Designing Spotify's Top-K Ranking System

Let’s design Spotify’s Top Songs feature, which highlights the most popular tracks based on user play counts by using the Top K methodology to rank these tracks.

Similar Services: Apple Music, Amazon Music, YouTube Music

Difficulty Level: Medium

What is the Top-K Problem?

The top-K problem helps us find a specific number of the most important or popular items from a larger collection. For example, Spotify uses this idea to curate playlists like "Top 50 Global" or "Top 10 Pop Hits." This process involves analyzing a wealth of data, including user preferences, play counts, and interactions. Its aim? To identify the most popular songs over a certain period, such as the top-ranked songs in a day, month, or year. In this lesson, we'll create a simplified version of this feature, focusing on how a music streaming service can utilize the top-K problem to help users discover the most played songs worldwide. To begin, let's look into the requirements and objectives of the system.

2. Requirements and Goals of the System

Let’s design a Top Songs feature for Spotify with the following requirements:

Functional Requirements:

- **Real-time updates:** Play counts should update in real-time, based on users' song play counts.
- **Uniform visibility:** Ensure all users see the same list regardless of location or device.

- **Capacity:** The top K songs list should have the capacity for storing fewer than 1000 songs.
- **Timeframes:** Manage and update the list according to different timeframes (e.g., daily, weekly, monthly) to reflect current user preferences and trends.

Non-Functional Requirements :

- **High Availability:** The feature should be operational 99.99% of the time for almost constant access.
- **Fault Tolerance:** The system must continue operating smoothly even if some components fail.
- **Low Latency:** Responses for updating and accessing the song list should be within few milliseconds.
- **Scalability:** Handle high traffic volumes without performance degradation.

Some Design Considerations: The system would be large-scale, so we will focus on building a system that can retrieve the song list quickly.

- To meet scalability and performance targets with large data sets, the system could use approximate ranking techniques, which introduce a controlled error margin in the Top K results.
- Low latency is expected while viewing the top-ranked list.

3. Capacity Estimation and Constraints

Traffic estimates: Let's assume Spotify's total active user count is 270 million daily active users who, on average, play 10 songs per user each day. Calculating the total number of daily plays for all songs:

```
270 million * 10 songs/user = 2.7 billion songs played per day
```

From these plays, 20% are for top-ranked songs. Calculating the total number of daily plays for top-ranked songs:

```
2.7 billion songs/day * 20% = 540 million top-ranked song plays per day
```

This results in 540 million song requests per day for top-ranked songs, and approximately *6250* requests per second.

Storage estimates: Now let's first calculate the storage required for the top-k ranking list as we have already defined that the maximum list size would be `K<1000` considering an average data size of 16 bytes per song data:

```
16 * 999 = 15984 Bytes
```

Considering that the top-ranking list is updated on a real-time basis, we also need to calculate the data requirements for the songs played. Assuming the same metadata space for each played song, which is

16 bytes, and multiplying it by the total number of top-ranked song plays per day, which is 540 million:

```
16 Bytes * 540 million = 8.64GB/day
```

This calculation shows that we need approximately 8.64 GB per day just for storing plays of top-ranked songs. Additionally, if considering the storage needs for non-top-ranked songs. With 80% of the total plays being for non-top-ranked songs played daily, with each metadata also taking up 16 bytes:

```
16Bytes * 2.7 billion plays/day * 80% = 34.56 GB/day
```

Now let's sum up both storage needs:

```
8.64 GB + 34.56 GB = 43.2 GB/day
```

This indicates that approximately a total of 43.2 GB per day is needed to store data for all songs played on Spotify each day.

4. System APIs

> Now that we have finalized the requirements and estimates, it's time to define the system APIs. This should explicitly state what is expected from the system.

We can have SOAP or REST APIs to expose the functionality of our service. The following could be the definition of the API for getting top ranked playlist:

```
GET /top-ranked
```

Request Parameters

Parameters	Types	Description
api_dev_key (requierd)	string	Developer API key for authentication
user_id (required)	integer	User Id to identify the user requesting the list
limit (optional)	integer	Maximum number of items to include in the list
timeframe (optional)	string	Timeframe for which the top-ranked list is generated (e.g., "daily", "weekly", "monthly")
orderBy (optional)	string	Criteria for ordering the list and specifying ascending or descending order. Options include "asc" for ascending or "desc" for descending order.

Returns: (JSON) Returns a JSON object containing a list of Top ranked songs.

Response:


```
{
  "status": "success",
  "top_songs": [
    {
      "rank": 1,
      "song_id": 1,
      "title": "Song Title 1",
      "artist_id": 123,
      "album_id": 456,
      "play_count": 54321,
      "last_played_at": "2024-05-09T10:00:00Z",
      "genre": "Pop",
      "song_path": "https://example.com/song/1",
      "period": "weekly"
    },
    {
      "rank": 2,
      "song_id": 2,
      "title": "Song Title 2",
      "artist_id": 124,
      "album_id": 457,
      "play_count": 43210,
      "last_played_at": "2024-05-08T15:30:00Z",
      "genre": "Rock",
      "song_path": "https://example.com/song/2",
      "period": "weekly"
    }
    // More songs...
```

```
    ]
}
```

We would have another endpoint for playing the song:

```
GET /play-song
```

Request Parameters

Parameters	Types	Description
api_dev_key (requierd)	string	Developer API key for authentication
user_id (required)	string	User Id to identify the user requesting the list
song_id (required)	string	Song ID that a user will play

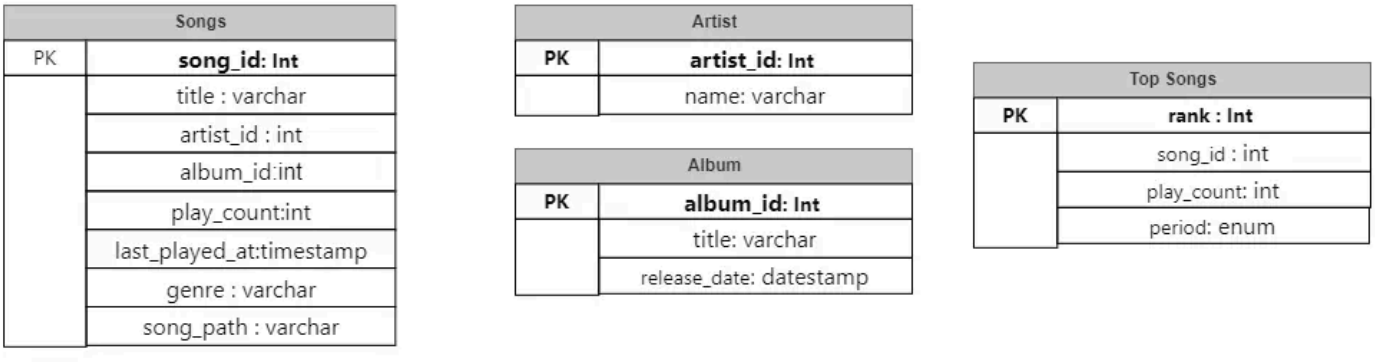
Returns: (JSON) Returns a JSON object containing all the relevant details about the song like Song name, artist, genre, and link of the song that will be used to play the song.

