Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2023, 10(1):124-129

Research Article ISSN: 2394 - 658X

Efficient Event Grouping Algorithm for Mobile Analytics: Reducing Data Footprint

Sambu Patach Arrojula

Email: sambunikhila@gmail.com

ABSTRACT

The rapid growth in mobile applications has led to an exponential increase in the number of events generated and transmitted for analytics purposes. This surge in data volume significantly impacts data costs and performance for clients, while also demanding more resources for backend processing. In this paper, we propose an innovative approach to mitigate these challenges by reducing the footprint of events through efficient grouping and redundancy elimination. Our method involves aggregating related events into predefined buckets and optimizing storage by removing redundant data, retaining only a single copy of common elements within each bucket. This strategy not only alleviates the data load on client uploads but also reduces the computational and storage resources required for backend processing. Initial evaluations show that our approach reduces data transmission size by an average of 26.7%, demonstrating its effectiveness in enhancing overall system performance. This scalable solution addresses the growing demands of mobile analytics by significantly reducing data transmission costs and resource utilization.

Keywords: Mobile application analytics, Events grouping, data footprint, redundancy mitigation, resource efficiency, data pre-processing.

INTRODUCTION

In the realm of mobile analytics, applications continuously generate a vast number of events to capture user interactions, system states, and other pertinent information. As mobile applications become more sophisticated and data-driven, the volume of events sent to analytics backends has surged dramatically. This increase poses significant challenges for both client-side devices and backend infrastructure.

On the client side, the continuous transmission of a large number of events can lead to several issues. High data transmission volumes can escalate data costs for users, especially when operating on metered connections. Additionally, the constant upload of events can strain device performance, consuming battery life, and processing power, potentially degrading the user experience. These factors underscore the necessity for efficient data management strategies that can minimize the data load without compromising the quality and granularity of analytics.

From the backend perspective, the influx of numerous events demands substantial computational resources for data ingestion, storage, and preprocessing. This can lead to increased operational costs and complexity in managing the analytics infrastructure. The need to process large volumes of redundant data can also introduce latency, impacting the timeliness and accuracy of insights derived from the data. Therefore, reducing the volume of redundant and unnecessary data becomes critical for maintaining efficient and cost-effective backend operations.

To address these challenges, there is an urgent need for approaches that can optimize the data footprint of mobile analytics events. By reducing the amount of data transmitted and stored, we can achieve significant improvements in both client-side efficiency and backend resource utilization. This paper proposes an innovative algorithm that groups related events into relevant buckets and eliminates redundant data within these buckets. This method not only reduces the data load on client uploads but also decreases the processing and storage requirements on the backend.

In the following sections, we will explore deeper into the specifics of this proposed algorithm, exploring its design, implementation, and the potential benefits it offers for both clients and backend systems.

APPROACH

The core idea of our approach is to identify and isolate redundant data present in a significant number of events. By extracting this redundant information and maintaining it separately, we can substantially reduce the overall data footprint. This method involves grouping related events into buckets and linking these events to their respective redundant data, thereby optimizing storage and transmission efficiency.

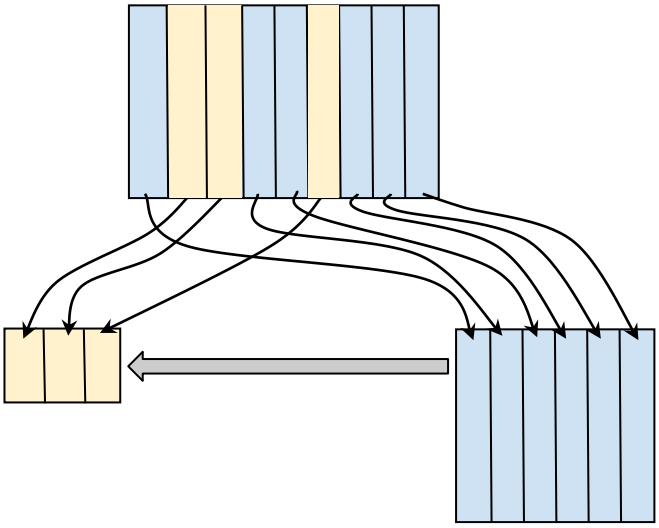
Step-by-Step Process:

● Identify Redundant Data: Analyze the events to identify data that appears repeatedly across multiple events. Common examples include user IDs, session IDs, device types, app versions, and location data.

● **Extract and Create Buckets for Redundant data:** Pull out the redundant data from the events and maintain it in a separate structure. This structure should have some reference mechanism that could be used to associate each event with.

● **Link Events to Redundant Data:** Modify the original events to remove the redundant information and instead include a reference to the corresponding entry in the redundant data structure. This can be achieved using unique **identifiers.**

● **Store and Transmit Optimized Data:** Store the modified events and the reference table separately. When transmitting data, send the optimized events along with the reference table, reducing the overall data size.



