Optimizing Storage Overhead of User Behavior Log for ML-embedded Mobile Apps

CHEN GONG, Shanghai Jiao Tong University, China
YAN ZHUANG, Shanghai Jiao Tong University, China
ZHENZHE ZHENG*, Shanghai Jiao Tong University, China
YILIU CHEN, ByteDance, China
SHENG WANG, ByteDance, China
FAN WU, Shanghai Jiao Tong University, China
GUIHAI CHEN, Shanghai Jiao Tong University, China

Machine learning (ML) models are increasingly integrated into modern mobile apps to enable personalized and intelligent services. These models typically rely on rich input features derived from historical user behaviors to capture user intents. However, as ML-driven services become more prevalent, recording necessary user behavior data imposes substantial storage cost on mobile apps, leading to lower system responsiveness and more app uninstalls. To address this storage bottleneck, we present AdaLog, a lightweight and adaptive system designed to improve the storage efficiency of user behavior log in ML-embedded mobile apps, without compromising model inference accuracy or latency. We identify two key inefficiencies in current industrial practices of user behavior log: (i) redundant logging of overlapping behavior data across different features and models, and (ii) sparse storage caused by storing behaviors with heterogeneous attribute descriptions in a single log file. To solve these issues, AdaLog first formulates the elimination of feature-level redundant data as a maximum weighted matching problem in hypergraphs, and proposes a hierarchical algorithm for efficient on-device deployment. Then, AdaLog employs a virtually hashed attribute design to distribute heterogeneous behaviors into a few log files with physically dense storage. Finally, to ensure scalability to dynamic user behavior patterns, AdaLog designs an incremental update mechanism to minimize the I/O operations needed for adapting outdated behavior log. We implement a prototype of AdaLog and deploy it into popular mobile apps in collaboration with our industry partner. Evaluations on real-world user data show that AdaLog reduces behavior log size by 19% to 44% with minimal system overhead (only 2 seconds latency and 15 MB memory usage), providing a more efficient data foundation for broader adoption of on-device ML.

CCS Concepts: • **Human-centered computing** → **Mobile computing**; Ubiquitous and mobile computing systems and tools; • **Computing methodologies** → **Machine learning**.

Additional Key Words and Phrases: On-Device Machine Learning; Resource-Efficient On-Device Model Inference; Storage Optimization; Mobile Application Log

---

*Zhenzhe Zheng is the corresponding author.

1 Introduction

The rapid evolution of smartphones has driven the widespread integration of machine learning (ML) models into modern mobile apps [3, 20, 22, 47, 67, 77], empowering a new era of context-aware and personalized app services. Representative examples include personalized product recommendations in e-commerce platforms [9, 10, 18, 27], search results ranking in search engines [33, 82, 83], as well as video preloading and bandwidth management in multimedia streaming apps [55, 71, 78]. Unlike traditional vision or language models that use *static* input features (*e.g.,* pixel values or token embeddings), ML models deployed in mobile apps rely on *dynamic* input features extracted from evolving user behaviors to capture dynamic contexts and user intents [39, 46, 63] (*e.g.,* the recent product clicks and video views could partially reflect a user's shifting preference).

To enable accurate feature computation for on-device model inferences, current mobile apps store all necessary user behavior data in a dedicated file known as the *user behavior log*, which acts as a critical middle layer to connect physical user behaviors and on-device ML models (§2.1). Specifically, every user interaction with the app's graphical user interface (GUI) can be captured as a structured *behavior event*, which consists of numerous different *attributes* to provide a rich, detailed description of each user behavior. When a behavior event occurs in real time, different model features apply their own *filters* to check if the current event is relevant, pick out necessary attribute subsets and store them as event rows in behavior log, which is typically implemented as SQLite databases for both Android and iOS devices [17, 37, 62]. To facilitate the fast retrieval of relevant events for feature computation, behavior log also maintains an index structure to map each feature to the physical storage addresses of its relevant event rows within the database.

Storage Bottleneck. Despite its essential role, behavior log introduces an overlooked storage bottleneck that limits the scalability and broader adoption of ML-powered mobile services (§2.2). Through our empirical study of over 20 ML models deployed in real-world mobile apps, spanning domains of live streaming, e-commerce, searching and advertising, we observe that behavior log size increases dramatically with the growing number and scale of on-device ML models, consuming up to 50% of the app size. Public statistics [1] show that excessive app size is the primary cause of app uninstalls, and our industrial statistics also reveal that each additional 10 MB in app size of TikTok can lead to around 61,000 fewer daily active users and $7000 financial losses per day. Consequently, the storage overhead of behavior log has emerged as a practical obstacle to the widespread adoption of ML-powered mobile services.

Our Motivation. In this work, we aim to tackle a crucial but unexplored problem: optimizing storage efficiency of behavior log for ML-embedded mobile apps. The optimization opportunities stem from our two key observations of real-world mobile data (§2.3). First, many features of ML models rely on partially overlapping attributes derived from the same behavior event, but are recorded as separate event rows in behavior log for fast retrieval, resulting in redundant data storage. Second, different behavior events are described by heterogeneous attributes but are stored in a unified log file, which leads to massive null values for each event's irrelevant attributes and results in sparse storage. These inefficiencies reveal promising opportunities for storage optimization of behavior logs.

To this end, we propose AdaLog, the first system designed to reduce the storage cost of behavior log without sacrificing on-device inference accuracy or latency. AdaLog is built on two core insights: (i) Reduce data storage redundancy by merging event rows that are logged by different features but

come from the same behavior events; (ii) Eliminate storage sparsity by distributing behavior events with heterogeneous attribute sets into separate sub-log files for dense storage. By tacking these storage-level inefficiencies, AdaLog provides a more powerful data foundation for ML-embedded mobile apps and offers a new optimization dimension that complements existing works on resource-efficient model inference, as they primarily focused on memory, computation, energy, etc (§5).

Challenges. Designing and implementing AdaLog system involves three key challenges.

First, identifying which features to merge their event rows for redundancy elimination is an NP-hard problem. While merging redundant event rows across any features can reduce data storage cost, it introduces additional overhead for the index structure, as more physical addresses have to be tracked to distinguish each feature's relevant event rows from the merged ones (§3.2). We show that deriving the optimal feature-level data merging strategy is equivalent to solving the maximum weighted matching problem in a hypergraph: each node denotes a feature and each hyperedge connecting multiple nodes denotes a candidate feature group, weighted by the potential storage savings after data merging. This optimization problem is NP-hard and computationally expensive.

Second, deciding where to store each behavior's event rows to eliminate storage sparsity is not straightforward either. Ideally, different user behaviors described by distinct attribute sets should be stored in separate log files, ensuring that event rows in one file share the same attribute columns. However, user behaviors captured by mobile apps are massive and highly diverse, leading to hundreds of small, fragmented log files (§3.3). Managing such a fragmented storage system incurs substantial metadata overhead (e.g., tracking table names, attribute column formats, file sizes, etc), making it difficult to simultaneously achieve low storage sparsity and high storage efficiency.

Third, maintaining an up-to-date behavior log for dynamic user behavior patterns is essential but costly. As behaviors of mobile users are unpredictable and evolving over time, the optimal strategies of feature-level data merging and behavior-level log splitting are also dynamic, requiring frequent re-optimization to preserve storage efficiency. Unfortunately, applying the strategy changes typically necessitates the reconstruction of behavior logs, involving large-scale I/O operations such as reading, writing and indexing. This imposes serious scalability challenges for deploying the log optimization system in industrial-scale mobile apps.

Our Solutions. To address these challenges, AdaLog introduces three core techniques to unlock the full optimization potential of behavior log. First, to identify an effective feature-level data merging strategy without excessive computation, we develop a hierarchical merging algorithm with polynomial time complexity. Instead of directly solving the hypergraph-based NP-hard problem, we decompose the solving process into multiple iterations. In each iteration, we reduce the hypergraph to a weighted 2D graph and merge only pairs of feature groups, iteratively refining the merging plan. Second, to achieve both low storage sparsity and metadata overhead, we introduce a virtually hashed attribute naming scheme, which allows heterogeneous attributes of different user behaviors to share the same virtual name and to be stored in one physical column of the log file. This reduces the required number of log files from the number of behavior types (≈ 250) to the number of possible attribute counts (≈ 20). Finally, to support efficient adaptation to dynamic user behavior patterns, we develop an incremental update mechanism that avoids full log reconstruction. Observing that changes in behavior patterns typically impact only partial optimization strategies and logged data, we propose a shrink-and-expand method that aligns past and new strategies to reuse as much existing data as possible and incrementally update the affected portions. This design minimizes I/O overhead and enables fast and scalable adaptation for resource-constrained devices.

Implementation and Evaluation. We have implemented AdaLog as a lightweight Python package and evaluate it in industrial mobile apps with the help of our industrial partner (§4), involving tens of practical on-device models, hundreds of real-world testing users and four service domains (live streaming, e-commerce, search and advertising). Compared to industry-standard
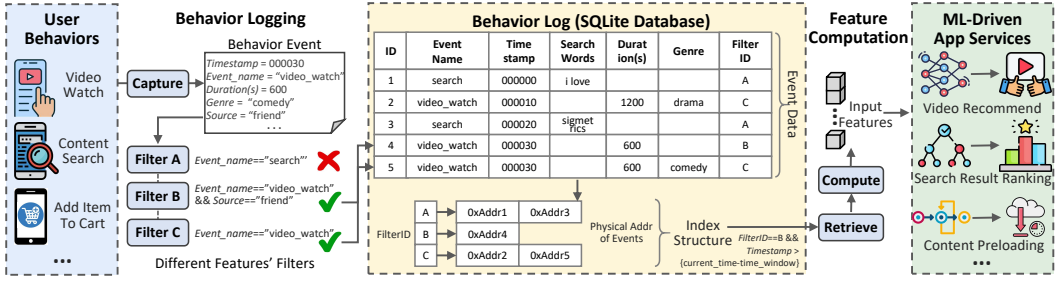
Fig. 1. In mobile apps like TikTok, behavior log acts as a middle layer between user behaviors and ML-driven services: (i) Each user behavior is *captured* as an event containing multiple attributes for description; (ii) Different features apply distinct *filters* to judge event relevance and select necessary attributes to store in behavior log, consisting of event data and index structure; (iii) During real-time feature computation, each feature uses index structure and time window constraints to *retrieve* relevant rows for accurate computation.

behavior log designs, AdaLog achieves up to 44% average reduction for behavior log size without compromising inference latency or accuracy, and maintaining minimal system costs of 2 seconds latency and 15MB memory footprint.

Contributions of this work are summarized as follows:

• We identify a critical but overlooked storage bottleneck in deploying ML models within industrial mobile apps, analyzing its root causes and optimization opportunities.

• We design and implement the first storage optimization system for behavior logs in ML-embedded mobile apps, which reduces the storage redundancy and sparsity without sacrificing on-device model inference accuracy or latency, providing a more powerful data foundation for on-device ML.

• We evaluate AdaLog on industrial mobile apps and real-world users, demonstrating significant storage savings and superior system efficiency compared to industry baselines.

2 Background and Motivation

In this section, we first elaborate on the critical role and workflow of behavior log in supporting ML-embedded mobile apps (§2.1). Next, we analyze the storage bottleneck of behavior log based on statistics from industrial mobile apps (§2.2). Finally, we identify key optimization opportunities and corresponding design choices that motivate our system (§2.3).

2.1 Role and Workflow of Behavior Log in Mobile Apps

ML models have become a core component of modern mobile apps [3, 21, 22, 47, 67, 77], powering various intelligent and personalized services by consuming private user data on mobile devices [10, 18, 19, 23, 27, 32, 33, 48–50, 55, 71, 78, 82, 83]. For modern mobile apps, ML models practically deployed on mobile devices rely on massive input features derived from a user's various historical behaviors to capture evolving contextual information and user intents [39, 46, 63]. This necessitates recording relevant behavior data in a specified file called *behavior log*, which is typically implemented as lightweight SQLite databases on both Android and iOS platforms [15, 61, 62].

