# 17 Design a dashboard of top 10 products on Amazon by sales volume

This chapter covers

* Scaling an aggregation operation on a large data stream
* Using a Lambda architecture for fast approximate results and slow accurate results
* Using Kappa architecture as an alternative to Lambda architecture
* Approximating an aggregation operation for faster speed

Analytics is a common discussion topic in a system design interview. We will always log certain network requests and user interactions, and we will perform analytics based on the data we collect.

The *Top K Problem (Heavy Hitters)* is a common type of dashboard. Based on the popularity or lack thereof of certain products, we can make decisions to promote or discontinue them. Such decisions may not be straightforward. For example, if a product is unpopular, we may decide to either discontinue it to save the costs of selling it, or we may decide to spend more resources to promote it to increase its sales.

The Top K Problem is a common topic we can discuss in an interview when discussing analytics, or it may be its own standalone interview question. It can take on endless forms. Some examples of the Top K Problem include

* Top-selling or worst-selling products on an ecommerce app by volume (this question) or revenue.
* The most-viewed or least-viewed products on an ecommerce app.
* Most downloaded apps on an apps store.
* Most watched videos on a video app like YouTube.
* Most popular (listened to) or least popular songs on a music app like Spotify.
* Most traded stocks on an exchange like Robinhood or E\*TRADE.
* Most forwarded posts on a social media app, such as the most retweeted Twitter tweets or most shared Instagram post.

## 17.1 Requirements

Let’s ask some questions to determine the functional and non-functional requirements. We assume that we have access to the data centers of Amazon or whichever ecommerce app we are concerned with

* How do we break ties?

High accuracy may not be important, so we can choose any item in a tie:

* Which time intervals are we concerned with?

Our system should be able to aggregate by certain specified intervals such as hour, day, week, or year:

* The use cases will influence the desired accuracy (and other requirements like scalability). What are the use cases of this information? What is the desired accuracy and desired consistency/latency?

That’s a good question. What do you have in mind?

It will be resource-intensive to compute accurate volumes and ranking in real time. Perhaps we can have a Lambda architecture, so we have an eventually consistent solution that offers approximate sales volumes and rankings within the last few hours and accurate numbers for time periods older than a few hours.

We can also consider trading off accuracy for higher scalability, lower cost, lower complexity, and better maintainability. We expect to compute a particular Top K list within a particular period at least hours after that period has passed, so consistency is not a concern.

Low latency is not a concern. We can expect that generating a list will require many minutes:

* Do we need just the Top K or top 10, or the volumes and ranking of an arbitrary number of products?

Similar to the previous question, we can accept a solution that provides the approximate volumes and ranking of the top 10 products within the last few hours, and volumes and ranking of any arbitrary number of products for time periods older than a few hours, potentially up to years. It’s also fine if our solution can display more than 10 products:

* Do we need to show the sale counts on the Top K list or just the product sales rankings?

We will show both the rankings and counts. This seems like a superfluous question, but there might be possible design simplifications if we don’t need to display certain data:

* Do we need to consider events that occur after a sale? A customer may request a refund, an exchange for the same or different product(s), or a product may be recalled.

This is a good question that demonstrates one’s industry experience and attention to detail. Let’s assume we can consider only the initial sales events and disregard subsequent events like disputes or product recalls:

* Let’s discuss scalability requirements. What is the sales transaction rate? What is the request rate for our Heavy Hitters dashboard? How many products do we have?

Assume 10 billion sales events per day (i.e., heavy sales transaction traffic). At 1 KB/event, the write rate is 10 TB/day. The Heavy Hitters dashboard will only be viewed by employees, so it will have low request rate. Assume we have ~1M products.

We do not have other non-functional requirements. High availability or low latency (and the corresponding complexities they will bring to our system design) are not required.

## 17.2 Initial thoughts

Our first thought may be to log the events to a distributed storage solution, like HDFS or Elasticsearch, and run a MapReduce, Spark, or Elasticsearch query when we need to compute a list of Top K products within a particular period. However, this approach is computationally intensive and may take too long. It may take hours or days to compute a list of Top K products within a particular month or year.

If we don’t have use cases for storing our sales event logs other than generating this list, it will be wasteful to store these logs for months or years just for this purpose. If we log millions of requests per second, it can add up to PBs/year. We may wish to store a few months or years of raw events for various purposes, including serving customer disputes and refunds, for troubleshooting or regulatory compliance purposes. However, this retention period may be too short for generating our desired Top K list.

We need to preprocess our data prior to computing these Top K lists. We should periodically perform aggregation and count the sales of our products, bucketing by hour, day, week, month, and year. Then we can perform these steps when we need a Top K list:

1. If needed, sum the counts of the appropriate buckets, depending on the desired period. For example, if we need the Top K list of a period of one month, we simply use that month’s bucket. If we need a particular three-month period, we sum the counts of the one-month buckets of that period. This way, we can save storage by deleting events after we sum the counts.
2. Sort these sums to obtain the Top K list.

We need to save the buckets because the sales can be very uneven. In an extreme situation, a product “A” may have 1M sales within a particular hour during a particular year, and 0 sales at all other times during that year, while sales of all other products may sum to far less than 1M total sales in that year. Product A will be in the Top K list of any period that includes that hour.

The rest of this chapter is about performing these operations at scale in a distributed manner.

