
tab
caps lock
shift
control

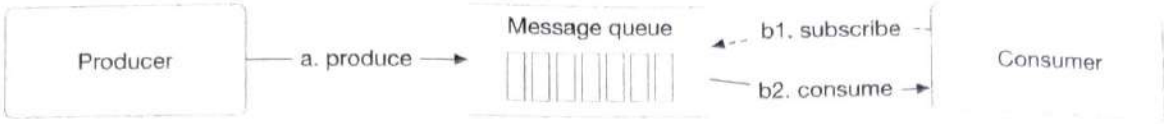


Figure 4.2: Key components in a message queue

- Producer sends messages to a message queue.
- Consumer subscribes to a queue and consumes the subscribed messages.
- Message queue is a service in the middle that decouples the producers from the consumers, allowing each of them to operate and scale independently.
- Both producer and consumer are clients in the client/server model, while the message queue is the server. The clients and servers communicate over the network.

Messaging models

The most popular messaging models are point-to-point and publish-subscribe.

Point-to-point

This model is commonly found in traditional message queues. In a point-to-point model, a message is sent to a queue and consumed by one and only one consumer. There can be multiple consumers waiting to consume messages in the queue, but each message can only be consumed by a single consumer. In Figure 4.3, message A is only consumed by consumer 1.

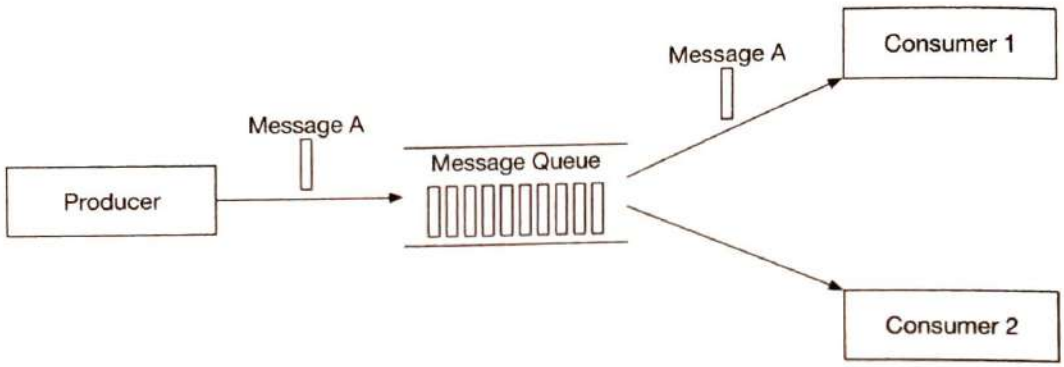


Figure 4.3: Point-to-point model

Once the consumer acknowledges that a message is consumed, it is removed from the queue. There is no data retention in the point-to-point model. In contrast, our design includes a persistence layer that keeps the messages for two weeks, which allows messages to be repeatedly consumed.

While our design could simulate a point-to-point model, its capabilities map more naturally to the publish-subscribe model.

Publish-subscribe

First, let's introduce a new concept, the topic. Topics are the categories used to organize messages. Each topic has a name that is unique across the entire message queue service.

Messages are sent to and read from a specific topic.

In the publish-subscribe model, a message is sent to a topic and received by the consumers subscribing to this topic. As shown in Figure 4.4, message A is consumed by both consumer 1 and consumer 2.

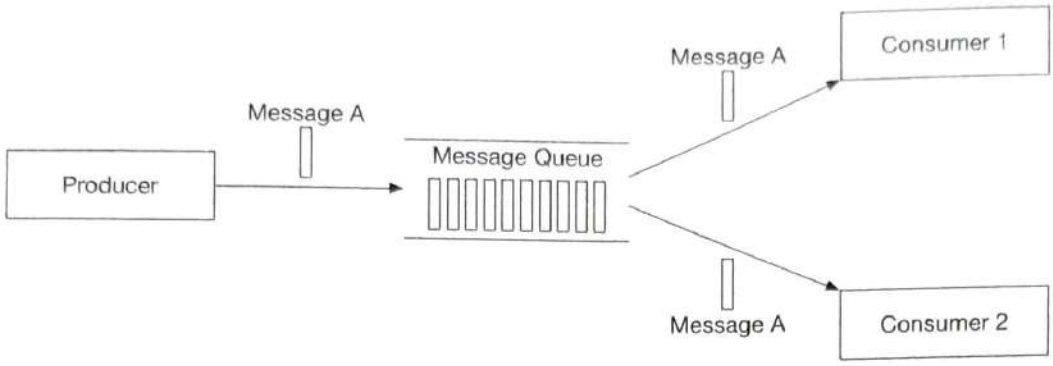


Figure 4.4: Publish-subscribe model

Our distributed message queue supports both models. The publish-subscribe model is implemented by **topics**, and the point-to-point model can be simulated by the concept of the **consumer group**, which will be introduced in the consumer group section.

Topics, partitions, and brokers

As mentioned earlier, messages are persisted by topics. What if the data volume in a topic is too large for a single server to handle?

One approach to solve this problem is called **partition (sharding)**. As Figure 4.5 shows, we divide a topic into partitions and deliver messages evenly across partitions. Think of a partition as a small subset of the messages for a topic. Partitions are evenly distributed across the servers in the message queue cluster. These servers that hold partitions are called **brokers**. The distribution of partitions among brokers is the key element to support high scalability. We can scale the topic capacity by expanding the number of partitions.

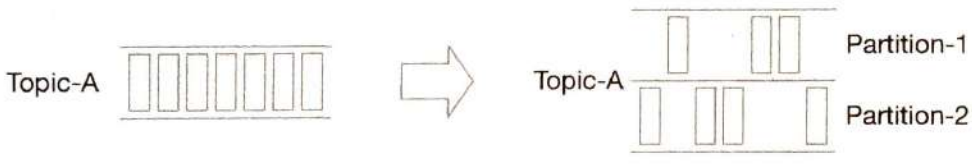


Figure 4.5: Partitions

Each topic partition operates in the form of a queue with the FIFO (first in, first out) mechanism. This means we can keep the order of messages inside a partition. The position of a message in the partition is called an **offset**.

When a message is sent by a producer, it is actually sent to one of the partitions for the topic. Each message has an optional message key (for example, a user's ID), and all messages for the same message key are sent to the same partition. If the message key is absent, the message is randomly sent to one of the partitions.

When a consumer subscribes to a topic, it pulls data from one or more of these partitions. When there are multiple consumers subscribing to a topic, each consumer is responsible for a subset of the partitions for the topic. The consumers form a **consumer group** for a topic.

The message queue cluster with brokers and partitions is represented in Figure 4.6.

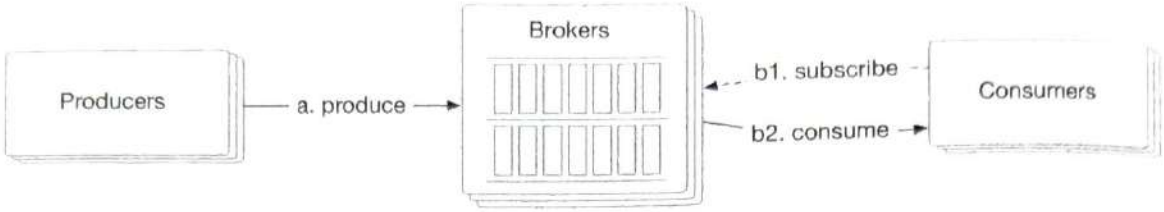


Figure 4.6: Message queue cluster

Consumer group

As mentioned earlier, we need to support both point-to-point and subscribe-publish models. **A consumer group** is a set of consumers, working together to consume messages from topics.

Consumers can be organized into groups. Each consumer group can subscribe to multiple topics and maintain its own consuming offsets. For example, we can group consumers by use cases, one group for billing and the other for accounting.

The instances in the same group can consume traffic in parallel, as in Figure 4.7.

- Consumer group 1 subscribes to topic A.
- Consumer group 2 subscribes to both topics A and B.
- Topic A is subscribed by both consumer groups-1 and group-2, which means the same message is consumed by multiple consumers. This pattern supports the subscribe/publish model.

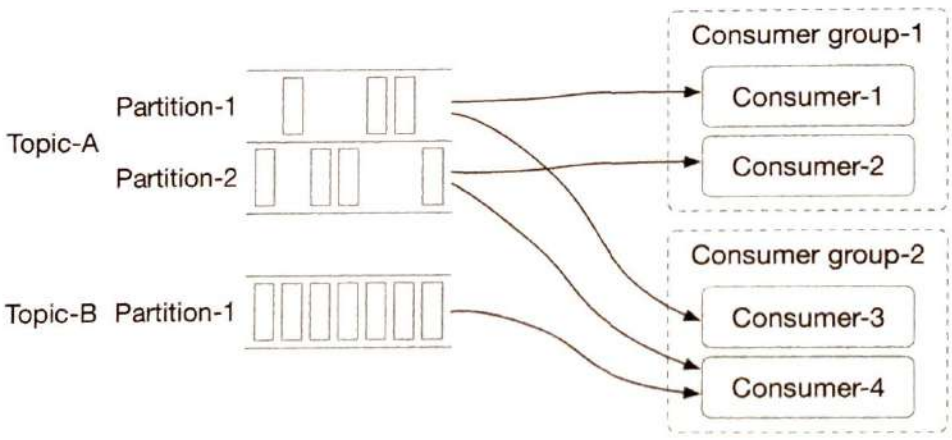


Figure 4.7: Consumer groups

However, there is one problem. Reading data in parallel improves the throughput, but

the consumption order of messages in the same partition cannot be guaranteed. For example, if Consumer-1 and Consumer-2 both read from Partition-1, we will not be able to guarantee the message consumption order in Partition-1.

The good news is we can fix this by adding a constraint, that a single partition can only be consumed by one consumer in the same group. If the number of consumers of a group is larger than the number of partitions of a topic, some consumers will not get data from this topic. For example, in Figure 4.7, Consumer-3 in Consumer group-2 cannot consume messages from topic B because it is consumed by Consumer-4 in the same consumer group, already.

With this constraint, if we put all consumers in the same consumer group, then messages in the same partition are consumed by only one consumer, which is equivalent to the point-to-point model. Since a partition is the smallest storage unit, we can allocate enough partitions in advance to avoid the need to dynamically increase the number of partitions. To handle high scale, we just need to add consumers.

High-level architecture

Figure 4.8 shows the updated high-level design.

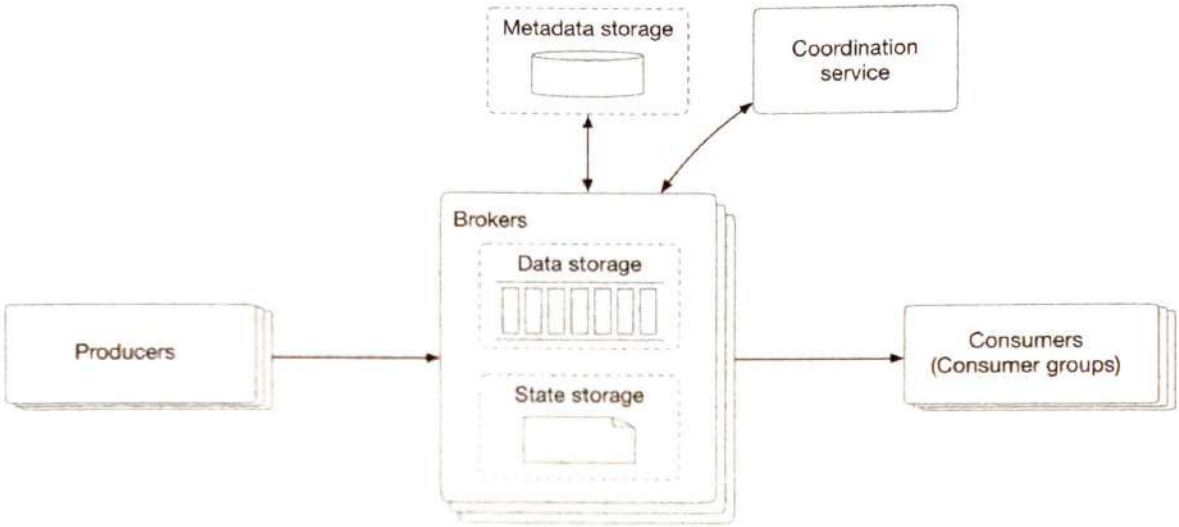


Figure 4.8: High-level design

Clients

- Producer: pushes messages to specific topics.
- Consumer group: subscribes to topics and consumes messages.

Core service and storage

- Broker: holds multiple partitions. A partition holds a subset of messages for a topic.
- Storage:
    - Data storage: messages are persisted in data storage in partitions.

- State storage: consumer states are managed by state storage.
- Metadata storage: configuration and properties of topics are persisted in metadata storage.

- Coordination service:
  - Service discovery: which brokers are alive.
  - Leader election: one of the brokers is selected as the active controller. There is only one active controller in the cluster. The active controller is responsible for assigning partitions.
  - Apache ZooKeeper [2] or etcd [3] are commonly used to elect a controller.

Step 3 - Design Deep Dive

To achieve high throughput while satisfying the high data retention requirement, we made three important design choices, which we explain in detail now.

- We chose an on-disk data structure that takes advantage of the great sequential access performance of rotational disks and the aggressive disk caching strategy of modern operating systems.
- We designed the message data structure to allow a message to be passed from the producer to the queue and finally to the consumer, with no modifications. This minimizes the need for copying which is very expensive in a high volume and high traffic system.
- We designed the system to favor batching. Small I/O is an enemy of high throughput. So, wherever possible, our design encourages batching. The producers send messages in batches. The message queue persists messages in even larger batches. The consumers fetch messages in batches when possible, too.

Data storage

Now let's explore the options to persist messages in more detail. In order to find the best choice, let's consider the traffic pattern of a message queue.

- Write-heavy, read-heavy.
- No update or delete operations. As a side note, a traditional message queue does not persist messages unless the queue falls behind, in which case there will be "delete" operations when the queue catches up. What we are talking about here is the persistence of a data streaming platform.
- Predominantly sequential read/write access.

Option 1: Database

The first option is to use a database.

- Relational database: create a topic table and write messages to the table as rows.

- NoSQL database: create a collection as a topic and write messages as documents.

Databases can handle the storage requirement, but they are not ideal because it is hard to design a database that supports both write-heavy and read-heavy access patterns at a large scale. The database solution does not fit our specific data usage patterns very well.

This means a database is not the best choice and could become a bottleneck of the system.

Option 2: Write-ahead log (WAL)

The second option is write-ahead log (WAL). WAL is just a plain file where new entries are appended to an append-only log. WAL is used in many systems, such as the redo log in MySQL [4] and the WAL in ZooKeeper.

We recommend persisting messages as WAL log files on disk. WAL has a pure sequential read/write access pattern. The disk performance of sequential access is very good [5]. Also, rotational disks have large capacity and they are pretty affordable.

As shown in Figure 4.9, a new message is appended to the tail of a partition, with a monotonically increasing offset. The easiest option is to use the line number of the log file as the offset. However, a file cannot grow infinitely, so it is a good idea to divide it into segments.

With segments, new messages are appended only to the active segment file. When the active segment reaches a certain size, a new active segment is created to receive new messages, and the currently active segment becomes inactive, like the rest of the non-active segments. Non-active segments only serve read requests. Old non-active segment files can be truncated if they exceed the retention or capacity limit.

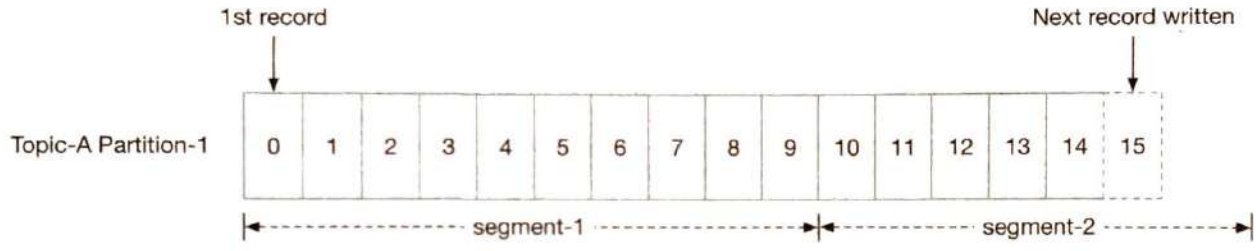


Figure 4.9: Append new messages

Segment files of the same partition are organized in a folder named `Partition-{:partition_id}`. The structure is shown in Figure 4.10.

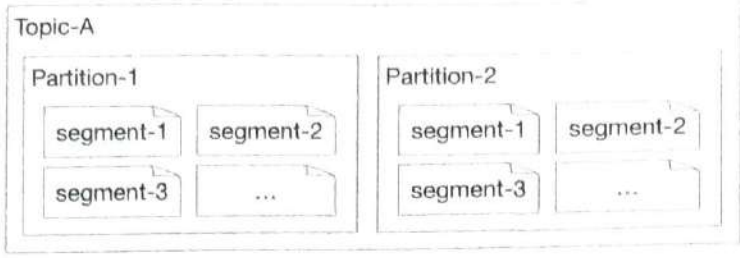


Figure 4.10: Data segment file distribution in topic partitions

A note on disk performance

To meet the high data retention requirement, our design relies heavily on disk drives to hold a large amount of data. There is a common misconception that rotational disks are slow, but this is really only the case for random access. For our workload, as long as we design our on-disk data structure to take advantage of the sequential access pattern, the modern disk drives in a RAID configuration (i.e., with disks striped together for higher performance) could comfortably achieve several hundred MB/sec of read and write speed. This is more than enough for our needs, and the cost structure is favorable.

Also, a modern operating system caches disk data in main memory very aggressively, so much so that it would happily use all available free memory to cache disk data. The WAL takes advantage of the heavy OS disk caching, too, as we described above.

Message data structure

The data structure of a message is key to high throughput. It defines the contract between the producers, message queue, and consumers. Our design achieves high performance by eliminating unnecessary data copying while the messages are in transit from the producers to the queue and finally to the consumers. If any parts of the system disagree on this contract, messages will need to be mutated which involves expensive copying. It could seriously hurt the performance of the system.

Below is a sample schema of the message data structure:

Field Name	Data Type
`key`	byte[]
`value`	byte[]
`topic`	string
`partition`	integer
`offset`	long
`timestamp`	long
`size`	integer
`crc`	integer

Table 4.1: Data schema of a message

Message key

The key of the message is used to determine the partition of the message. If the key is not defined, the partition is randomly chosen. Otherwise, the partition is chosen by `hash(key) % numPartitions`. If we need more flexibility, the producer can define its own mapping algorithm to choose partitions. Please note that the key is not equivalent to the partition number.

The key can be a string or a number. It usually carries some business information. The partition number is a concept in the message queue, which should not be explicitly exposed to clients.

With a proper mapping algorithm, if the number of partitions changes, messages can still be evenly sent to all the partitions.

Message value

The message value is the payload of a message. It can be plain text or a compressed binary block.

Reminder
The key and value of a message are different from the key-value pair in a key-value (KV) store. In the KV store, keys are unique, and we can find the value by key. In a message, keys do not need to be unique. Sometimes they are not even mandatory, and we don't need to find a value by key.

Other fields of a message

- Topic: the name of the topic that the message belongs to.
- Partition: the ID of the partition that the message belongs to.
- Offset: the position of the message in the partition. We can find a message via the combination of three fields: topic, partition, offset.
- Timestamp: the timestamp of when this message is stored.
- Size: the size of this message.
- CRC: Cyclic redundancy check (CRC) is used to ensure the integrity of raw data.

To support additional features, some optional fields can be added on demand. For example, messages can be filtered by tags, if tags are part of the optional fields.

Batching

Batching is pervasive in this design. We batch messages in the producer, the consumer, and the message queue itself. Batching is critical to the performance of the system. In this section, we focus primarily on batching in the message queue. We discuss batching for producer and consumer in more detail, shortly.

Batching is critical to improving performance because:

- It allows the operating system to group messages together in a single network request and amortizes the cost of expensive network round trips
- The broker writes messages to the append logs in large chunks, which leads to larger blocks of sequential writes and larger contiguous blocks of disk cache, maintained by the operating system. Both lead to much greater sequential disk access throughput

There is a tradeoff between throughput and latency. If the system is deployed as a traditional message queue where latency might be more important, the system could be tuned to use a smaller batch size. Disk performance will suffer a little bit in this use case. If tuned for throughput, there might need to be a higher number of partitions per topic, to make up for the slower sequential disk write throughput.

So far, we've covered the main disk storage subsystem and its associated on-disk data structure. Now, let's switch gears and discuss the producer and consumer flows. Then we will come back and finish the deep dive into the rest of the message queue.

Producer flow

If a producer wants to send messages to a partition, which broker should it connect to? The first option is to introduce a routing layer. All messages sent to the routing layer are routed to the "correct" broker. If the brokers are replicated, the "correct" broker is the leader replica. We will cover replication later.

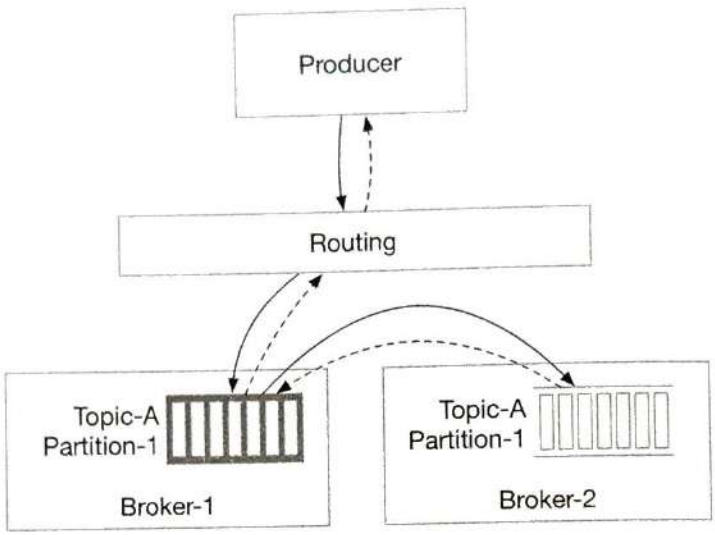


Figure 4.11: Routing layer

As shown in Figure 4.11, the producer tries to send messages to Partition-1 of Topic-A.

1. The producer sends messages to the routing layer.
2. The routing layer reads the replica distribution plan[1] from the metadata storage and caches it locally. When a message arrives, it routes the message to the leader replica of Partition-1, which is stored in Broker-1.

[1]The distribution of replicas for each partition is called a replica distribution plan

3. The leader replica receives the message and follower replicas pull data from the leader.
4. When "enough" replicas have synchronized the message, the leader commits the data (persisted on disk), which means the data can be consumed. Then it responds to the producer.

You might be wondering why we need both leader and follower replicas. The reason is fault tolerance. We dive deep into this process in the "In-sync replicas" section on page 113.

This approach works, but it has a few drawbacks:

- A new routing layer means additional network latency caused by overhead and additional network hops.
- Request batching is one of the big drivers of efficiency. This design doesn't take that into consideration.

Figure 4.12 shows the improved design.

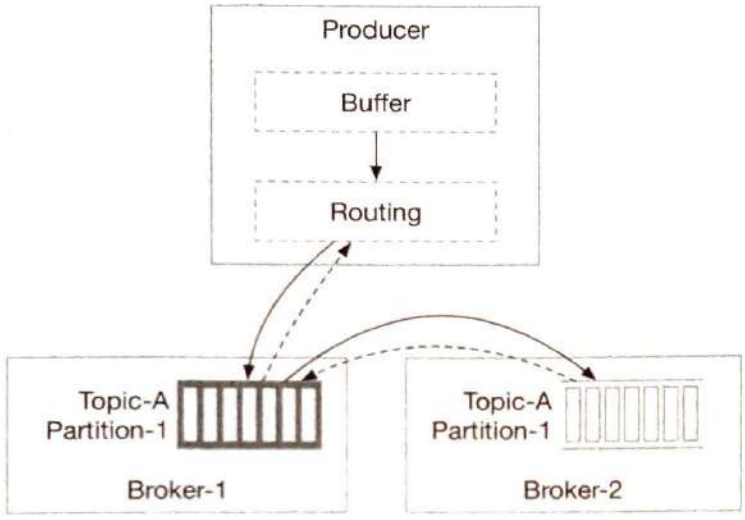


Figure 4.12: Producer with buffer and routing

The routing layer is wrapped into the producer and a buffer component is added to the producer. Both can be installed in the producer as part of the producer client library. This change brings several benefits:

- Fewer network hops mean lower latency.
- Producers can have their own logic to determine which partition the message should be sent to.
- Batching buffers messages in memory and sends out larger batches in a single request. This increases throughput.

The choice of the batch size is a classic tradeoff between throughput and latency (Figure

4.13). With a large batch size, the throughput increases but latency is higher, due to a longer wait time to accumulate the batch. With a small batch size, requests are sent sooner so the latency is lower, but throughput suffers. Producers can tune the batch size based on use cases.

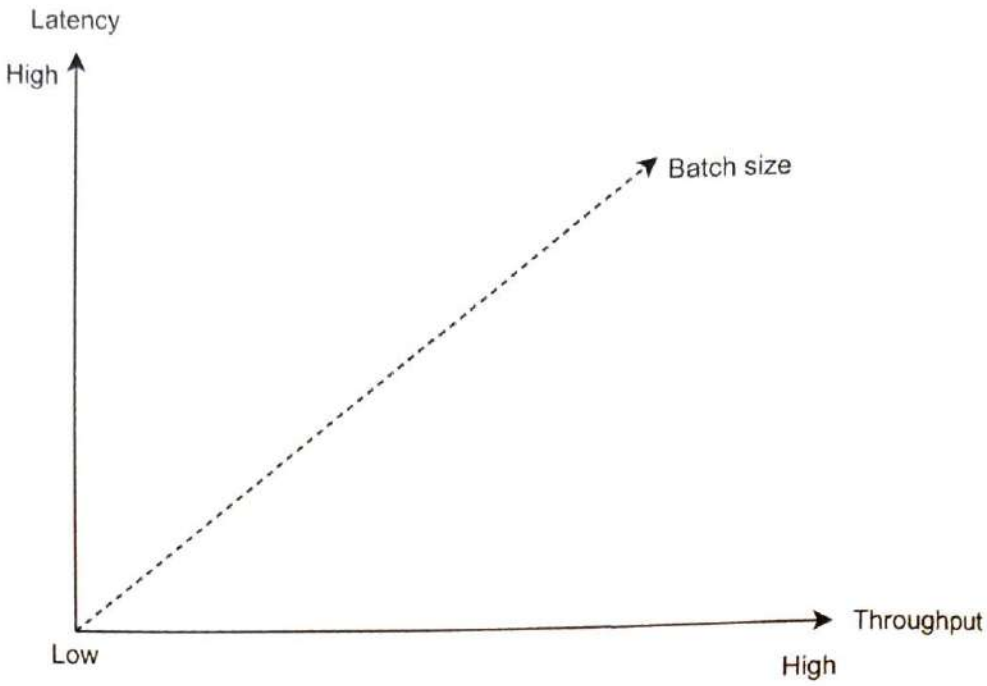


Figure 4.13: The choice of the batch size

Consumer flow

The consumer specifies its offset in a partition and receives back a chunk of events beginning from that position. An example is shown in Figure 4.14.