The work flow of behavior log is shown in Figure 1, which serves as a critical middle layer to connect physical user behaviors and on-device ML models through two stages: behavior logging and feature computation.

Behavior Logging. During app usage, each user interaction with the smartphone can be captured as a behavior *event* (*e.g.*, "video watch"), which is represented as a structured format of numerous *attributes*, including behavior-independent attributes (*e.g.*, event name, timestamp) for identification
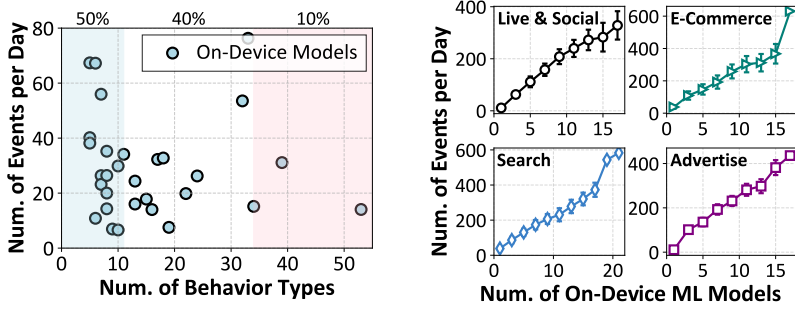
(a) Each individual model logs massive data.   (b) Log size grows linearly with model number.

Fig. 2. Statistics of real-world mobile data to illustrate behavior log storage bottleneck: (a) Each single ML model logs numerous behavior types and event rows daily; (b) Deploying more ML models linearly increases the storage cost due to their heterogeneous logging requirements.

and behavior-specific attributes to describe the behavior in multiple orthogonal dimensions (*e.g.*, duration, genre, volume for a "video watch" event). However, each input feature is typically used to reflect a certain dimension over a specified context (*e.g.*, the average watch time of videos shared by friends). Thus, in industrial practices, each feature applies a *filter* to (i) check whether current behavior is relevant by examining specific attribute values (*e.g.*, "event name"="video watch" and "source"="friend"), (ii) select necessary attributes from the matching event and append FilterID attribute, (iii) record them as an event row in behavior log. To enable low-latency data retrieval for real-time feature computation and model inference, an index structure is maintained to map each feature (identified by FilterID) to the physical storage addresses of its corresponding event rows.

Feature Computation. When an app service (*e.g.*, video recommendation) triggers an on-device model inference , the mobile device computes each required input feature in the following steps: (i) retrieve necessary attributes from the relevant event rows in behavior log using index structure and time window constraints, and (ii) compute the feature value through predefined computation functions like averaging, concatenating, etc. These resulting features are then concatenated and fed into the ML model to general final outputs for the app service.

2.2 Storage Bottleneck of Behavior Log

Despite its critical role in supporting on-device ML, the storage overhead of behavior log has become an emerging resource bottleneck for mobile apps, driven by the increasing scale and number of ML-powered app services. Our observation is grounded in analysis of over 20 ML models practically deployed on mobile devices, spanning service domains of live streaming, e-commerce, search and advertising domains, as detailed in §4.1.

Observation 1: A single model accumulates massive user behavior data. To comprehensively capture evolving user context and intents, each on-device ML model involves input features extracted from a wide range of user behavior types under various contexts and time windows. Figure 2(a) shows that over 50% of the examined ML models track at least 11 types of user behaviors and 10% track more than 34 types. This leads to a daily accumulation of 5-70 (around 0.18 MB) new event rows per model in behavior log. Considering that current input features can consider behaviors within time windows for up to 6 months, this accumulation significantly increases the long-term storage footprint of behavior log (around 30 MB per model).

Observation 2: Storage overhead scales linearly with the number of ML models. For popular mobile apps, numerous online services are powered by ML models, such as recommendation, preloading and ranking of data with all modalities. As each service has unique computing objective and scenario, different on-device ML models have distinct filters even for the same behavior type to compute features tailored to its task. For example, the video recommendation model logs only long-duration video watches to track user preferences, whereas an app exit prediction model logs all recent video watches to perceive the shifts of user attention. Consequently, as shown in Figure 2(b), our domain-wise analysis reveals that behavior log size increases linearly with the number of on-device ML models, with each additional model contributing 20-35 new event rows per day.

Significance of Storage Bottleneck. The overall storage overhead of a mobile app can be divided into two components[1]: (i) *App Size*, which includes the core application logic, software development kits (SDKs) and other runtime dependencies; and (ii) *Document&Data*, which stores data requiring persistent storage like chat history, cached videos and files, etc. In practice, the app size is typically constrained to a few hundred MBs to ensure fast app launching and smooth usage experiences, while document&data can grow to several GBs. Unfortunately, behavior log falls under the first category as it has to be consistently kept to support latency-sensitive services at any time and cannot be arbitrarily edited by users.

As a result, behavior log can account for over 50% of the app size according to our industrial data. The latest public statistics show that excessive app size is a primary driver of app uninstallation [1] and leads to significant financial losses for service providers [60, 66]. The reasons are many-facet. First, when the total size of an app becomes excessive, mobile OS platforms often issue warnings, prompting users to manually uninstall large apps to free up device storage. Also, a larger app size caused by behavior log can directly contribute to longer app launch time and degrade user experience. Further, our industrial data reveals that for every additional 10 MB in app size, the number of daily active users decreases by around 30, 000 for Douyin and 61, 000 for TikTok, leading to over $7,000 financial loss per day. This highlights a fundamental dilemma: while on-device ML models enables personalized and responsive user experiences, it simultaneously imposes growing storage burdens that can degrade user retention. By effectively relieving the pervasive mobile storage bottleneck, app developers could adopt more sophisticated and numerous ML-powered mobile services, which translates into a perceptibly superior user experience and thus drives user retention and recruitment. Consequently, optimizing the storage of behavior log has become a critical obstacle for the broader adoption of on-device ML.

2.3 Optimization Opportunities and Design Choices

Our investigation into behavior logs of mobile apps reveal two critical characteristics, *feature-level correlation* and *behavior-level heterogeneity*, which expose significant opportunities for improving storage efficiency. These insights stem from our observations on large-scale industrial deployments and empirical analysis of real-world mobile data, in collaboration with our enterprise partner. We uncover substantial redundancy and sparsity in current behavior log structures that can be effectively minimized without information loss.

Feature-Level Correlation. While input features apply distinct filters to extract respective necessary attributes from relevant behavior events, we find that their filtering conditions often overlap rather than being completely disjoint. This results in the same behavior data being recorded multiple times by different features, each associated with an individual event row and a unique FilterID. To quantify this redundancy, we analyze behavior logs collected from a production-scale

---

[1]On iOS devices, the storage cost per app can be observed via "Settings>General>iPhone Storage>$App Name$".

(a) Redundant event rows.
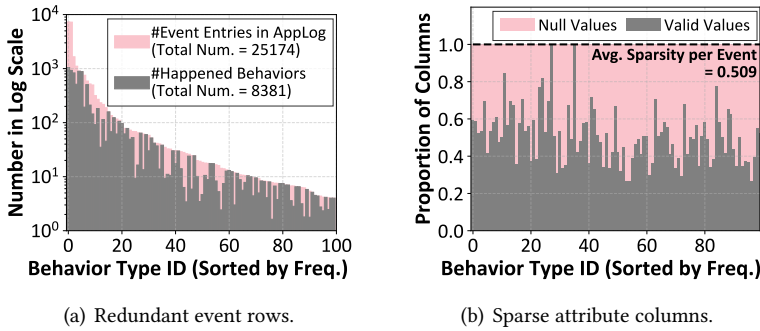
(b) Sparse attribute columns.

Fig. 3. Storage redundancy and sparsity of behavior log for an average user, where x-axis denotes different behavior types and y-axis depicts their redundancy and sparsity.

video app, Douyin(TikTok), which includes around 250 different types of user behaviors[2]. As shown in Figure 3(a), the behavior log accumulates an average of 25,174 event rows over 14 days, yet only 8,381 unique behavior events occurred in that period, implying up to 67% redundant data storage. This highlights a substantial optimization opportunity that can be achieved by eliminating redundant data across features.

To reduce such data redundancy without information loss, we have two distinct design choices: value-level optimization and row-level optimization. The first approach maps identical attribute values into smaller symbols, which reduces the size of individual attribute value but requires maintaining an additional mapping dictionary. The second approach merges redundant rows corresponding to the same behavior event into a unified row, reducing the total count of attribute values in behavior log. Our work considers row-level optimization due to its higher optimization potential. Value-level optimization is effective when the stored data exhibits high value-wise redundancy, *i.e.*, massive attributes of different behaviors events have identical and large-size values. However, in practice, different user behaviors have quite heterogeneous attributes with diverse types and values, and each value has only small size.

Behavior-Level Heterogeneity. Different types of user behaviors are inherently heterogeneous and have distinct sets of attributes. However, in current industrial practices, all behavior data is stored together in a single unified log file for index simplicity and centralized data access. This one-size-fits-all format introduces high storage sparsity, as irrelevant attributes are represented with null values. Our analysis of the 14-day dataset shows that: (i) Over 95% of event rows contain at least one null-valued attribute, (ii) On average, 50.9% of all attribute values per event row are null. While each null occupies only 1 B of space, massive event rows and numerous attributes result in a 11% wasted storage per user.

To reduce storage sparsity, we have three design choices: sparse data encoding, column-level and row-level splitting. Sparse data encoding aims to reduce the physical space occupied by each null value using special markers with smaller sizes. Column-level splitting involves splitting the log file based on column similarity, *i.e.*, grouping attributes according to their null value distributions across behavior events and storing each attribute group in one log file. Similarly, row-level splitting decomposes the log file into multiple sub-logs based on row similarity, *i.e.*, event rows that share a similar set of non-null attributes are stored densely in one log file. The first approach is ineffective in our context, as null values in mobile databases are already represented efficiently and the

---

[2]Details on the top 50 behavior types are presented in Appendix for justification A.
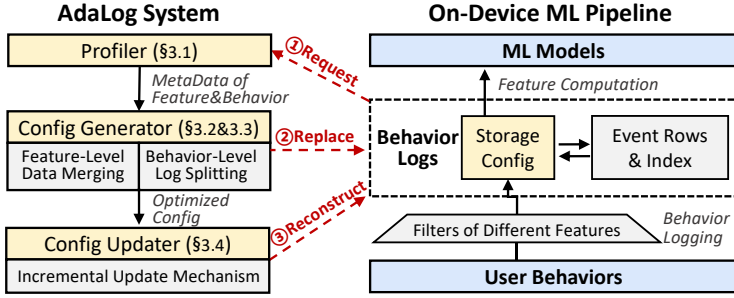
Fig. 4. AdaLog architecture and workflow within on-device ML pipelines.

sparse storage cost is mainly caused by numerous null values rather than their individual size. The second approach also fails as diverse user behaviors with heterogeneous attributes make their null distributions differ significantly. As a result, our work considers row-level splitting due to its optimal performance with the help of our dedicated designs.

3 AdaLog Design

We introduce AdaLog, a system designed to optimize the storage efficiency of behavior log for ML-embedded mobile apps by reducing storage redundancy and sparsity while remaining compatible with existing on-device ML pipelines. In this section, we first provide an overview of AdaLog (§3.1) and then elaborate its each key component, including feature-level data merging (§3.2), behavior-level log splitting (§3.3) and incremental update mechanism (§3.4).

3.1 Overview

As depicted in Figure 4, AdaLog works as a shim layer atop existing on-device ML pipelines, making it a flexible and generalized solution without requiring modifications to device operating systems or mobile inference engines.

Architecture. As illustrated in the left part of Figure 4, AdaLog consists of three main components that work together to optimize behavior log storage.

