Journal of Computing and Electronic Information Management
ISSN: 2413-1660 | Vol. 14, No. 2, 2024

Designing a Multi-Level Caching Forward Index for Recommendation System Services

Hongye Zheng*

Chinese University of Hong Kong, Hong Kong, China
* **Corresponding author Email:** zhenghongye@link.cuhk.edu.hk

Abstract: This paper introduces a forward index based on microservices, which effectively addresses the issue of forward read amplification in recommendation systems through a multi-layer caching mechanism. This allows the forward service to handle a large number of concurrent requests with limited resources while ensuring service stability and final value rate. The paper provides a detailed description of the forward index's data structure design, traffic penetration calculation, and overall service architecture design, dividing the system into three parts: SDK, proxy service, and storage, corresponding to data query, data processing, and data storage respectively. In the SDK design, three components are employed: Cache Slot, synchronous queue, and timer, supporting synchronous, asynchronous, and flexible request forms. The proxy service design handles heterogeneous data and online computation, supporting batch and asynchronous designs, and provides a dynamic data protocol to adapt to the characteristic needs of different microservices. The storage layer design considers performance, cost, and maintainability, utilizing a Cache & SSD model to cope with high-concurrency scenarios.

Keywords: Recommendation system; Multi-level caching; Forward index Microservices architecture; Video recommendation; Cache hit rate; Heterogeneous data storage; Recommendation index; Recommendation system architecture; Index architecture.

1. Introduction

In recent years, with the continuous expansion of video, e-commerce, and other online service sectors, recommendation systems have become an important part of enhancing user experience and business value. By analyzing user behavior and preferences, recommendation systems can accurately suggest content or products that users might be interested in, thereby increasing user engagement and satisfaction. Among the many components of a recommendation system, the forward index is a crucial element [1].

In a typical distributed microservice recommendation system, a recommended item will have various dimension features that are repeatedly read in steps such as recommendation gateway, coarse ranking, mixed ranking, and recall. Generally, the functionality responsible for indexing from key to specific features is known as the forward index.

The forward index records the properties and features of each document or object, enabling the system to quickly retrieve and process relevant data. In recommendation systems, forward indexing is typically used to store user or recommended item data, such as user IDs, names, product numbers, categories, price information, etc. This data is not only vast and complex but also needs to be quickly accessed and processed in a high-concurrency environment to ensure that the recommendation system can respond to user demands in real-time.

As the application scenarios of recommendation systems continue to expand, the challenges faced by forward indexes are also increasing. First, recommendation systems need to handle massive amounts of data, which places high demands on storage and computing resources. Second, recommendation systems often need to complete data queries and processing in a very short time to provide real-time recommendations, putting enormous pressure on system performance and stability. Lastly, the diversity of different application scenarios and business requirements necessitates that forward index systems possess a high degree of flexibility and scalability to adapt to various complex business logics and data structures.

In recommendation systems, there is often a need to read multiple features of recommendable items repeatedly and continuously. If the content volume is not large, it can be distributed within the memory of various microservices. However, when the volume of recommended content is very large, solely using memory is insufficient. If this module is extracted as an individual microservice, it will face enormous traffic pressure from all microservices in the recommendation chain online. A single request entering from the recommendation gateway will result in massive amplification of RPC calls in the forward index service due to repeated, large-scale RPC calls. This process is known as read amplification.

Typically, retrieving detailed information using an ID is called indexing. A well-designed indexing system can quickly locate the corresponding content from a key. In recommendation systems, common indexes include forward indexes and inverted indexes. A forward index refers to obtaining an ID corresponding to a key. In video recommendation systems, this usually means getting details from a unique video ID. Conversely, an inverted index needs to retrieve corresponding IDs (often one-to-many) from a feature. In a typical recommendation system, let's take video/product recommendation as an example, many areas require obtaining item features for strategy development. For classic algorithms like Collaborative Filtering [2], user viewing history is used as one of the features. Deep Learning-based Recommender Systems [3] often use video metadata (such as titles and descriptions), user behavior data (such as viewing time and click-through rates), and contextual features (such as time and location) as input features. In recent years, with the iteration of recommendation algorithms, many new recommendation algorithms/models have been proposed,

