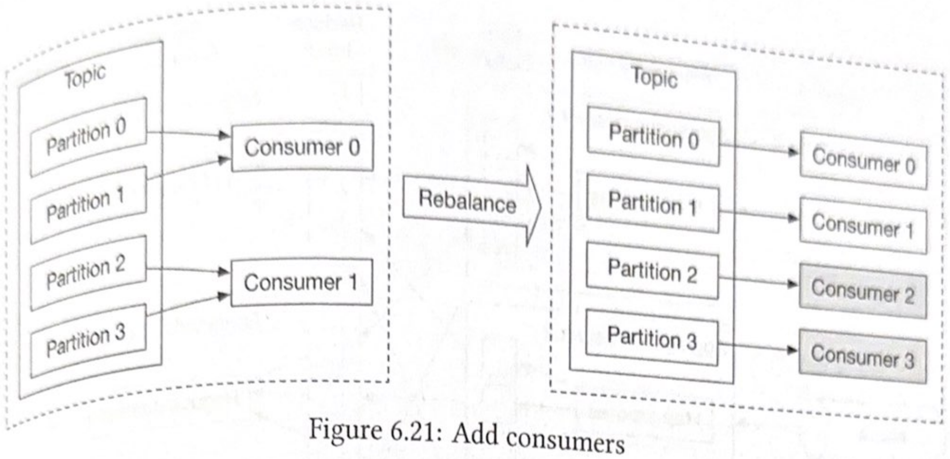


Figure 6.21: Add consumers

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.

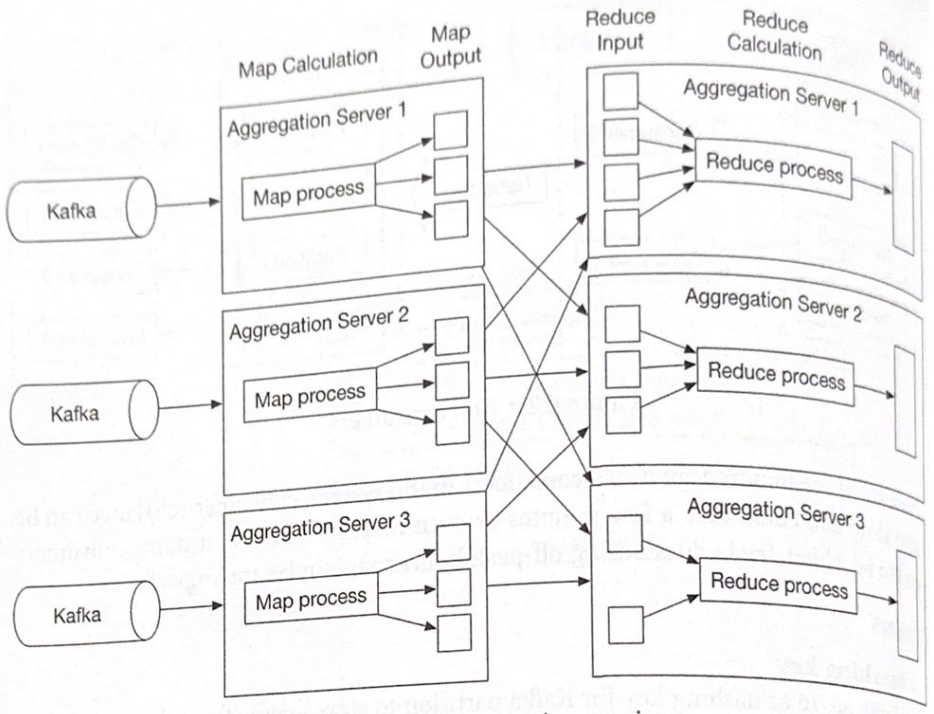


Figure 6.22: Aggregation service

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.