• *Profiler*: It monitors and collects two types of lightweight metadata from behavior log, including (i) IDs of event rows that are logged for each feature's filter (i.e., FilterID), and (ii) each behavior type's attributes that have to be logged for different filters. This metadata enables AdaLog to analyze both feature-level data redundancy and behavior-level attribute heterogeneity, providing foundation for subsequent optimization.

• *Config Generator*: Given the profiler's metadata, it computes an optimal storage configuration, consisting of two aspects: (i) Feature-level data merging: identify which features should have their redundant event rows merged to reduce redundancy with minimal extra cost; (ii) Behavior-level log splitting: determine which log file to store each behavior type's event rows to eliminate sparsity.

• *Config Updater*: It applies the new storage configuration to existing behavior log in a resource-efficient manner by performing incremental updates. Through matching the current and previous storage configurations, it reuses as much existing data as possible to minimize I/O operations.

Workflow. As shown in the right part of Figure 4, AdaLog's workflow is integrated into existing on-device ML pipeline through the following two stages, ensuring compatibility and efficiency.

• *Data Processing for Existing Pipeline*: During behavior logging, each behavior event is filtered as usual by different features. Instead of storing separate event rows for each filter, AdaLog merges multiple features' attribute subsets into a single event row according to the feature-level data

ID	Event Name	Time stamp	Source	Genre	Duration (s)	Volume (%)	Filter ID
1	video_watch	000010	Friend	Comedy	60		A
2	video_watch	000010		Comedy	60		B
3	video_watch	000010			60		C
4	video_watch	000010			60	50	D
5	video_watch	000020			120		C
6	video_watch	000020			120	30	D
7	video_watch	000030	Refresh	Drama	600		A

Index on FilterID

A	→	Addr1	Addr7
B	→	Addr2	
C	→	Addr3	Addr5
D	→	Addr4	Addr6

(a) Event rows and index structure before feature-level data merging.

ID	Event Name	Time stamp	Source	Genre	Duration (s)	Volume (%)	Filter A	Filter B	Filter C	Filter D
1	video_watch	000010	Friend	Comedy	60	50	True	True	True	True
2	video_watch	000020			120	30			True	True
3	video_watch	000030	Refesh	Drama	600		True			

Index on FilterA

True	→	Addr1	Addr3
Null	→	Addr2	

Index on FilterB

True	→	Addr1	
Null	→	Addr2	Addr3

Index on FilterC

True	→	Addr1	Addr2
Null	→	Addr3	

Index on FilterD

True	→	Addr1	Addr2
Null	→	Addr3	

(b) Event rows and index structure after merging all redundant event rows, where: 7-3=4 repetitive rows are removed, while 12-7=5 additional addresses are recorded by index.

Fig. 5. Comparing behavior logs before and after conducting feature-level data merging.

merging configuration, and stores the merged event rows in log files designated by the behavior-level splitting configuration. When computing features for model inference, AdaLog uses the current storage configuration to retrieve necessary attributes of relevant event rows from the appropriate log file, ensuring correctness and efficiency.

• *Periodic Behavior Log Optimization*: AdaLog periodically (e.g., daily or during app updates allowed by users) invokes a lightweight behavior log reconstruction process, involving: ① requesting the profiler to update metadata based on the latest behavior log, ② replacing the outdated configuration with the new one computed by config generator, ③ incrementally updating the behavior log to adapt to the new configuration.

3.2 Feature-Level Data Merging: Reduce Redundant Event Rows

A straightforward method to eliminate redundancy in behavior log is to merge all event rows across features that correspond to the same behavior event. As illustrated in Figure 5, this involves two steps: (i) Merging event rows with identical event name and timestamp attributes into a single row containing the union of all required attributes; (ii) Appending a set of FilterID columns to the merged row to differentiate which filters the merged row satisfies. While this method effectively eliminates redundant data, it introduces a new significant challenge: index inflation.

In modern mobile apps, behavior logs are indexed on FilterID attribute column to support fast data retrieval for computing each feature. As shown in the green part of Figure 5, the database index structure maps every value of the indexed column, including nulls, to the physical addresses of the rows where those values appear. However, since each behavior event satisfies only a subset of features' filters, the merged rows inevitably contain null values in the appended FilterID columns (shown in blue part of Figure 5(b)). These nulls inflate the index structure by forcing it to store unnecessary mappings (shown in green part of 5(b)). Consequently, if redundant data is merged across inappropriate features, the overhead of the expanded index can easily surpass the storage saved by eliminating redundant data. This trade-off becomes more severe with massive models and features, as the number of possible feature groups grows exponentially.
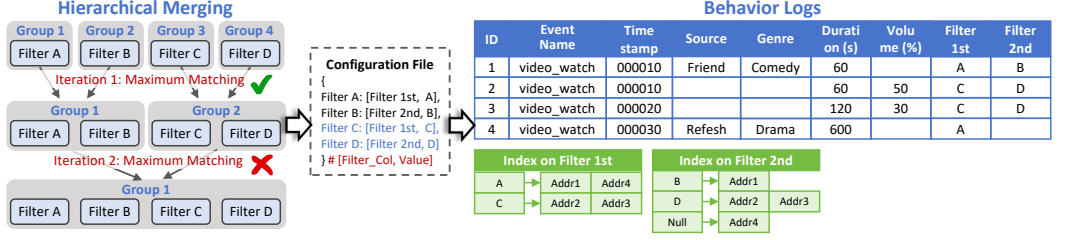
Fig. 6. An example to illustrate hierarchical merging algorithm (left), storage configuration of feature merging (middle) and behavior log after merging (right). Compared to Figure 5(a), 7-4=3 redundant event rows are eliminated while only 8-7=1 address is additionally recorded by index.

To address this issue, we formulate the feature merging decision as a classic maximum weighted matching problem in a hypergraph, and design a hierarchical merging algorithm with polynomial time complexity for scalable on-device execution.

Problem Formulation. Given a set of features $\mathcal{F}$ and each feature $f \in \mathcal{F}$'s relevant event rows $\mathrm{E}(f)$, attributes $\mathrm{A}(f)$ and physical address size $\mathrm{Size}(Addr)$, we aim to minimize the total storage cost by partitioning features into disjoint feature groups $\mathcal{G} = \{g_1, \ldots, g_M\}$ where intra-group features share merged event rows. The optimization problem can be formally expressed as:

$$\mathcal{G}^* = \underset{\mathcal{G}: \cup_{g \in \mathcal{G}} g = \mathcal{F}}{\arg\min} \sum_{i=1}^{M} \Big[\mathrm{Data\_Size}(g_i) + \mathrm{Index\_Size}(g_i) \Big],$$

$$\text{s.t. } \mathrm{Data\_Size}(g_i) = \underbrace{\big| \cup_{f \in g_i} \mathrm{E}(f) \big|}_{\text{Num. of Event Rows}} \times \underbrace{\mathrm{Size}\big(\cup_{f \in g_i} \mathrm{A}(f) \big)}_{\text{Size per Event Row}},$$

$$\mathrm{Index\_Size}(g_i) = \underbrace{| \cup_{f \in g_i} E(f) | \times \mathrm{Size}(Addr)}_{\text{Address Size per Index}} \times \underbrace{\max_{g \in \mathcal{G}} |g|}_{\text{Index Num}}.$$

For each feature group g_i, data_size captures the space for storing event data, measured as the product of the number of event rows and the size per row, while index_size represents the index structure space, quantified as the product of the number of indexed columns and the total address size of all event rows.

NP-Hardness. We notice that this problem can be interpreted as a maximum weighted matching problem in a hypergraph $G = (V, E)$, where: (i) Each feature $f \in \mathcal{F}$ is represented as as a node $v \in V$; (ii) Each potential feature group $g \subseteq \mathcal{F}$ is represented as a hyperedge $e \in E$ connecting its member features' nodes; (iii) The weight of hyperedge e equals the overall storage savings from merging features in g. A valid feature grouping strategy corresponds to a matching in the hypergraph, *i.e.*, a set of disjoint hyperedges. Thus, finding the optimal feature grouping strategy is equivalent to solving the maximum weighted matching problem on the hypergraph G, a well-known NP-hard problem in graph theory [6, 11, 52].

Hierarchical Merging Algorithm. To efficiently solve the NP-hard hypergraph-based problem in practical mobile settings, we propose a hierarchical merging algorithm that avoids direct hypergraph optimization by decomposing it into a series of tractable 2D-graph matchings. This is because maximum matchings in 2D-graphs can be found by Blossom algorithm [36] with only polynomial $O(|V|^3)$ complexity. Our key idea is similar to hierarchical clustering [57] as shown in the left part of Figure 6: we start from a fine-grained configuration where each feature is treated as

a standalone group and iteratively merge *pairs of feature groups* that lead to the highest storage reduction, providing a practical balance between optimization quality and system efficiency.

Specifically, in each iteration t, we construct a weighted 2D-graph $G = (V, E)$ for current feature groups $\mathcal{G}^t$, where each node $v \in V$ denotes a feature group $g \in \mathcal{G}^t$, each edge $e = (g_i, g_j) \in E$ connects two feature groups g_i and g_j and the edge weight $w(g_i, g_j)$ quantifies the storage savings if event rows belonging to feature groups g_i and g_j are merged:

$$w(g_i', g_j') = \Delta\text{Data\_Size} + \Delta\text{Index\_Size} \approx$$

$$\underbrace{|\text{E}(g_i') \cap \text{E}(g_j')|}_{\text{Redundant Rows}} \times \underbrace{\Big[\text{Size}\big(A(g_i')\big) + \text{Size}\big(A(g_j')\big) - \text{Size}\big(A(g_i') \cup A(g_j')\big)\Big]}_{\text{Size of Overlapped Attributes per Row}}$$

$$+ \Big[\underbrace{|\text{E}(g_i') \cup \text{E}(g_j')| \times |g_i' \cup g_j'|}_{\text{Address Num. after Merging}} - \underbrace{\big(|\text{E}(g_i')| \times |g_i'| + |\text{E}(g_j')| \times |g_j'|\big)}_{\text{Address Num. Before Merging}} \Big] \times \text{Size}(Addr).$$

The data_size term captures data storage reduction due to eliminating attributes of overlapping event rows across features and the index_size term accounts for index size changes. Then, using the Blossom algorithm, we identify a maximum weighted matching on the 2D-graph G, *i.e.*, a set of disjoint pairs (g_i, g_j) with the highest storage savings. If the total gain is positive, the matched pairs are merged into new feature groups $\mathcal{G}^{t+1}$ for next iteration. Otherwise, the algorithm terminates and outputs current feature groups as feature-level data merging configuration.

The hierarchical merging algorithm runs for at most $\log_2 |\mathcal{F}|$ iterations, as the number of feature groups (*i.e.*, nodes in the graph) is halved in each iteration, implying $\frac{|\mathcal{F}|}{2^{t-1}}$ nodes in the t-th iteration. Consequently, the total time complexity becomes polynomial:

$$\sum_{t=1}^{\log_2 |\mathcal{F}|} O\left(\left(\frac{|\mathcal{F}|}{2^{t-1}}\right)^3\right) = O(|\mathcal{F}|^3) \cdot \sum_{t=1}^{\log_2 |\mathcal{F}|} \left(\frac{1}{8}\right)^{t-1} = O(|\mathcal{F}|^3).$$

System Implementation and Optimization. The hierarchical merging algorithm can be integrated into `AdaLog` with further system optimization. First, `AdaLog`'s profiler can directly collect IDs of event rows relevant to each feature using the index structure. Other necessary information such as attribute size and address size is fixed and can be profiled in advance. Second, `AdaLog`'s generator computes the optimal merging configuration in an efficient manner. It pre-clusters features based on the targeted behavior types and performs hierarchical merging algorithm independently for each behavior type's related features, which reduces the problem size and enables parallel execution. As shown in Figure 6, the configuration is stored in a dictionary-like structure, which designates (i) features in the same group to distinct FilterID columns and (ii) features across different groups to shared FilterID columns with different specific values. Our evaluations in §4.2 demonstrate that `AdaLog` incurs ≤ 1 second of latency to complete the entire algorithm on device and the configuration size is ≤ 10 KB, demonstrating high system efficiency.

