[24] Pushgateway. https://github.com/prometheus/pushgateway.

[25] Building Applications with Serverless Architectures. https://aws.amazon.com/lambda/serverless-architectures-learn-more/.

[26] Gorilla. AFast,Scalable,In-MemoryTimeSeriesDatabase:http://www.vldb.org/pvldb/vol8/p1816-teller.pdf.

[27] Why We're Building Flux, a New Data Scripting and Query Language. https://www.influxdata.com/blog/why-were-building-flux-a-new-data-scripting-and-query-language/.

[28] InfluxDB storage engine. https://docs.influxdata.com/influxdb/v2.0/reference/internals/storage-engine/.

[29] YAML. https://en.wikipedia.org/wiki/YAML.

[30] Grafana Demo. https://play.grafana.org/.

6 Ad Click Event Aggregation

With the rise of Facebook, YouTube, TikTok, and the online media economy, digital advertising is taking an ever-bigger share of the total advertising spending. As a result, tracking ad click events is very important. In this chapter, we explore how to design an ad click event aggregation system at Facebook or Google scale.

Before we dive into technical design, let's learn about the core concepts of online advertising to better understand this topic. One core benefit of online advertising is its measurability, as quantified by real-time data.

Digital advertising has a core process called Real-Time Bidding (RTB), in which digital advertising inventory is bought and sold. Figure 6.1 shows how the online advertising process works.

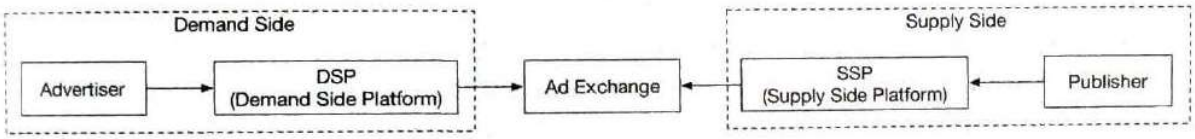


Figure 6.1: RTB process

The speed of the RTB process is important as it usually occurs in less than a second.

Data accuracy is also very important. Ad click event aggregation plays a critical role in measuring the effectiveness of online advertising, which essentially impacts how much money advertisers pay. Based on the click aggregation results, campaign managers can control the budget or adjust bidding strategies, such as changing targeted audience groups, keywords, etc. The key metrics used in online advertising, including click-through rate (CTR) [1] and conversion rate (CVR) [2], depend on aggregated ad click data.

Step 1 - Understand the Problem and Establish Design Scope

The following set of questions helps to clarify requirements and narrow down the scope.

Candidate: What is the format of the input data?
Interviewer: It's a log file located in different servers and the latest click events are appended to the end of the log file. The event has the following attributes: `ad_id`, `click_timestamp`, `user_id`, `ip`, and `country`.

Candidate: What's the data volume?
Interviewer: 1 billion ad clicks per day and 2 million ads in total. The number of ad click events grows 30% year-over-year.

Candidate: What are some of the most important queries to support?
Interviewer: The system needs to support the following 3 queries:

- Return the number of click events for a particular ad in the last M minutes.
- Return the top 100 most clicked ads in the past 1 minute. Both parameters should be configurable. Aggregation occurs every minute.
- Support data filtering by `ip`, `user_id`, or `country` for the above two queries.

Candidate: Do we need to worry about edge cases? I can think of the following:

- There might be events that arrive later than expected.
- There might be duplicated events.
- Different parts of the system might be down at any time, so we need to consider system recovery.

Interviewer: That's a good list. Yes, take these into consideration.

Candidate: What is the latency requirement?
Interviewer: A few minutes of end-to-end latency. Note that latency requirements for RTB and ad click aggregation are very different. While latency for RTB is usually less than one second due to the responsiveness requirement, a few minutes of latency is acceptable for ad click event aggregation because it is primarily used for ad billing and reporting.

With the information gathered above, we have both functional and non-functional requirements.

Functional requirements

- Aggregate the number of clicks of `ad_id` in the last M minutes.
- Return the top 100 most clicked `ad_id` every minute.
- Support aggregation filtering by different attributes.
- Dataset volume is at Facebook or Google scale (see the back-of-envelope estimation section below for detailed system scale requirements).

Non-functional requirements

- Correctness of the aggregation result is important as the data is used for RTB and ads billing.

- Properly handle delayed or duplicate events.
- Robustness. The system should be resilient to partial failures.
- Latency requirement. End-to-end latency should be a few minutes, at most.

Back-of-the-envelope estimation

Let's do an estimation to understand the scale of the system and the potential challenges we will need to address.

- 1 billion DAU (Daily Active Users).
- Assume on average each user clicks 1 ad per day. That's 1 billion ad click events per day.
- Ad click QPS $= \frac{10^9 \text{ events}}{10^5 \text{ seconds in a day}} = 10{,}000$
- Assume peak ad click QPS is 5 times the average number. Peak QPS = 50,000 QPS.
- Assume a single ad click event occupies 0.1KB storage. Daily storage requirement is: 0.1KB $\times$ 1 billion = 100GB. The monthly storage requirement is about 3TB.

Step 2 - Propose High-level Design and Get Buy-in

In this section, we discuss query API design, data model, and high-level design.

Query API design

The purpose of the API design is to have an agreement between the client and the server. In a consumer app, a client is usually the end-user who uses the product. In our case, however, a client is the dashboard user (data scientist, product manager, advertiser, etc.) who runs queries against the aggregation service.

Let's review the functional requirements so we can better design the APIs:

- Aggregate the number of clicks of `ad_id` in the last M minutes.
- Return the top N most clicked `ad_ids` in the last M minute.
- Support aggregation filtering by different attributes.

We only need two APIs to support those three use cases because filtering (the last requirement) can be supported by adding query parameters to the requests.

API 1: Aggregate the number of clicks of `ad_id` in the last M minutes.

API	Detail
`GET /v1/ads/{:ad_id}/aggregated_count`	Return aggregated event count for a given `ad_id`

Table 6.1: API for aggregating the number of clicks

Request parameters are:

Field	Description	Type
from	Start minute (default is now minus 1 minute)	long
to	End minute (default is now)	long
filter	An identifier for different filtering strategies. For example, filter = 001 filters out non-US clicks	long

Table 6.2: Request parameters for /v1/ads/{:ad_id}/aggregated_count

Response:

Field	Description	Type
ad_id	The identifier of the ad	string
count	The aggregated count between the start and end minutes	long

Table 6.3: Response for /v1/ads/{:ad_id}/aggregated_count

API 2: Return top N most clicked ad_ids in the last M minutes

API	Detail
GET /v1/ads/popular_ads	Return top N most clicked ads in the last M minutes

Table 6.4: API for /v1/ads/popular_ads

Request parameters are:

Field	Description	Type
count	Top N most clicked ads	integer
window	The aggregation window size (M) in minutes	integer
filter	An identifier for different filtering strategies	long

Table 6.5: Request parameters for /v1/ads/popular_ads

Response:

Field	Description	Type
ad_ids	A list of the most clicked ads	array

Table 6.6: Response for /v1/ads/popular_ads

Data model

There are two types of data in the system: raw data and aggregated data.

Raw data

Below shows what the raw data looks like in log files:

```
[AdClickEvent] ad001, 2021-01-01 00:00:01, user 1, 207.148.22.22, USA
```

Table 6.7 lists what the data fields look like in a structured way. Data is scattered on different application servers.

ad_id	click_timestamp	user_id	ip	country
ad001	2021-01-01 00:00:01	user1	207.148.22.22	USA
ad001	2021-01-01 00:00:02	user1	207.148.22.22	USA
ad002	2021-01-01 00:00:02	user2	209.153.56.11	USA

Table 6.7: Raw data

Aggregated data

Assume that ad click events are aggregated every minute. Table 6.8 shows the aggregated result.

ad_id	click_minute	count
ad001	202101010000	5
ad001	202101010001	7

Table 6.8: Aggregated data

To support ad filtering, we add an additional field called `filter_id` to the table. Records with the same `ad_id` and `click_minute` are grouped by `filter_id` as shown in Table 6.9, and filters are defined in Table 6.10.

ad_id	click_minute	filter_id	count
ad001	202101010000	0012	2
ad001	202101010000	0023	3
ad001	202101010001	0012	1
ad001	202101010001	0023	6

Table 6.9: Aggregated data with filters

filter_id	region	ip	user_id
0012	US	0012	*
0013	*	0023	123.1.2.3

Table 6.10: Filter table

To support the query to return the top N most clicked ads in the last M minutes, the following structure is used.

Most_clicked_ads		
window_size	integer	The aggregation window size M in minutes
update_time_minute	timestamp	Last updated timestamp in 1-minute granularity
most_clicked_ads	array	List of ad IDs in JSON format

Table 6.11: Support top N most clicked ads in the last M minutes

Comparison

The comparison between storing raw data and aggregated data is shown below:

	Raw data only	Aggregated data only
Pros	• Full data set • Support data filter and recalculation	• Smaller data set • Fast query
Cons	• Huge data storage • Slow query	• Data loss. This is derived data. For example, 10 entries might be aggregated to 1 entry

Table 6.12: Raw data vs aggregated data

Should we store raw data or aggregated data? Our recommendation is to store both. Let's take a look at why.

- It's a good idea to keep the raw data. If something goes wrong, we could use the raw data for debugging. If the aggregated data is corrupted due to a bad bug, we can recalculate the aggregated data from the raw data, after the bug is fixed.
- Aggregated data should be stored as well. The data size of the raw data is huge. The large size makes querying raw data directly very inefficient. To mitigate this problem, we run read queries on aggregated data.
- Raw data serves as backup data. We usually don't need to query raw data unless recalculation is needed. Old raw data could be moved to cold storage to reduce costs.
- Aggregated data serves as active data. It is tuned for query performance.

Choose the right database

When it comes to choosing the right database, we need to evaluate the following:

- What does the data look like? Is the data relational? Is it a document or a blob?
- Is the workflow read-heavy, write-heavy, or both?
- Is transaction support needed?
- Do the queries rely on many online analytical processing (OLAP) functions [3] like SUM, COUNT?

Let's examine the raw data first. Even though we don't need to query the raw data during normal operations, it is useful for data scientists or machine learning engineers to study user response prediction, behavioral targeting, relevance feedback, etc. [4].

As shown in the back of the envelope estimation, the average write QPS is 10,000, and the peak QPS can be 50,000, so the system is write-heavy. On the read side, raw data is used as backup and a source for recalculation, so in theory, the read volume is low.

Relational databases can do the job, but scaling the write can be challenging. NoSQL databases like Cassandra and InfluxDB are more suitable because they are optimized for write and time-range queries.

Another option is to store the data in Amazon S3 using one of the columnar data formats like ORC [5], Parquet [6], or AVRO [7]. We could put a cap on the size of each file (say, 10GB) and the stream processor responsible for writing the raw data could handle the file rotation when the size cap is reached. Since this setup may be unfamiliar for many, in this design we use Cassandra as an example.

For aggregated data, it is time-series in nature and the workflow is both read and write heavy. This is because, for each ad, we need to query the database every minute to display the latest aggregation count for customers. This feature is useful for auto-refreshing the dashboard or triggering alerts in a timely manner. Since there are two million ads in total, the workflow is read-heavy. Data is aggregated and written every minute by the aggregation service, so it's write-heavy as well. We could use the same type of database to store both raw data and aggregated data.

Now we have discussed query API design and data model, let's put together the high-level design.

High-level design

In real-time big data [8] processing, data usually flows into and out of the processing system as unbounded data streams. The aggregation service works in the same way; the input is the raw data (unbounded data streams), and the output is the aggregated results (see Figure 6.2).

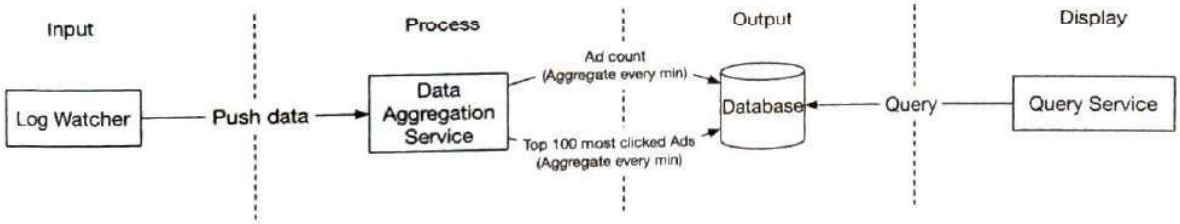


Figure 6.2: Aggregation workflow

Asynchronous processing

The design we currently have is synchronous. This is not good because the capacity of producers and consumers is not always equal. Consider the following case; if there is a sudden increase in traffic and the number of events produced is far beyond what consumers can handle, consumers might get out-of-memory errors or experience an unex-

pected shutdown. If one component in the synchronous link is down, the whole system stops working.

A common solution is to adopt a message queue (Kafka) to decouple producers and consumers. This makes the whole process asynchronous and producers/consumers can be scaled independently.

Putting everything we have discussed together, we come up with the high-level design as shown in Figure 6.3. Log watcher, aggregation service, and database are decoupled by two message queues. The database writer polls data from the message queue, transforms the data into the database format, and writes it to the database.

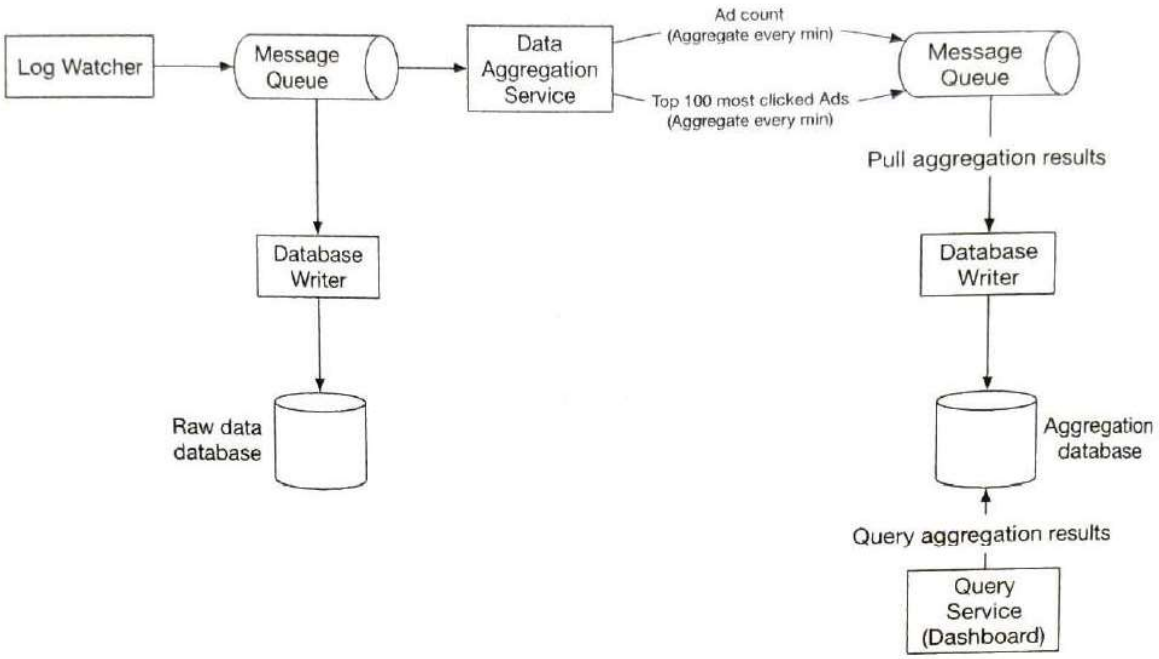


Figure 6.3: High-level design

What is stored in the first message queue? It contains ad click event data as shown in Table 6.13.

ad_id	click_timestamp	user_id	ip	country

Table 6.13: Data in the first message queue

What is stored in the second message queue? The second message queue contains two types of data:

1. Ad click counts aggregated at per-minute granularity.

ad_id	click_minute	count

Table 6.14: Data in the second message queue

2. Top N most clicked ads aggregated at per-minute granularity.

update_time_minute	most_clicked_ads

Table 6.15: Data in the second message queue

You might be wondering why we don't write the aggregated results to the database directly. The short answer is that we need the second message queue like Kafka to achieve end-to-end exactly once semantics (atomic commit) [9].

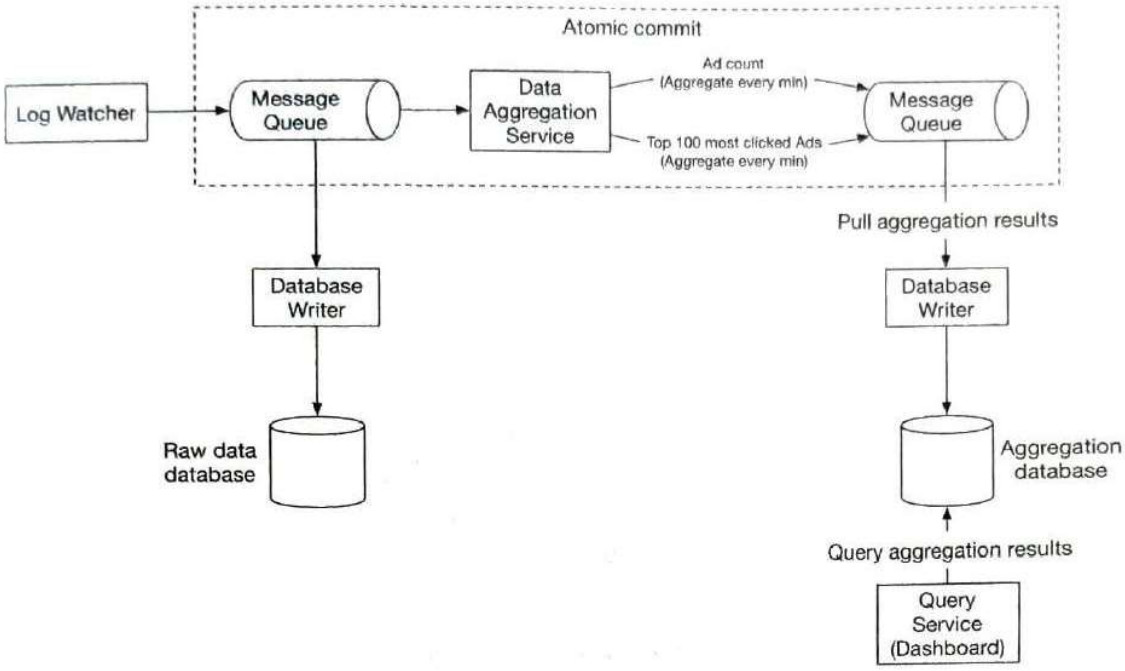


Figure 6.4: End-to-end exactly once

Next, let's dig into the details of the aggregation service.

Aggregation service

The MapReduce framework is a good option to aggregate ad click events. The directed acyclic graph (DAG) is a good model for it [10]. The key to the DAG model is to break down the system into small computing units, like the Map/Aggregate/Reduce nodes, as shown in Figure 6.5.

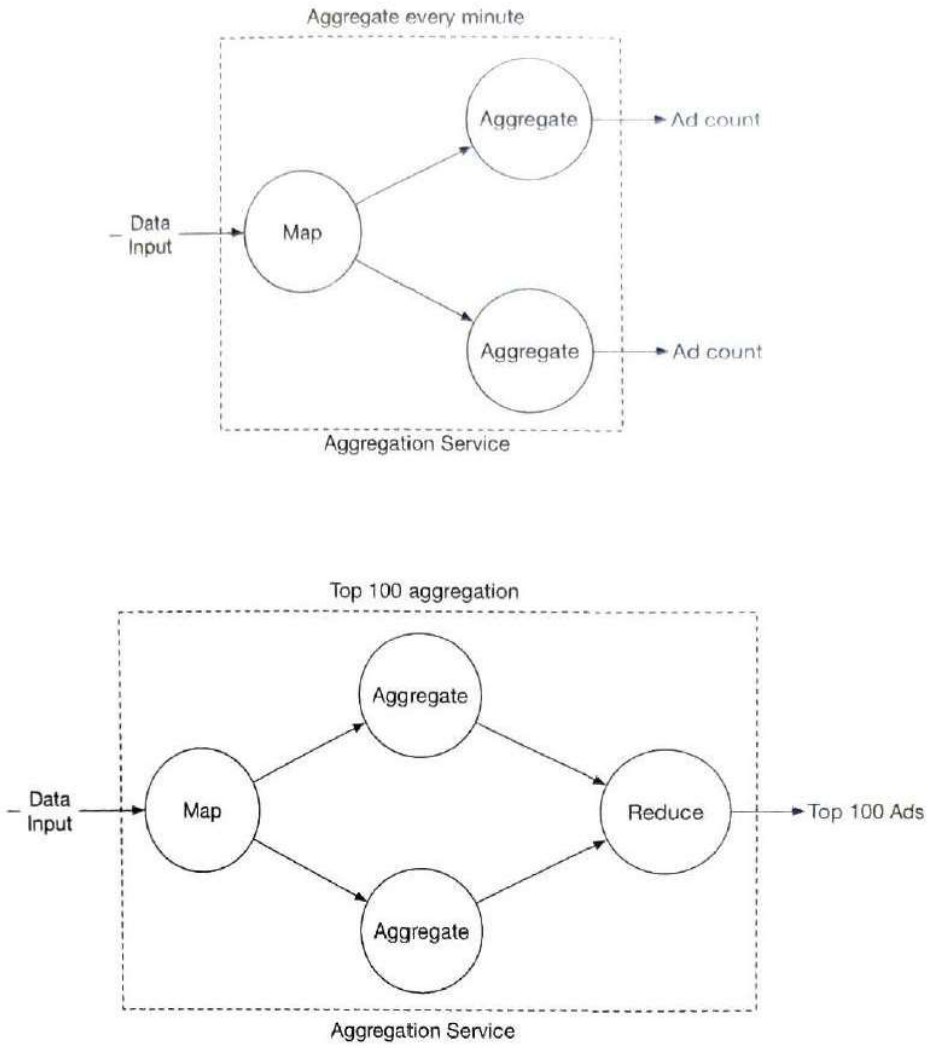


Figure 6.5: Aggregation service

Each node is responsible for one single task and it sends the processing result to its downstream nodes.

Map node

A Map node reads data from a data source, and then filters and transforms the data. For example, a Map node sends ads with `ad_id` %2 = 0 to node 1, and the other ads go to node 2, as shown in Figure 6.6.

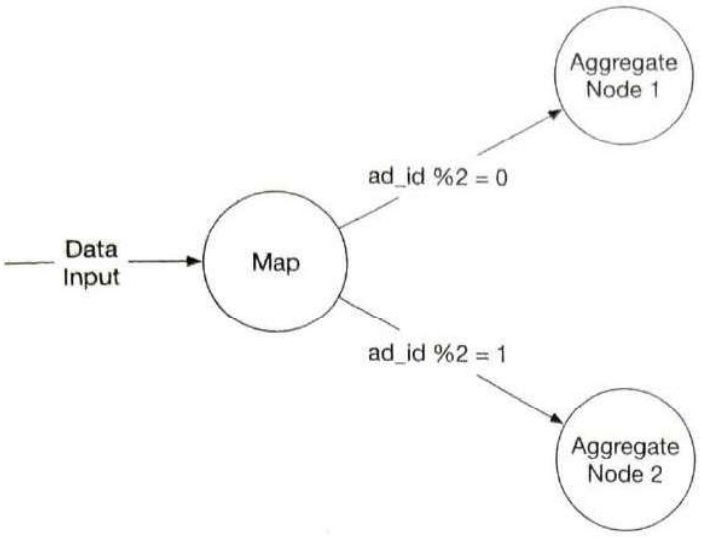


Figure 6.6: Map operation

You might be wondering why we need the Map node. An alternative option is to set up Kafka partitions or tags and let the aggregate nodes subscribe to Kafka directly. This works, but the input data may need to be cleaned or normalized, and these operations can be done by the Map node. Another reason is that we may not have control over how data is produced and therefore events with the same `ad_id` might land in different Kafka partitions.