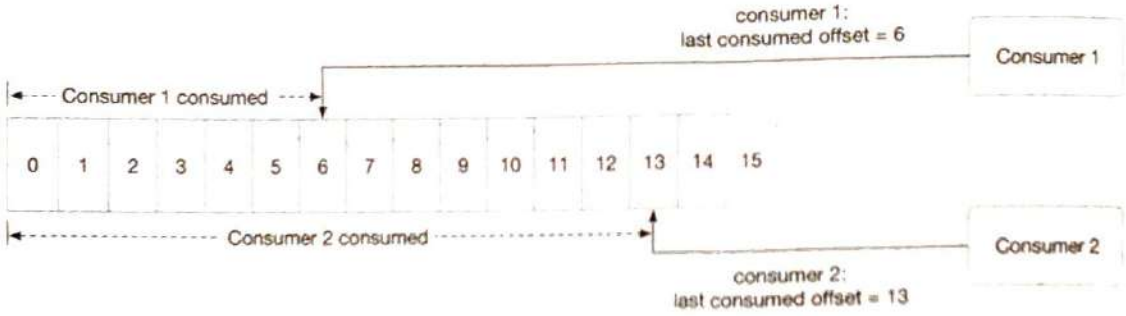


Figure 4.14: Consumer flow

Push vs pull

An important question to answer is whether brokers should push data to consumers, or if consumers should pull data from the brokers.

Push model

Pros:

- Low latency. The broker can push messages to the consumer immediately upon receiving them.

Cons:

- If the rate of consumption falls below the rate of production, consumers could be overwhelmed.
- It is difficult to deal with consumers with diverse processing power because the brokers control the rate at which data is transferred.

Pull model

Pros:

- Consumers control the consumption rate. We can have one set of consumers process messages in real-time and another set of consumers process messages in batch mode.
- If the rate of consumption falls below the rate of production, we can scale out the consumers, or simply catch up when it can.
- The pull model is more suitable for batch processing. In the push model, the broker has no knowledge of whether consumers will be able to process messages immediately. If the broker sends one message at a time to the consumer and the consumer is backed up, new messages will end up waiting in the buffer. A pull model pulls all available messages after the consumer's current position in the log (or up to the configurable max size). It is suitable for aggressive batching of data.

Cons:

- When there is no message in the broker, a consumer might still keep pulling data, wasting resources. To overcome this issue, many message queues support long polling mode, which allows pulls to wait a specified amount of time for new messages [6].

Based on these considerations, most message queues choose the pull model.

Figure 4.15 shows the workflow of the consumer pull model.

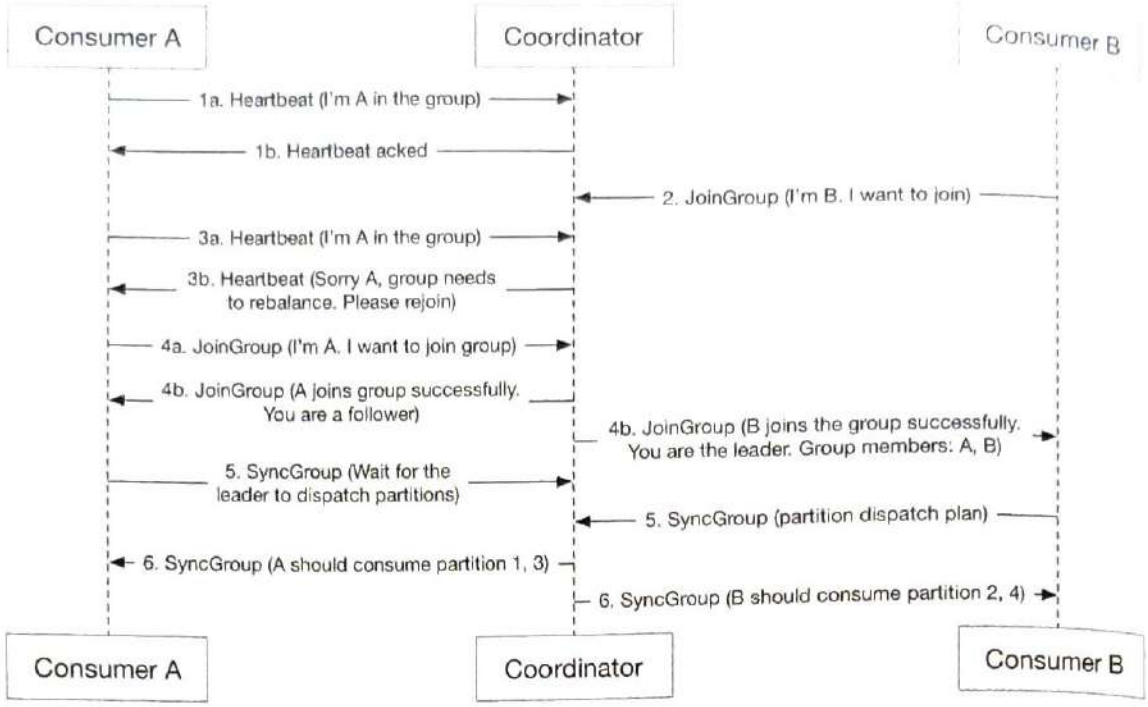


Figure 4.18: New consumer joins

1. Initially, only Consumer A is in the group. It consumes all the partitions and keeps the heartbeat with the coordinator.
2. Consumer B sends a request to join the group.
3. The coordinator knows it's time to rebalance, so it notifies all the consumers in the group in a passive way. When Consumer A's heartbeat is received by the coordinator, it asks Consumer A to rejoin the group.
4. Once all the consumers have rejoined the group, the coordinator chooses one of them as the leader and informs all the consumers about the election result.
5. The leader consumer generates the partition dispatch plan and sends it to the coordinator. Follower consumers ask the coordinator about the partition dispatch plan.
6. Consumers start consuming messages from newly assigned partitions.

Figure 4.19 shows the flow when an existing Consumer A leaves the group.

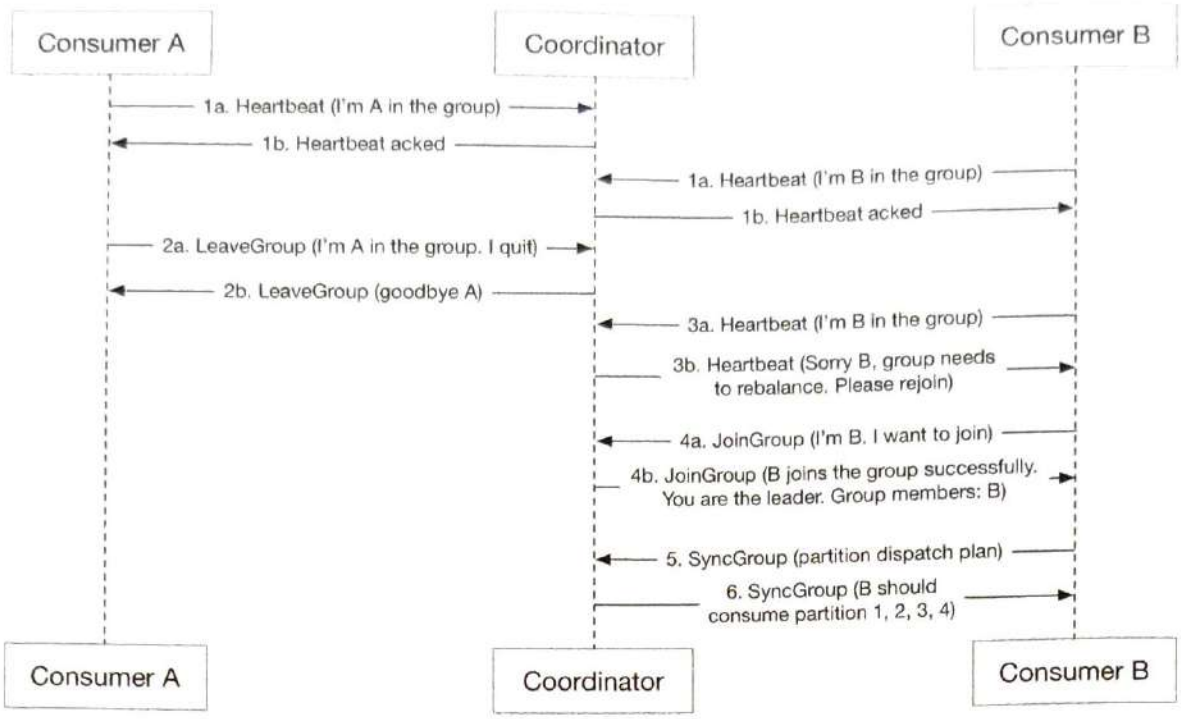


Figure 4.19: Existing consumer leaves

1. Consumer A and B are in the same consumer group.
2. Consumer A needs to be shut down, so it requests to leave the group.
3. The coordinator knows it's time to rebalance. When Consumer B's heartbeat is received by the coordinator, it asks Consumer B to rejoin the group.
4. The remaining steps are the same as the ones shown in Figure 4.18.

Figure 4.20 shows the flow when an existing Consumer A crashes.

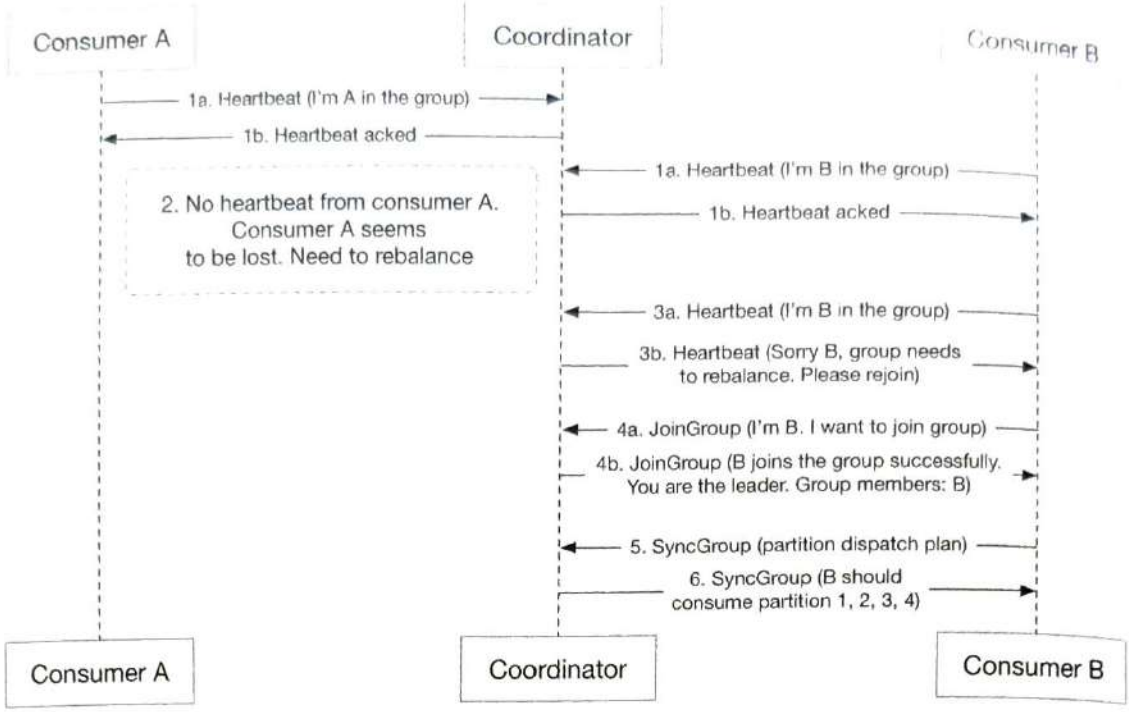


Figure 4.20: Existing consumer crashes

1. Consumer A and B keep heartbeats with the coordinator.
2. Consumer A crashes, so there is no heartbeat sent from Consumer A to the coordinator. Since the coordinator doesn't get any heartbeat signal within a specified amount of time from Consumer A, it marks the consumer as dead.
3. The coordinator triggers the rebalance process.
4. The following steps are the same as the ones in the previous scenario.

Now that we finished the detour on producer and consumer flows, let's come back and finish the deep dive on the rest of the message queue broker.

State storage

In the message queue broker, the state storage stores:

- The mapping between partitions and consumers.
- The last consumed offsets of consumer groups for each partition. As shown in Figure 4.21, the last consumed offset for consumer group-1 is 6 and the offset for consumer group-2 is 13.

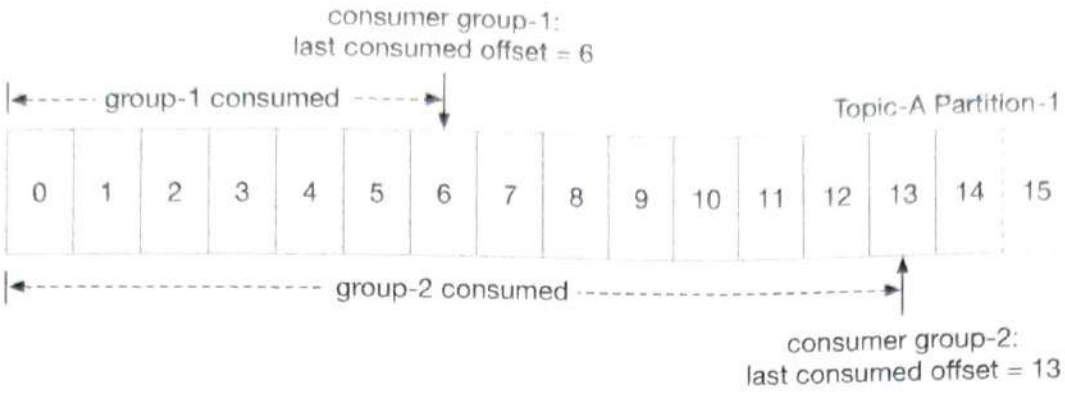


Figure 4.21: Last consumed offset of consumer groups

For example, as shown in Figure 4.21, a consumer in group-1 consumes messages from the partition in sequence and commits the consumed offset 6. This means all the messages before and at offset 6 are already consumed. If the consumer crashes, another new consumer in the same group will resume consumption by reading the last consumed offset from the state storage.

The data access patterns for consumer states are:

- Frequent read and write operations but the volume is not high.
- Data is updated frequently and is rarely deleted.
- Random read and write operations.
- Data consistency is important.

Lots of storage solutions can be used for storing the consumer state data. Considering the data consistency and fast read/write requirements, a KV store like ZooKeeper is a great choice. Kafka has moved the offset storage from ZooKeeper to Kafka brokers. Interested readers can read the reference material [8] to learn more.

Metadata storage

The metadata storage stores the configuration and properties of topics, including a number of partitions, retention period, and distribution of replicas.

Metadata does not change frequently and the data volume is small, but it has a high consistency requirement. ZooKeeper is a good choice for storing metadata.

ZooKeeper

By reading previous sections, you probably have already sensed that ZooKeeper is very helpful for designing a distributed message queue. If you are not familiar with it, ZooKeeper is an essential service for distributed systems offering a hierarchical key-value store. It is commonly used to provide a distributed configuration service, synchronization service, and naming registry [2].

ZooKeeper is used to simplify our design as shown in Figure 4.22.

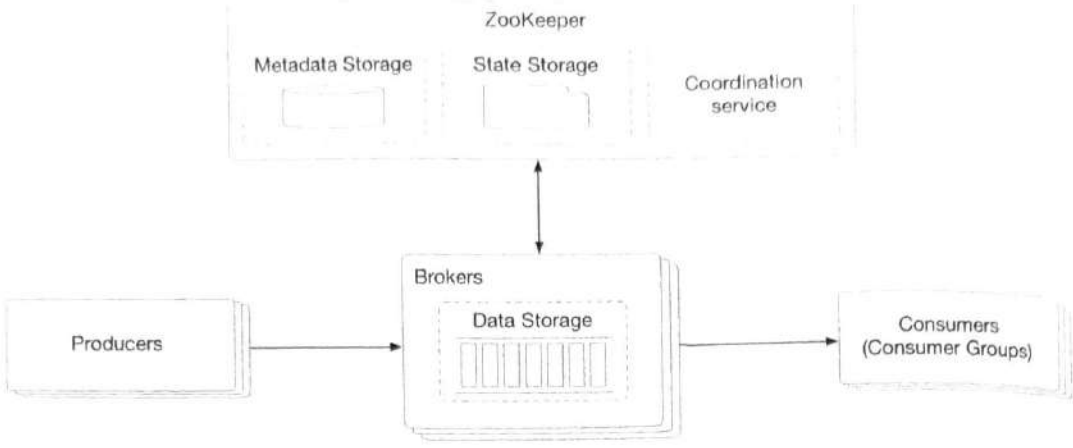


Figure 4.22: ZooKeeper

Let's briefly go over the change.

- Metadata and state storage are moved to ZooKeeper.
- The broker now only needs to maintain the data storage for messages.
- ZooKeeper helps with the leader election of the broker cluster.

Replication

In distributed systems, hardware issues are common and cannot be ignored. Data gets lost when a disk is damaged or fails permanently. Replication is the classic solution to achieve high availability.

As in Figure 4.23, each partition has 3 replicas, distributed across different broker nodes.

For each partition, the highlighted replicas are the leaders and the others are followers. Producers only send messages to the leader replica. The follower replicas keep pulling new messages from the leader. Once messages are synchronized to enough replicas, the leader returns an acknowledgment to the producer. We will go into detail about how to define "enough" in the In-sync Replicas section on page 113.

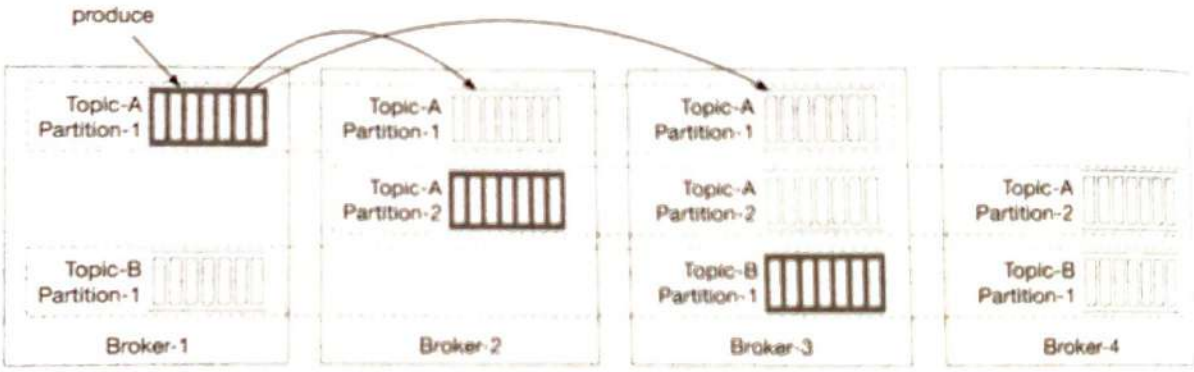


Figure 4.23: Replication

The distribution of replicas for each partition is called a replica distribution plan. For

example, the replica distribution plan in Figure 4.23 can be described as:

- Partition-1 of Topic-A: 3 replicas, leader in Broker-1, followers in Broker-2 and 3;
- Partition-2 of Topic-A: 3 replicas, leader in Broker-2, followers in Broker-3 and 4;
- Partition-1 of Topic-B: 3 replicas, leader in Broker-3, followers in Broker-4 and 1.

Who makes the replica distribution plan? It works as follows; with the help of the coordination service, one of the broker nodes is elected as the leader. It generates the replica distribution plan and persists the plan in metadata storage. All the brokers now can work according to the plan.

If you are interested in knowing more about replications, check out "Chapter 5. Replication" of the book "Design Data-Intensive Applications" [9].

In-sync replicas

We mentioned that messages are persisted in multiple partitions to avoid single node failure, and each partition has multiple replicas. Messages are only written to the leader, and followers synchronize data from the leader. One problem we need to solve is keeping them in sync.

In-sync replicas (ISR) refer to replicas that are "in-sync" with the leader. The definition of "in-sync" depends on the topic configuration. For example, if the value of replica.lag.max.messages is 4, it means that as long as the follower is behind the leader by no more than 3 messages, it will not be removed from ISR [10]. The leader is an ISR by default.

Let's use an example as shown in Figure 4.24 to shows how ISR works.

- The committed offset in the leader replica is 13. Two new messages are written to the leader, but not committed yet. Committed offset means that all messages before and at this offset are already synchronized to all the replicas in ISR.
- Replica-2 and replica-3 have fully caught up with the leader, so they are in ISR and can fetch new messages.
- Replica-4 did not fully catch up with the leader within the configured lag time, so it is not in ISR. When it catches up again, it can be added to ISR.

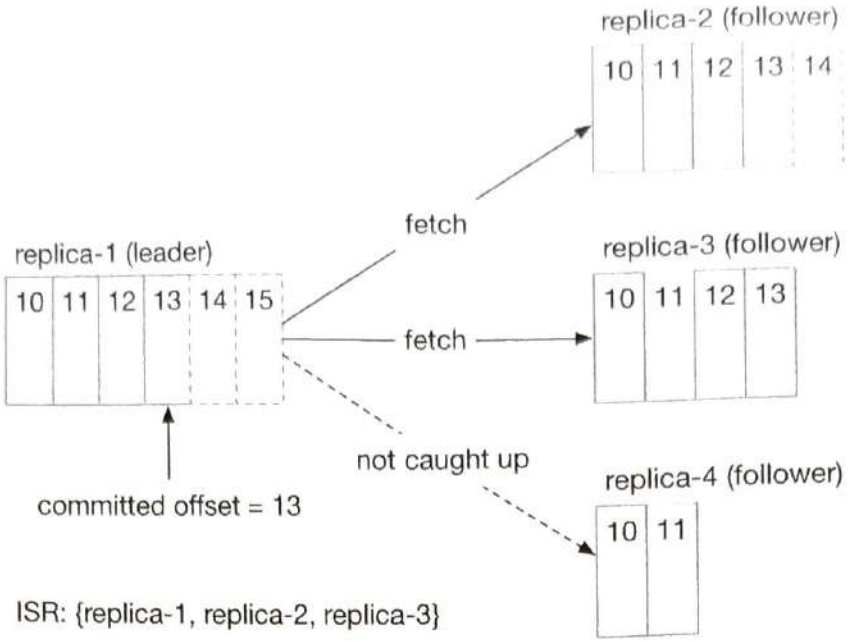


Figure 4.24: How ISR works

Why do we need ISR? The reason is that ISR reflects the trade-off between performance and durability. If producers don't want to lose any messages, the safest way to do that is to ensure all replicas are already in sync before sending an acknowledgment. But a slow replica will cause the whole partition to become slow or unavailable.

Now that we've discussed ISR, let's take a look at acknowledgment settings. Producers can choose to receive acknowledgments until the k number of ISRs has received the message, where k is configurable.

ACK=all

Figure 4.25 illustrates the case with ACK=all. With ACK=all, the producer gets an ACK when all ISRs have received the message. This means it takes a longer time to send a message because we need to wait for the slowest ISR, but it gives the strongest message durability.

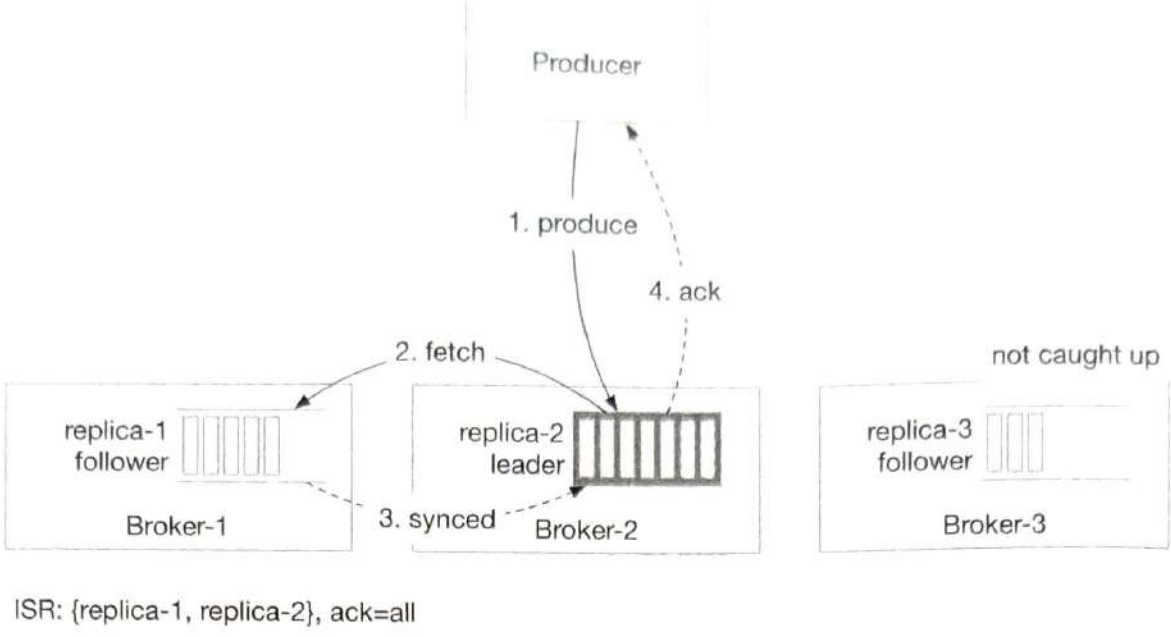


Figure 4.25: ACK=all

ACK=1

With ACK=1, the producer receives an ACK once the leader persists the message. The latency is improved by not waiting for data synchronization. If the leader fails immediately after a message is acknowledged but before it is replicated by follower nodes, then the message is lost. This setting is suitable for low latency systems where occasional data loss is acceptable.