widely utilizing indexed features. Chunyan Mao and others proposed constructing recommendation results based on user behavior data, extensively using user-dimensional features in the process. Some expected Markov model-based recommendation prediction systems also heavily utilize indexed features, which are widely used in model training [4,5]. Some new engineering approaches have also been proposed; Dong Liu proposed an efficient memory-based retrieval approach. Its cache-based design allows for faster data retrieval, but the article does not specifically explain how to implement it in the context of recommendation microservices [6].Yiyi Yang attempted to explain recommendation results by utilizing regression trees, During the model training process, a large number of index queries are also used, directly affecting the recommendation effectiveness. [7-9]. Shaojie Li proposed an efficient distributed feature storage scheme for feature extraction, which ensures that features maintain statistical significance while preserving data availability. However, he did not mention how to handle heterogeneous data under recommendation engineering scenarios [10].

The aforementioned algorithms all assume the presence of data. In actual production environments, when these algorithms are applied to online services, factors such as system stability, response time limits, and traffic limits may prevent these algorithms from always obtaining data or make the cost of obtaining data very high.

This paper provides an efficient solution for a recommendation indexing system to solve the read amplification problem encountered in production environments of recommendation systems at a relatively low hardware cost. It offers stable and fast text content retrieval for other microservices.

2. System Design

2.1. Read Amplification Problem Solution

In a typical recommendation system, the features of a recalled item are read repeatedly. Even within the same service, the recommendation algorithm may involve repeated reading and excessive single-instance reads. Such high-frequency read operations place multiple pressures on forward index storage. In a typical video recall system, a single recall may involve 3,000 to 10,000 items, making this large-scale data reading particularly prominent in microservice environments.

Forward index storage faces significant performance pressure. Common open-source storage solutions like Redis and MySQL, while performing well with small-scale data, exhibit performance bottlenecks in large-scale data reading scenarios. For example, when the number of items becomes too large in Redis, there is a noticeable increase in latency, which is unacceptable for high-concurrency recommendation systems. Additionally, forward index storage also faces substantial cost pressure. When storage volumes are vast, the capacity and configuration requirements for Redis clusters increase accordingly, inadvertently raising the overall system cost. In production environments, cost remains a critical concern, especially for large recommendation systems. Enterprises often prioritize cost-effective storage solutions, such as SSD-based storage systems. However, under heavy traffic, SSD-based storage systems may struggle to handle the online pressure, forcing enterprises to consider memory-based storage solutions.

Thirdly, forward index storage faces hardware pressure. When a large number of items are returned, each containing rich dimensional information, the system needs to handle significant data transmission and storage operations, placing high demands on network bandwidth and hardware resources. For example, in a video recommendation system, the dimensional information of a single video can often amount to hundreds of attributes, putting tremendous pressure on network bandwidth. Lastly, data in recommendation systems is often heterogeneous, stored in various different storage structures rather than a single storage system. This multi-source heterogeneous data storage and access further increases system complexity and pressure. Efficiently managing and reading data across multiple storage structures becomes a major challenge in forward index storage design.

To address these challenges, this paper proposes a multi-level cached forward index service. This design aims to balance performance, cost, and hardware resource usage by introducing a multi-tier caching mechanism, providing an efficient, stable, and flexible storage solution. In this architecture, data is first accessed from caller's memory cache; if the memory cache cannot meet the demand, the system then progressively accesses other storage tiers. This multi-level caching design allows the system to maintain high performance while effectively controlling costs and reducing hardware resource pressure.

In summary, finding the optimal balance between performance, cost, and hardware resources is a critical issue in the design of forward index storage for recommendation systems. The multi-level cached forward index service proposed in this paper aims to provide an effective solution to this problem, thereby promoting the application and development of recommendation systems in practical production environments.

