## Audience

This article is the next in my series of how I would design popular applications. It is recommended (although not entirely necessary) to read the previous posts I've helpfully compiled in a list here. We will expect a basic familiarity with architecture principles and AWS, but hopefully this post is approachable for most engineers.

## Argument

Initially, let's look at our problem statement.

### The System to Design

We would like to design a system similar to TicketMaster. For those of you unfamiliar, this is a system for booking tickets to live events. This might be gigs, the theatre, comedy, anything within that general umbrella.

The exact requirements of our system is that it should support multiple cities, and when a user selects a city they should see the available venues in that location. When a user selects a venue they can see the performances and dates for that venue. They can select a performance, date and a number of tickets to add to their cart.

The user can then use a third party supplier to pay for their tickets, after which they will receive an email and phone notification containing their receipt.

Our usual non-functional requirements stand. It must be reliable, scalable and available.

### The Approach

We have a standard approach to system design which is explained more thoroughly in the article here. However the steps are summarised below:

1. **Requirements clarification:** Making sure we have all the information before starting. This may include how many requests or users we are expecting.
2. **Back of the envelope estimation:** Doing some quick calculations to gauge the necessary system performance. For example, how much storage or bandwidth do we need?
3. **System interface design:** What will our system look like from the outside, how will people interact with it? Generally this is the API contract.
4. **Data model design:** What our data will look like when we store it. At this point we could be thinking about relational vs non-relational models.
5. **Logical design:** Fitting it together in a rough system! At this point I'm thinking at a level of 'how would I explain my idea to someone who knows nothing about tech?'

6. **Physical design:** Now we start worrying about servers, programming languages and the implementation details. We can superimpose these on top of the logical design.
7. **Identify and resolve bottlenecks:** At this stage we will have a working system! We now refine the design.

With that said, let's get stuck in!

### Requirements Clarification

The questions I would be asking include how many cities are we expecting, how many venues per city, how many performances per venue per year, and the capacity of each venue!

### Back of the envelope estimation

Let's say we have 100 cities, each with 10 venues, each holding 10 performances a day, with an average of 1000 tickets per venue. Let's also assume everything sells out (hopefully). This gives us:

```
100 cities * 10 venues * 10 performance * 1000 tickets = 10,000,000 tickets transactions a day!
```

This corresponds to around 15 sales per second (spread evenly).

If we estimate a city/ venue/ performance/ ticket storage to take up roughly 50B per row, then our total (approximate) storage requirements are:

1. `100 * 50 = 5000B = 5KB` for cities
2. `100 * 10 * 50 = 50,000B = 50KB` for venues
3. `100 * 10 * 50 * 10 = 500,000B = 500KB` for performances
4. `10,000,000 * 50 = 5,000,000,000B = 500MB` for tickets

So the total becomes:
`500MB + 500KB + 50KB + 5KB = 500.555MB (for the first day)`

If we save the ticket data each day then we will increase by 500MB a day, leaving us needing 182.5GB of storage by the end of the year.

To estimate traffic we would need the approximate number of times a person would access pages/ objects across the site. It's possible to do, but probably not worth covering for the exercise.

### System interface design

Now we have the rough estimates of how much storage we would like to use, let's think about data access. Initially, we will need to access all the cities, venues, performances and tickets to select from.

We want to be slightly pragmatic about how we load data. We have a series of nested objects: cities contain venues contain performances contain tickets. We could potentially have a single `/cities` endpoint which returns all of the nested data. However, this seems like overkill — we will get all the tickets every time!

Instead we have a single endpoint per data object.

- **Cities:** `/cities`
- **Venues:** `/cities/{id}/venues`
- **Performances:** `/cities/{id}/venues/{id}/performances`
- **Tickets:** `/cities/{id}/venues/{id}/performances/{id}/tickets`

We receive a list of data objects per request, with a `200` response and the regular `4XX`, `5XX` codes.