Aggregate node

An Aggregate node counts ad click events by `ad_id` in memory every minute. In the MapReduce paradigm, the Aggregate node is part of the Reduce. So the map-aggregate-reduce process really means map-reduce-reduce.

Reduce node

A Reduce node reduces aggregated results from all "Aggregate" nodes to the final result. For example, as shown in Figure 6.7, there are three aggregation nodes and each contains the top 3 most clicked ads within the node. The Reduce node reduces the total number of most clicked ads to 3.

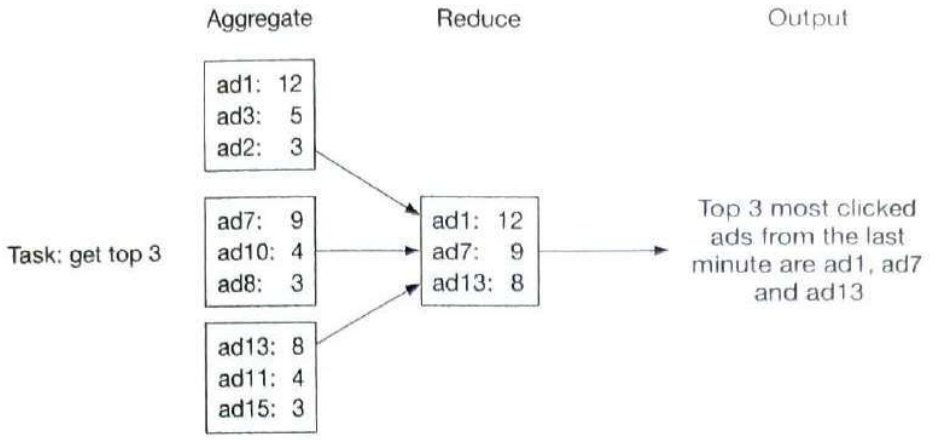


Figure 6.7: Reduce node

The DAG model represents the well-known MapReduce paradigm. It is designed to take big data and use parallel distributed computing to turn big data into little- or regular-sized data.

In the DAG model, intermediate data can be stored in memory and different nodes communicate with each other through either TCP (nodes running in different processes) or shared memory (nodes running in different threads).

Main use cases

Now that we understand how MapReduce works at the high level, let's take a look at how it can be utilized to support the main use cases:

- Aggregate the number of clicks of `ad_id` in the last M mins.
- Return top N most clicked `ad_ids` in the last M minutes.
- Data filtering.

Use case 1: aggregate the number of clicks

As shown in Figure 6.8, input events are partitioned by `ad_id` (`ad_id` %3) in Map nodes and are then aggregated by Aggregation nodes.

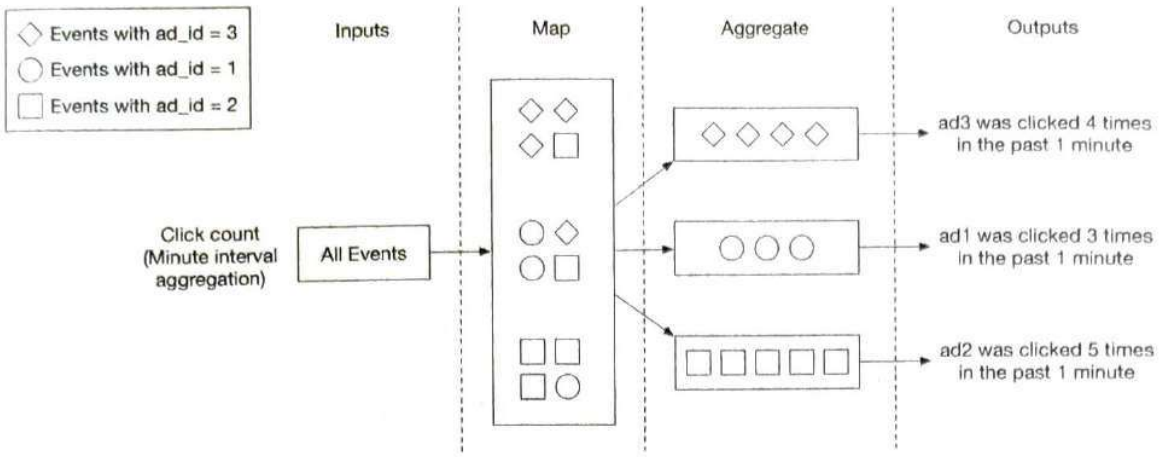


Figure 6.8: Aggregate the number of clicks

Use case 2: return top N most clicked ads

Figure 6.9 shows a simplified design of getting the top 3 most clicked ads, which can be extended to top N. Input events are mapped using ad_id and each Aggregate node maintains a heap data structure to get the top 3 ads within the node efficiently. In the last step, the Reduce node reduces 9 ads (top 3 from each aggregate node) to the top 3 most clicked ads every minute.

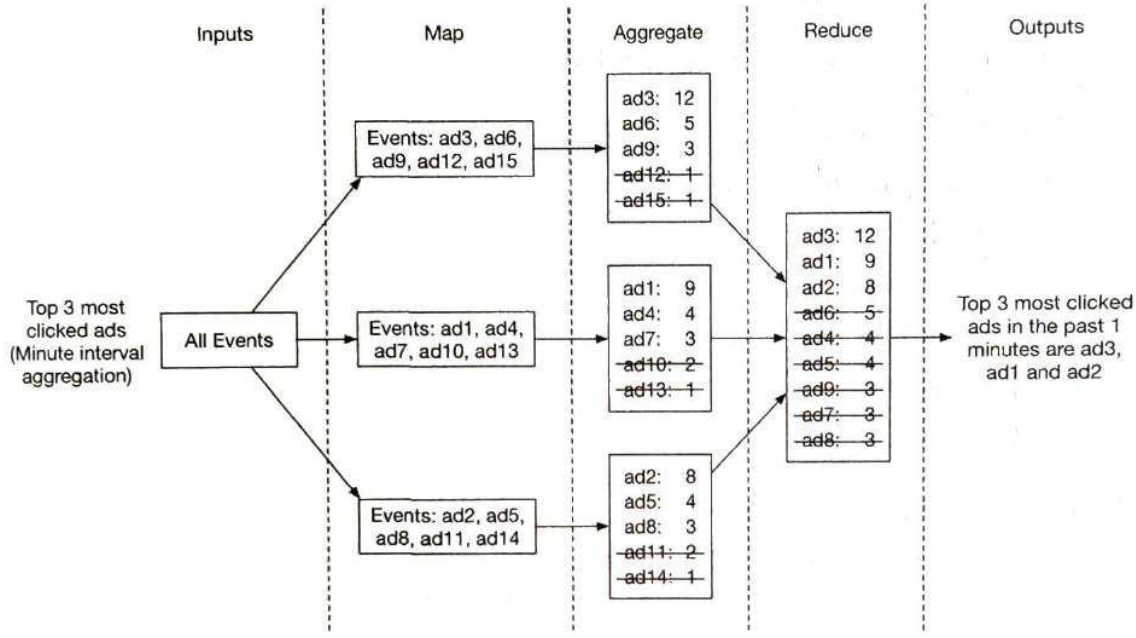


Figure 6.9: Return top N most clicked ads

Use case 3: data filtering

To support data filtering like "show me the aggregated click count for ad001 within the USA only", we can pre-define filtering criteria and aggregate based on them. For example, the aggregation results look like this for `ad001` and `ad002`:

ad_id	click_minute	country	count
ad001	202101010001	USA	100
ad001	202101010001	GPB	200
ad001	202101010001	others	3000
ad002	202101010001	USA	10
ad002	202101010001	GPB	25
ad002	202101010001	others	12

Table 6.16: Aggregation results (filter by country)

This technique is called the star schema [11], which is widely used in data warehouses. The filtering fields are called dimensions. This approach has the following benefits:

- It is simple to understand and build.
- The current aggregation service can be reused to create more dimensions in the star schema. No additional component is needed.
- Accessing data based on filtering criteria is fast because the result is pre-calculated.

A limitation with this approach is that it creates many more buckets and records, especially when we have a lot of filtering criteria.

Step 3 - Design Deep Dive

In this section, we will dive deep into the following:

- Streaming vs batching
- Time and aggregation window
- Delivery guarantees
- Scale the system
- Data monitoring and correctness
- Final design diagram
- Fault tolerance

Streaming vs batching

The high-level architecture we proposed in Figure 6.3 is a type of stream processing system. Table 6.17 shows the comparison of three types of systems [12]:

	Services (Online system)	Batch system (offline system)	Streaming system (near real-time system)
Responsiveness	Respond to the client quickly	No response to the client needed	No response to the client needed
Input	User requests	Bounded input with finite size. A large amount of data	Input has no boundary (infinite streams)
Output	Responses to clients	Materialized views, aggregated metrics, etc.	Materialized views, aggregated metrics, etc.
Performance measurement	Availability, latency	Throughput	Throughput, latency
Example	Online shopping	MapReduce	Flink [13]

Table 6.17: Comparison of three types of systems

In our design, both stream processing and batch processing are used. We utilized stream processing to process data as it arrives and generates aggregated results in a near real-time fashion. We utilized batch processing for historical data backup.

For a system that contains two processing paths (batch and streaming) simultaneously, this architecture is called lambda [14]. A disadvantage of lambda architecture is that you have two processing paths, meaning there are two codebases to maintain. Kappa architecture [15], which combines the batch and streaming in one processing path, solves the problem. The key idea is to handle both real-time data processing and continuous data reprocessing using a single stream processing engine. Figure 6.10 shows a comparison of lambda and kappa architecture.

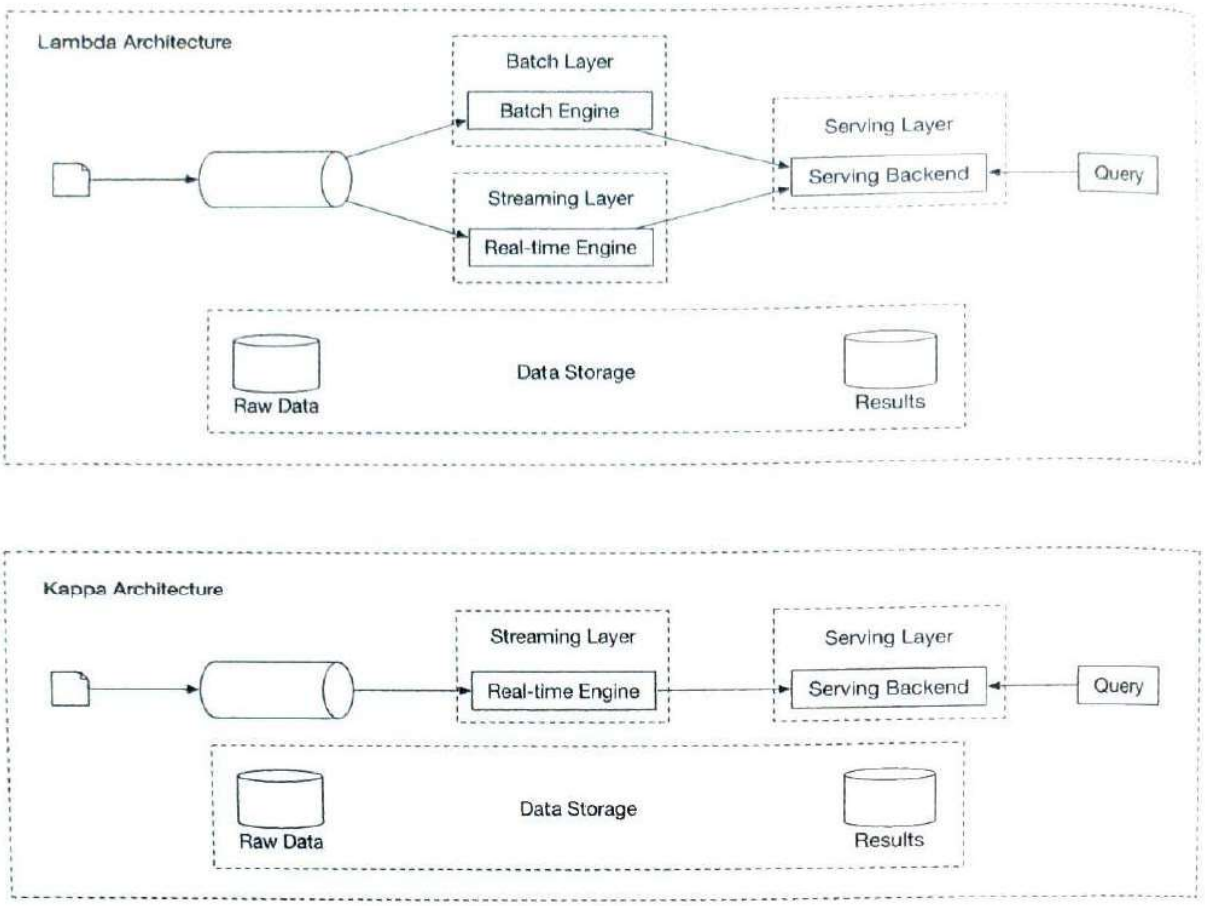


Figure 6.10: Lambda and Kappa architectures

Our high-level design uses Kappa architecture, where the reprocessing of historical data also goes through the real-time aggregation service. See the "Data recalculation" section below for details.

Data recalculation

Sometimes we have to recalculate the aggregated data, also called historical data replay. For example, if we discover a major bug in the aggregation service, we would need to recalculate the aggregated data from raw data starting at the point where the bug was introduced. Figure 6.11 shows the data recalculation flow:

1. The recalculation service retrieves data from raw data storage. This is a batched job.
2. Retrieved data is sent to a dedicated aggregation service so that the real-time processing is not impacted by historical data replay.
3. Aggregated results are sent to the second message queue, then updated in the aggregation database.

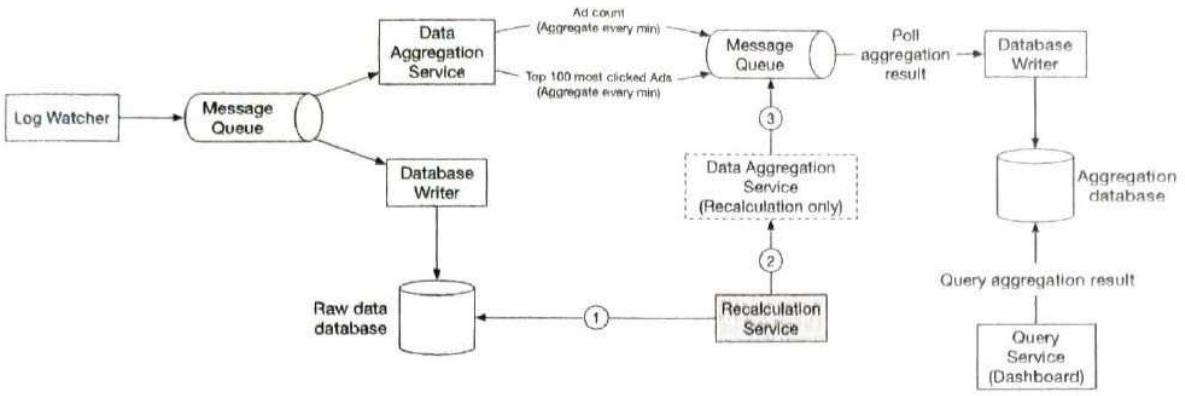


Figure 6.11: Recalculation service

The recalculation process reuses the data aggregation service but uses a different data source (the raw data).

Time

We need a timestamp to perform aggregation. The timestamp can be generated in two different places:

- Event time: when an ad click happens.
- Processing time: refers to the system time of the aggregation server that processes the click event.

Due to network delays and asynchronous environments (data go through a message queue), the gap between event time and processing time can be large. As shown in Figure 6.12, event 1 arrives at the aggregation service very late (5 hours later).

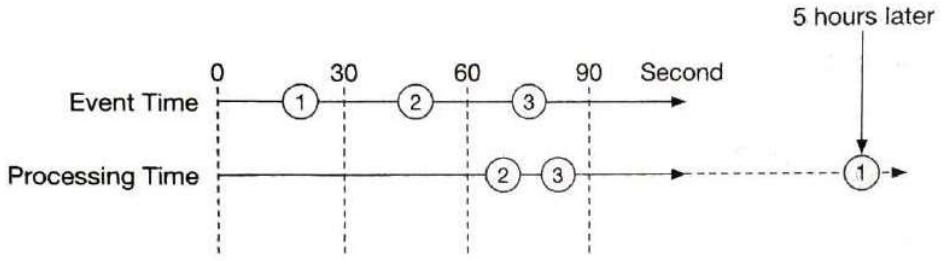


Figure 6.12: Late events

If event time is used for aggregation, we have to deal with delayed events. If processing time is used for aggregation, the aggregation result may not be accurate. There is no perfect solution, so we need to consider the trade-offs.

	Pros	Cons
Event time	Aggregation results are more accurate because the client knows exactly when an ad is clicked	It depends on the timestamp generated on the client-side. Clients might have the wrong time, or the timestamp might be generated by malicious users
Processing time	Server timestamp is more reliable	The timestamp is not accurate if an event reaches the system at a much later time

Table 6.18: Event time vs processing time

Since data accuracy is very important, we recommend using event time for aggregation. How do we properly process delayed events in this case? A technique called "watermark" is commonly utilized to handle slightly delayed events.

In Figure 6.13, ad click events are aggregated in the one-minute tumbling window (see the "Aggregation window" section on page 177 for more details). If event time is used to decide whether the event is in the window, window 1 misses event 2, and window 3 misses event 5 because they arrive slightly later than the end of their aggregation windows.

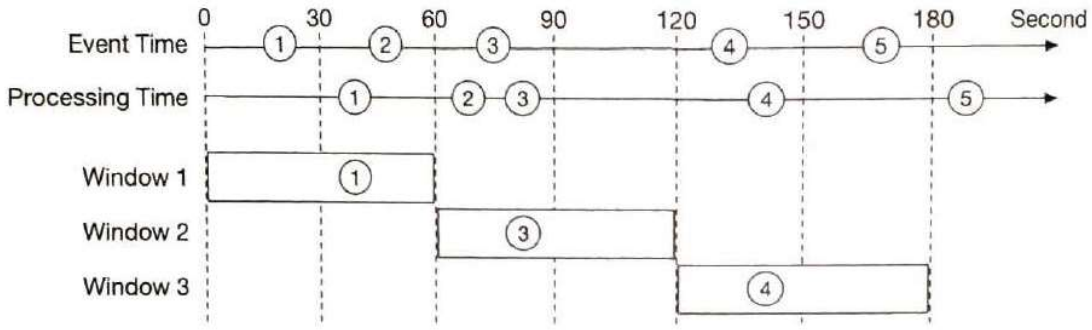


Figure 6.13: Miss events in an aggregation window

One way to mitigate this problem is to use "watermark" (the extended rectangles in Figure 6.14), which is regarded as an extension of an aggregation window. This improves the accuracy of the aggregation result. By extending an extra 15 second (adjustable) aggregation window, window 1 is able to include event 2, and window 3 is able to include event 5.

The value set for the watermark depends on the business requirement. A long watermark could catch events that arrive very late, but it adds more latency to the system. A short watermark means data is less accurate, but it adds less latency to the system.

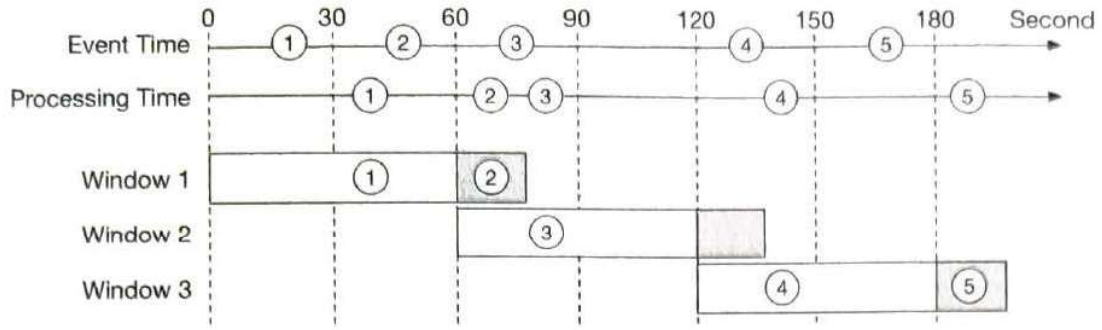


Figure 6.14: Watermark

Notice that the watermark technique does not handle events that have long delays. We can argue that it is not worth the return on investment (ROI) to have a complicated design for low probability events. We can always correct the tiny bit of inaccuracy with end-of-day reconciliation (see "Reconciliation" section on page 189). One trade-off to consider is that using watermark improves data accuracy but increases overall latency, due to extended wait time.

Aggregation window

According to the "Designing data-intensive applications" book by Martin Kleppmann [16], there are four types of window functions: tumbling window (also called fixed window), hopping window, sliding window, and session window. We will discuss the tumbling window and sliding window as they are most relevant to our system.

In the tumbling window (highlighted in Figure 6.15), time is partitioned into same-length, non-overlapping chunks. The tumbling window is a good fit for aggregating ad click events every minute (use case 1).

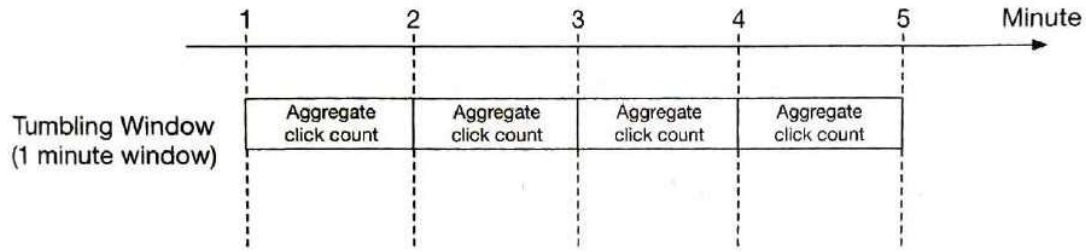


Figure 6.15: Tumbling window

In the sliding window (highlighted in Figure 6.16), events are grouped within a window that slides across the data stream, according to a specified interval. A sliding window can be an overlapping one. This is a good strategy to satisfy our second use case; to get the top N most clicked ads during the last M minutes.

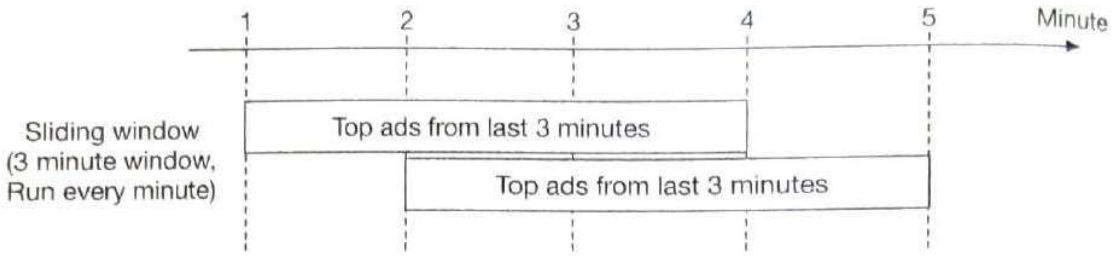


Figure 6.16: Sliding window

Delivery guarantees

Since the aggregation result is utilized for billing, data accuracy and completeness are very important. The system needs to be able to answer questions such as:

- How to avoid processing duplicate events?
- How to ensure all events are processed?

Message queues such as Kafka usually provide three delivery semantics: at-most once, at-least once, and exactly once.

Which delivery method should we choose?

In most circumstances, at-least once processing is good enough if a small percentage of duplicates are acceptable.

However, this is not the case for our system. Differences of a few percent in data points could result in discrepancies of millions of dollars. Therefore, we recommend exactly-once delivery for the system. If you are interested in learning more about a real-life ad aggregation system, take a look at how Yelp implements it [17].

Data deduplication

One of the most common data quality issues is duplicated data. Duplicated data can come from a wide range of sources and in this section, we discuss two common sources.