Response:

```
{
  "status": "success",
  "song": {
    "song_id": 1,
    "title": "Song Title 1",
    "artist_id": 123,
    "album_id": 456,
    "play_count": 54321,
    "last_played_at": "2024-05-09T10:00:00Z",
    "genre": "Pop",
    "song_path": "https://example.com/song/1"
  }
}
```

5 Database Schema

We need these four tables to store data the `Songs` will keep all data related to a song and then there are two other tables the Album and the Artist which respectively maintain data of the artist and the album of the song. Lastly, we had the Top Song table which contains top ranking songs for us.



A straightforward approach for storing the ranking table schema would be to use a NoSQL database like Cassandra since we require high write throughput and scalability. However, NoSQL databases come with challenges, especially when handling complex queries involving multiple tables. For details, please refer to the SQL vs. NoSQL chapter. We can store song files in a distributed file storage like Amazon S3.

6. Core Challenges

In designing Spotify's Top Songs feature, we face two primary challenges that are critical to the system's effectiveness and user satisfaction:

- **Calculating the Top K List:** Developing an efficient method to dynamically determine the most popular songs based on real-time user data.
- **Managing Time Windows:** Ensuring the top K list can adapt to various timeframes, such as hourly, daily, weekly, monthly, or all-time rankings, to reflect the changing preferences of users.

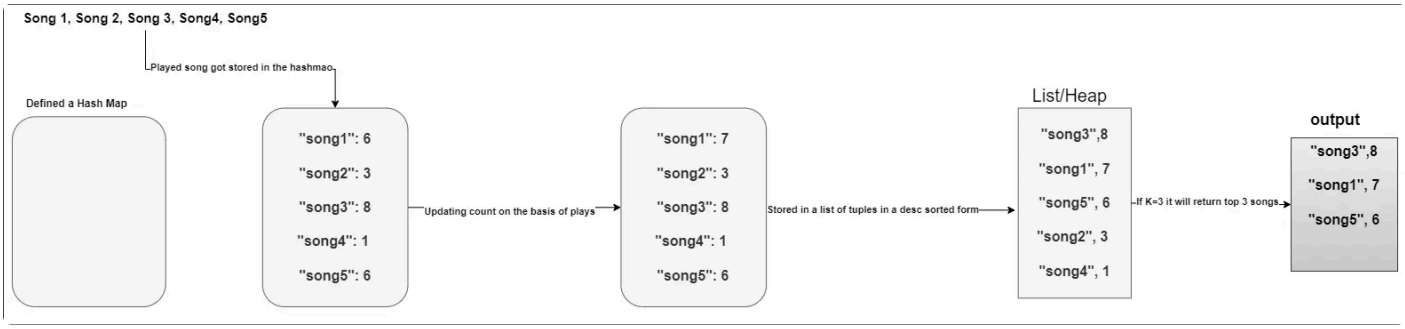
Here's a detailed look at the strategies and their considerations for tackling these challenges one by one:

7. Calculating the Top K List

Before moving toward the details it's important to identify what options we have to handle the top k list calculation and how they work, to make an efficient choice for our system. Let's start by going through all the approaches one by one and also explore their strengths and weaknesses as well.

Approach 1: Hash Map + Sorting

A naive but precise approach uses a hash map to count song plays and data structures like list or heap to store the sorted ranking Let's understand it with visuals.



Start by creating an empty hash map to store the play counts of each song.

As songs are played, add them to the hash map with their play counts. For example, as shown in the illustration, songs like "song1", "song2", "song3", "song4", and "song5" are added to the hash map with their respective counts.

When a song is played again, simply increment its count in the hash map. In the illustration, "song1" is played again, so its count updates from 6 to 7.

Next, convert the hash map entries into a list of tuples and sort them by play count in descending order. As shown in the illustration, the sorted list looks like this: ("song3", 8), ("song1", 7), ("song5", 6), ("song2", 3), ("song4", 1).

Finally, to get the top K songs, access the first K elements from the sorted list. For example, if K=3, the top three songs are "song3", "song1", and "song5", as shown in the output section of the illustration. This method can become inefficient with a large number of songs due to the sorting step.

This approach has its benefits. It provides precise play counts for each song and is easy to implement with basic data structures. However, there are drawbacks. Sorting takes $O(n \log n)$ time for tuples and $O(k \log n)$ for a heap, making it inefficient for a large number of songs and impacting latency. The hashmap consumes significant memory, and the sorting operation lacks high availability, scalability, and fault tolerance, making it unsuitable for large-scale, real-time systems.