The other thing we need to do is to be able to reserve and purchase tickets. Let's say we have a ticket object similar to the below.

```
{
  "id": "<Id of the ticket object>",
  "state": "<AVAILABLE/ RESERVED/ PURCHASED>"
  ...
}
```

To reserve a ticket we could send a `POST` request to `/user/{id}/tickets`. The ticket object would then be linked to the user, and the state changed to reserved. To purchase a ticket a `PUT` (or `PATCH`) request would be made to the same endpoint, updating the state to purchased.

This is extra useful as the ticket would remain linked to the performance, but the state would have changed, showing it as purchased or reserved.

Another approach we could try using is GraphQL. Using traditional REST principles we can sometimes get embroiled in lots of different endpoints, multiple calls, and the unnecessary loading of data. GraphQL is a flexible query language for our APIs.

We initially define a schema for our cities:

```
type City {
  id: ID!
  name: String!
  venues: [Venue!]!
}
```

Each city has an Id (which is non-nullable, hence the exclamation mark), a name, and a list of venues. We can then query this using something of the format.

```
query
   {
     city(id: 1) {
       venues {
          name
       }
   }
```

Note, this is more of an aside. We'll stick with REST for the time being!

### Data model design

Now we need to design our data model. From the description so far we can see a relational pattern emerging. The other thing we need is **transactions** with **ACID properties** to handle the purchasing of tickets.

Let's dive into the definition of a transaction. A transaction is a unit of work for a database. For example, purchasing a ticket may be a transaction. There are two core focuses:

1. To provide units of work that allow for consistent state and recovery in the event of failures midway through.
2. To provide isolation between programs accessing a database concurrently.

We can put these in the context of a ticket purchase. Let's say a ticket costs £10, and the user has £10 in their bank account. In the context of a purchase we would like to:

1. Take £10 from the user.
2. Mark the ticket as purchased.
3. Assign the ticket to the user.

If our database falls over (as they are wont to do), we don't want to get into a state where the user has had £10 deducted, but the ticket is still available and not assigned (point 1)! Equally, if we have two users purchasing the same ticket, we don't want to deduct £10 from each of them, mark the ticket as purchased, then only be able to assign it to one of them (point 2)!

In Spring we use the `@Transactional` annotation to declare transactions. In the database layer itself we use a DBMS dependent syntax. For MySQL it is `START TRANSACTION;` and `COMMIT;`.

Another important concept to understand is ACID properties of transactions. ACID stands for:

1. **Atomicity:** The whole transaction comes as one unit of work. Either all of it completes, or none of it does.
2. **Consistency:** Our database will have a number of constraints on it (foreign keys, unique keys, etc.). This property guarantees that post transaction we will be left with a valid, constraint-abiding, set of data in our database.
3. **Isolation:** This is the ability to process multiple concurrent transactions such that they do not affect one another. If you and another person are both trying to buy a ticket simultaneously, one transaction must happen first.

4. **Durability:** Once a transaction is completed then the results are permanent. For example, if we buy a ticket, then there's a power cut, our ticket will still be purchased.

With all of that out the way, let's do our table design.

### City

- `id BIGINT PRIMARY KEY`
- `name VARCHAR`

### Venue

- `id BIGINT PRIMARY KEY`
- `name VARCHAR`
- `city BIGINT FOREIGN KEY REFERENCES city(ID)`

### Performance

- `id BIGINT PRIMARY KEY`
- `name VARCHAR`
- `venue BIGINT FOREIGN KEY REFERENCES venue(ID)`

