IPS: Unified Profile Management for Ubiquitous Online Recommendations

Rui Shi, Yang Liu, Jianjun Chen, Xuan Zou, Yanbin Chen, Minghua Fan, Zhihao Cai
Guanghui Zhang, Zhiwen Li, Yuming Liang
ByteDance, Inc.
ips-paper@bytedance.com

***Abstract*—ByteDance offers several massively popular products such as TikTok, Jinri Toutiao and Douyin for creating, sharing and discovering a variety of content, in which recommendation plays an indispensable role for helping billions of users to interact with highly personalized content. The personalized experience in products largely comes from the ability of sophisticated machine learning models to make accurate predictions based on users' interests and one key component in such systems is the user profile service.**

In this paper, we introduce Instance Profile Service (IPS), a large scale distributed system for managing unstructured profile data as well as serving various feature computations at ByteDance. Different products leverage IPS in many different ways and place various demands on the system, in terms of complex computation logic and latency requirements. One major challenge in the design of a large scale user profile system is how to strike the right balance among efficiency, scalability, reliability and versatility. With deliberated choices made on its design and implementation, we demonstrate IPS can provide a simple yet flexible solution to all these products while meeting the targeted high availability and performance goals. At ByteDance, IPS has successfully replaced many legacy profile systems and runs on thousands of machines. One of our largest production instances can process a hundred million feature queries and tens of millions writes per second.

***Index Terms*—Feature Management, Feature Serving, Recommendation System**

I. Introduction

In recent years, personalized recommendations have become vital in providing relevant content, such as news feed, in consumer oriented internet services. A typical recommendation process includes the following two stages:

- Retrieve from multiple content repositories to get a candidate set of contents based on a user's interests.
- Rank the candidate set above based on relevance scores predicted by machine learning models then return the top K most relevant results to the user.

As matching user interests to content items is the core of the above process, the profile service has played a key role in a recommendation system. We consider a good profile service has the following requirements:

a) Capture both long and short term interests: When making recommendations to users, a news feed should not merely show recent hot topics, but also understand users' long-term interests and hobbies.

(a) TikTok (b) Jinri Toutiao (c) Douyin

Fig. 1. Screenshots of ByteDance's most popular products with hundreds of millions Daily Active Users (DAU) each.

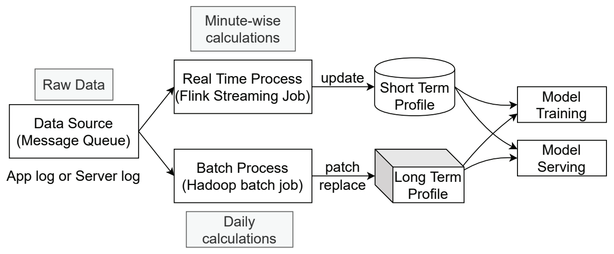

Fig. 2. Short and long term profile services as a lambda architecture.

b) Flexible in feature computation: Features used for model training and prediction should be readily computed in service to various feature engineering needs.

ByteDance offers several products for creating, sharing and discovering a variety of contents, in which recommendation plays an indispensable role for helping billions of users to interact with highly personalized contents from an ever-changing corpus. The personalized experience in products such as TikTok [1], Jinri Toutiao (a.k.a Today's Headline) [2] and Douyin (the Chinese version of TikTok) as shown in Figure 1 largely comes from the ability of sophisticated machine learning models to make accurate predictions based on users' interests. Over the last two years, we have designed and implemented a distributed system named Instance Profile Service (IPS) for managing profile data, which serves as one major source of features fed to model training and inference at ByteDance.

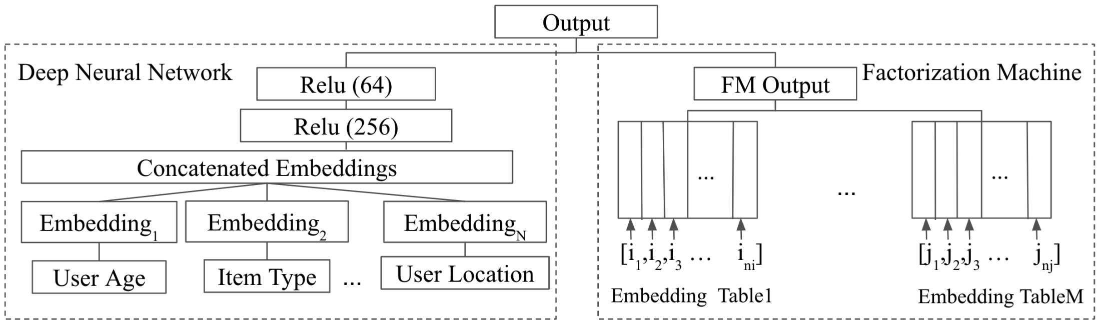

Fig. 3. A simplified model structure for a wide and deep recommendation model.

In the early practice of ByteDance's recommendation system, the user profile service was divided into two independent ones: *Long Term Profile* and *Short Term Profile*, depicted as a Lambda architecture [3] in Figure 2:

- **Long Term Profile** focuses on the long term behaviors or profiles of users, e.g. the topmost content category clicked by a user over the last year. The actual implementation keeps each user's profile in a key-value store mapping from the profile ID to the top features that user clicked or viewed during the entire history. Due to the large amount of historical data that needs to be processed, the *Long Term Profile* can not be updated in real time. A daily offline batch job processes the previous day's logs then updates the long term profile with the most recent data.
- **Short Term Profile** focuses on the short-term interests of users. Unlike the *long term profile*, only the content IDs of the user's most recent clicks are stored. While serving a request, the list of content IDs clicked by the user are used to retrieve the detailed categorical information from a content data store. This step is simply a key to ID list mapping, which relies on the upstream service to construct the input features for model scoring.

However, with the rapid evolution of the real-time recommendation system as well as the massive growth in data volume, we found some major problems in this early practice. Firstly, every product needs to maintain two separate profile services which serve similar purposes. In addition, each profile service has its own dependencies to compute features, which further complicates the operations in production. Secondly, engineers of each product need to implement customized logic to retrieve additional information then compute the input features for machine learning models, making it hard to reuse the feature computation logic across multiple products. Lastly, as only limited types (long and short) of time window are supported, the existing system is not able to flexibly control the time span over the history data. As a result, it is highly non-trivial to experiment with new features such as the aggregated statistics of user actions over last week or last 30 days. To address the above problems, the design of IPS aims to provide a general service to replace the various profile services within the company, which reduces the overall maintenance overhead, increase the engineering flexibility and improves resource efficiency.

As for recommendation systems, wide and deep models [4]–[6] are proven to be effective in recommendation systems. At ByteDance, we employ different extensions of such models as shown in 3 to improve user engagements. One important characteristic of the models is extensive use of sparse features and embeddings [7]–[9] together with deep neural networks. The number of features are often huge for the categorical variables. As an example, for video-side features, the feature size is the total number of videos in our database, which can range up to tens of billions in a single model. With the help of IPS, we can extract thousands of features for a single request, assemble them for serving and flush them into training data in parallel to avoid training-serving skew. In the following, we provide some insights into how different products are leveraging IPS by delving into more details for two major use cases:

c) Content Feeds: We heavily rely on IPS as the hub for feature extraction. One main advantage of IPS is its ability to capture both short and long term features. Short term features allow the app to quickly promote the trendy content. For instance, most models rely on the number of clicks and the click through rate (CTR) as input features, so quickly updated values are crucial for recommending contents like breaking news. The long term features enable the app to better understand the latent characteristics of articles and users. For example, if a user started reading about cooking but then switched to hiking, the model may recommend some trail cooking recipes based on this.

d) Advertising: While IPS in advertising shares similar benefits with content feeds, it also has some unique challenges. Flow control is very important in online advertising to maximize the potential conversions and profits. IPS is able to capture features like impressions and conversions responsively, which delivers a smooth targeting of ads over a certain period of time. The price is also critical since our models rely on it heavily to determine the advertisement's value. As today's online advertising is often auction based, the bidding price is very sensitive and volatile. IPS is able to update the bidding prices in a timely manner to help delivering the most valuable ads, which is crucial to our business.