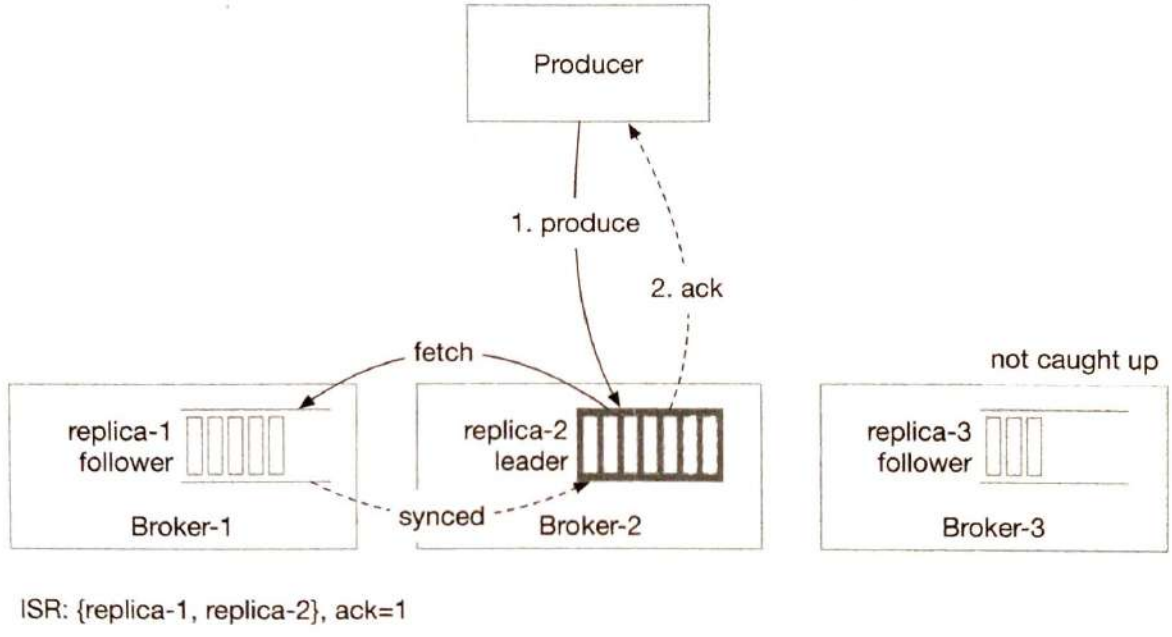


Figure 4.26: ACK=1

ACK=0

The producer keeps sending messages to the leader without waiting for any acknowledgment, and it never retries. This method provides the lowest latency at the cost of potential message loss. This setting might be good for use cases like collecting metrics or logging data since data volume is high and occasional data loss is acceptable.

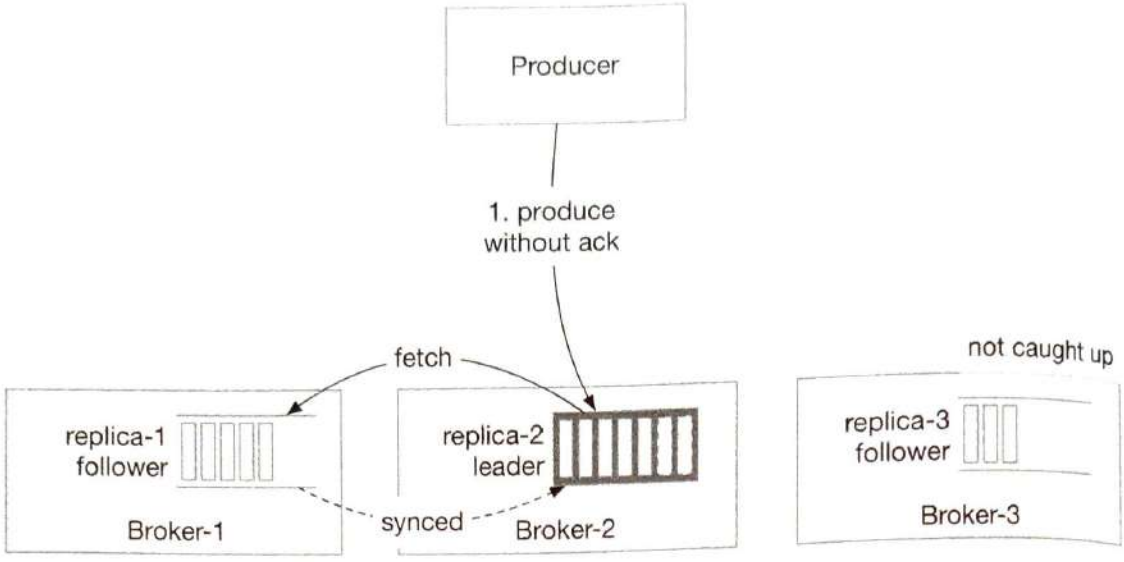


Figure 4.27: ACK=0

Configurable ACK allows us to trade durability for performance.

Now let's look at the consumer side. The easiest setup is to let consumers connect to a leader replica to consume messages.

You might be wondering if the leader replica would be overwhelmed by this design and why messages are not read from ISRs. The reasons are:

- Design and operational simplicity.
- Since messages in one partition are dispatched to only one consumer within a consumer group, this limits the number of connections to the leader replica.
- The number of connections to the leader replicas is usually not large as long as a topic is not super hot.
- If a topic is hot, we can scale by expanding the number of partitions and consumers.

In some scenarios, reading from the leader replica might not be the best option. For example, if a consumer is located in a different data center from the leader replica, the read performance suffers. In this case, it is worthwhile to enable consumers to read from the closest ISRs. Interested readers can check out the reference material about this [11].

ISR is very important. How does it determine if a replica is ISR or not? Usually, the leader

for every partition tracks the ISR list by computing the lag of every replica from itself. If you are interested in detailed algorithms, you can find the implementations in reference materials [12] [13].

Scalability

By now we have made great progress designing the distributed message queue system. In the next step, let's evaluate the scalability of different system components:

- Producers
- Consumers
- Brokers
- Partitions

Producer

The producer is conceptually much simpler than the consumer because it doesn't need group coordination. The scalability of producers can easily be achieved by adding or removing producer instances.

Consumer

Consumer groups are isolated from each other, so it is easy to add or remove a consumer group. Inside a consumer group, the rebalancing mechanism helps to handle the cases where a consumer gets added or removed, or when it crashes. With consumer groups and the rebalance mechanism, the scalability and fault tolerance of consumers can be achieved.

Broker

Before discussing scalability on the broker side, let's first consider the failure recovery of brokers.

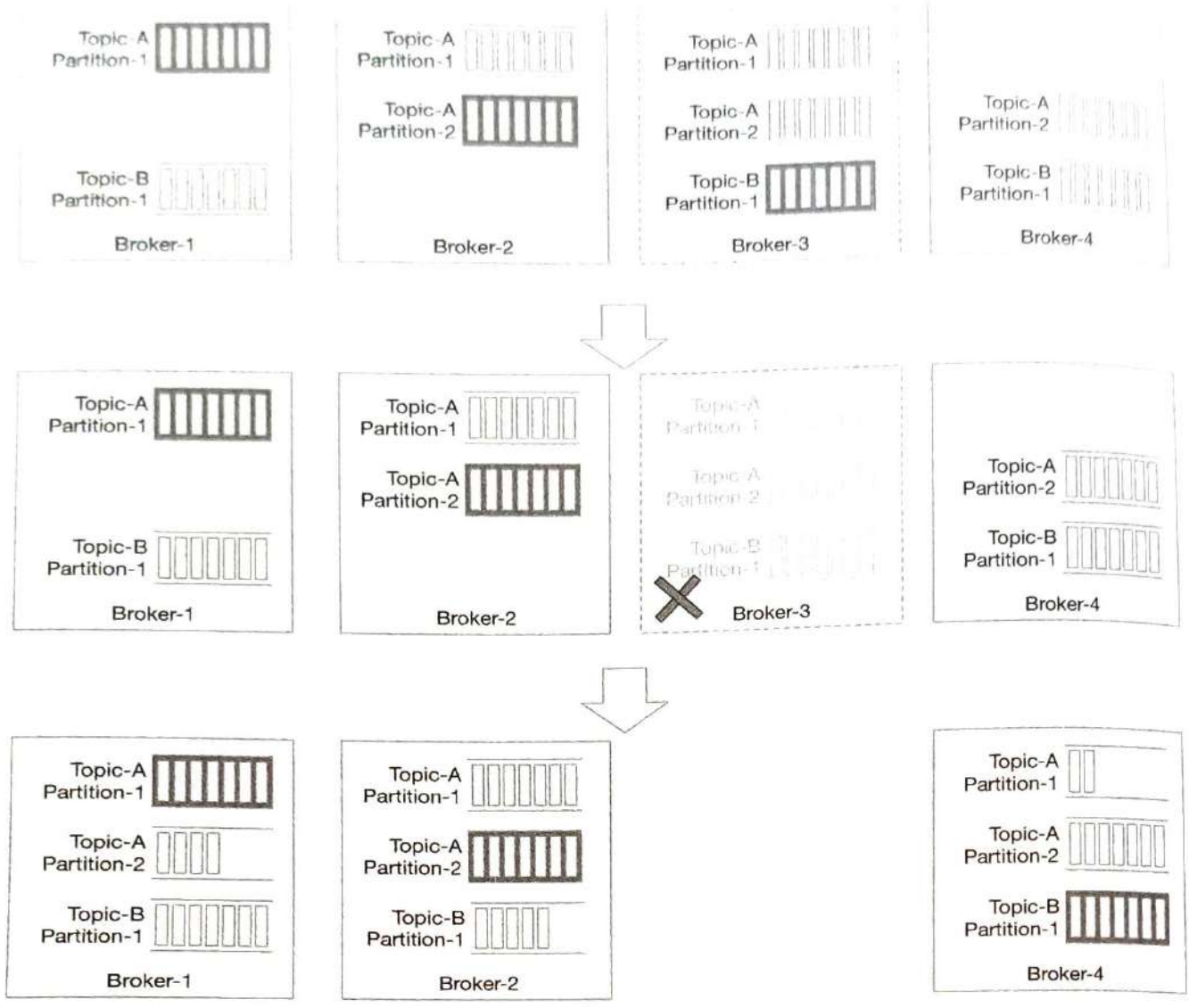


Figure 4.28: Broker node crashes

Let's use an example in Figure 4.28 to explain how failure recovery works.

1. Assume there are 4 brokers and the partition (replica) distribution plan is shown below:
    - Partition-1 of topic A: replicas in Broker-1 (leader), 2, and 3.
    - Partition-2 of topic A: replicas in Broker-2 (leader), 3, and 4.
    - Partition-1 of topic B: replicas in Broker-3 (leader), 4, and 1.
2. Broker-3 crashes, which means all the partitions on the node are lost. The partition distribution plan is changed to:
    - Partition-1 of topic A: replicas in Broker-1 (leader) and 2.
    - Partition-2 of topic A: replicas in Broker-2 (leader) and 4.
    - Partition-1 of topic B: replicas in Broker-4 and 1.

3. The broker controller detects Broker-3 is down and generates a new partition distribution plan for the remaining broker nodes:
    - Partition-1 of topic A: replicas in Broker-1 (leader), 2, and 4 (new).
    - Partition-2 of topic A: replicas in Broker-2 (leader), 4, and 1 (new).
    - Partition-1 of topic B: replicas in Broker-4 (leader), 1, and 2 (new).
4. The new replicas work as followers and catch up with the leader.

To make the broker fault-tolerant, here are additional considerations:

- The minimum number of ISRs specifies how many replicas the producer must receive before a message is considered to be successfully committed. The higher the number, the safer. But on the other hand, we need to balance latency and safety.
- If all replicas of a partition are in the same broker node, then we cannot tolerate the failure of this node. It is also a waste of resources to replicate data in the same node. Therefore, replicas should not be in the same node.
- If all the replicas of a partition crash, the data for that partition is lost forever. When choosing the number of replicas and replica locations, there's a trade-off between data safety, resource cost, and latency. It is safer to distribute replicas across data centers, but this will incur much more latency and cost, to synchronize data between replicas. As a workaround, data mirroring can help to copy data across data centers, but this is out of scope. The reference material [14] covers this topic.

Now let's get back to discussing the scalability of brokers. The simplest solution would be to redistribute the replicas when broker nodes are added or removed.

However, there is a better approach. The broker controller can temporarily allow more replicas in the system than the number of replicas in the config file. When the newly added broker catches up, we can remove the ones that are no longer needed. Let's use an example as shown in Figure 4.29 to understand the approach.

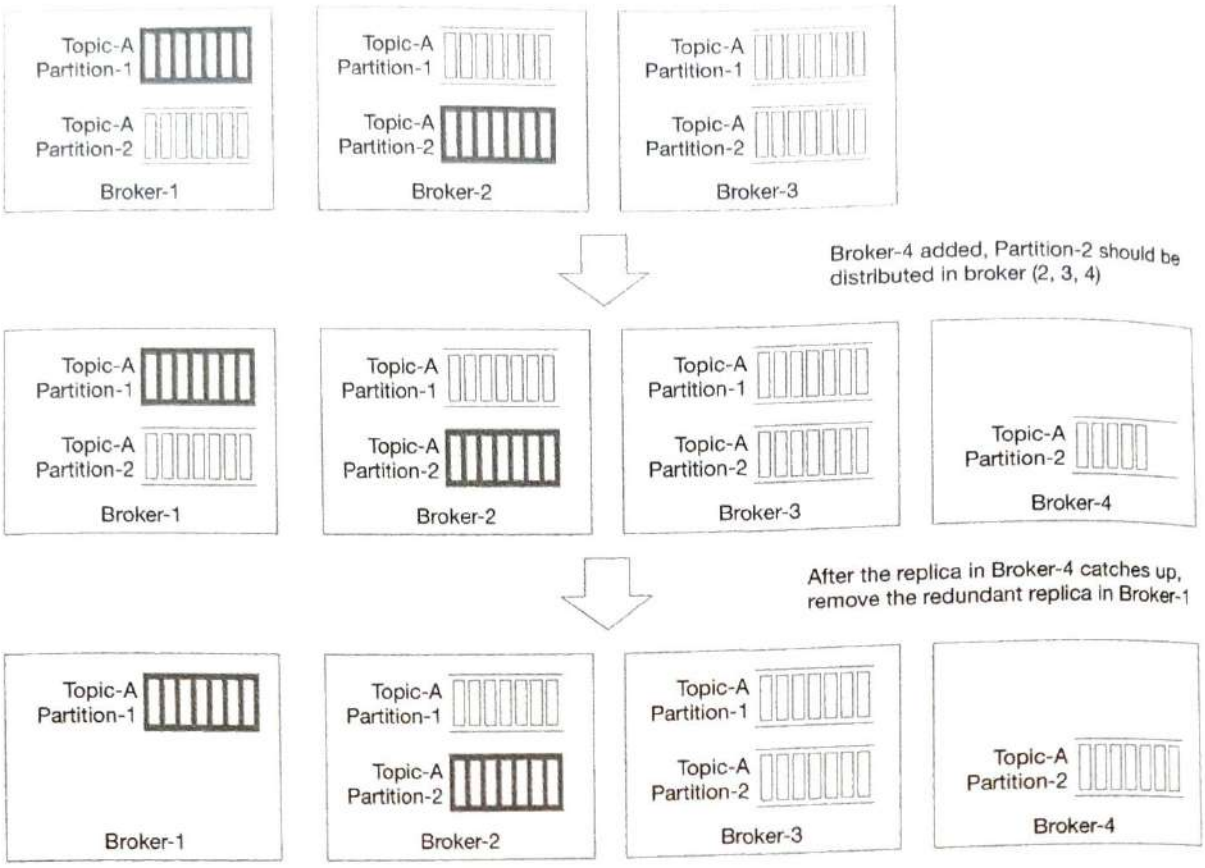


Figure 4.29: Add new broker node

1. The initial setup: 3 brokers, 2 partitions, and 3 replicas for each partition.
2. New Broker-4 is added. Assume the broker controller changes the replica distribution of Partition-2 to the broker (2, 3, 4). The new replica in Broker-4 starts to copy data from leader Broker-2. Now the number of replicas for Partition-2 is temporarily more than 3.
3. After the replica in Broker-4 catches up, the redundant partition in Broker-1 is gracefully removed.

By following this process, data loss while adding brokers can be avoided. A similar process can be applied to remove brokers safely.

Partition

For various operational reasons, such as scaling the topic, throughput tuning, balancing availability/ throughput, etc., we may change the number of partitions. When the number of partitions changes, the producer will be notified after it communicates with any broker, and the consumer will trigger consumer rebalancing. Therefore, it is safe for both the producer and consumer.

Now let's consider the data storage layer when the number of partitions changes. As in Figure 4.30, we have added a partition to the topic.

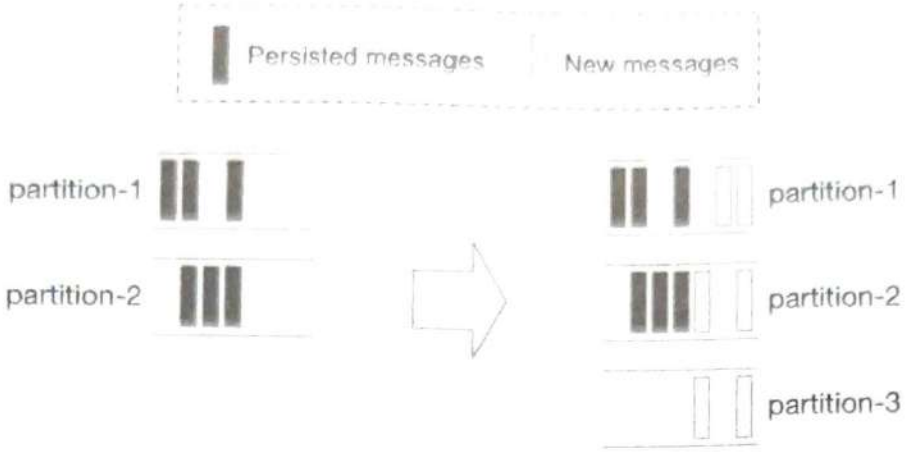


Figure 4.30: Partition increase

- Persisted messages are still in the old partitions, so there's no data migration.
- After the new partition (partition-3) is added, new messages will be persisted in all 3 partitions.

So it is straightforward to scale the topic by increasing partitions.

Decrease the number of partitions

Decreasing partitions is more complicated, as illustrated in Figure 4.31.

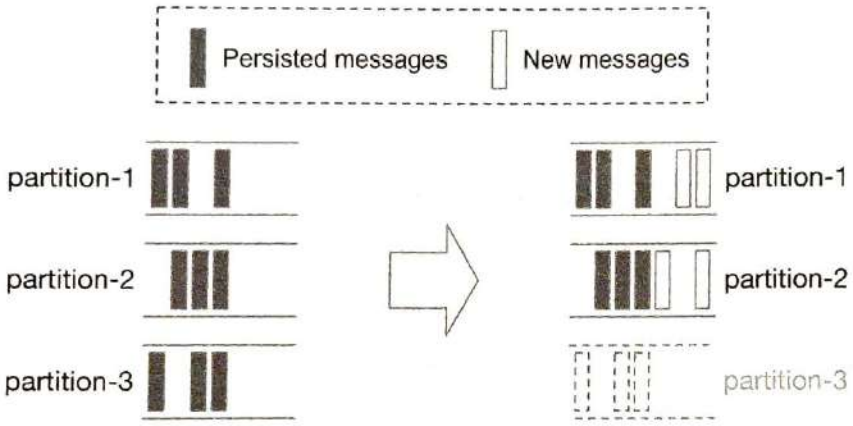


Figure 4.31: Partition decrease

- Partition-3 is decommissioned so new messages are only received by the remaining partitions (partition-1 and partition-2).
- The decommissioned partition (partition-3) cannot be removed immediately because data might be currently consumed by consumers for a certain amount of time. Only after the configured retention period passes, data can be truncated and storage space is freed up. Reducing partitions is not a shortcut to reclaiming data space.
- During this transitional period (while partition-3 is decommissioned), producers only send messages to the remaining 2 partitions, but consumers can still consume from all 3 partitions. After the retention period of the decommissioned partition expires,

consumer groups need rebalancing.

Data delivery semantics

Now that we understand the different components of a distributed message queue, let's discuss different delivery semantics: at-most once, at-least once, and exactly once.

At-most once

As the name suggests, at-most once means a message will be delivered not more than once. Messages may be lost but are not redelivered. This is how at-most once delivery works at the high level.

- The producer sends a message asynchronously to a topic without waiting for an acknowledgment (ACK=0). If message delivery fails, there is no retry.
- Consumer fetches the message and commits the offset before the data is processed. If the consumer crashes just after offset commit, the message will not be re-consumed.

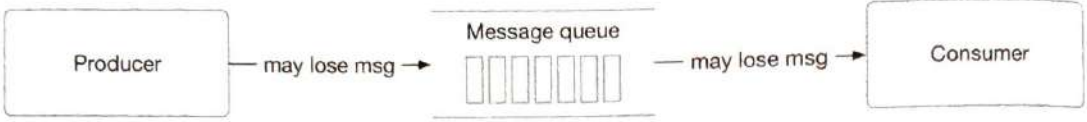


Figure 4.32: At-most once

It is suitable for use cases like monitoring metrics, where a small amount of data loss is acceptable.

At-least once

With this data delivery semantic, it's acceptable to deliver a message more than once, but no message should be lost. Here is how it works at a high level.

- Producer sends a message synchronously or asynchronously with a response callback, setting ACK=1 or ACK=all, to make sure messages are delivered to the broker. If the message delivery fails or timeouts, the producer will keep retrying.
- Consumer fetches the message and commits the offset only after the data is successfully processed. If the consumer fails to process the message, it will re-consume the message so there won't be data loss. On the other hand, if a consumer processes the message but fails to commit the offset to the broker, the message will be re-consumed when the consumer restarts, resulting in duplicates.
- A message might be delivered more than once to the brokers and consumers.

Figure 4.33: At-least once

Use cases: With at-least once, messages won't be lost but the same message might be delivered multiple times. While not ideal from a user perspective, at-least once delivery semantics are usually good enough for use cases where data duplication is not a big issue or deduplication is possible on the consumer side. For example, with a unique key in each message, a message can be rejected when writing duplicate data to the database.

Exactly once

Exactly once is the most difficult delivery semantic to implement. It is friendly to users, but it has a high cost for the system's performance and complexity.

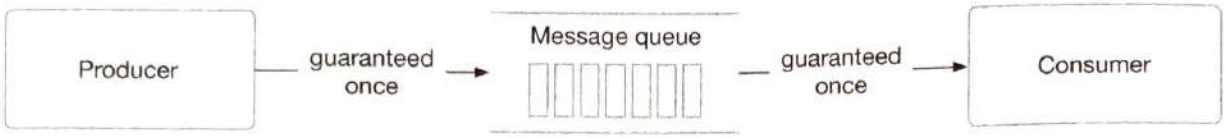


Figure 4.34: Exactly once

Use cases: Financial-related use cases (payment, trading, accounting, etc.). Exactly once is especially important when duplication is not acceptable and the downstream service or third party doesn't support idempotency.

Advanced features

In this section, we talk briefly about some advanced features, such as message filtering, delayed messages, and scheduled messages.

Message filtering

A topic is a logical abstraction that contains messages of the same type. However, some consumer groups may only want to consume messages of certain subtypes. For example, the ordering system sends all the activities about the order to a topic, but the payment system only cares about messages related to checkout and refund.

One option is to build a dedicated topic for the payment system and another topic for the ordering system. This method is simple, but it might raise some concerns.

- What if other systems ask for different subtypes of messages? Do we need to build dedicated topics for every single consumer request?
- It is a waste of resources to save the same messages on different topics.
- The producer needs to change every time a new consumer requirement comes, as the producer and consumer are now tightly coupled.

Therefore, we need to resolve this requirement using a different approach. Luckily, mes-

sage filtering comes to the rescue.

A naive solution for message filtering is that the consumer fetches the full set of messages and filters out unnecessary messages during processing time. This approach is flexible but introduces unnecessary traffic that will affect system performance.

A better solution is to filter messages on the broker side so that consumers will only get messages they care about. Implementing this requires some careful consideration. If data filtering requires data decryption or deserialization, it will degrade the performance of the brokers. Additionally, if messages contain sensitive data, they should not be readable in the message queue.

Therefore, the filtering logic in the broker should not extract the message payload. It is better to put data used for filtering into the metadata of a message, which can be efficiently read by the broker. For example, we can attach a tag to each message. With a message tag, a broker can filter messages in that dimension. If more tags are attached, the messages can be filtered in multiple dimensions. Therefore, a list of tags can support most of the filtering requirements. To support more complex logic such as mathematical formulae, the broker will need a grammar parser or a script executor, which might be too heavyweight for the message queue.

With tags attached to each message, a consumer can subscribe to messages based on the specified tag, as shown in Figure 4.35. Interested readers can refer to the reference material [15].

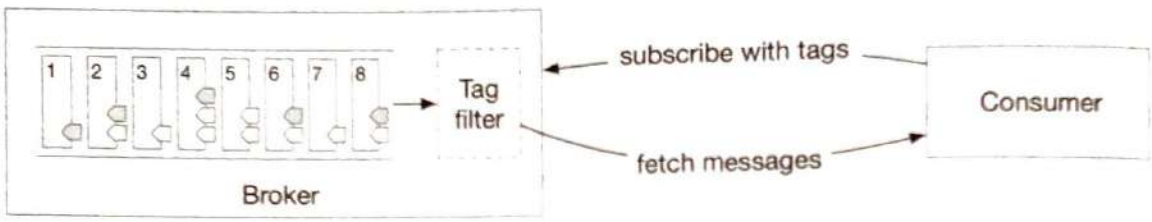


Figure 4.35: Message filtering by tags

Delayed messages & scheduled messages

Sometimes you want to delay the delivery of messages to a consumer for a specified period of time. For example, an order should be closed if not paid within 30 minutes after the order is created. A delayed verification message (check if the payment is completed) is sent immediately but is delivered to the consumer 30 minutes later. When the consumer receives the message, it checks the payment status. If the payment is not completed, the order will be closed. Otherwise, the message will be ignored.

Different from sending instant messages, we can send delayed messages to temporary storage on the broker side instead of to the topics immediately, and then deliver them to the topics when time's up. The high-level design for this is shown in Figure 4.36.

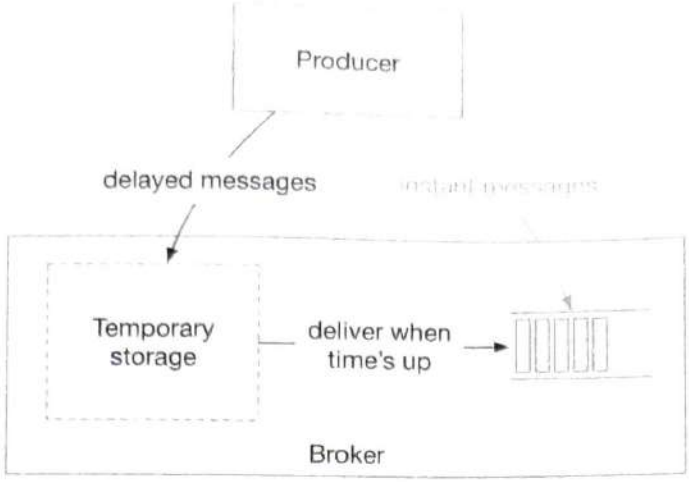


Figure 4.36: Delayed messages

Core components of the system include the temporary storage and the timing function.

