment. For a few minutes the system sees its heaviest load of the day, aimed at a handful of popular trains.

Two things make this harder than an ordinary flash sale:

- **A seat is not one item.** A berth can carry one passenger from Delhi to Kanpur and a different passenger from Kanpur to Howrah. The thing being sold is a seat for a stretch of the route.
- **Money is involved.** Payment takes time and can fail, and people tap "Pay" twice when a screen freezes.

2. REQUIREMENTS

What users must be able to do

- Search trains between two stations on a date.
- See how many seats are available for that journey.
- Book seats for one or more passengers and pay.
- Cancel a booking.

What the system must guarantee

Guarantee	What it means
No double booking	Two passengers never hold the same seat on the same stretch. This must hold even if the application code has a bug.
No double charge	A retry or a second tap never takes money twice or creates two bookings.
Survive the opening rush	The system slows down gracefully. It does not fall over.
Fair order	People are served roughly in the order they arrived.
Seat counts may be slightly old	The number on the search page can lag by a few seconds. The booking step itself is always exact.

3. WHY THE SIMPLE DESIGN BREAKS

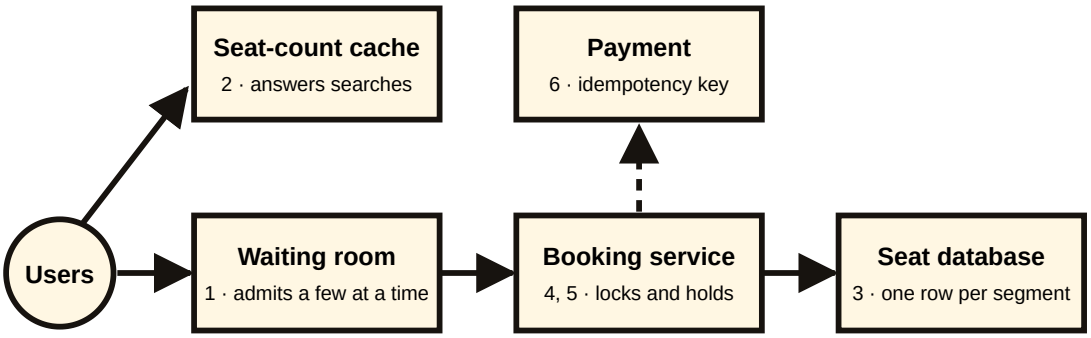
Start with one application server and one database. A booking reads "is seat 14 free?", then writes "seat 14 is taken".

- **The server falls over.** Every request opens a database connection. When the crowd arrives together, connections run out, requests pile up, and everything times out, including for people who only wanted to search.
- **A seat is sold twice.** Two requests read "seat 14 is free" at the same instant. Both then write. This is a race condition, and it happens precisely when load is highest.
- **Money goes missing.** A payment succeeds at the bank but the reply is lost. The user taps again and is charged a second time.

Each idea below removes one of these failures.

4. THE FULL PICTURE

A request travels left to right. The numbers match the ideas that follow.



Searches read the cache. Bookings go through the waiting room, then the booking service, which is the only part that writes to the seat database and refreshes the cache. Dashed arrow: the booking service calls payment only after the seat is held.

5. IDEA 1: WAITING ROOM

Problem it solves: the crowd arriving in one second.

Do not let everyone reach the booking service. Put a waiting room in front of it. Each user who arrives gets a place in a queue and a token. The waiting room admits people at a steady rate that the booking service is known to handle, in arrival order, and shows everyone else their position.

- The queue lives in a fast in-memory store, not in the booking database.
- An admitted user gets a signed token that expires after a short time. The booking service rejects any request without a valid token, so nobody can skip the line by calling the service directly.
- The admission rate is a dial. If the database slows down, turn it down.

It is the line at a ration shop: the shop serves at its own pace, however long the line outside.

> **Trade-off:** people wait, and some wait only to find the seats gone. In return the system stays up and the order is fair.

6. IDEA 2: CACHE FOR SEAT COUNTS

Problem it solves: searches drowning the database.

Far more people look at availability than book. Counting free seats for every page view would keep the database busy with reads at the worst moment. Keep the counts in a cache, keyed by train run, class and journey, and answer searches from there.