In summary, our key contributions are as follows:

- We introduce a novel yet general approach to designing profile services that power feature computations for large scale online recommendation systems.
- We describe a time-series based data model that supports complex feature computations such as muti-dimensional top K query and user defined aggregate functions over arbitrary time windows, which allows sufficient flexibility for feature engineering.
- We present a highly scalable and geo-replicated architecture that is resilient to various data center and network failures.
- We describe key techniques implemented to sustain tens of millions queries per second, achieving high update throughput, data freshness and low query latency in an integrated manner.
- We describe some implementation aspects of this approach on a large scale, our experience to resolve certain challenges, its limitations as well as trade-offs made to meet the targeted high availability and performance goals.

The rest of paper is organized as follows: Section II describes the data model of IPS in more detail and provides an overview of the APIs; Section III describes the overall architecture of the system as well as implementations of its key components along with optimizations to resolve challenges we faced in production; in Section IV, we provide some empirical measurements of IPS' scale and performance; Finally, Section VI describes the related work, and Section VII concludes our work.

II. Data Model and APIs

In this section, we describe IPS' data model and APIs using motivating examples.

A. Data Model

IPS, as a generic profile service, is modeled after time, category, action and feature stats. This design can effectively characterize a user's interests over time and also be used to store various application specific data with sufficient flexibility, which provides enough expressiveness to meet most application needs at ByteDance.

Let's illustrate using a motivating example: after a user named Alice was recommended some cool contents by a mobile app, she watched a few short videos in it. She 'liked' and 'commented' on one video about Los Angeles Lakers then re-shared this video to her friends. A few days later, she viewed a couple of videos about Golden State Warriors and 'liked' some of them. These interest related signals such as liked basketball teams are aggregated and anonymously stored in IPS with rigorous privacy controls as Alice continues to use the mobile app.

Our recommendation engine would query the aggregated interest related stats every time it is suggesting new contents to a user. The stored stats in IPS are used to create features such as the *CTR of basketball related contents in the last 30 days*, which are fed into a machine learning model to make recommendations to the user.

```
SELECT
  feature,
  SUM(like) AS total_likes
FROM
  user_profile_table
WHERE
  uid = "Alice" AND
  timestamp > TEN_DAYS_AGO AND
  slot = "Sports" AND type = "Basketball"
GROUP BY feature
ORDER BY total_likes DESC
LIMIT 1;
```

Listing 1. A SQL example to illustrate feature queries computed by IPS.

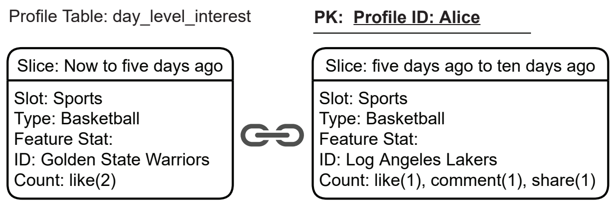
Profile Table: day_level_interest
PK: Profile ID: Alice
Slice: Now to five days ago
Slot: Sports
Type: Basketball
Feature Stat:
ID: Golden State Warriors
Count: like(2)
Slice: five days ago to ten days ago
Slot: Sports
Type: Basketball
Feature Stat:
ID: Log Angeles Lakers
Count: like(1), comment(1), share(1)

Fig. 4. IPS data model.

To facilitate understanding, we layout the above example's data in I, where Alice's two actions correspond to two rows as part of her profile. As an example, a typical pattern computed by IPS can be expressed as a traditional SQL query in Listing 1, which returns Alice's most favorite basketball team over the last 10 days as a feature, which is *Golden State Warriors*. This is one of the most common compute patterns that IPS deals with and the key challenge is to design an efficient architecture that can scale to serve billions of users online.

We explain in the following how the above motivating example is represented in IPS data model, as illustrated in Figure 4. Alice's profile data, along with other users, are kept in a *Profile Table* in which each profile is keyed by a unique ID. A user's entire profile is recorded in a time-serial list of *Slices*, each of which represents a piece of profile history within a non-overlapping time range. Features, typically with a unique feature ID, are categorized into various *Slots and Types* (e.g. Los Angeles Lakers could be a feature, which belongs to Sports Slot and Basketball Type) and associated with a vector of counts (e.g. number of clicks, shares and comments about Lakers). Based on this data model, IPS can readily answer feature queries, such as "Alice's topmost liked feature in Sports and Basketball category over the last 10 days?", which is "Golden State Warriors". Note that the examples above shows actual textual information is purely for illustration purpose. In reality, all user profile data are stored as hashed literals along with strict privacy and access controls.

Recommendation systems relying on IPS often retrieve 10s to 100s features every time they render personalized contents to a user. In each request, feature counts are collected from different categories and aggregated in varied window sizes. Due to the real-time nature of online recommendation, the computed features can not be cached effectively but have to be computed every time.

TABLE I
AN ILLUSTRATION OF OUR MOTIVATING EXAMPLE IN A DATABASE TABLE.

uid	timestamp	feature	slot	type	like	comment	share
Alice	Ten days ago	Los Angeles Lakers	Sports	Baketball	1	1	1
Alice	Two days ago	Golden State Warriors	Sports	Baketball	2	0	0

B. APIs

IPS provides APIs to write, read and manage profile data. The write operations are append or insert without conducting in-place update while the read operations are primarily focused on efficiently retrieving relevant data according to the conditions specified by the application layer. The management operations are mostly internal for online operations so we omit them for brevity.

1) Write APIs: As described earlier in this section, the conceptual data model of IPS is a time-serial linked list of multi-level hash tables, in which profile data are stored in a strict time order. The most commonly used write operations are:

- **add_profile(table, profile_id, timestamp, slot, type, feature_id, feature_counts)**
- **add_profiles(table, profile_id, timestamp, slot, type, feature_id*, feature_counts*)**

The *table* indicates the IPS table the request is written to. The *timestamp* is used to determine where the profile data should be placed in the time-serial list of the entire profile. If the *timestamp* is greater than the most recent data of the current profile, a new Slice will be placed at the beginning of the list. The *slot* and *type* are used as indexes to locate the actual feature data: *feature_id* and its corresponding *feature_counts*. The second interface is the batched version of the first one to improve the write efficiency.

2) Read APIs: The read interfaces of IPS retrieve relevant profile data with additional sorting or filtering operations based on application specified criterion. Consider the example in Figure 4, in order to find Alice's favorite sports teams in the last ten days, an application needs to query IPS for the sports slot and find the top 1 team 'liked' by her. The query processing can be roughly divided into two steps: First, locates the *Slices* needed for the computation based on the given time range; Second, performs a multi-path merge and aggregation over all features under the 'Sports' slot followed by a top K search to return the final results.