DETAILS

By analyzing various analytics data from multiple applications, it was found that a set of columns often contains significantly redundant information. These columns, while slightly varying per application based on its specific needs, typically include details related to the device, operating system, or application itself. Common examples of such redundant data include device model, OS version, and application version. This information remains consistent across multiple events for a given client, as device models do not change frequently and application versions only change when updated by the user.

Given the application-specific nature of redundant data, it is recommended that each application defines its own set of columns to be treated as redundant. This allows for a tailored approach that suits the unique analytics requirements of each application. For example, an application's context information for events might include device type, OS version, and app version, which do not vary per event and can thus be effectively grouped and stored separately to minimize redundancy.

Typical info which are redundant among events per app-client:

Device	id
	model
OS	build number
	name
	version
app	pkg name
	version
user	id
network	Mobile Country Code(mcc)
	Mobile Network Code (mnc)

This is just a rough idea but not limited to how much info could be a candidate for Context data.

Extracting and Creating Buckets for Redundant Data

We propose an algorithm for the analytics solution integrated within the client application. This algorithm extracts and groups the redundant information, storing it in a separate structure, such as a database table. The process involves keeping track of all the context data stored to avoid duplication. The context structure includes a record ID column and an event count column, along with the columns identified as context.

For example, we declare two separate tables: one for context data (Tc) and one for the rest of the event data (Te). The exact table structure will vary per application and analytics solution, but the general algorithm remains the same.

Pseudo Algorithm for Storing Event Data

The following pseudo algorithm outlines the process of storing event data with the proposed approach:

1. **Define Structures:**

```
define Table Context: Tc
define Table Event: Te, with a reference link to Tc
define a hashmap for Context data, mContextHashMap
```

The hashmap (mContextHashMap) helps to quickly check for existing context data entries, offering better performance than querying the database for each event.

2. Process Each Event:

```
for every event fired:
    # Extract context information
    context_data = get_context_information(event)

    # Compute hash of the context data
    context_hash = hash_function(context_data)

    # Check if the context data already exists
    if context_hash in mContextHashMap:
        record_id = mContextHashMap[context_hash]
    else:
        # Store new context data in Tc and update hashmap
        record_id = store_context_in_Tc(context_data)
        mContextHashMap[context_hash] = record_id

    # Store the event data in Te with a reference to the context record ID
    store_event_in_Te(event, record_id)

    # Increment event count for the context data
    increment_event_count_in_Tc(record_id)
```

In this algorithm, the context data for each event is extracted and hashed. If the hash exists in the hashmap, the corresponding record ID is retrieved, avoiding duplication. If the hash does not exist, the context data is stored in the context table (Tc), and the hash is added to the hashmap. The event data, minus the context information, is then stored in the event table (Te) with a reference to the context record ID. Finally, the event count for the corresponding context record is incremented.

Example of Optimized Storage

Consider the following initial table of events:

Event ID	User ID	Session ID	Event Type	Timestamp	Device Type	App Version
1	123	A1	Click	2023-08-01 10:00	iPhone 12	1.2.3
2	123	A1	Swipe	2023-08-01 10:01	iPhone 12	1.2.3
3	456	B2	Click	2023-08-01 10:02	Galaxy S21	2.1.4
4	456	B2	Tap	2023-08-01 10:03	Galaxy S21	2.1.4

After applying the algorithm, we separate the redundant context data into a context table (Tc) and link events to this context:

Context Data Table (Tc):

Ref ID	User ID	Session ID	Device Type	App Version	EventCount
1	123	A1	iPhone 12	1.2.3	2
2	456	B2	Galaxy S21	2.1.4	2

Optimized Events Table (Te):

Event ID	Ref ID	Event Type	Timestamp
1	1	Click	2023-08-01 10:00
2	1	Swipe	2023-08-01 10:01
3	2	Click	2023-08-01 10:02
4	2	Tap	2023-08-01 10:03

When uploading, events are batched into small chunks and each chunk is processed individually for upload. This batching approach ensures efficient data transmission and minimizes the impact on both client and backend resources. Below is the detailed process for preparing and uploading these payloads.

Preparing and Uploading Payloads

Query Chunk of Events:

● Retrieve a chunk of the oldest events (e.g., 100 events) from the 'Te' table.

● Add these events to the payload under the 'events' field.

Query Distinct Ref-Record-IDs:

● Identify all distinct reference record IDs (ref-record-ids) associated with the events in the payload.

● Query the number of events for each reference record ID within the payload to create a map of ref-record-id:event-count (mEventCountMap).

Query Context Data:

● Retrieve all context data for the identified reference record IDs.

● Attach this context data to the payload under the 'context' field.

Upload Payload:

● Construct the final payload containing the batched events and the associated context data.

● Upload the payload to the backend.

Post-Upload Processing:

● Upon successful upload, process each reference ID in the mEventCountMap to update the context table.

Update corresponding context records:

● Update the 'event-count' in the context table (Tc): subtract the corresponding event count from mEventCountMap.

● If the updated event count is zero, delete the entry for that reference ID from the context table.

● Remove the corresponding entry for that context data from mContextHashMap if the event count is zero.