2.2. Protocol

In this architecture, when used within a forward index system, a key-value structure is typically employed to organize data. Here, the key is generally a basic type, such as a string or an int, while the value is the actual data to be read into memory. This data can be stored and transmitted in various formats, such as JSON strings, Protocol Buffers, or Flatbuffers. In practical production environments, due to the significantly low performance overhead of Flatbuffers during data decompression [11], we prefer to use Flatbuffers as the data exchange protocol.

To better serve the various typical application scenarios of recommendation systems (such as recall, ranking, feature extraction, and user profiling), this architecture requires clients to incorporate our designed SDK. When querying the key in the forward index, clients do so by invoking the public methods provided by the SDK. The SDK interface is designed to be simple and user-friendly, effectively reducing the development complexity for clients.

2.3. Cache hit rate calculation

This paper introduces multi-level caching to address the read amplification problem. We define the number of concurrent requests N as the number of requests made simultaneously at time t. The number of invocations per request R refers to the number of times each request needs to access the cache. The request success rate S is the probability of successfully processing a request. The number of cache layers L is the number of layers in the caching system. The

base hit rate of the i-th level cache H_i is the hit rate under ideal conditions. The hit rate threshold of the i-th level cache T_i is the minimum required hit rate for the cache. The capacity of the i-th level cache C_i is the storage capacity of the cache. The timeliness of the i-th level cache E_i refers to the effective time of the cached data.

Calculation of actual hit rate:

Considering the total request magnitude R, we define the actual hit rate of the i-th level cache $H_{i,actual}(t)$ as:

$$H_{i,actual}(t) = H_i \times min(1, \frac{C_i}{N \times R}) \times (1 - \frac{R}{E_i}) \quad (1)$$

Where: $H_i \times min(1, \frac{C_i}{N \times R})$ represents the impact of cache capacity on the hit rate. If the total request $N \times R$ exceeds the cache capacity C_i, the cache hit rate will be limited. $(1 - \frac{R}{E_i})$ represents the impact of cache timeliness on the hit rate. When the request frequency R approaches the cache timeliness E_i, the cache hit rate decreases.

Single-Layer Traffic Penetration: For each level of cache, we need to determine whether the actual hit rate is below the threshold t_i. If it is below the threshold, the traffic penetrates to the next level of cache:

$$P_i(t) = \begin{cases} N \times R \times (1 - H_{i,actual}(t)), if\ H_{i,actual}(t) < T_i \\ 0, \quad otherwise \end{cases} \quad (2)$$

Recursive Traffic Penetration Calculation: We need to recursively calculate the traffic penetration for each level of cache until the lowest-level cache L:

$$p_{i+1}(t) = p_i(t) \times (1 - H_{i,actual}(t)) \quad (3)$$

The total traffic penetration to the lowest-level cache $p_{(L)}$ is:

$$P_L(t) = N \times R \times \prod_{i=1}^{L}(1 - H_{i,actual}(t)) \quad (4)$$

Considering the request success rate S, the actual traffic penetration to the lowest-level cache is:

$$P_{L,success}(t) = P_L(t) \times S \quad (5)$$

Let's assume the following situation: Number of concurrent requests N: 500, Number of calls per request R: 5,Request success rate S: 0.999,Number of cache layers L: 3, Base hit rate of the first level cache H_1: 0.8, Base hit rate of the second level cache H_2: 0.7, Base hit rate of the third level cache H_3: 0.6, Hit rate threshold for each level of cache T_1 :0.95, Capacity of the first level cache C_1:20,000,Capacity of the second level cache C_2:15,000,Capacity of the third level cache C_3: 10,000. Initially, there are 500 concurrent requests. Each request makes 5 calls, resulting in a total of 500 x 5 = 2500 requests. Among these requests, approximately 72.385 requests penetrate to the lowest-level cache (SSD). This demonstrates that multi-level caching can effectively alleviate the pressure on the lowest-level storage.