- Client-side. For example, a client might resend the same event multiple times. Duplicated events sent with malicious intent are best handled by ad fraud/risk control components. If this is of interest, please refer to the reference material [18].
- Server outage. If an aggregation service node goes down in the middle of aggregation and the upstream service hasn't yet received an acknowledgment, the same events might be sent and aggregated again. Let's take a closer look.

Figure 6.17 shows how the aggregation service node (Aggregator) outage introduces duplicate data. The Aggregator manages the status of data consumption by storing the offset in upstream Kafka.

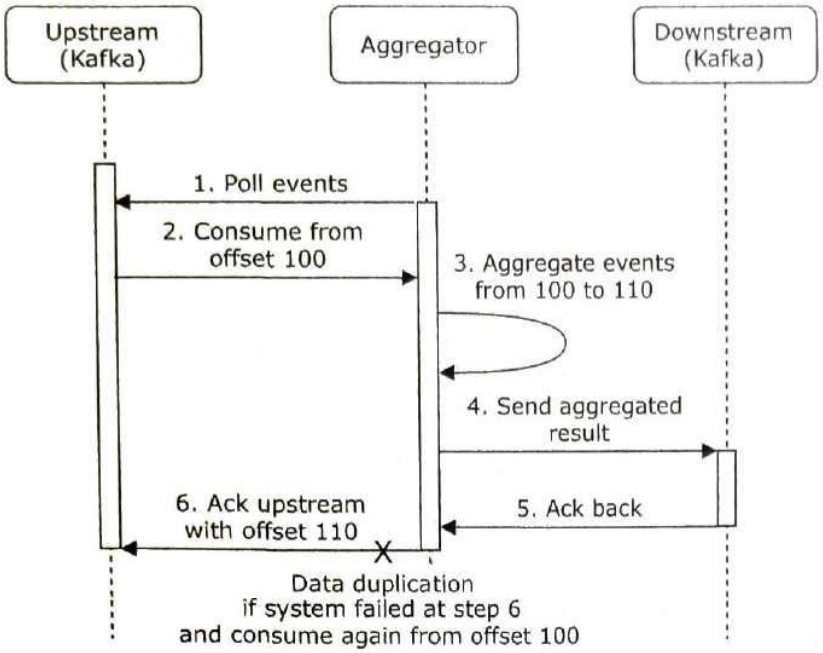


Figure 6.17: Duplicate data

If step 6 fails, perhaps due to Aggregator outage, events from 100 to 110 are already sent to the downstream, but the new offset 110 is not persisted in upstream Kafka. In this case, a new Aggregator would consume again from offset 100, even if those events are already processed, causing duplicate data.

The most straightforward solution (Figure 6.18) is to use external file storage, such as HDFS or S3, to record the offset. However, this solution has issues as well.

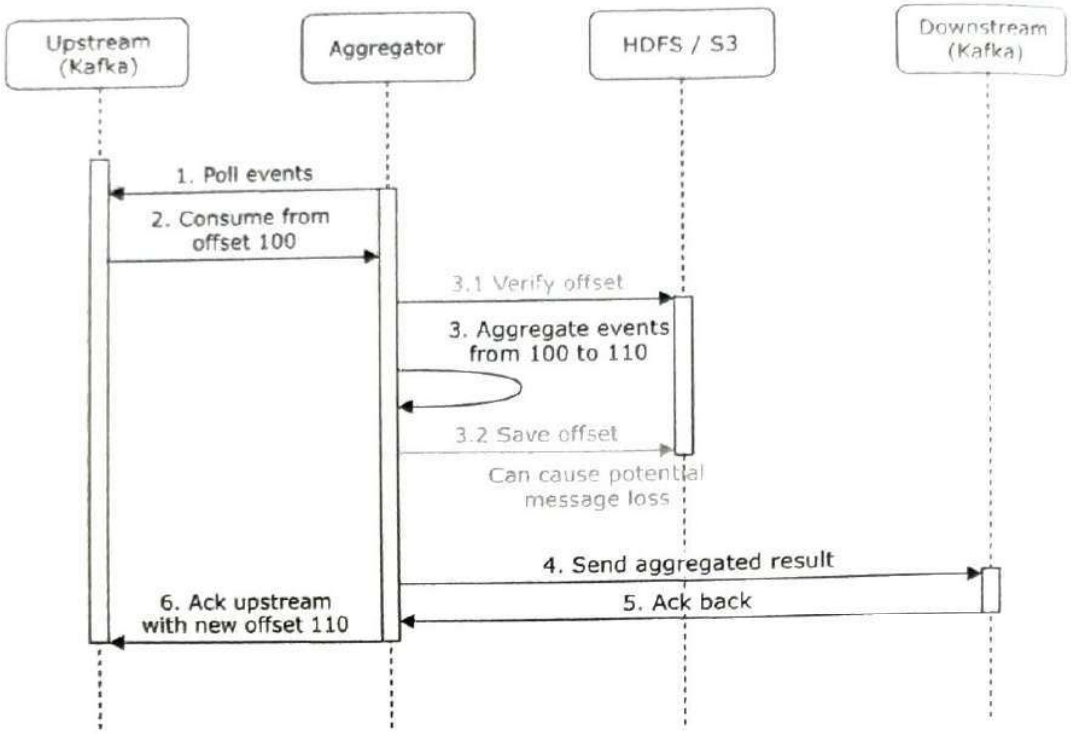


Figure 6.18: Record the offset

In step 3, the aggregator will process events from offset 100 to 110, only if the last offset stored in external storage is 100. If the offset stored in the storage is 110, the aggregator ignores events before offset 110.

But this design has a major problem: the offset is saved to HDFS or S3 (step 3.2) before the aggregation result is sent downstream. If step 4 fails due to Aggregator outage, events from 100 to 110 will never be processed by a newly brought up aggregator node, since the offset stored in external storage is 110.

To avoid data loss, we need to save the offset once we get an acknowledgment back from downstream. The updated design is shown in Figure 6.19.

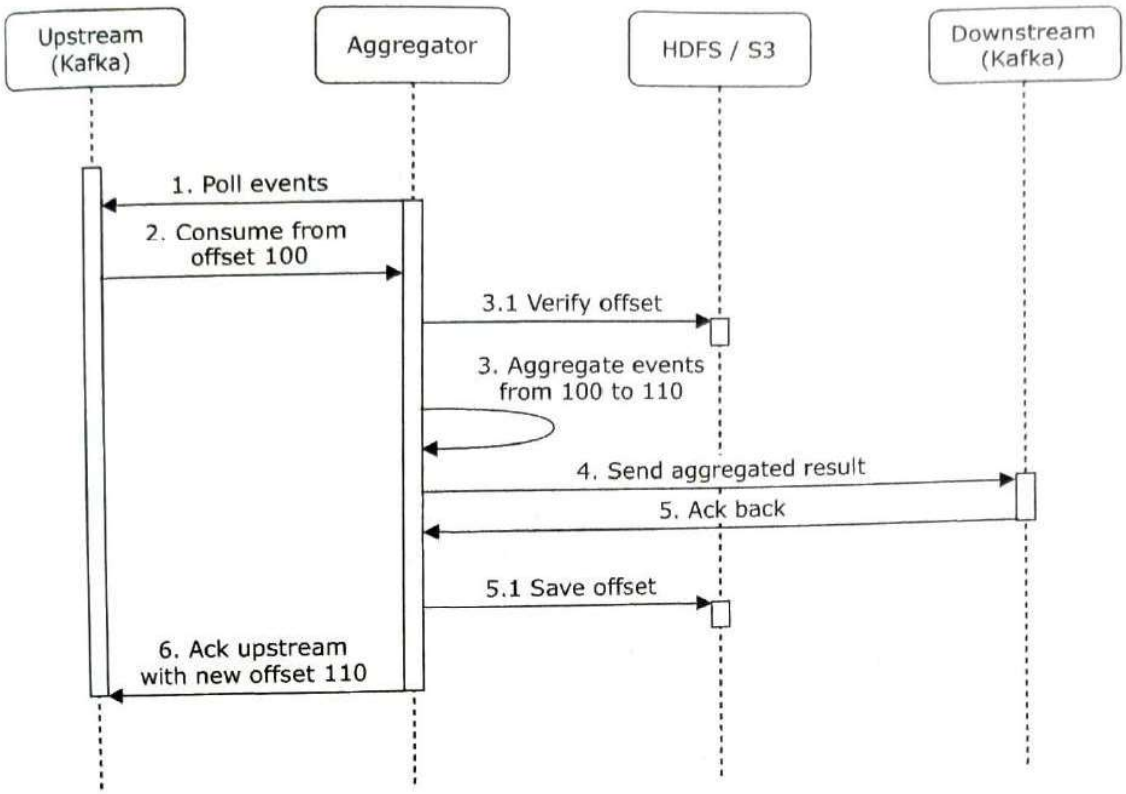


Figure 6.19: Save offset after receiving ack

In this design, if the Aggregator is down before step 5.1 is executed, events from 100 to 110 will be sent downstream again. To achieve exactly once processing, we need to put operations between step 4 to step 6 in one distributed transaction. A distributed transaction is a transaction that works across several nodes. If any of the operations fails, the whole transaction is rolled back.

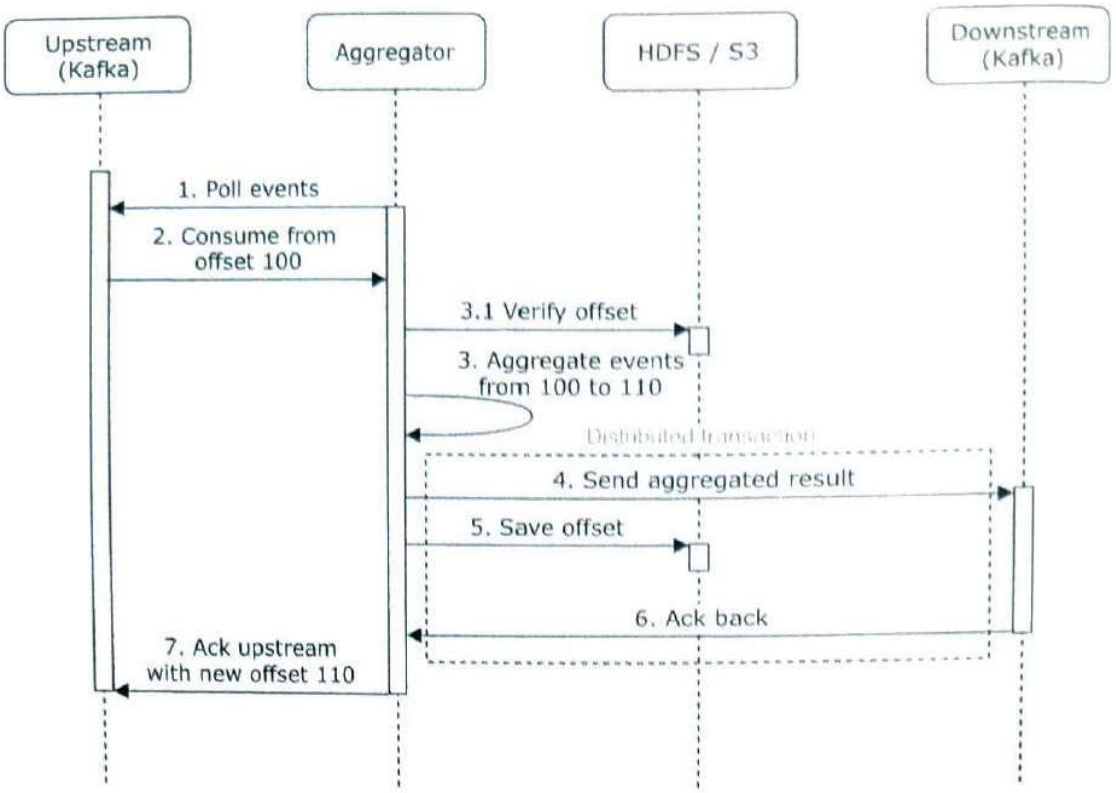


Figure 6.20: Distributed transaction

As you can see, it's not easy to dedupe data in large-scale systems. How to achieve exactly-once processing is an advanced topic. If you are interested in the details, please refer to reference material [9].

Scale the system

From the back-of-the-envelope estimation, we know the business grows 30% per year, which doubles traffic every 3 years. How do we handle this growth? Let's take a look.

Our system consists of three independent components: message queue, aggregation service, and database. Since these components are decoupled, we can scale each one independently.

Scale the message queue

We have already discussed how to scale the message queue extensively in the "Distributed Message Queue" chapter, so we'll only briefly touch on a few points.

Producers. We don't limit the number of producer instances, so the scalability of producers can be easily achieved.

Consumers. Inside a consumer group, the rebalancing mechanism helps to scale the consumers by adding or removing nodes. As shown in Figure 6.21, by adding two more consumers, each consumer only processes events from one partition.

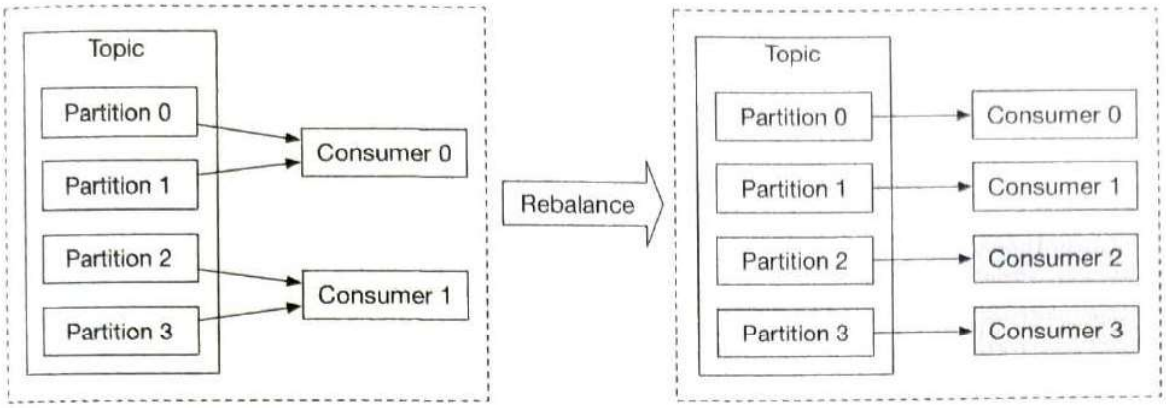


Figure 6.21: Add consumers

When there are hundreds of Kafka consumers in the system, consumer rebalance can be quite slow and could take a few minutes or even more. Therefore, if more consumers need to be added, try to do it during off-peak hours to minimize the impact.

Brokers

- **Hashing key**
  Using `ad_id` as hashing key for Kafka partition to store events from the same `ad_id` in the same Kafka partition. In this case, an aggregation service can subscribe to all events of the same `ad_id` from one single partition.
- **The number of partitions**
  If the number of partitions changes, events of the same `ad_id` might be mapped to a different partition. Therefore, it's recommended to pre-allocate enough partitions in advance, to avoid dynamically increasing the number of partitions in production.
- **Topic physical sharding**
  One single topic is usually not enough. We can split the data by geography (`topic_north_america`, `topic_europe`, `topic_asia`, etc.) or by business type (`topic_web_ads`, `topic_mobile_ads`, etc).
  - Pros: Slicing data to different topics can help increase the system throughput. With fewer consumers for a single topic, the time to rebalance consumer groups is reduced.
  - Cons: It introduces extra complexity and increases maintenance costs.

Scale the aggregation service

In the high-level design, we talked about the aggregation service being a map/reduce operation. Figure 6.22 shows how things are wired together.

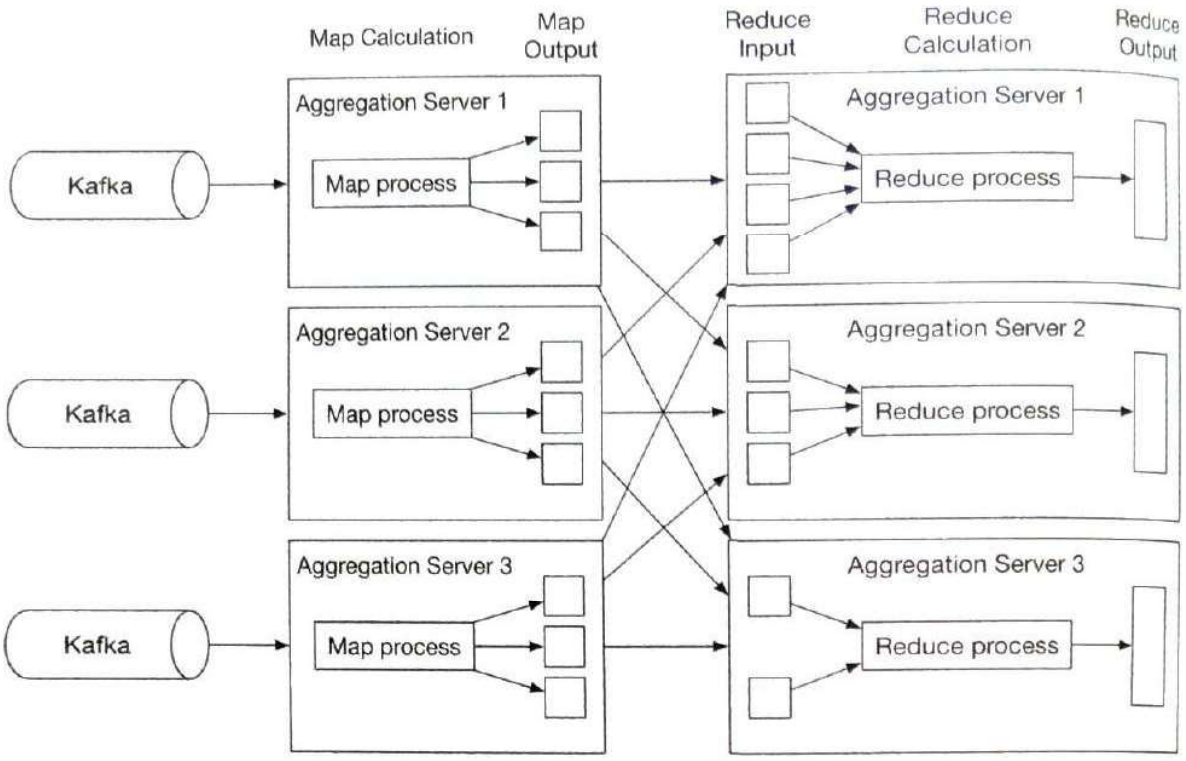


Figure 6.22: Aggregation service

If you are interested in the details, please refer to reference material [19]. Aggregation service is horizontally scalable by adding or removing nodes. Here is an interesting question; how do we increase the throughput of the aggregation service? There are two options.

Option 1: Allocate events with different `ad_ids` to different threads, as shown in Figure 6.23.

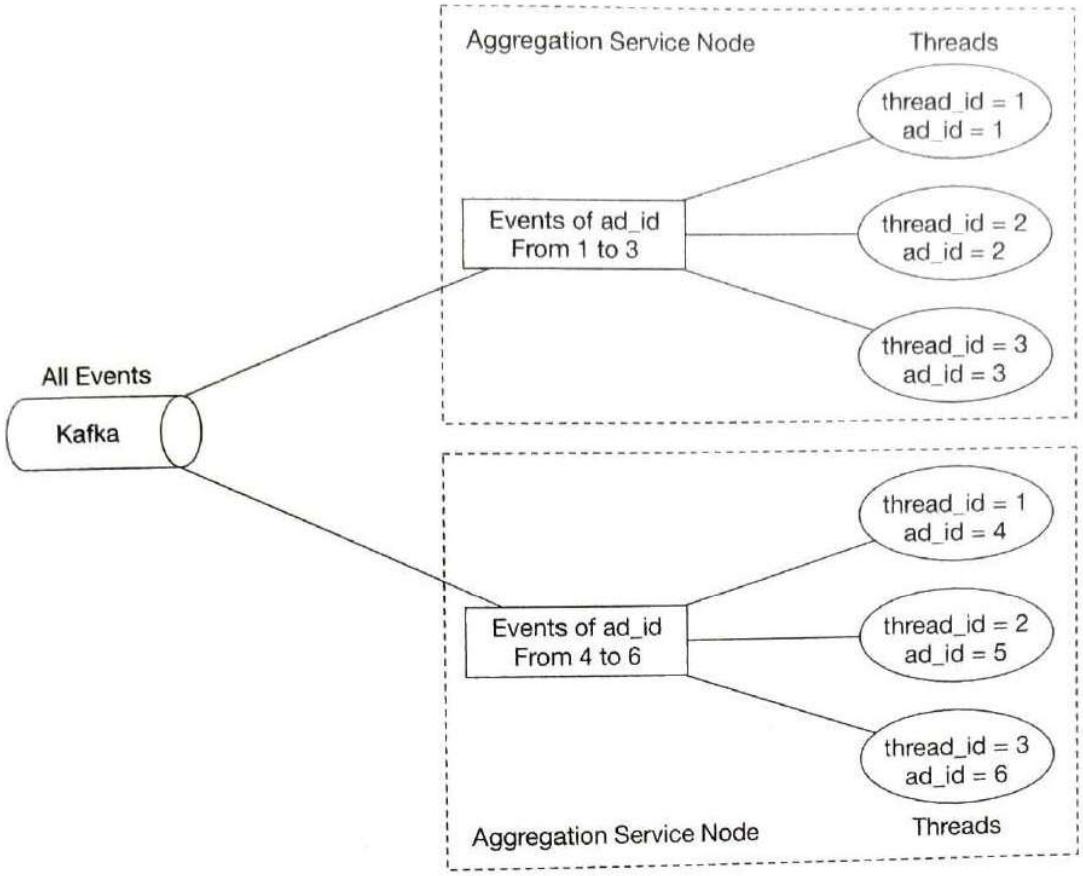


Figure 6.23: Multi-threading

Option 2: Deploy aggregation service nodes on resource providers like Apache Hadoop YARN [20]. You can think of this approach as utilizing multi-processing.

Option 1 is easier to implement and doesn't depend on resource providers. In reality, however, option 2 is more widely used because we can scale the system by adding more computing resources.

Scale the database

Cassandra natively supports horizontal scaling, in a way similar to consistent hashing.

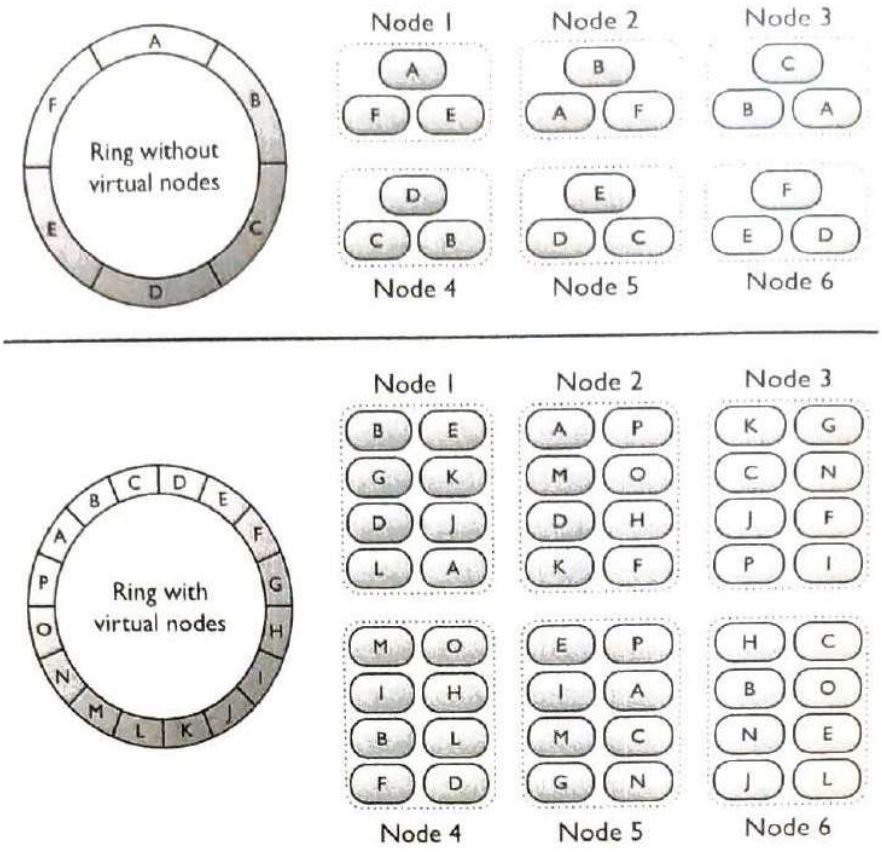


Figure 6.24: Virtual nodes [21]

Data is evenly distributed to every node with a proper replication factor. Each node saves its own part of the ring based on hashed value and also saves copies from other virtual nodes.