## Ticket

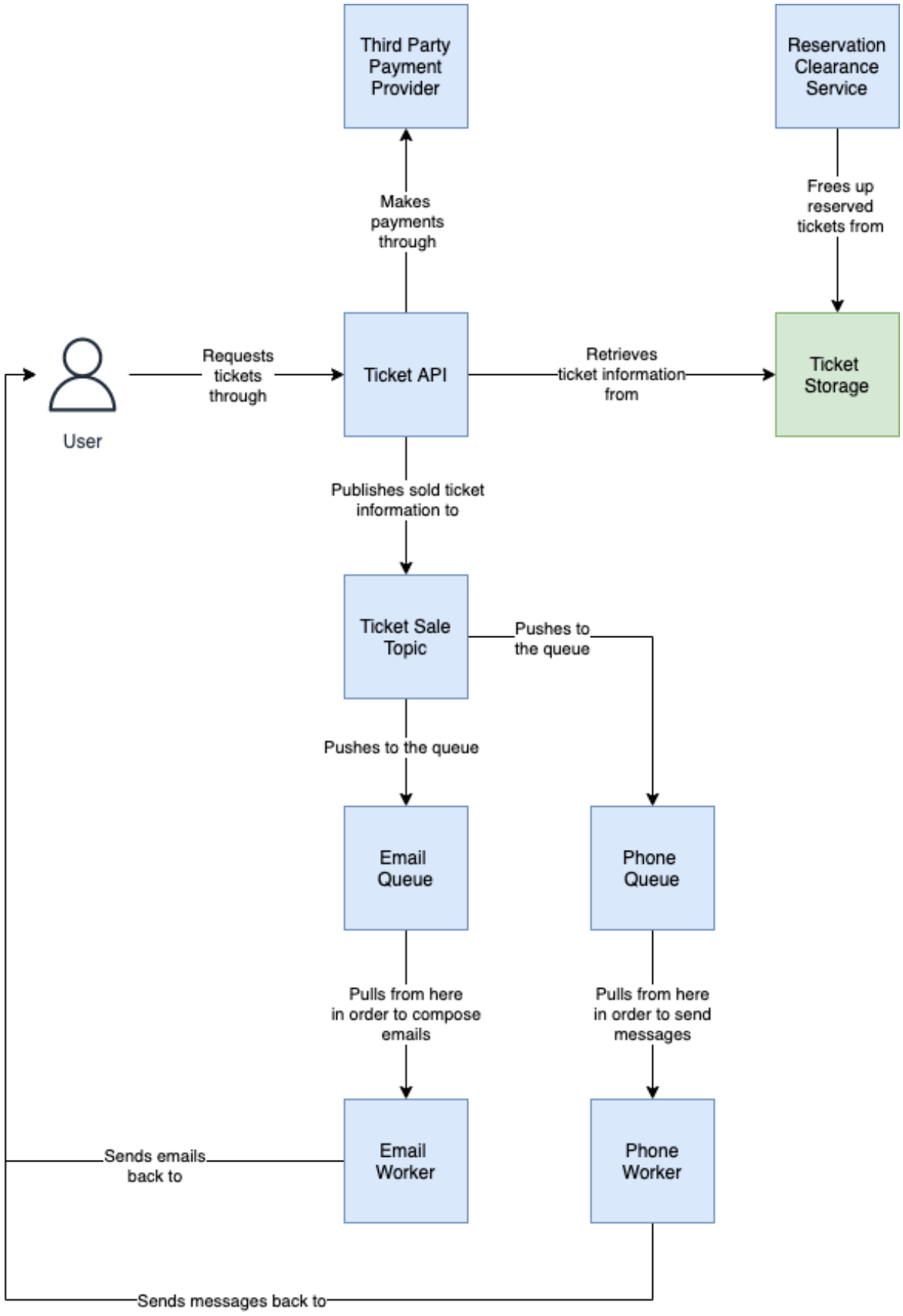
- id BIGINT PRIMARY KEY
- name VARCHAR
- performance BIGINT FOREIGN KEY REFERENCES performance(ID)
- user BIGINT FOREIGN KEY REFERENCES user(ID)
- reserved_until TIMESTAMP

## User

- id BIGINT PRIMARY KEY
- name VARCHAR
- phone_number INT
- email_address VARCHAR

### Logical design

The basic logical design is reasonably straightforward.