## 17.3 Initial high-level architecture

We first consider Lambda architecture. *Lambda architecture* is an approach to handling massive quantities of data by using both batch and streaming methods (Refer to <https://www.databricks.com/glossary/lambda-architecture> or <https://www.snowflake.com/guides/lambda-architecture>.) Referring to figure 17.1, our lambda architecture consists of two parallel data processing pipelines and a serving layer that combines the results of these two pipelines:

1. A streaming layer/pipeline that ingests events in real time from all data centers where sales transactions occur and uses an approximation algorithm to compute the sales volumes and rankings of the most popular products.
2. A batch layer, or batch pipelines that run periodically (hourly, daily, weekly, and yearly) to compute accurate sales volumes and rankings. For our users to see the accurate numbers as they become available, our batch pipeline ETL job can contain a task to overwrite the results of the streaming pipeline with the batch pipeline’s whenever the latter are ready.

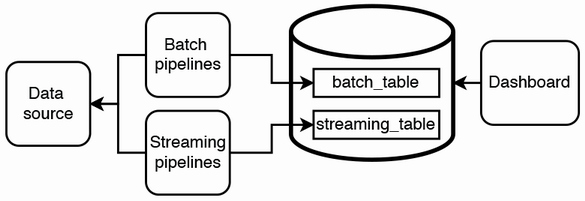


Figure 17.1 A high-level sketch of our Lambda architecture. Arrows indicate the direction of requests. Data flows through our parallel streaming and batch pipelines. Each pipeline writes its final output to a table in a database. The streaming pipeline writes to the speed\_table while the batch pipeline writes to the batch\_table. Our dashboard combines data from the speed\_table and batch\_table to generate the Top K lists.

Following an EDA (Event Driven Architecture) approach, the sales backend service sends events to a Kafka topic, which can be used for all downstream analytics such as our Top K dashboard.

## 17.4 Aggregation service

An initial optimization we can make to our Lambda architecture is to do some aggregation on our sales events and pass these aggregated sales events to both our streaming and batch pipelines. Aggregation can reduce the cluster sizes of both our streaming and batch pipelines. We sketch a more detailed initial architecture in figure 17.2. Our streaming and batch pipelines both write to an RDBMS (SQL), which our dashboard can query with low latency. We can also use Redis if all we need is simple key-value lookups, but we will likely desire filter and aggregation operations for our dashboard and other future services.

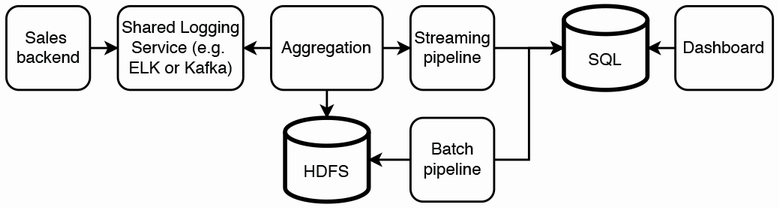


Figure 17.2 Our Lambda architecture, consisting of an initial aggregation service and streaming and batch pipelines. Arrows indicate the direction of requests. The sales backend logs events (including sales events) to a shared logging service, which is the data source for our dashboard. Our aggregation service consumes sales events from our shared logging service, aggregates them, and flushes these aggregated events to our streaming pipeline and to HDFS. Our batch pipeline computes the counts from our HDFS data and writes it to the SQL batch\_table. Our streaming pipeline computes the counts faster and less accurately than our batch pipeline and writes it to the SQL speed\_table. Our dashboard uses a combination of data from batch\_table and speed\_table to generate the Top K lists.

**NOTE** Event Driven Architecture (EDA) uses events to trigger and communicate between decoupled services (<https://aws.amazon.com/event-driven-architecture/>). Refer to other sources for more information, such as section 5.1 or page 295 of *Web Scalability for Startup Engineers* (2015) by Artur Ejsmont for an introduction to event-driven architecture.

We discussed aggregation and its benefits and tradeoffs in section 4.5. Our aggregation service consists of a cluster of hosts that subscribe to the Kafka topic that logs sales events, aggregates the events, and flushes/writes the aggregated events to HDFS (via Kafka) and to our streaming pipeline.

### 17.4.1 Aggregating by product ID

For example, a raw sales event may contain fields like (timestamp, product ID) while an aggregated event may be of the form (product\_id, start\_time, end\_time, count, aggregation\_host\_id). We can aggregate the events since their exact timestamps are unimportant. If certain time intervals are important (e.g., hourly), we can ensure that (start\_time, end\_time) pairs are always within the same hour. For example, (0100, 0110) is ok, but (0155, 0205) is not.

### 17.4.2 Matching host IDs and product IDs

Our aggregation service can partition by product ID, so each host is responsible for aggregating a certain set of IDs. For simplicity, we can manually maintain a map of (host ID, product ID). There are various implementation options for this configuration, including

1. A configuration file included in the service’s source code. Each time we change the file, we must restart the entire cluster.
2. A configuration file in a shared object store. Each host of the service reads this file on startup and stores in memory the product IDs that it is responsible for. The service also needs an endpoint to update its product IDs. When we change the file, we can call this endpoint on the hosts that will consume different product IDs.
3. Storing the map as a database table in SQL or Redis.
4. Sidecar pattern, in which a host makes a fetch request to the sidecar. The sidecar fetches an event of the appropriate product IDs and returns it to the host.