If we add a new node to the cluster, it automatically rebalances the virtual nodes among all nodes. No manual resharding is required. See Cassandra's official documentation for more details [21].

Hotspot issue

A shard or service that receives much more data than the others is called a hotspot. This occurs because major companies have advertising budgets in the millions of dollars and their ads are clicked more often. Since events are partitioned by `ad_id`, some aggregation service nodes might receive many more ad click events than others, potentially causing server overload.

This problem can be mitigated by allocating more aggregation nodes to process popular ads. Let's take a look at an example as shown in Figure 6.25. Assume each aggregation node can handle only 100 events.

1. Since there are 300 events in the aggregation node (beyond the capacity of a node can handle), it applies for extra resources through the resource manager.
2. The resource manager allocates more resources (for example, add two more aggregation nodes) so the original aggregation node isn't overloaded.

3. The original aggregation node split events into 3 groups and each aggregation node handles 100 events.
4. The result is written back to the original aggregate node.

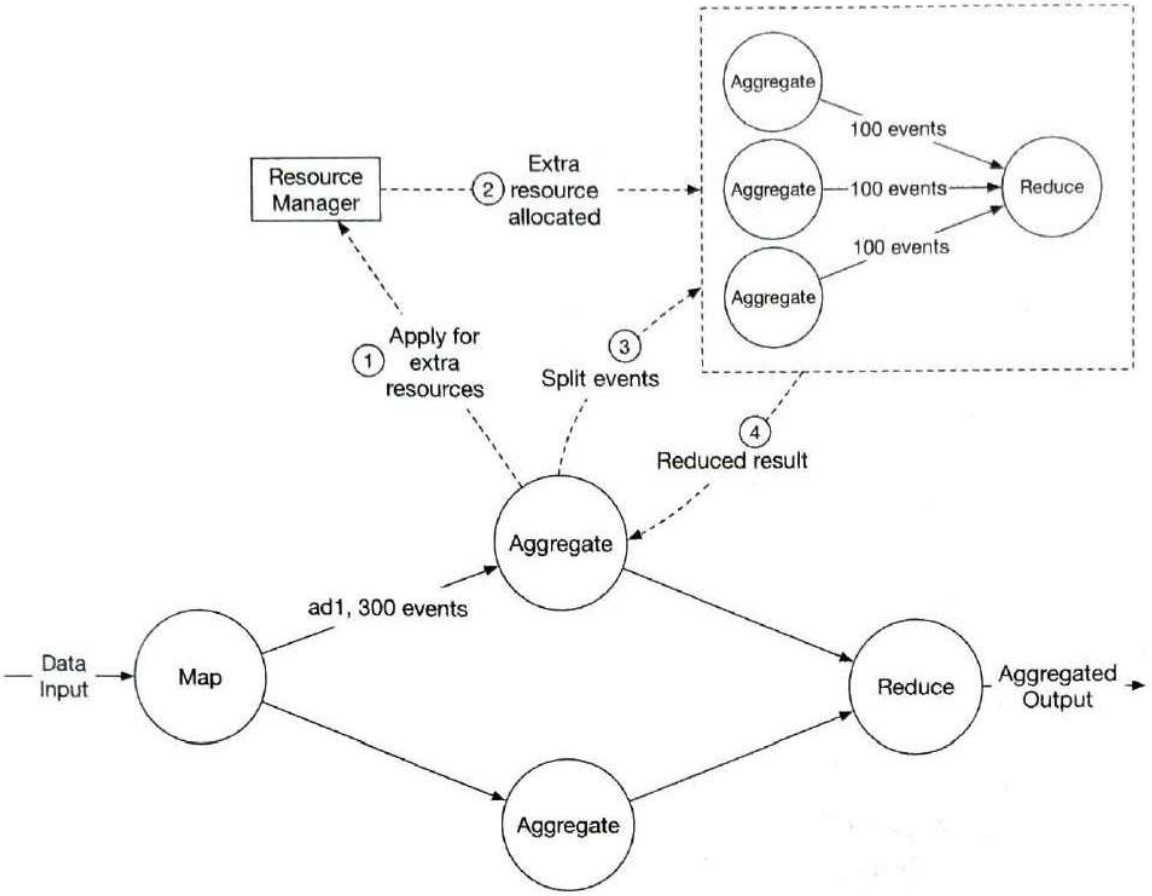


Figure 6.25: Allocate more aggregation nodes

There are more sophisticated ways to handle this problem, such as Global-Local Aggregation or Split Distinct Aggregation. For more information, please refer to [22].

Fault tolerance

Let's discuss the fault tolerance of the aggregation service. Since aggregation happens in memory, when an aggregation node goes down, the aggregated result is lost as well. We can rebuild the count by replaying events from upstream Kafka brokers.

Replaying data from the beginning of Kafka is slow. A good practice is to save the "system status" like upstream offset to a snapshot and recover from the last saved status. In our design, the "system status" is more than just the upstream offset because we need to store data like top N most clicked ads in the past M minutes.

Figure 6.26 shows a simple example of what the data looks like in a snapshot.

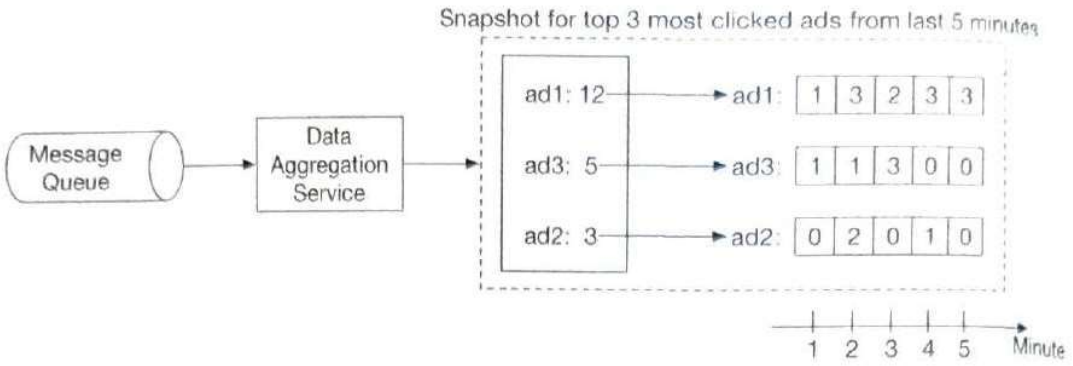


Figure 6.26: Data in a snapshot

With a snapshot, the failover process of the aggregation service is quite simple. If one aggregation service node fails, we bring up a new node and recover data from the latest snapshot (Figure 6.27). If there are new events that arrive after the last snapshot was taken, the new aggregation node will pull those data from the Kafka broker for replay.

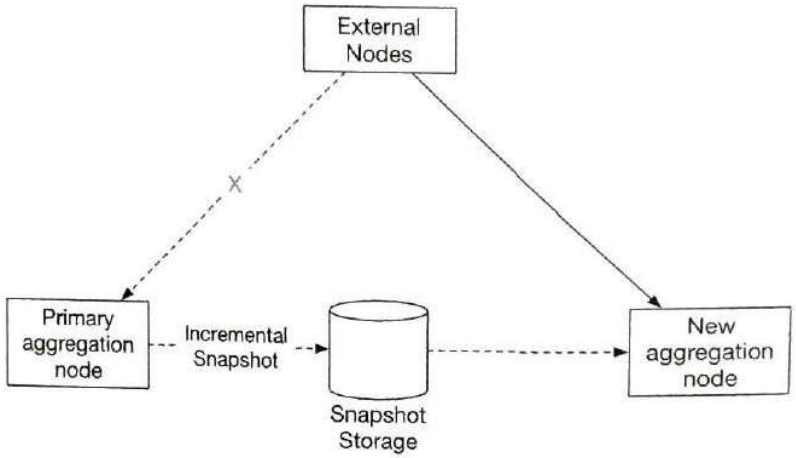


Figure 6.27: Aggregation node failover

Data monitoring and correctness

As mentioned earlier, aggregation results can be used for RTB and billing purposes. It's critical to monitor the system's health and to ensure correctness.

Continuous monitoring

Here are some metrics we might want to monitor:

- Latency. Since latency can be introduced at each stage, it's invaluable to track timestamps as events flow through different parts of the system. The differences between those timestamps can be exposed as latency metrics.
- Message queue size. If there is a sudden increase in queue size, we may need to add more aggregation nodes. Notice that Kafka is a message queue implemented as a distributed commit log, so we need to monitor the records-lag metrics instead.

- System resources on aggregation nodes: CPU, disk, JVM, etc.

Reconciliation

Reconciliation means comparing different sets of data in order to ensure data integrity. Unlike reconciliation in the banking industry, where you can compare your records with the bank's records, the result of ad click aggregation has no third-party result to reconcile with.

What we can do is to sort the ad click events by event time in every partition at the end of the day, by using a batch job and reconciling with the real-time aggregation result. If we have higher accuracy requirements, we can use a smaller aggregation window; for example, one hour. Please note, no matter which aggregation window is used, the result from the batch job might not match exactly with the real-time aggregation result, since some events might arrive late (see "Time" section on page 175).

Figure 6.28 shows the final design diagram with reconciliation support.

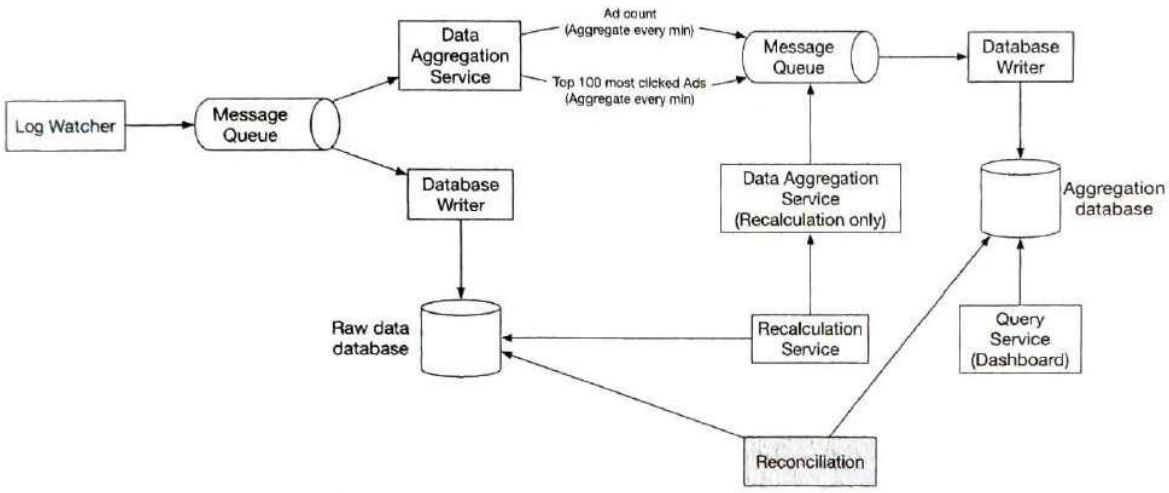


Figure 6.28: Final design

Alternative design

In a generalist system design interview, you are not expected to know the internals of different pieces of specialized software used in a big data pipeline. Explaining your thought process and discussing trade-offs is very important, which is why we propose a generic solution. Another option is to store ad click data in Hive, with an ElasticSearch layer built for faster queries. Aggregation is usually done in OLAP databases such as ClickHouse [23] or Druid [24]. Figure 6.29 shows the architecture.

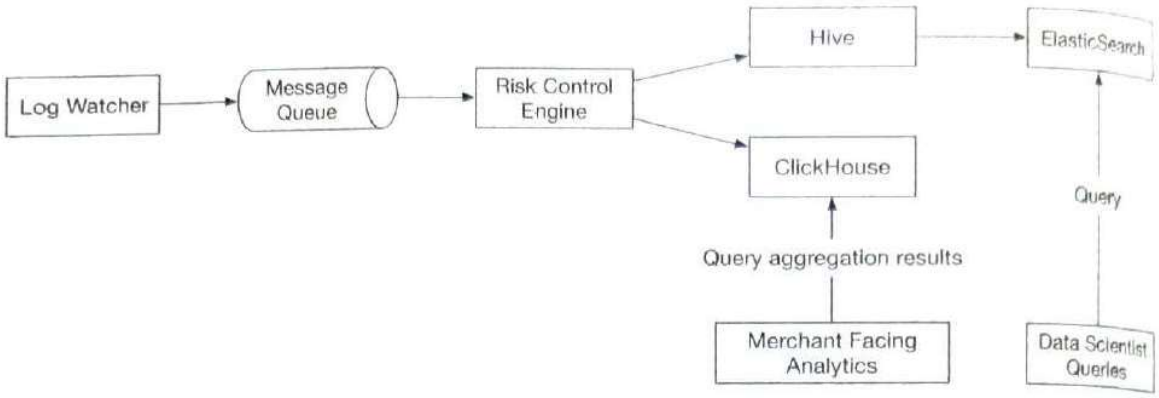


Figure 6.29: Alternative design

For more detail on this, please refer to reference material [25].

Step 4 - Wrap Up

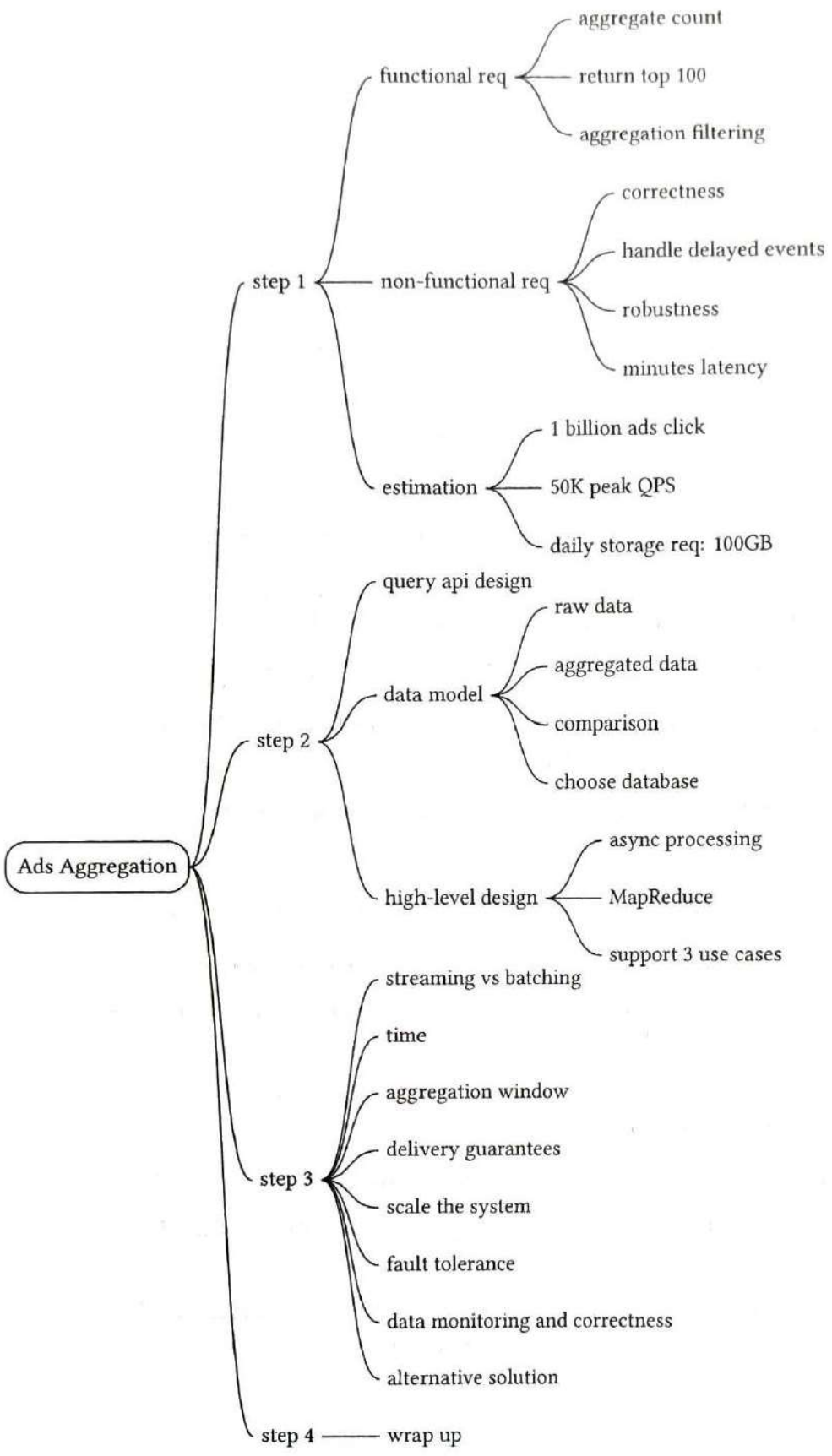
In this chapter, we went through the process of designing an ad click event aggregation system at the scale of Facebook or Google. We covered:

- Data model and API design.
- Use MapReduce paradigm to aggregate ad click events.
- Scale the message queue, aggregation service, and database.
- Mitigate hotspot issue.
- Monitor the system continuously.
- Use reconciliation to ensure correctness.
- Fault tolerance.

The ad click event aggregation system is a typical big data processing system. It will be easier to understand and design if you have prior knowledge or experience with industry-standard solutions such as Apache Kafka, Apache Flink, or Apache Spark.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] Clickthrough rate (CTR): Definition. https://support.google.com/google-ads/answer/2615875?hl=en.

[2] Conversion rate: Definition. https://support.google.com/google-ads/answer/2684489?hl=en.

[3] OLAP functions. https://docs.oracle.com/database/121/OLAXS/olap_functions.htm#OLAXS169.

[4] Display Advertising with Real-Time Bidding (RTB) and Behavioural Targeting. https://arxiv.org/pdf/1610.03013.pdf.

[5] LanguageManual ORC. https://cwiki.apache.org/confluence/display/hive/languagemanual+orc.

[6] Parquet. https://databricks.com/glossary/what-is-parquet.

[7] What is avro. https://www.ibm.com/topics/avro.

[8] Big Data. https://www.datakwery.com/techniques/big-data/.

[9] An Overview of End-to-End Exactly-Once Processing in Apache Flink. https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html.

[10] DAG model. https://en.wikipedia.org/wiki/Directed_acyclic_graph.

[11] Understand star schema and the importance for Power BI. https://docs.microsoft.com/en-us/power-bi/guidance/star-schema.

[12] Martin Kleppmann. *Designing Data-Intensive Applications.* O'Reilly Media, 2017.

[13] Apache Flink. https://flink.apache.org/.

[14] Lambda architecture. https://databricks.com/glossary/lambda-architecture.

[15] Kappa architecture. https://hazelcast.com/glossary/kappa-architecture/.

[16] Martin Kleppmann. Stream Processing. In *Designing Data-Intensive Applications.* O'Reilly Media, 2017.

[17] End-to-end Exactly-once Aggregation Over Ad Streams. https://www.youtube.com/watch?v=hzxytnPcAUM.

[18] Ad traffic quality. https://www.google.com/ads/adtrafficquality/.

[19] Understanding MapReduce in Hadoop. https://www.section.io/engineering-education/understanding-map-reduce-in-hadoop/.

[20] Flink on Apache Yarn. https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/deployment/resource-providers/yarn/.

[21] How data is distributed across a cluster (using virtual nodes). https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/architecture/archDataDistributeDistribute.html.

[22] Flink performance tuning. https://nightlies.apache.org/flink/flink-docs-master/docs/dev/table/tuning/.

[23] ClickHouse. https://clickhouse.com/.

[24] Druid. https://druid.apache.org/.

[25] Real-Time Exactly-Once Ad Event Processing with Apache Flink, Kafka, and Pinot. https://eng.uber.com/real-time-exactly-once-ad-event-processing/.

7 Hotel Reservation System

In this chapter, we design a hotel reservation system for a hotel chain such as Marriott International. The design and techniques used in this chapter are also applicable to other popular booking-related interview topics:

- Design Airbnb
- Design a flight reservation system
- Design a movie ticket booking system

Step 1 - Understand the Problem and Establish Design Scope

The hotel reservation system is complicated and its components vary based on business use cases. Before diving into the design, we should ask the interviewer clarification questions to narrow down the scope.

Candidate: What is the scale of the system?
Interviewer: Let's assume we are building a website for a hotel chain that has 5,000 hotels and 1 million rooms in total.

Candidate: Do customers pay when they make reservations or when they arrive at the hotel?
Interviewer: For simplicity, they pay in full when they make reservations.

Candidate: Do customers book hotel rooms through the hotel's website only? Do we need to support other reservation options such as phone calls?
Interviewer: Let's assume people could book a hotel room through the hotel website or app.

Candidate: Can customers cancel their reservations?
Interviewer: Yes.

Candidate: Are there any other things we need to consider?
Interviewer: Yes, we allow 10% overbooking. In case you do not know, overbooking means the hotel will sell more rooms than they actually have. Hotels do this in anticipation that some customers will cancel their reservations.

Candidate: Since we have limited time, I assume the hotel room search is not in scope. We focus on the following features.

- Show the hotel-related page.
- Show the hotel room-related detail page.
- Reserve a room.
- Admin panel to add/remove/update hotel or room info.
- Support the overbooking feature.

Interviewer: Sounds good.

Interviewer: One more thing, hotel prices change dynamically. The price of a hotel room depends on how full the hotel is expected to be on a given day. For this interview, we can assume the price could be different each day.
Candidate: I'll keep this in mind.

Next, you might want to talk about the most important non-functional requirements.

Non-functional requirements

- Support high concurrency. During peak season or big events, some popular hotels may have a lot of customers trying to book the same room.
- Moderate latency. It's ideal to have a fast response time when a user makes the reservation, but it's acceptable if the system takes a few seconds to process a reservation request.

Back-of-the-envelope estimation

- 5,000 hotels and 1 million rooms in total.
- Assume 70% of the rooms are occupied and the average stay duration is 3 days.
- Estimated daily reservations: $\frac{1 \text{ million} \times 0.7}{3} = 233{,}333$ (rounding up to $\sim 240{,}000$)
- Reservations per second = $\frac{240{,}000}{10^5 \text{ seconds in a day}} = \sim 3$. As we can see, the average reservation transaction per second (TPS) is not high.

Next, let's do a rough calculation of the QPS of all pages in the system. There are three steps in a typical customer flow:

1. View hotel/room detail page. Users browse this page (query).
2. View the booking page. Users can confirm the booking details, such as dates, number of guests, payment information before booking (query).
3. Reserve a room. Users click on the "book" button to book the room and the room is reserved (transaction).

Let's assume around 10% of the users reach the next step and 90% of users drop off the flow before reaching the final step. We can also assume that no prefetching feature (prefetching the content before the user reaches the next step) is implemented. Figure 7.1 shows a rough estimation of what the QPS looks like for different steps. We know the final reservation TPS is 3 so we can work backward along the funnel. The QPS of the order confirmation page is 30 and the QPS for the detail page is 300.