2.4. Architecture

The entire system is composed of three main parts: the SDK, the proxy service, and the storage.

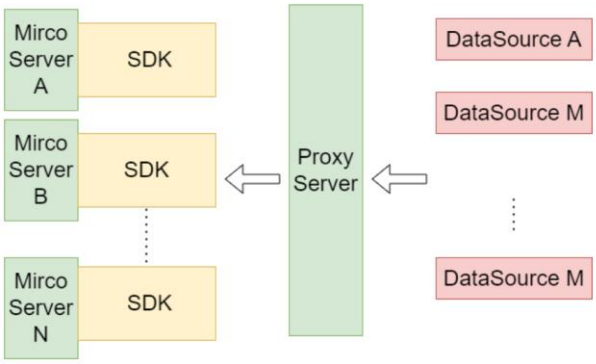

Figure 1. Architecture Designment

The above diagram is an architecture diagram of the entire indexing system. As shown, the forward index provides an SDK embedded in every service. The SDK exposes a common interface, MGET, whose input is the IDs that need to be fetched, and the output is the corresponding values and status codes for these IDs. Regardless of the number of IDs, the client does not need to interact directly with the proxy service. The proxy service acts as an intermediary between the storage and the client, responsible for processing and assembling data. The data sources are diverse, typically being KV-type storage, but can also be an RPC interface.

The data of forward indexing is diverse, and this diversity is reflected in aspects such as the storage medium of the data, the way the data is updated, and the sources of the data. Based on timeliness, indexing can be divided into second-level indexing and scheduled updating indexing. As the name suggests, second-level indexing updates data within seconds. Scheduled updating indexing takes more time, usually ranging from hours to several days. According to the method of data updating, it can be divided into streaming updates and batch updates. Streaming updates update data one item at a time, ensuring data real-time but bringing performance burdens. Batch updates pre-calculate all data in an offline environment and then push it to the online system in one go. This method can significantly save bandwidth costs for large-scale data updates but is less timely compared to streaming updates.

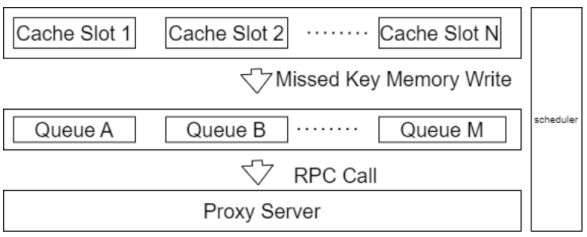

Figure 2. SDK Designment

In terms of storage, the system should support heterogeneous storage, generally including memory-based storage (such as Redis) and SSD-based storage (such as Rocks DB). Memory-based storage is suitable for high-concurrency scenarios due to its high-speed read and write characteristics, but its cost is significantly higher than SSD storage. On the other hand, SSD storage offers cost advantages but has inherent bottlenecks in read and write performance under high concurrency. Therefore, in practical applications, we adopt a Cache & SSD storage mode, which includes both memory caching and SSD storage. When the memory cache misses, the system will choose to either return directly or penetrate to the SSD for data retrieval based on

configuration options.

In the microservice chain of the entire recommendation system, the access heat of each microservice to the key is different. Some microservices' requests are mainly concentrated on hot data, while others may frequently access less popular items. To address this, we introduced a flexible option to either penetrate the data or return immediately. This design makes the system more flexible in dealing with different access patterns.

2.5. SDK Design

The SDK consists of three parts: Cache, Synchronous Queue, and Timer.

2.5.1. Cache

The cache module exposes get and set interfaces to support efficient data reading and writing. The implementation of the cache can be varied and customized according to each system's requirements for recommendation timeliness. Common implementations include caches based on the LRU (Least Recently Used) strategy. To enhance concurrency, we often use multiple Cache Slots, which are multiple independent cache memory blocks. When a read request arrives, the system performs a hash computation to route the request to a specific cache slot. This design can significantly improve the system's parallel access capabilities. Due to the advantages of the Murmur Hash algorithm in terms of computational efficiency and hash distribution, we chose it as the hashing method [12].