We will usually choose option 2 or 4 so we will not need to restart the entire cluster for each configuration change. We choose a file over a database for the following reasons:

* It is easy to parse a configuration file format such as YAML and JSON directly into a hash map data structure. More code is required to achieve the same effect with a database table. We will need to code with an ORM framework, code the database query and the data access object, and match the data access object with the hash map.
* The number of hosts will likely not exceed a few hundred or a few thousand, so the configuration file will be tiny. Each host can fetch the entire file. We do not need a solution with the low latency read performance of a database.
* The configuration does not change frequently enough to justify the overhead of a database like SQL or Redis.

### 17.4.3 Storing timestamps

If we need the exact timestamps to be stored somewhere, this storage should be handled by the sales service and not by an analytics or Heavy Hitters service. We should maintain separation of responsibility. There will be numerous analytics pipelines being defined on the sales events besides Heavy Hitters. We should have full freedom to develop and decommission these pipelines without regard to other services. In other words, we should be careful in deciding if other services should be dependencies of these analytics services.

### 17.4.4 Aggregation process on a host

An aggregation host contains a hash table with key of product ID and value of count. It also does checkpointing on the Kafka topic that it consumes, writing checkpoints to Redis. The checkpoints consist of the IDs of the aggregated events. The aggregation service can have more hosts than the number of partitions in the Kafka topic, though this is unlikely to be necessary since aggregation is a simple and fast operation. Each host repeatedly does the following:

1. Consume an event from the topic.
2. Update its hash table.

An aggregation host may flush its hash table with a set periodicity or when its memory is running out, whichever is sooner. A possible implementation of the flush process is as follows:

1. Produce the aggregated events to a Kafka topic that we can name “Flush.” If the aggregated data is small (e.g., a few MB), we can write it as a single event, consisting of a list of product ID aggregation tuples with the fields (“product ID,” “earliest timestamp,” “latest timestamp,” “number of sales”), such as, for example, [(123, 1620540831, 1620545831, 20), (152, 1620540731, 1620545831, 18), . . . ].
2. Using change data capture (CDC, refer to section 5.3), each destination has a consumer that consumes the event and writes to it:
   1. Write the aggregated events to HDFS.
   2. Write a tuple checkpoint to Redis with the status “complete” (e.g., {“hdfs”: “1620540831, complete”}).
   3. Repeat steps 2a–c for the streaming pipeline.

If we did not have this “Flush” Kafka topic, and a consumer host fails while writing an aggregated event to a particular destination, the aggregation service will need to reaggregate those events.

Why do we need to write two checkpoints? This is just one of various possible algorithms to maintain consistency.

If a host fails during step 1, another host can consume the flush event and perform the writes. If the host fails during step 2a, the write to HDFS may have succeeded or failed, and another host can read from HDFS to check if the write succeeded or if it needs to be retried. Reading from HDFS is an expensive operation. As a host failure is a rare event, this expensive operation will also be rare. If we are concerned with this expensive failure recovery mechanism, we can implement the failure recovery mechanism as a periodic operation to read all “processing” checkpoints between a minute and a few minutes old.

The failure recovery mechanism should itself be idempotent in case it fails while in progress and has to be repeated.

We should consider fault-tolerance. Any write operation may fail. Any host in the aggregation service, Redis service, HDFS cluster, or streaming pipeline can fail at any time. There may be network problems that interrupt write requests to any host on a service. A write event response code may be 200 but a silent error actually occurred. Such events will cause the three services to be in an inconsistent state. Therefore, we write a separate checkpoint for HDFS and our streaming pipeline. The write event should have an ID, so the destination services may perform deduplication if needed.

In such situations where we need to write an event to multiple services, what are the possible ways to prevent such inconsistency?

1. Checkpoint after each write to each service, which we just discussed.
2. We can do nothing if our requirements state that inconsistency is acceptable. For example, we may tolerate some inaccuracy in the streaming pipeline, but the batch pipeline must be accurate.
3. Periodic auditing (also called supervisor). If the numbers do not line up, discard the inconsistent results and reprocess the relevant data.
4. Use distributed transaction techniques such as 2PC, Saga, Change Data Capture, or Transaction Supervisor. These were discussed in chapter 4 and appendix D.

As discussed in section 4.5, the flip side of aggregation is that real-time results are delayed by the time required for aggregation and flushing. Aggregation may be unsuitable for if our dashboard requires low-latency updates.

## 17.5 Batch pipeline

Our batch pipeline is conceptually more straightforward than the streaming pipeline, so we can discuss it first.

Figure 17.3 shows a simplified flow diagram of our batch pipeline. Our batch pipeline consists of a series of aggregation/rollup tasks by increasing intervals. We roll up by hour, then day, then week, and then month and year. If we have 1M product IDs:

1. Rollup by hour will result in 24M rows/day or 168M rows/week.
2. Rollup by month will result in 28–31M rows/month or 336–372M rows/year.
3. Rollup by day will result in 7M rows/week or 364M rows/year.