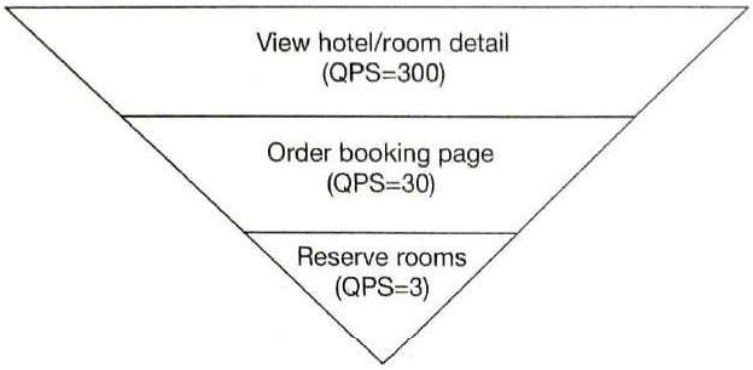


Figure 7.1: QPS distribution

Step 2 - Propose High-level Design and Get Buy-in

In this section, we'll discuss:

- API design
- Data models
- High-level design

API design

We explore the API design for the hotel reservation system. The most important APIs are listed below using the RESTful conventions.

Note that this chapter focuses on the design of a hotel reservation system. For a complete hotel website, the design needs to provide intuitive features for customers to search for rooms based on a large array of criteria. The APIs for these search features, while important, are not technically challenging. They are out of scope for this chapter.

Hotel-related APIs

API	Detail
GET /v1/hotels/ID	Get detailed information about a hotel.
POST /v1/hotels	Add a new hotel. This API is only available to hotel staff.
PUT /v1/hotels/ID	Update hotel information. This API is only available to hotel staff.
DELETE /v1/hotels/ID	Delete a hotel. This API is only available to hotel staff.

Table 7.1: Hotel-related APIs

Room-related APIs

API	Detail
GET /v1/hotels/ID/rooms/ID	Get detailed information about a room.
POST /v1/hotels/ID/rooms	Add a room. This API is only available to hotel staff.
PUT /v1/hotels/ID/rooms/ID	Update room information. This API is only available to hotel staff.
DELETE /v1/hotels/ID/rooms/ID	Delete a room. This API is only available to hotel staff.

Table 7.2: Hotel-related APIs

Reservation related APIs

API	Detail
GET /v1/reservations	Get the reservation history of the logged-in user.
GET /v1/reservations/ID	Get detailed information about a reservation.
POST /v1/reservations	Make a new reservation.
DELETE /v1/reservations/ID	Cancel a reservation.

Table 7.3: Reservation-related APIs

Making a new reservation is a very important feature. The request parameters of making a new reservation (`POST /v1/reservations`) could look like this.

```
{
  "startDate": "2021-04-28",
  "endDate":"2021-04-30",
  "hotelID":"245",
  "roomID":"U12354673389",
  "reservationID":"13422445"
}
```

Please note `reservationID` is used as the idempotency key to prevent double booking. Double booking means multiple reservations are made for the same room on the same day. The details are explained in the "Concurrency issue" section on page 206.

Data model

Before we decide which database to use, let's take a close look at the data access patterns. For the hotel reservation system, we need to support the following queries:

Query 1: View detailed information about a hotel.

Query 2: Find available types of rooms given a date range.

Query 3: Record a reservation.

Query 4: Look up a reservation or past history of reservations.

From the back-of-the-envelope estimation, we know the scale of the system is not large but we need to prepare for traffic surges during big events. With these requirements in mind, we choose a relational database because:

- A relational database works well with read-heavy and write less frequently workflow. This is because the number of users who visit the hotel website/apps is a few orders of magnitude higher than those who actually make reservations. NoSQL databases are generally optimized for writes and the relational database works well enough for read-heavy workflow.
- A relational database provides ACID (atomicity, consistency, isolation, durability) guarantees. ACID properties are important for a reservation system. Without those properties, it's not easy to prevent problems such as negative balance, double charge, double reservations, etc. ACID properties make application code a lot simpler and make the whole system easier to reason about. A relational database usually provides these guarantees.
- A relational database can easily model the data. The structure of the business data is very clear and the relationship between different entities (`hotel`, `room`, `room_type`, etc) is stable. This kind of data model is easily modeled by a relational database.

Now that we have chosen the relational database as our data store, let's explore the schema design. Figure 7.2 shows a straightforward schema design and it is the most natural way for many candidates to model the hotel reservation system.

Figure 7.2: Database schema

Most attributes are self-explanatory and we will only explain the `status` field in the reservation table. The `status` field can be in one of these states: `pending`, `paid`, `refunded`, `canceled`, `rejected`. The state machine is shown in Figure 7.3.

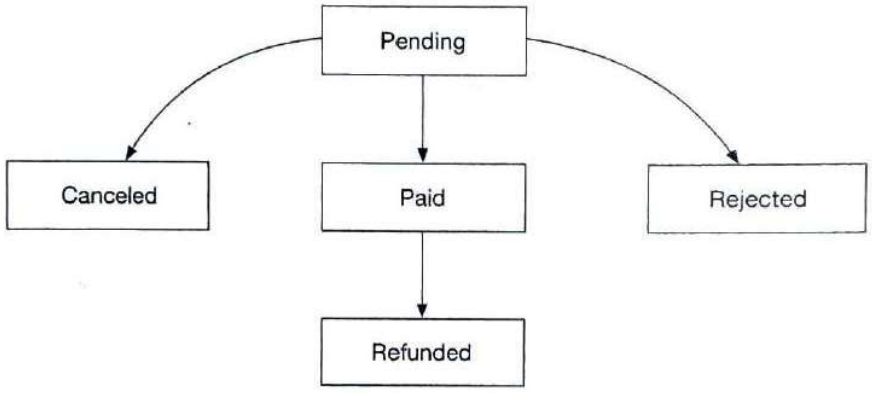


Figure 7.3: Reservation status

This schema design has a major issue. This data model works for companies like Airbnb as `room_id` (might be called `listing_id`) is given when users make reservations. However, this isn't the case for hotels. A user actually reserves **a type of room** in a given hotel instead of a specific room. For instance, a room type can be a standard room, king-size room, queen-size room with two queen beds, etc. Room numbers are given when the guest checks in and not at the time of the reservation. We need to update our data model to reflect this new requirement. See "Improved data model" in the deep dive section on

page 203 for more details.

High-level design

We use the microservice architecture for this hotel reservation system. Over the past few years, microservice architecture has gained great popularity. Companies that use microservice include Amazon, Netflix, Uber, Airbnb, Twitter, etc. If you want to learn more about the benefits of a microservice architecture, you can check out some good resources [1] [2] .

Our design is modeled with the microservice architecture and the high-level design diagram is shown in Figure 7.4.

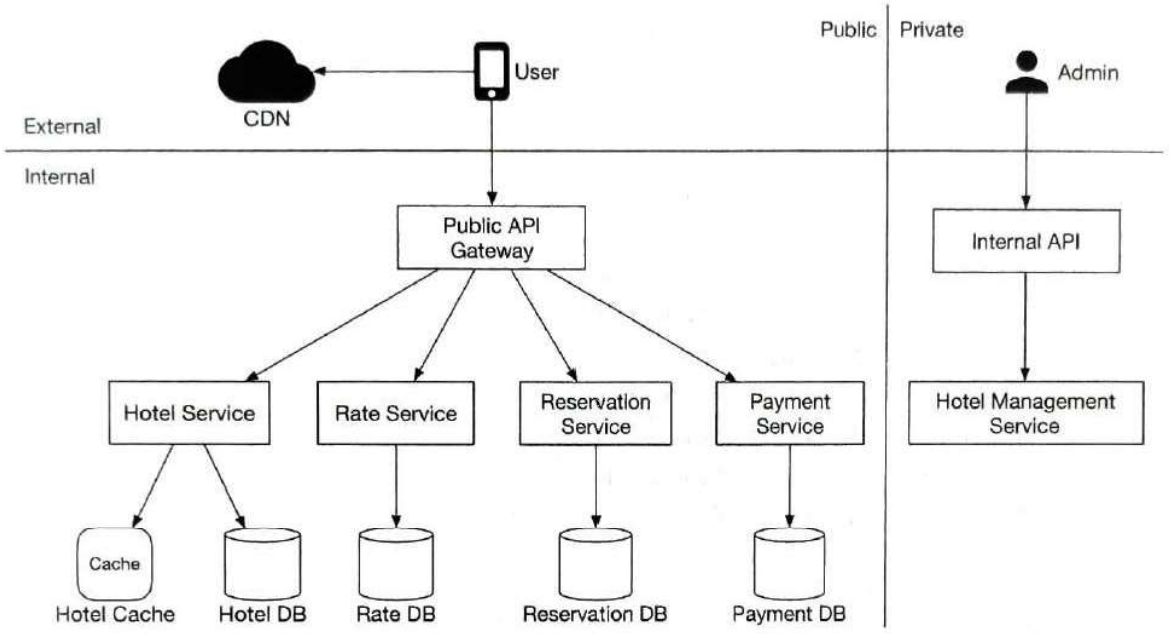


Figure 7.4: High-level design

We will briefly go over each component of the system from top to bottom.

- User: a user books a hotel room on their mobile phone or computer.
- Admin (hotel staff): authorized hotel staff perform administrative operations such as refunding a customer, canceling a reservation, updating room information, etc.
- CDN (content delivery network): for better load time, CDN is used to cache all static assets, including javascript bundles, images, videos, HTML, etc.
- Public API Gateway: this is a fully managed service that supports rate limiting, authentication, etc. The API gateway is configured to direct requests to specific services based on the endpoints. For example, requests to load the hotel homepage are directed to the hotel service and requests to book a hotel room are routed to the reservation service.
- Internal APIs: those APIs are only available for authorized hotel staff. They are accessible through internal software or websites. They are usually further protected

by a VPN (virtual private network).

- Hotel Service: this provides detailed information on hotels and rooms. Hotel and room data are generally static, so can be easily cached.
- Rate Service: this provides room rates for different future dates. An interesting fact about the hotel industry is that the price of a room depends on how full the hotel is expected to be for a given day.
- Reservation Service: receives reservation requests and reserves the hotel rooms. This service also tracks room inventory as rooms are reserved or reservations are canceled.
- Payment Service: executes payment from a customer and updates the reservation status to `paid` once a payment transaction succeeds, or `rejected` if the transaction fails.
- Hotel Management Service: only available to authorized hotel staff. Hotel staff are eligible to use the following features: view the record of an upcoming reservation, reserve a room for a customer, cancel a reservation, etc.

For clarity, Figure 7.4 omits many arrows of interactions between microservices. For example, as shown in Figure 7.5, there should be an arrow between Reservation service and Rate service. Reservation service queries Rate service for room rates. This is used to compute the total room charge for a reservation. Another example is that there should be many arrows connecting the Hotel Management Service with most of the other services. When an admin makes changes via Hotel Management Service, the requests are forwarded to the actual service owning the data, to handle the changes.

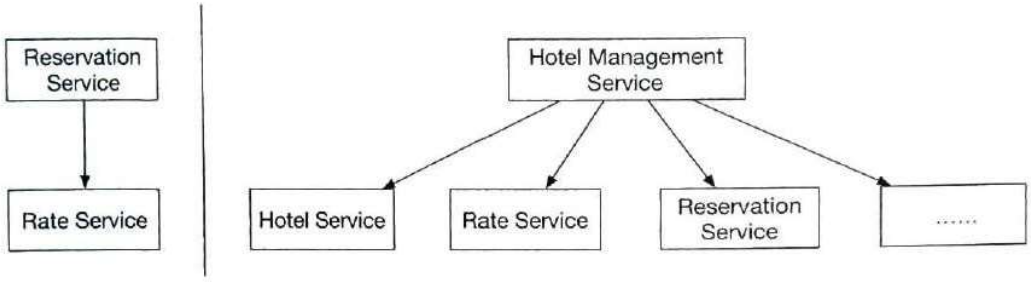


Figure 7.5: Connections between services

For production systems, inter-service communication often employs a modern and high-performance remote procedure call (RPC) framework like gPRC. There are many benefits to using such frameworks. To learn more about gPRC in particular, check out [3].

Step 3 - Design Deep Dive

Now we've talked about the high-level design, let's go deeper into the following.

- Improved data model
- Concurrency issues

- Scaling the system
- Resolving data inconsistency in the microservice architecture

Improved data model

As mentioned in the high-level design, when we reserve a hotel room, we actually reserve a type of room, as opposed to a specific room. What do we need to change about the API and schema to accommodate this?

For the reservation API, `roomID` is replaced by `roomTypeID` in the request parameter. The API to make a reservation looks like this:

```
POST /v1/reservations
```

Request parameters:

```
{
  "startDate": "2021-04-28",
  "endDate":"2021-04-30",
  "hotelID":"245",
  "roomTypeID":"12354673389",
  "reservationID":"13422445"
}
```

The updated schema is shown in Figure 7.6.

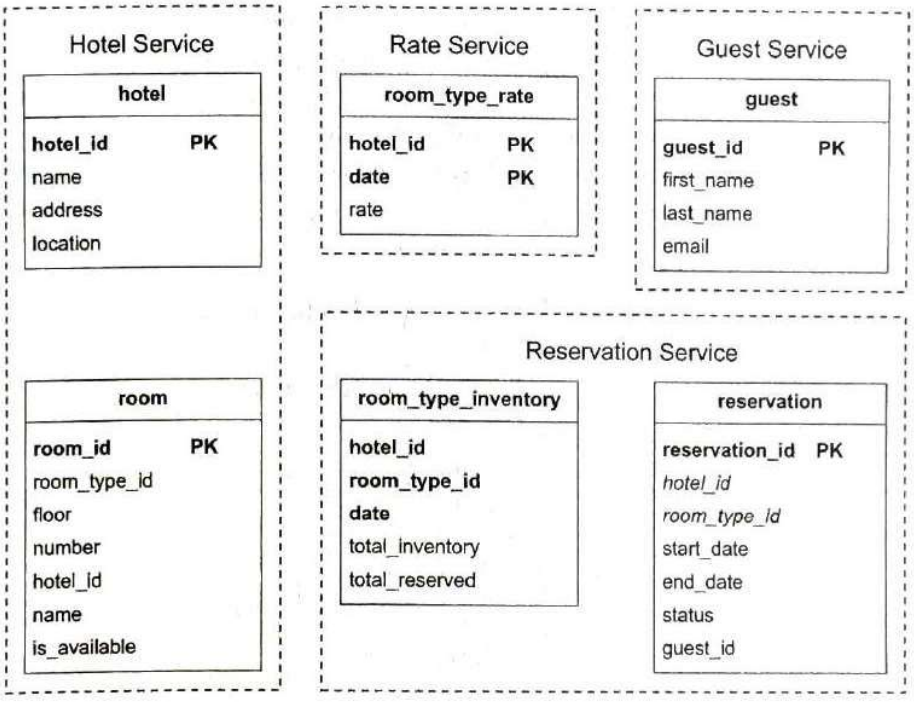


Figure 7.6: Updated schema

We'll briefly go over some of the most important tables.

room: contains information regarding a room.

room_type_rate: stores price data for a specific room type, for future dates.

reservation: records guest reservation data.

room_type_inventory: stores inventory data about hotel rooms. This table is very important for the reservation system, so let's take a close look at each column.

- hotel_id: ID of the hotel
- room_type_id: ID of a room type.
- date: a single date.
- total_inventory: the total number of rooms minus those that are temporarily taken off the inventory. Some rooms might be taken off from the market for maintenance.
- total_reserved: the total number of rooms booked for the specified hotel_id, room_type_id, and date.

There are other ways to design the room_type_inventory table, but having one row per date makes managing reservations within a date range and queries easy. As shown in Figure 7.6, (hotel_id, room_type_id, date) is the composite primary key. The rows of the table are pre-populated by querying the inventory data across all future dates within 2 years. We have a scheduled daily job that pre-populates inventory data when the dates advance further.

Now that we've finalized the schema design, let's do some estimation about the storage volume. As mentioned in the back-of-the-envelope estimation, we have 5,000 hotels. Assume each hotel has 20 types of rooms. That's (5,000 hotels $\times$20 types of rooms $\times$2 years $\times$365 days) = 73 million rows. 73 million is not a lot of data and a single database is enough to store the data. However, a single server means a single point of failure. To achieve high availability, we could set up database replications across multiple regions or availability zones.

Table 7.4 shows the sample data of the room_type_inventory table.

hotel_id	room_type_id	date	total_inventory	total_reserved
211	1001	2021-06-01	100	80
211	1001	2021-06-02	100	82
211	1001	2021-06-03	100	86
211	1001	...	...	
211	1001	2023-05-31	100	0
211	1002	2021-06-01	200	164
2210	101	2021-06-01	30	23
2210	101	2021-06-02	30	25

Table 7.4: Sample data of the room_type_inventory table

The room_type_inventory table is utilized to check if a customer can reserve a specific type of room or not. The input and output for a reservation might look like this:

- Input: `startDate` (2021-07-01), `endDate` (2021-07-03), `roomTypeId`, `hotelId`, `numberOfRoomsToReserve`
- Output: `True` if the specified type of room has inventory and users can book it. Otherwise, it returns `False`.

From the SQL perspective, it contains the following two steps:

1. Select rows within a date range

```
SELECT date, total_inventory, total_reserved
FROM room_type_inventory
WHERE room_type_id = ${roomTypeId} AND hotel_id = ${
hotelId}
AND date between ${startDate} and ${endDate}
```

   This query returns data like this:

date	total_inventory	total_reserved
2021-07-01	100	97
2021-07-02	100	96
2021-07-03	100	95

Table 7.5: Hotel inventory

2. For each entry, the application checks the condition below:

```
if ((total_reserved + ${numberOfRoomsToReserve}) <=
total_inventory)
```

If the condition returns `True` for all entries, it means there are enough rooms for each date within the date range.

One of the requirements is to support 10% overbooking. With the new schema, it is easy to implement:

```
if ((total_reserved + ${numberOfRoomsToReserve}) <= 110% *
total_inventory)
```

At this point, the interviewer might ask a follow-up question: "if the reservation data is too large for a single database, what would you do?" There are a few strategies:

- Store only current and future reservation data. Reservation history is not frequently accessed. So they can be archived and some can even be moved to cold storage.
- Database sharding. The most frequent queries include making a reservation or looking up a reservation by name. In both queries, we need to choose the hotel first, meaning `hotel_id` is a good sharding key. The data can be sharded by `hash(hotel_id) % number_of_servers`.

Concurrency issues

Another important problem to look at is double booking. We need to solve two problems:

1. The same user clicks on the book button multiple times.
2. Multiple users try to book the same room at the same time.

Let's take a look at the first scenario. As shown in Figure 7.7, two reservations are made.

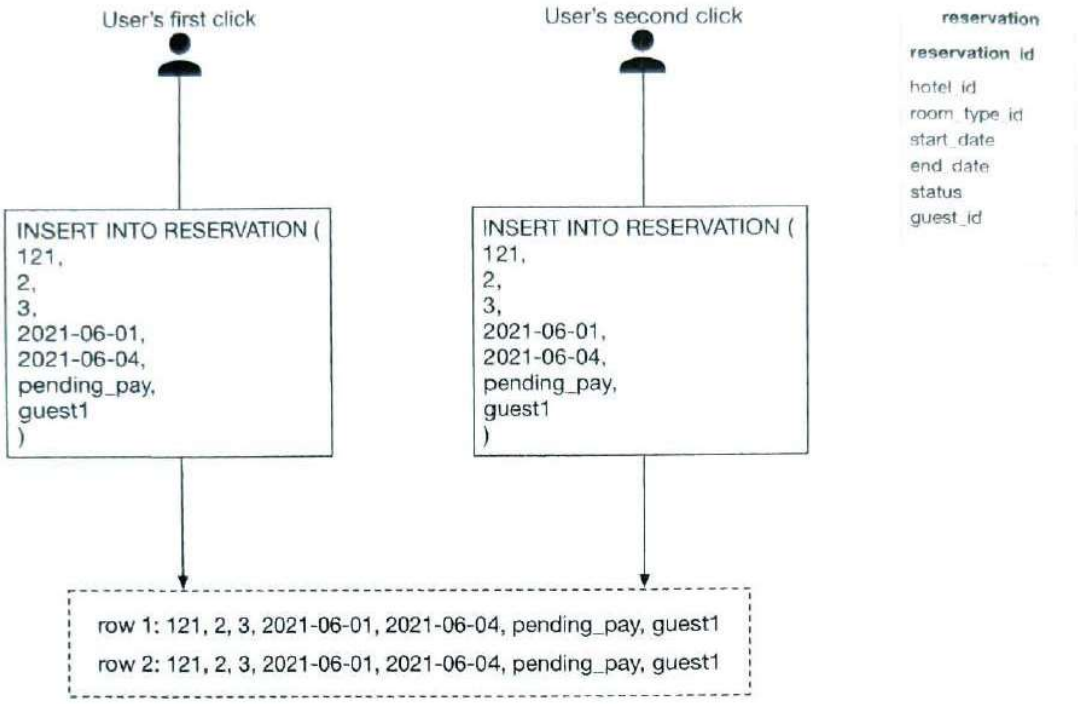


Figure 7.7: Two reservations are made

There are two common approaches to solve this problem:

- Client-side implementation. A client can gray out, hide or disable the "submit" button once a request is sent. This should prevent the double-clicking issue most of the time. However, this approach is not very reliable. For example, users can disable javascript, thereby bypassing the client check.
- Idempotent APIs. Add an idempotency key in the reservation API request. An API call is idempotent if it produces the same result no matter how many times it is called. Figure 7.8 shows how to use the idempotency key `reservation_id` to avoid the double-reservation issue. The detailed steps are explained below.

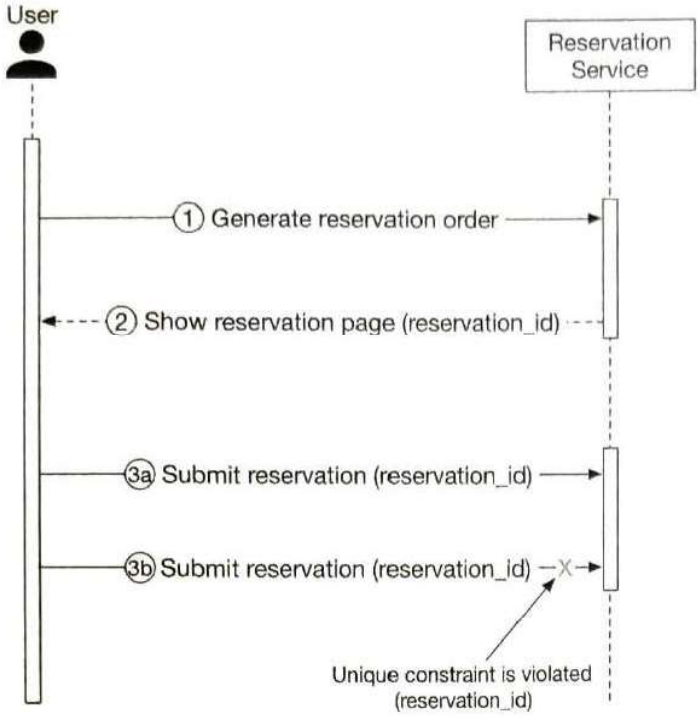


Figure 7.8: Unique constraint

1. Generate a reservation order. After a customer enters detailed information about the reservation (room type, check-in date, check-out date, etc) and clicks the "continue" button, a reservation order is generated by the reservation service.