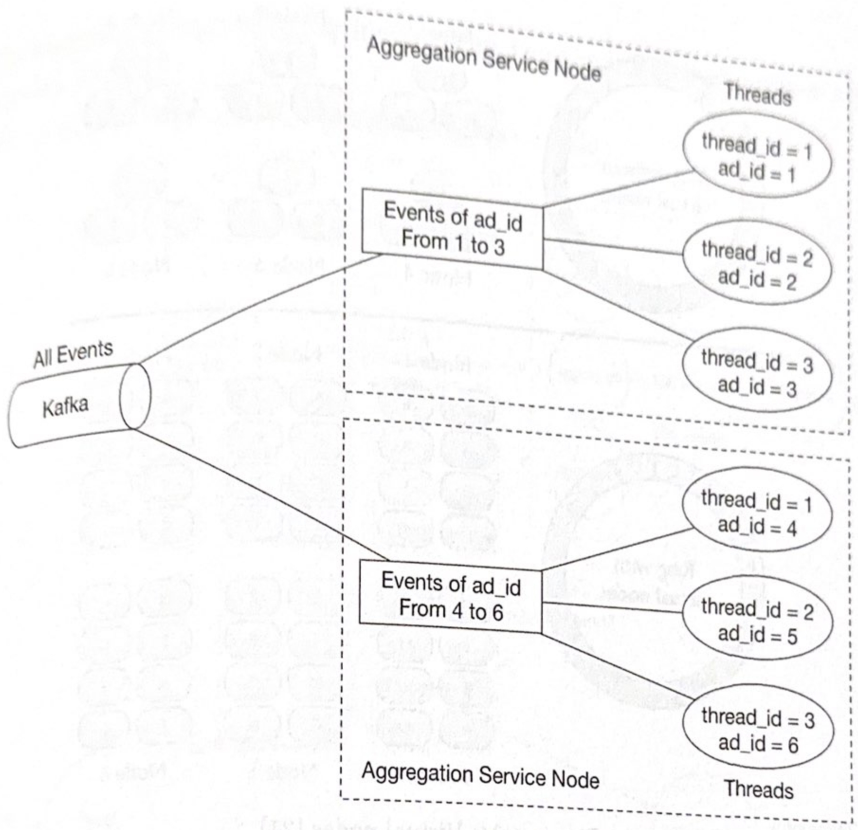


Figure 6.23: Multi-threading

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.

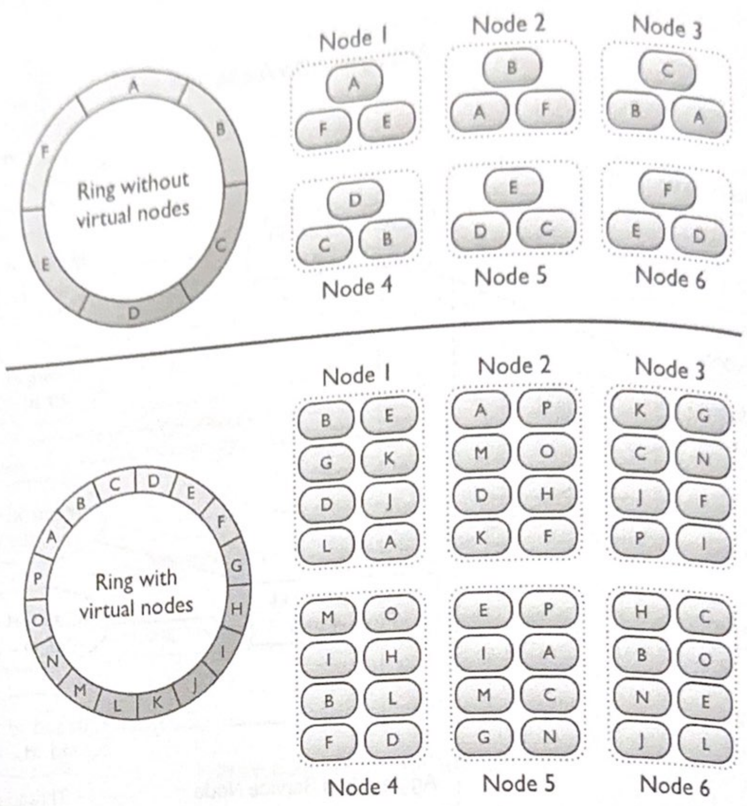


Figure 6.24: Virtual nodes [21]

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.