- The temporary storage can be one or more special message topics.
- The timing function is out of scope, but here are 2 popular solutions:
  - Dedicated delay queues with predefined delay levels [16]. For example, RocketMQ doesn't support delayed messages with arbitrary time precision, but delayed messages with specific levels are supported. Message delay levels are 1s, 5s, 10s, 30s, 1m, 2m, 3m, 4m, 6m, 8m, 9m, 10m, 20m, 30m, 1h, and 2h.
  - Hierarchical time wheel [17].

A scheduled message means a message should be delivered to the consumer at the scheduled time. The overall design is very similar to delayed messages.

Step 4 - Wrap Up

In this chapter, we have presented the design of a distributed message queue with some advanced features commonly found in data streaming platforms. If there is extra time at the end of the interview, here are some additional talking points:

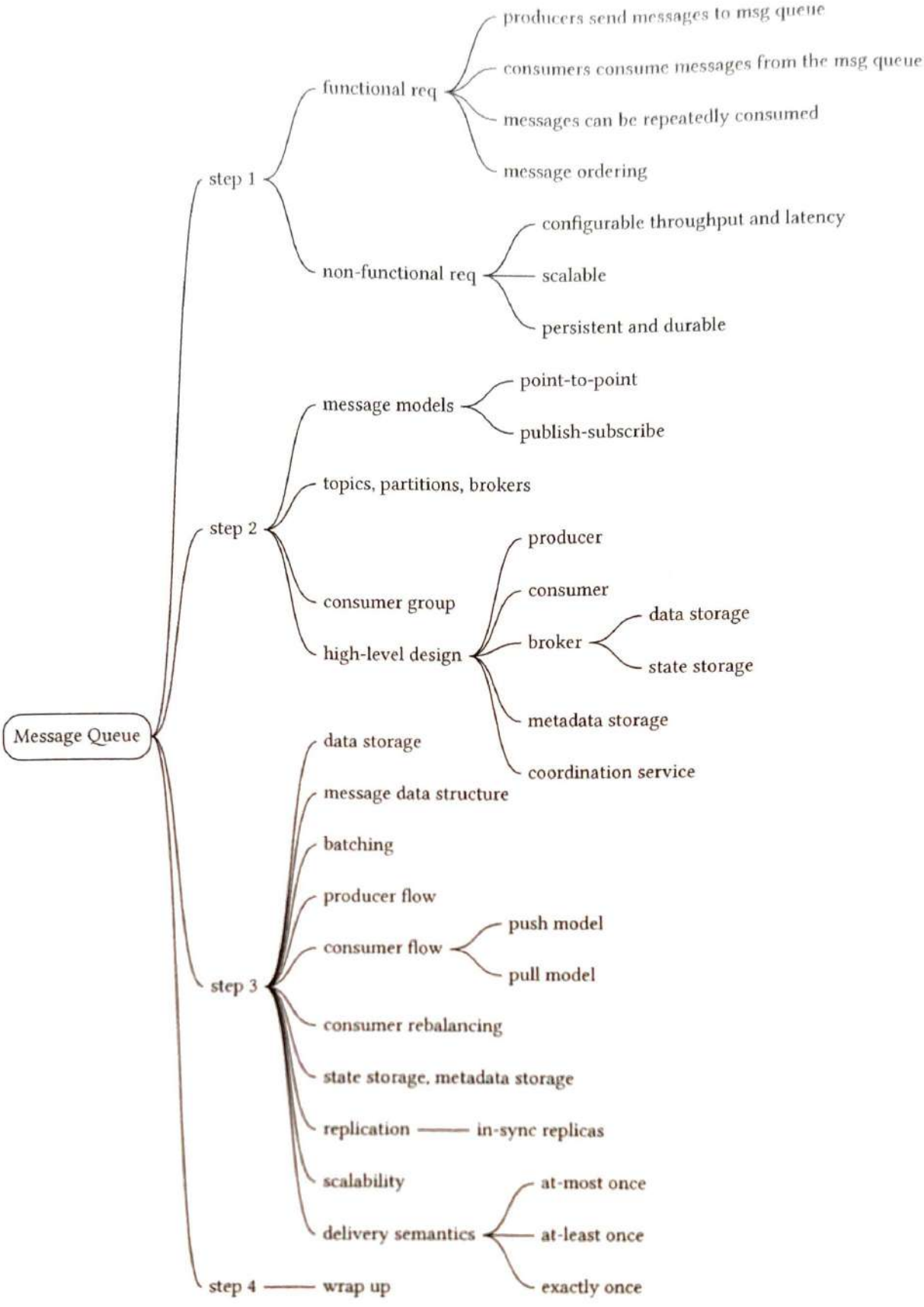
- Protocol: it defines rules, syntax, and APIs on how to exchange information and transfer data between different nodes. In a distributed message queue, the protocol should be able to:
  - Cover all the activities such as production, consumption, heartbeat, etc.
  - Effectively transport data with large volumes.
  - Verify the integrity and correctness of the data.

  Some popular protocols include Advanced Message Queuing Protocol (AMQP) [18] and Kafka protocol [19].

- Retry consumption. If some messages cannot be consumed successfully, we need to retry the operation. In order not to block incoming messages, how can we retry the operation after a certain time period? One idea is to send failed messages to a dedicated retry topic, so they can be consumed later.
- Historical data archive. Assume there is a time-based or capacity-based log retention mechanism. If a consumer needs to replay some historical messages that are already truncated, how can we do it? One possible solution is to use storage systems with large capacities, such as HDFS [20] or object storage, to store historical data.

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

Chapter Summary



Reference Material

[1] Queue Length Limit. https://www.rabbitmq.com/maxlength.html.

[2] Apache ZooKeeper - Wikipedia. https://en.wikipedia.org/wiki/Apache_ZooKeeper.

[3] etcd. https://etcd.io/.

[4] MySQL. https://www.mysql.com/.

[5] Comparison of disk and memory performance. https://deliveryimages.acm.org/10.1145/1570000/1563874/jacobs3.jpg.

[6] Push vs. pull. https://kafka.apache.org/documentation/#design_pull.

[7] Kafka 2.0 Documentation. https://kafka.apache.org/20/documentation.html#consumerconfigs.

[8] Kafka No Longer Requires ZooKeeper. https://towardsdatascience.com/kafka-no-longer-requires-zookeeper-ebfbf3862104.

[9] Martin Kleppmann. Replication. In *Designing Data-Intensive Applications*, pages 151–197. O'Reilly Media, 2017.

[10] ISR in Apache Kafka. https://www.cloudkarafka.com/blog/what-does-in-sync-in-apache-kafka-really-mean.html.

[11] Global map in a geographic Coordinate Reference System. https://cwiki.apache.org/confluence/display/KAFKA/KIP-392%3A+Allow+consumers+to+fetch+from+closest+replica.

[12] Hands-free Kafka Replication. https://www.confluent.io/blog/hands-free-kafka-replication-a-lesson-in-operational-simplicity/.

[13] Kafka high watermark. https://rongxinblog.wordpress.com/2016/07/29/kafka-high-watermark/.

[14] Kafka mirroring. https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=27846330.

[15] Message filtering in RocketMQdtree. https://partners-intl.aliyun.com/help/doc-detail/29543.htm.

[16] Scheduled messages and delayed messages in Apache RocketMQ. https://partners-intl.aliyun.com/help/doc-detail/43349.htm.

[17] Hashed and hierarchical timing wheels. http://www.cs.columbia.edu/~nahum/w6998/papers/sosp87-timing-wheels.pdf.

[18] Advanced Message Queuing Protocol. https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol.

[19] Kafka protocol guide. https://kafka.apache.org/protocol.

[20] HDFS. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

5 Metrics Monitoring and Alerting System

In this chapter, we explore the design of a scalable metrics monitoring and alerting system. A well-designed monitoring and alerting system plays a key role in providing clear visibility into the health of the infrastructure to ensure high availability and reliability.

Figure 5.1 shows some of the most popular metrics monitoring and alerting services in the marketplace. In this chapter, we design a similar service that can be used internally by a large company.

Figure 5.1: Popular metrics monitoring and alerting services

Step 1 - Understand the Problem and Establish Design Scope

A metrics monitoring and alerting system can mean many different things to different companies, so it is essential to nail down the exact requirements first with the interviewer. For example, you do not want to design a system that focuses on logs such as web server error or access logs if the interviewer has only infrastructure metrics in mind.

Let's first fully understand the problem and establish the scope of the design before diving into the details.

Candidate: Who are we building the system for? Are we building an in-house system for a large corporation like Facebook or Google, or are we designing a SaaS service like Datadog [1], Splunk [2], etc?
Interviewer: That's a great question. We are building it for internal use only.

Candidate: Which metrics do we want to collect?
Interviewer: We want to collect operational system metrics. These can be low-level usage data of the operating system, such as CPU load, memory usage, and disk space consumption. They can also be high-level concepts such as requests per second of a service or the running server count of a web pool. Business metrics are not in the scope of this design.

Candidate: What is the scale of the infrastructure we are monitoring with this system?
Interviewer: 100 million daily active users, 1,000 server pools, and 100 machines per pool.

Candidate: How long should we keep the data?
Interviewer: Let's assume we want 1 year retention.

Candidate: May we reduce the resolution of the metrics data for long-term storage?
Interviewer: That's a great question. We would like to be able to keep newly received data for 7 days. After 7 days, you may roll them up to a 1 minute resolution for 30 days. After 30 days, you may further roll them up at a 1 hour resolution.

Candidate: What are the supported alert channels?
Interviewer: Email, phone, PagerDuty [3], or webhooks (HTTP endpoints).

Candidate: Do we need to collect logs, such as error log or access log?
Interviewer: No.

Candidate: Do we need to support distributed system tracing?
Interviewer: No.

High-level requirements and assumptions

Now you have finished gathering requirements from the interviewer and have a clear scope of the design. The requirements are:

- The infrastructure being monitored is large-scale.
  - 100 million daily active users
  - Assume we have 1,000 server pools, 100 machines per pool, 100 metrics per machine ⇒ ~ 10 million metrics
  - 1 year data retention
  - Data retention policy: raw form for 7 days, 1 minute resolution for 30 days, 1 hour resolution for 1 year.
- A variety of metrics can be monitored, for example:
  - CPU usage

- Request count
- Memory usage
- Message count in message queues

Non-functional requirements

- Scalability. The system should be scalable to accommodate growing metrics and alert volume.
- Low latency. The system needs to have low query latency for dashboards and alerts.
- Reliability. The system should be highly reliable to avoid missing critical alerts.
- Flexibility. Technology keeps changing, so the pipeline should be flexible enough to easily integrate new technologies in the future.

Which requirements are out of scope?

- Log monitoring. The Elasticsearch, Logstash, Kibana (ELK) stack is very popular for collecting and monitoring logs [4].
- Distributed system tracing [5] [6]. Distributed tracing refers to a tracing solution that tracks service requests as they flow through distributed systems. It collects data as requests go from one service to another.

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

In this section, we discuss some fundamentals of building the system, the data model, and the high-level design.

Fundamentals

A metrics monitoring and alerting system generally contains five components, as illustrated in Figure 5.2.

- Data collection: collect metric data from different sources.
- Data transmission: transfer data from sources to the metrics monitoring system.
- Data storage: organize and store incoming data.
- Alerting: analyze incoming data, detect anomalies, and generate alerts. The system must be able to send alerts to different communication channels.
- Visualization: present data in graphs, charts, etc. Engineers are better at identifying patterns, trends, or problems when data is presented visually, so we need visualization functionality.

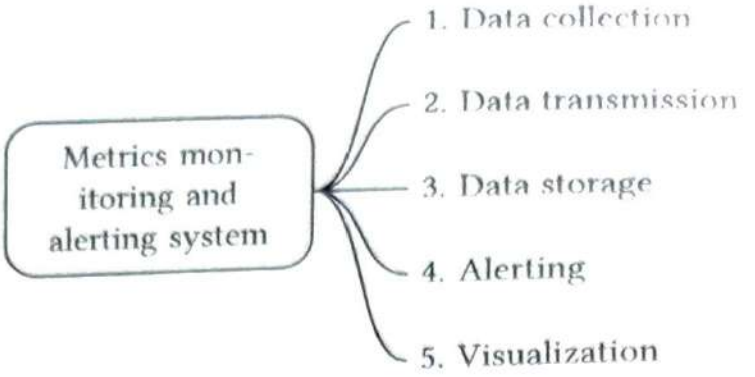


Figure 5.2: Five components of the system

Data model

Metrics data is usually recorded as a time series that contains a set of values with their associated timestamps. The series itself can be uniquely identified by its name, and optionally by a set of labels.

Let's take a look at two examples.

Example 1:

What is the CPU load on production server instance `i631` at `20:00`?

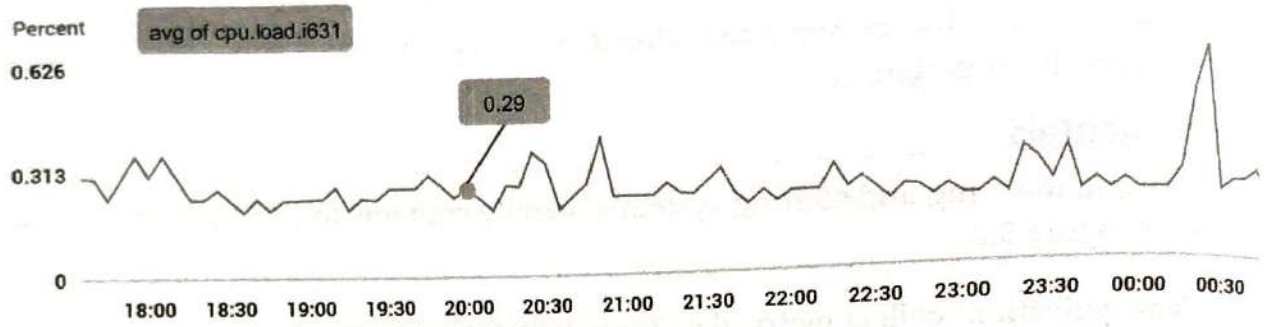


Figure 5.3: Popular metrics monitoring and alerting services

The data point highlighted in Figure 5.3 can be represented by Table 5.1.

metric_name	cpu.load
labels	host:i631,env:prod
timestamp	1613707265
value	0.29

Table 5.1: The data point represented by a table

In this example, the time series is represented by the metric name, the labels (`host:i631,env:prod`), and a single point value at a specific time.

Example 2:

What is the average CPU load across all web servers in the us-west region for the last 10 minutes? Conceptually, we would pull up something like this from storage where the metric name is `CPU.load` and the region label is us-west:

```
CPU.load host=webserver01,region=us-west 1613707265 50
CPU.load host=webserver01,region=us-west 1613707265 62
CPU.load host=webserver02,region=us-west 1613707265 43
CPU.load host=webserver02,region=us-west 1613707265 53
...
CPU.load host=webserver01,region=us-west 1613707265 76
CPU.load host=webserver01,region=us-west 1613707265 83
```

The average CPU load could be computed by averaging the values at the end of each line. The format of the lines in the above example is called the line protocol. It is a common input format for many monitoring software in the market. Prometheus [7] and OpenTSDB [8] are two examples.

Every time series consists of the following [9]:

Name	Type
A metric name	String
A set of tags/labels	List of <key:value> pairs
An array of values and their timestamps	An array of <value, timestamp> pairs

Table 5.2: Time series

Data access pattern

In Figure 5.4, each label on the y-axis represents a time series (uniquely identified by the names and labels) while the x-axis represents time.

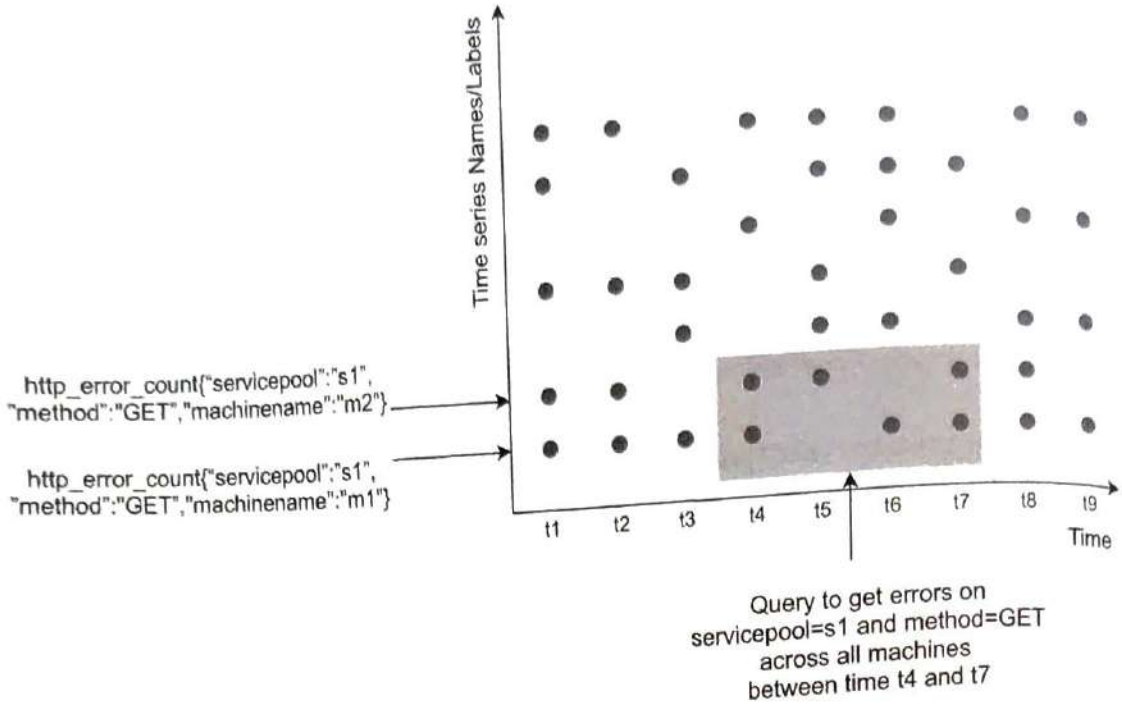


Figure 5.4: Data access pattern

The write load is heavy. As you can see, there can be many time-series data points written at any moment. As we mentioned in the "High-level requirements" section on page 132, about 10 million operational metrics are written per day, and many metrics are collected at high frequency, so the traffic is undoubtedly write-heavy.

At the same time, the read load is spiky. Both visualization and alerting services send queries to the database, and depending on the access patterns of the graphs and alerts, the read volume could be bursty.

In other words, the system is under constant heavy write load, while the read load is spiky.

Data storage system

The data storage system is the heart of the design. It's not recommended to build your own storage system or use a general-purpose storage system (for example, MySQL [10]) for this job.

A general-purpose database, in theory, could support time-series data, but it would require expert-level tuning to make it work at our scale. Specifically, a relational database is not optimized for operations you would commonly perform against time-series data. For example, computing the moving average in a rolling time window requires complicated SQL that is difficult to read (there is an example of this in the deep dive section). Besides, to support tagging/labeling data, we need to add an index for each tag. Moreover, a general-purpose relational database does not perform well under constant heavy write load. At our scale, we would need to expend significant effort in tuning the database, and even then, it might not perform well.

How about NoSQL? In theory, a few NoSQL databases on the market could handle time-series data effectively. For example, Cassandra and Bigtable [11] can both be used for time series data. However, this would require deep knowledge of the internal workings of each NoSQL to devise a scalable schema for effectively storing and querying time-series data. With industrial-scale time-series databases readily available, using a general-purpose NoSQL database is not appealing.

There are many storage systems available that are optimized for time-series data. The optimization lets us use far fewer servers to handle the same volume of data. Many of these databases also have custom query interfaces specially designed for the analysis of time-series data that are much easier to use than SQL. Some even provide features to manage data retention and data aggregation. Here are a few examples of time-series databases.

OpenTSDB is a distributed time-series database, but since it is based on Hadoop and HBase, running a Hadoop/HBase cluster adds complexity. Twitter uses MetricsDB [12], and Amazon offers Timestream as a time-series database [13]. According to DB-engines [14], the two most popular time-series databases are InfluxDB [15] and Prometheus, which are designed to store large volumes of time-series data and quickly perform real-time analysis on that data. Both of them primarily rely on an in-memory cache and on-disk storage. And they both handle durability and performance quite well. As shown in Figure 5.5, an InfluxDB with 8 cores and 32GB RAM can handle over 250,000 writes per second.

vCPU or CPU	RAM	IOPS	Writes per second	Queries* per second	Unique series
2-4 cores	2-4 GB	500	< 5,000	< 5	< 100,000
4-6 cores	8-32 GB	500-1000	< 250,000	< 25	< 1,000,000
8+ cores	32+ GB	1000+	> 250,000	> 25	> 1,000,000

Figure 5.5: InfluxDb benchmarking

Since a time-series database is a specialized database, you are not expected to understand the internals in an interview unless you explicitly mentioned it in your resume. For the purpose of an interview, it's important to understand the metrics data are time-series in nature and we can select time-series databases such as InfluxDB for storage to store them.

Another feature of a strong time-series database is efficient aggregation and analysis of a large amount of time-series data by labels, also known as tags in some databases. For example, InfluxDB builds indexes on labels to facilitate the fast lookup of time-series by labels [15]. It provides clear best-practice guidelines on how to use labels, without overloading the database. The key is to make sure each label is of low cardinality (having

a small set of possible values). This feature is critical for visualization, and it would take a lot of effort to build this with a general-purpose database.

High-level design

The high-level design diagram is shown in Figure 5.6.

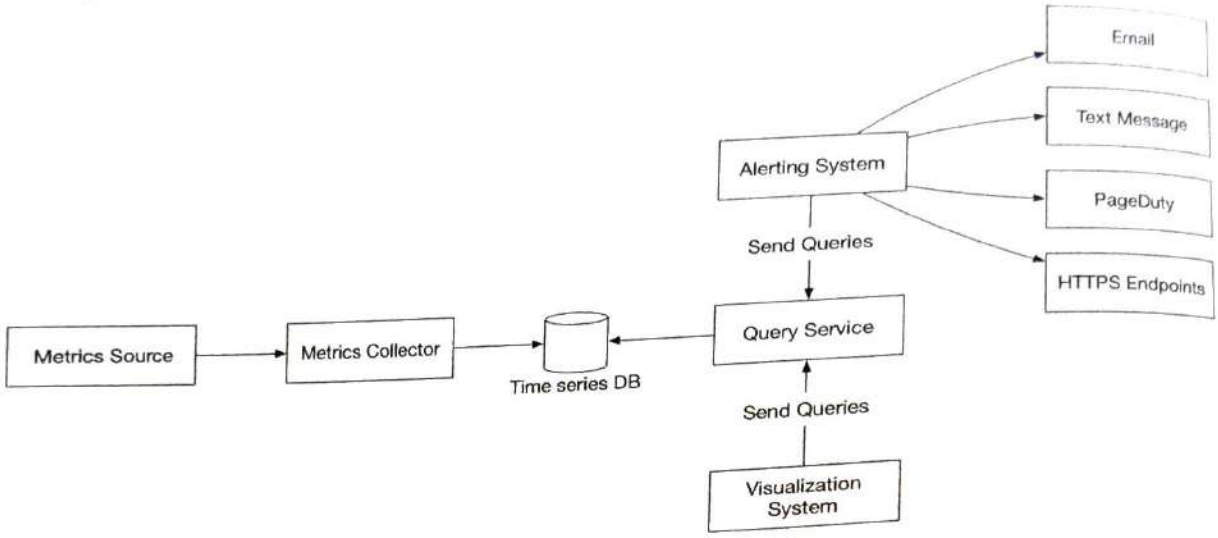


Figure 5.6: High-level design

- **Metrics source.** This can be application servers, SQL databases, message queues, etc.
- **Metrics collector.** It gathers metrics data and writes data into the time-series database.
- **Time-series database.** This stores metrics data as time series. It usually provides a custom query interface for analyzing and summarizing a large amount of time-series data. It maintains indexes on labels to facilitate the fast lookup of time-series data by labels.
- **Query service.** The query service makes it easy to query and retrieve data from the time-series database. This should be a very thin wrapper if we choose a good time-series database. It could also be entirely replaced by the time-series database's own query interface.
- **Alerting system.** This sends alert notifications to various alerting destinations.
- **Visualization system.** This shows metrics in the form of various graphs/charts.

Step 3 - Design Deep Dive

In a system design interview, candidates are expected to dive deep into a few key components or flows. In this section, we investigate the following topics in detail:

- Metrics collection
- Scaling the metrics transmission pipeline

- Query service
- Storage layer
- Alerting system
- Visualization system

Metrics collection

For metrics collection like counters or CPU usage, occasional data loss is not the end of the world. It's acceptable for clients to fire and forget. Now let's take a look at the metrics collection flow. This part of the system is inside the dashed box (Figure 5.7).

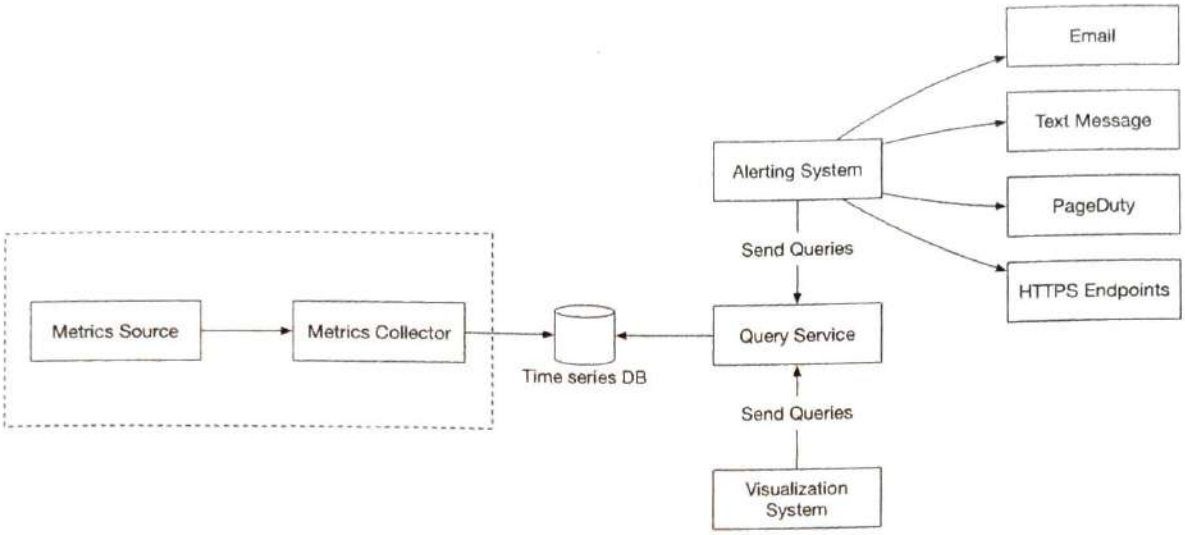


Figure 5.7: Metrics collection flow

Pull vs push models

There are two ways metrics data can be collected, pull or push. It is a routine debate as to which one is better and there is no clear answer. Let's take a close look.