IPS supports common query operations as follows:

- **get_profile_topK(table, profile_id, slot, type, time_range, sort_type, k)** returns the top K features sorted by the specified *sort_type* over the specified time range, where the *sort_type* includes: sort by a certain attribute count (e.g. 'likes', 'comments'), timestamp or feature id etc.
- **get_profile_filter(table, profile_id, slot, type, time_range, filter_type)** returns the features filtered by a certain type over the specified time range.
- **get_profile_decay(table, profile_id, slot, type, time_range, decay_function, decay_factor)** returns the features over the specified time range after applying the specified *decay_function* to the feature counts with the *decay_factor* at different time scale. This API is most useful when an application wants to favor the most recent profile data over the old ones.

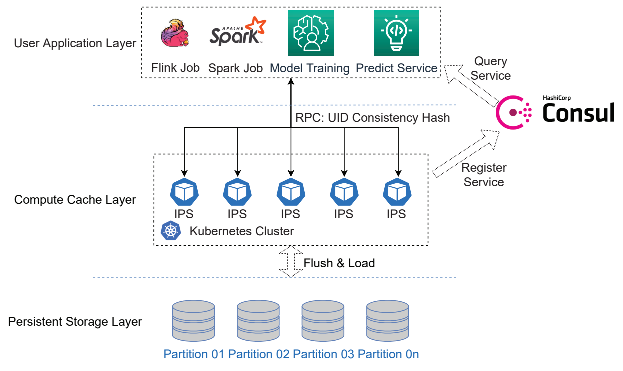

Fig. 5. High-level architecture of IPS

In all APIs above, a time range has to be specified, which is used to determine the *Slices* that fall into the range. There are three kinds of time ranges supported: CURRENT, RELATIVE, ABSOLUTE. The CURRENT time range denotes the queried window should end at the current moment. The RELATIVE time range denotes the queried window should start from when a most recent action happens. The ABSOLUTE time range is used to specify arbitrary time window in history. The *slot* and *type* are required to narrow down the search space of feature counts.

Once the *Slices* of interest are collected, IPS performs a multi-way merge and aggregation over all feature counts. As part of this process, the pre-configured aggregate function and the specified decay function may be applied. A final sort is then conducted on the aggregated feature counts according to the specified attributes (e.g. sort by thumb-ups, by shares or by clicks). It is a common practice for an upstream application to use different combinations of filtering, sorting and decaying to produce features used for various recommendation scenarios.

III. SYSTEM ARCHITECTURE

Figure 5 shows the high-level architecture of IPS, which can be divided into three layers: the *user application layer*, the *compute cache layer*, and the *persistent storage layer*. The top layer includes various applications such as the Flink streaming processing jobs that are responsible for ingesting data at real time, the Spark batch processing jobs that are responsible for importing data in bulk mode, the model training jobs as well as the ranking service that extracts features from IPS for online training and prediction. The middle layer is the core compute and cache layer of IPS, which conducts all the feature computations. The upstream user applications rely on a unified IPS client to communicate with this layer. IPS uses an internal C++ Thrift [10] based RPC framework for communications between different layers. We use ID-based *Consistent Hash* for load balancing and *Consul* for service discovery. IPS instances register the IP and port with Consul when the service is ready and the upstream clients refresh the IPS instance list from Consul periodically. Each IPS instance serves a fraction of the data in the cluster, which allows the system to scale horizontally. As the cache layer resides in memory, we rely on a high performance distributed key-value store like HBase to provide data durability in case of fatal failures.

A. Data Ingestion

Instance data is the basic data source for model training in machine learning, which also serves as the major data source for IPS. Instance data include three input sources: impression, action and features from different data streams. The impression data represents the actual presentation of an article or video to the user, which includes the server impression and the client impression. The action data represents the actions performed by users such as 'like' or 'comment'. The feature data comes from the back end servers, which include all kinds of signals used for recommendation and ranking. Various Flink streaming jobs are used to join the data from the above streams to form the instance data as training samples. The joined instance is then written to the corresponding Kafka topics for downstream consumption. Finally, one Flink Streaming job with user defined extraction logic consumes the Instance data from Kafka and ingests into the corresponding IPS instances. The end-to-end latency between a user's action and the data being available in IPS in a normal data flow path is usually within a minute.

B. In-memory Data Structures

The in-memory data structures employed in IPS are key to IPS's flexible feature extraction and high performance. In general, the internal storage structure of the compute cache layer in IPS can be thought as a time-serial list embedded with multi-layer hash maps. The time-serial list allows flexible time window query and the embedded multi-layer hash maps support fast feature querying. Figure 6 shows the memory data structure in the compute cache layer, which includes four major data structures: *Profile Table*, *Slice*, *Instance Set* and *Indexed Feature Stat*. Those concepts are briefly introduced in Section II and below we explain them in more technical details:

- *Profile Table* is the concept for logically organizing different profile data in IPS. Data in different tables are stored separately. In each table, profiles are uniquely identified by a 64-bit unsigned integer (*Profile ID*). The basic data structure of *Profile Table* is an unordered map in which the key is the profile ID and the value is a *Profile Data*, which contains a list of *Slice*.
- *Slice* stores users' feature behavior in a series of time intervals. Each *Slice* represents a snapshot of the user actions over a period of time and a list of *Slice* constitutes the entire profile history. Each profile data instance written to IPS has an associated timestamp, which is used to determine the actual position to place in a slice. The slices are automatically managed as part of the write process and there are background threads to continuously merge the consecutive slices into ones of longer time ranges as discussed in III-D. The basic data structure of *Slice* is an unordered_map in which the key is a slot ID and the value is an *Instance Set*.
- *Instance Set* is another map structure to represent a set of user behaviors across multiple action type defined by upstream applications. The basic data structure of *Instance Set* is an unordered_map in which the key is an action_type ID and the value is an *Indexed Feature Stat*.
- *Indexed Feature Stat* represents the feature statistics in the format of either an int64 pair or a list to satisfy various needs. To support efficient multi-way feature merging and sorting, we also add an fid_index that tracks the index of the current feature in the entire user feature list.

C. Cache Management

In Figure 6, GCache is a write back cache that consists of two lists: the *LRU List* and the *Dirty List*. A set of swap threads periodically check the *LRU List* in GCache and swap out old data from memory based on the LRU strategy. In the mean time, another set of flush threads periodically check the *Dirty List* for the updated or newly written data in GCache then persist them into the key-value store for data persistence.

The effectiveness of the cache management directly affects the overall performance of IPS. As upstream query load getting more intensive, the swap and flush activities may cause periodic fluctuations in CPU load and processing latency, especially in a virtual machine environment. To solve this problem, IPS shards the LRU cache into multiple partitions hashed by the profile ID, which can effectively reduce the lock contention among the swap threads as shown in Figure 7. When the swap threads find that the memory usage exceeds a pre-defined threshold, they will swap out cached entries starting from the largest shard in the LRU cache until the memory usage falls below the threshold. In addition, a swap thread first attempts accessing the last entry with try_lock. If the try_lock fails, the entry must be processed by another thread. Instead of blocking, the current swap thread simply proceeds to process the next entry up in the list, which can further reduce the lock contention.

In figure 9, the *Dirty List* is also divided into multiple shards like the LRU cache except the number of flush threads must be a multiple of total shards, which is to ensure that every dirty list shard is assigned at least one flush thread and minimize the interference between them.