3.3 Behavior-Level Log Splitting: Minimize Overall Sparsity

In modern mobile apps, hundreds types of user behaviors are captured as heterogeneous events, each containing a unique set of behavior-specific attributes, as shown in Figure 7(a). Current industrial practices of app behavior logs commonly store all behavior events in a single unified log file to simplify indexing, querying and management. However, this approach results in severe storage sparsity when more types of user behaviors are consumed by ever-growing ML-embedded services, as each event row can contain massive null values for irrelevant attributes.

(a) Heterogeneous attribute distribution.
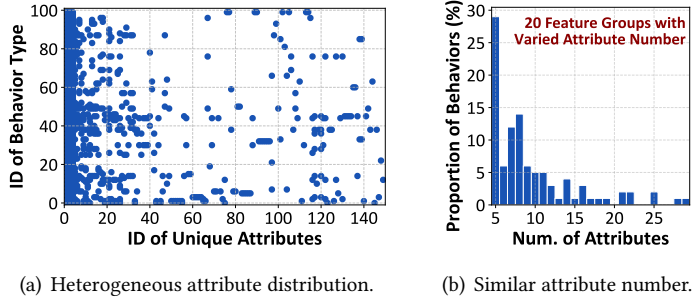

(b) Similar attribute number.

Fig. 7. Different user behavior types have heterogeneous attributes but similar attribute number.
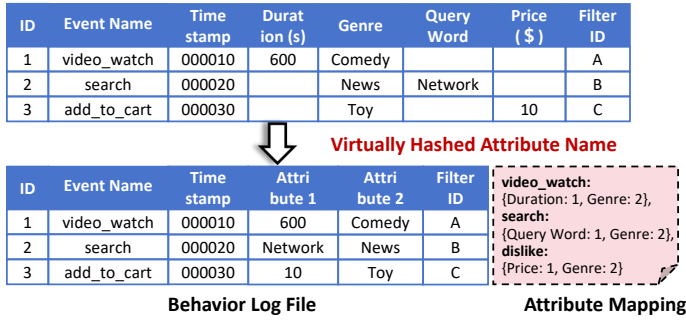

Fig. 8. VHAN uses attribute mapping to densely storage behavior events with identical attribute numbers.

A direct solution to eliminate sparsity is behavior-level log splitting, where behavior events are stored in separate log files according to their attribute sets, ensuring that all event rows within one log file share identical attributes. However, this solution is impractical: the heterogeneous attribute sets of different behavior types require splitting the behavior log into hundreds of small, fragmented files, as illustrated in Figure 7(a). While this strategy minimizes storage sparsity, it introduces significant overhead for file management and metadata storage, as each small file requires its own set of metadata such as table names, column names, index structures, file sizes, etc.

Virtually Hashed Attribute Name. To overcome the above challenge, we propose virtually hashed attribute naming (VHAN) design, a logically sparse but physically dense storage design for behavior logs to reduce sparsity without creating massive files. We observe that different behavior types often have similar numbers of attributes but different attribute names (Figure 7(b)), which is the root cause preventing them from storing in the same dense log file. Therefore, we propose to decouple the storage of attribute values from their physical names by using virtual attribute IDs. This design is analogous to virtual memory in operation systems [12] where virtual addresses abstract away physical memory locations. As shown in Figure 8, VHAN consists of two main parts:

- *Attribute Mapping*: A dictionary-like configuration file that maps each physical attribute name into a virtual ID for each user behavior type.
- *Behavior Log File*: The physical storage for event rows, which retains the same structure as existing behavior log but replaces attribute names with their corresponding virtual IDs from the attribute mapping.

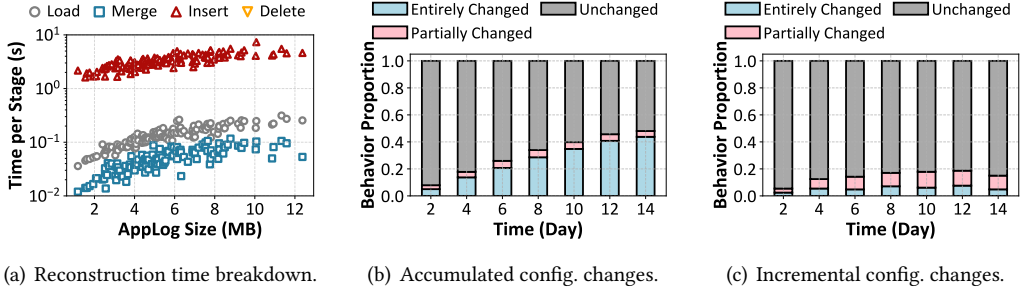(a) Reconstruction time breakdown. (b) Accumulated config. changes. (c) Incremental config. changes.

Fig. 9. Analysis of real-world behavior logs to (a) break down reconstruction time, and (b)(c) reveal the proportions of an average user's behaviors that experience changes in storage configuration over time.

By leveraging VHAN, AdaLog enables the dense storage of any behavior events with identical numbers of attributes in one behavior log file, eliminating the strict requirement of totally same attribute sets. As a result, we propose to cluster user behaviors according to their cardinality of attribute sets, and store event rows of behaviors within the same cluster in one log file, which reduces the number of log files from the number of unique attribute sets (≈250) to the number of unique attribute count (≈20), as shown in Figure 7(b).

System Implementation Overhead. The VHAN design introduces an additional attribute mapping file, which is used during behavior logging and feature computation to map physical attribute names to virtual IDs for event storage and attribute retrieval. Therefore, two choices of overhead are introduced: *(i) Memory*: Maintaining the attribute mapping in device memory facilitates real-time event logging and feature computation, but introduces a memory footprint of around 30KB, which is negligible for even low-end smartphones. *(ii) Latency*: Alternatively, loading the attribute mapping on-demand for each event logging or model inference process introduces millisecond-level latency, which is typically acceptable for most real-time applications.

3.4 Behavior Log Reconstruction at Scale

The previous designs successfully optimize the storage efficiency for given static behavior data. However, mobile users' behavior patterns are inherently dynamic and unpredictable, which presents significant scalability challenges in maintaining up-to-date configurations and behavior logs in real-world mobile settings.

Specifically, as new user behavior events are continuously recorded in behavior log, the distribution of event rows across features also shifts. This leads to changes in the optimal storage configurations and requires reconstruction of the behavior log. However, reconstruction typically incurs substantial latency due to the large volume of event rows in behavior logs. For example, reconstructing a 10 MB behavior log on iPhone 13Pro requires around 10 seconds. This can lead to intolerant app performance degradation due to preempting computation and I/O resources necessitated by other concurrent services, like video rendering and content loading.

To address this challenge, we start with analyzing the bottleneck operations during the log reconstruction process, and further propose an incremental update mechanism that allows AdaLog to adapt existing behavior log files to new configurations with minimal system overhead.

Overhead Breakdown. The reconstruction of behavior logs involves four major steps: *(i) Loading* relevant event rows for each feature, *(ii) Merging* these rows according to the feature-level data merging strategy, *(iii) Inserting* the merged rows into newly created behavior log files
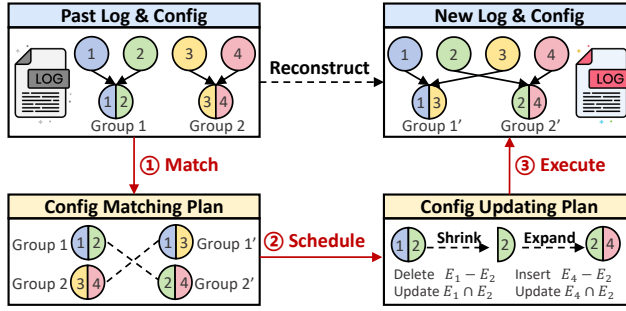
Fig. 10. AdaLog's Incremental update mechanism.

determined by the behavior-level log splitting configuration, *(iv) Deleting* outdated log files, which incurs negligible cost. To analyze the bottleneck operations, we conducted an extensive analysis of hundreds of real-world behavior log reconstruction workloads across various file sizes. As shown in Figure 9(a), we find that approximately 95% of the reconstruction time is dominated by I/O-intensive operations, such as data loading and inserting. This observation motivates us to minimize unnecessary I/O operations by reusing as much of the existing data as possible, thereby enabling incremental updates instead of full reconstructions.

Incremental Update Opportunities. To assess the feasibility and potential of incremental updates, we investigate how the optimal storage configuration of a mobile user evolves over time. Specifically, we collect daily optimal configurations for TikTok users over a 14-day period and decompose them into behavior-wise configurations. Each behavior's configuration is then categorized into three types:

- *Entirely Changed*: Both data merging and log splitting configurations are altered, requiring a full reconstruction of corresponding event rows.
- *Partially Changed*: Only specific feature groups within the data merging configuration are modified, requiring updates to the affected event rows.
- *Unchanged*: The configuration remains stable, and no updates are needed.

As shown in Figure 9(b), while the proportion of behaviors with entirely changed configurations accumulates over time, most behaviors' configurations are unchanged or partially modified. When examining changes over shorter intervals, such as a 2-day window (Figure 9(c)), 86% of behaviors retain the same configuration, 10% exhibit partial changes, and only 4% require a complete reconstruction. These findings suggest that most configuration changes can be handled through incremental updates rather than full reconstructions, leading to significant optimization potential.

Incremental Update Mechanism. Building on the above analysis of configuration evolution, we propose a novel incremental update mechanism for AdaLog's updater module. Our core idea is to reuse as many of the existing event rows as possible and minimize the I/O operations required to adapt them to new configurations. As elaborated in Figure 10, our incremental updater operates through three key steps:

① *Match.* To efficiently adapt the behavior log with the old configuration to the new one, we model the optimal adaption process as a maximum weighted matching problem in a bipartite graph:

- Each node in the left partition of the bipartite graph represents a feature group $g_i \in \mathcal{G}$ in the old configuration.
- Each node in the right partition represents a feature group $g'_j \in \mathcal{G}'$ in the new configuration.
- Each pair of feature groups (g_i, g'_j) is connected by an edge whose weight is defined as the cardinality of the intersection between their event rows, i.e., $|E(g_i) \cap E(g'_j)|$.

(a) Behavior types used by different models. (b) Number of event rows. (c) Sizes of behavior log.
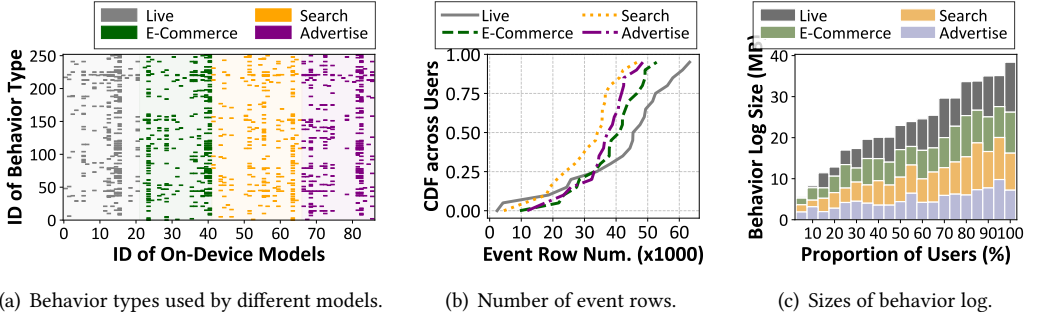
Fig. 11. High-level statistics of user behaviors, ML models and datasets involved in our evaluation. The top 50 user behaviors are presented in Appendix for justification.