Pull model

Figure 5.8 shows data collection with a pull model over HTTP. We have dedicated metric collectors which pull metrics values from the running applications periodically.

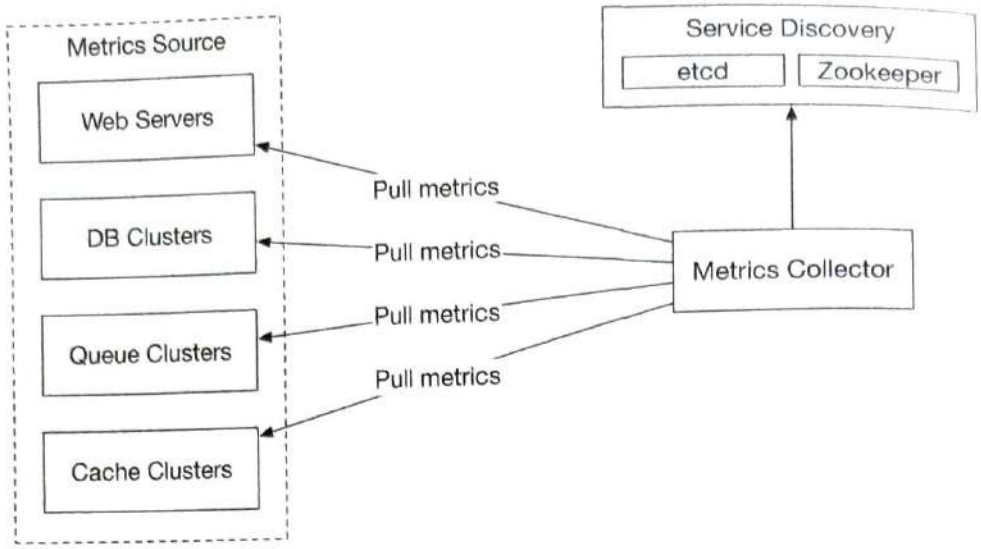


Figure 5.8: Pull model

In this approach, the metrics collector needs to know the complete list of service endpoints to pull data from. One naive approach is to use a file to hold DNS/IP information for every service endpoint on the "metric collector" servers. While the idea is simple, this approach is hard to maintain in a large-scale environment where servers are added or removed frequently, and we want to ensure that metric collectors don't miss out on collecting metrics from any new servers. The good news is that we have a reliable, scalable, and maintainable solution available through Service Discovery, provided by etcd [16], ZooKeeper [17], etc., wherein services register their availability and the metrics collector can be notified by the Service Discovery component whenever the list of service endpoints changes.

Service discovery contains configuration rules about when and where to collect metrics as shown in Figure 5.9.

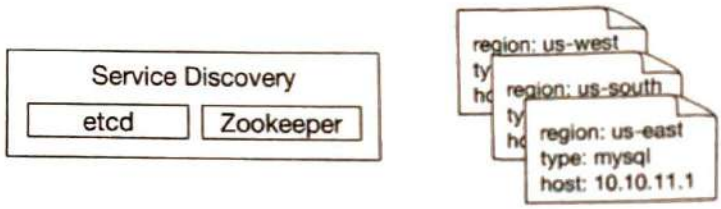


Figure 5.9: Service discovery

Figure 5.10 explains the pull model in detail.

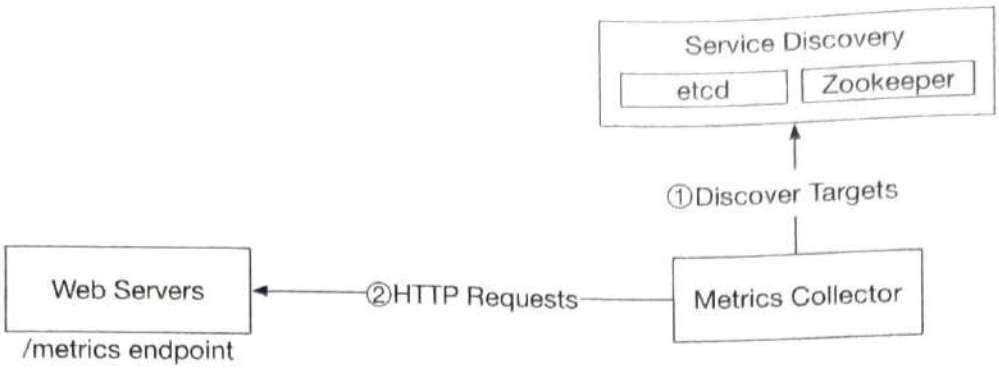


Figure 5.10: Pull model in detail

1. The metrics collector fetches configuration metadata of service endpoints from Service Discovery. Metadata include pulling interval, IP addresses, timeout and retry parameters, etc.
2. The metrics collector pulls metrics data via a pre-defined HTTP endpoint (for example, `/metrics`). To expose the endpoint, a client library usually needs to be added to the service. In Figure 5.10, the service is Web Servers.
3. Optionally, the metrics collector registers a change event notification with Service Discovery to receive an update whenever the service endpoints change. Alternatively, the metrics collector can poll for endpoint changes periodically.

At our scale, a single metrics collector will not be able to handle thousands of servers. We must use a pool of metrics collectors to handle the demand. One common problem when there are multiple collectors is that multiple instances might try to pull data from the same resource and produce duplicate data. There must exist some coordination scheme among the instances to avoid this.

One potential approach is to designate each collector to a range in a consistent hash ring, and then map every single server being monitored by its unique name in the hash ring. This ensures one metrics source server is handled by one collector only. Let's take a look at an example.

As shown in Figure 5.11, there are four collectors and six metrics source servers. Each collector is responsible for collecting metrics from a distinct set of servers. Collector 2 is responsible for collecting metrics from Server 1 and Server 5.

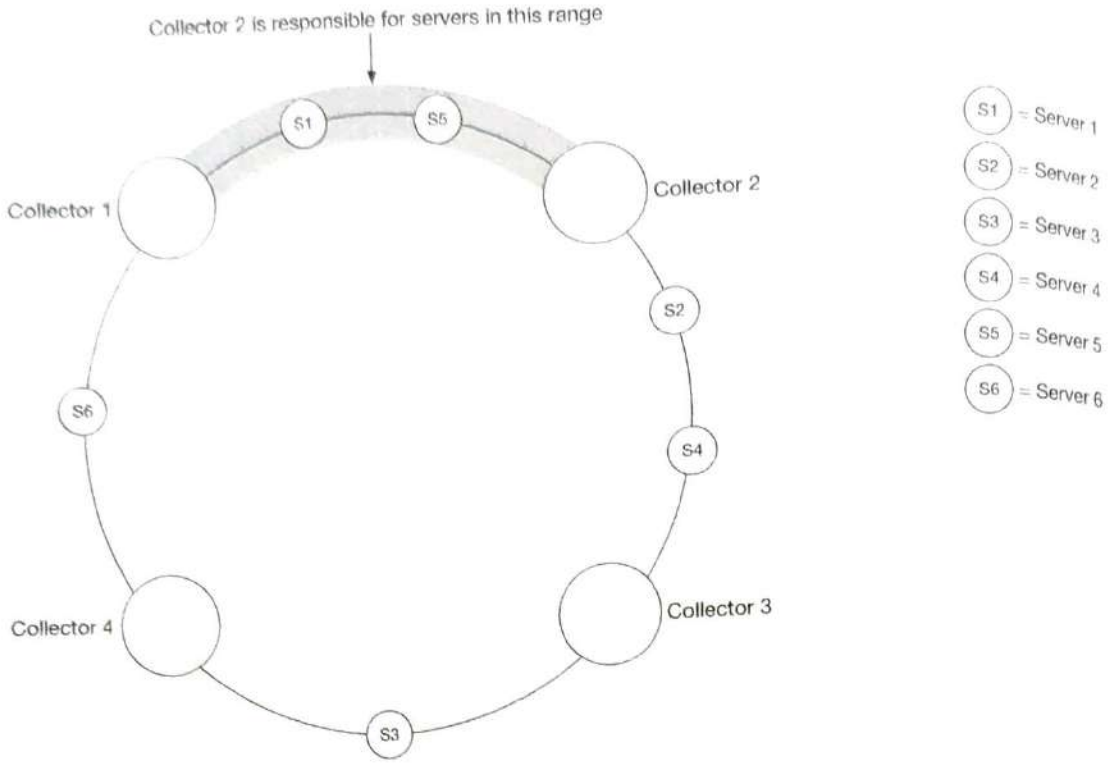


Figure 5.11: Consistent hashing

Push model

As shown in Figure 5.12, in a push model various metrics sources, such as web servers, database servers, etc., directly send metrics to the metrics collector.

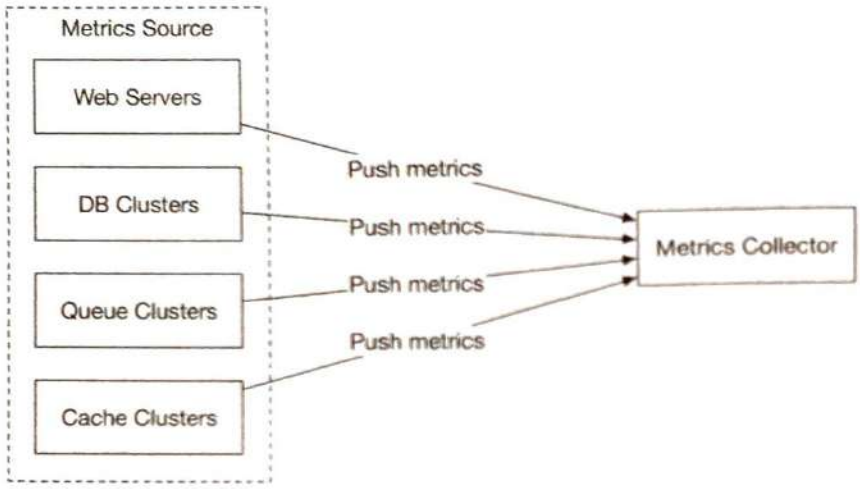


Figure 5.12: Push model

In a push model, a collection agent is commonly installed on every server being monitored. A collection agent is a piece of long-running software that collects metrics from the services running on the server and pushes those metrics periodically to the metrics collector. The collection agent may also aggregate metrics (especially a simple counter) locally, before sending them to metric collectors.

Aggregation is an effective way to reduce the volume of data sent to the metrics collector. If the push traffic is high and the metrics collector rejects the push with an error, the agent could keep a small buffer of data locally (possibly by storing them locally on disk), and resend them later. However, if the servers are in an auto-scaling group where they are rotated out frequently, then holding data locally (even temporarily) might result in data loss when the metrics collector falls behind.

To prevent the metrics collector from falling behind in a push model, the metrics collector should be in an auto-scaling cluster with a load balancer in front of it (Figure 5.13). The cluster should scale up and down based on the CPU load of the metric collector servers.

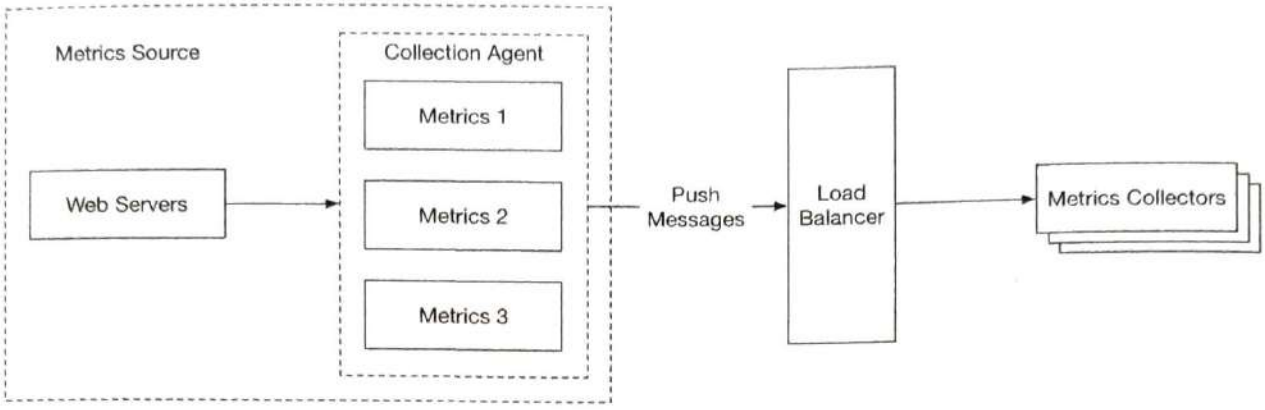


Figure 5.13: Load balancer

Pull or push?

So, which one is the better choice for us? Just like many things in life, there is no clear answer. Both sides have widely adopted real-world use cases.

- Examples of pull architectures include Prometheus.
- Examples of push architectures include Amazon CloudWatch [18] and Graphite [19].

Knowing the advantages and disadvantages of each approach is more important than picking a winner during an interview. Table 5.3 compares the pros and cons of push and pull architectures [20] [21] [22] [23].

	Pull	Push
Easy debugging	The /metrics endpoint on application servers used for pulling metrics can be used to view metrics at any time. You can even do this on your laptop. **Pull wins**.	
Health check	If an application server doesn't respond to the pull, you can quickly figure out if an application server is down. **Pull wins**.	If the metrics collector doesn't receive metrics, the problem might be caused by network issues.
Short-lived jobs		Some of the batch jobs might be short-lived and don't last long enough to be pulled. **Push wins**. This can be fixed by introducing push gateways for the pull model [24].
Firewall or complicated network setups	Having servers pulling metrics requires all metric endpoints to be reachable. This is potentially problematic in multiple data center setups. It might require a more elaborate network infrastructure.	If the metrics collector is set up with a load balancer and an auto-scaling group, it is possible to receive data from anywhere. **Push wins**.

Performance	Pull methods typically use TCP.	Push methods typically use UDP. This means the push method provides lower-latency transports of metrics. The counterargument here is that the effort of establishing a TCP connection is small compared to sending the metrics payload.
Data authenticity	Application servers to collect metrics from are defined in config files in advance. Metrics gathered from those servers are guaranteed to be authentic.	Any kind of client can push metrics to the metrics collector. This can be fixed by whitelisting servers from which to accept metrics, or by requiring authentication.

Table 5.3: Pull vs push

As mentioned above, pull vs push is a routine debate topic and there is no clear answer. A large organization probably needs to support both, especially with the popularity of serverless [25] these days. There might not be a way to install an agent from which to push data in the first place.

Scale the metrics transmission pipeline

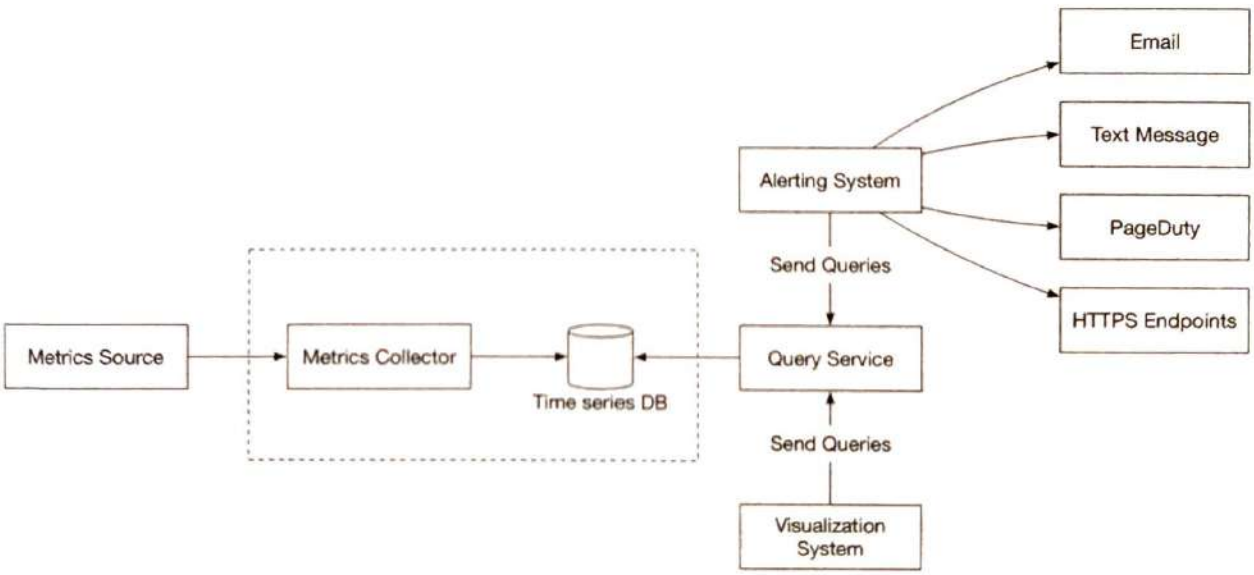


Figure 5.14: Metrics transmission pipeline

Let's zoom in on the metrics collector and time-series databases. Whether you use the push or pull model, the metrics collector is a cluster of servers, and the cluster receives enormous amounts of data. For either push or pull, the metrics collector cluster is set up for auto-scaling, to ensure that there are an adequate number of collector instances to handle the demand.

However, there is a risk of data loss if the time-series database is unavailable. To mitigate this problem, we introduce a queueing component as shown in Figure 5.15.

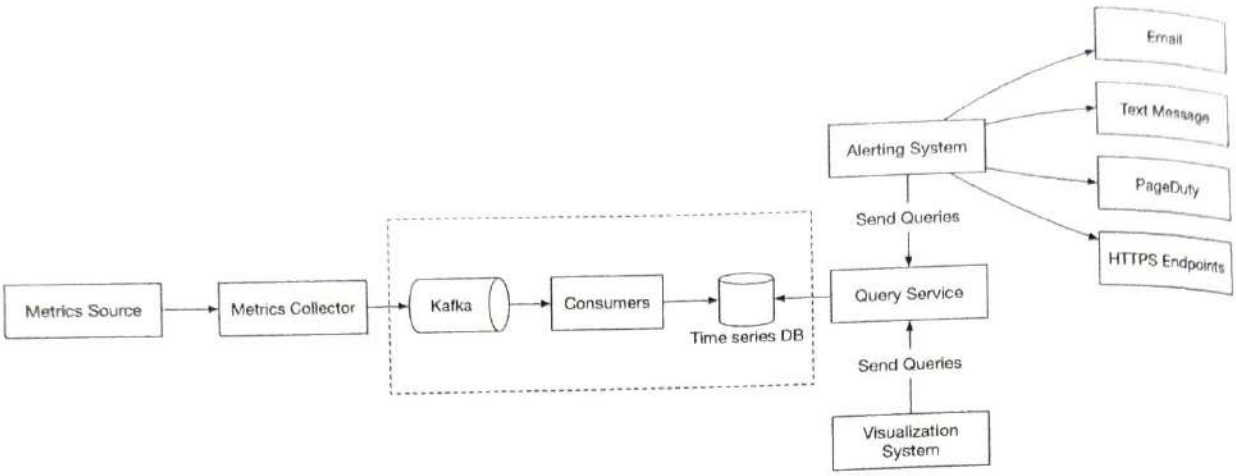


Figure 5.15: Add queues

In this design, the metrics collector sends metrics data to queuing systems like Kafka. Then consumers or streaming processing services such as Apache Storm, Flink, and Spark, process and push data to the time-series database. This approach has several advantages:

- Kafka is used as a highly reliable and scalable distributed messaging platform.
- It decouples the data collection and data processing services from each other.
- It can easily prevent data loss when the database is unavailable, by retaining the data in Kafka.

Scale through Kafka

There are a couple of ways that we can leverage Kafka's built-in partition mechanism to scale our system.

- Configure the number of partitions based on throughput requirements.
- Partition metrics data by metric names, so consumers can aggregate data by metrics names.
- Further partition metrics data with tags/labels.
- Categorize and prioritize metrics so that important metrics can be processed first.

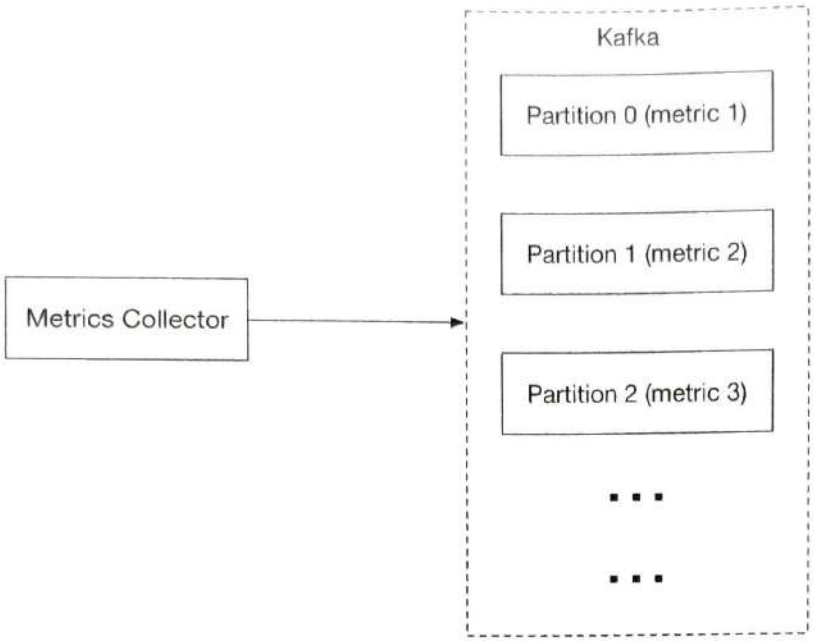


Figure 5.16: Kafka partition

Alternative to Kafka

Maintaining a production-scale Kafka system is no small undertaking. You might get pushback from the interviewer about this. There are large-scale monitoring ingestion systems in use without using an intermediate queue. Facebook's Gorilla [26] in-memory time-series database is a prime example; it is designed to remain highly available for writes, even when there is a partial network failure. It could be argued that such a design is as reliable as having an intermediate queue like Kafka.

Where aggregations can happen

Metrics can be aggregated in different places; in the collection agent (on the client-side), the ingestion pipeline (before writing to storage), and the query side (after writing to storage). Let's take a closer look at each of them.

Collection agent. The collection agent installed on the client-side only supports simple aggregation logic. For example, aggregate a counter every minute before it is sent to the metrics collector.

Ingestion pipeline. To aggregate data before writing to the storage, we usually need stream processing engines such as Flink. The write volume will be significantly reduced since only the calculated result is written to the database. However, handling late-arriving events could be a challenge and another downside is that we lose data precision and some flexibility because we no longer store the raw data.

Query side. Raw data can be aggregated over a given time period at query time. There is no data loss with this approach, but the query speed might be slower because the query result is computed at query time and is run against the whole dataset.

Query service

The query service comprises a cluster of query servers, which access the time-series databases and handle requests from the visualization or alerting systems. Having a dedicated set of query servers decouples time-series databases from the clients (visualization and alerting systems). And this gives us the flexibility to change the time-series database or the visualization and alerting systems, whenever needed.

Cache layer

To reduce the load of the time-series database and make query service more performant, cache servers are added to store query results, as shown in Figure 5.17.

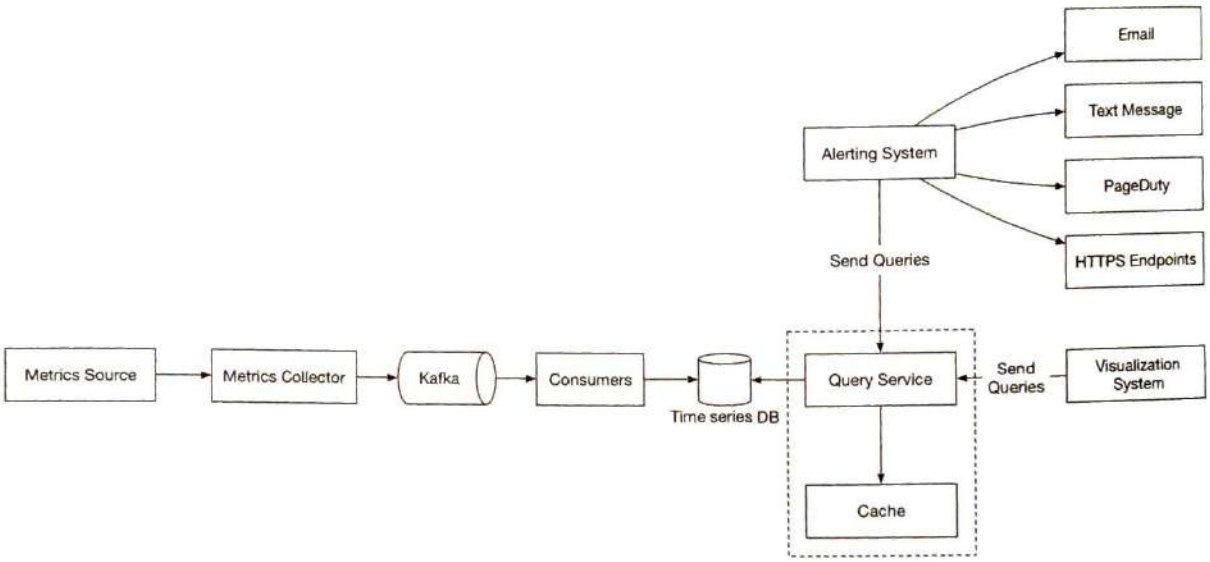


Figure 5.17: Cache layer

The case against query service

There might not be a pressing need to introduce our own abstraction (a query service) because most industrial-scale visual and alerting systems have powerful plugins to interface with well-known time-series databases on the market. And with a well-chosen time-series database, there is no need to add our own caching, either.

Time-series database query language

Most popular metrics monitoring systems like Prometheus and InfluxDB don't use SQL and have their own query languages. One major reason for this is that it is hard to build SQL queries to query time-series data. For example, as mentioned here [27], computing an exponential moving average might look like this in SQL:

```
select id,
       temp,
       avg(temp) over (partition by group_nr order by
time_read)
       as rolling_avg
from (
    select id,
         temp,
         time_read,
         interval_group,
         id - row_number() over (partition by interval_group
order
         by time_read) as group_nr
    from (
         select id,
         time_read,
         "epoch"::timestamp + "900 seconds"::interval * (
extract(epoch from time_read)::int4 / 900) as interval_group,
         temp
         from readings
    ) t1
) t2
order by time_read;
```

While in Flux, a language that's optimized for time-series analysis (used in InfluxDB), it looks like this. As you can see, it's much easier to understand.

```
from(db:"telegraf")
    |> range(start:-1h)
    |> filter(fn: (r) => r._measurement == "foo")
    |> exponentialMovingAverage(size:-10s)
```

Storage layer

Now let's dive into the storage layer.