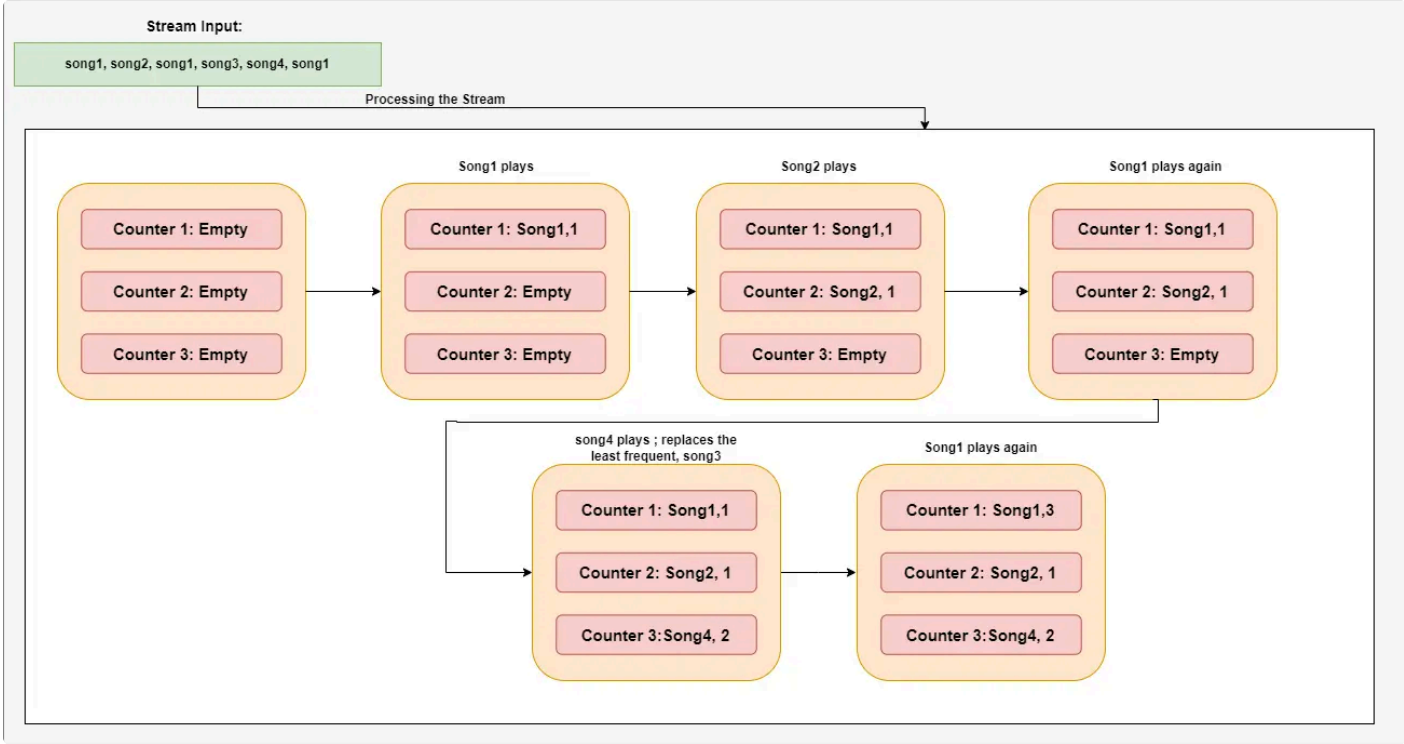
Let's move on to the next approach.

Approach 2: Space-Saving Algorithm

Transitioning from the Hash Map + Sorting method, which is straightforward but not efficient for large-scale or continuous data, let's explore the Space-Saving Algorithm. The Space-Saving Algorithm is designed for scenarios where memory resources are constrained, making it ideal for real-time applications like music streaming services. Here's how it works:

- **Empty Counters:** If a counter is empty, the incoming song is stored there, and its count is initialized at one.
- **Increment Existing Counters:** If a counter is already tracking a song, its count is incremented with each new occurrence.
- **Replace Least Frequent Song:** If all counters are full and the incoming song isn't tracked, it replaces the song with the lowest count. The new song's count is set to the evicted song's count plus one.

Now let's understand it further with an example below:



Start by initializing a fixed number of counters. For simplicity, let's assume we have three counters, as shown in the illustration.

As songs are played, store them in the counters. For example, when "song1" plays, it is stored in the first counter with a count of 1. When "song2" plays, it goes into the second counter with a count of 1. If "song1" plays again, its count in the first counter increases to 2.

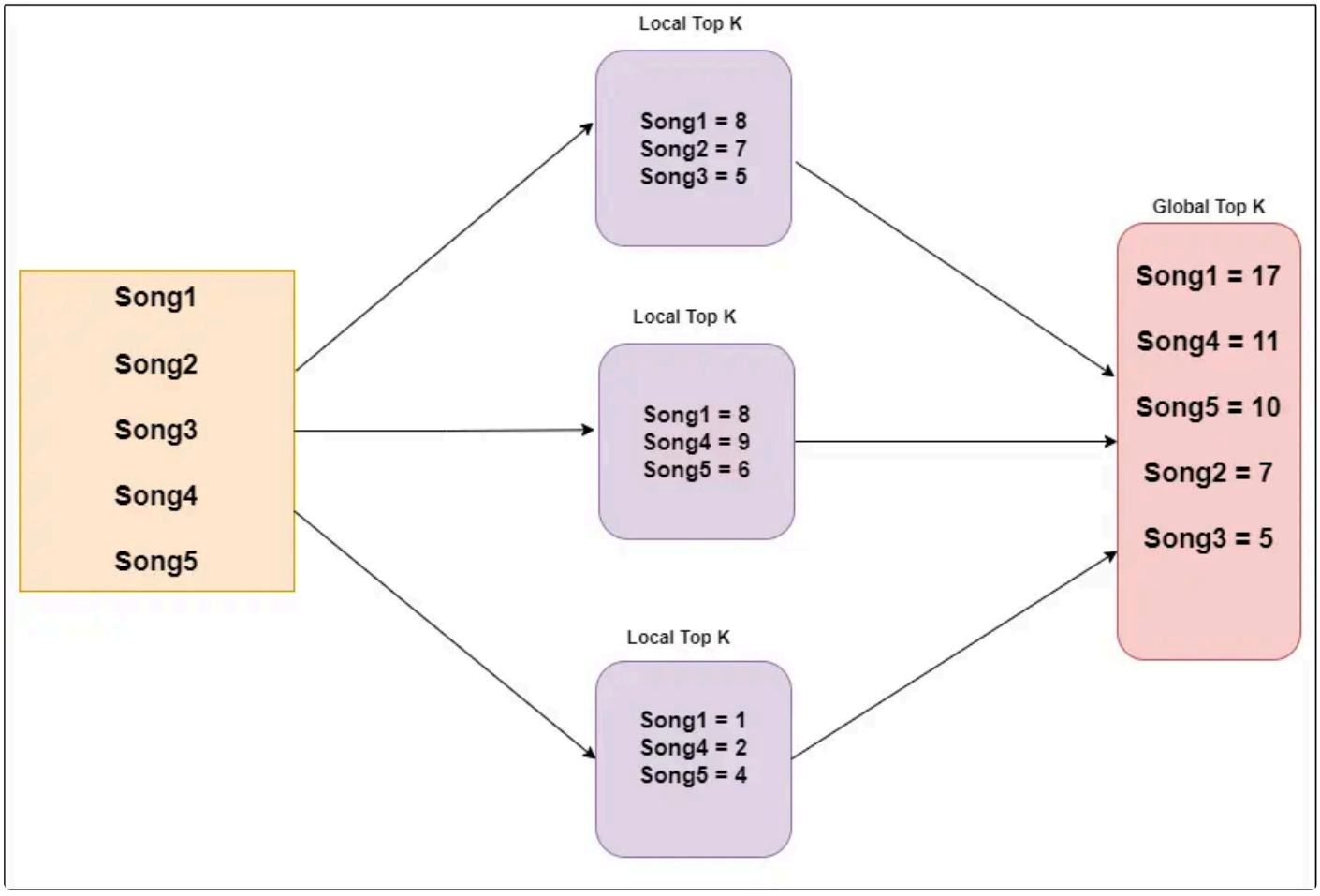
If a new song plays and all counters are full, the new song replaces the song with the lowest count. For instance, in the illustration, when "song4" plays and all counters are full, it replaces "song3", which has the lowest count, and its count is set to the replaced song's count plus one, making it 2.