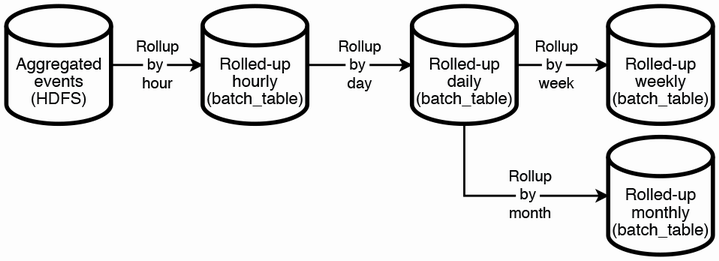


Figure 17.3 Simplified flow diagram of the rollup tasks in our batch pipeline. We have a rollup job that progressively rolls up by increasing time intervals to reduce the number of rows processed in each stage.

Let’s estimate the storage requirements. 400M rows each with 10 64-bit columns occupy 32 GB. This can easily fit into a single host. The hourly rollup job may need to process billions of sales events, so it can use a Hive query to read from HDFS and then write the resulting counts to the SQL batch\_table. The rollups for other intervals use the vast reduction of the number of rows from the hourly rollup, and they only need to read and write to this SQL batch\_table.

In each of these rollups, we can order the counts by descending order, and write the Top K (or perhaps K\*2 for flexibility) rows to our SQL database to be displayed on our dashboard.

Figure 17.4 is a simple illustration of our ETL DAG for one stage in our batch pipeline (i.e., one rollup job). We will have one DAG for each rollup (i.e., four DAG in total). An ETL DAG has the following four tasks. The third and fourth are siblings. We use Airflow terminology for DAG, task, and run:

1. For any rollup greater than hourly, we need a task to verify that the dependent rollup runs have successfully completed. Alternatively, the task can verify that the required HDFS or SQL data is available, but this will involve costly database queries.
2. Run a Hive or SQL query to sum the counts in descending order and write the result counts to the batch\_table.
3. Delete the corresponding rows on the speed\_table. This task is separate from task 2 because the former can be rerun without having to rerun the latter. Should task 3 fail while it is attempting to delete the rows, we should rerun the deletion without having to rerun the expensive Hive or SQL query of step 2.
4. Generate or regenerate the appropriate Top K lists using these new batch\_table rows. As discussed later in section 17.5, these Top K lists most likely have already been generated using both our accurate batch\_table data and inaccurate speed\_table table, so we will be regenerating these lists with only our batch\_table. This task is not costly, but it can also be rerun independently if it fails, so we implement it as its own task.

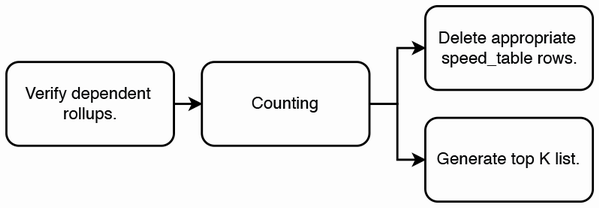


Figure 17.4 An ETL DAG for one rollup job. The constituent tasks are to verify that dependent rollups have completed, perform the rollup/counting and persist the counts to SQL, and then delete the appropriate speed\_table rows because they are no longer needed.

Regarding task 1, the daily rollup can only happen if all its dependent hourly rollups have been written to HDFS, and likewise for the weekly and monthly rollups. One daily rollup run is dependent on 24 hourly rollup runs, one weekly rollup run is dependent on seven daily rollup runs, and one monthly rollup run is dependent on 28–30 daily rollup runs depending on the month. If we use Airflow, we can use ExternalTaskSensor (<https://airflow.apache.org/docs/apache-airflow/stable/howto/operator/external_task_sensor.html#externaltasksensor>) instances with the appropriate execution\_date parameter values in our daily, weekly, and monthly DAGs to verify that the dependent runs have successfully completed.

## 17.6 Streaming pipeline

A batch job may take many hours to complete, which will affect the rollups for all intervals. For example, the Hive query for the latest hourly rollup job may take 30 minutes to complete, so the following rollups and by extension their Top K lists will be unavailable:

* The Top K list for that hour.
* The Top K list for the day that contains that hour will be unavailable.
* The Top K lists for the week and month that contains that day will be unavailable.

The purpose of our streaming pipeline is to provide the counts (and Top K lists) that the batch pipeline has not yet provided. The streaming pipeline must compute these counts much faster than the batch pipeline and may use approximation techniques.

After our initial aggregation, the next steps are to compute the final counts and sort them in descending order, and then we will have our Top K lists. In this section, we approach this problem by first considering an approach for a single host and then find how to make it horizontally scalable.

### 17.6.1 Hash table and max-heap with a single host

Our first attempt is to use a hash table and sort by frequency counts using a max-heap of size K. Listing 17.1 is a sample top K Golang function with this approach.

Listing 17.1 Sample Golang function to compute Top K list

type HeavyHitter struct {

identifier string

frequency int

}

func topK(events []String, int k) (HeavyHitter) {

frequencyTable := make(map[string]int)

for \_, event := range events {

value := frequencyTable[event]

if value == 0 {

frequencyTable[event] = 1

} else {

frequencyTable[event] = value + 1

}

}

pq = make(PriorityQueue, k)

i := 0

for key, element := range frequencyTable {

pq[i++] = &HeavyHitter{

identifier: key,

frequency: element

}

if pq.Len() > k {

pq.Pop(&pq).(\*HeavyHitter)

}

}

/\*

\* Write the heap contents to your destination.

\* Here we just return them in an array.