Choose a time-series database carefully

According to a research paper published by Facebook [26], at least 85% of all queries to the operational data store were for data collected in the past 26 hours. If we use a time-series database that harnesses this property, it could have a significant impact on overall system performance. If you are interested in the design of the storage engine, please refer to the design document of the InfluxDB storage engine [28].

Space optimization

As explained in high-level requirements, the amount of metric data to store is enormous. Here are a few strategies for tackling this.

Data encoding and compression

Data encoding and compression can significantly reduce the size of data. Those features are usually built into a good time-series database. Here is a simple example.

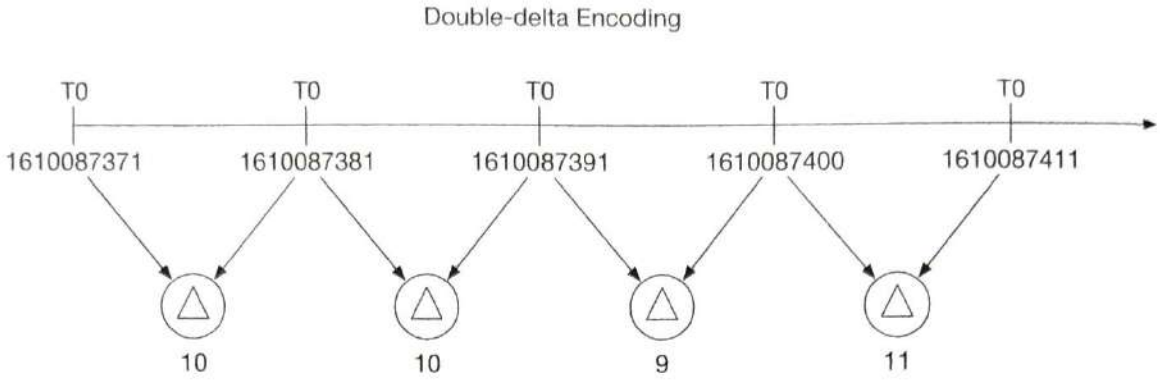


Figure 5.18: Data encoding

As you can see in the image above, 1610087371 and 1610087381 differ by only 10 seconds, which takes only 4 bits to represent, instead of the full timestamp of 32 bits. So, rather than storing absolute values, the delta of the values can be stored along with one base value like: 1610087371, 10, 10, 9, 11.

Downsampling

Downsampling is the process of converting high-resolution data to low-resolution to reduce overall disk usage. Since our data retention is 1 year, we can downsample old data. For example, we can let engineers and data scientists define rules for different metrics. Here is an example:

- Retention: 7 days, no sampling
- Retention: 30 days, downsample to 1 minute resolution
- Retention: 1 year, downsample to 1 hour resolution

Let's take a look at another concrete example. It aggregates 10-second resolution data to 30-second resolution data.

metric	timestamp	hostname	metric_value
cpu	2021-10-24T19:00:00Z	host-a	10
cpu	2021-10-24T19:00:10Z	host-a	16
cpu	2021-10-24T19:00:20Z	host-a	20
cpu	2021-10-24T19:00:30Z	host-a	30
cpu	2021-10-24T19:00:40Z	host-a	20
cpu	2021-10-24T19:00:50Z	host-a	30

Table 5.4: 10-second resolution data

Rollup from 10 second resolution data to 30 second resolution data.

metric	timestamp	hostname	Metric_value (avg)
cpu	2021-10-24T19:00:00Z	host-a	19
cpu	2021-10-24T19:00:30Z	host-a	25

Table 5.5: 30-second resolution data

Cold storage

Cold storage is the storage of inactive data that is rarely used. The financial cost for cold storage is much lower.

In a nutshell, we should probably use third-party visualization and alerting systems, instead of building our own.

Alerting system

For the purpose of the interview, let's look at the alerting system, shown in Figure 5.19 below.

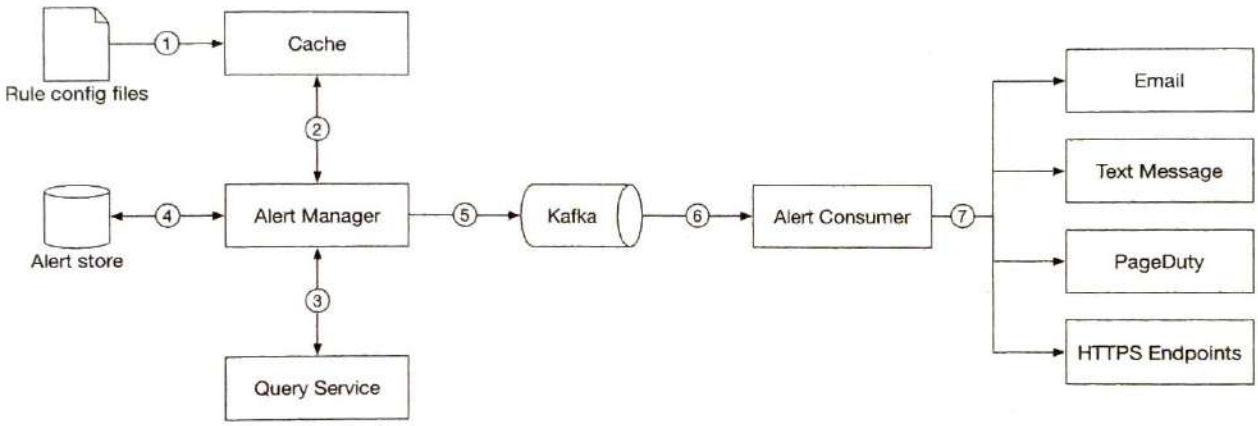


Figure 5.19: Alerting system

The alert flow works as follows:

1. Load config files to cache servers. Rules are defined as config files on the disk. YAML [29] is a commonly used format to define rules. Here is an example of alert rules:

```
- name: instance_down
rules:

# Alert for any instance that is unreachable for >5
minutes.
- alert: instance_down
    expr: up == 0
    for: 5m
    labels:
    severity: page
```

2. The alert manager fetches alert configs from the cache.

3. Based on config rules, the alert manager calls the query service at a predefined interval. If the value violates the threshold, an alert event is created. The alert manager is responsible for the following:
    - Filter, merge, and dedupe alerts. Here is an example of merging alerts that are triggered within one instance within a short amount of time (instance 1) (Figure 5.20).

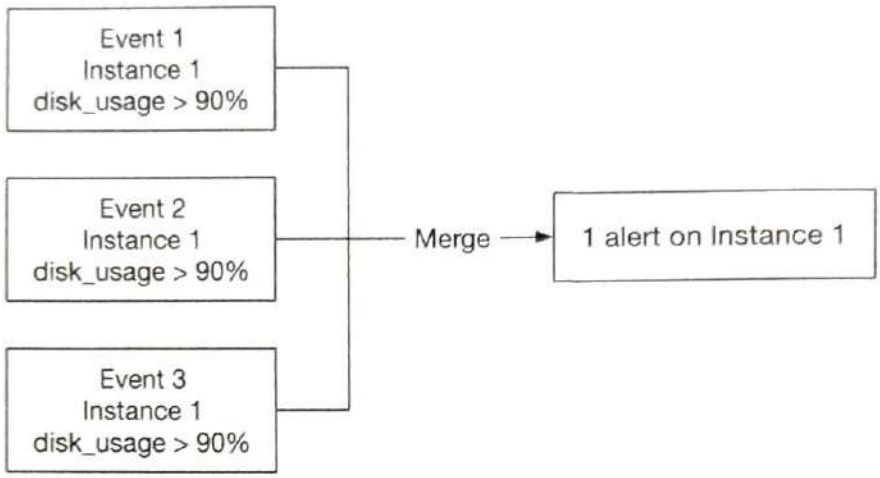


Figure 5.20: Merge alerts

    - Access control. To avoid human error and keep the system secure, it is essential to restrict access to certain alert management operations to authorized individuals only.
    - Retry. The alert manager checks alert states and ensures a notification is sent at least once.
4. The alert store is a key-value database, such as Cassandra, that keeps the state (inactive, pending, firing, resolved) of all alerts. It ensures a notification is sent at least once.
5. Eligible alerts are inserted into Kafka.
6. Alert consumers pull alert events from Kafka.
7. Alert consumers process alert events from Kafka and send notifications over to different channels such as email, text message, PagerDuty, or HTTP endpoints.

Alerting system - build vs buy

There are many industrial-scale alerting systems available off-the-shelf, and most provide tight integration with the popular time-series databases. Many of these alerting systems integrate well with existing notification channels, such as email and PagerDuty. In the real world, it is a tough call to justify building your own alerting system. In interview settings, especially for a senior position, be ready to justify your decision.

Visualization system

Visualization is built on top of the data layer. Metrics can be shown on the metrics dashboard over various time scales and alerts can be shown on the alerts dashboard. Figure 5.21 shows a dashboard that displays some of the metrics like the current server requests, memory/CPU utilization, page load time, traffic, and login information [30].

Figure 5.21: Grafana UI

A high-quality visualization system is hard to build. The argument for using an off-the-shelf system is very strong. For example, Grafana can be a very good system for this purpose. It integrates well with many popular time-series databases which you can buy.

Step 4 - Wrap Up

In this chapter, we presented the design for a metrics monitoring and alerting system. At a high level, we talked about data collection, time-series database, alerts, and visualization. Then we went in-depth into some of the most important techniques/components:

- Pull vs pull model for collecting metrics data.
- Utilize Kafka to scale the system.
- Choose the right time-series database.

- Use downsampling to reduce data size.
- Build vs buy options for alerting and visualization systems.

We went through a few iterations to refine the design, and our final design looks like this:

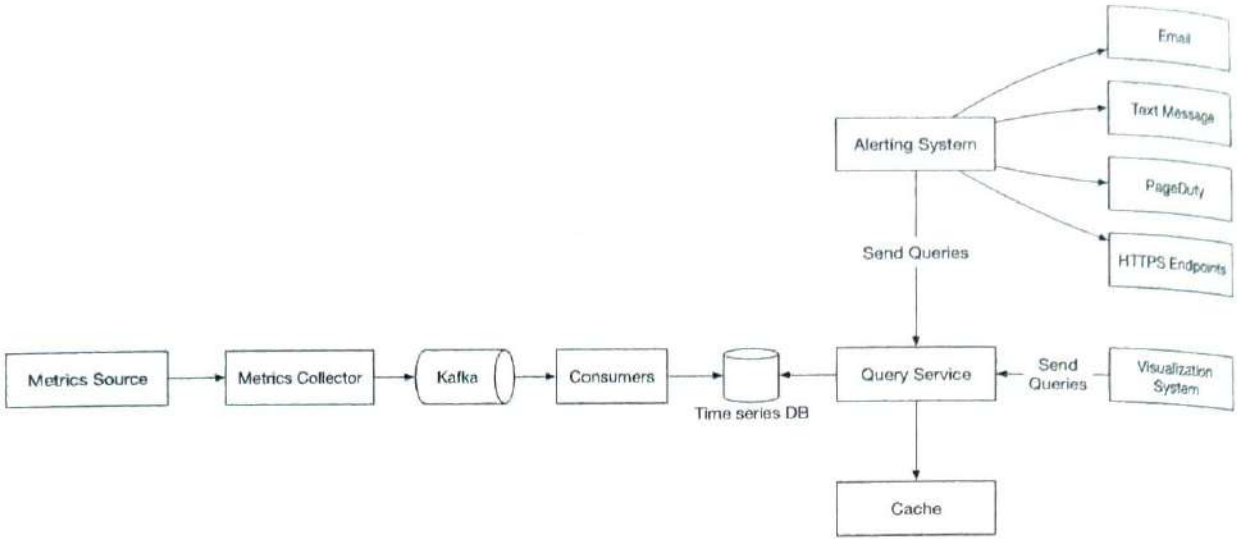


Figure 5.22: Final design

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

Chapter Summary

Reference Material

[1] Datadog. https://www.datadoghq.com/.

[2] Splunk. https://www.splunk.com/.

[3] PagerDuty. https://www.pagerduty.com/.

[4] Elastic stack. https://www.elastic.co/elastic-stack.

[5] Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. https://research.google/pubs/pub36356/.

[6] Distributed Systems Tracing with Zipkin. https://blog.twitter.com/engineering/en_us/a/2012/distributed-systems-tracing-with-zipkin.html.

[7] Prometheus. https://prometheus.io/docs/introduction/overview/.

[8] OpenTSDB - A Distributed, Scalable Monitoring System. http://opentsdb.net/.

[9] Data model. :https://prometheus.io/docs/concepts/data_model/.

[10] MySQL. https://www.mysql.com/.

[11] Schema design for time-series data | Cloud Bigtable Documentation. https://cloud.google.com/bigtable/docs/schema-design-time-series.

[12] MetricsDB. TimeSeriesDatabaseforstoringmetricsatTwitter:https://blog.twitter.com/engineering/en_us/topics/infrastructure/2019/metricsdb.html.

[13] Amazon Timestream. https://aws.amazon.com/timestream/.

[14] DB-Engines Ranking of time-series DBMS. https://db-engines.com/en/ranking/time+series+dbms.

[15] InfluxDB. https://www.influxdata.com/.

[16] etcd. https://etcd.io/.

[17] Service Discovery with ZooKeeper. https://cloud.spring.io/spring-cloud-zookeeper/1.2.x/multi/multi_spring-cloud-zookeeper-discovery.html.

[18] Amazon CloudWatch. https://aws.amazon.com/cloudwatch/.

[19] Graphite. https://graphiteapp.org/.

[20] Push vs Pull. http://bit.ly/3aJEPxE.

[21] Pull doesn't scale - or does it? https://prometheus.io/blog/2016/07/23/pull-does-not-scale-or-does-it/.

[22] Monitoring Architecture. https://developer.lightbend.com/guides/monitoring-at-scale/monitoring-architecture/architecture.html.

[23] Push vs Pull in Monitoring Systems. https://giedrius.blog/2019/05/11/push-vs-pull-in-monitoring-systems/.

[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.

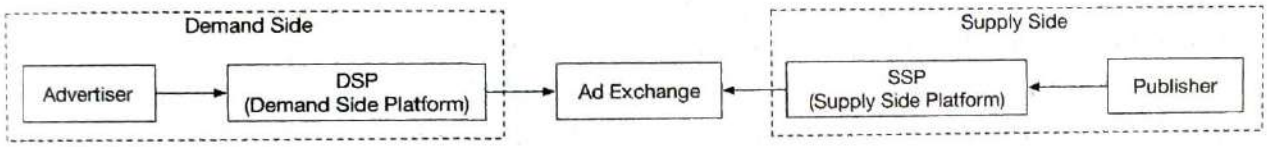


Figure 6.1: RTB process

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).

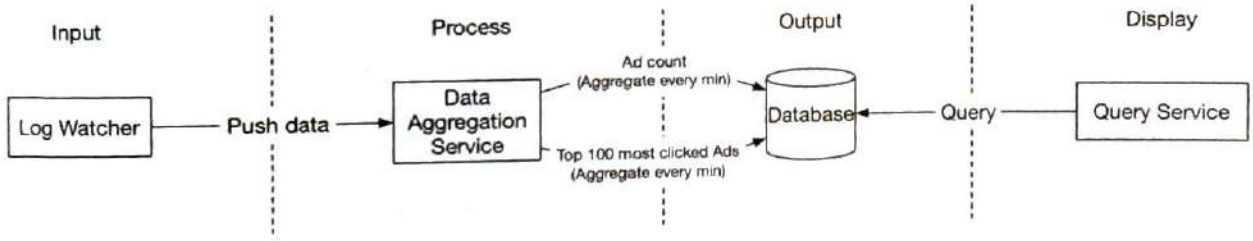


Figure 6.2: Aggregation workflow

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.

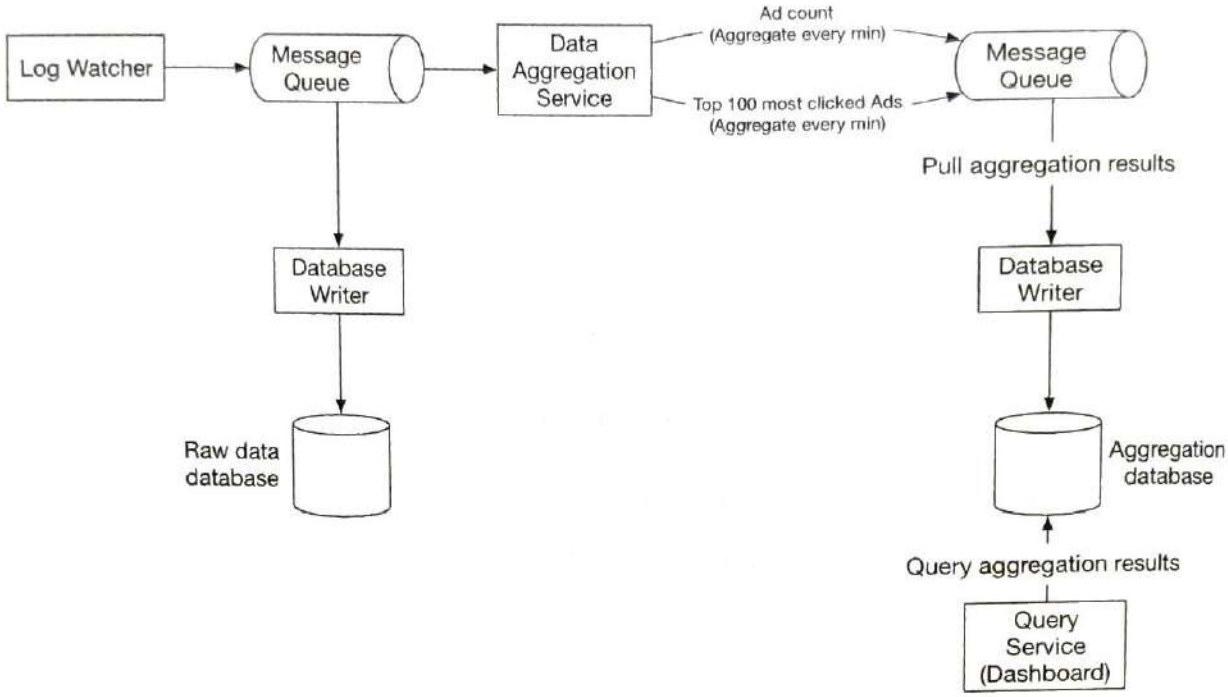


Figure 6.3: High-level design

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].

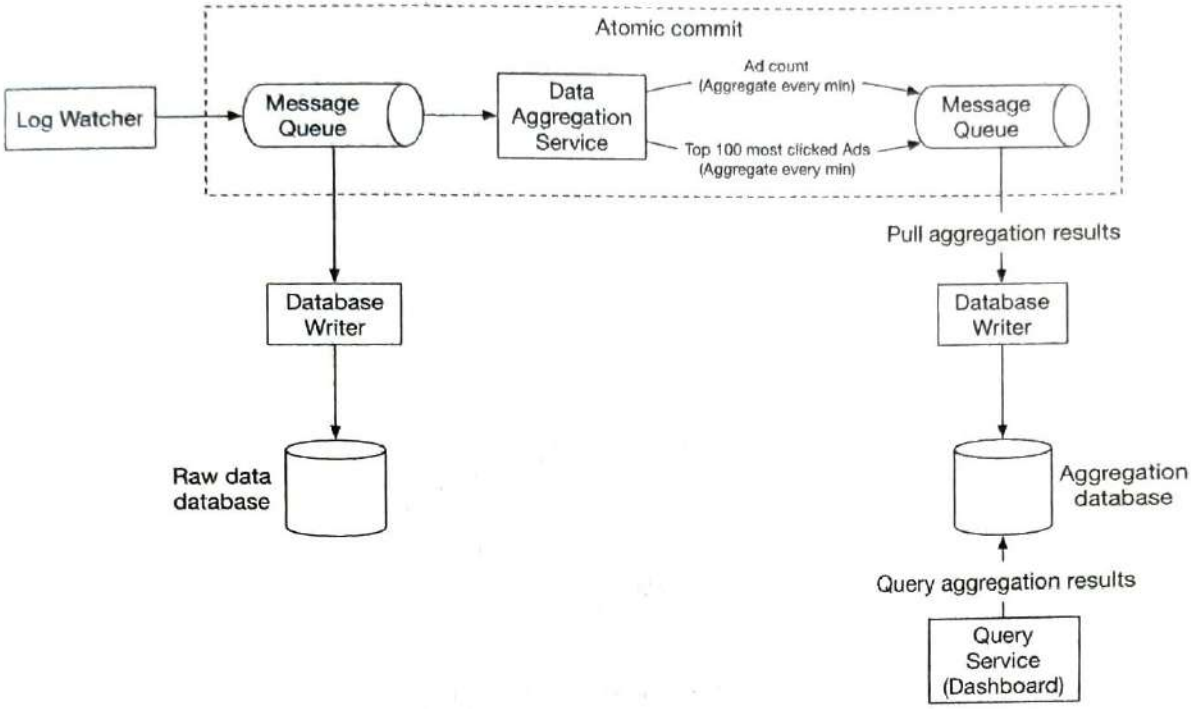


Figure 6.4: End-to-end exactly once

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.

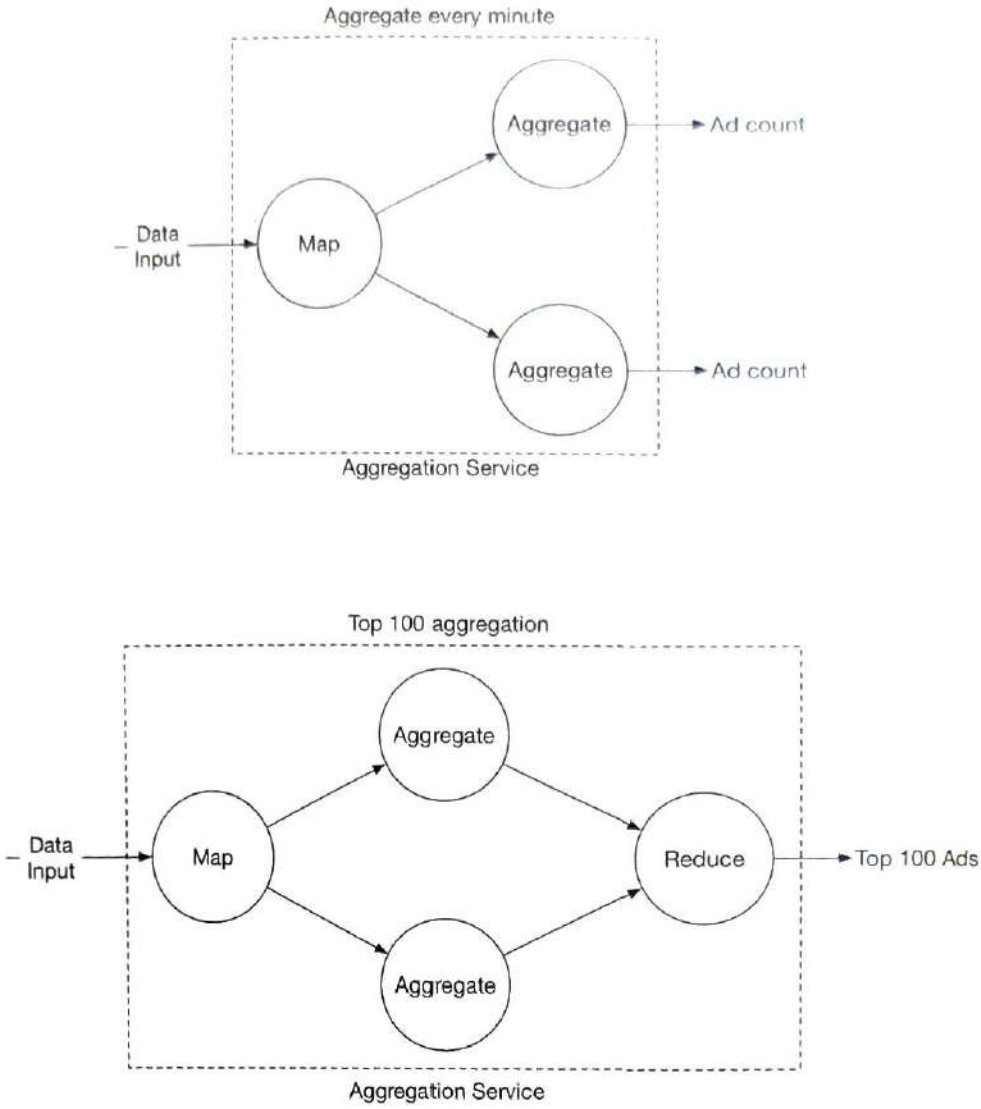


Figure 6.5: Aggregation service

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.

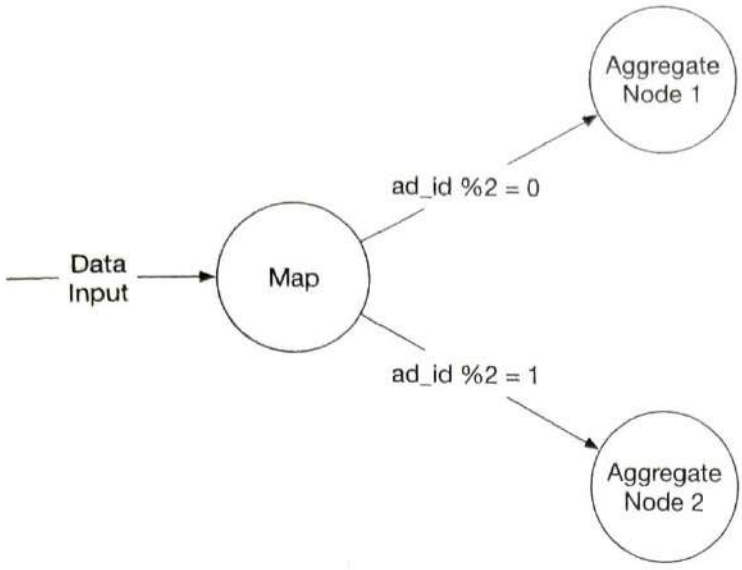


Figure 6.6: Map operation

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.

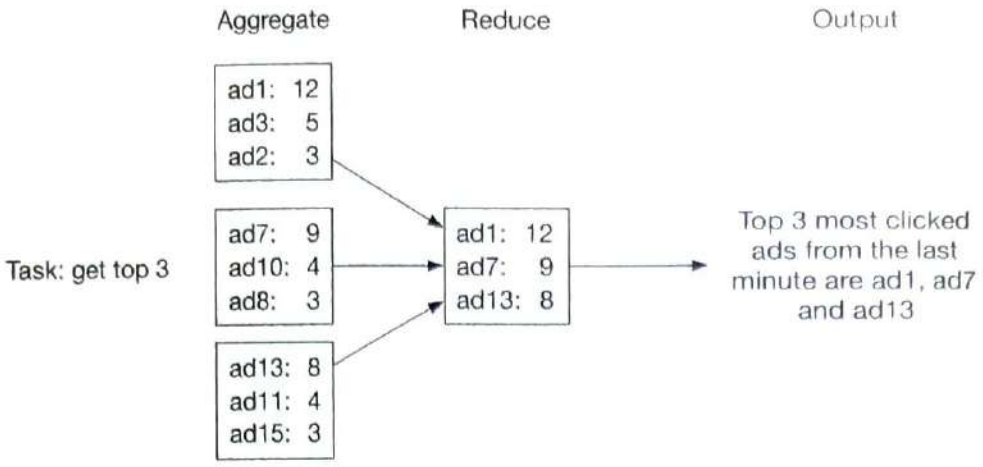


Figure 6.7: Reduce node

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.