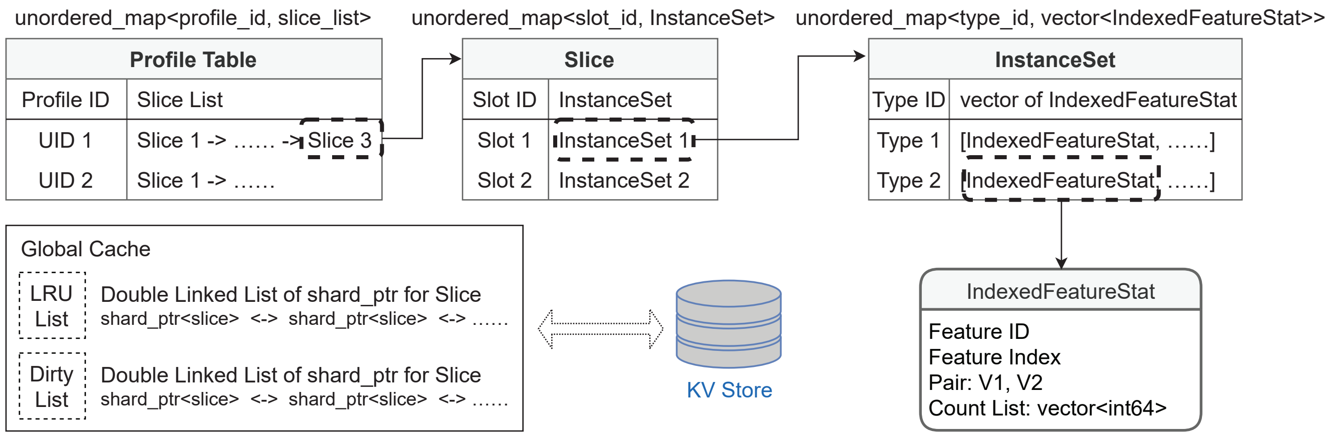
unordered_map<profile_id, slice_list>
unordered_map<slot_id, InstanceSet>
unordered_map<type_id, vector<IndexedFeatureStat>>
Profile Table
Profile ID
Slice List
UID 1
Slice 1 -> ...... -> Slice 3
UID 2
Slice 1 -> ......
Slice
Slot ID
InstanceSet
Slot 1
InstanceSet 1
Slot 2
InstanceSet 2
InstanceSet
Type ID
vector of IndexedFeatureStat
Type 1
[IndexedFeatureStat, ...]
Type 2
[IndexedFeatureStat, ...]
Global Cache
LRU List
Double Linked List of shard_ptr for Slice
shard_ptr<slice> <-> shard_ptr<slice> <-> ......
Dirty List
Double Linked List of shard_ptr for Slice
shard_ptr<slice> <-> shard_ptr<slice> <-> ......
KV Store
IndexedFeatureStat
Feature ID
Feature Index
Pair: V1, V2
Count List: vector<int64>

Fig. 6. IPS in-memory data structures

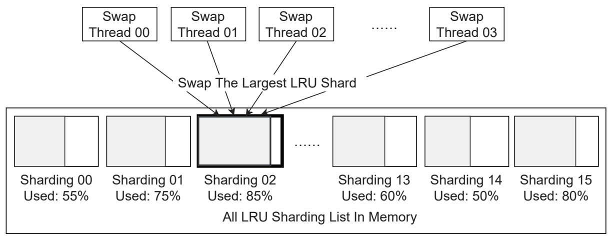

Fig. 7. Sharded LRU list in memory

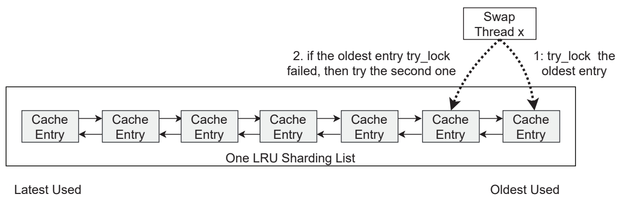
Swap Thread x
2. if the oldest entry try_lock failed, then try the second one
1: try_lock the oldest entry
Cache Entry
One LRU Sharding List
Latest Used
Oldest Used

Fig. 8. Access cache entry with try_lock

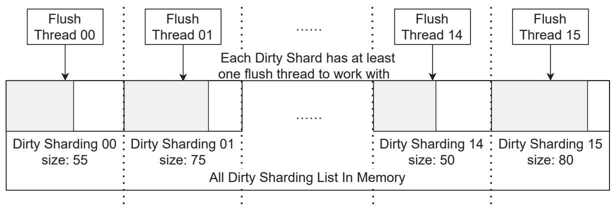
Flush Thread 00
Flush Thread 01
......
Flush Thread 14
Flush Thread 15
Each Dirty Shard has at least one flush thread to work with
......
Dirty Sharding 00 size: 55
Dirty Sharding 01 size: 75
Dirty Sharding 14 size: 50
Dirty Sharding 15 size: 80
All Dirty Sharding List In Memory

Fig. 9. Sharded dirty list in memory

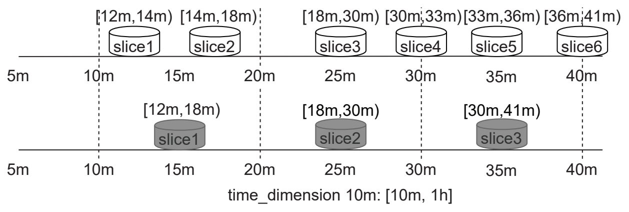

Fig. 10. Profile compaction

D. Compact and Truncate

The explosive growth may lead to dramatic profile expansion. In the practice of online recommendations, we found that the user profile has two characteristics:

- *Temporal*: e.g. in the recommendation scenario, the user behaviors within a week usually have a much higher significance than those happened a month ago.
- *Long-tail*: e.g. among the massive number of user behaviors, there are a large number of low-value long-tail features.

IPS relies on the compaction and elimination mechanisms to prevent profile data explosion while keeping the desired data quality. Note that one *Profile Data* consists of a linked list of *Slice*. To prevent the size of this linked list from growing too long, the compaction process provides two alternative methods: *Compact* and *Truncate*.

a) Compact: Compaction merges multiple continuous slices into one based on a pre-configured time window. When merging adjacent slices, the feature count of the same FID can be aggregated according to the pre-configured reduce function (e.g. SUM, MAX). Listing 2 shows an example denoting *Slice* data from 10 minutes to 1 hour to be merged into one aggregated slice every 10 minutes. In Figure 10, we can see how a Slice list of original size six are merged into a new list of size three according to the configuration.

```
"time_dimension": {
    "10m": ["10m", "1h"]
}
```

Listing 2. IPS demo time-dimension config

Compaction does not drop any data but merges several consecutive slices into one. Although this process may reduce time precision, the recommendation practice shows that it can significantly reduce the profile data size without compromising the effectiveness of recommendation. List 3 shows a compact configuration widely used at the company.

b) Truncate: In some recommendation scenarios, models and algorithms do not care about user behavior after certain period of time (e.g. user's behaviors a month ago), or just need the user's most recent behaviors (e.g. user's last 100

```
"time_dimension": {
    "1s": ["0s", "1m"],
    "1m": ["1m", "1h"],
    "1h": ["1h", "24h"],
    "1d": ["24h", "30d"],
    "30d": ["30d", "365d"]
}
```

Listing 3. IPS time-dimension config example

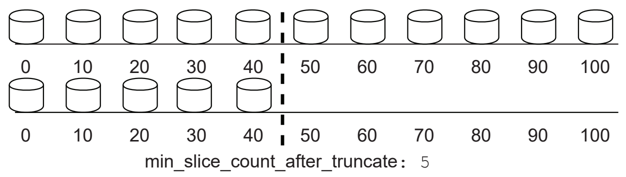

Fig. 11. Profile truncation example

clicks). For these purpose, IPS uses the *Truncate* process to remove the old data with low value. Depending on the upstream requirements, IPS supports truncate by time range and slice count, e.g. Figure 11 shows only the first five slices are preserved in the *truncate by count* scenario.