\*/

var result [k]HeavyHitter

i := 0

for pq.Len() > 0 {

result[i++] = pq.Pop(&pq).(\*HeavyHitter)

}

return result

}

In our system, we can run multiple instances of the function in parallel for our various time buckets (i.e., hour, day, week, month, and year). At the end of each period, we can store the contents of the max-heap, reset the counts to 0, and start counting for the new period.

### 17.6.2 Horizontal scaling to multiple hosts and multi-tier aggregation

Figure 17.5 illustrates horizontal scaling to multiple hosts and multi-tier aggregation. The two hosts in the middle column sum the (product, hour) counts from their upstream hosts in the left column, while the max-heaps in the right column aggregate the (product, hour) counts from their upstream hosts in the middle column.

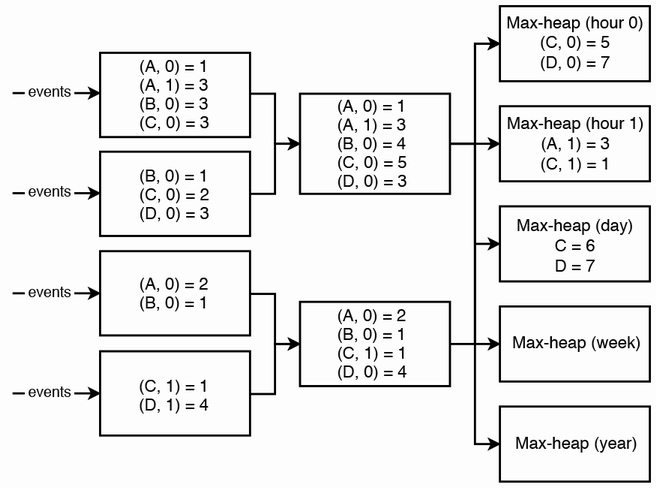


Figure 17.5 If the traffic to the final hash table host is too high, we can use a multi-tier approach for our streaming pipeline. For brevity, we display a key in the format (product, hour). For example, “(A, 0)” refers to product A at hour 0. Our final layer of hosts can contain max heaps, one for each rollup interval. This design is very similar to a multi-tier aggregation service discussed in section 4.5.2. Each host has an associated Kafka topic, which we don’t illustrate here.

In this approach, we insert more tiers between the first layer of hosts and the final hash table host, so no host gets more traffic than it can process. This is simply shifting the complexity of implementing a multi-tier aggregation service from our aggregation service to our streaming pipeline. The solution will introduce latency, as also described in section 4.5.2. We also do partitioning, following the approach described in section 4.5.3 and illustrated in figure 4.6. Note the discussion points in that section about addressing hot partitions. We partition by product ID. We may also partition by sales event timestamp.

Aggregation

Notice that we aggregate by the combination of product ID and timestamp. Before reading on, think about why.

Why do we aggregate by the combination of product ID and timestamp? This is because a Top K list has a period, with a start time and an end time. We need to ensure that each sales event is aggregated in its correct time ranges. For example, a sales event that occurred at 2023-01-01 10:08 UTC should be aggregated in

1. The hour range of [2023-01-01 10:08 UTC, 2023-01-01 11:00 UTC).
2. The day range of [2023-01-01, 2023-01-02).
3. The week range of [2022-12-28 00:00 UTC, 2023-01-05 00:00 UTC). 2022-12-28 and 2023-01-05 are both Mondays.
4. The month range of [2023-01-01, 2013-02-01).
5. The year range of [2023, 2024).

Our approach is to aggregate by the smallest period (i.e., hour). We expect any event to take only a few seconds to go through all the layers in our cluster, so it is unlikely for any key in our cluster that is more than an hour old. Each product ID has its own key. With the hour range appended to each key, it is unlikely that the number of keys will be greater than the number of product IDs times two.

One minute after the end of a period—for example, at 2023-01-01 11:01 UTC for [2023-01-01 10:08 UTC, 2023-01-01 11:00 UTC) or 2023-01-02 00:01 UTC for [2023-01-01, 2023-01-02)—the respective host in the final layer (whom we can refer to as *final hosts*) can write its heap to our SQL speed\_table, and then our dashboard is ready to display the corresponding Top K list for this period. Occasionally, an event may take more than a minute to go through all the layers, and then the final hosts can simply write their updated heaps to our speed\_table. We can set a retention period of a few hours or days for our final hosts to retain old aggregation keys, after which they can delete them.

An alternative to waiting one minute is to implement a system to keep track of the events as they pass through the hosts and trigger the final hosts to write their heaps to the speed\_table only after all the relevant events have reached the final hosts. However, this may be overly complex and also prevents our dashboard from displaying approximations before all events have been fully processed.

## 17.7 Approximation

To achieve lower latency, we may need to limit the number of layers in our aggregation service. Figure 17.6 is an example of such a design. We have layers that consist of just max-heaps. This approach trades off accuracy for faster updates and lower cost. We can rely on the batch pipeline for slower and highly accurate aggregation.

Why are max-heaps in separate hosts? This is to simplify provisioning new hosts when scaling up our cluster. As mentioned in section 3.1, a system is considered scalable if it can be scaled up and down with ease. We can have separate Docker images for hash table hosts and the max-heap host, since the number of hash table hosts may change frequently while there is never more than one active max-heap host (and its replicas).