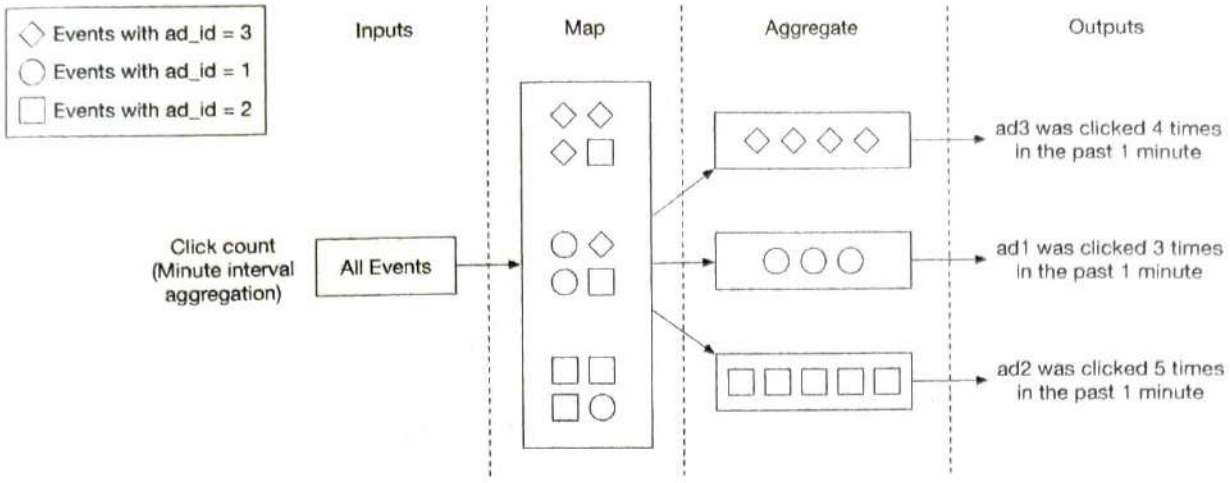


Figure 6.8: Aggregate the number of clicks

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.

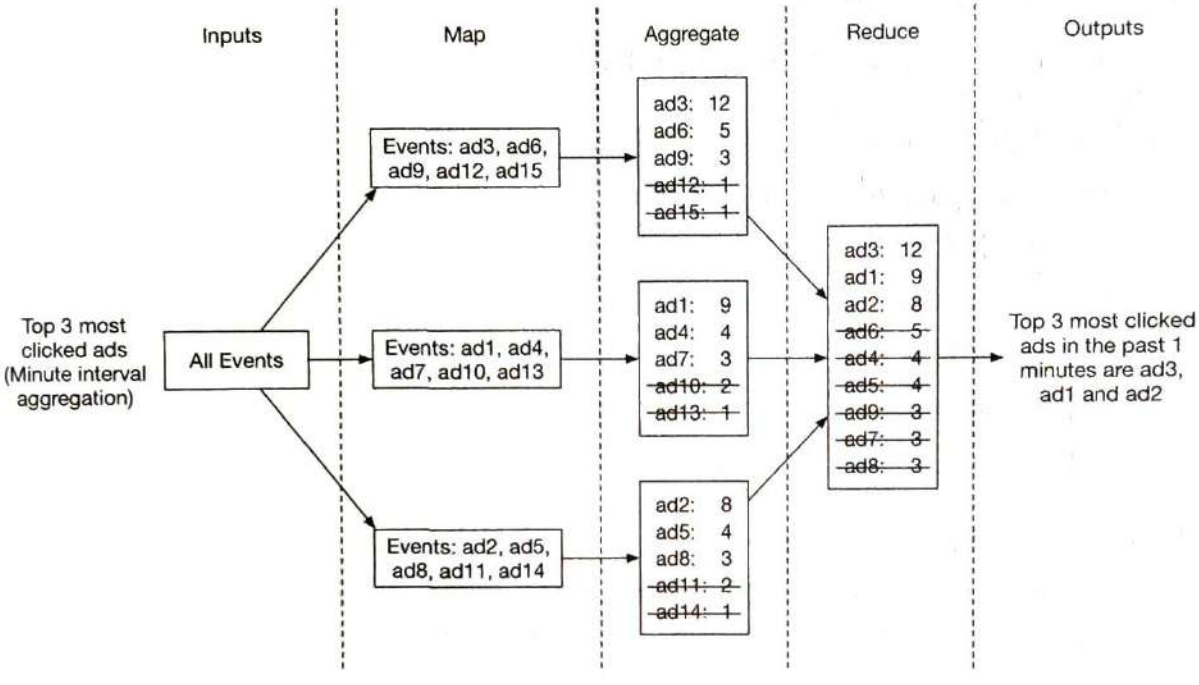


Figure 6.9: Return top N most clicked ads

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.

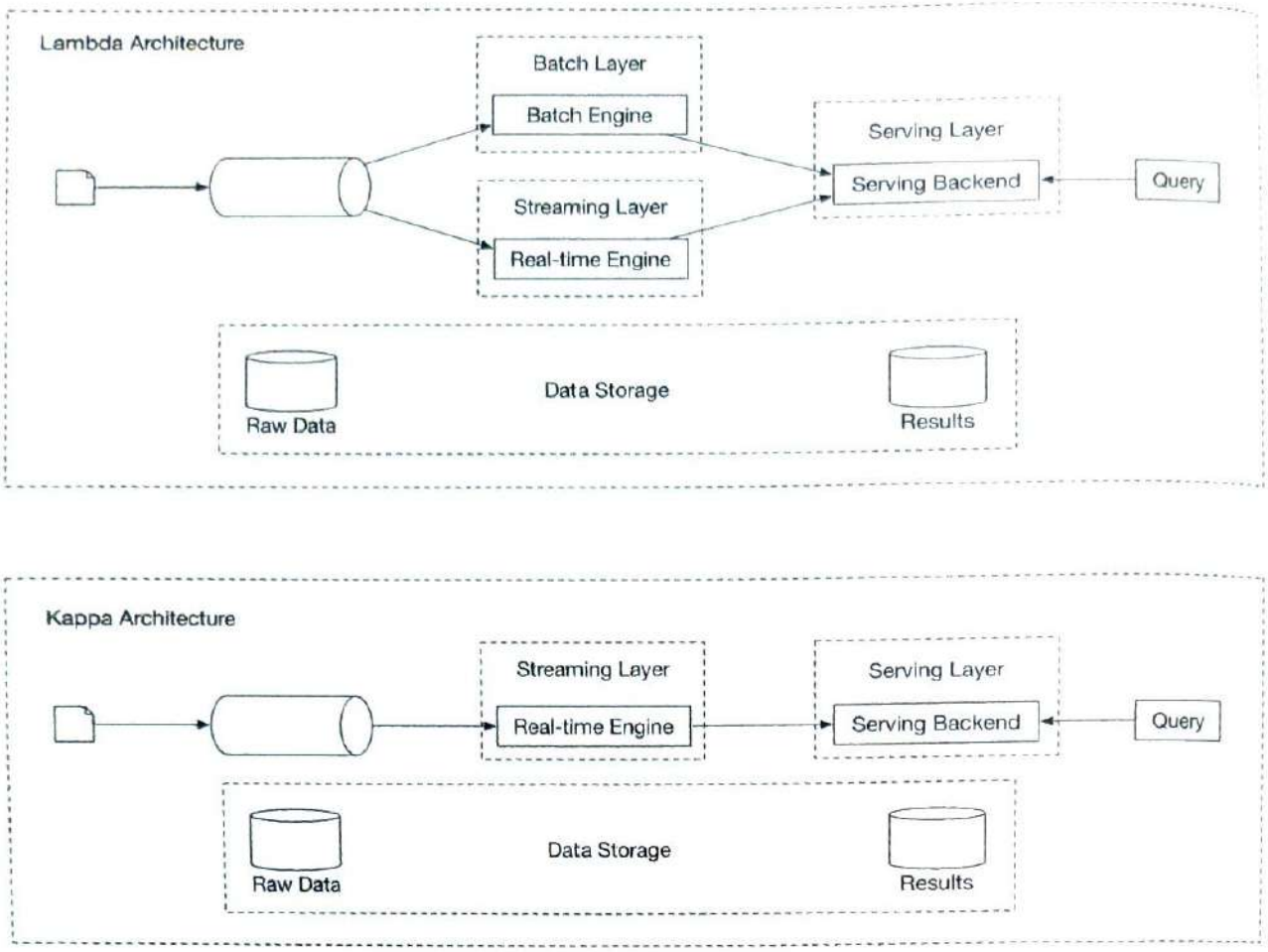


Figure 6.10: Lambda and Kappa architectures

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.

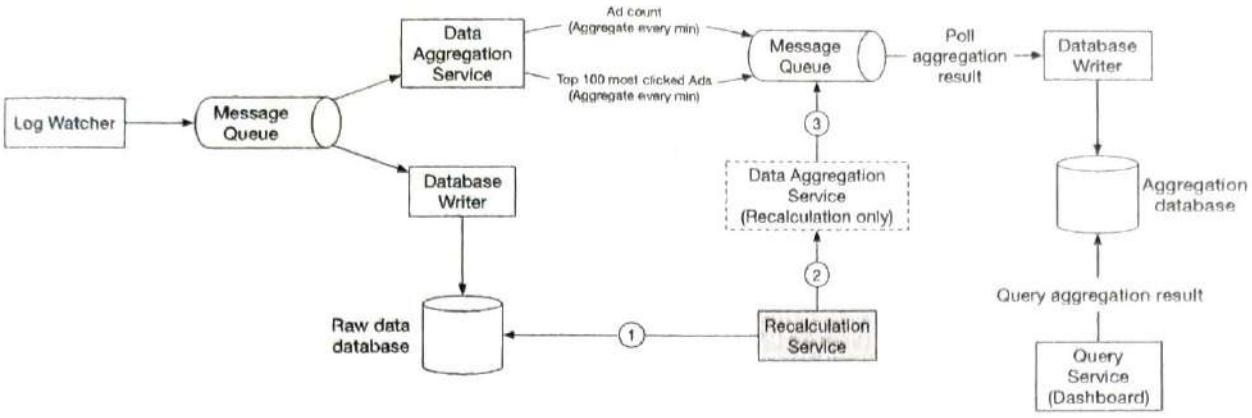


Figure 6.11: Recalculation service

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).

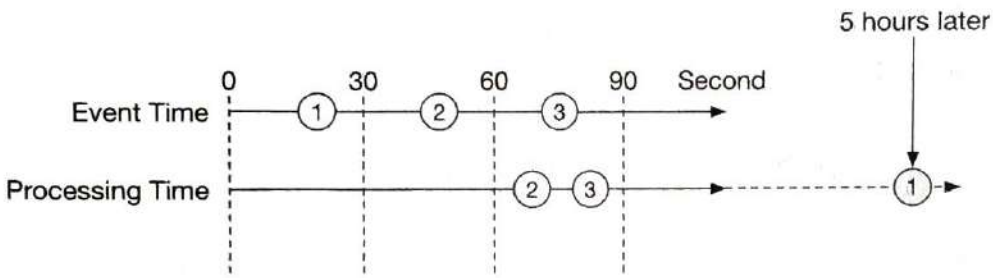


Figure 6.12: Late events

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.

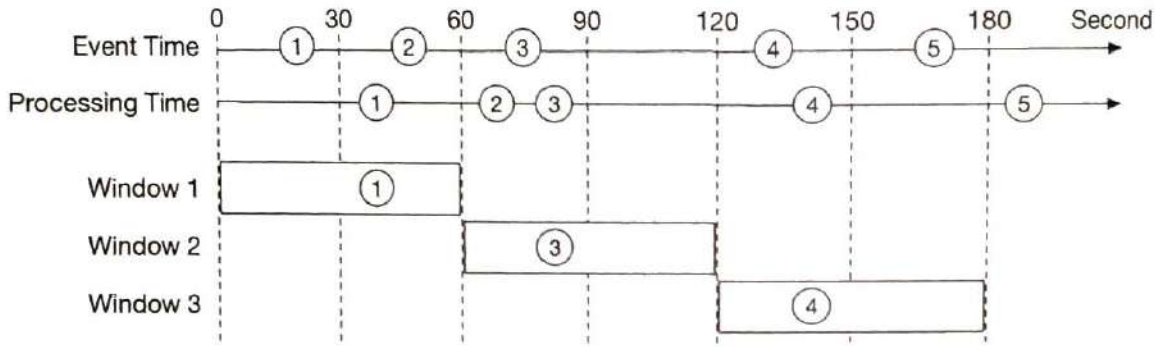


Figure 6.13: Miss events in an aggregation window

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.

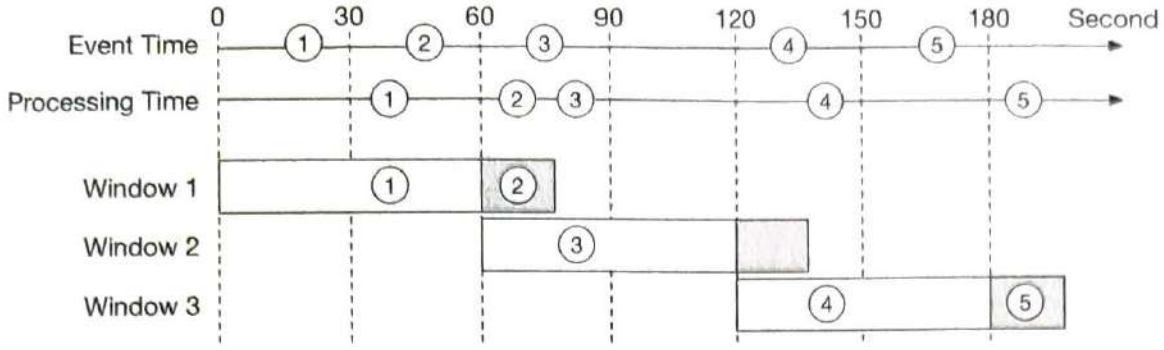


Figure 6.14: Watermark

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).

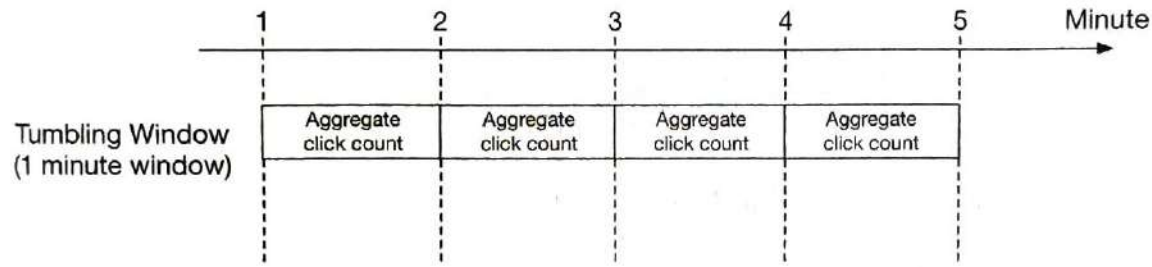


Figure 6.15: Tumbling window

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.

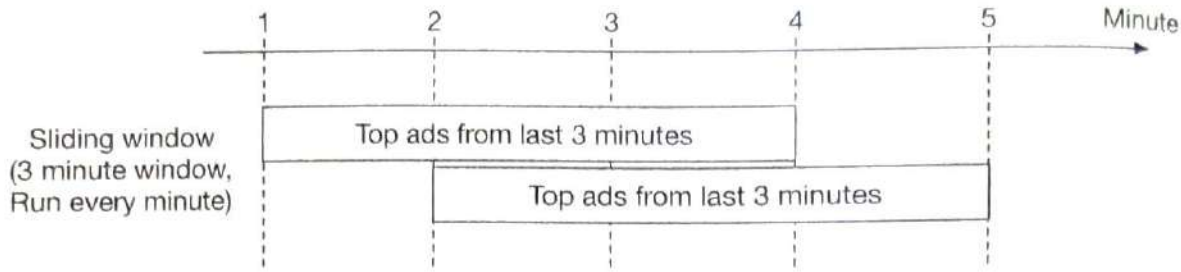


Figure 6.16: Sliding window

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.

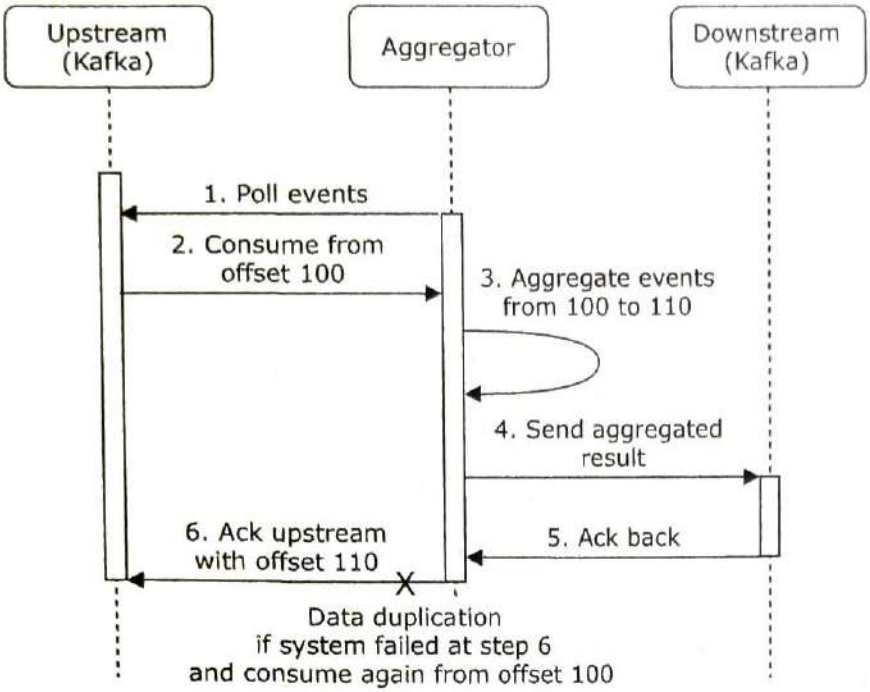


Figure 6.17: Duplicate data

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.

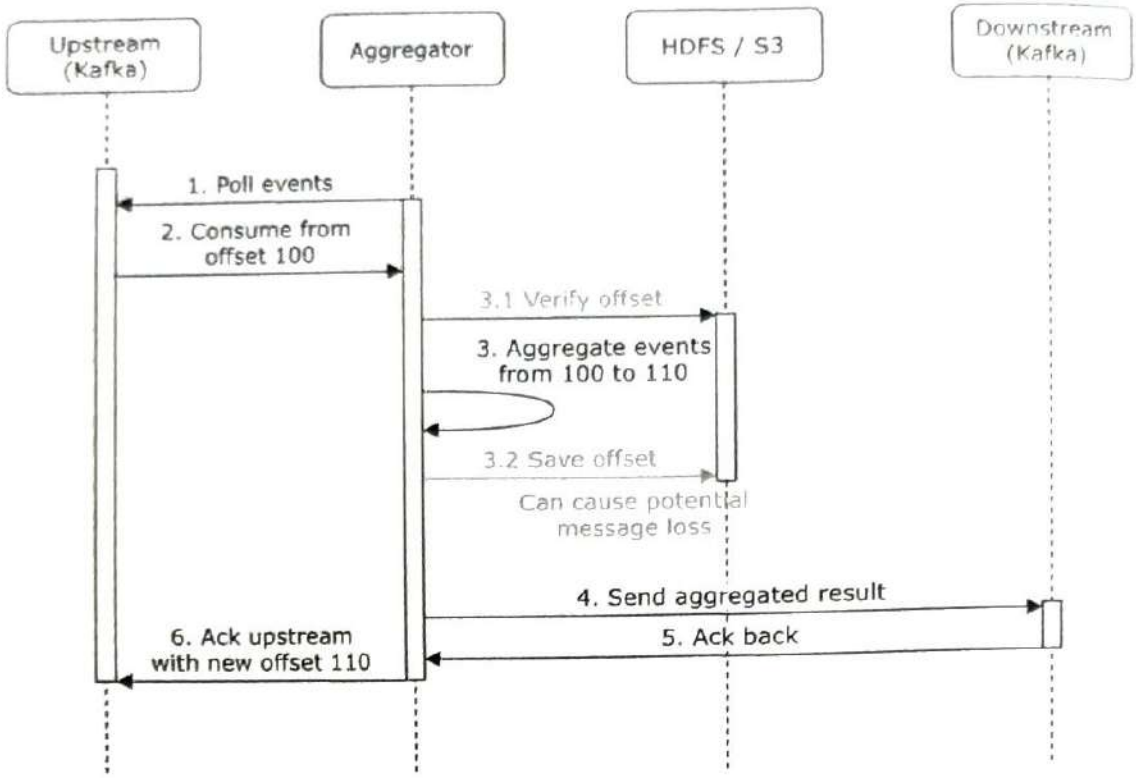


Figure 6.18: Record the offset

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.

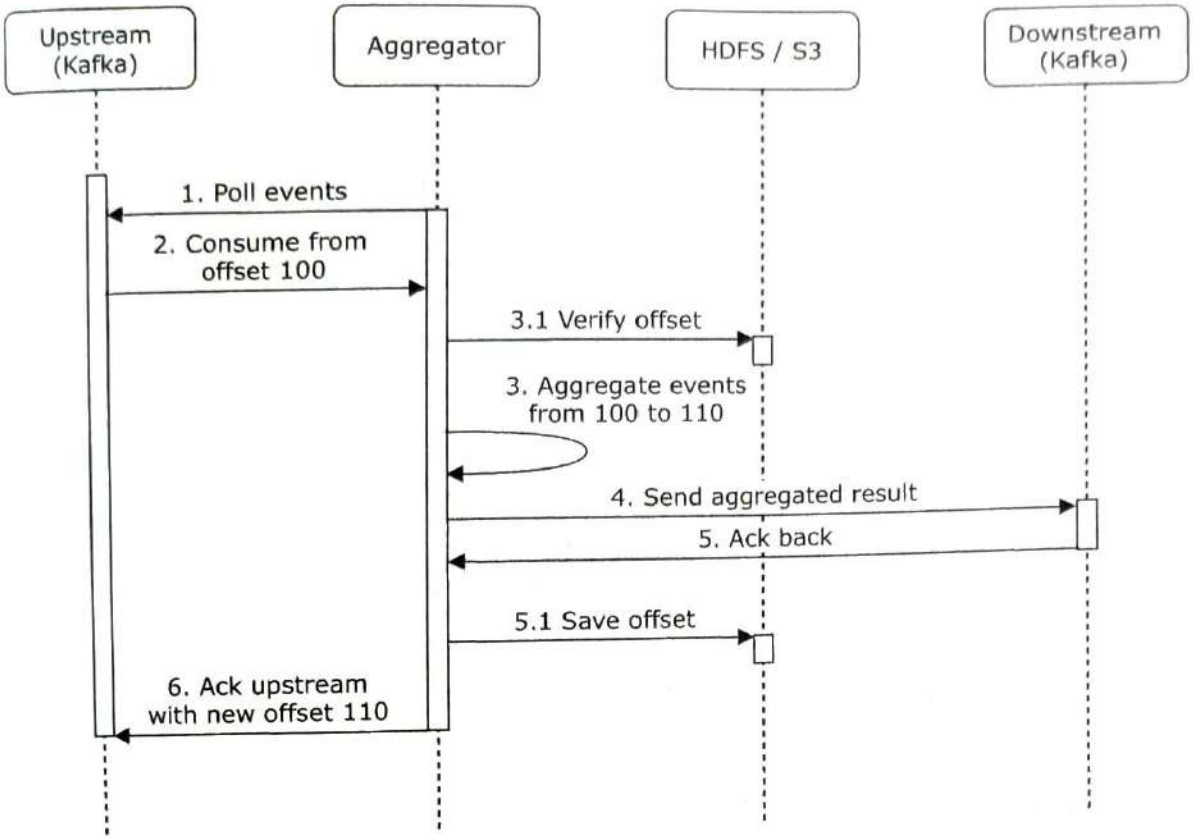


Figure 6.19: Save offset after receiving ack

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.

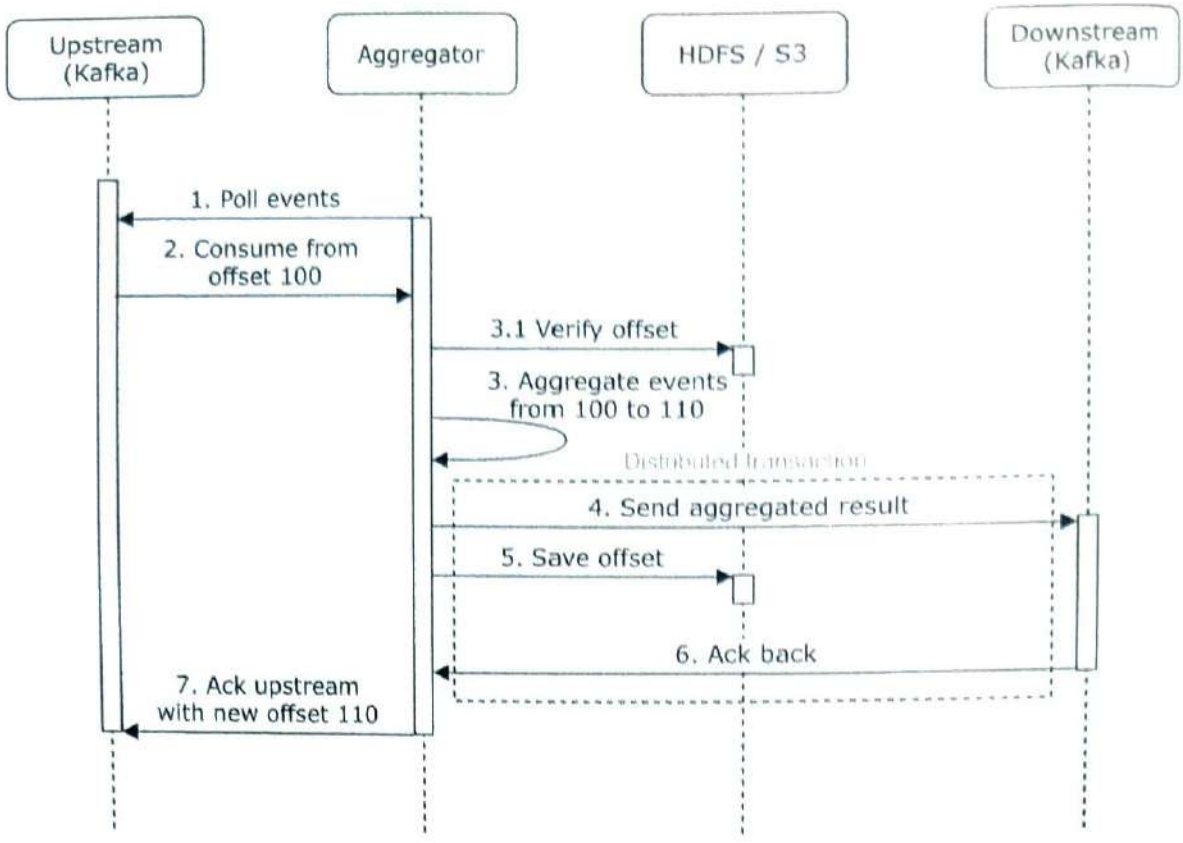


Figure 6.20: Distributed transaction

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.