Third Party Payment Provider
Reservation Clearance Service
Makes payments through
Frees up reserved tickets from
User
Requests tickets through
Ticket API
Retrieves ticket information from
Ticket Storage
Publishes sold ticket information to
Ticket Sale Topic
Pushes to the queue
Pushes to the queue
Email Queue
Phone Queue
Pulls from here in order to compose emails
Pulls from here in order to send messages
Email Worker
Phone Worker
Sends emails back to
Sends messages back to

Basic Logical Design

Our user accesses ticket information through our ticket API. When a user adds a ticket to their basket, we reserve it, changing its state in the database. We also set a time `x` minutes in the future when the tickets will be removed from their basket.

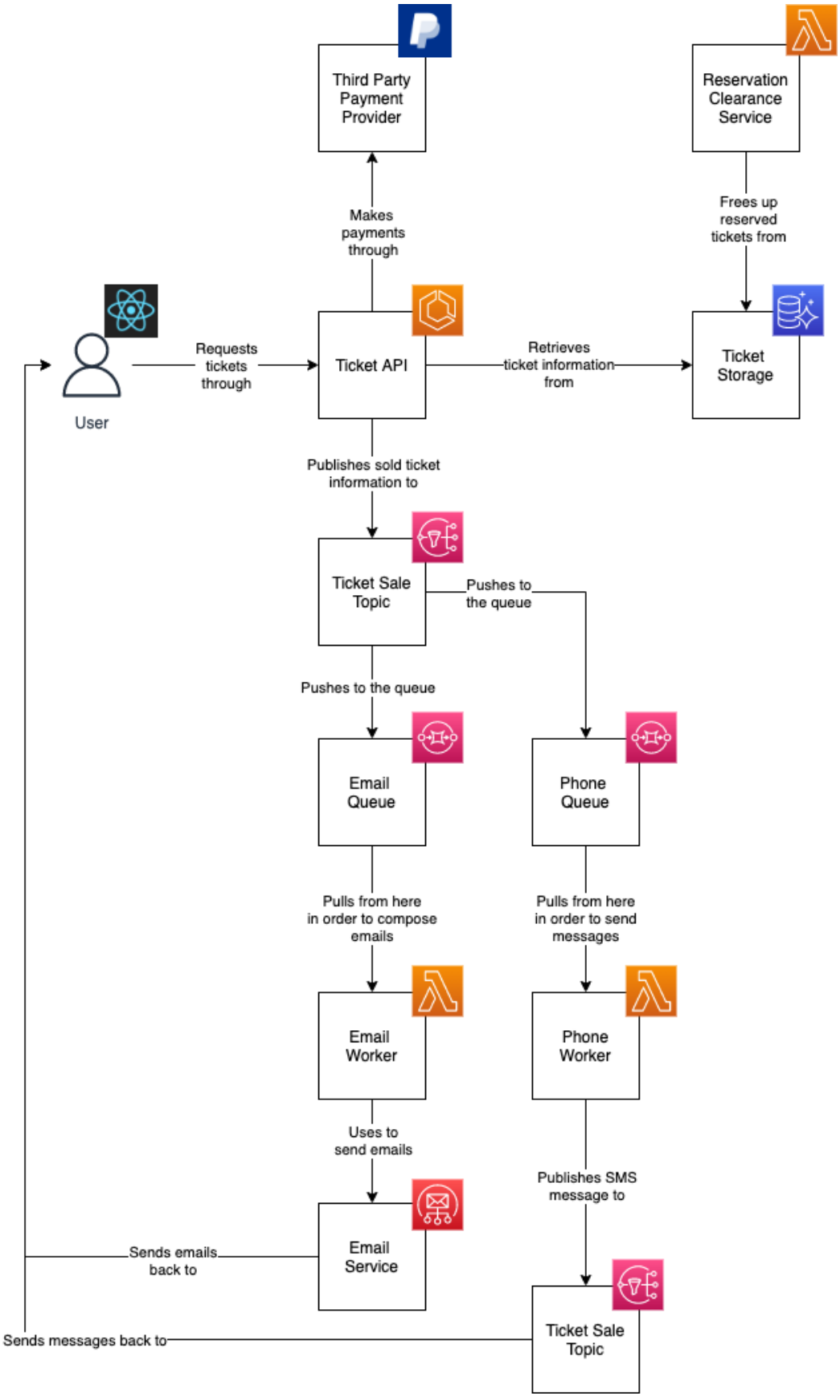
When a user purchases their ticket they will be redirected to a third party payment provider, and on success they will be redirected back to our ticket API to show a success page.