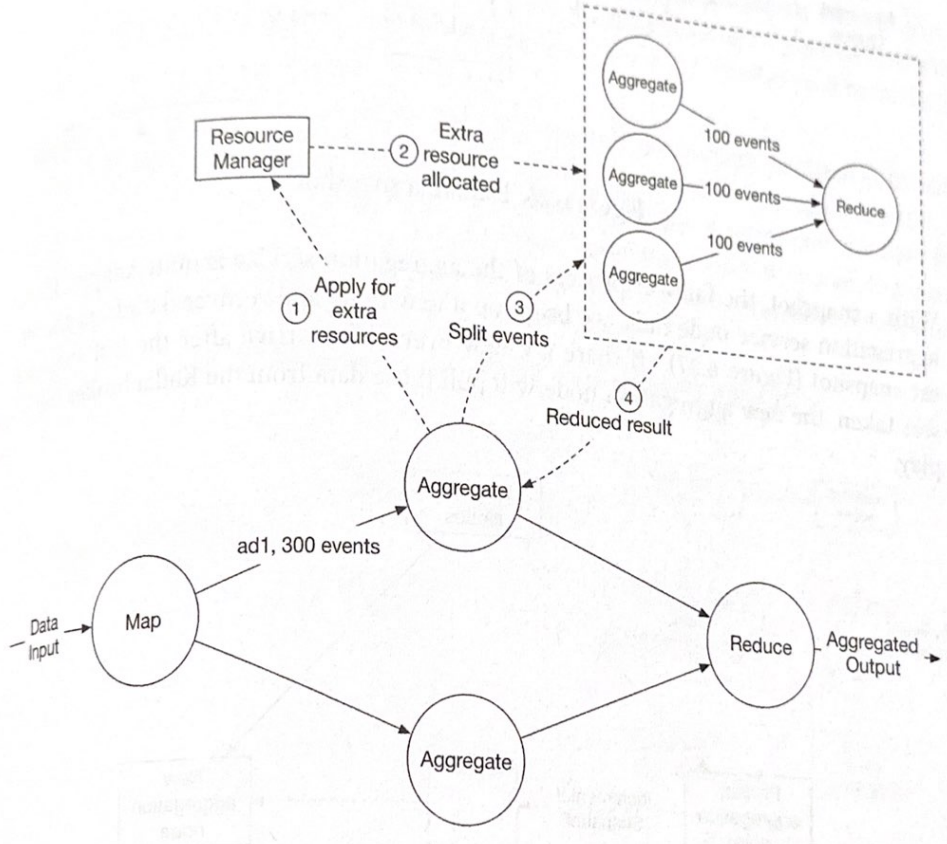


Figure 6.25: Allocate more aggregation nodes

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.

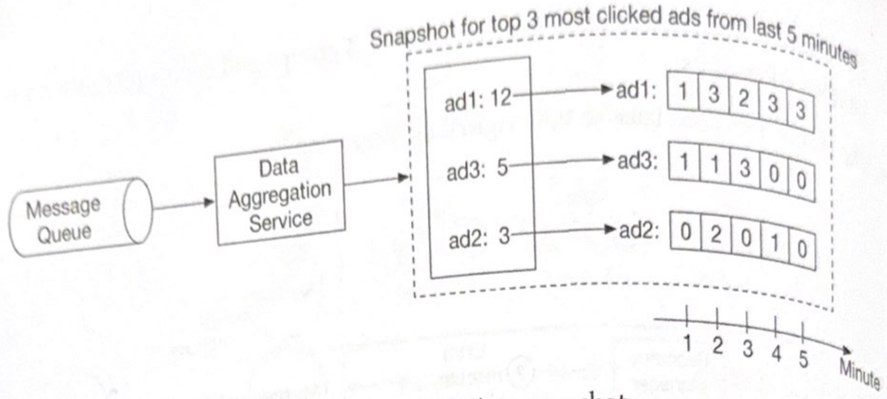


Figure 6.26: Data in a snapshot

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.