This process ensures that the counters always track the most frequently played songs. As songs continue to play, their counts are updated or they replace the least frequent songs as needed.

This approach has its benefits. It is efficient within a fixed memory space and dynamically adjusts to changes in song play frequencies, making it suitable for real-time applications. However, there are drawbacks. The fixed number of counters can limit flexibility and scalability, and the method may not track all songs accurately if the variety of songs played is very high.

Approach 3: Distributed Systems (MapReduce)

Transitioning from the Space-Saving Algorithm, which efficiently handles scenarios with fixed memory limitations, we now explore Distributed Systems, specifically the MapReduce model. This approach is designed for large-scale data processing across multiple machines, making it suitable for environments with very large datasets or high availability requirements. Here's how MapReduce operates for solving the top-k problem:



The diagram illustrates the MapReduce approach for computing the top-k problem in a distributed system. Each node calculates its local top-k list from its data segment, which includes songs and their respective play counts. These local lists are then aggregated by a reducer to form the global top-k list. For example, play counts for songs like Song1, Song4, and Song5 are summed across nodes, reflecting their overall popularity in the global list. This demonstrates how MapReduce effectively handles large-scale distributed data processing.

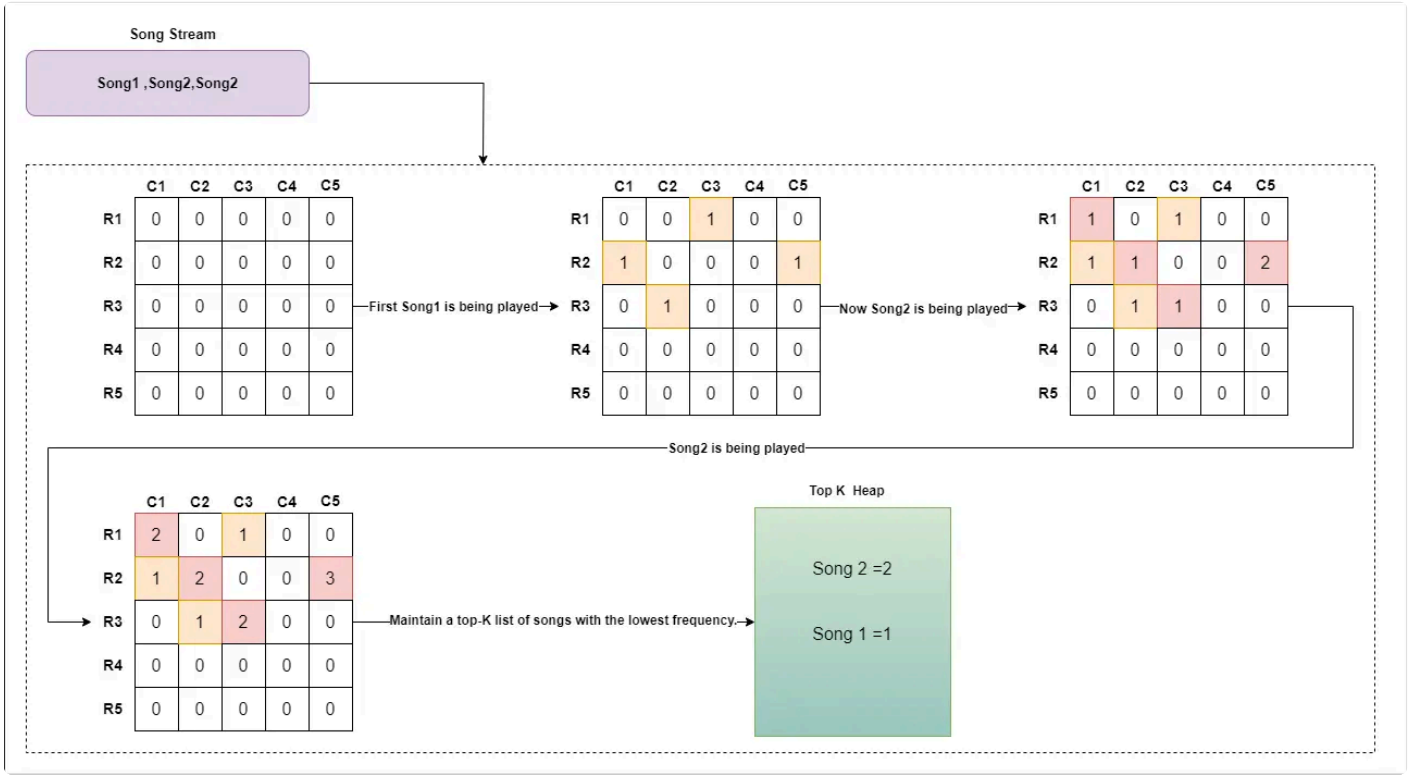
The MapReduce approach for Spotify's "Top Songs" feature efficiently manages large-scale data across distributed nodes, ensuring consistent song rankings and real-time updates. It supports high scalability and maintains a 99.99% uptime, essential for robust fault tolerance and quick response times. However, while MapReduce is powerful, it may not always respond quickly due to the time required for aggregating data across multiple nodes. The complexity and resource demands of this system can also be challenging.

Next, we will explore the Count-Min Sketch (CMS) approach, which offers a different set of advantages.