Profile data may accumulate in another way under the *Compact* method. Each *Slice* is a snapshot of the user's feature behaviors over a period of time. With the continuous slices in the specified time range merged, the long-tail feature data in each slice still grows over time. These long-tail features generally have low counts and little impact on recommendations, but they can consume large amount of memory. As IPS needs to sort all features by its count during query processing, excessive long-tail feature data may consume a lot of compute resources.

To control the amount of the low quality long-tail data, IPS uses the *Shrink* process to clean up the long-tail data while keep the high quality features. In the online recommendation practice, the *Shrink* process is designed based on the following principles:

- *Data Freshness*: Even if a feature has a low count, its quality is high if it occurred relatively recently. Although the current behavior count of one feature is low, it may gets greater over a longer period of time with subsequent actions. Therefore, *Shrink* should choose old data over recent data.
- *Multi-dimensional Sorting*: It is a common practice to determine the importance of different features by their counts, which vary significantly. For instance, IPS may store counts of various actions such as clicks, shares, likes and comments with different significance. The *Shrink* process needs to take into account the significance of each action by implementing multi-dimensional sorting.
- *Balance between short term and long term interests*: For instance, it may not be a good strategy to always favor the most recent data because the historical data may represent a user's long-term interests. In this case, simply eliminating the old data may cause bad experiences in recommendation. Hence, balance in time and categories should also be considered in feature elimination.

One important step in the *Shrink* process is to determine the number of features that need to be retained. As different recommendation models may have different requirements, IPS provides a configurable way to guide the number of features that need to be retained internally. Listing 4 shows an example configuration for *Shrink* method used in production in which the number of features retained in each slot are specified.

```
"slot": {
    "140": 500,
    "141": 2000,
    "143": 1500
}
```

Listing 4. IPS Shrink config example

Like the Garbage Collection in JVM, while data compaction can effectively reduce the profile data volume and processing latency, it also introduces some additional overhead. As the compaction of a profile is triggered by an incoming request and consumes non-trivial CPU time, the overall query performance may be adversely affected. To address this problem, we have been fine-tuning various compaction strategies to strike a balance between resource consumption and system performance. For instance, we migrate the compaction out of the main serving path and delegate them to run asynchronously in a dedicated thread pool with capped parallelism. In addition, as a full compaction is usually not necessary for most of profiles, we implemented a few strategies to determine whether to conduct a full compaction versus a partial one based on the actual load. By these means, we are able to control the CPU time spent on compaction and leave more resources to the query serving during the peak time.

Based on our production metrics, the average length of a slice list is 62 and the average slice data size is around 730 bytes, which means each user profile uses about 45KB memory and it remains fairly stable. Suppose each slice contains 5-minute worth of data, if no compact and truncate in place, a user's profile may grow to 76MB after a year, which is clearly not economically practical. With the help of these mechanisms, we are able to control the total memory cost of IPS clusters while ensuring the expected feature data quality (as proved by extensive A/B tests), which makes it possible to store more profiles in memory, obtain higher cache hit ratio and reduce serving latency with the same amount of resources.

E. Persistent Storage

As discussed above, the flush threads periodically flush out the updated profile data in the dirty list to the underlying key-value store. In the case of a cache miss, the requested data can be loaded into the cache from the persistent storage. IPS currently adopts a simple model: the key in the key-value structure is the profile ID while the value is a binary representation of the user's entire profile data. IPS now only relies on simple APIs like *set* and *get* to store and retrieve a user's profile as a whole.

As shown in Figure 12, IPS serializes the in-memory user profile data into a Protocol Buffer [11] format that encodes the hierarchy of feature data. In addition, IPS compresses the

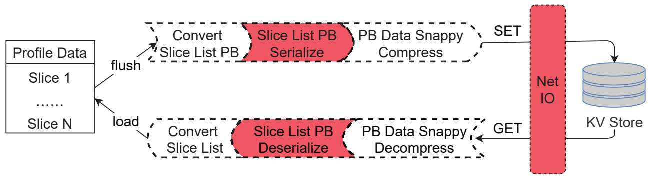

Fig. 12. Bulk profile persistence mode

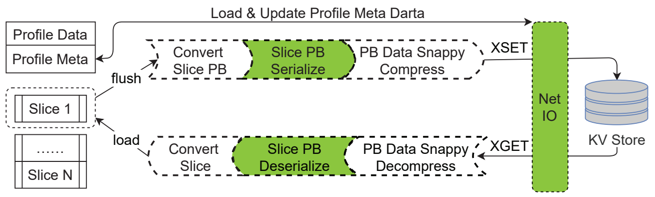

Fig. 14. Fine-grained profile persistence mode

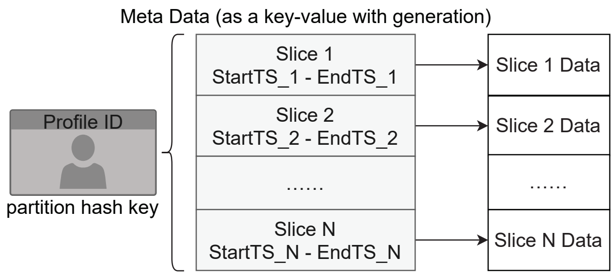
Meta Data (as a key-value with generation)
Profile ID
partition hash key
Slice 1 StartTS_1 - EndTS_1
Slice 2 StartTS_2 - EndTS_2
......
Slice N StartTS_N - EndTS_N
Slice 1 Data
Slice 2 Data
......
Slice N Data

Fig. 13. Slice meta structure

serialized data using Snappy [12] to reduce the network traffic and storage space. Based on our experience, a single user's profile usually takes less than 40KB in space after serialization and compression.

Occasionally, very large values may be generated due to either large test profile data or intensive usages. With excessive profile value size, IPS has to spend more CPU time for both serialization and deserialization, which limits the amount of cached profiles and sometimes exhausts the network IO bandwidth of the persistent storage. To solve this problem, IPS splits profile value and adjusts the granularity of data flushing and reloading from the entire profile to slice level. As shown in Figure 13, IPS adds a new layer of slice meta data on top of the overall profile data, which records all slice list meta-information inside a profile. In this way, IPS splits a large profile value into a meta data value and a series of slice values. Typically, a single slice data is small and can be controlled by the time interval configuration in compaction. In this scheme, it is critical to ensure consistency between the stored metadata and the corresponding slice values as the updates to them may not be performed atomically.

Figure 14 shows how IPS relies on the data version to control the sequence of operations so as to ensure consistency. Each *xset* operation to the underlying storage generates a unique version for the value written. In order to perform an operation, each *xget* or subsequent *xset* operation must hold a valid version. If the version held in the operation is older than the current version in the key-value store, it means that the value is not up to date and needs to be reloaded before further operations. The slice meta data can only be updated until the relevant slice data is successfully updated to ensure the correctness and consistency.