The ticket API will put a message on a sales topic, which is subscribed to by two queues. These two queues then push to two workers, who are responsible for sending emails and text messages.

The final part of the puzzle is how we free up tickets that have been reserved. This is the responsibility of another service, who polls the database every few minutes looking for expired reservations, freeing them up.

This is our very basic logical design, let's look at how we might build this.

## Physical design


Third Party Payment Provider
Reservation Clearance Service
Makes payments through
Frees up reserved tickets from
User
Requests tickets through
Ticket API
Retrieves ticket information from
Ticket Storage
Publishes sold ticket information to
Ticket Sale Topic
Pushes to the queue
Pushes to the queue
Email Queue
Phone Queue
Pulls from here in order to compose emails
Pulls from here in order to send messages
Email Worker
Phone Worker
Uses to send emails
Publishes SMS message to
Email Service
Sends emails back to
Ticket Sale Topic
Sends messages back to

A basic physical design

Here is our basic physical design. Our client is in React, which is provided by a Node server sitting on an AWS ECS cluster. This backend service is responsible for communicating to the MySQL database sitting on Aurora.

The reservation clearances are done by triggering a Lambda that runs on a timer, querying the DB and removing any outdated reservations.

Finally, on a successful sale we publish to an SNS topic subscribed to by two SQS queues. Both workers are Lambdas that use either SES to send emails, or publish to a topic that handles SMS.

Another interesting thing to examine while we're here is how we might integrate our third party payment platform. We've used PayPal as an example (other platforms available).

A really good article is here, and some demonstration client/ server code is here. The idea is that you embed a button on your website with two methods: `createOrder` and `onApprove`.

The `createOrder` method is used to create an order, telling PayPal what is being purchased. When you press the button we launch the PayPal checkout. On successfully completing the checkout we call the `onApprove` function, displaying a message to our user. The funds will safely be transferred to our PayPal account.

### Identify and resolve bottlenecks

There are a number of bottlenecks in our design. Initially, we could introduce things like a CDN for a level of protection/ caching. We could also add a caching layer for data that rarely changes (the list of cities, venues and performances is a good one).

Another optimisation would be the separation of the backend APIs: one Node service for serving up the React app, and one as a backend service for dealing with the database/ third party payments.