When a request arrives, the system first tries to retrieve data from the in-memory cache. If the key does not exist in the cache (i.e., cache miss), the system stores all missed keys in the local synchronous queue. Then, the Timer module (Scheduler) asynchronously batches RPC requests to fetch the required data. This approach reduces the direct access pressure on the backend storage system while improving the overall response speed of the system.

2.5.2. Data Access Modes

We have defined three data access modes: synchronous access, asynchronous access, and flexible access.

Synchronous Access: In traditional RPC calls, when a query request is passed to the SDK, the SDK immediately queries data from the cache or remote service and returns the result to the client. In this mode, the client waits for the query result to return.

Asynchronous Access: In asynchronous access mode, a single request from the client always returns immediately, whether or not data is retrieved. For keys that do not exist in the cache, the system performs asynchronous requests through the synchronous queue. This mode can improve the system's response speed but may result in some data delays.

Flexible Access: Flexible access mode combines the advantages of synchronous and asynchronous access. We set a threshold, and when the cache hit rate of a single request is below this threshold, the system automatically switches to synchronous request mode; otherwise, it uses asynchronous request mode. This design allows the system to flexibly adjust in different access scenarios, improving the efficiency and accuracy of data access.

2.5.3. SDK Snapshot

On the other hand, for an SDK that heavily relies on caching, memory snapshots are crucial. When the service restarts, the cache becomes invalid, and all in-memory data disappears. This can lead to the machine pulling a large amount of data in a short period. Although modern microservice clusters use batch releases to ensure service stability, the SDK still needs to be adequately prepared. Therefore, the scheduler periodically dumps the cache into a file. When the service restarts, it loads this file back into the service, preventing traffic surges caused by service restarts or version releases.

2.6. Proxy Server Designment

For a complex recommendation system, the data sources used in various servers are often diverse, and their required timeliness varies. In addition to general machine understanding and manually annotated tags, there is also a large amount of internal mining data with timeliness ranging from seconds to days and varying in scale. This diversity means that data may not be uniformly stored in one storage system, and the protocols may differ as well. Data sources might include a Redis instance, an internally developed memory & SSD type, or even just a microservice interface, which in turn may have another storage system behind it.

On the other hand, some features might involve online computation. For example, we might set an adjustable timeliness for a recommended item to ensure that it can expire within the recommendation system. This expiration needs to be judged in real-time to ensure that the item expires or is not recommended at the precise moment.

Therefore, to better support extensibility and heterogeneous data, we introduce an intermediary service, Proxy, rather than allowing the SDK to directly access storage. In production environments, situations can be much more complex. Apart from implementing special logic based on the request itself, or needing special processing for values, there might also be dependencies on external systems. Thus, introducing a proxy service to handle these issues is reasonable.

We consider a typical forward index data to be in a KV structure, where the value is generally in Protobuf, String JSON, or FlatBuffer data structure. More specifically, each value contains several features of this key. Taking a video as an example, a video can contain its physical characteristics, length, resolution, etc., as well as some video understanding dimensions, such as content tags and categories. We call such a column a plaintext feature. These data are uniformly stored in the value and should be specific data that the requestor needs to read.

The entire Proxy system can be abstracted into several concepts: Cluster, which represents an abstraction of storage, such as a Redis instance; Fields, which represent a feature, such as video resolution or length; Key, which comprises several Fields and represents a minimum data unit; KV, which comprises several Keys and Clusters, representing a data group; Biz, which comprises several KVs, representing a category of business.

To better understand each abstract concept, we can materialize them into configurations, based on actual business needs. For example, in a shopping website's recommendation index system: biz represents different business scenarios, such as shopping for live broadcasts and mobile searches, which require different ID representations. Cluster corresponds to information about feature data storage, such as Redis IP and password. Key represents this storage feature collection, which can be seen as an explanation of this KV storage. For example, a product storage has three features: ID, name, and price. The corresponding key should include these

feature IDs, field names, and their positions in the complete table. KV is the collection of this category of data, usually requiring the protocol for accessing storage under the KV category to be consistent.