2. The system generates a reservation order for the customer to review. The unique `reservation_id` is generated by a globally unique ID generator and returned as part of the API response. The UI of this step might look like this:

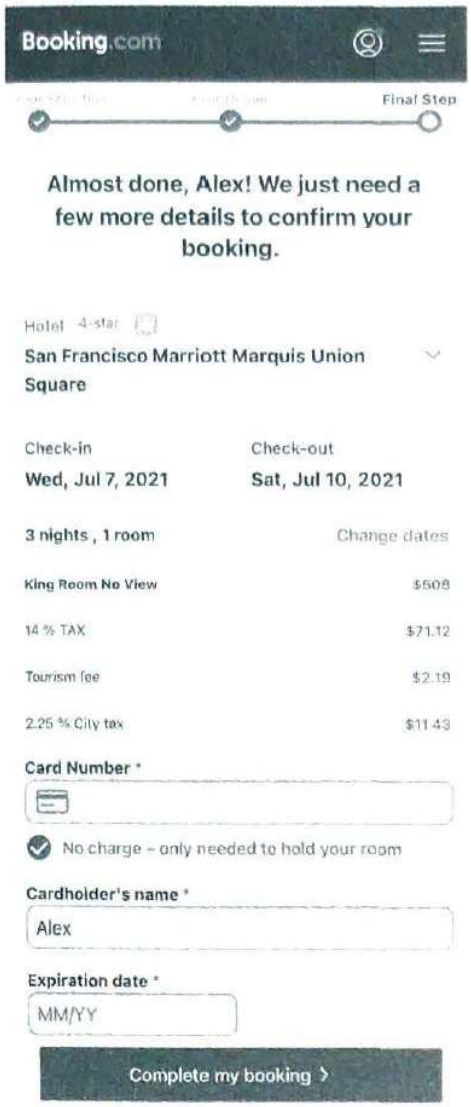


Figure 7.9: Confirmation page (Source: [4])

3a. Submit reservation 1. The `reservation_id` is included as part of the request. It is the primary key of the reservation table (Figure 7.6). Please note that the idempotency key doesn't have to be the `reservation_id`. We choose `reservation_id` because it already exists and works well for our design.

3b. If a user clicks the "Complete my booking" button a second time, reservation 2 is submitted. Because `reservation_id` is the primary key of the reservation table, we can rely on the unique constraint of the key to ensure no double reservation happens.

Figure 7.10 explains why double reservation can be avoided.

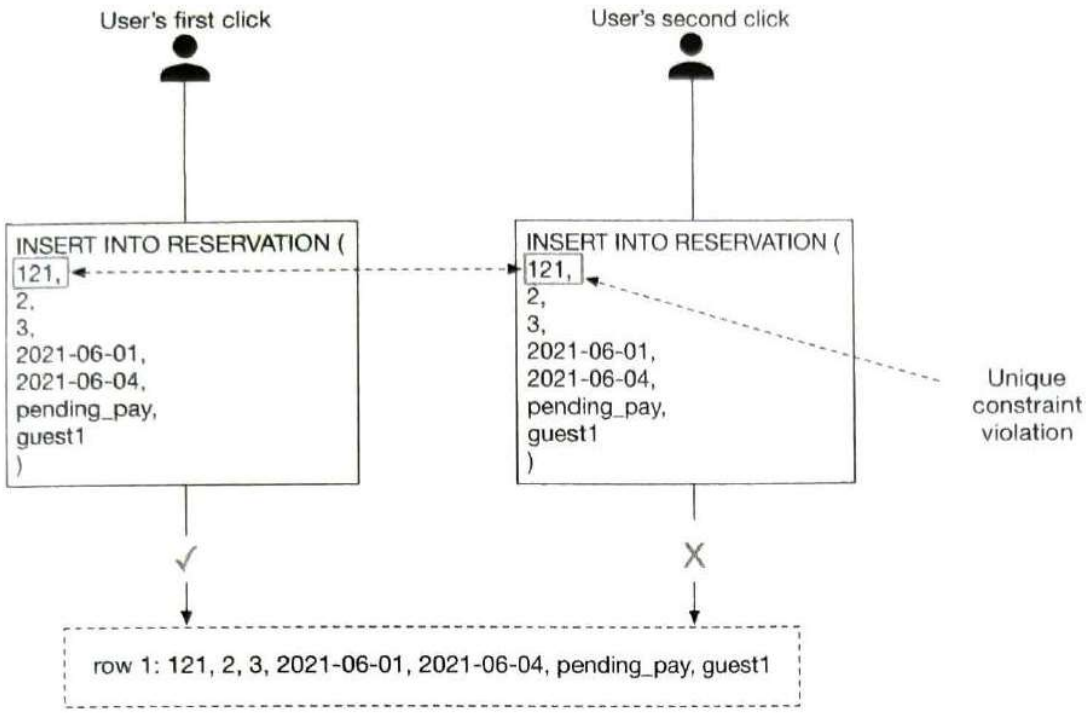


Figure 7.10: Unique constraint violation

Scenario 2: what happens if multiple users book the same type of room at the same time when there is only one room left? Let's consider the scenario as shown in Figure 7.11.

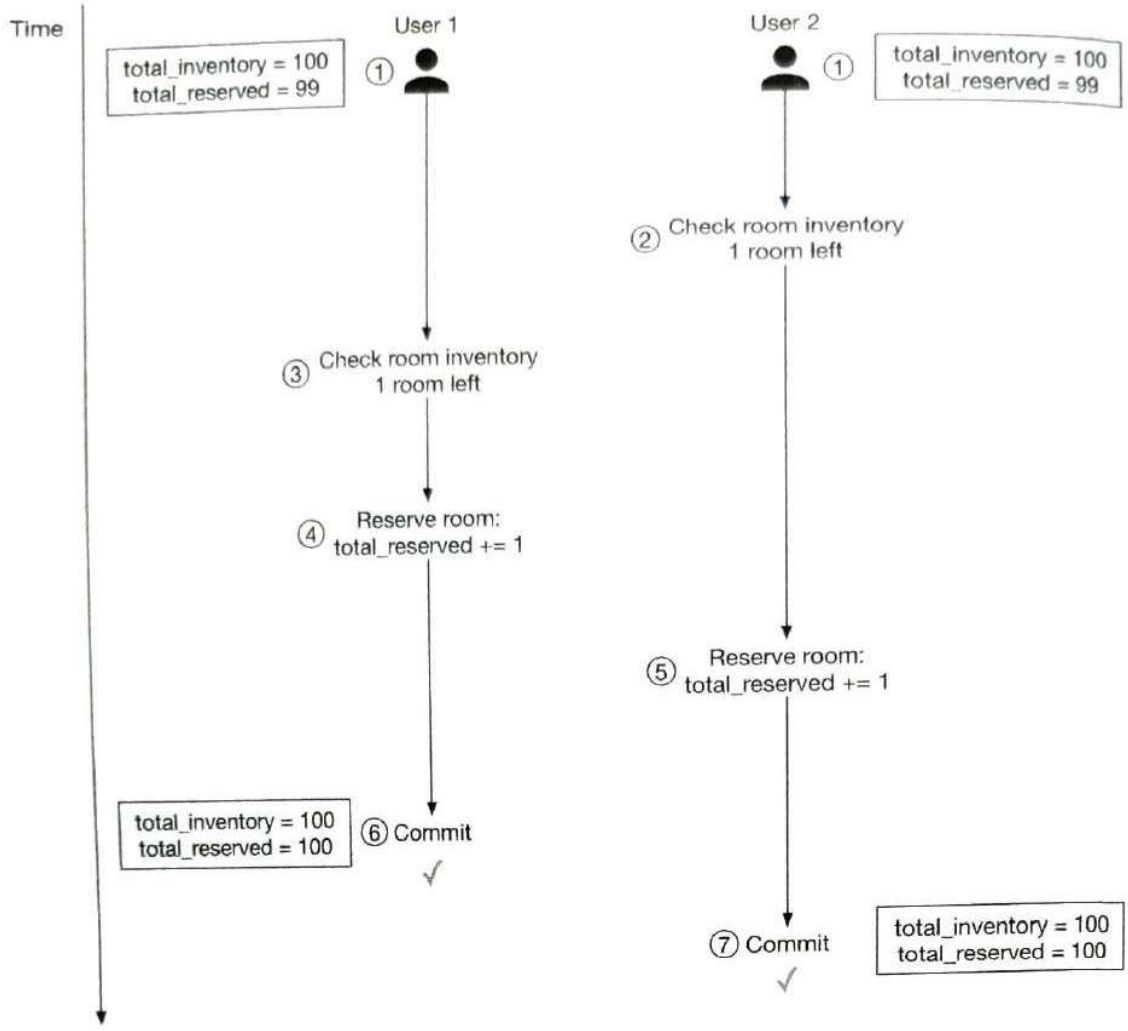


Figure 7.11: Race condition

1. Let's assume the database isolation level is not serializable [5]. User 1 and User 2 try to book the same type of room at the same time, but there is only 1 room left. Let's call User 1's execution transaction 1 and User 2's execution transaction 2. At this time, there are 100 rooms in the hotel and 99 of them are reserved.
2. Transaction 2 checks if there are enough rooms left by checking if (`total_reserved` + `rooms_to_book`) $\leq$ `total_inventory`. Since there is 1 more room left, it returns `True`.
3. Transaction 1 checks if there are enough rooms by checking if (`total_reserved` + `rooms_to_book`) $\leq$ `total_inventory`. Since there is 1 more room left, it returns `True` as well.
4. Transaction 1 reserves the room and updates the inventory: `reserved_room` becomes 100.
5. Then transaction 2 reserves the room. The **isolation** property in ACID means database transactions must complete their tasks independently from other transactions. So data changes made by transaction 1 are not visible to transaction 2 until transaction 1 is completed (committed). So transaction 2 still sees `total_reserved`

as 99 and reserves the room by updating the inventory: reserved_room becomes 100. This results in the system allowing both users to book a room, but there is only 1 room left.

6. Transaction 1 successfully commits the change.
7. Transaction 2 successfully commits the change.

The solution to this problem generally requires some form of locking mechanism. We explore the following techniques:

- Pessimistic locking
- Optimistic locking
- Database constraints

Before jumping into a fix, let's take a look at the SQL pseudo-code utilized to reserve a room. The SQL has two parts:

- Check room inventory
- Reserve a room

```
# step 1: check room inventory
SELECT date, total_inventory, total_reserved
FROM room_type_inventory
WHERE room_type_id = ${roomTypeId} AND hotel_id = ${hotelId}
AND date between ${startDate} and ${endDate}

# For every entry returned from step 1
if((total_reserved + ${numberOfRoomsToReserve}) > 110% *
  total_inventory) {
    Rollback
}

# step 2: reserve rooms
UPDATE room_type_inventory
SET total_reserved = total_reserved + ${numberOfRoomsToReserve}
WHERE room_type_id = ${roomTypeId}
AND date between ${startDate} and ${endDate}

Commit
```

Option 1: Pessimistic locking

The pessimistic locking [6], also called pessimistic concurrency control, prevents simultaneous updates by placing a lock on a record as soon as one user starts to update it. Other users who attempt to update the record have to wait until the first user has released the lock (committed the changes).

For MySQL, the "SELECT ... FOR UPDATE" statement works by locking the rows returned by a selection query. Let's assume a transaction is started by "transaction 1". Other

transactions have to wait for transaction 1 to finish before beginning another transaction. A detailed explanation is shown in Figure 7.12.

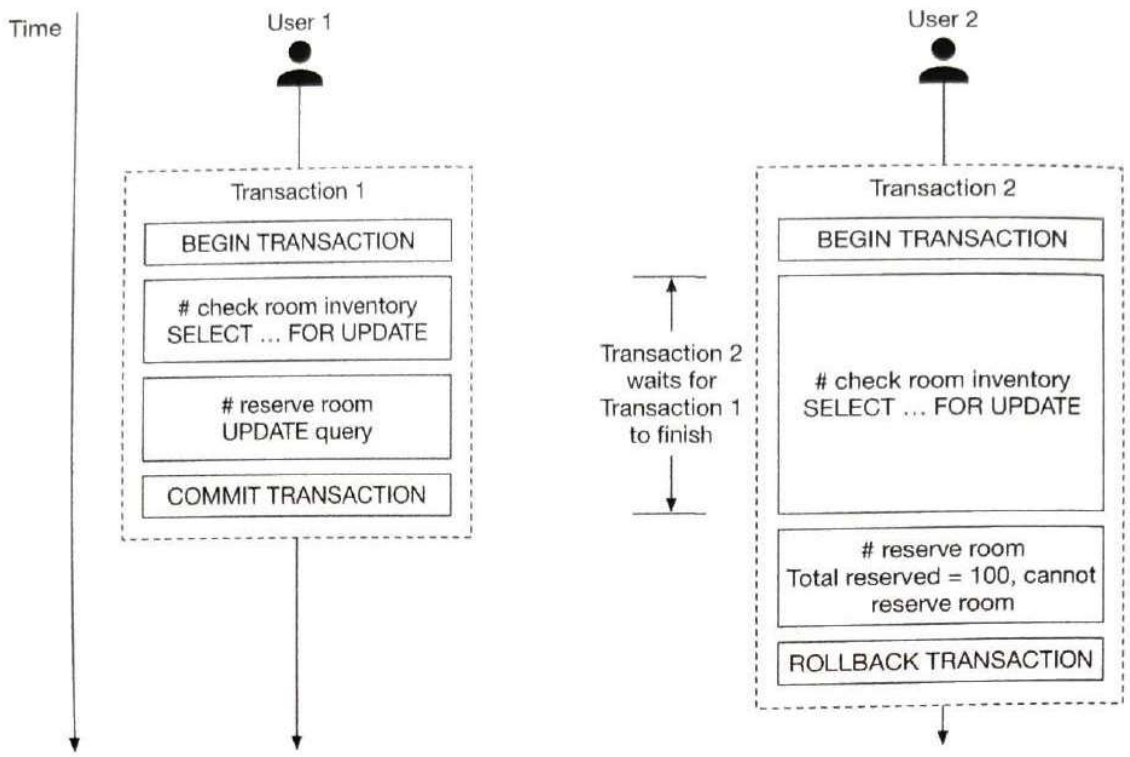


Figure 7.12: Pessimistic locking

In Figure 7.12, the "`SELECT ... FOR UPDATE`" statement of transaction 2 waits for transaction 1 to finish because transaction 1 locks the rows. After transaction 1 finishes, `total_reserved` becomes 100, which means there is no room for user 2 to book.

Pros:

- Prevents applications from updating data that is being or has been changed.
- It is easy to implement and it avoids conflict by serializing updates. Pessimistic locking is useful when data contention is heavy.

Cons:

- Deadlocks may occur when multiple resources are locked. Writing deadlock-free application code could be challenging.
- This approach is not scalable. If a transaction is locked for too long, other transactions cannot access the resource. This has a significant impact on database performance, especially when transactions are long-lived or involve a lot of entities.

Due to these limitations, we do not recommend pessimistic locking for the reservation system.

Option 2: Optimistic locking

Optimistic locking [7], also referred to as optimistic concurrency control, allows multiple concurrent users to attempt to update the same resource.

There are two common ways to implement optimistic locking: version number and timestamp. Version number is generally considered to be a better option because the server clock can be inaccurate over time. We explain how optimistic locking works with version number.

Figure 7.13 shows a successful case and a failure case.

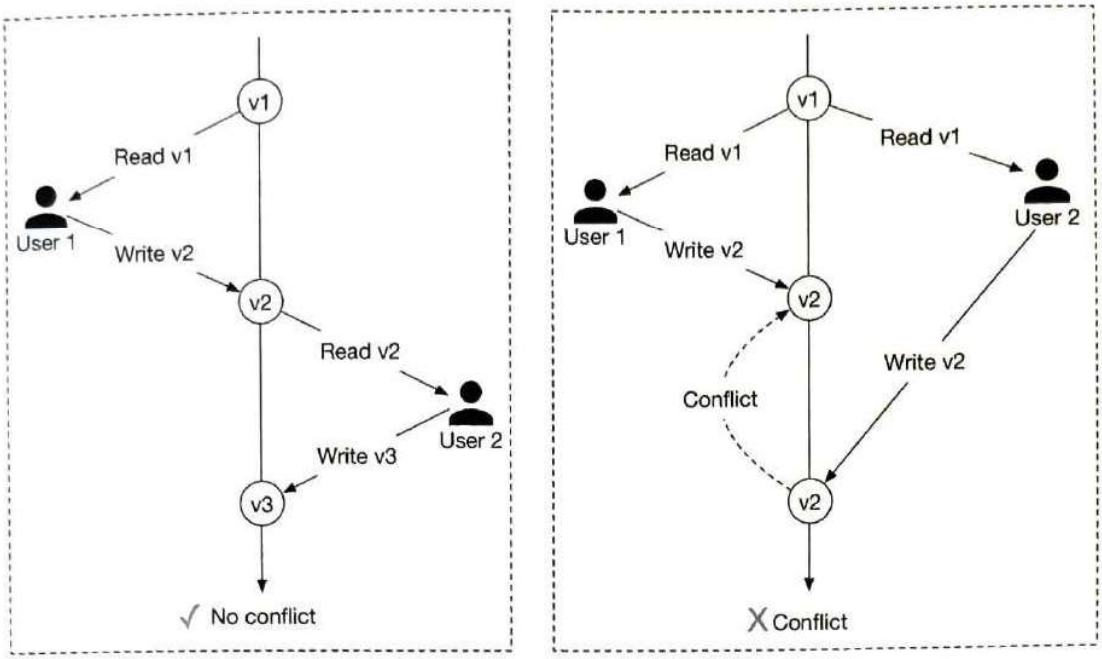


Figure 7.13: Optimistic locking

1. A new column called `version` is added to the database table.
2. Before a user modifies a database row, the application reads the version number of the row.
3. When the user updates the row, the application increases the version number by 1 and writes it back to the database.
4. A database validation check is put in place; the next version number should exceed the current version number by 1. The transaction aborts if the validation fails and the user tries again from step 2.

Optimistic locking is usually faster than pessimistic locking because we do not lock the database. However, the performance of optimistic locking drops dramatically when concurrency is high.

To understand why, consider the case when many clients try to reserve a hotel room at the same time. Because there is no limit on how many clients can read the available room

count, all of them read back the same available room count and the current version number. When different clients make reservations and write back the results to the database, only one of them will succeed, and the rest of the clients receive a version check failure message. These clients have to retry. In the subsequent round of retries, there is only one successful client again, and the rest have to retry. Although the end result is correct, repeated retries cause a very unpleasant user experience.

Pros:

- It prevents applications from editing stale data.
- We don't need to lock the database resource. There's actually no locking from the database point of view. It's entirely up to the application to handle the logic with the version number.
- Optimistic locking is generally used when the data contention is low. When conflicts are rare, transactions can complete without the expense of managing locks.

Cons:

- Performance is poor when data contention is heavy.

Optimistic locking is a good option for a hotel reservation system since the QPS for reservations is usually not high.

Option 3: Database constraints

This approach is very similar to optimistic locking. Let's explore how it works. In the `room_type_inventory` table, add the following constraint:

```
CONSTRAINT `check_room_count` CHECK((`total_inventory - total_reserved
  ` >= 0))
```

Using the same example as shown in Figure 7.14, when user 2 tries to reserve a room, `total_reserved` becomes 101, which violates the `total_inventory` (100)−`total_reserved` (101) ≥ 0 constraint. The transaction is then rolled back.

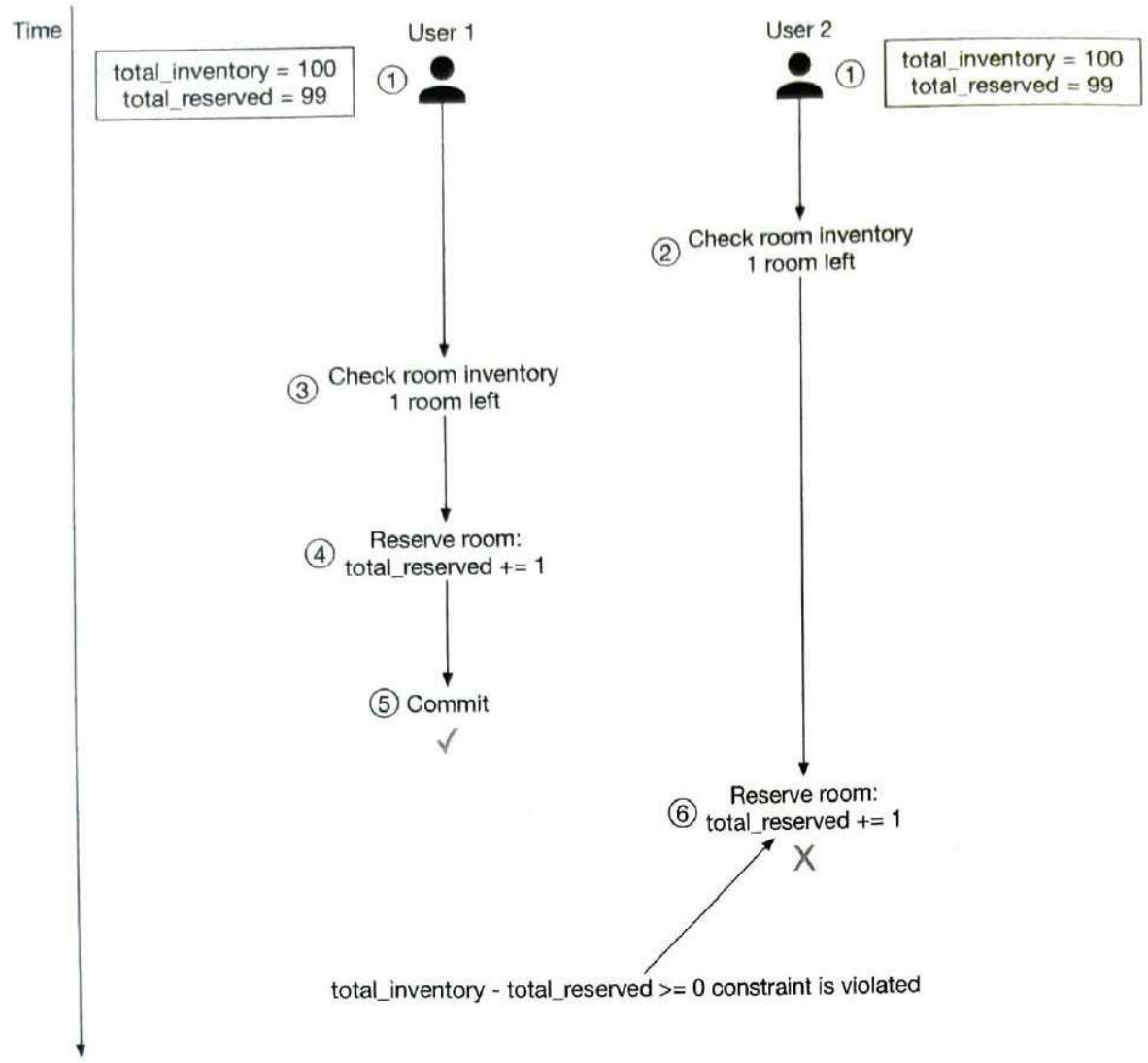


Figure 7.14: Database constraint

Pros

- Easy to implement.
- It works well when data contention is minimal.

Cons

- Similar to optimistic locking, when data contention is heavy, it can result in a high volume of failures. Users could see there are rooms available, but when they try to book one, they get the "no rooms available" response. The experience can be frustrating to users.
- The database constraints cannot be version-controlled easily like the application code.
- Not all databases support constraints. It might cause problems when we migrate from one database solution to another.

Since this approach is easy to implement and the data contention for a hotel reservation is usually not high (low QPS), it is another good option for the hotel reservation system.

Scalability

Usually, the load of the hotel reservation system is not high. However, the interviewer might have a follow-up question: "what if the hotel reservation system is used not just for a hotel chain but for a popular travel site such as booking.com or expedia.com?" In this case, the QPS could be 1,000 times higher.

When the system load is high, we need to understand what might become the bottleneck. All our services are stateless, so they can be easily expanded by adding more servers. The database, however, contains all the states and cannot be scaled up by simply adding more databases. Let's explore how to scale the database.

Database sharding

One way to scale the database is to apply database sharding. The idea is to split the data into multiple databases so that each of them only contains a portion of data.

When we shard a database, we need to consider how to distribute the data. As we can see from the data model section, most queries need to filter by `hotel_id`. So a natural conclusion is we shard data by `hotel_id`. In Figure 7.15, the load is spread among 16 shards. Assume the QPS is 30,000. After database sharding, each shard handles $\frac{30,000}{16} = 1,875$ QPS, which is within a single MySQL server's load capacity.