Approach 4: Count-Min-Sketch approach

The Count-Min Sketch (CMS) is a probabilistic data structure that approximates the play counts of songs with minimal memory usage. Think of it as a 2D array (matrix) where each row is associated with a different hash function. When an item (like a song) is added, its identifier is hashed using several hash functions, and the corresponding counters in each row of the matrix are incremented. The minimum count across all rows provides an approximate but accurate count. In our system, the CMS helps track the play counts of songs.

Let's understand it further with the help of the illustration:



CMS Approach

Start by initializing an empty matrix of counters. Each row corresponds to a different hash function, and each column is a counter initialized to zero.

As songs are played, they are hashed into specific counters across multiple rows. For instance, when "song1" plays, it is hashed into counters R1C2, R2C3, R3C1,and R2C5, and each of these counters is incremented by 1. The matrix updates to show the first play of "song1".

Next, "song2" plays, and it is hashed into counters R1C4, R2C1, R3C2, and R2C5. All these counters are incremented by 1. The R2C5 counter, which was already incremented by "song1," now updates to 2. The matrix reflects these updates for "song2."

When "song2" plays again, it hashes to the same counters (R1C4, R2C1, R3C3, and R2C5), and these counters are incremented again, showing the second play of "song2".

To estimate the frequency of each song, find the minimum value across all the counters it is hashed to. For example, the minimum count for "song2" across its hashed positions (R1C4, R2C1, R3C3, and R2C5) gives its approximate play count. The same applies to "song1".

Finally, use the CMS matrix to maintain a top-K list of songs with the highest frequency. To do this, implement a min-heap to store the top-K songs. The min-heap allows efficient updates and retrievals. When a new song is played, calculate its estimated frequency using the CMS. Insert the new song into the heap if it has a higher frequency than the current minimum in the heap. Continuously update the heap as songs are played to ensure it always contains the top-K songs.

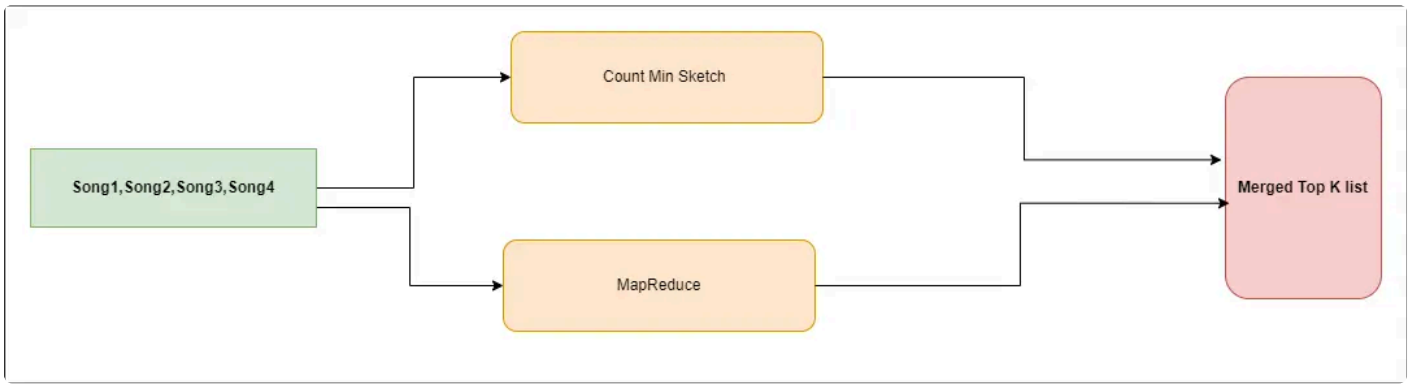
This approach is efficient within a fixed memory space and dynamically adjusts to changes in song play frequencies, making it suitable for large-scale, real-time data processing. However, it provides only approximate counts and lacks support for complex queries, requiring careful parameter tuning to balance accuracy and resource use.

Next, we'll explore a hybrid approach that combines CMS with MapReduce, offering additional capabilities for real-time data analysis.

Approach 5 : Combining Count-Min Sketch and MapReduce

In our prior discussions, we examined the individual capabilities and limitations of the Count-Min Sketch (CMS) and MapReduce approaches in song ranking applications. While MapReduce is well-suited for processing large datasets in a batch mode, ensuring comprehensive data analysis and historical accuracy, it falls short in providing real-time updates. On the other hand, CMS excels in delivering fast, approximate updates through its efficient handling of streaming data, though it sacrifices some accuracy and depth of analysis.

To leverage the strengths of both, we propose a strategy that integrates CMS and MapReduce into a cohesive system. This system is designed to ensure Spotify can offer up-to-date song rankings while also maintaining the historical integrity of the data analysis. Here's how it is structured:



This diagram depicts the use of both for updating Spotify's Top K song rankings. Incoming song data is split between two processing paths: Count Min Sketch for immediate, approximate updates, and MapReduce for detailed, historical analysis. Both paths process the data concurrently and their outputs directly contribute to forming the Merged Top K List, ensuring the song rankings are both timely and accurate.

Evaluation for Top K Ranking Calculation Approaches

Now that we have discussed different approaches to calculate top k songs let's evaluate and select one approach that we will use in our system.
So analyzing various approaches for calculating Spotify's Top K rankings reveals distinct strengths and limitations. Hash Map + Sorting offers precision but falls short on scalability, making it impractical for large systems like Spotify. The Space-Saving Algorithm provides a balance with faster updates but approximates counts. Distributed Systems (MapReduce) excel in managing large datasets with high availability and fault tolerance, albeit with some complexity and latency. Count-Min Sketch (CMS) prioritizes minimal memory usage and quick updates at the cost of accuracy and query complexity. The Hybrid (CMS + MapReduce) approach, combining real-time and robust data processing capabilities, emerges as the most appropriate for Spotify's requirements.