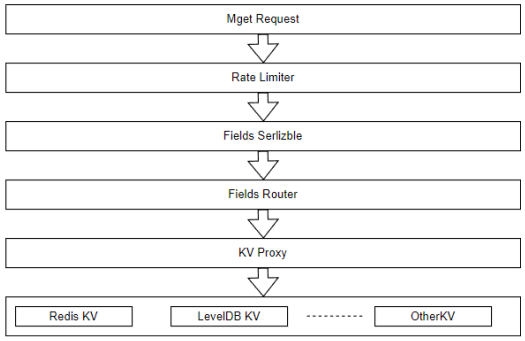

Figure 3. Proxy Designment

As shown in Figure 3 above, Proxy process can be broken down into the following steps: When the SDK calls the Mget, it first confirms the rate limiter Token corresponding to the BizID, then parses the requested fields and automatically routes to the corresponding KV. Each KV module then makes requests to the storage separately, and finally, the Proxy service merges the results.

2.6.1. Batch Design

For large-scale requests, it is difficult to control the number of items the client required, which can range from 1 to several thousand. When the request volume and single request size are particularly large, storage systems like Redis clusters may exhibit a significant increase in read latency and noticeable tail latency [13], leading to an overall increase in system latency. Therefore, compared to allowing clients to directly request storage, the batch processing logic in both the Proxy service and SDK can effectively manage the response time issues during reads.

2.6.2. Dynamic Data Protocol

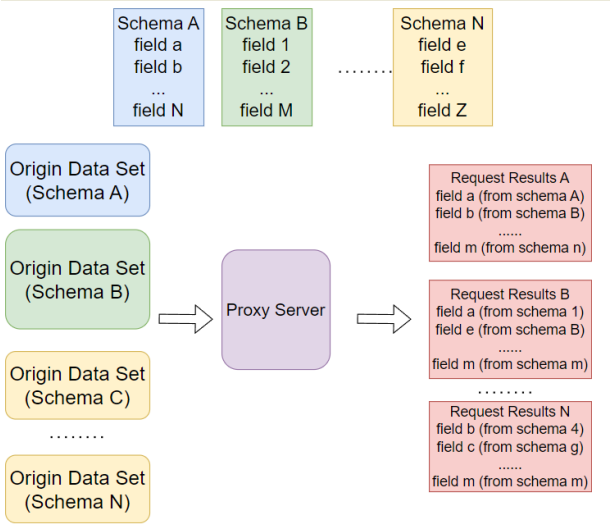

Figure 4. Protocol Designment

In a recommendation system composed of multiple microservices, the number and content of features required by each microservice are not the same. For example, in an actual production environment, the recall server may rely on hundreds of features, while user profiling only requires fewer than 10 features. If these microservices all use a unified protobuf protocol, the user profiling service would have to bear additional bandwidth, memory, and CPU overhead. Although string formats (such as JSON) can be used to handle this situation, the parsing of string formats (even with efficient libraries like fastjson) is still much slower than using flatbuffers. To address this issue, we designed a dynamic routing mechanism at the Proxy layer.

2.6.3. Protocol Design and Implementation

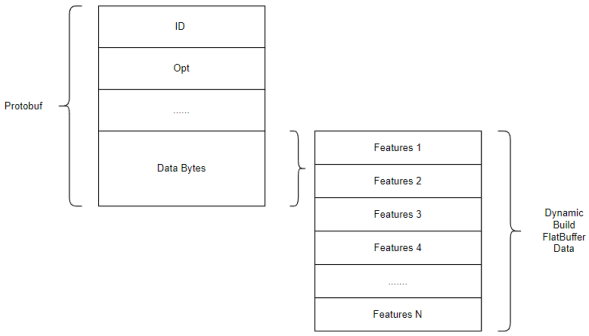

Figure 5. Protocol Designment

As shown in the diagram (Figure 5), the actual returned data consists of two layers:

General Protocol Layer: The first layer is the general protocol, composed of Protobuf. In the proto definition, in addition to the ordinary ID protocol, it also includes some other monitoring dimensions and business-level information.