The intuition behind this formulation is that a larger data overlap between two feature groups implies greater potential for data reuse. By maximizing the total weight of matched pairs, we minimize the number of I/O operations required for insertion, deletion, or attribute updates, since a high-overlap pair can be transformed with minimal modification. Next, we apply the Blossom algorithm to determine the optimal one-to-one mapping between old and new feature groups[3], ensuring that the transformation plan preserves as much existing data as possible.

② *Schedule.* Once the optimal mapping between old and new feature groups is obtained, we introduce a shrink-and-expand strategy to transform event rows from the old configuration to the new one with minimal rewriting overhead:

- (i) Shrink: For each matched pair (g_i, g'_j), we reduce g_i to the intersection $g_{ij'} = g_i \cap g'_j$. This involves removing event rows from $E(g_i)$ that are not present in $E(g_{ij'})$, as well as pruning obsolete attributes from the overlapping rows $E(g_i) \cap E(g_{ij'})$.
- (ii) Expand: After shrinking, we expand the pruned set $g_{ij'}$ to fully match the structure of g'_j. This expansion step inserts new event rows that are required by g'_j but not present in $E(g_{ij'})$, and inserts newly introduced attributes to the existing rows in $E(g_{ij'}) \cap E(g'_j)$.

③ *Execute.* Finally, the incremental update plan is executed for each behavior type. If the configuration has entirely changed, such that $g_{ij'} = \varnothing$, the mechanism falls back to a full reconstruction of the new feature group. If the configuration is partially changed, we leverage the matching and scheduling plan to perform minimal updates, effectively reusing the majority of event data.

4 Evaluation

In this section, we systematically evaluate the performance of AdaLog to answer the following key questions: How effectively and efficiently does AdaLog reduce behavior log storage overhead across diverse mobile users and application domains (§4.2)? What is the contribution of each core design component to AdaLog's performance (§4.3)? How is AdaLog affected by different factors (§4.4)?

4.1 Methodology

Implementation. We have fully implemented AdaLog as a system prototype comprising around $3K$ lines of code, which was deployed and evaluated on real-world mobile apps within the ByteDance ecosystem, a company with advanced mobile AI technology and billions of daily active users.

---

[3]For each feature group in the new configuration: If it is in the matching, we transform event rows of the matched past feature group; Otherwise, we simply reconstruct the required event rows.

Specifically, AdaLog is packaged as a lightweight Python library with developer-friendly APIs , and thus can be seamlessly integrated into the app SDKs of online users who consented to participate in our system evaluation and data collection. During evaluation, AdaLog operates autonomously to optimize behavior log on a daily basis[4] without manual intervention from users or developers. *It is important to note that all data collection and system operations strictly comply with privacy-preserving standards set by both industry and academia.*

Mobile App Domains. To demonstrate the broad applicability of AdaLog, we evaluate its performance across four representative mobile app domains, each involving various application services and on-device ML models.

• *Live streaming* (e.g., TikTok): This domain of app involves 21 on-device ML models for services like customized video preloading, recommendation, bandwidth management, user engagement prediction, etc. User behaviors include comments, (dis)likes, shares, subscribes, etc.

• *E-Commerce* (e.g., Taobao): This domain of app employs 20 ML models for personalized product recommendation, item ranking and preloading as well as comment selection by analyzing user-product interactions such as item clicks, favorites, adding to cart and purchases.

• *Search* (e.g., Baidu): This domain of app leverages 25 on-device ML models to improve users' searching experiences through predicting query keywords, ranking returned results, preloading multi-modal content, predicting search and exit timing, etc.

• *Advertisement & Monetization* (e.g., Google Ads): 20 on-device ML models are used to optimize advertisement delivery, targeting and monetization for maximizing user engagement and application revenue opportunities.

• *Unified Application*: An ideal case where user data across multiple app domains can be stored and optimized in an unified manner. This setup is feasible for (i) services or mobile apps belonging to the same parent company, and (ii) an operating system authorized to manage various native apps.

Due to strict enterprise confidentiality requirements, we cannot disclose the specific names of our testing mobile apps. The number and name of ML-powered services within an mobile app are quite sensitive due to their importance to user experience guarantee and high economic profit. We acknowledge that our primary evaluation focuses on apps where the user base is predominantly Chinese, which may lead to potential biases on app usage patterns and performance evaluation.

Models. In our experiments, the on-device ML models span a range of complexity, from lightweight models such as decision trees [65] and multilayer perceptrons (MLPs) [64] that leverage a few behavior features, to complex deep neural networks [9, 10, 19] that leverage hundreds of behavior features[5]. We provide high-level statistics in Figure 11(a) to depict the user behaviors leveraged by different models and features. While we cannot disclose the specific structure of testing models, we present a general model architecture adopted by most mobile services in Figure 12, which composes of three layers. *(i) Input Layer*: An on-device model takes three categories of features as inputs: cloud features to provide global information, device features to describe the current device state and massive behavior features to summarize various historical user behaviors; *(ii) Processing Layer*: These features are then processed by different layers. Statistical features of

---

[4]In our default evaluation setting, we consider updating each user's behavior log on a daily basis, driven by both system constraints and the temporal characteristics of user data. First, to avoid disrupting on-device model inference and compromising user experience, the log update process is expected to be scheduled during a guaranteed low-usage time period, typically deep at night when the mobile device is idle or charging. Second, we observe that the impact of a single day's user interactions on the data distribution of overall behavior log is relatively limited. Updating more frequently than daily would incur more system overhead without substantial gain in log size reduction, as analyzed in §4.4.

[5]While our evaluation focuses on ML models that have been deployed within mobile apps, we believe that advanced mobile intelligence with large language models [45, 74, 79] will require much more user behavior data for personalized finetuning and inference, making our work more applicable in the future.
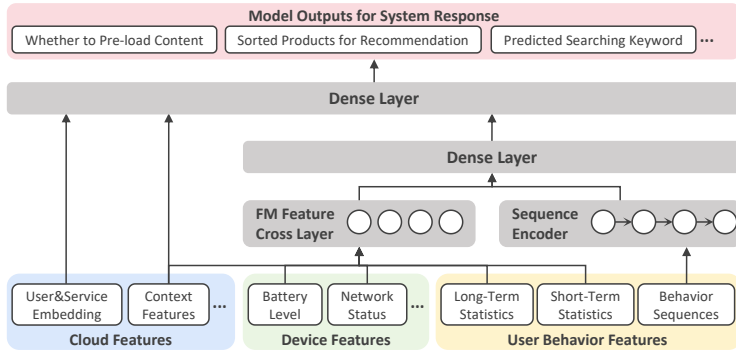
Fig. 12. Abstracted model architecture of common on-device machine learning models in practice.

user behaviors and device features are passed into an FM (Factorization Machine) layer for feature crossing, while sequential features of user behaviors are sent to an sequence encoder to capture temporal dynamics and periodical patterns; *(iii)* Output Layer: Finally, the combined feature outputs are passed through several dense layers to generate final predictions for personalized system responses. The on-device ML models are typically limited to tens of MBs in size. This constraint is driven by two critical factors: *(i) Model size limitation:* Mobile platforms impose strict limits on app size (e.g., 2GB for iOS [13] and 4GB for Android [24]), which directly limits the size of each ML model deployed within a mobile app; *(ii) Inference latency:* Ensuring low-latency model execution (typically within hundreds of milliseconds) is essential for a good user experience, which further limits model complexity and size.

Datasets. For each app domain, we evaluate AdaLog using real-world behavior log collected from real-world mobile users over a 14-day period. The datasets include operating systems of iOS and Android, various app usage frequencies (ranging from 2,340 to 52,852 behavior events per user as shown in Figure 11(b)) and a wide range of behavior log file sizes as shown in Figure 11(c). The observed behavior log sizes are on the order of tens of MBs because the analysis is based on only 14 days of data per user, constrained by the enterprise's online evaluation and data collection process. In real-world deployments, the log sizes would be significantly larger, as they typically record behavior events spanning much longer time periods up to 6 months. Note that if 14 days of data yields substantial storage reduction, the benefits will only scale as the log size grows, as analyzed in §4.4.

Baselines. To the best of our knowledge, AdaLog is the first system designed to optimize behavior log storage for practical ML-embedded mobile apps. Thus, we compare AdaLog against two baselines: (i) *w/o AdaLog*: A standard industry behavior log system that does not incorporate AdaLog support. In this system, all behavior events relevant to on-device ML models are stored in behavior log without optimization, leading to higher storage overhead. (ii) *AdaLog variants*: Different variants of AdaLog where individual components are disabled or modified, allowing us to isolate the impact of each design technique on system performance.

Metrics. We comprehensively evaluate AdaLog using three key metrics. *(i) Compression Ratio*: We measure the behavior log size reduction achieved by AdaLog to quantify storage efficiency, expressed as the percentage decrease in log size compared to the original behavior log. *(ii) Feature Computation Time*: To assess the impact of AdaLog on ML inference speed, we measure the wall-clock time required to compute features for model inference with and without AdaLog. This metric is crucial for verifying that the storage optimizations do not negatively affect real-time inference
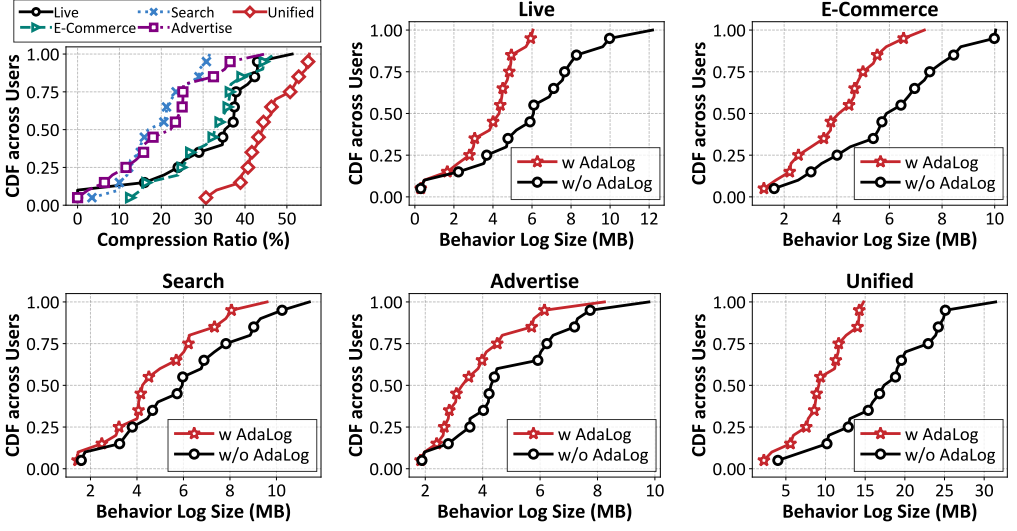
Fig. 13. Distributions of compression ratio and behavior log size across users with and without AdaLog.

performance on mobile devices. *(iii) System Overhead*: We measure the execution time and peak memory usage when AdaLog is invoked. This provides insight into the system's efficiency and its suitability for deployment on resource-constrained devices.

4.2 Overall Performance

We start with measuring the overall performance of AdaLog across diverse application scenarios and mobile users.

AdaLog **significantly reduces behavior log storage overhead.** Figure 13 presents the distribution of AdaLog's compression ratio across different users in various application domains. Compared to the industry-standard behavior log design, AdaLog achieves substantial reductions in storage consumption, with an average compression ratio of 35.1% for live streaming domain, 32.8% for e-commerce, 18.9% for search and 23.4% for advertisement. In the unified application case, where cross-scenario optimization is possible, AdaLog achieves an impressive 44% reduction in behavior log size for an average user. This reduction translate to a 1.82× increase in the numbers of on-device ML models (or ML-powered application services) that can be supported under the same storage cost. Figure 13 further illustrates how AdaLog impacts behavior log size distribution across users. Notably, users with larger behavior logs benefit the most, as they typically exhibit higher application usage and generate a greater volume of hot behaviors. These behaviors are often necessitated by massive features of on-device ML models and lead to high redundancy. This trend highlights the potential of AdaLog in reducing more storage space for active users, ultimately improving the number of daily active users and application revenue.