- The booking service updates or clears the cached count whenever a seat is held, booked or released.

- Each entry also expires after a few seconds, so a missed update corrects itself.
- The cache is never used to decide a booking. That decision is made in the database (Idea 4).

> **Trade-off:** the count can be a few seconds old. A user may see "2 available" and then fail to book. That is acceptable; selling a seat twice is not.

7. IDEA 3: SELL A SEAT IN SEGMENTS

Problem it solves: a seat that is empty for part of the route is wasted.

Cut each train's route into segments, one between each pair of neighbouring stops. A train that stops at Delhi, Kanpur, Prayagraj and Howrah has three segments. Store one row for every seat on every segment of every dated run of the train.

```
train_stop    (train_id, stop_seq, station_id, arrive, depart, day_offset)
              -- the fixed timetable; stop_seq gives direction and order
train_run     (run_id, train_id, start_date)
              -- one row for each date the train runs
seat          (seat_id, train_id, coach, berth_no, class)
seat_segment  (run_id, seat_id, segment_seq, booking_id, hold_expires_at)
              -- PRIMARY KEY (run_id, seat_id, segment_seq)
              -- segment_seq 1 = stop 1 to stop 2, and so on
              -- created empty for the whole train when the run opens
```

THE RULE

A seat is free for your journey only if it is free on every segment of that journey. A journey from stop `a` to stop `b` uses segments `a` to `b - 1`.

A common mistake is to check each station separately and say "fewer seats are taken than the train has, so there is room". Take a train with two seats:

Seat	Delhi to Kanpur	Kanpur to Prayagraj
Seat 1	taken	free
Seat 2	free	taken

On each segment only one of two seats is taken, yet nobody can travel from Delhi to Prayagraj, because no single seat is free the whole way. Count seats that are free on all segments, not free seats per segment.

```
-- seats free for the whole journey (stop :a to stop :b)
SELECT seat_id
FROM   seat_segment
WHERE  run_id = :run
AND    segment_seq BETWEEN :a AND :b - 1
AND    (booking_id IS NULL OR hold_expires_at < now())
GROUP  BY seat_id
HAVING count(*) = :b - :a;
```

> **Trade-off:** more rows and more work per booking: a long journey touches many rows. In return a berth can be sold several times along the route.

8. IDEA 4: LOCK, PLUS A RULE THE DATABASE ENFORCES

Problem it solves: two requests taking the same seat at the same instant.

The booking service picks a candidate seat, then takes it inside one short database transaction:

```
BEGIN;

-- 1. lock this seat's rows for the journey, always in station order
SELECT 1 FROM seat_segment
WHERE  run_id = :run AND seat_id = :seat
AND    segment_seq BETWEEN :a AND :b - 1
ORDER  BY segment_seq
FOR UPDATE;

-- 2. take the rows only if they are still free
UPDATE seat_segment
SET    booking_id = :booking, hold_expires_at = now() + :hold_time
WHERE  run_id = :run AND seat_id = :seat
AND    segment_seq BETWEEN :a AND :b - 1
AND    (booking_id IS NULL OR hold_expires_at < now());

-- 3. the number of rows updated must equal (b - a)
--    if it does: COMMIT.  If not: ROLLBACK and try the next free seat.
```

- **Why lock in station order?** If one request locks Delhi then Kanpur while another locks Kanpur then Delhi, each waits for the other for ever (a deadlock). When everyone locks in the same order, that cannot happen.
- **The rule the database enforces.** The primary key allows exactly one row per seat, per segment, per run, and that row holds one booking. Step 2 writes only into an empty row. So even a buggy caller cannot put two passengers on one seat segment; the database refuses.
- **Keep the transaction short.** It lasts milliseconds. It is never held open while the user pays; that is the job of Idea 5.

> **Trade-off:** when two people reach for the same seat, one waits, loses and has to try the next seat. Under heavy demand that means retries. A refinement is to let each request skip rows that are already locked (`FOR UPDATE SKIP LOCKED`), so people are spread across different free seats.

9. IDEA 5: SEAT HOLD DURING PAYMENT

Problem it solves: payment takes time, and it can fail.