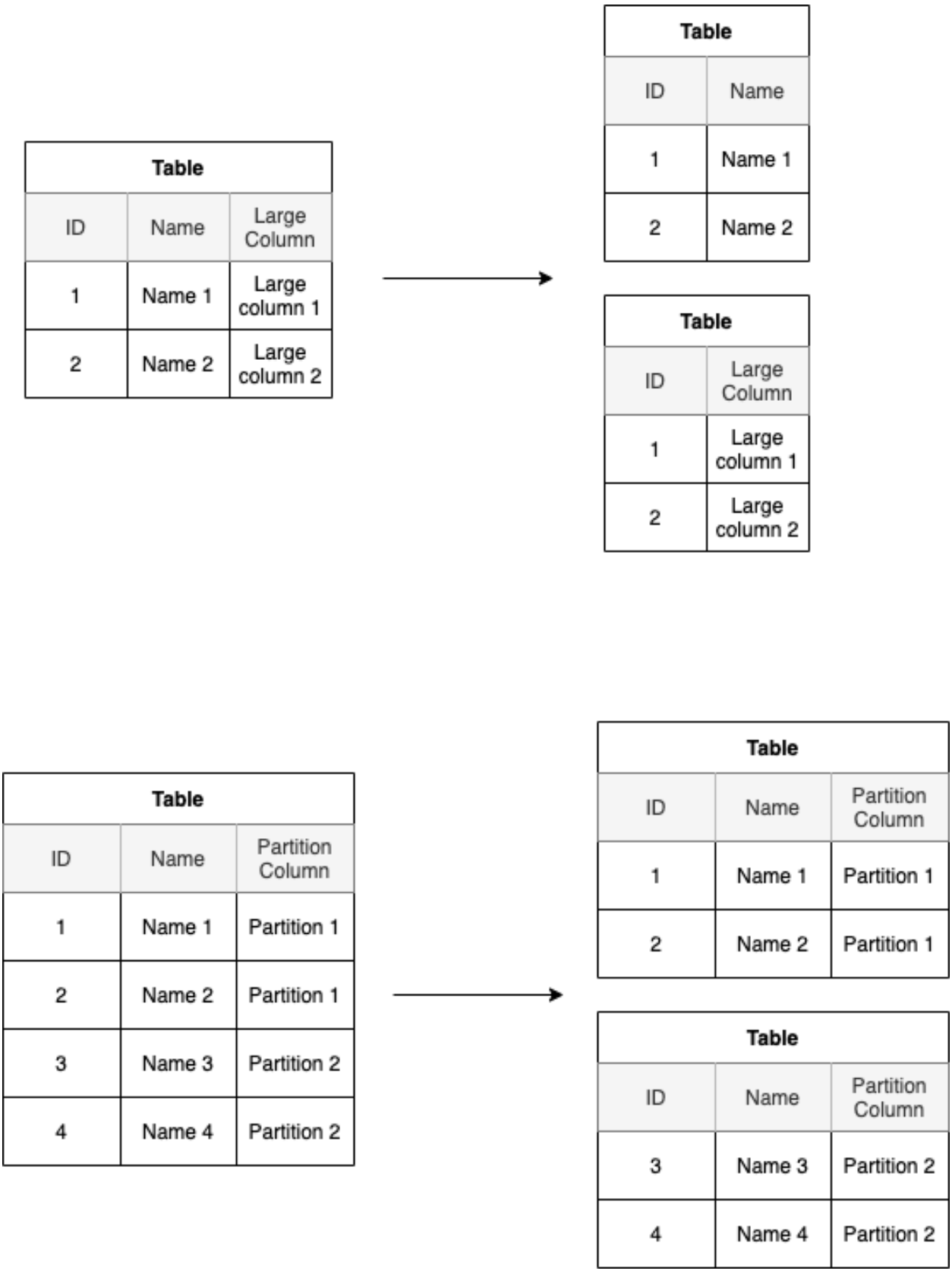
The core optimisations we can do are around the database. To do this, let's define some terms.

**Partitioning** is the process of breaking up large tables into smaller chunks. By doing this we reduce query times as there is less data to query at once. There are two main types: horizontal and vertical.

Vertical scaling is where we break a table up dependent on columns. Perhaps we have some columns that contain large amounts of data. It makes sense to store them separately so we don't query them each time. This is similar to **normalisation**, where we break tables down to reduce dependency and redundancy.

Horizontal partitioning is where we break a table up dependent on rows. Each row will have the same number of columns, however there

will now be multiple tables. Usually we horizontally partition based on a certain column.