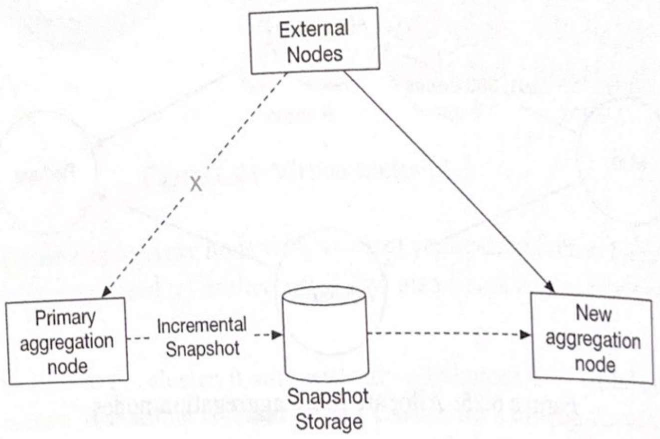


Figure 6.27: Aggregation node failover

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.

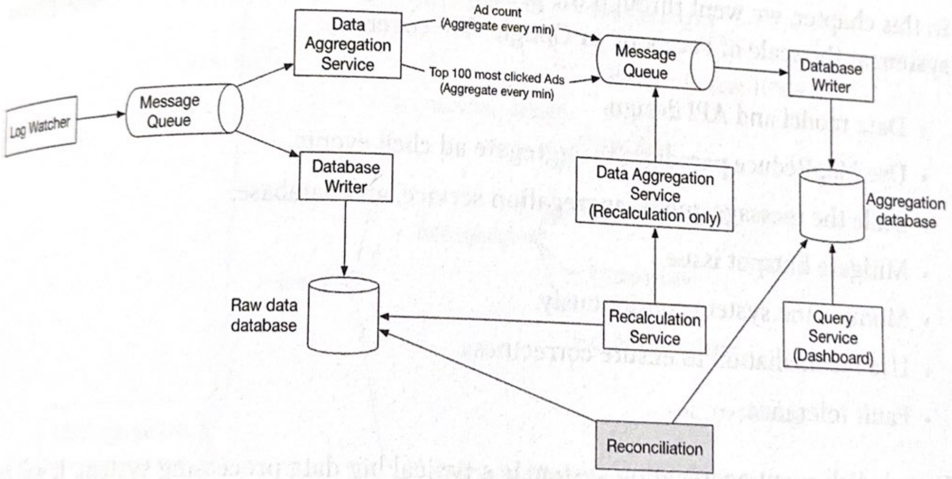


Figure 6.28: Final design

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.