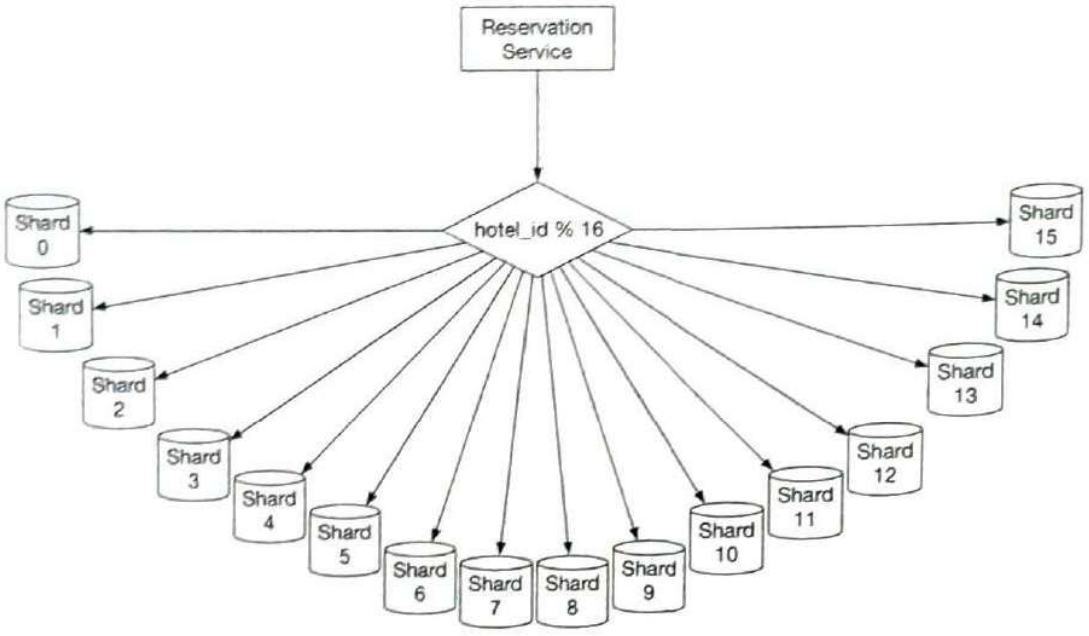


Figure 7.15: Database sharding

Caching

The hotel inventory data has an interesting characteristic; only current and future hotel inventory data are meaningful because customers can only book rooms in the near

future.

So for the storage choice, ideally we want to have a time-to-live (TTL) mechanism to expire old data automatically. Historical data can be queried on a different database. Redis is a good choice because TTL and Least Recently Used (LRU) cache eviction policy help us make optimal use of memory.

If the loading speed and database scalability become an issue (for instance, we are designing at booking.com or Expedia's scale), we can add a cache layer on top of the database and move the check room inventory and reserve room logic to the cache layer, as shown in Figure 7.16. In this design, only a small percentage of the requests hit the inventory database as most ineligible requests are blocked by the inventory cache. One thing worth mentioning is that even when there is enough inventory shown in Redis, we still need to recheck the inventory at the database side as a precaution. The database is the source of truth for the inventory data.

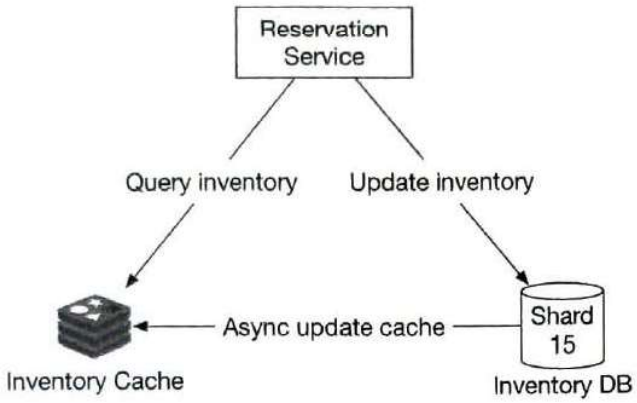


Figure 7.16: Caching

Let's first go over each component in this system.

Reservation service: supports the following inventory management APIs:

- Query the number of available rooms for a given hotel, room type, and date range.
- Reserve a room by executing `total_reserved` +1.
- Update inventory when a user cancels a reservation.

Inventory cache: all inventory management query operations are moved to the inventory cache (Redis) and we need to pre-populate inventory data to the cache. The cache is a key-value store with the following structure:

```
key: hotelID_roomTypeID_{date}
value: the number of available rooms for the given hotel ID,
  room type ID and date.
```

For a hotel reservation system, the volume of read operations (check room inventory) is an order of magnitude higher than write operations. Most of the read operations are

answered by the cache.

Inventory DB: stores inventory data as the source of truth.

New challenges posed by the cache

Adding a cache layer significantly increases the system scalability and throughput, but it also introduces a new challenge: how to maintain data consistency between the database and the cache.

When a user books a room, two operations are executed in the happy path:

1. Query room inventory to find out if there are enough rooms left. The query runs on the Inventory cache.
2. Update inventory data. The inventory DB is updated first. The change is then propagated to the cache asynchronously. This asynchronous cache update could be invoked by the application code, which updates the inventory cache after data is saved to the database. It could also be propagated using change data capture (CDC) [8]. CDC is a mechanism that reads data changes from the database and applies the changes to another data system. One common solution is Debezium [9]. It uses a source connector to read changes from a database and applies them to cache solutions such as Redis [10].

Because the inventory data is updated on the database first, there is a possibility that the cache does not reflect the latest inventory data. For example, the cache may report there is still an empty room when the database says there is no room left or vice versa.

If you think carefully, you find that the inconsistency between inventory cache and database actually does not matter, as long as the database does the final inventory validation check.

Let's take a look at an example. Let's say the cache says there is still an empty room, but the database says no. In this case, when the user queries the room inventory, they find there is still room available, so they try to reserve it. When the request reaches the inventory database, the database does the validation and finds that there is no room left. In this case, the client receives an error, indicating someone else just booked the last room before them. When a user refreshes the website, they probably see there is no room left because the database has synchronized inventory data to the cache, before they click the refresh button.

Pros

- Reduced database load. Since read queries are answered by the cache layer, database load is significantly reduced.
- High performance. Read queries are very fast because results are fetched from memory.

Cons

- Maintaining data consistency between the database and cache is hard. We need to think carefully about how this inconsistency affects user experience.

Data consistency among services

In a traditional monolithic architecture [11], a shared relational database is used to ensure data consistency. In our microservice design, we chose a hybrid approach by having Reservation Service handle both reservation and inventory APIs so that the inventory and reservation database tables are stored in the same relational database. As explained in the "Concurrency issues" section on page 206, this arrangement allows us to leverage the ACID properties of the relational database to elegantly handle many concurrency issues that arise during the reservation flow.

However, if your interviewer is a microservice purist, they might challenge this hybrid approach. In their mind, for a microservice architecture, each microservice has its own databases as shown on the right in Figure 7.17.

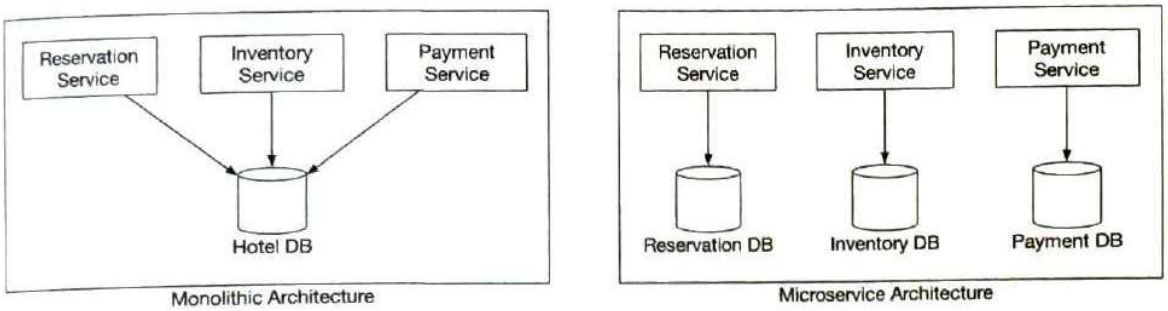


Figure 7.17: Monolithic vs microservice

This pure design introduces many data consistency issues. Since this is the first time we cover microservices, let's explain how and why it happens. To make it easier to understand, only two services are used in this discussion. In the real world, there could be hundreds of microservices within a company. In a monolithic architecture, as shown in Figure 7.18, different operations can be wrapped in a single transaction to ensure ACID properties.

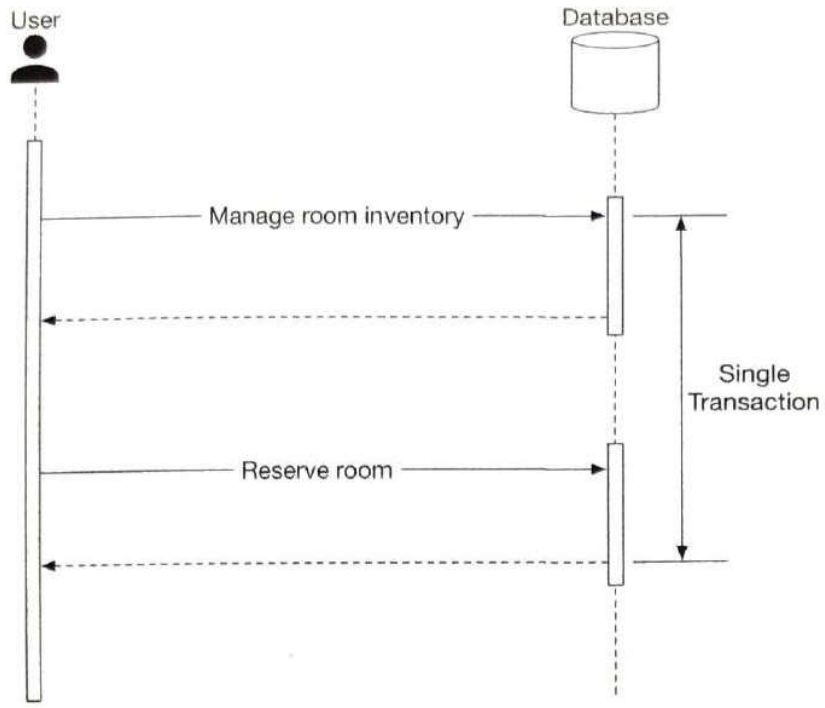


Figure 7.18: Monolithic architecture

However, in a microservice architecture, each service has its own database. One logically atomic operation can span multiple services. This means we cannot use a single transaction to ensure data consistency. As shown in Figure 7.19, if the update operation fails in the reservation database, we need to roll back the reserved room count in the inventory database. Generally, there is only one happy path, but many failure cases that could cause data inconsistency.

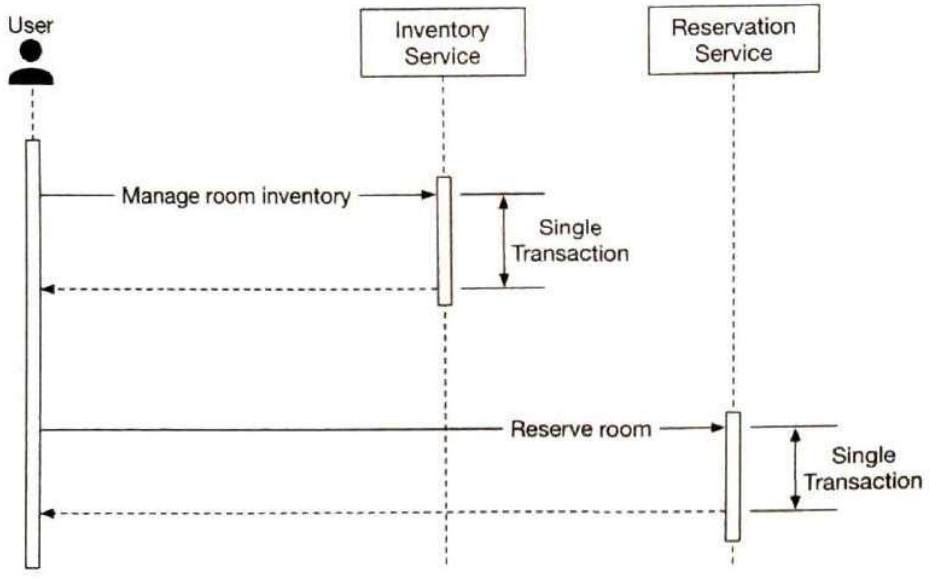


Figure 7.19: Microservice architecture

To address the data inconsistency, here is a high-level summary of industry-proven tech-

niques. If you want to read the details, please refer to the reference materials.

- Two-phase commit (2PC) [12]. 2PC is a database protocol used to guarantee atomic transaction commit across multiple nodes, i.e., either all nodes succeeded or all nodes failed. Because 2PC is a blocking protocol, a single node failure blocks the progress until the node has recovered. It's not performant.
- Saga. A Saga is a sequence of local transactions. Each transaction updates and publishes a message to trigger the next transaction step. If a step fails, the saga executes compensating transactions to undo the changes that were made by preceding transactions [13]. 2PC works as a single commit to perform ACID transactions while Saga consists of multiple steps and relies on eventual consistency.

It is worth noting that addressing data inconsistency between microservices requires some complicated mechanisms that greatly increase the complexity of the overall design. It is up to you as an architect to decide if the added complexity is worth it. For this problem, we decided that it was not worth it and so went with the more pragmatic approach of storing reservation and inventory data under the same relational database.

Step 4 - Wrap Up

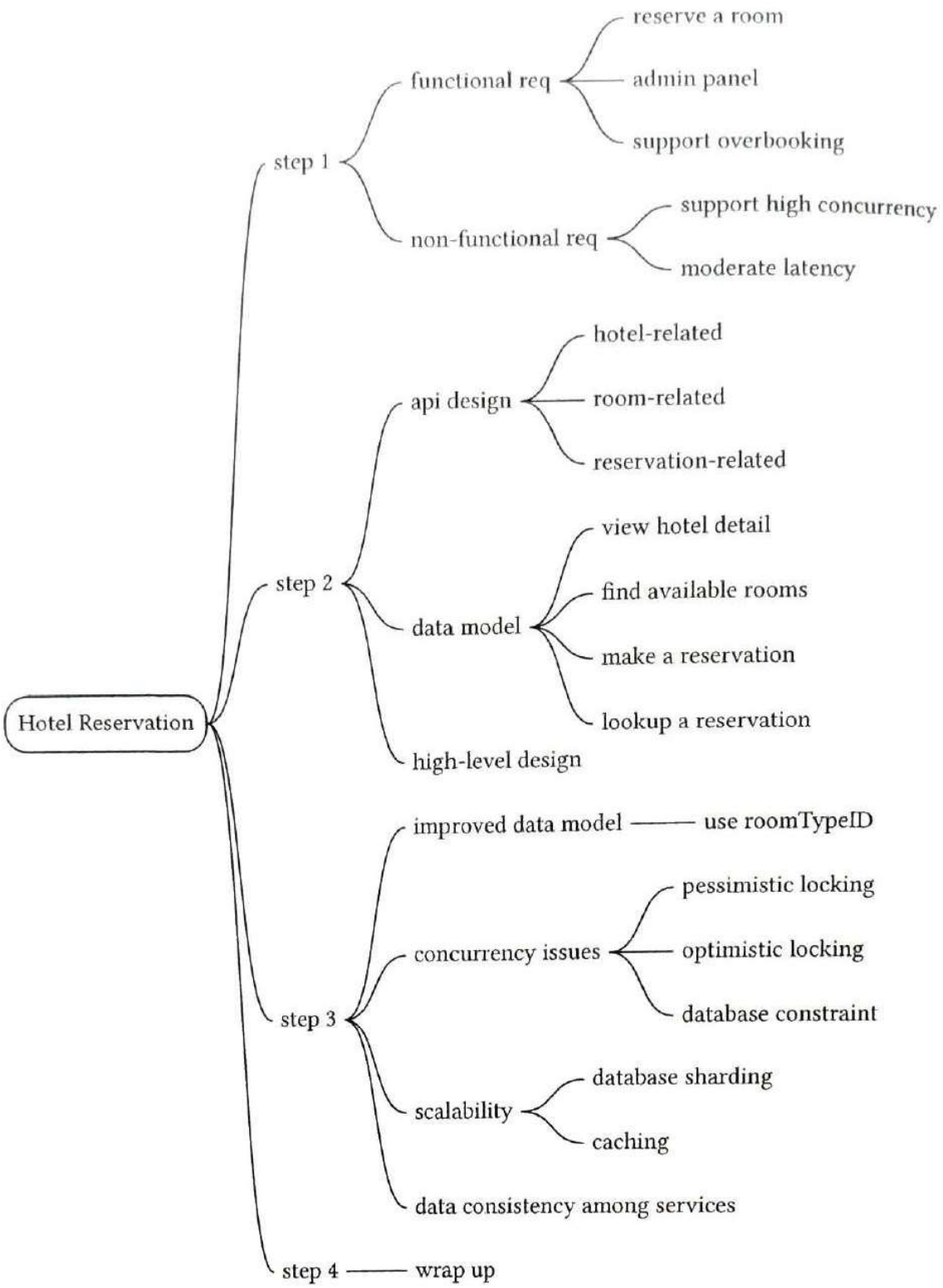
In this chapter, we presented a design for a hotel reservation system. We started by gathering requirements and calculating a back-of-the-envelope estimation to understand the scale. In the high-level design, we presented the API design, the first draft of the data model, and the system architecture diagram. In the deep dive, we explored alternative database schema design as we realized reservations should be made at the room type level, as opposed to specific rooms. We discussed race conditions in depth and proposed a few potential solutions:

- pessimistic locking
- optimistic locking
- database constraints

We then discussed different approaches to scale the system, including database sharding and using Redis cache. Lastly, we went through data consistency issues in microservice architecture and briefly went through a few solutions.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] What Are The Benefits of Microservices Architecture? https://www.appdynamics.com/topics/benefits-of-microservices.

[2] Microservices. https://en.wikipedia.org/wiki/Microservices.

[3] gRPC. https://www.grpc.io/docs/what-is-grpc/introduction/.

[4] Booking.com iOS app.

[5] Serializability. https://en.wikipedia.org/wiki/Serializability.

[6] Optimistic and pessimistic record locking. https://ibm.co/3Eb293O.

[7] Optimistic concurrency control. https://en.wikipedia.org/wiki/Optimistic_concurrency_control.

[8] Change data capture. https://docs.oracle.com/cd/B10500_01/server.920/a96520/cdc.htm.

[9] Debizium. https://debezium.io/.

[10] Redis sink. https://bit.ly/3r3AEUD.

[11] Monolithic Architecture. https://microservices.io/patterns/monolithic.html.

[12] Two-phase commit protocol. https://en.wikipedia.org/wiki/Two-phase_commit_protocol.

[13] Saga. https://microservices.io/patterns/data/saga.html.

8 Distributed Email Service

In this chapter, we design a large-scale email service, such as Gmail, Outlook, or Yahoo Mail. The growth of the internet has led to an explosion in the volume of emails. In 2020, Gmail had over 1.8 billion active users and Outlook had over 400 million users worldwide [1] [2].

Figure 8.1: Popular email providers

Step 1 - Understand the Problem and Establish Design Scope

Over the years, email services have changed significantly in complexity and scale. A modern email service is a complex system with many features. There is no way we can design a real-world system in 45 minutes. So before jumping into the design, we definitely want to ask clarifying questions to narrow down the scope.

Candidate: How many people use the product?
Interviewer: One billion users.

Candidate: I think the following features are important:

- Authentication.
- Send and receive emails.
- Fetch all emails.
- Filter emails by read and unread status.
- Search emails by subject, sender, and body.
- Anti-spam and anti-virus.

Interviewer: That's a good list. We don't need to worry about authentication. Let's focus on the other features you mentioned.

Candidate: How do users connect with mail servers?
Interviewer: Traditionally, users connect with mail servers through native clients that use SMTP, POP, IMAP, and vendor-specific protocols. Those protocols are legacy to some extent, yet still very popular. For this interview, let's assume HTTP is used for client and server communication.

Candidate: Can emails have attachments?
Interviewer: Yes.

Non-functional requirements

Next, let's go over the most important non-functional requirements.

Reliability. We should not lose email data.

Availability. Email and user data should be automatically replicated across multiple nodes to ensure availability. Besides, the system should continue to function despite partial system failures.

Scalability. As the number of users grows, the system should be able to handle the increasing number of users and emails. The performance of the system should not degrade with more users or emails.

Flexibility and extensibility. A flexible/extensible system allows us to add new features or improve performance easily by adding new components. Traditional email protocols such as POP and IMAP have very limited functionality (more on this in high-level design). Therefore, we may need custom protocols to satisfy the flexibility and extensibility requirements.

Back-of-the-envelope estimation

Let's do a back-of-the-envelope calculation to determine the scale and to discover some challenges our solution will need to address. By design, emails are storage heavy applications.

- 1 billion users.
- Assume the average number of emails a person sends per day is 10. QPS for sending emails = $\frac{10^9 \times 10}{10^5} = 100{,}000$.
- Assume the average number of emails a person receives in a day is 40 [3] and the average size of email metadata is 50 KB. Metadata refers to everything related to an email, excluding attachment files.
- Assume metadata is stored in a database. Storage requirement for maintaining metadata in 1 year: 1 billion users $\times$ 40 emails/day $\times$ 365 days $\times$ 50 KB = 730 PB.
- Assume 20% of emails contain an attachment and the average attachment size is 500 KB.
- Storage for attachments in 1 year is: 1 billion users $\times$40 emails/day $\times$ 365 days $\times$ 20% $\times$ 500 KB = 1,460 PB

From this back-of-the-envelope calculation, it's clear we would deal with a lot of data. So, it's likely that we need a distributed database solution.

Step 2 - Propose High-level Design and Get Buy-in

In this section, we first discuss some basics about email servers and how email servers evolve over time. Then we look at the high-level design of distributed email servers. The content is structured as follows:

- Email knowledge 101
- Traditional mail servers
- Distributed mail servers

Email knowledge 101

There are various email protocols that are used to send and receive emails. Historically, most mail servers use email protocols such as POP, IMAP, and SMTP.

Email protocols

SMTP: Simple Mail Transfer Protocol (SMTP) is the standard protocol for **sending** emails from one mail server to another.

The most popular protocols for **retrieving** emails are known as Post Office Protocol (POP) and the Internet Mail Access Protocol (IMAP).

POP is a standard mail protocol to receive and download emails from a remote mail server to a local email client. Once emails are downloaded to your computer or phone, they are deleted from the email server, which means you can only access emails on one computer or phone. The details of POP are covered in RFC 1939 [4]. POP requires mail clients to download the entire email. This can take a long time if an email contains a large attachment.

IMAP is also a standard mail protocol for receiving emails for a local email client. When you read an email, you are connected to an external mail server, and data is transferred to your local device. IMAP only downloads a message when you click it, and emails are not deleted from mail servers, meaning that you can access emails from multiple devices. IMAP is the most widely used protocol for individual email accounts. It works well when the connection is slow because only the email header information is downloaded until opened.

HTTPS is not technically a mail protocol, but it can be used to access your mailbox, particularly for web-based email. For example, it's common for Microsoft Outlook to talk to mobile devices over HTTPS, on a custom-made protocol called ActiveSync [5].

Domain name service (DNS)

A DNS server is used to look up the mail exchanger record (MX record) for the recipient's domain. If you run DNS lookup for gmail.com from the command line, you may get MX records as shown in Figure 8.2.