Demonstrating vertical (top) and horizontal (bottom) partitioning

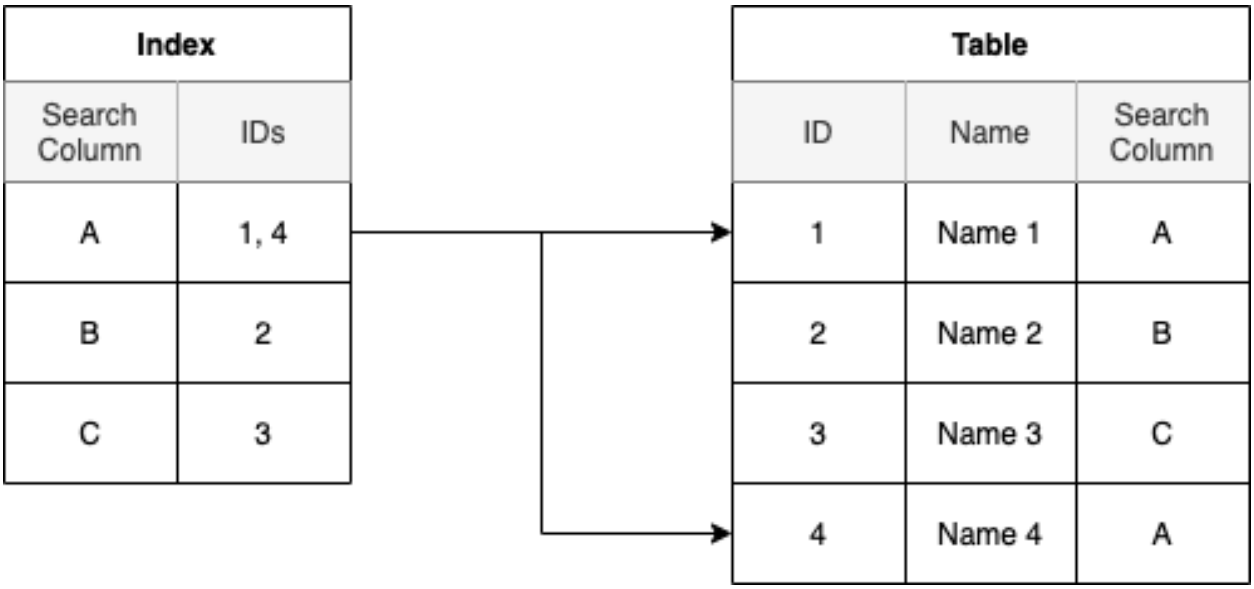
In the above we have vertically shared out the large column, and horizontally partitioned on the partition column.

**Sharding** is slightly different, and is a subset of horizontal partitioning. Horizontal partitioning is a logical separation of data, whereas sharding generally involves putting these logically separate blocks on different physical servers, identifying which server the data should sit on via a shard key.

But how does this relate to **indexing** I (perhaps) hear you cry? To answer this question, let's dig into what an index is.

When we store data for a table we store it in a block, with a pointer to the next block (think linked list). To find a particular block of data we need to search through all of them, one after another. As you can imagine, if what we need is right at the end of this list, this can be very slow.

To speed it up, we might have a data structure which maps the column we are searching for to the list of IDs with that value. We demonstrate below.



| Index | |
|---|---|
| Search Column | IDs |
| A | 1, 4 |
| B | 2 |
| C | 3 |

| Table | | |
|---|---|---|
| ID | Name | Search Column |
| 1 | Name 1 | A |
| 2 | Name 2 | B |
| 3 | Name 3 | C |
| 4 | Name 4 | A |

An index on our search column

This is a bit of a simplification, in reality they use a B+ Tree structure.