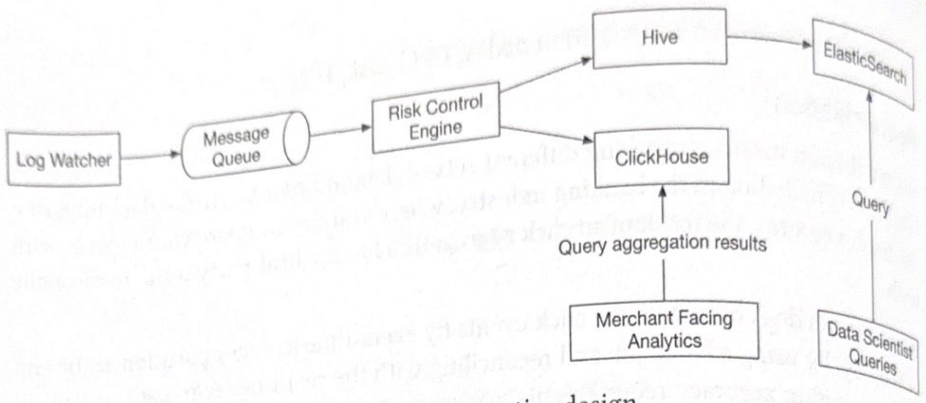


Figure 6.29: Alternative design

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

Step 4 - Wrap Up

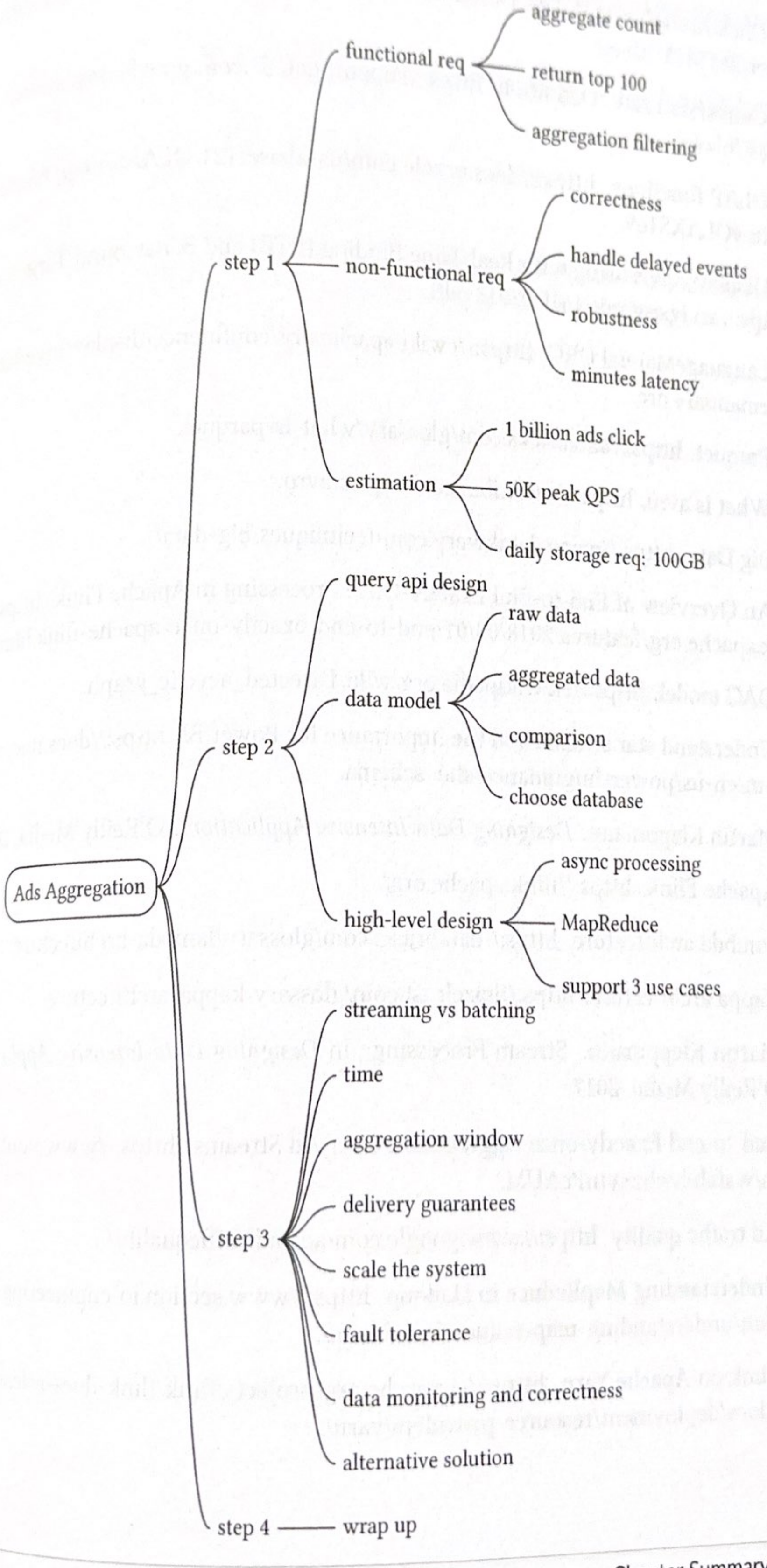
In this chapter, we went through the process of designing an ad click event aggregation system at the scale of Facebook or Google. We covered:

- 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/.