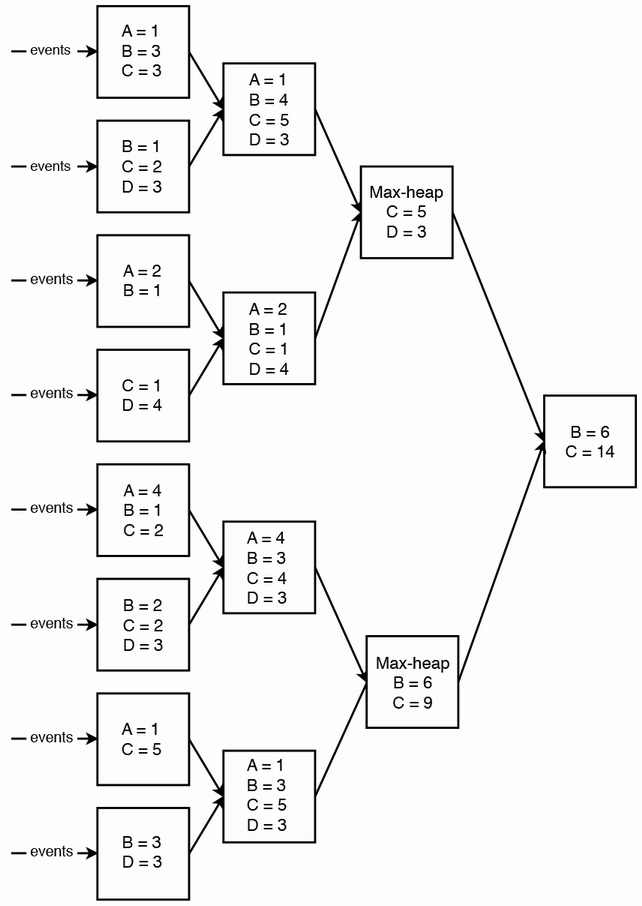


Figure 17.6 Multi-tier with max-heaps. The aggregation will be faster but less accurate. For brevity, we don’t display time buckets in this figure.

However, the Top K list produced by this design may be inaccurate. We cannot have a max-heap in each host and simply merge the max-heaps because if we do so, the final max-heap may not actually contain the Top K products. For example, if host one had a hash table {A: 7, B: 6, C: 5}, and host B had a hash table {A: 2, B: 4, C: 5}, and our max-heap is of size 2, host 1’s max-heap will contain {A: 7, B: 6} and host 2’s max-heap will contain {B: 4, C: 5}. The final combined max-heap will be {A: 7, B: 10}, which erroneously leaves C out of the top two list. The correct final max-heap should be {B: 10, C: 11}.

### 17.7.1 Count-min sketch

The previous example approaches require a large amount of memory on each host for a hash table of the same size as the number of products (in our case ~1M). We can consider trading off accuracy for lower memory consumption by using approximations.

Count-min sketch is a suitable approximation algorithm. We can think of it as a two-dimensional (2D) table with a width and a height. The width is usually a few thousand while the height is small and represents a number of hash functions (e.g., 5). The output of each hash function is bounded to the width. When a new item arrives, we apply each hash function to the item and increment the corresponding cell.

Let’s walk through an example of using count-min sketch with a simple sequence “A C B C C.” C is the most common letter and occurs three times. Tables 17.1–17.5 illustrate a count-min sketch table. We bold the hashed value in each step to highlight it.

1. Hash the first letter “A” with each of the five hash functions. Table 17.1 illustrates that each hash function hashes “A” to a different value.

Table 17.1 Sample count-min sketch table after adding a single letter “A”

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 |  |  |  |  |  |
|  | 1 |  |  |  |  |
|  |  |  |  | 1 |  |
|  | 1 |  |  |  |  |
|  | 1 |  |  |  |  |

2. Hash the second letter “C.” Table 17.2 illustrates that the first four hash functions hash “C” to a different value than “A.” The fifth hash function has a collision. The hashed values of “A” and “C” are identical, so that value is incremented.

Table 17.2 Sample count-min sketch table after adding “A C”

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 |  |  |  | **1** |  |
|  | 1 |  | **1** |  |  |
| **1** |  |  |  | 1 |  |
|  | 1 |  | **1** |  |  |
|  | **2 (collision)** |  |  |  |  |

3. Hash the third letter “B.” Table 17.3 illustrates that the fourth and fifth hash functions have collisions.

Table 17.3 Sample count-min sketch table after adding “A C B”

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 |  | **1** |  | 1 |  |
|  | 1 |  | 1 |  | **1** |
| 1 | **1** |  |  | 1 |  |
|  | **2 (collision)** |  | 1 |  |  |
|  | **3 (collision)** |  |  |  |  |

4. Hash the fourth letter “C.” Table 17.4 illustrates that only the fifth hash function has a collision.

Table 17.4 Sample count-min sketch table after adding “A C B C”

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 |  | 1 |  | **2** |  |
|  | 1 |  | **2** |  | 1 |
| **2** | 1 |  |  | 1 |  |
|  | 2 |  | **2** |  |  |
|  | **4 (collision)** |  |  |  |  |

5. Hash the fifth letter “C.” The operation is identical to the previous step. Table 17.5 is the count-min sketch table after a sequence “A C B C C.”