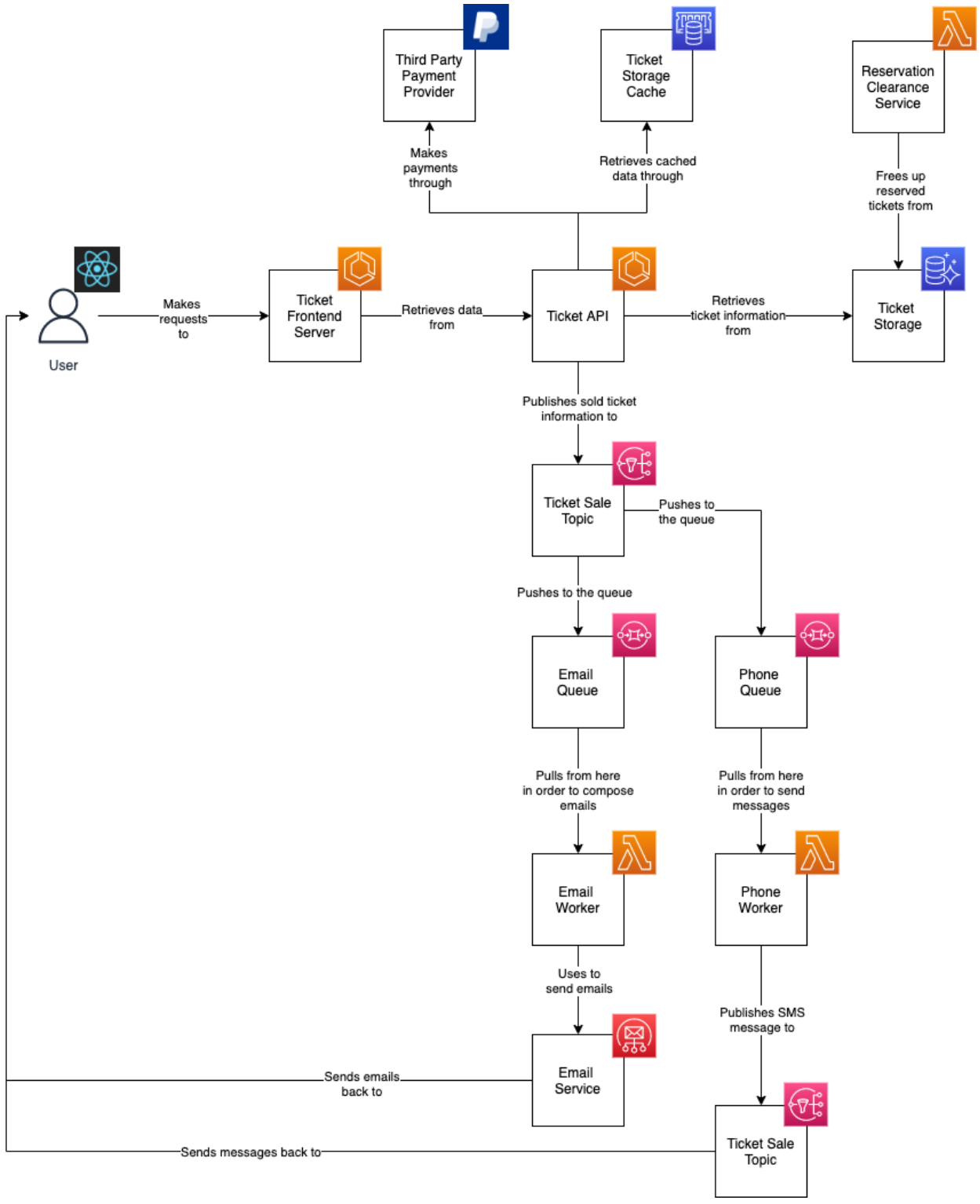
Another way to think about indexes is with a pack of cards. If I asked you to find the ace of spades, you would flick through the whole pack until you got it. However, if I separated them out into suits (equivalent to indexing on suits), it would be much easier to find!

So when include an index, and when include a partition? An index is good if you only want to access a small proportion of the data. Partitions are great if you want to access large portions of the data that you know will be grouped together.

Bringing that all together, we could partition/ shard by city! This makes sense as all venues, performances and tickets will be located per city.

The other issue we may want to address is resilience through replicas. We could have a replica per shard with a failover mechanism for if the main instance goes down.

Our final diagram is as below.




Final physical design

# Conclusion

In conclusion, we have discussed, designed and critiqued our own plan for a TicketMaster like service. I hope you have enjoyed!