**Technical Design Document: Airport System Monitoring Dashboard**

**1. Overview**

The Airport System Monitoring Dashboard is a terminal-based application designed to provide a real-time, high-level overview of critical airport operations. The system aggregates continuous data streams related to flight delays, baggage handling, and flight connection statuses. The primary goal is to present a live, ranked list of the most pressing issues, enabling operations staff to make quick, data-driven decisions.

This document outlines the engineering decisions, data structures, and algorithms used to build the performant data aggregation and ranking layer of this system.

**2. System Architecture & Design**

The system is designed around a simple, robust data flow that prioritizes performance at the aggregation and ranking stages.

**2.1. Component Diagram**

Code snippet

graph TD

A[Data Sources] -->|Events (Delay, Jam, etc.)| B(Data Aggregation Layer);

B -->|Updated Metrics| C{Ranking Engine};

C -->|Top 5 Lists| D[Dashboard Display];

subgraph Data Aggregation Layer

B(Hash Maps);

end

subgraph Ranking Engine

C(Heaps);

end

**2.2. Component Descriptions**

* **Data Sources**: Represents external systems that generate events, such as a 10-minute delay for a specific route or a new baggage jam report.
* **Data Aggregation Layer**: The core of the system. It uses hash maps to maintain a master record of all operational metrics. Its responsibility is to ingest raw events and update the state in O(1) time.
* **Ranking Engine**: This component is responsible for processing the data from the aggregation layer to find the most critical items (e.g., "Top 5 most delayed routes"). It uses a heap-based selection algorithm for maximum efficiency.
* **Dashboard Display**: The user-facing component that polls the Ranking Engine at a regular interval and renders the ranked lists in the terminal.

**3. Data Structure Selection & Trade-Offs**

The choice of data structures is critical for meeting the real-time performance requirements of the dashboard.

**3.1. Hash Maps (Python dict)**

* **Usage**: The primary data structure for the **Data Aggregation Layer**. We use three distinct hash maps: route\_delays, baggage\_jams, and flights\_at\_risk.
* **Justification**: Hash maps provide an average time complexity of **O(1)** for insertions, deletions, and lookups. When a new event arrives (e.g., a delay for flight UA456), we can instantly access and update its corresponding value. This constant-time performance is essential for a high-throughput system that cannot slow down as the number of events grows.
* **Trade-offs**:
  + **Pro**: Unbeatable average-case speed for key-based access.
  + **Con**: Higher memory footprint compared to a more compact structure like a list of tuples. The elements are inherently unordered, but this is acceptable as ranking is handled by a separate component (the heap).

**3.2. Heaps (Python heapq module)**

* **Usage**: The core data structure for the **Ranking Engine**. We use a fixed-size heap to determine the "Top N" items from the master hash maps.
* **Justification**: A heap is a specialized tree-based structure ideal for maintaining a list of the "largest" or "smallest" items. Using a heap allows us to find the Top N items in O(n log k) time, which is significantly more performant than the alternative.
* **Trade-offs**:
  + **Pro**: Excellent time complexity for finding Top N items, especially when k is much smaller than n. Extremely memory-efficient, as it only ever stores k items.
  + **Con**: More complex to implement and reason about than a simple sorting approach. The standard heapq library in Python is a min-heap, requiring a workaround (storing negative values) to simulate a max-heap for finding the smallest items.

**4. Core Functions & Complexity Analysis**

The following functions are critical to the system's performance. n refers to the total number of items in the source dictionary, and k refers to the number of top items to find (e.g., 5).

**4.1. get\_top\_largest(data\_dict, n)**

This function finds the k items with the largest values in a dictionary.

* **Time Complexity**: **O(n log k)**.
  + The initial heapify call on the first k items takes O(k).
  + The loop runs n-k times. Each heapreplace operation within the loop takes O(log k).
  + The final complexity is O(k + (n-k) \* log k), which simplifies to O(n log k).
* **Space Complexity**: **O(k)**.
  + The function creates a heap that stores, at most, k items. Memory usage does not scale with n, making this approach highly memory-efficient.

**4.2. get\_top\_smallest(data\_dict, n)**

This function finds the k items with the smallest values. It uses the same heap logic but with negative numbers to simulate a max-heap.

* **Time Complexity**: **O(n log k)**. The complexity is identical to get\_top\_largest.
* **Space Complexity**: **O(k)**. The space requirement is also identical.

**5. Algorithm Justification: Heap Selection vs. Full Sort**

A critical algorithmic decision was how to derive the "Top N" lists from the aggregated data.

* **Chosen Algorithm**: **Heap-based Selection**. We iterate through the n items once, maintaining a fixed-size heap of size k to keep track of the top candidates.
* **Alternative Considered**: **Full Sort**. This approach involves converting the dictionary items to a list (O(n)), sorting the entire list (O(n log n)), and then taking the top k items (O(k)). The total complexity is dominated by the sort, resulting in O(n log n).
* **Justification**: While a full sort is easier to write (sorted(list)[:k]), its O(n log n) performance is unacceptable for a large-scale, real-time system. The heap-based selection algorithm's O(n log k) complexity is far superior. As n (the total number of flights or routes) grows into the thousands or millions, the performance difference becomes dramatic. The heap method aligns with our primary goal of **performance-aware engineering**.

**6. Pseudocode: Fixed-Size Heap Algorithm**

The following pseudocode outlines the logic for get\_top\_largest.

FUNCTION get\_top\_largest(data\_dictionary, k):

// Create a list of all items from the dictionary

all\_items = data\_dictionary.items()

// Initialize a min-heap with the first k items

heap = create\_heap\_from(all\_items[:k])

// Loop through the rest of the items

FOR item in all\_items[k:]:

// If the current item is larger than the smallest item in our heap

IF item.value > heap.get\_min().value:

// Replace the smallest item with the current item

heap.replace\_min(item)

END IF

END FOR

// Sort the final heap for display and return it

RETURN sort\_descending(heap)

END FUNCTION

**7. Performance Benchmarks (Hypothetical)**

To illustrate the performance difference between Heap Selection and Full Sort, the following table presents hypothetical execution times as the dataset size (n) increases, with k=50.

| Total Items (n) | Heap Selection O(n log k) | Full Sort O(n log n) | Performance Gain |
| --- | --- | --- | --- |
| 1,000 | 0.001 s | 0.005 s | 5x |
| 100,000 | 0.1 s | 1.1 s | 11x |
| 1,000,000 | 1 s | 15 s | 15x |
| 10,000,000 | 10 s | ~3 minutes | 18x |

As the data scales, the Heap Selection algorithm's runtime grows linearly, while the Full Sort method's runtime grows super-linearly, demonstrating the clear advantage of our chosen algorithm.