AdaLog **preserves or improves feature computation speed for model inferences.** A critical concern for storage optimization is its impact on real-time feature computation for on-device ML inference. Since AdaLog introduces a one-time overhead by loading attribute mappings from a stored configuration file into memory, we evaluate its effect on feature computation latency. Figure 14 presents the wall-clock time required to compute various features across mobile users, which are categorized by the time window of behaviors considered by each feature. We observe that AdaLog maintains near-identical computation speed for short-period features (minute and hour-level) while
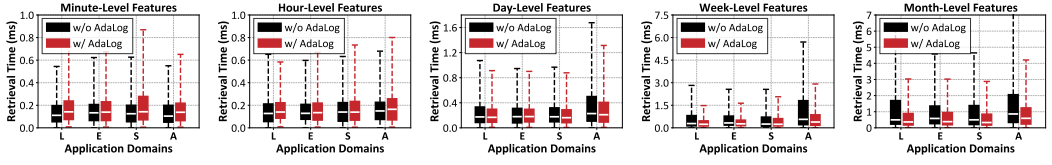
Fig. 14. Wall-clock time across mobile users for computing different features with various time windows, where L, E, S, A represent scenarios of live streaming, e-commerce, search and advertisement.
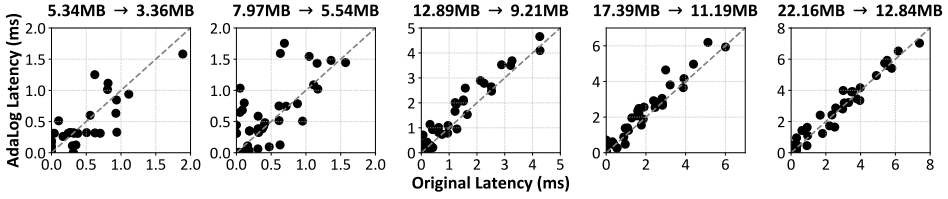

Fig. 15. We compare the data retrieval latency during on-device model inferences when using and not using AdaLog system. Each subfigure represents a user with a distinct behavior log size. Within each subfigure, every data point represents the measured latency for a specific on-device ML model.

consistently improving retrieval efficiency for long-period features. This improvement stems from two key factors. *(i) Amortized Overhead*: The millisecond-level cost of loading attribute mappings is spread across multiple features during each on-device model inference, reducing its impact; *2) Reduced Redundancy*: By reducing the number of event rows in behavior log files, AdaLog speeds up database retrieval operations.

To thoroughly measure the impact of AdaLog on on-device model inferences, we collected behavior logs with diverse sizes from various users ranging from 5.34 MB to 22.16 MB, re-structured them with and without AdaLog, and measured the time required for each on-device model to retrieve its necessary data on our testing device iPhone 13 Pro. In Figure 15, we visualize the retrieval latency between original industry-standard log design and AdaLog system across 5 representative users' data. Overall, we observe that the data retrieval latency of AdaLog is consistent with the original industry-standard logging system. This confirms our goal that AdaLog achieves storage reduction without imposing a penalty on model inference speed. For smaller behavior logs, AdaLog's performance exhibits instability and large variance, sometimes appearing inferior to the original log design. This is because the data retrieval latency is naturally very short for these small logs, making the measurement highly sensitive to transient device hardware states. In contrast, for larger logs, AdaLog maintains a stable and comparable retrieval speed to the original, unoptimized design, validating the effectiveness of our optimized storage structure. As a result, AdaLog effectively reduces storage overhead without compromising the responsiveness or personalization of ML-powered application services.

AdaLog **introduces minimal system overhead.** To ensure real-world deployment feasibility, we analyze AdaLog's execution overhead on mobile devices. Figure 16 provides a breakdown of both time and peak memory consumption during AdaLog's daily execution.

• *Execution Time*: On average, AdaLog completes its optimization process in 2.03 seconds for the unified case, with wall-clock time ranging from 0.642 to 0.797 seconds across different users and application scenarios. As shown in Figure 16(a), 41% of the time is spent on the incremental update process, 42% is used for profiling necessary information, while only 17%($0.05 \sim 0.32$s) is consumed

(a) Time overhead of overall system and each key stage.
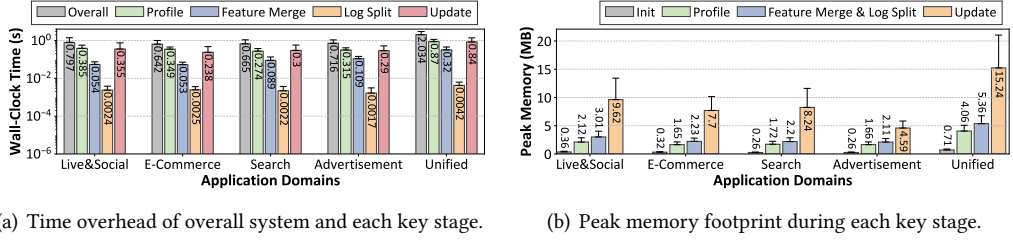
(b) Peak memory footprint during each key stage.

Fig. 16. System overhead of AdaLog across mobile users for different mobile app domains.

for generating optimal configurations via feature merging and log splitting. The remarkably low latency is enabled by: (i) The low complexity of our proposed hierarchical merging algorithm and simple attribute-count-based splitting strategy, and (ii) System-level optimizations that conduct hierarchical merging algorithms for different behaviors in parallel.

• *Memory Usage*: Figure 16(b) illustrates the memory consumption during different processing stages. Peak memory usage occurs during the incremental update process, averaging 4.59-15.24MB across application scenarios. This is primarily due to the need to load and process event rows affected by configuration changes, typically involving hundreds of rows. A secondary memory peak arises during the feature merging and log splitting stage, which incurs an 2.2-5.36MB footprint due to constructing graphs for each behavior's related features. Overall, AdaLog maintains a memory footprint below 20MB, making it lightweight and well-suited for modern mobile devices.

4.3 Component-Wise Analysis

To further validate the effectiveness of each key design in AdaLog, we implement multiple modified versions on 20 voluntary mobile users in the unified application case.

Feature-Level Data Merging. A core innovation in AdaLog is hierarchical merging algorithm, which balances data storage reduction, index size increase and computational efficiency. To evaluate its significance, we compare it against two alternative strategies: *(i) Random*: Features are randomly grouped for data merging, with performance averaged over 10 trials; *(ii) Greedy*: Features are grouped using a classic greedy algorithm [4] that prioritizes hyperedges (i.e., feature groups) with the highest weights (i.e., storage reduction) when solving the maximum weighted matching problem. Figure 17 presents a comparative analysis of compression ratio, execution time, and redundancy elimination across these methods. Our findings reveal two critical insights. (i) AdaLog achieves the optimal trade-off between compression and efficiency. While *Greedy* approach achieves a compression ration comparable to AdaLog, it suffers from 86× higher execution time due to the exponential number of hyperedges for given features. Conversely, *Random* runs faster but performs 26% worse in compression on average. (ii) Despite both *Random* and *Greedy* remove a similar number of redundant event rows, their practical storage reduction is lower. In some cases, *Random* even increases storage costs due to the increased index overhead. This highlights the necessity of hierarchical merging algorithm in ensuring maximal compression ratio without excessive computational burden.

We further analyze the specific scenarios where our pairwise hierarchical merging strategy might be practically sub-optimal. Our hierarchical pairwise algorithm operates by greedily merging the two feature groups that yield the largest potential storage reduction in each step. This method is near-perfect when the main source of redundancy is found between simple pairs of features. However, the algorithm's performance could theoretically degrade if the absolute maximum redundancy
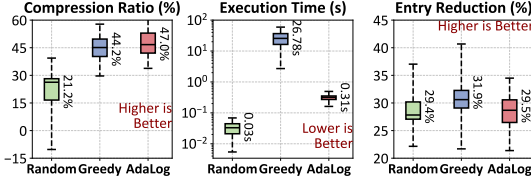
Fig. 17. Effect of hierarchical merging algorithm in balancing storage reduction and efficiency.
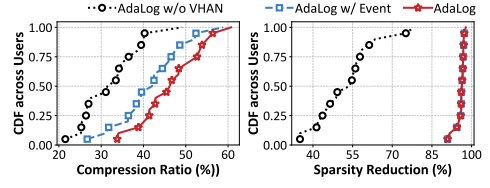
Fig. 18. Effect of VHAN in reducing storage overhead and minimizing storage sparsity.

were only achievable by merging an odd number of groups simultaneously (e.g., three, five, or seven). However, the conditions for this theoretical sub-optimality are demonstrably extremely rare in real-world behavior logs. Our empirical analysis confirms that features rarely align in the complex, odd groupings that would challenge our algorithm. Across all testing users, we found that only up to 3 out of 100+ user behaviors involve 3, 5, or 7 features sharing highly redundant event rows. This means the maximal gain from optimally merging these rare high-order groups would contribute negligible storage reduction to the overall behavior log.

Behavior-Level Log Splitting. Next, we assess the role of virtually hashed attribute name (VHAN) design in reducing storage sparsity and overall footprint. To quantify its impact, we compare to two modified versions of AdaLog: (1) *AdaLog w/o VHAN*: Uses attribute-count-based log splitting but disables VHAN, storing physical attribute names instead of virtual IDs. (2) *AdaLog w/ Event*: Groups behavior events by shared attribute sets, ensuring that all event rows within a single behavior log file have the same relevant attributes. Figure 18 illustrates the compression ratios and storage sparsity reductions across users. We notice that without VHAN, compression ratio decreases by 14% and storage sparsity increases by 35%. This highlights VHAN's effectiveness in consolidating diverse attributes into a single virtual attribute representation, eliminating null values. Also, the *AdaLog w/ Event* method achieves storage sparsity reduction comparable to VHAN but suffers from 8% lower compression performance, caused by the substantial metadata overhead from managing hundreds of fragmented database files.

Incremental Update Mechanism. Finally, we analyze the system efficiency of AdaLog's incremental update mechanism in adapting outdated behavior log files to new configurations over time. We compare AdaLog with the *reconstruction* method, which rebuilds behavior log files from scratch. Figure 19 shows the time and memory footprints for both methods over a two-week period. Our analysis reveals that as event rows accumulate, full reconstruction leads to linearly increasing execution time and memory consumption, whereas AdaLog's incremental update mechanism remains stable. Compared to full reconstruction, incremental updates achieve a 3.4× to 8.1× speedup and a 2.2× to 2.5× reduction in memory consumption. By efficiently updating only event rows affected by configuration updates, AdaLog prevents excessive resource consumption, ensuring minimal interference with other important on-device applications.

4.4 Sensitivity Analysis

Given that AdaLog involves few hyperparameters and avoids trial-and-error processes, we focus on evaluating its performance under environmental factors such as timeline and model numbers.

Impact of Time. Figure 20(a) shows the evolution of behavior log file size over a 14-day period, which includes a public holiday. We observe that behavior log size of the industry-standard solution grows exponentially over time, attributed to two factors: (i) During holidays, users tend to interact more with the applications, resulting in more hot behaviors recorded in behavior log; (ii) New ML
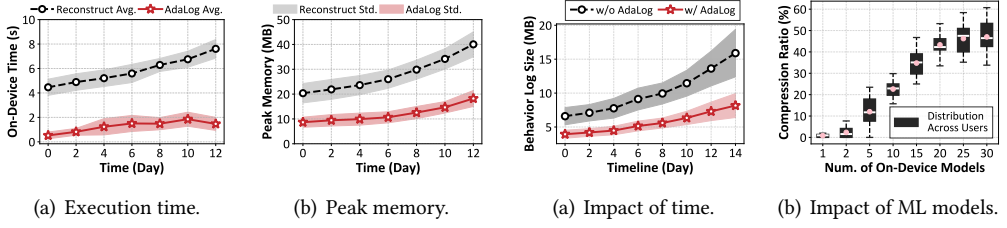
(a) Execution time.　　　　(b) Peak memory.