Step 2 above does not confirm the booking. It places a **hold**: the seat's rows point to the booking, with an expiry time a few minutes ahead.

What happens	What the system does
Payment succeeds	In one transaction, mark the booking `CONFIRMED` and clear `hold_expires_at` on its rows. The seat is now sold.
Payment fails	Clear `booking_id` on the rows. The seat is free again at once.
User walks away	Nothing needs to run. Once the expiry passes, every query above already treats the rows as free. A background job tidies up later.
Payment succeeds after the hold expired	Try to take the seat again. If it has gone, refund automatically. This case must be designed, not ignored.

> **Trade-off:** seats that are only held look sold out, then reappear when a payment fails. That is why a "sold out" train sometimes shows a seat a few minutes later.

10. IDEA 6: IDEMPOTENCY KEY

Problem it solves: double taps, retries and lost replies charging twice.

An operation is idempotent when doing it twice has the same effect as doing it once. When the payment screen opens, the app creates one unique ID for this payment attempt and sends it with every request for that attempt.

```
payment (payment_id, booking_id, idempotency_key UNIQUE, status, amount)

-- first request with this key: the INSERT succeeds, so go and charge
-- any later request with the same key: the INSERT hits the UNIQUE rule,
--   so do not charge; return the result of the first request
```

- The same key is passed on to the payment gateway, so a retry from our own server is also safe.
- The key belongs to the attempt, not the user. A fresh, deliberate attempt gets a new key.

> **Trade-off:** one more lookup on every payment request, and the keys must be stored for as long as a retry could arrive.

11. ONE BOOKING, START TO FINISH

1. The user searches. The seat count comes from the cache.
2. The user taps "Book" and joins the waiting room. When admitted, they receive a short-lived token.
3. The booking service finds a seat that is free on every segment of the journey.
4. It locks that seat's rows in station order and places a hold with an expiry.
5. The user pays. The request carries an idempotency key.
6. On success the hold becomes a confirmed booking. On failure or expiry the seat returns to the pool.
7. The cached seat count is updated.

12. WHAT THIS DESIGN LEAVES OUT

- **Quotas.** General, Tatkal, ladies, senior citizen and other quotas are separate pools of seats. Add a quota column to the seat inventory and filter on it.
- **Coach and berth preference,** and keeping a family together in one coach.
- **RAC** (two passengers sharing one berth until a full berth frees up) and chart preparation before departure.
- **Fares, refunds and cancellation charges.**
- **Bots and agents.** Captchas, rate limits per account and device checks sit in front of the waiting room.
- **Splitting the database.** Seat rows for different train runs never interact, so they can be divided across databases by run.
- **Search itself.** Finding trains between two stations uses `train_stop`: same train, source `stop_seq` less than destination `stop_seq`. Ordering by `stop_seq` rather than by clock time also keeps overnight trains correct.

13. ANSWER: WHO GETS A CANCELLED SEAT?

The video asked: when someone cancels, how does a waiting-list passenger get that seat?

Keep a waiting list for each train run and class, in the order people joined, each entry with its own journey (from stop, to stop).

```
waitlist (run_id, class, position, booking_id, from_seq, to_seq)
```

1. **Free the seat.** In one transaction, mark the booking cancelled and clear its rows in `seat_segment`.
2. **Hand the work to one worker per train run.** Put a "seat freed" message on a queue. One worker at a time handles a given run, so two promotions never race.
3. **Walk the waiting list in order.** For each waiting passenger, ask the usual question: is some seat now free on every segment of their journey? Give the seat to the first passenger for whom the answer is yes, using the same lock and update as Idea 4.
4. **Keep going.** A freed Delhi to Howrah seat might serve one waiting passenger from Delhi to Kanpur and another from Kanpur to Howrah. Continue down the list until nothing more fits.
5. **Tell the passenger,** and update the cached count.

> **THE SUBTLE PART**
> The first person in the list does not always get the seat. If the freed stretch is Delhi to Kanpur and the first person is waiting for Delhi to Howrah, the seat does not fit their journey. It goes to the first person whose whole journey it covers, and the first person keeps their place.