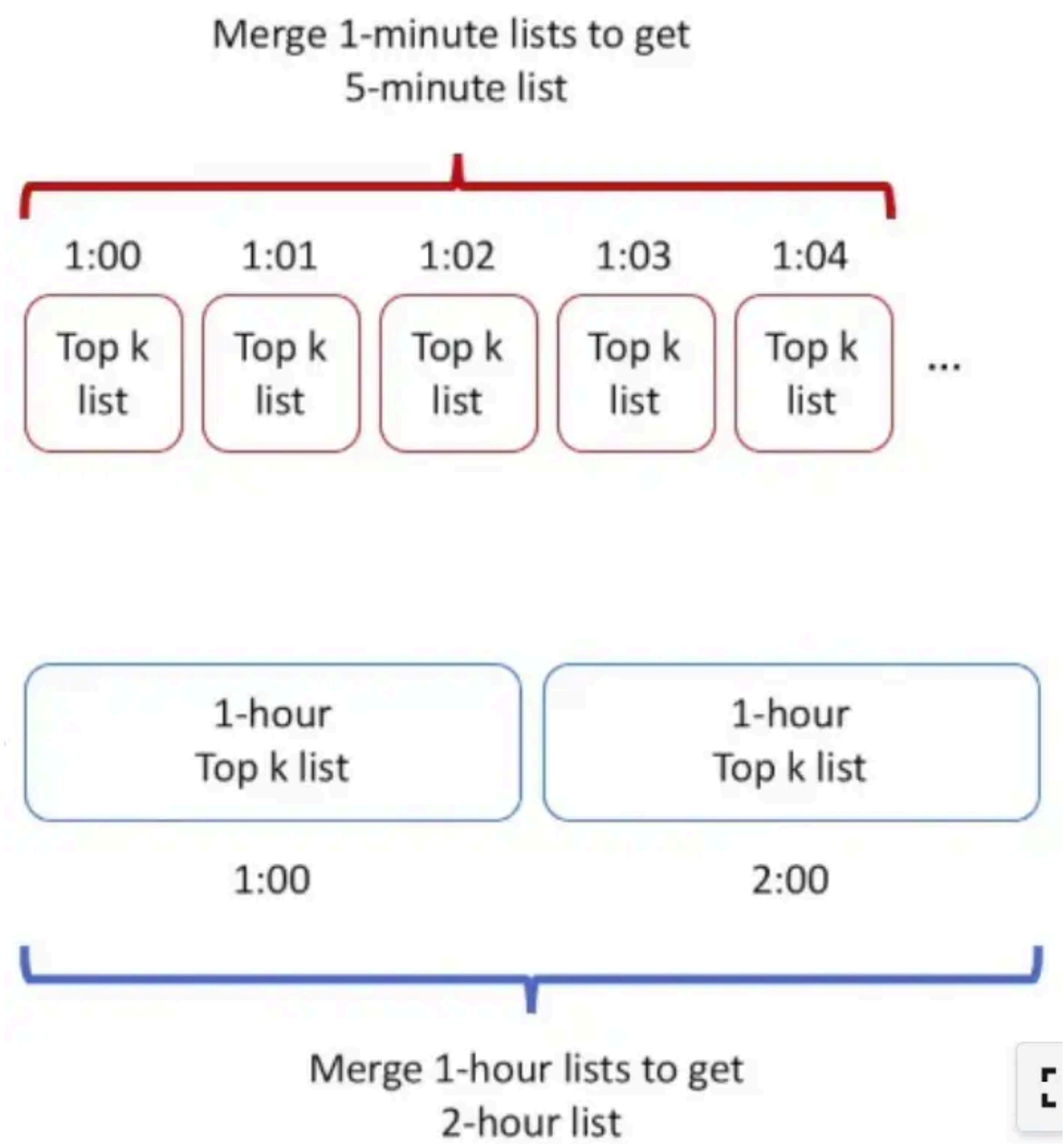
However, we are still missing one requirement for our system which is handling time windows of our system for managing ranking for different timeframes like `hour`, `day`, `week`, `month` or `All time`. So let's select a strategy for it before designing our system.

8. Managing Time Windows:

In Spotify's system, managing different time windows (like daily, weekly, and monthly) efficiently ensures that the most popular songs are accurately highlighted according to recent trends. Here's how we integrate time window management with our data processing technologies, Count-Min Sketch (CMS) and MapReduce, to balance real-time responsiveness with accurate historical data analysis.

Time-Based Partitioning with MapReduce

Time-based partitioning organize data into separate segments based on time intervals—daily, weekly, or monthly. This method, known as Time-Based Partitioning, allows for more efficient data processing because only relevant data segments are accessed for each query. Using MapReduce, this approach can handle big data across multiple servers, making it possible to run complex analytics on historical trends and generate detailed reports. This is particularly useful for creating features like "Monthly Top Hits," where understanding long-term trends is key.