Table 17.5 Sample count-min sketch table after a sequence “A C B C C”

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 |  | 1 |  | **3** |  |
|  | 1 |  | **3** |  | 1 |
| **3** | 1 |  |  | 1 |  |
|  | 2 |  | **3** |  |  |
|  | **5 (collision)** |  |  |  |  |

To find the item with the highest number of occurrences, we first take the maximum of each row {3, 3, 3, 3, 5} and then the minimum of these maximums “3.” To find the item with the second highest number of occurrences, we first take the second highest number in each row {1, 1, 1, 2, 5} and then the minimum of these numbers “1.” And so on. By taking the minimum, we decrease the chance of overestimation.

There are formulas that help to calculate the width and height based on our desired accuracy and the probability we achieve that accuracy. This is outside the scope of this book.

The count-min sketch 2D array replaces the hash table in our previous approaches. We will still need a heap to store a list of heavy hitters, but we replace a potentially big hash table with a count-min sketch 2D array of predefined size that remains fixed regardless of the data set size.

## 17.8 Dashboard with Lambda architecture

Referring to figure 17.7, our dashboard may be a browser app that makes a GET request to a backend service, which in turn runs a SQL query. The discussion so far has been about a batch pipeline that writes to batch\_table and a streaming pipeline that writes to speed\_table, and the dashboard should construct the Top K list from both tables.

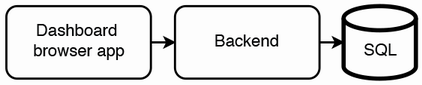


Figure 17.7 Our dashboard has a simple architecture, consisting of a browser app that makes GET requests to a backend service, which in turn makes SQL requests. The functional requirements of our browser app may grow over time, from simply displaying the top 10 lists of a particular period (e.g., the previous month), to include larger lists, more periods, filtering, or aggregation (like percentile, mean, mode, max, min).

However, SQL tables do not guarantee order, and filtering and sorting the batch\_table and speed\_table may take seconds. To achieve P99 of <1 second, the SQL query should be a simple SELECT query against a single view that contains the list of rankings and counts, which we refer to as the top\_1000 view. This view can be constructed by selecting the top 1,000 products from the speed\_table and batch\_table in each period. It can also contain an additional column that indicates whether each row is from the speed\_table or batch\_table. When a user requests a Top K dashboard for a particular interval, our backend can query this view to obtain as much data from the batch table as possible and fill in the blanks with the speed table. Referring to section 4.10, our browser app and backend service can also cache the query responses.

Exercise

As an exercise, define the SQL query for the top\_1000 view.

## 17.9 Kappa architecture approach

*Kappa architecture* is a software architecture pattern for processing streaming data, performing both batch and streaming processing with a single technology stack (<https://hazelcast.com/glossary/kappa-architecture>). It uses an append-only immutable log like Kafka to store incoming data, followed by stream processing and storage in a database for users to query.

In this section, we compare Lambda and Kappa architecture and discuss a Kappa architecture for our dashboard.

### 17.9.1 Lambda vs. Kappa architecture

Lambda architecture is complex because the batch layer and streaming layer each require their own code base and cluster, along with associated operational overhead and the complexity and costs of development, maintenance, logging, monitoring, and alerting.

Kappa architecture is a simplification of Lambda architecture, where there is only a streaming layer and no batch layer. This is akin to performing both streaming and batch processing on a single technology stack. The serving layer serves the data computed from the streaming layer. All data is read and transformed immediately after it is inserted into the messaging engine and processed by streaming techniques. This makes it suitable for low-latency and near real-time data processing like real-time dashboards or monitoring. As discussed earlier regarding the Lambda architecture streaming layer, we may choose to trade off accuracy for performance. But we may also choose not to make this tradeoff and compute highly accurate data.

Kappa architecture originated from the argument that batch jobs are never needed, and streaming can handle all data processing operations and requirements. Refer to <https://www.oreilly.com/radar/questioning-the-lambda-architecture/> and <https://www.kai-waehner.de/blog/2021/09/23/real-time-kappa-architecture-mainstream-replacing-batch-lambda/>, which discuss the disadvantages of batch and how streaming does not have them.

In addition to the points discussed in these reference links, another disadvantage of batch jobs compared to streaming jobs is the former’s much higher development and operational overheads because a batch job that uses a distributed file system like HDFS tends to take at least minutes to complete even when running on a small amount of data. This is due to HDFS’s large block size (64 or 128 MB compared to 4 KB for UNIX file systems) to trade off low latency for high throughput. On the other hand, a streaming job processing a small amount of data may only take seconds to complete.

Batch job failures are practically inevitable during the entire software development lifecycle from development to testing to production, and when a batch job fails, it must be rerun. One common technique to reduce the amount of time to wait for a batch job is to divide it into stages. Each stage outputs data to intermediate storage, to be used as input for the next stage. This is the philosophy behind Airflow DAGs. As developers, we can design our batch jobs to not take more than 30 minutes or one hour each, but developers and operations staff will still need to wait 30 minutes or one hour to see if a job succeeded or failed. Good test coverage reduces but does not eliminate production problems.

Overall, errors in batch jobs are more costly than in streaming jobs. In batch jobs, a single bug crashes an entire batch job. In streaming, a single bug only affects processing of that specific event.

Another advantage of Kappa vs. Lambda architecture is that the relative simplicity of the former, which uses a single processing framework while the latter may require different frameworks for its batch and streaming pipelines. We may use frameworks like Redis, Kafka, and Flink for streaming.