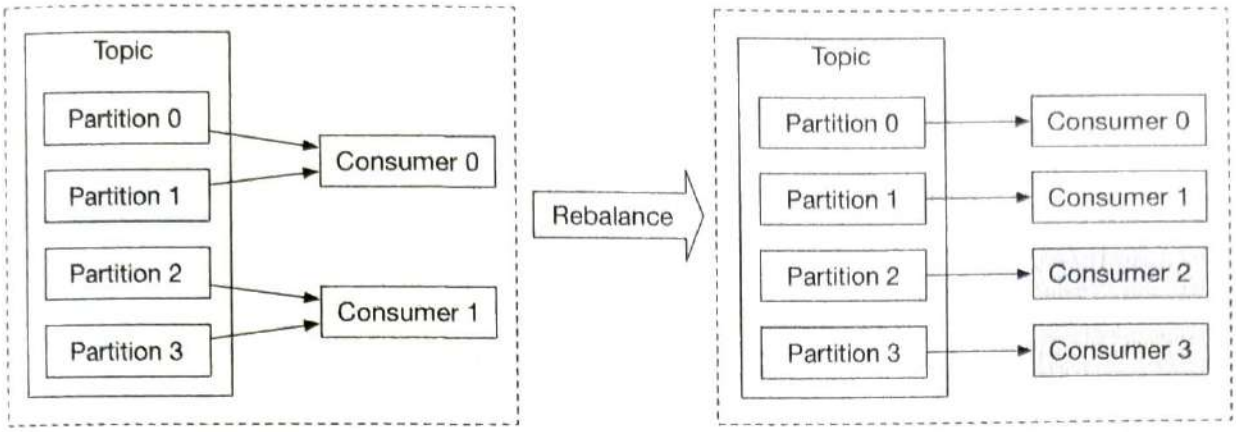


Figure 6.21: Add consumers

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.

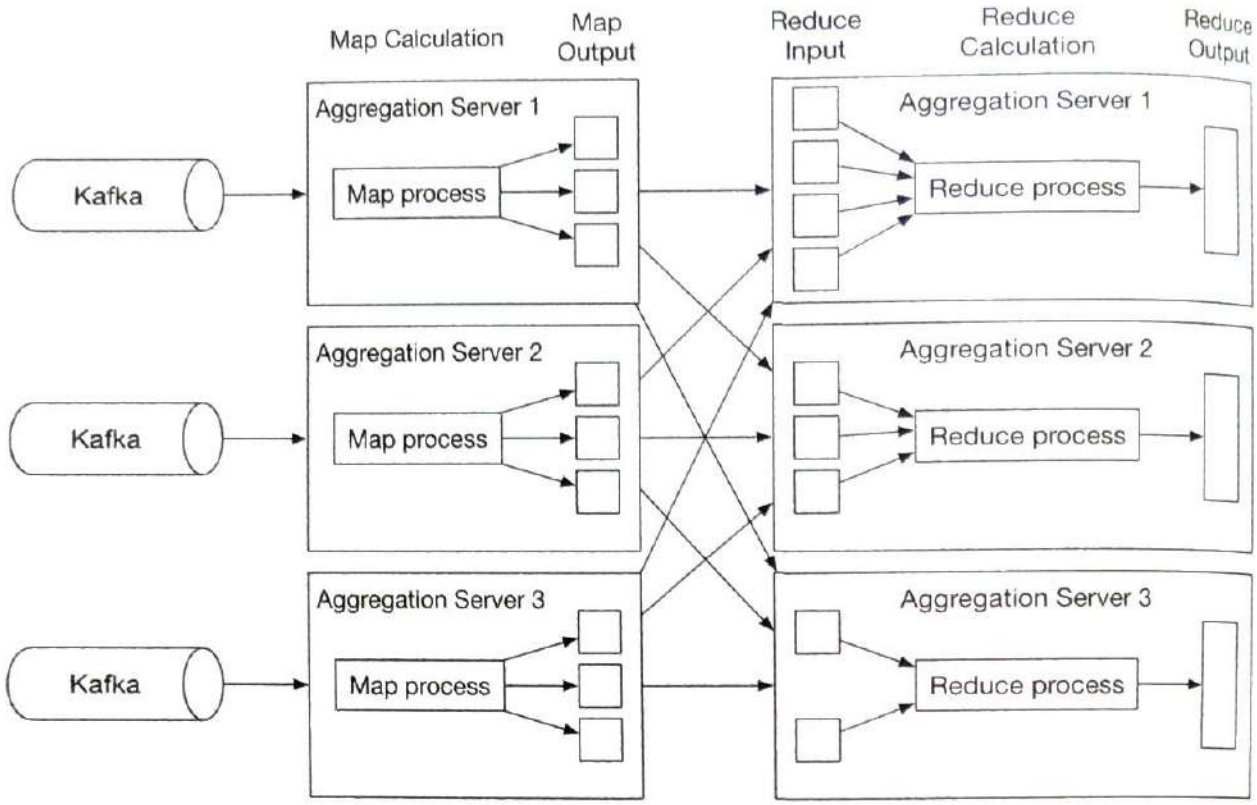


Figure 6.22: Aggregation service

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.

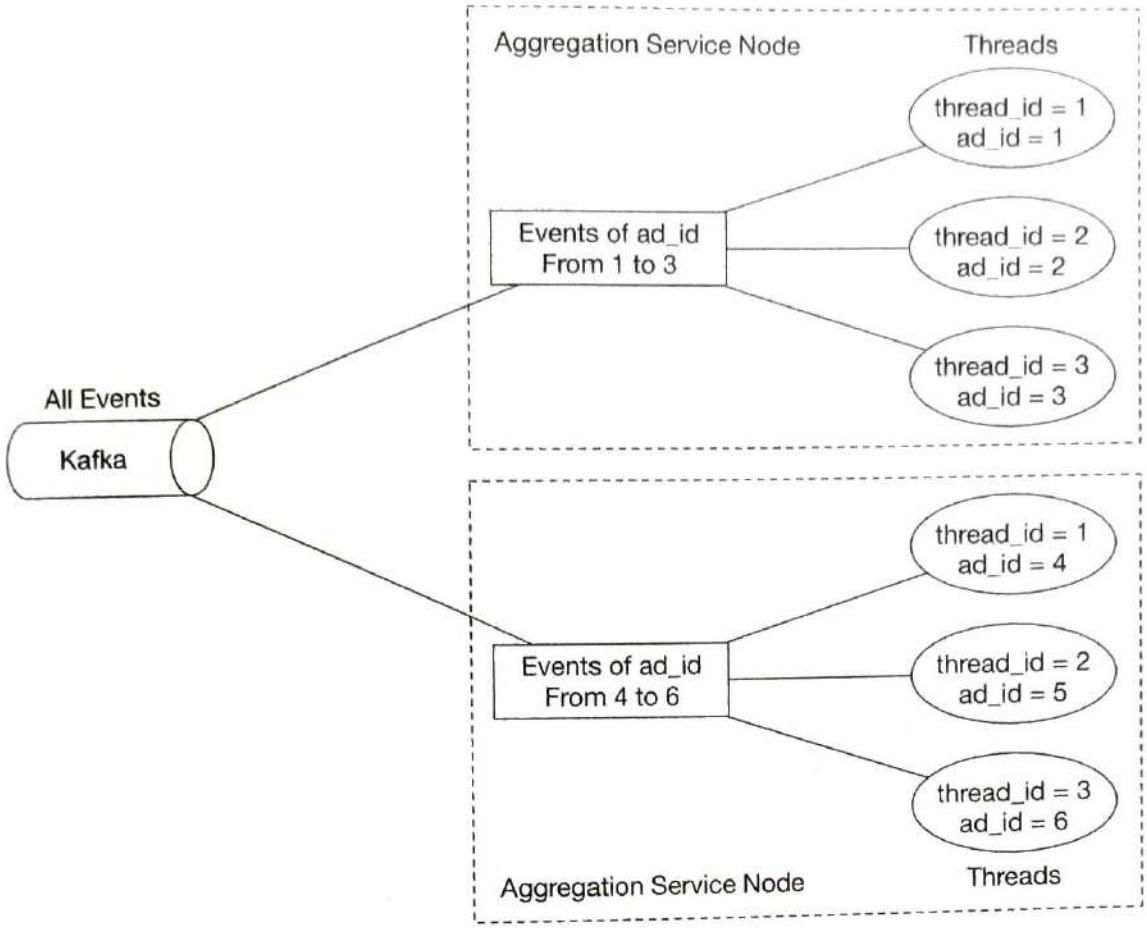


Figure 6.23: Multi-threading

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.

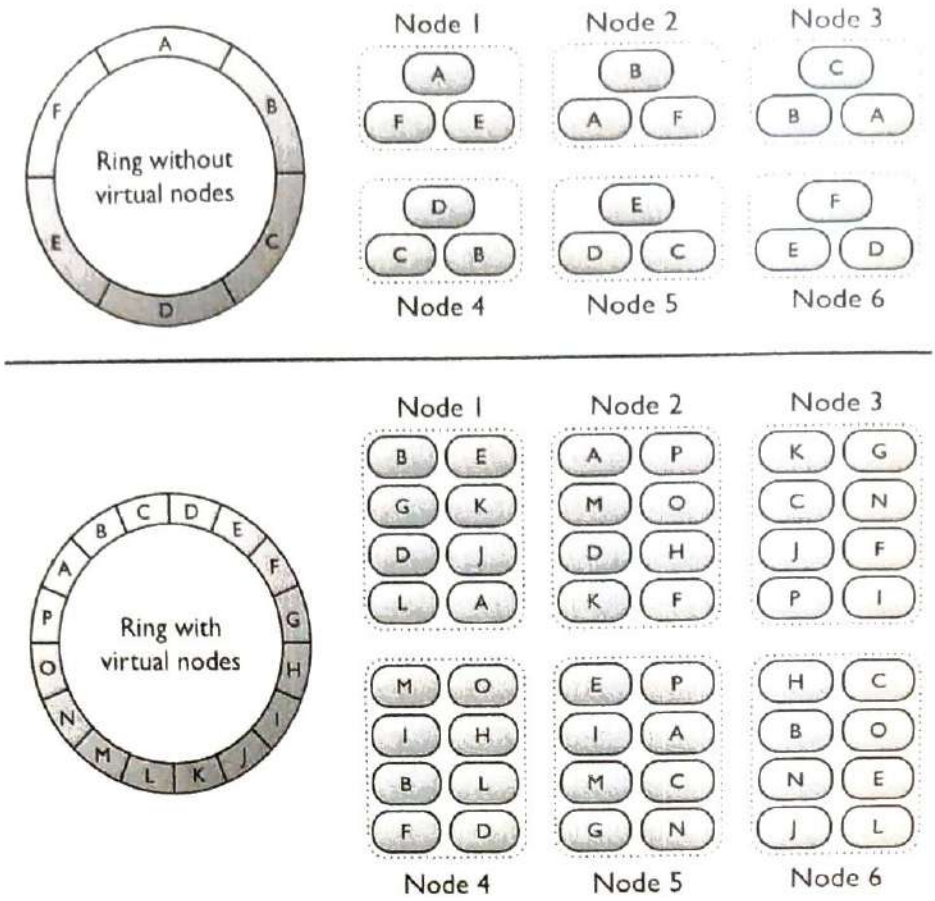


Figure 6.24: Virtual nodes [21]

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.

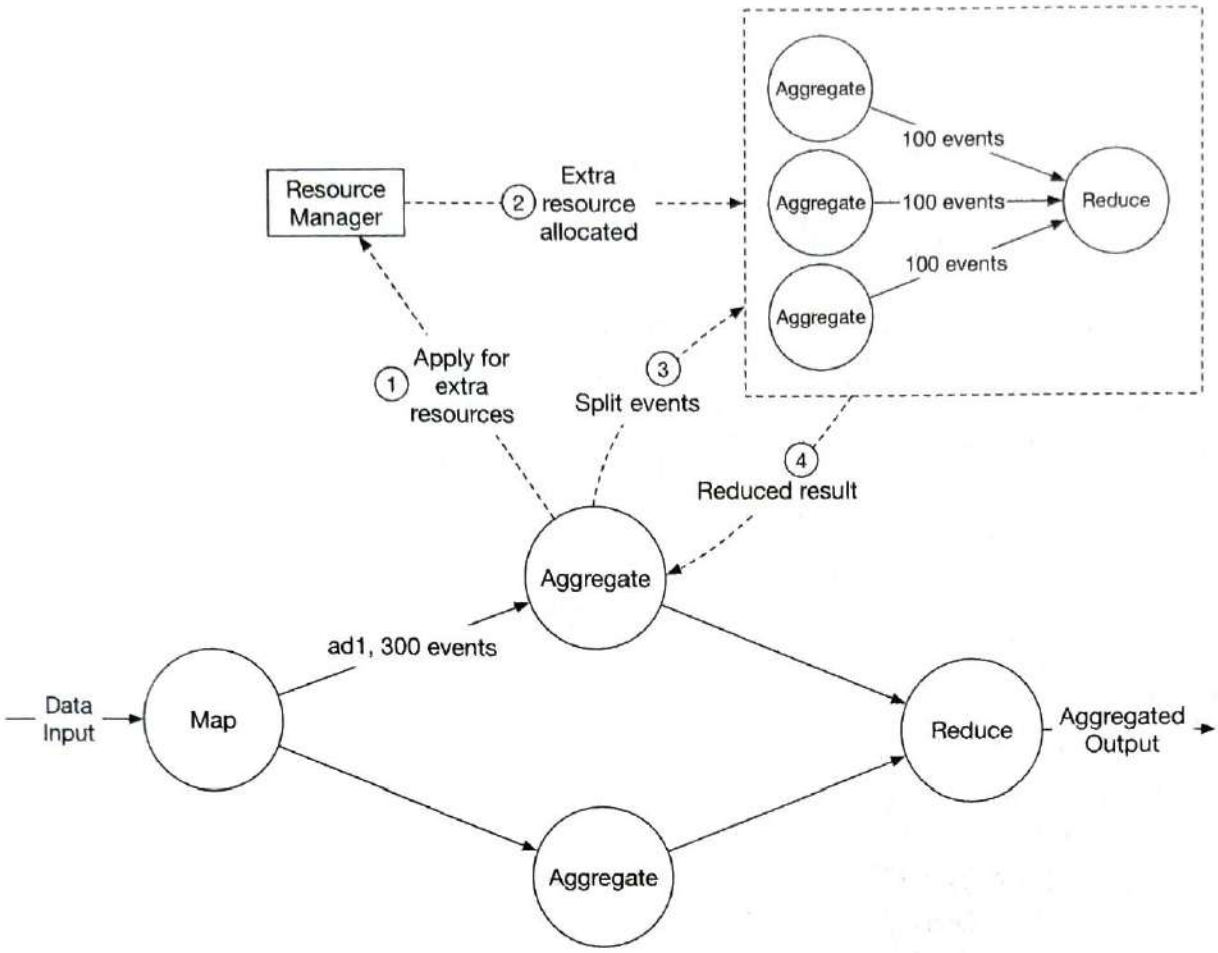


Figure 6.25: Allocate more aggregation nodes

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.

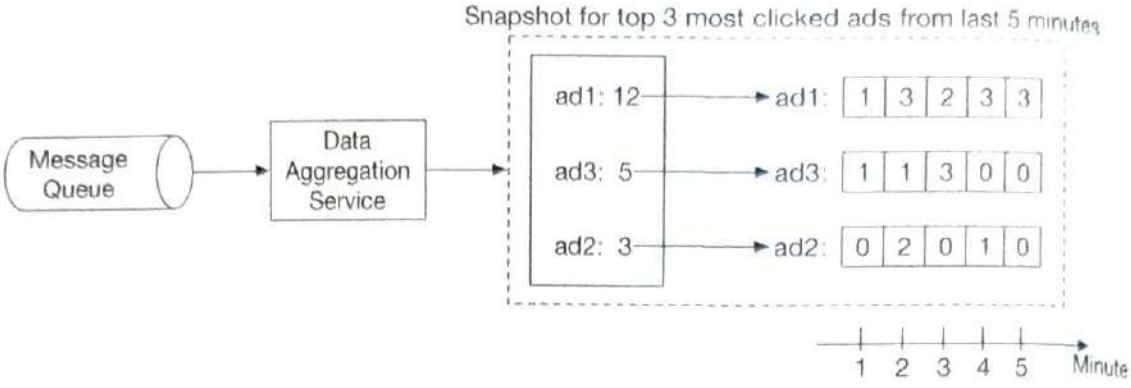


Figure 6.26: Data in a snapshot

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.

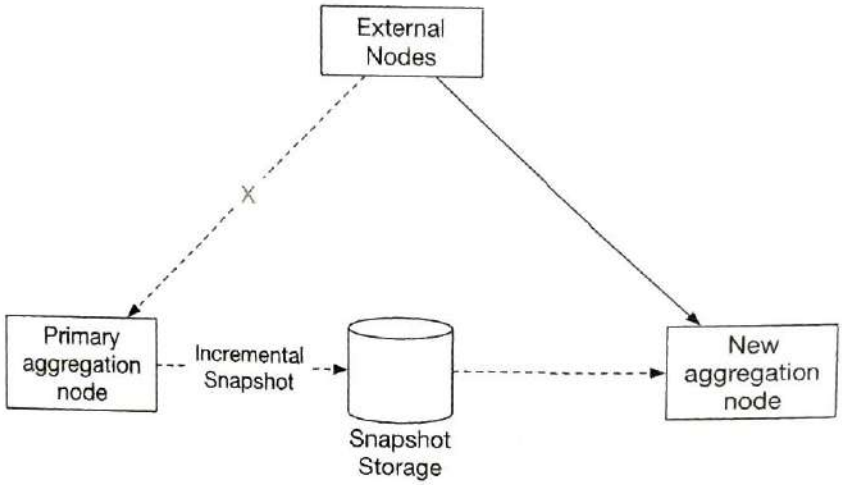


Figure 6.27: Aggregation node failover

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.

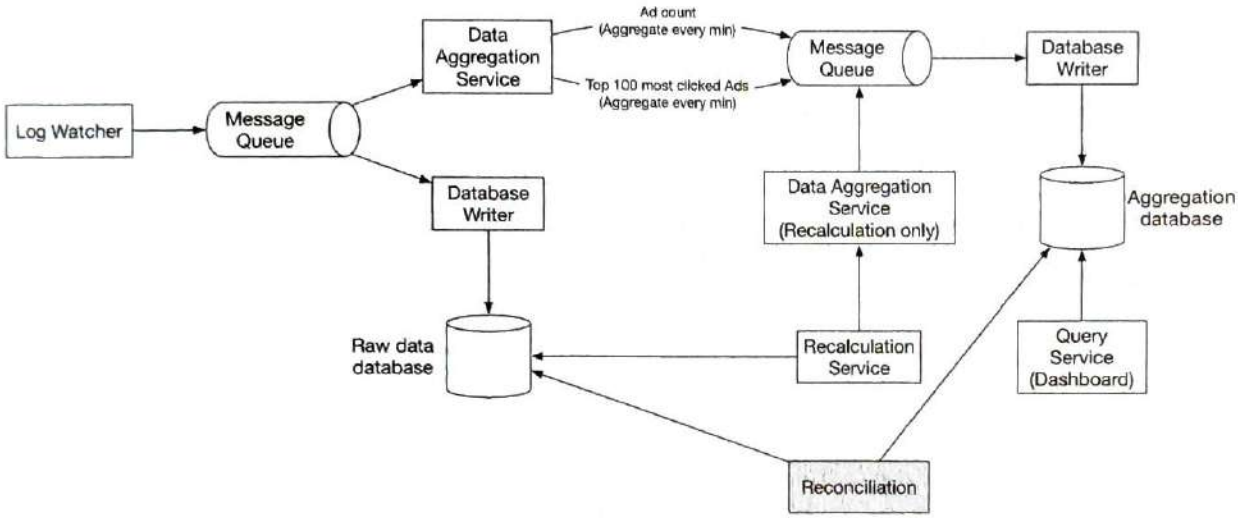


Figure 6.28: Final design

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.

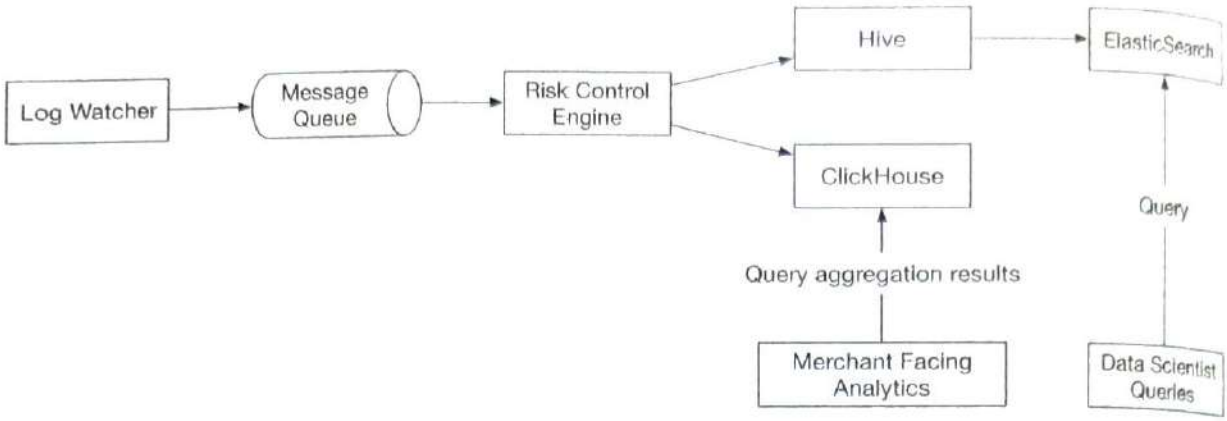


Figure 6.29: Alternative design

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

Step 4 - Wrap Up

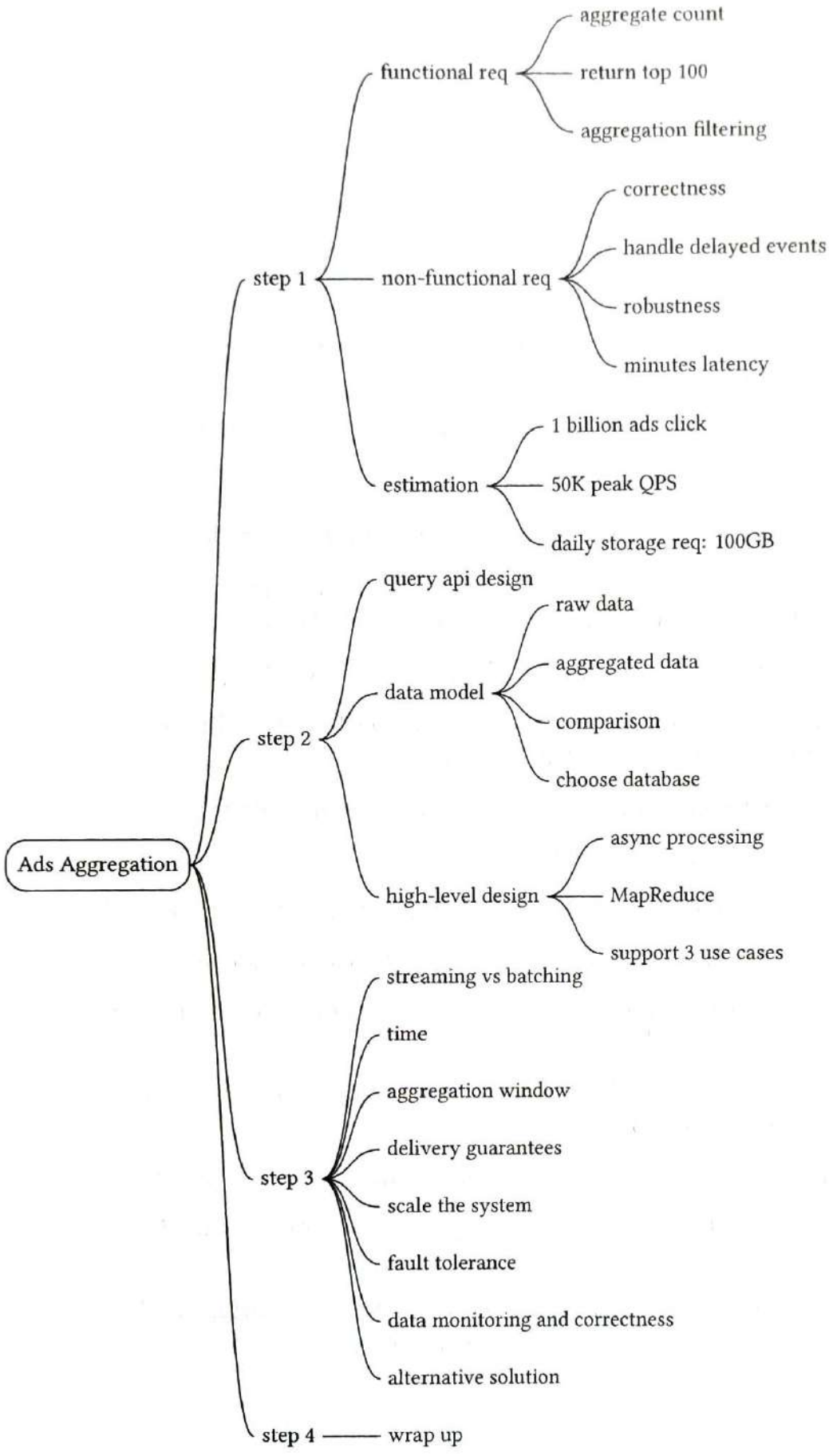
In this chapter, we went through the process of designing an ad click event aggregation system at the scale of Facebook or Google. We covered:

- 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).