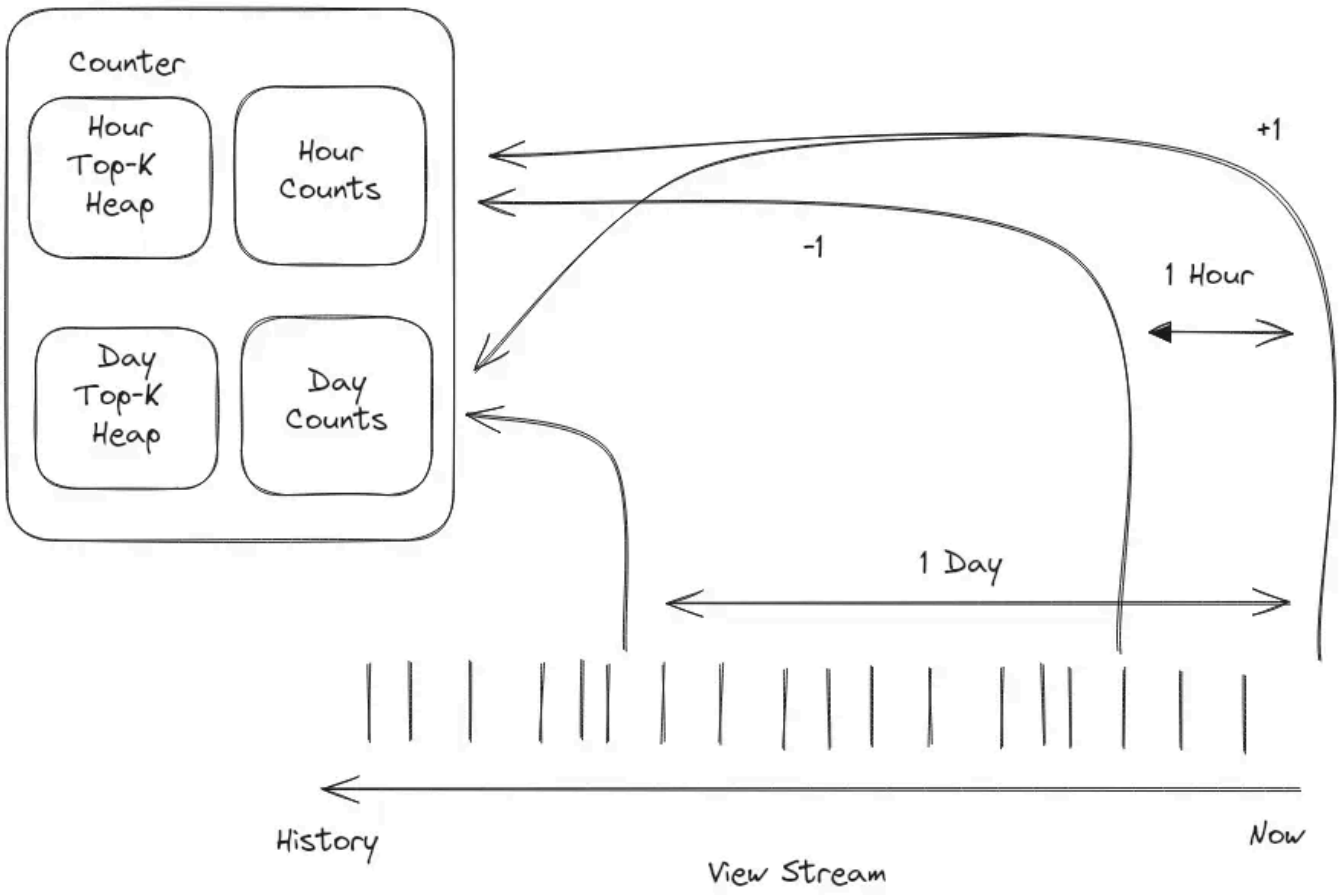


The image shows a data aggregation system where top item lists are compiled every minute and then merged into larger time windows, like 5-minute and 1-hour lists. This method, used in platforms like Spotify, helps efficiently manage and analyze trends over different periods. By merging these short intervals into longer ones, the system balances immediate responsiveness with long-term trend analysis, making it ideal for features that track popular items over time.

Sliding Window Technique with Count-Min Sketch

The Sliding Window Technique is a powerful method for managing and analyzing streams of data in real-time. It continuously updates a "window" of recent data by adding new information and removing old data as time progresses. This approach is highly adaptable, allowing for precise tracking across different time frames such as minutes, hours, or days. By focusing only on the most recent data within the defined window, the Sliding Window Technique ensures that analytics remain relevant and up-to-date, making it ideal for applications where timely insights are crucial.

Combined with the Count-Min Sketch (CMS), which efficiently estimates item counts, this setup is perfect for real-time features like "What's Hot Right Now." The CMS tracks play counts within the sliding window, ensuring the data is always current and quickly processed. This dynamic duo adjusts to new trends without reprocessing the entire dataset, providing a robust solution for real-time popularity tracking.



The diagram illustrates the use of the Sliding Window Technique combined with the Count-Min Sketch (CMS) in real-time data management. It shows a system where data is constantly updated by adding new information and removing outdated data at set intervals—hourly and daily. This setup includes separate counters and top-k heaps for both hourly and daily data, ensuring that the system efficiently manages and tracks the most current trends for real-time features, such as "What's Hot Right Now," without needing to reprocess the entire dataset. now there is another question that can be pointed that:

Why we are using two different approaches to manage time windows?

We are using two different approaches to manage time windows to optimize both historical data analysis and real-time trend monitoring:

Time-Based Partitioning with MapReduce is used for analyzing historical trends over set periods (daily, weekly, monthly). This method helps in efficiently processing large datasets across multiple servers, crucial for features like "Monthly Top Hits" that require understanding long-term trends.

Sliding Window Technique with Count-Min Sketch (CMS) is designed for real-time data processing, where data is continuously updated to quickly reflect current listening trends. This approach is essential for dynamic features like "What's Hot Right Now," allowing the system to adjust to new data without reprocessing the entire dataset.

As we got the answer for our first question there is another question that can come to mind :

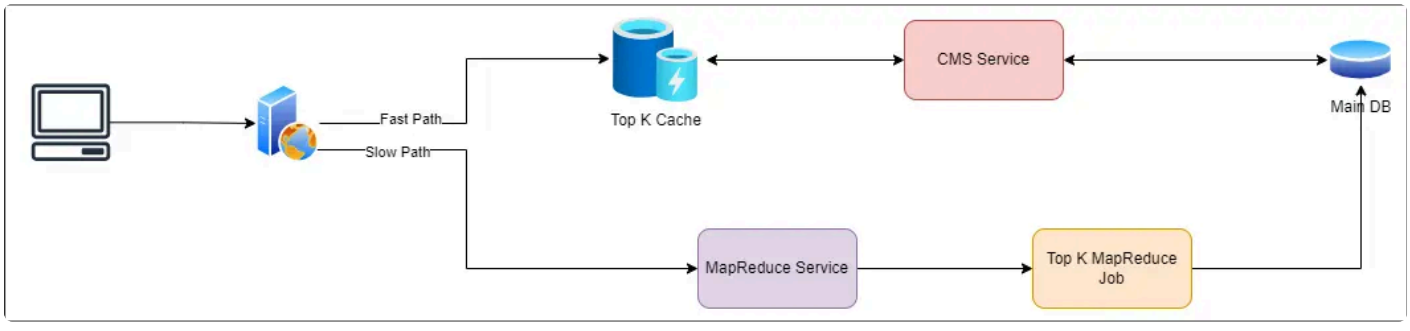
How does the Sliding Window Technique respond to a request for a one-hour top song list made at 12:30?