F. Read-Write Isolation

For online applications, read performance is usually more important than the write performance. In the presence of real-time data ingestion, we need to ensure that the query performance is not adversely affected. To address this problem, IPS isolates the read and write traffic by keeping a separate write table from the main table. The input data is first added to a write-only table and periodically merged into the main table by applying the configured aggregate functions every a few seconds. This approach significantly reduces the contention on the main table therefore greatly improves the overall stability of query operations. Note that this approach requires an extra portion of memory for the write table and delays the data visibility slightly, but this trade-off is worthwhile since it achieves significant performance gains. To remedy the potential issues introduced by the isolation (e.g. lower cache hit ratio or more frequent swaps), IPS controls the memory usage of the write table within a configurable limit to prevent it from using too much memory. All operations over the write table must be lightweight and performed through a fine-controlled thread pool. In production, we provide a hot switch so that users can choose to turn on/off the isolation feature dynamically. For instance, if there is a need for an offline Map-Reduce job to ingest large amount of historical data into an IPS cluster, the user can choose to turn on the isolation temporarily to avoid interference with the online feature serving.

G. Fault Tolerance

As a production service that serves tens of millions requests per second, IPS must meet the stringent high availability criteria in a multiple-region data center setup. When a region fails, the other regions are able to take over all the traffic within minutes. In each local region, the IPS instances are vertically co-located with the services to minimize the access latency.

To ensure data consistency across the multi-region deployments, IPS employs the approach of multi-region writing in upstream Flink job. As shown in Figure 15, the upstream applications write data to all IPS instances regardless of region but only query those instances in the local region. In this multi-region deployment, only one IPS instance of a region persists data to the master cluster of the key-value store and the rest instances only query the slave cluster of the key-value store in their local regions. Note that this design can only guarantee weak data consistency across multiple regions because a failed node may have chances to load stale data. In our experience, this simple strategy works pretty well as minor data inconsistency is negligible in most recommendation based applications.

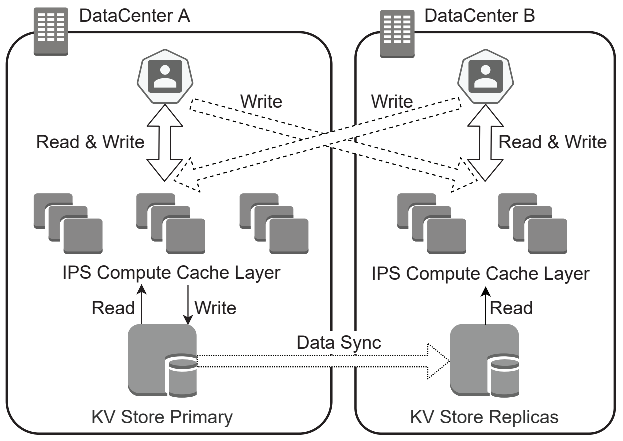

Fig. 15. IPS multi-regions deployment

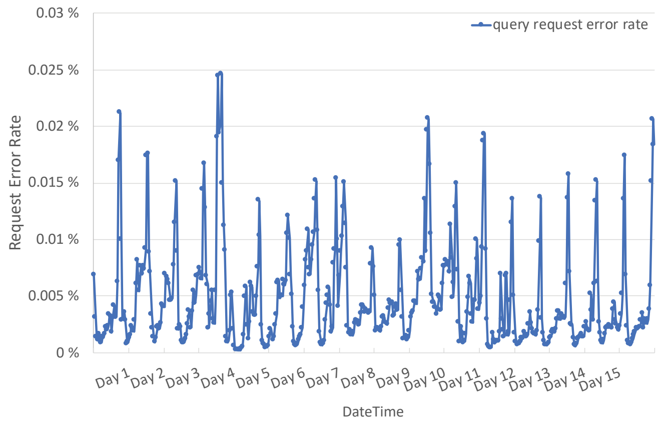

Fig. 17. IPS request error rate in client side over twenty days

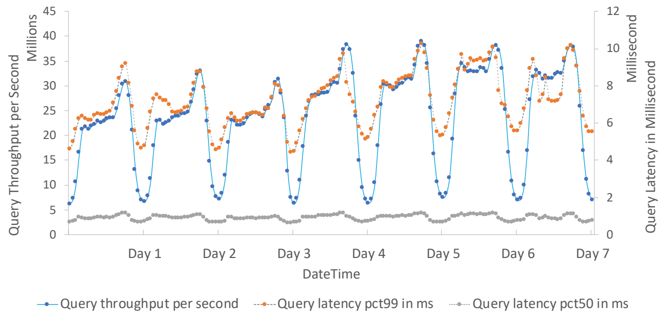

Fig. 16. Query throughput, 99th-percentile and 50th-percentile latency of Jinri Toutiao IPS cluster during Spring Festival of 2020

IV. Production Metrics

ByteDance has more than 50 IPS deployments in production, serving a variety of recommendation based products. All these clusters run on top of Kubernetes in a cloud native manner, which provides great resource efficiency and flexibility. In order to reduce resource fragmentation, a pod of IPS usually occupies the entire RAM resources on a host. IPS pod can auto-scale up and down depending on the workload. Each IPS cluster is usually shared by multiple applications in a multi-tenancy manner and a quota in terms of QPS is enforced for each upstream application. If an upstream client's usage exceeds its quota, IPS server will reject the requests from the same client until its usage falls below the limit.

At present, one of the largest IPS clusters in production is used by Jinri Toutiao with more than one thousand machines. The rest of this section shows key production metrics of this cluster as examples to demonstrate the scalability, performance and availability of IPS. Note that all metrics are measured by monitoring requests to/from IPS where the upstream callers are the online recommendation services of Jinri Toutiao product.

A. Scalability

Figure 16 shows the query throughput and latency of the IPS cluster that serves feature retrieval from Jinri Toutiao App during the Spring Festival of 2020. Each single pull-down in the App by a user will result in several articles or videos refreshed, which gets translated into a number of feature queries in the backend to the IPS cluster as part of the recommendation process. During the peak hours, the total throughput of this IPS cluster ranges from 30 millions to 40 millions feature queries per second; the 99th-percentile latency went from 9ms to 10ms and the 50th-percentile latency basically remains flat at about 1ms, attesting our approach scales as expected even though the online traffic fluctuates significantly.

B. Availability

Online recommendation applications usually impose a higher requirement on throughput and availability over profile services than strong consistency and accuracy. In production, it is critical for a large IPS cluster to continue functioning in the presence of machine crashes, network outages and data center failovers.

Figure 17 shows the upstream request error rate of Jinri Toutiao IPS cluster over 20 days. The maximum error rate is around 0.025% and the average is below 0.01%. The overall Service Level Agreement (SLA) can reach 99.99% in our production environment.

C. Performance

As described in section III, the end to end latency of IPS requests consists of network transmissions and the time for IPS to compute and retrieve profile data from the persistent key-value store in case of cache misses. Table II shows the IPS query latency on both client and server side, categorized by cache hit and cache miss, respectively. The overhead of package transmission on network is about 3ms and grows proportionally to the response data size.

In comparison, the cache hit cases save approximately 2 to 4 ms for each query. As shown in Figure 18, the typical cache hit ratio of an IPS cluster is above 90% and the memory usage ratio of the cluster remains stable at around 85%, thanks to the profile split optimization and the corresponding cache management strategy described in Section III.