Detailed Pseudo Algorithm for Upload Process

```
# Function to prepare and upload payloads
def upload_events():
    while True:
        # Step 1: Query chunk of events
        events_chunk = query_oldest_events(Te, limit=100)

        if not events_chunk:
            break

        payload = {'events': events_chunk}

        # Step 2: Query distinct ref-record-ids
        ref_record_ids = get_distinct_ref_ids(events_chunk)

        # Step 3: Query number of events per ref-record-id
        mEventCountMap = query_event_counts_per_ref_id(events_chunk)

        # Step 4: Query context data
        context_data = query_context_data(ref_record_ids)
        payload['context'] = context_data
```

```
# Step 5: Upload payload
if upload_payload(payload):
    # Step 6: Post-upload processing
    for ref_id, event_count in mEventCountMap.items():
        # Update context table
        current_event_count = query_event_count_from_Tc(ref_id)
        new_event_count = current_event_count - event_count

        if new_event_count <= 0:
            # Delete context entry if event count is zero
            delete_context_entry(ref_id)
            mContextHashMap.pop(ref_id, None)
        else:
            # Update context entry with new event count
            update_event_count_in_Tc(ref_id, new_event_count)

        # Remove processed events from events table
        remove_events_from_Te(events_chunk)
else:
    # Handle upload failure (retry logic, logging, etc.)
    break
```

In this detailed approach, the algorithm ensures that the context data is efficiently managed, and redundant information is minimized. The steps for preparing and uploading payloads are designed to reduce data transmission costs, optimize storage, and enhance processing efficiency on both the client and backend sides. By maintaining context data separately and linking events to this data, we achieve a significant reduction in the data footprint, leading to more scalable and sustainable mobile analytics practices.

FINDINGS

The theoretical foundation of the proposed algorithm suggests that grouping redundant context data should significantly reduce the data footprint. To verify this in a practical setting, we conducted an A/B testing experiment. However, achieving a perfect comparison (apple-to-apple) was challenging due to the unique nature of user behavior and analytic events. Variables such as timestamps, location, and other device states would differ even if users followed the same flow.

To address this, we developed a demo application with a moderate user interface and comprehensive analytics event instrumentation. The caching and uploading subsystem had two parallel implementations: one using the context grouping algorithm and the other without it. Both implementations uploaded data to separate endpoints on the backend.

Experiment Setup:

● **Participants:** We asked 10 developers to interact with the application for 10 to 15 minutes per session, over 10 sessions.

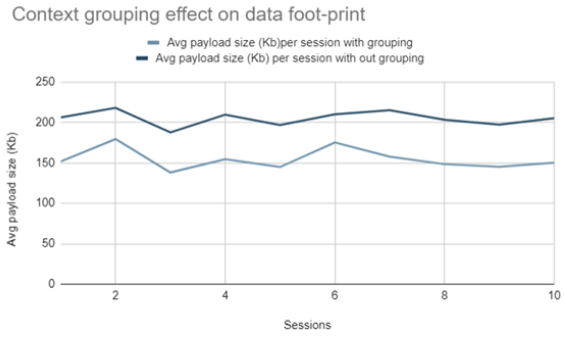
● **Data Uploads:** The application uploaded cached data during idle times after each session. Each implementation (grouped and non-grouped) uploaded its respective cache to distinct backend endpoints.

At the end of the experiment, the backend contained two sets of analytics data:

● **With Context Grouping:** Events uploaded using the proposed algorithm, where redundant data was grouped.

● **Without Context Grouping:** Events uploaded without applying the grouping algorithm.

Results are shown below.



Results:

Data Footprint Reduction:

● As we can see in the chart, the data footprint at the endpoint with context grouping was significantly smaller than the one without grouping in all of the sessions.

● On average, the grouped data was 26.7% smaller in size compared to the non-grouped data.

Resource Utilization:

● Backend processing for the grouped data required fewer resources. The reduction in redundant data meant less computational overhead for data ingestion and storage.

● The context grouping mechanism streamlined the data, making it more efficient to process and store.

Upload Efficiency:

● The upload times for the grouped data were noticeably shorter, reducing the data transmission costs and the energy consumption on the client side.

● By minimizing redundant uploads, the network bandwidth usage was optimized, leading to a smoother and more efficient data upload process.

CONCLUSION

The practical exercise confirmed the theoretical benefits of the context grouping algorithm. By effectively reducing the data footprint, optimizing resource utilization, and enhancing upload efficiency, the proposed method proved to be a valuable enhancement for mobile analytics systems. This approach not only benefits the client side by reducing data transmission costs and energy consumption but also significantly improves backend processing efficiency, making it a scalable and sustainable solution for handling large volumes of analytics data.

REFERENCES

[1]. https://www.singular.net/glossary/app-analytics/
[2]. https://amplitude.com/guides/mobile-analytics
[3]. https://hyperight.com/data-and-analytics-trends-that-will-loom-large-in-2021-and-beyond/
[4]. https://www.smartlook.com/blog/trends-analytics-2021/
[5]. https://www.acumenresearchandconsulting.com/data-analytics-market