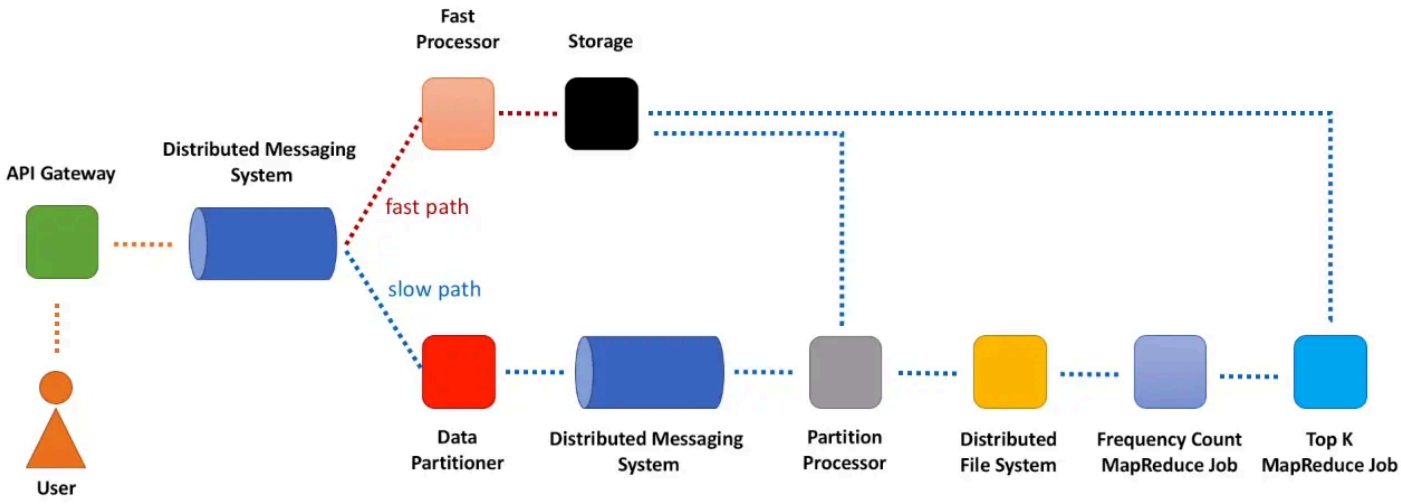
If a user requests the top songs for the last hour at 12:30, the Sliding Window Technique, integral to our real-time data processing system, provides the list from 11:30 to 12:30. This technique updates continuously, ensuring that at any given request time—whether on the hour or halfway through—the data reflects the most recent one-hour window. This method enables the system to adapt swiftly and accurately to changing data, offering up-to-date insights directly aligned with the latest user interactions and trends.

Now that we have solved both challenges that we had in our system. Let's see how the system would work.

9 High-level design for Spotify Top K

Considering all factors, here's a review of our high-level design, which leverages the principles of Lambda architecture. This approach effectively handles real-time data processing through the fast path while also managing extensive batch processing for deep analytics via the slow path. The integration of both paths allows for the seamless provision of both immediate updates and long-term insights, ensuring a robust and scalable solution for managing song popularity data.





As we can see in the above illustration we have two paths one is the fast path and the other one is the slow path let's understand their process one by one.

Fast Path The fast path is designed for real-time processing, ensuring immediate responsiveness to user interactions. As users engage with the system, such as playing a song, data is sent through an API Gateway that routes these requests to the Count-Min Sketch (CMS) Service. This service quickly updates play counts and other relevant metrics, storing them in a Top K Cache. This cache provides rapid access to the most frequently requested data, offering users up-to-the-minute updates on the most popular songs.

CONTRIBUTOR
Design Gurus Team

Slow Path The slow path focuses on batch processing for comprehensive data analysis and generating historical insights. Data collected in the CMS Service is transferred to the MapReduce Service, where it undergoes extensive processing to compute detailed analytics, such as song popularity trends over time. The results are then stored in a main database, which supports complex queries and reports, helping the system refine recommendations and strategic decisions based on long-term user behavior patterns.

ensure data availability and durability. In the event of a data node failure, the system can quickly switch to a replica without impacting data access. Regular backups and snapshotting are also part of the data management strategy to facilitate quick recovery from data corruption or loss.

Explore Answers

What is constructor in OOPs?

How to prepare for Palantir interview?

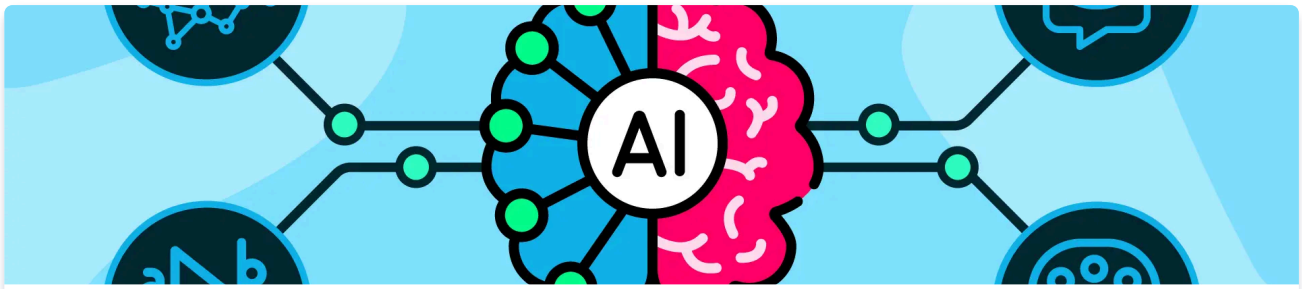
Structuring design proposals with modular component diagrams

Related Courses

Grokking the Coding Interview: Patterns for Coding Questions

Grokking the Coding Interview Patterns in Java, Python, JS, C++, C#, and Go. The most comprehensive course with 476 Lessons.

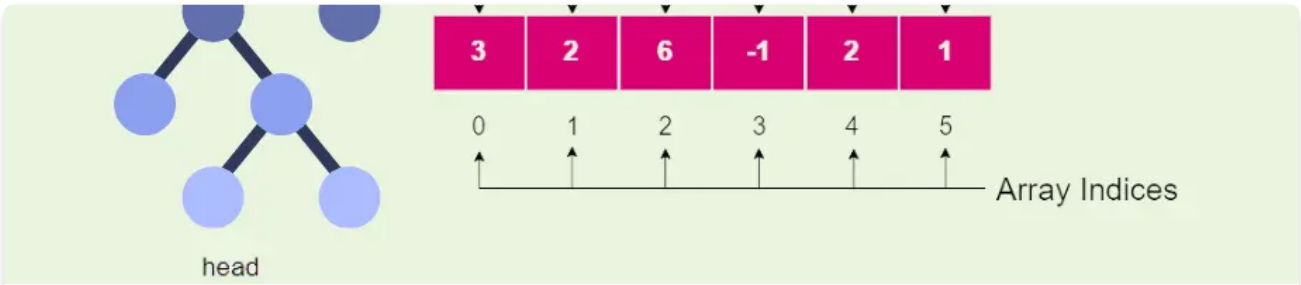
Preview



Grokking Modern AI Fundamentals

Master the fundamentals of AI today to lead the tech revolution of tomorrow.

Preview



Grokking Data Structures & Algorithms for Coding Interviews

Unlock Coding Interview Success: Dive Deep into Data Structures and Algorithms.

Preview