Fig. 19. System efficiency of incremental updates.

(a) Impact of time.　　　　(b) Impact of ML models.
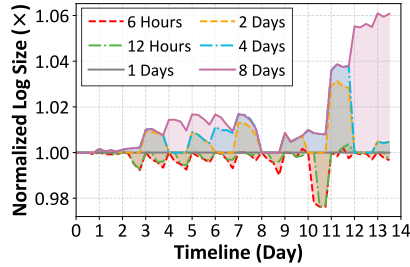
Fig. 20. Sensitivity analysis on natural factors.

Fig. 21. Impact of update frequency on AdaLog's storage efficiency. For better comparison, behavior log sizes are normalized against the size achieved under a standard daily update schedule.

models are deployed on mobile devices to test new application services, leading to more recorded event rows. In contrast, AdaLog exhibits a nearly linear growth in storage size, which is significantly slower than the original design. This trend is directly attributed to AdaLog's feature-level data merging technique, effectively reducing redundancy for hot behaviors. As a result, AdaLog is more efficient in managing long-term storage, even during periods of increased user activity and model deployment.

Impact of Model Number. Next, we explore how the performance of AdaLog is affected by the number of ML models deployed within an application. For this analysis, we conducted experiments where testing users are forced to generate behavior log files for varying numbers of on-device models. Figure 20(b) presents the relation between the number of models and the achieved compression ratio. As the number of models increases from 2 to 20, the compression ratio achieved by AdaLog rises rapidly, from 2% to 43%. This demonstrates that AdaLog scales effectively as more models are deployed on the device, efficiently reducing storage overhead. After the number of models exceeds 25, the compression ratio gradually stabilizes at around 45%. The primary reason is that with the increase of on-device models, newly deployed model is more likely to have input features with identical filtering conditions (i.e., the same FilterIDs) with previous models, implying that less redundant data is introduced.

Impact of Update Frequency. Further, we conduct experiments to analyze the impact of varying the log update frequency on AdaLog's performance. We measured the average log size across testing users, varying the update intervals from 6 hours up to 8 days. As shown in Figure 21, we plot the normalized behavior log sizes achieved by AdaLog across various update frequencies. All sizes are normalized to the baseline value recorded when the log is updated daily for better visualization. We notice that increasing the update frequency beyond one day (e.g., 6 hours and 12 hours) only results in a marginal improvement in compression, typically less than 2.2%. Slowing

down the frequency from one day to 4 and 8 days makes the behavior log gradually drifts from the optimal storage configuration, leading to a noticeable increase in log size of 4%-6%. Based on these findings, we propose that the practical update process should be executed every 1 to 4 days, which can be dynamically controlled by factors such as user activity profile and available device hardware resources.

5 Related Work

Resource-Efficient On-Device ML. Extensive research is conducted to optimize resource utilization for on-device ML through two main levels. System-level optimization directly improves resource efficiency, such as reducing memory footprint [14, 40, 41, 72, 73], enhancing computational hardware utilization [7, 29, 72, 76] and minimizing energy consumption [14, 35, 40]. Model-level optimization compresses on-device models to make inferences more efficient, using techniques such as quantization [34, 45, 51], pruning [53, 54, 58, 68, 75], and sparsification [5, 53]. Notably, most existing research overlooks storage as a critical resource. This is because they focused on traditional CV and NLP models that use static features, which do not require storing massive raw data for computing dynamic features. Consequently, our work is complementary to them by improving storage efficiency for ML models in modern mobile applications.

Input Filtering for Edge Computing. Input filtering aims to reduce unnecessary computation by filtering redundant or irrelevant inputs. Examples include raw data (e.g., undecoded packets [81]), entire input features (e.g., frames [8, 25, 26, 42]), and partial features (e.g., pixels [30, 80]). In this sense, our work can be seen as a new form of raw data filtering tailored for storage optimization rather than computation reduction, where we adaptively filter out redundant and null data in user behaviors recorded in application logs.

Data Management for Mobile Apps. In the area of mobile data management, existing works mainly fall into two categories. The first category optimizes data I/O costs during app usage, i.e., faster data writing and reading. Common optimization techniques include virtual page writing [62], SSD access optimization [43, 70], cache policy design [69], etc. The second category targets storage optimization for time-series and sensor data, such as Apple HealthKit [59] and Samsung Health [31] aggregating and compressing IMU data collected from mobile device. We observe that few existing works consider the storage cost of user behavior *events*, a commonly seen data format in mobile apps, because traditional apps simply upload them to cloud server for centralized storage. However, with the prevalence of intelligent app services supported by on-device ML models, keeping an on-device behavior log introduces an inevitable storage bottleneck, which is the focus of our work.

Database Optimization Techniques. In the broader database community, redundancy and sparsity have traditionally been mitigated through structural and operational optimizations. Columnar storage formats like Apache Parquet [28] and ORC [16] reduce redundancy by grouping similar values and applying compression methods to encode attributes into smaller symbols [2, 38, 56]. Sparse indexing [44] further optimizes storage by selectively indexing non-redundant rows. However, these database-centric approaches focus on structural and system-level efficiencies without optimizing original storage content, and require customized modifications to the database backend. In contrast, AdaLog tackles redundancy and sparsity at the data content layer and offers a more flexible and generalizable solution that can seamlessly integrate with existing database implementations and optimizations.

6 Conclusion

In this work, we identified an overlooked behavior log storage bottleneck in ML-embedded mobile apps, which poses a significant challenge to development and broader deployment of on-device ML models. To address this, we proposed AdaLog system to enhance data storage efficiency by

adaptively reducing redundancy and sparsity in behavior log, without compromising on-device inference accuracy or latency. Extensive evaluations across real-world mobile users and application scenarios demonstrate that AdaLog effectively reduces behavior log storage overhead with minimal system impact, providing a powerful and efficient data foundation for on-device ML deployment.

Acknowledgments

References

[1] 2025. App uninstall report – 2025 edition. https://www.appsflyer.com/resources/reports/app-uninstall-benchmarks/.
[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SigMod)*. 671–682.
[3] Mário Almeida, Stefanos Laskaridis, Abhinav Mehrotra, Lukasz Dudziak, Ilias Leontiadis, and Nicholas D. Lane. 2021. Smart at what cost?: characterising mobile deep neural networks in the wild. In *ACM Internet Measurement Conference (IMC)*. 658–672.
[4] Bert Besser and Matthias Poloczek. 2017. Greedy matching: Guarantees and limitations. *Algorithmica* 77, 1 (2017), 201–234.
[5] Sourav Bhattacharya and Nicholas D Lane. 2016. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *ACM Conference on Embedded Networked Sensor Systems (Sensys)*. 176–189.
[6] Andreas Brandstädt and Raffaele Mosca. 2018. Maximum weight independent set for ℓ claw-free graphs in polynomial time. *Discrete Appled Mathematics* 237 (2018), 57–64.
[7] Qingqing Cao, Niranjan Balasubramanian, and Aruna Balasubramanian. 2017. MobiRNN: Efficient Recurrent Neural Network Execution on Mobile GPU. In *Proceedings of the 1st International Workshop on Embedded and Mobile Deep Learning (Deep Learning for Mobile Systems and Applications)*. 1–6.
[8] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. 2015. Glimpse: Continuous, real-time object recognition on mobile devices. In *ACM Conference on Embedded Networked Sensor Systems (Sensys)*. 155–168.
[9] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*. 7–10.
[10] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *ACM Conference on Recommender Systems (RecSys)*. 191–198.
[11] Marek Cygan, Fabrizio Grandoni, and Monaldo Mastrolilli. 2013. How to Sell Hyperedges: The Hypermatching Assignment Problem. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 342–351.
[12] Peter J Denning. 1970. Virtual memory. *ACM Computing Surveys (CSUR)* 2, 3 (1970), 153–189.
[13] Apple Developer. 2023. Maximum build file sizes. https://developer.apple.com/help/app-store-connect/reference/maximum-build-file-sizes/.
[14] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *International Conference on Mobile Computing and Networking (MobiCom)*. 115–127.
[15] Jesse Feiler and Jesse Feiler. 2015. Using SQLite with Core Data (iOS and OS X). *Introducing SQLite for Mobile Developers* (2015), 61–73.
[16] Apache Software Foundation. 2025. The smallest, fastest columnar storage for Hadoop workloads. https://orc.apache.org/.
[17] Kevin P Gaffney, Martin Prammer, Larry Brasfield, D Richard Hipp, Dan Kennedy, and Jignesh M Patel. 2022. SQLite: past, present, and future. *Proceedings of the VLDB Endowment (VLDB)* 15, 12 (2022).
[18] Zhabiz Gharibshah and Xingquan Zhu. 2021. User response prediction in online advertising. *ACM Computing Surveys (CSUR)* 54, 3 (2021), 1–43.

[19] Carlos A Gomez-Uribe and Neil Hunt. 2015. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)* 6, 4 (2015), 1–19.

[20] Chen Gong, Rui Xing, Zhenzhe Zheng, and Fan Wu. 2025. A Two-Stage Data Selection Framework for Data-Efficient Model Training on Edge Devices. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2 (KDD)*. 673–684.

[21] Chen Gong, Zhenzhe Zheng, Yunfeng Shao, Bingshuai Li, Fan Wu, and Guihai Chen. 2024. ODE: An online data selection framework for federated learning with limited storage. *IEEE/ACM Transactions on Networking (TON)* 32, 4 (2024), 2794–2809.

[22] Chen Gong, Zhenzhe Zheng, Fan Wu, Xiaofeng Jia, and Guihai Chen. 2024. Delta: A cloud-assisted data enrichment framework for on-device continual learning. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (MobiCom)*. 1408–1423.

[23] Chen Gong, Zhenzhe Zheng, Fan Wu, Yunfeng Shao, Bingshuai Li, and Guihai Chen. 2023. To store or not? online data selection for federated learning with limited storage. In *Proceedings of the ACM Web Conference (WWW)*. 3044–3055.

[24] Google. [n. d.]. Android Developers: APK Expansion Files. https://developer.android.com/google/play/expansion-files.

[25] Peizhen Guo, Bo Hu, Rui Li, and Wenjun Hu. 2018. Foggycache: Cross-device approximate computation reuse. In *International Conference on Mobile Computing and Networking (MobiCom)*. 19–34.

[26] Peizhen Guo and Wenjun Hu. 2018. Potluck: Cross-Application Approximate Deduplication for Computation-Intensive Mobile Applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 271–284.

[27] Jialiang Han, Yun Ma, Qiaozhu Mei, and Xuanzhe Liu. 2021. DeepRec: On-device deep learning for privacy-preserving sequential recommendation in mobile commerce. In *Proceedings of the Web Conference (WWW)*. 900–911.

[28] Todor Ivanov and Matteo Pergolesi. 2020. The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet. *Concurrency and Computation: Practice and Experience* 32, 5 (2020), e5523.

[29] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. 2022. CoDL: efficient CPU-GPU co-execution for deep learning inference on mobile devices.. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, Vol. 22. 209–221.

[30] Shiqi Jiang, Zhiqi Lin, Yuanchun Li, Yuanchao Shu, and Yunxin Liu. 2021. Flexible high-resolution object detection on edge devices with tunable latency. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (MobiCom)*. 559–572.

[31] Se Young Jung, Jeong-Whun Kim, Hee Hwang, Keehyuck Lee, Rong-Min Baek, Ho-Young Lee, Sooyoung Yoo, Wongeun Song, and Jong Soo Han. 2019. Development of comprehensive personal health records integrating patient-generated health data directly from Samsung S-Health and Apple Health apps: retrospective cross-sectional observational study. *JMIR mHealth and uHealth* 7, 5 (2019), e12691.