One consideration of Kappa architecture is that storing a large volume of data in an event-streaming platform like Kafka is costly and not scalable beyond a few PBs, unlike HDFS, which is designed for large volumes. Kafka provides infinite retention with *log compaction* (<https://kafka.apache.org/documentation/#compaction>), so a Kafka topic saves storage by only storing the latest value for each message key and delete all earlier values for that key. Another approach is to use object storage like S3 for long-term storage of data that is seldom accessed. Table 17.6 compares Lambda and Kappa architecture.

Table 17.6 Comparison between Lambda and Kappa architecture

| Lambda | Kappa |
| --- | --- |
| Separate batch and streaming pipelines. Separate clusters, code bases, and processing frameworks. Each needs its own infrastructure, monitoring, logs, and support. | Single pipeline, cluster, code base, and processing framework. |
| Batch pipelines allow faster performance with processing large amounts of data. | Processing large amounts of data is slower and more expensive than Lambda architecture. However, data is processed as soon as it is ingested, in contrast to batch jobs which run on a schedule, so the latter may provide data sooner. |
| An error in a batch job may require all the data to be reprocessed from scratch. | An error in a streaming job only requires reprocessing of its affected data point. |

### 17.9.2 Kappa architecture for our dashboard

A Kappa architecture for our Top K dashboard can use the approach in section 17.3.2, where each sales event is aggregated by its product ID and time range. We do not store the sales event in HDFS and then perform a batch job. A count of 1M products can easily fit in a single host, but a single host cannot ingest 1B events/day; we need multi-tier aggregation.

A serious bug may affect many events, so we need to log and monitor errors and monitor the rate of errors. It will be difficult to troubleshoot such a bug and difficult to rerun our streaming pipeline on a large number of events, so we can define a critical error rate and stop the pipeline (stop the Kafka consumers from consuming and processing events) if the error rate exceeded this defined critical error rate.

Figure 17.8 illustrates our high-level architecture with Kappa architecture. It is simply our Lambda architecture illustrated in figure 17.2 without the batch pipeline and aggregation service.

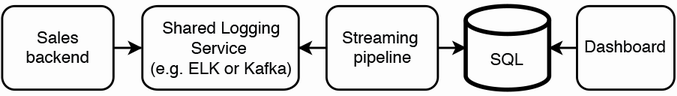


Figure 17.8 Our high-level architecture that uses Kappa architecture. It is simply our Lambda architecture in figure 17.2 without the batch pipeline and aggregation service.

## 17.10 Logging, monitoring, and alerting

Besides what was discussed in section 2.5, we should monitor and send alerts for the following.

Our shared batch ETL platform should already be integrated with our logging, monitoring, and alerting systems. We will be alerted to unusually long run time or failures of any of the tasks within our rollup job.

The rollup tasks write to HDFS tables. We can use the data quality monitoring tools described in chapter 10 to detect invalid datapoints and raise alerts.

## 17.11 Other possible discussion topics

Partition these lists by other characteristics, such as country or city. What design changes will we need to return the Top K products by revenue instead of sales volume? How can we track the Top K products by change in sales volume and/or revenue?

It may be useful to look up rankings and statistics of certain products with names or descriptions that match patterns. We can design a search system for such use cases.

We may discuss programmatic users of the Top K lists, such as machine learning and experimentation services. We had assumed a low request rate and that high availability and low latency were not required. These assumptions will no longer hold as programmatic users introduce new non-functional requirements.

Can our dashboard display approximations for the Top K lists before the events are fully counted, or perhaps even before the events occur?

A considerable complication with counting sales is disputes, such as customer requests for refunds or exchanges. Should sale numbers include disputes in progress? Do we need to correct the past data to consider refunds, returns, or exchanges? How do we recount sales events if refunds are granted or rejected or if there are exchanges for the same product or other product(s)?

We may offer a warranty of several years, so a dispute may occur years after the sale. Database queries may need to search for sales events that occurred years before. Such jobs may run out of memory. This is a challenging problem that is still faced by many engineers to this day.

There may be drastic events, such as a product recall. For example, we may need to recall a toy because it was suddenly found to be unsafe to children. We may discuss whether the counts of sales events should be adjusted if such problems occur.

Besides regenerating the Top K lists for the previous reasons, we may generalize this to regenerating the Top K lists from any data change.

Our browser app only displays the Top K list. We can extend our functional requirements, such as displaying sales trends or predicting future sales of current or new products.

## 17.12 References

This chapter used material from the Top K Problem (Heavy Hitters) (<https://youtu.be/kx-XDoPjoHw>) presentation in the System Design Interview YouTube channel by Mikhail Smarshchok.

## Summary

* When accurate large-scale aggregation operations take too long, we can run a parallel streaming pipeline that uses approximation techniques to trade off accuracy for speed. Running a fast, inaccurate and a slow, accurate pipeline in parallel is called Lambda architecture.
* One step in large-scale aggregation is to partition by a key that we will later aggregate over.
* Data that is not directly related to aggregation should be stored in a different service, so it can be easily used by other services.
* Checkpointing is one possible technique for distributed transactions that involve both destinations with cheap read operations (e.g., Redis) and expensive read operations (e.g., HDFS).
* We can use a combination of heaps and multi-tier horizontal scaling for approximate large-scale aggregation operations.
* Count-min sketch is an approximation technique for counting.
* We can consider either Kappa or Lambda architecture for processing a large data stream.