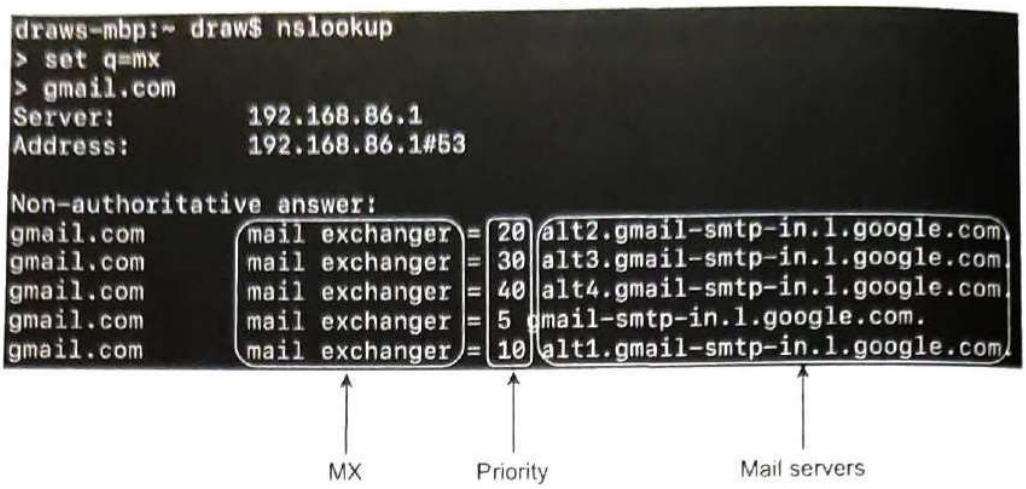


Figure 8.2: MX records

The priority numbers indicate preferences, where the mail server with a lower priority number is more preferred. In Figure 8.2, `gmail-smtp-in.l.google.com` is used first (priority 5). A sending mail server will attempt to connect and send messages to this mail server first. If the connection fails, the sending mail server will attempt to connect to the mail server with the next lowest priority, which is `alt1.gmail-smtp-in.l.google.com` (priority 10).

Attachment

An email attachment is sent along with an email message, commonly with Base64 encoding [6]. There is usually a size limit for an email attachment. For example, Outlook and Gmail limit the size of attachments to 20MB and 25MB respectively as of June 2021. This number is highly configurable and varies from individual to corporate accounts. Multipurpose Internet Mail Extension (MIME) [7] is a specification that allows the attachment to be sent over the internet.

Traditional mail servers

Before we dive into distributed mail servers, let's dig a little bit through the history and see how traditional mail servers work, as doing so provides good lessons about how to scale an email server system. You can consider a traditional mail server as a system that works when there are limited email users, usually on a single server.

Traditional mail server architecture

Figure 8.3 describes what happens when Alice sends an email to Bob, using traditional email servers.

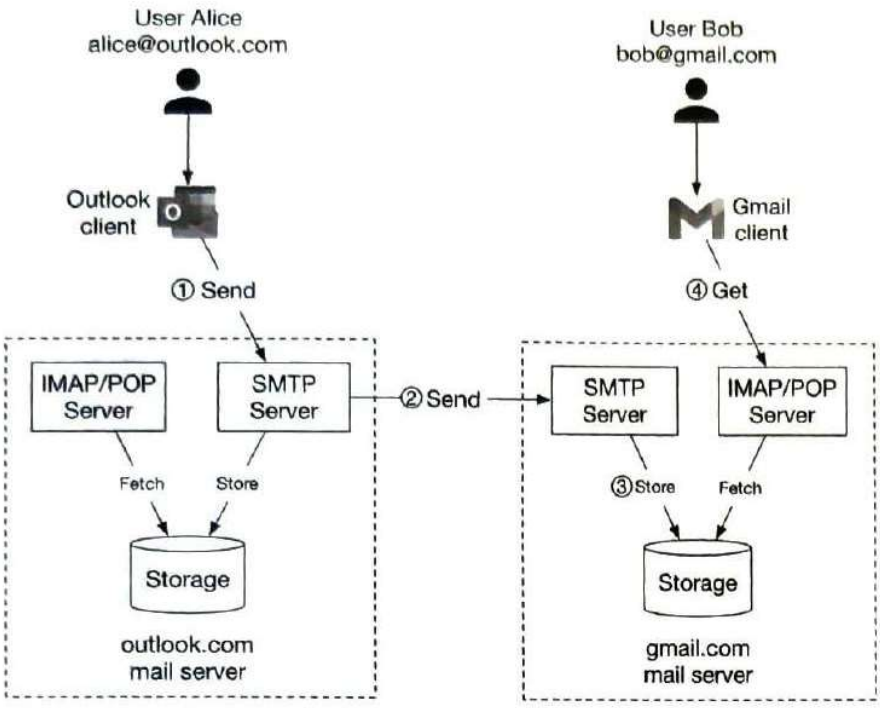


Figure 8.3: Traditional mail servers

The process consists of 4 steps:

1. Alice logs in to her Outlook client, composes an email, and presses the "send" button. The email is sent to the Outlook mail server. The communication protocol between the Outlook client and the mail server is SMTP.
2. Outlook mail server queries the DNS (not shown in the diagram) to find the address of the recipient's SMTP server. In this case, it is Gmail's SMTP server. Next, it transfers the email to the Gmail mail server. The communication protocol between the mail servers is SMTP.
3. The Gmail server stores the email and makes it available to Bob, the recipient.
4. Gmail client fetches new emails through the IMAP/POP server when Bob logs in to Gmail.

Storage

In a traditional mail server, emails were stored in local file directories and each email was stored in a separate file with a unique name. Each user maintained a user directory to store configuration data and mailboxes. Maildir was a popular way to store email messages on the mail server (Figure 8.4).

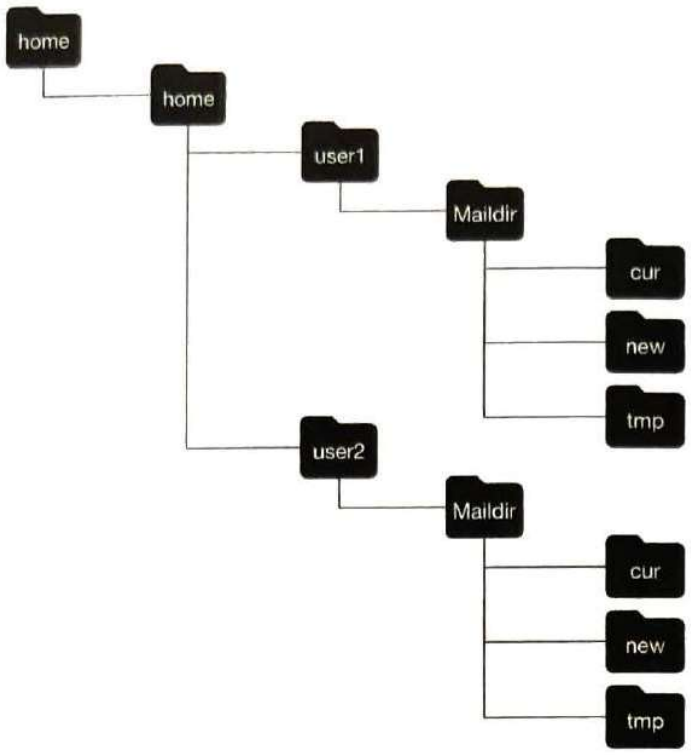


Figure 8.4: Maildir

File directories worked well when the user base was small, but it was challenging to retrieve and backup billions of emails. As the email volume grew and the file structure became more complex, disk I/O became a bottleneck. The local directories also don't satisfy our high availability and reliability requirements. The disk can be damaged and servers can go down. We need a more reliable distributed storage layer.

Email functionality has come a long way since it was invented in the 1960s, from text-based format to rich features such as multimedia, threading [8], search, labels, and more. But email protocols (POP, IMAP, and SMTP) were invented a long time ago and they were not designed to support these new features, nor were they scalable to support billions of users.

Distributed mail servers

Distributed mail servers are designed to support modern use cases and solve the problems of scale and resiliency. This section covers email APIs, distributed email server architecture, email sending, and email receiving flows.

Email APIs

Email APIs can mean very different things for different mail clients, or at different stages of an email's life cycle. For example;

- SMTP/POP/IMAP APIs for native mobile clients.
- SMTP communications between sender and receiver mail servers.

- RESTful API over HTTP for full-featured and interactive web-based email applications.

Due to the length limitations of this book, we cover only some of the most important APIs for webmail. A common way for webmail to communicate is through the HTTP protocol.

1. Endpoint: `POST /v1/messages`

Sends a message to the recipients in the To, Cc, and Bcc headers.

2. Endpoint: `GET /v1/folders`

Returns all folders of an email account.

Response:

```
[{id: string        Unique folder identifier.
  name: string      Name of the folder.
                    According to RFC6154 [9], the default folders can be one of
                    the following:
                    All, Archive, Drafts, Flagged, Junk, Sent, and Trash.
  user_id: string   Reference to the account owner
}]
```

3. Endpoint: `GET /v1/folders/{:folder_id}/messages`

Returns all messages under a folder. Keep in mind this is a highly simplified API. In reality, this needs to support pagination.

Response:

List of message objects.

4. Endpoint: `GET /v1/messages/{:message_id}`

Gets all information about a specific message. Messages are core building blocks for an email application, containing information about the sender, recipients, message subject, body, attachments, etc.

Response:

A message's object.

```
{
 user_id: string                    // Reference to the account owner.
 from: name: string, email: string  // <name, email> pair of the sender.
 to: [name: string, email: string]  // A list of <name, email> paris
 subject: string                    // Subject of an email
 body: string                       // Message body
 is_read: boolean                   // Indicate if a message is read or not.
}
```

Distributed mail server architecture

While it is easy to set up an email server that handles a small number of users, it is difficult to scale beyond one server. This is mainly because traditional email servers were designed to work with a single server only. Synchronizing data across servers can be difficult, and keeping emails from being misclassified as spam by recipients' mail servers is very challenging. In this section, we explore how to leverage cloud technologies to make it easier. The high-level design is shown in Figure 8.5.

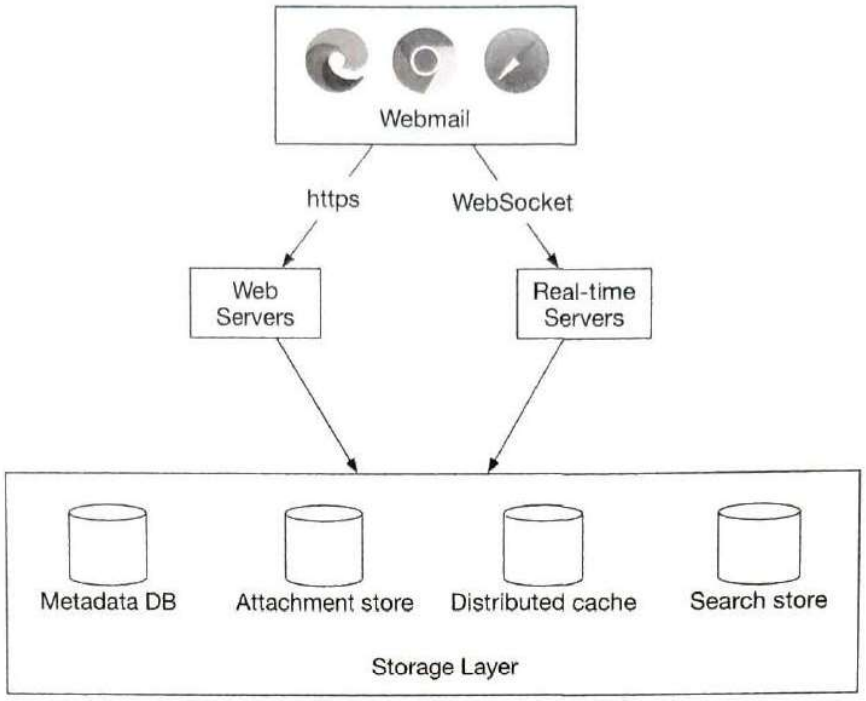


Figure 8.5: High-level design

Let us take a close look at each component.

Webmail. Users use web browsers to receive and send emails.

Web servers. Web servers are public-facing request/response services, used to manage features such as login, signup, user profile, etc. In our design, all email API requests, such as sending an email, loading mail folders, loading all mails in a folder, etc., go through web servers.

Real-time servers. Real-time servers are responsible for pushing new email updates to clients in real-time. Real-time servers are stateful servers because they need to maintain persistent connections. To support real-time communication, we have a few options, such as long polling and WebSocket. WebSocket is a more elegant solution but one drawback of it is browser compatibility. A possible solution is to establish a WebSocket connection whenever possible and to use long-polling as a fallback.

Here is an example of a real-world mail server (Apache James [10]) that implements the JSON Meta Application Protocol (JMAP) subprotocol over WebSocket [11].

Metadata database. This database stores mail metadata including mail subject, body,

from user, to users, etc. We discuss the database choice in the deep dive section.

Attachment store. We choose object stores such as Amazon Simple Storage Service (S3) as the attachment store. S3 is a scalable storage infrastructure that's suitable for storing large files such as images, videos, files, etc. Attachments can take up to 25 MB in size. NoSQL column-family databases like Cassandra might not be a good fit for the following two reasons:

- Even though Cassandra supports blob data type and its maximum theoretical size for a blob is 2GB, the practical limit is less than 1MB [12].
- Another problem with putting attachments in Cassandra is that we can't use a row cache as attachments take too much memory space.

Distributed cache. Since the most recent emails are repeatedly loaded by a client, caching recent emails in memory significantly improves the load time. We can use Redis here because it offers rich features such as lists and it is easy to scale.

Search store. The search store is a distributed document store. It uses a data structure called inverted index [13] that supports very fast full-text searches. We will discuss this in more detail in the deep dive section.

Now that we have discussed some of the most important components to build distributed mail servers, let's assemble together two main workflows.

- Email sending flow.
- Email receiving flow.

Email sending flow

The email sending flow is shown in Figure 8.6.

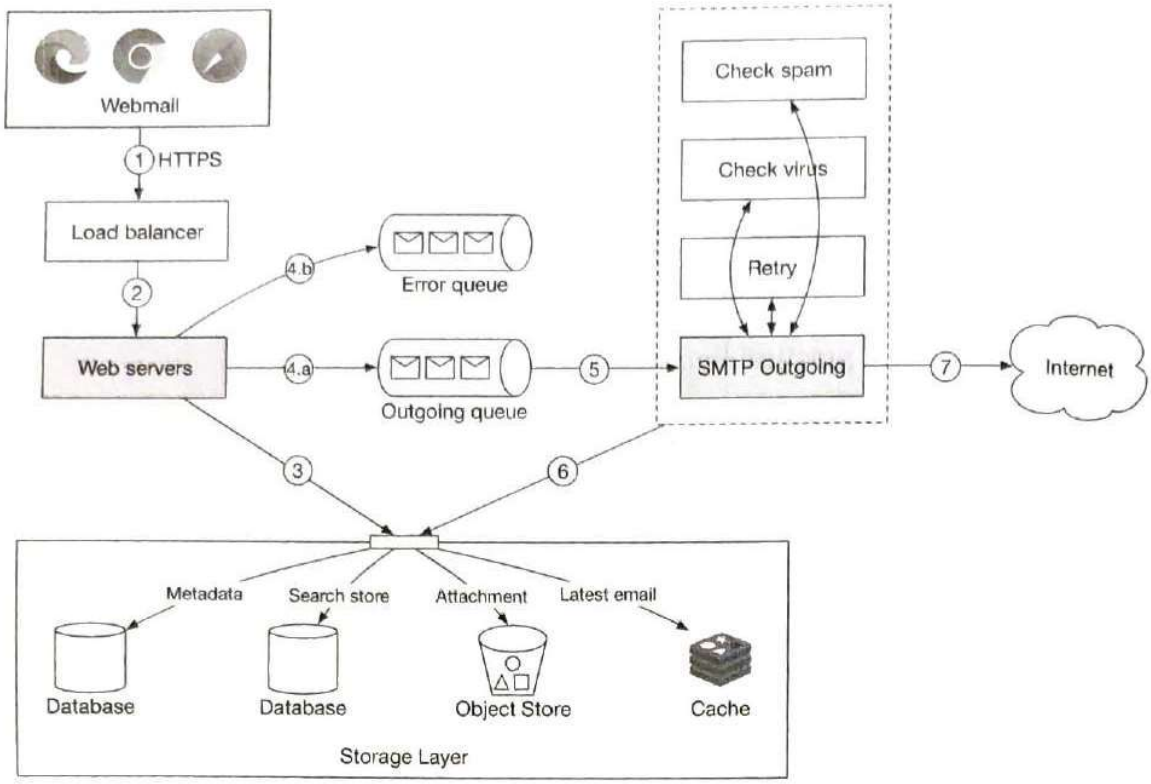


Figure 8.6: Email sending flow

1. A user writes an email on webmail and presses the send button. The request is sent to the load balancer.
2. The load balancer makes sure it doesn't exceed the rate limit and routes traffic to web servers.
3. Web servers are responsible for:
   - Basic email validation. Each incoming email is checked against pre-defined rules such as email size limit.
   - Checking if the domain of the recipient's email address is the same as the sender. If it is the same, the web server ensures the email data is spam and virus free. If so, email data is inserted into the sender's "Sent Folder" and recipient's "Inbox Folder". The recipient can fetch the email directly via the RESTful API. There is no need to go to step 4.
4. Message queues.
   4.1. If basic email validation succeeds, the email data is passed to the outgoing queue. If the attachment is too large to fit in the queue, we could store the attachment in the object store and save the object reference in the queued message.
   4.2. If basic email validation fails, the email is put in the error queue.
5. SMTP outgoing workers pull messages from the outgoing queue and make sure emails are spam and virus free.

6. The outgoing email is stored in the "Sent Folder" of the storage layer.
7. SMTP outgoing workers send the email to the recipient mail server.

Each message in the outgoing queue contains all the metadata required to create an email. A distributed message queue is a critical component that allows asynchronous mail processing. By decoupling SMTP outgoing workers from the web servers, we can scale SMTP outgoing workers independently.

We monitor the size of the outgoing queue very closely. If there are many emails stuck in the queue, we need to analyze the cause of the issue. Here are some possibilities:

- The recipient's mail server is unavailable. In this case, we need to retry sending the email at a later time. Exponential backoff [14] might be a good retry strategy.
- Not enough consumers to send emails. In this case, we may need more consumers to reduce the processing time.

Email receiving flow

The following diagram demonstrates the email receiving flow.

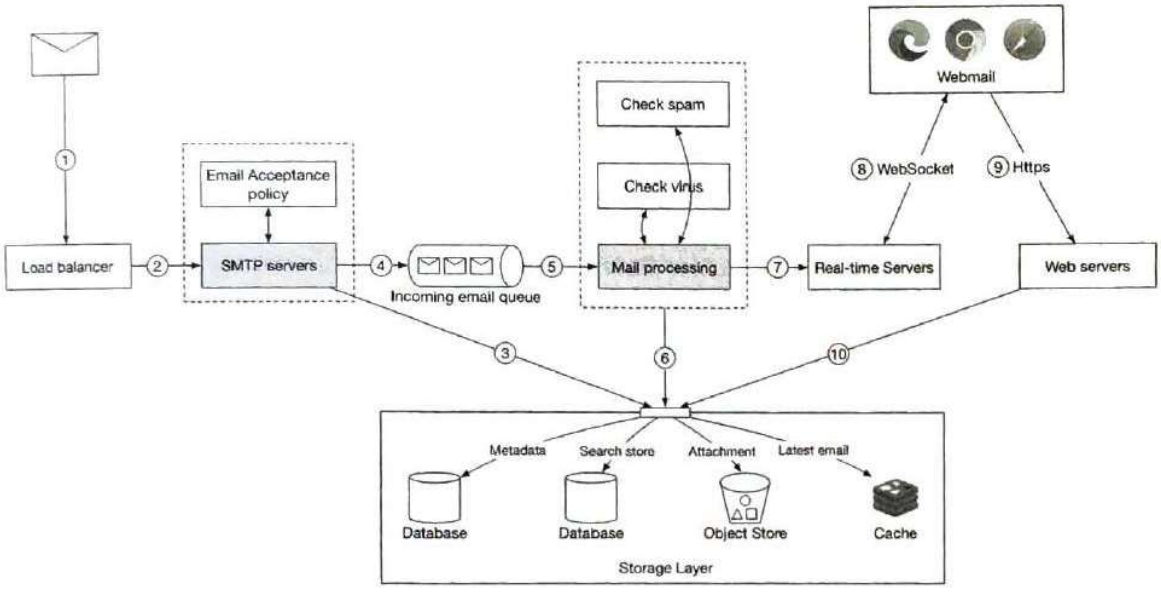


Figure 8.7: Email receiving flow

1. Incoming emails arrive at the SMTP load balancer.
2. The load balancer distributes traffic among SMTP servers. Email acceptance policy can be configured and applied at the SMTP-connection level. For example, invalid emails are bounced to avoid unnecessary email processing.
3. If the attachment of an email is too large to put into the queue, we can put it into the attachment store (S3).
4. Emails are put in the incoming email queue. The queue decouples mail processing

workers from SMTP servers so they can be scaled independently. Moreover, the queue serves as a buffer in case the email volume surges.

5. Mail processing workers are responsible for a lot of tasks, including filtering out spam mails, stopping viruses, etc. The following steps assume an email passed the validation.
6. The email is stored in the mail storage, cache, and object data store.
7. If the receiver is currently online, the email is pushed to real-time servers.
8. Real-time servers are WebSocket servers that allow clients to receive new emails in real-time.
9. For offline users, emails are stored in the storage layer. When a user comes back online, the webmail client connects to web servers via RESTful API.
10. Web servers pull new emails from the storage layer and return them to the client.

Step 3 - Design Deep Dive

Now that we have talked about all the parts of the email server, let's go deeper into some key components and examine how to scale the system.

- Metadata database
- Search
- Deliverability
- Scalability

Metadata database

In this section, we discuss the characteristics of email metadata, choosing the right database, data model, and conversation threads (bonus point).

Characteristics of email metadata

- Email headers are usually small and frequently accessed.
- Email body sizes can range from small to big but are infrequently accessed. You normally only read an email once.
- Most of the mail operations, such as fetching mails, marking an email as read, and searching are isolated to an individual user. In other words, mails owned by a user are only accessible by that user and all the mail operations are performed by the same user.
- Data recency impacts data usage. Users usually only read the most recent emails. 82% of read queries are for data younger than 16 days [15].
- Data has high-reliability requirements. Data loss is not acceptable.

Choosing the right database

At Gmail or Outlook scale, the database system is usually custom-made to reduce input/output operations per second (IOPS) [16], as this can easily become a major constraint in the system. Choosing the right database is not easy. It is helpful to consider all the options we have on the table before deciding the most suitable one.

- Relational database. The main motivation behind this is to search through emails efficiently. We can build indexes for email header and body. With indexes, simple search queries are fast. However, relational databases are typically optimized for small chunks of data entries and are not ideal for large ones. A typical email is usually larger than a few KB and can easily be over 100KB when HTML is involved. You might argue that the BLOB data type is designed to support large data entries. However, search queries over unstructured BLOB data type are not efficient. So relational databases such as MySQL or PostgreSQL are not good fits.
- Distributed object storage. Another potential solution is to store raw emails in cloud storage such as Amazon S3, which can be a good option for backup storage, but it's hard to efficiently support features such as marking emails as read, searching emails based on keywords, threading emails, etc.
- NoSQL databases. Google Bigtable is used by Gmail, so it's definitely a viable solution. However, Bigtable is not open sourced and how email search is implemented remains a mystery. Cassandra might be a good option as well, but we haven't seen any large email providers use it yet.

Based on the above analysis, very few existing solutions seem to fit our needs perfectly. Large email service providers tend to build their own highly customized databases. However, in an interview setting, we won't have time to design a new distributed database, though it's important to explain the following characteristics that the database should have.

- A single column can be a single-digit of MB.
- Strong data consistency.
- Designed to reduce disk I/O.
- It should be highly available and fault-tolerant.
- It should be easy to create incremental backups.

Data model

One way to store the data is to use `user_id` as a partition key so data for one user is stored on a single shard. One limitation of this data model is that messages are not shared among multiple users. Since this is not a requirement for us in this interview, it's not something we need to worry about.

Now let us define the tables. The primary key contains two components, the partition key, and the clustering key.