TABLE II
CLIENT AND SERVER SIDE IPS LATENCY IN MILLISECONDS FOR QUERY REQUEST, CATEGORIZED BY CACHE HIT AND CACHE MISS

	cache hit lat. (msec)			*cache miss lat. (msec)*		
Operation	*50%*	*avg*	*99%*	*50%*	*avg*	*99%*
Query (client side)	3.5	3.9	10.2	4.1	4.7	14.1
Query (server side)	0.9	1.3	6.0	1.52	2.1	10.0

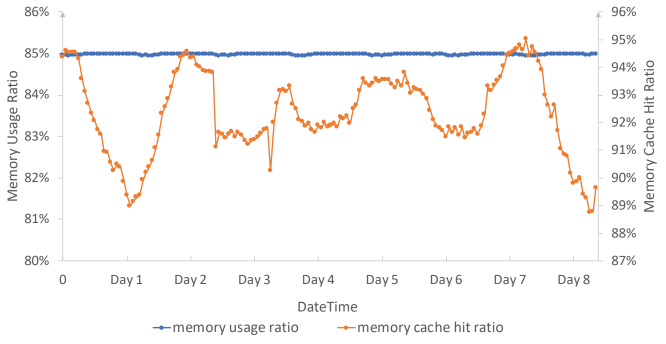

Fig. 18. IPS memory usage and cache hit ratio

Figure 19 shows the add throughput and latency of Jinri Toutiao IPS cluster over five days. During the peak hours, the throughput ranges from 3 million to 4 million writes per second; the 99th-percentile latency went from 4ms to 6ms and the 50th-percentile latency basically remains flat at about 0.5ms. Comparing to the query throughput above, we can see that the read traffic is about 10 times as that of the write traffic, which is typical in the targeted scenarios.

As described in Section III, IPS adopts read-write isolation to protect the query performance from being affected by the write requests. After the feature is enabled in production, the 99th-percentile latency of write operation went down about 80% while the query latency remains fairly stable.

V. Experiences & Lessons Learned

In this section, we briefly discuss some operational experiences and lessons we have learned after running IPS in production over past two years. As IPS serves as a general feature service, we have cooperated with many teams from recommendation, ads and search. The upstream customers provided us a lot of valuable feedback, which also greatly accelerated the iterations of the system.

a) Simplify User Adoption: Although IPS offers many advantages over legacy systems, as a new service, it inevitably imposed some barriers for customers to adopt at very early stage. Many customers need to thoroughly research the code and tune the right parameters to achieve satisfactory performance. In order to facilitate further adoption of IPS within the company, we summarized the typical usage scenarios and provided higher-level APIs or templating tools to ease the integration. For instance, we provide a number of streaming job templates for users to quickly create data ingestion pipelines as well as multilingual clients with higher level APIs.

b) Support Resource Quota and Online Reconfigurations: One IPS cluster is usually shared by multiple

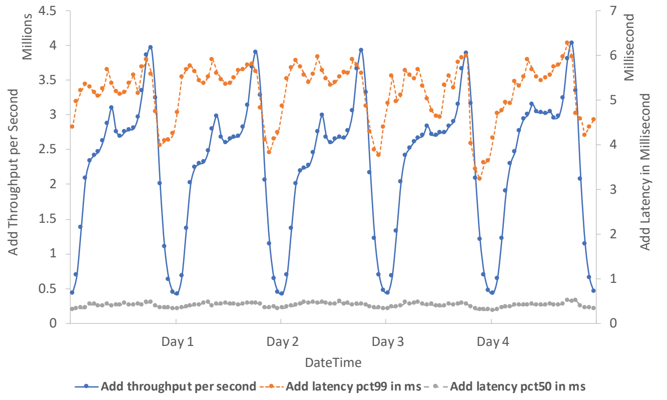

Fig. 19. Add throughput, 99th-percentile and 50th-percentile latency of Jinri Toutiao IPS cluster

upstream services, which aims to maximize the feature reuse. These callers may have different product strategies which vary a lot in terms of query QPS. In addition, there may be concurrent offline ingestion jobs during peak hours for back filling historical profile data. To prevent interference with each other, IPS provides resource quota mechanism based on the caller identity. A QPS quota is enforced for each caller on the server side to ensure the serving capacity required by customers of different SLAs. Besides the online quota, machine learning engineers often need to quickly test the effectiveness of different features. For example, a customer may need to run repeated experiments to obtain the best configurations of compaction and truncation inside IPS that can affect the time precision of certain features. With hot-reload functionality enabled for all feature dependent configurations, most changes can be made live in minutes without the overhead to restart the services.

c) Feature Engineering Efficiency Really Matters: As IPS has gradually become the unified profile management solution for online recommendations inside ByteDance, we got many feedback and insights from the key customers. First and foremost, IPS allows engineers to focus on their business logic without the overhead to create siloed profile services as before. Second, the implementation of IPS is often more efficient than legacy systems, which reduces a lot of wasted efforts. Third, since one IPS cluster can be reused by multiple products, one useful feature can be quickly propagated to all relevant upstream callers with almost zero additional cost. Last, many more enhancements over model accuracy become possible with the help of IPS. For instance, in one advertising related launch, the quick introduction of more IPS powered features led to 8% more ad clicks and 4% more conversions overall, which gets translated to significant amount of business values.

VI. Related Work

There has been a rich literature on the machine learning systems, especially focusing on the sparse models. In general there are two types of designs, with or without Parameter Servers. The parameter server architecture [13]–[18] relies on a centralized or distributed key-value store to store and update the parameters. The non-parameter server architecture [19]–[21] treats many nodes as homogeneous computing nodes, where each node can independently compute and store model weights. They often focus on the model itself, e.g., model embeddings, especially on the communication and computation efficiency. These work can also benefit from IPS in making better use of the features.

Traditionally, distributed key-value stores [22]–[24] are often used to implement the user profile systems. For example, Google's Personalized Search relies on Bigtable to record the user queries and clicks. Similar to the long-term profile described in Secition I, a MapReduce job over Bigtable periodically generates the user profile that is used to personalize the live search results. Another common way of implementing real-time model training is to leverage an external steaming processing system to aggregate events in sliding windows with different granularities, e.g. 5-min item clicks or 7-days item views. These aggregations are then written to a key-value store for online serving. In contrast, IPS explores a unique design by combining real-time ingestion, automatic aggregation and flexible window query in a coherent manner. As a result, the cost for incorporating new features into models in IPS can be significantly reduced, compared to the key-value store based approaches.

IPS also bears certain resemblance to OLAP systems [25]–[28], especially when considering aggregation and sorting operations in IPS are common SQL operators as well. These OLAP systems are primarily designed for analyzing large-scale data stored on distributed file systems and leverage in-memory cache to accelerate query processing. OLAP systems supports much more query operators such as joins than IPS. However, their intended use cases have much higher query latency (e.g. from 10s of seconds to 10s of minutes) than what is required for IPS (i.e. 10s of milliseconds). Also, the cycle for updating the underlying data of those OLAP systems is usually at least hours instead of seconds as required by the online machine learning systems.

Mesa [29] and Druid [30] are systems that are designed for near real-time data ingestion and low latency analytical queries over large data sets. There are also some recent systems [31]–[33] developed towards unifying serving and analytical processing. These systems are intended for use cases in which data needs to have well-defined schemas, with the help of rollups and materialized views to achieve the desired performance. However, as the training instance data is basically a bag of arbitrary key-value pairs, the data stored in IPS is inherently unstructured for maximizing the flexibility to incorporate new feature data. For Mesa like systems, there is a 5 to 10 minutes end-to-end update latency, which is deemed too high for online recommendation products as their performance are highly sensitive to feature data freshness. As for the query serving, a real-time analytical system may sustain up to tens of thousand queries per second (QPS), which makes it unsuitable to support the online recommendation use cases, where tens of millions feature queries per second are fairly common. In addition, none of the above systems are known to automatically compact and eliminate feature data as IPS does.