[32] Pavan Kamaraju, Pietro Lungaro, and Zary Segall. 2013. A novel paradigm for context-aware content pre-fetching in mobile networks. In *IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 4534–4539.

[33] Shubhra Kanti Karmaker Santu, Parikshit Sondhi, and ChengXiang Zhai. 2017. On application of learning to rank for e-commerce search. In *International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. 475–484.

[34] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In *Proceedings of the Fourteenth EuroSys Conference (EuroSys)*. 1–15.

[35] Young Geun Kim and Carole-Jean Wu. 2020. Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning. In *2020 53rd Annual IEEE/ACM international symposium on microarchitecture (MICRO)*. IEEE, 1082–1096.

[36] Vladimir Kolmogorov. 2009. Blossom V: a new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation* 1 (2009), 43–67.

[37] Jay Kreibich. 2010. *Using SQLite*. " O'Reilly Media, Inc.".

[38] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. Btrblocks: Efficient columnar compression for data lakes. *Proceedings of the ACM on Management of Data (SigMod)* 1, 2 (2023), 1–26.

[39] Fan Lai, Wei Zhang, Rui Liu, William Tsai, Xiaohan Wei, Yuxi Hu, Sabin Devkota, Jianyu Huang, Jongsoo Park, Xing Liu, et al. 2023. {AdaEmbed}: Adaptive embedding for {Large-Scale} recommendation models. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 817–831.

[40] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In *International Conference on Information Processing in Sensor Networks (IPSN)*. 23:1–23:12.

[41] Xiangyu Li, Yuanchun Li, Yuanzhe Li, Ting Cao, and Yunxin Liu. 2024. Flexnn: Efficient and adaptive dnn inference on memory-constrained edge devices. In *International Conference on Mobile Computing and Networking (MobiCom)*.

709–723.

[42] Yuanqi Li, Arthi Padmanabhan, Pengzhan Zhao, Yufei Wang, Guoqing Harry Xu, and Ravi Netravali. 2020. Reducto: On-Camera Filtering for Resource-Efficient Real-Time Video Analytics. In *Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 359–376.

[43] Yu Liang, Cheng Ji, Chenchen Fu, Rachata Ausavarungnirun, Qiao Li, Riwei Pan, Siyu Chen, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. 2020. iTRIM: I/O-aware TRIM for improving user experience on mobile devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 40, 9 (2020), 1782–1795.

[44] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, Peter Camble, et al. 2009. Sparse indexing: Large scale, inline deduplication using sampling and locality.. In *Fast*, Vol. 9. 111–123.

[45] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems (MLsys)* 6 (2024), 87–100.

[46] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. 2015. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *IEEE Annual International Computer Software and Applications Conference (COMPSAC)*, Vol. 2. 422–433.

[47] Cihang Liu, Lan Zhang, Zongqian Liu, Kebin Liu, Xiangyang Li, and Yunhao Liu. 2016. Lasagna: Towards deep hierarchical understanding and searching over mobile sensing data. In *Annual International Conference on Mobile Computing and Networking (MobiCom)*. 334–347.

[48] Haibo Liu, Chen Gong, Zhenzhe Zheng, Shengzhong Liu, and Fan Wu. 2025. Enabling Real-Time Inference in Online Continual Learning via Device-Cloud Collaboration. In *Proceedings of the ACM on Web Conference (WWW)*. 2043–2052.

[49] Haibo Liu, Da Huo, Zhenzhe Zheng, and Fan Wu. 2025. Latency-Aware Online Continual Learning for Non-Stationary Data Streams. In *IEEE INFOCOM 2025-IEEE Conference on Computer Communications (INFOCOM)*. 1–10.

[50] Haibo Liu, Zhenzhe Zheng, Fan Wu, and Guihai Chen. 2025. From Non-IID to IID: Mobility-aware Hierarchical Federated Learning with Client-Edge Association Control. *IEEE Transactions on Mobile Computing (TMC)* (2025).

[51] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. 2018. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 389–400.

[52] Vadim V Lozin and Martin Milanič. 2008. A polynomial algorithm to find an independent set of maximum weight in a fork-free graph. *Journal of Discrete Algorithms* 6, 4 (2008), 595–604.

[53] Sangkug Lym, Esha Choukse, Siavash Zangeneh, Wei Wen, Sujay Sanghavi, and Mattan Erez. 2019. Prunetrain: fast neural network training by dynamic sparse model reconfiguration. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–13.

[54] Xiaolong Ma, Fu-Ming Guo, Wei Niu, Xue Lin, Jian Tang, Kaisheng Ma, Bin Ren, and Yanzhi Wang. 2020. Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices. In *Annual AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 34. 5117–5124.

[55] Abbas Mehrabi, Matti Siekkinen, and Antti Ylä-Jääski. 2018. Edge computing assisted adaptive mobile video streaming. *IEEE Transactions on Mobile Computing (TMC)* 18, 4 (2018), 787–800.

[56] Ingo Müller, Cornelius Ratsch, Franz Färber, et al. 2014. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems.. In *EDBT*, Vol. 14. 283–294.

[57] Fionn Murtagh and Pedro Contreras. 2012. Algorithms for hierarchical clustering: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 2, 1 (2012), 86–97.

[58] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. 2020. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 907–922.

[59] Frederick North and Rajeev Chaudhry. 2016. Apple healthkit and health app: patient uptake and barriers in primary care. *Telemedicine and e-Health* 22, 7 (2016), 608–613.

[60] Emil Numminen, Henrik Sällberg, and Shujun Wang. 2022. The impact of app revenue model choices for app revenues: A study of apps since their initial App Store launch. *Economic Analysis and Policy* 76 (2022), 325–336.

[61] Nikola Obradovic, Aleksandar Kelec, and Igor Dujlovic. 2019. Performance analysis on Android SQLite database. In *International Symposium INFOTEH-JAHORINA (INFOTEH)*. IEEE, 1–4.

[62] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. 2015. SQLite optimization with phase change memory for mobile applications. *Proceedings of the VLDB Endowment (VLDB)* 8, 12 (2015), 1454–1465.

[63] Yi Ouyang, Bin Guo, Qianru Wang, Yunji Liang, and Zhiwen Yu. 2022. Learning dynamic app usage graph for next mobile app recommendation. *IEEE Transactions on Mobile Computing (TMC)* 22, 8 (2022), 4742–4753.

[64] Marius-Constantin Popescu, Valentina E Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. 2009. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems* 8, 7 (2009), 579–588.

[65] J. Ross Quinlan. 1996. Learning decision tree classifiers. *ACM Computing Surveys (CSUR)* 28, 1 (1996), 71–72.

[66] Paolo Roma and Daniele Ragaglia. 2016. Revenue models, in-app purchase, and the app performance: Evidence from Apple's App Store and Google Play. *Electronic commerce research and applications* 17 (2016), 173–190.

[67] Iqbal H Sarker, Mohammed Moshiul Hoque, Md Kafil Uddin, and Tawfeeq Alsanoosy. 2021. Mobile data science and intelligent apps: concepts, AI-based modeling and research directions. *Mobile Networks and Applications* (2021), 285–303.

[68] Leming Shen, Qiang Yang, Kaiyan Cui, Yuanqing Zheng, Xiao-Yong Wei, Jianwei Liu, and Jinsong Han. 2024. Fedconv: A learning-on-model paradigm for heterogeneous federated clients. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 398–411.

[69] Zhaoyan Shen, Lei Han, Renhai Chen, Chenlin Ma, Zhiping Jia, and Zili Shao. 2020. An Efficient Directory Entry Lookup Cache With Prefix-Awareness for Mobile Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 39, 12 (2020), 4575–4586.

[70] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Prism: Optimizing key-value store for modern heterogeneous storage devices. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 588–602.

[71] Tuyen X Tran and Dario Pompili. 2018. Adaptive bitrate video caching and processing in mobile-edge computing networks. *IEEE Transactions on Mobile Computing (TMC)* 18, 9 (2018), 1965–1978.

[72] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. 2021. AsyMo: scalable and efficient deep-learning inference on asymmetric mobile CPUs. In *International Conference on Mobile Computing and Networking (MobiCom)*. 215–228.

[73] Qipeng Wang, Mengwei Xu, Chao Jin, Xinran Dong, Jinliang Yuan, Xin Jin, Gang Huang, Yunxin Liu, and Xuanzhe Liu. 2022. Melon: breaking the memory wall for resource-efficient on-device machine learning. In *International Conference on Mobile Systems, Applications and Services (MobiSys)*. 450–463.

[74] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. Autodroid: Llm-powered task automation in android. In *International Conference on Mobile Computing and Networking (MobiCom)*. 543–557.

[75] Hao Wen, Yuanchun Li, Zunshuai Zhang, Shiqi Jiang, Xiaozhou Ye, Ye Ouyang, Yaqin Zhang, and Yunxin Liu. 2023. AdaptiveNet: Post-deployment Neural Architecture Adaptation for Diverse Edge Environments. In *International Conference on Mobile Computing and Networking (MobiCom)*. 28:1–28:17.

[76] Daliang Xu, Mengwei Xu, Qipeng Wang, Shangguang Wang, Yun Ma, Kang Huang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2022. Mandheling: mixed-precision on-device DNN training with DSP offloading. In *International Conference on Mobile Computing and Networking (MobiCom)*. 214–227.

[77] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. 2019. A First Look at Deep Learning Apps on Smartphones. In *The ACM Web Conference (WWW)*. 2125–2136.

[78] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. 2018. Neural adaptive content-aware internet video delivery. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 645–661.

[79] Jinliang Yuan, Chen Yang, Dongqi Cai, Shihe Wang, Xin Yuan, Zeling Zhang, Xiang Li, Dingge Zhang, Hanzi Mei, Xianqing Jia, et al. 2024. Mobile foundation model as firmware. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (MobiCom)*. 279–295.

[80] Mu Yuan, Lan Zhang, Fengxiang He, Xueting Tong, and Xiang-Yang Li. 2022. InFi: End-to-end learnable input filter for resource-efficient mobile-centric inference. In *International Conference on Mobile Computing and Networking (MobiCom)*. 228–241.

[81] Mu Yuan, Lan Zhang, Xuanke You, and Xiang-Yang Li. 2023. Packetgame: Multi-stream packet gating for concurrent video inference at scale. In *Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. 724–737.

[82] Yongfeng Zhang, Xu Chen, Qingyao Ai, Liu Yang, and W Bruce Croft. 2018. Towards conversational search and recommendation: System ask, user respond. In *ACM International Conference on Information and Knowledge Management (CIKM)*. 177–186.

[83] Christos Ziakis, Maro Vlachopoulou, Theodosios Kyrkoudis, and Makrina Karagkiozidou. 2019. Important factors for improving Google search rank. *Future Internet* 11, 2 (2019), 32.

A Diverse User Behaviors

We list the top 50 user behaviors in industrial mobile apps in the following 4 tables for justification, which are partially modified to satisfy confidential requirements.

Table 1. Top 1-25 Behaviors.

0	play_time
1	video_play
2	show
3	page_show
4	homepage_slide_up
5	play_session
6	live_show
7	trending_words_show
8	video_finish
9	click_comment
10	video_stop
11	stay
12	like
13	live_window_show
14	show_product
15	ad_gap
16	othershow
17	enter_room_duration
18	live_window_duration
19	room_not_render
20	play
21	trending_show
22	silence_launch_app
23	video_pause
24	back_quit

Table 2. Top 26-50 Behaviors.

25	live_duration
26	inner_push
27	wormhole_preview
28	enter_personal_detail
29	homepage_notice
30	search_result_show
31	homepage_slide_down
32	performance_monitor
33	homepage_tab_stay_time
34	enter_homepage
35	wormhole_preview_reuse
36	live_play
37	enter_homepage_message
38	search
39	show_product_v2
40	product_entrance_show
41	adjust_volume
42	product_entrance
43	post_comment
44	search_result_click
45	show_card
46	share
47	enter_tab
48	search_click
49	click_product