IPS shares some similarity to Time Series Databases (TSDB) as well. For instance, IPS uses an in-memory data structures that are similar to Gorilla's [34] that also performs time roll up aggregation for the older data. In comparison, the TSDB systems are mostly used for online service monitoring which has much more writes than reads. IPS needs to keep all data in memory to query various windows simultaneously while TSDB usually only caches the most recent data. As a result, querying over a long time window in TSDB may incur much higher latency.

VII. Conclusion

We have described IPS, a one-stop service for storing and serving profile data. In the context of content recommendation, one key insight is that individual data points are far less important than the aggregated behaviors, and recent data is of much higher value than old one. IPS' design and implementation, such as time-serial multi-level maps and automatic feature aggregation and elimination, make deliberated trade offs which primarily optimize for high availability and performance but can tolerate small amount of data loss or inconsistency. In essence, IPS acts as an in-memory cache that can sustain both high throughput updates and low latency feature queries to an ever-changing profiles of a large user base.

IPS is geo-replicated across multiple data centers for fault-tolerance and its separation of cache and persistence layers allows them to scale independently, which also facilitates the component reuse and resource sharing. Since its introduction two years ago, IPS' usage has been doubling every a few months given its scalability and flexibility in managing and querying profile data, which has proven to be critical for real-time recommendation at ByteDance.

Acknowledgments

We would like to thank Mu Qiao, Zhe Li, Jingting Jin and Changli Gao for their suggestions in the early stage of the IPS design. We would also like to thank many collaborators across ByteDance, especially Wenbin Tang, Wentao Yang, Dongliang Li, Qi Dong, Conghui Liao, Liang Xiang and Xiaojia Chang, for their patience and feedback as early adopters of the service. We would also like to extend our gratefulness to the SRE team, especially Yan Le and Chuanbing Jia for their production support. We are also grateful to Zhenyuan Yang and Dingkun Hong for their support and guidance to the team. Finally, we thank the anonymous reviewers, whose insightful feedback significantly improved the paper.

References

[1] "Tiktok - wikipedia," https://en.wikipedia.org/wiki/TikTok, 2017.
[2] "Jinri toutiao - wikipedia," https://en.wikipedia.org/wiki/Toutiao, 2012.
[3] M. Kiran, P. Murphy *et al.*, "Lambda architecture for cost-effective batch and speed big data processing," IEEE, pp. 2785–2792, 2015.
[4] H.-T. Cheng, L. Koc *et al.*, "Wide & deep learning for recommender systems," 2016.

[5] M. Naumov, D. Mudigere *et al.*, "Deep learning recommendation model for personalization and recommendation systems," 2019.

[6] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM Conference on Recommender Systems*, ser. RecSys '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 191–198. [Online]. Available: https://doi.org/10.1145/2959100.2959190

[7] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, Aug 2009.

[8] S. Rendle, "Factorization machines," in *2010 IEEE International Conference on Data Mining*, Dec 2010, pp. 995–1000.

[9] Y. Juan, Y. Zhuang *et al.*, "Field-aware factorization machines for ctr prediction," in *Proceedings of the 10th ACM Conference on Recommender Systems*, ser. RecSys '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 43–50. [Online]. Available: https://doi.org/10.1145/2959100.2959134

[10] "Apache thrift," https://thrift.apache.org/, 2017.

[11] "Protocol buffers," https://developers.google.com/protocol-buffers, Google Inc., 2016.

[12] "Snappy," http://google.github.io/snappy/, Google Inc., 2011.

[13] J. Jiang, B. Cui *et al.*, "Heterogeneity-aware distributed parameter servers," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 463–478. [Online]. Available: https://doi.org/10.1145/3035918.3035933

[14] M. Li, D. G. Andersen *et al.*, "Scaling distributed machine learning with the parameter server," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 583–598.

[15] Q. Ho, J. Cipar *et al.*, "More effective distributed ml via a stale synchronous parallel parameter server," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'13. Red Hook, NY, USA: Curran Associates Inc., 2013, p. 1223–1231.

[16] A. Smola and S. Narayanamurthy, "An architecture for parallel topic models," *Proc. VLDB Endow.*, vol. 3, no. 1–2, p. 703–710, Sep. 2010. [Online]. Available: https://doi.org/10.14778/1920841.1920931

[17] J. Dean, G. S. Corrado *et al.*, "Large scale distributed deep networks," in *NIPS*, 2012.

[18] T. Chilimbi, Y. Suzue *et al.*, "Project adam: Building an efficient and scalable deep learning training system," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 571–582.

[19] M. Abadi, P. Barham *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[20] A. Paszke, S. Gross *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle *et al.*, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[21] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in tensorflow," 2018.

[22] F. Chang, J. Dean *et al.*, "Bigtable: A distributed storage system for structured data," in *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 205–218.

[23] G. DeCandia, D. Hastorun *et al.*, "Dynamo: amazon's highly available key-value store." in *SOSP*, T. C. Bressoud and M. F. Kaashoek, Eds. ACM, 2007, pp. 205–220. [Online]. Available: http://dblp.uni-trier.de/db/conf/sosp/sosp2007.html#DeCandiaHJKLPSVV07

[24] "Apache hbase," https://hbase.apache.org/, 2007.

[25] M. Armbrust, R. S. Xin *et al.*, "Spark SQL: relational data processing in spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, T. K. Sellis, S. B. Davidson, and Z. G. Ives, Eds. ACM, 2015, pp. 1383–1394. [Online]. Available: https://doi.org/10.1145/2723372.2742797

[26] M. Kornacker, A. Behm *et al.*, "Impala: A modern, open-source SQL engine for hadoop," in *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015. [Online]. Available: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper28.pdf

[27] A. Hall, O. Bachmann *et al.*, "Processing a trillion cells per mouse click," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1436–1446, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p1436_alexanderhall_vldb2012.pdf

[28] S. Melnik, A. Gubarev *et al.*, "Dremel: Interactive analysis of web-scale datasets," *Proc. VLDB Endow.*, vol. 3, no. 1, pp. 330–339, 2010. [Online]. Available: http://www.vldb.org/pvldb/vldb2010/pvldb_vol3/R29.pdf

[29] A. Gupta, F. Yang *et al.*, "Mesa: Geo-replicated, near real-time, scalable data warehousing," *Proc. VLDB Endow.*, vol. 7, no. 12, pp. 1259–1270, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p1259-gupta.pdf

[30] F. Yang, E. Tschetter *et al.*, "Druid: a real-time analytical data store," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 2014, pp. 157–168. [Online]. Available: https://doi.org/10.1145/2588555.2595631

[31] B. Chattopadhyay, P. Dutta *et al.*, "Procella: Unifying serving and analytical data at youtube," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2022–2034, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2022-chattopadhyay.pdf

[32] C. Zhan, M. Su *et al.*, "Analyticdb: Real-time OLAP database system at alibaba cloud," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2059–2070, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2059-zhan.pdf

[33] X. Jiang, Y. Hu *et al.*, "Alibaba hologres: A cloud-native service for hybrid serving/analytical processing," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3272–3284, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p3272-jiang.pdf

[34] T. Pelkonen, S. Franklin *et al.*, "Gorilla: A fast, scalable, in-memory time series database," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1816–1827, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1816-teller.pdf