Feature Data Layer: The actual returned feature data is all stored in a single field in the form of a bytes array. This bytes array can actually be parsed into a Flatbuffer format.

2.6.4. Workflow

Request Processing: When the client initiates a request, the Proxy first parses the basic information according to the general protocol layer.

Feature Assembly: According to the feature combination requested by the client, the Proxy extracts the corresponding feature data from the full feature set and assembles these data into flatbuffer format to return to the client.

Dynamic Routing: Through the dynamic routing mechanism, the Proxy can flexibly assemble and return the required feature data without returning the entire large table. This not only reduces bandwidth usage but also lowers the client's parsing overhead.

2.7. Storage Layer Design

When choosing a storage solution, it is essential to consider both data scale and cost factors. Mainstream open-source data storage systems such as Redis, FastKV, and RocksDB each have their own advantages. These can be summarized from the following aspects: 1. Performance, 2. Cost, and 3. Maintainability. For most memory-based KV storage systems, their performance is generally superior to SSD-based storage systems [14,15]. The high-speed read and write capabilities of memory make it perform exceptionally well in high-concurrency access scenarios. However, when the data volume is large, the high cost of memory becomes a significant limiting factor. The cost of memory is far higher than SSD When the data volume is small, pure memory-based databases are usually the first choice. However, when the data volume reaches a massive scale, the cost advantage of SSD storage cannot be ignored. For example, in a large-scale recommendation system, the number of items can reach tens of billions, with hundreds of forward index features. Assuming an average value size of 8KB per value, at a data

volume of 2 billion, the required storage space is approximately 15TB. If deploying in three locations and retaining three copies per location, the total storage requirement will reach about 134TB. Using purely memory-based storage for this would be a very significant expense.

Different storage systems also vary in maintainability. Memory-based storage systems are usually simpler, with relatively low maintenance costs. However, as the system scales and the storage media diversifies, the maintenance complexity will correspondingly increase. SSD-based storage systems, although having higher maintenance costs, show significant advantages in large-scale data management.

In recommendation systems, there are usually hot content items that are frequently accessed, resulted in hot keys. We can leverage this characteristic by adopting a Cache & SSD hybrid storage strategy. This strategy can greatly alleviate memory pressure: hot keys are always kept in the cache, whereas cold keys are loaded into the cache only when needed. This hot-cold separation strategy ensures the efficiency of high-frequency access while effectively utilizing the low-cost SSD storage. Additionally, the storage system should provide an option to directly penetrate to SSD. In some scenarios, a high success rate for a single read of the forward index is required, or when the accessed data is relatively cold, direct penetration to SSD for storage access is necessary. This design ensures that the system can flexibly adjust storage strategies when facing different access patterns and data characteristics, providing optimal performance and cost-efficiency.

2.8. Monitoring Design

Monitoring is crucial for ensuring system stability. Here are some common monitoring practices in forward indexing:

RPC Monitoring: For a non-local cache forward index, the RPC success rate is critical. Operations and development personnel should always monitor peak request volumes as well. Typically, a standard microservice should ensure a success rate of 99.99% or 99.9% and above.

Cache Hit Rate Monitoring: For this SDK, another important business metric is whether the value can ultimately be retrieved. As mentioned earlier, the method of retrieving a value for a request can be asynchronous, synchronous, or a mix of both. Introducing cache hit rate monitoring can provide a direct view of whether users ultimately read the value. We call this business metric non-empty item hit rate, calculated as the number of items with values divided by the total number of requests.

On the other hand, Proxy should also have monitoring. As mentioned earlier, traffic control is crucial, so monitoring should clearly reflect the traffic and corresponding RPC success rate for each business and each primary call source. Additionally, Proxy should have primary call monitoring to clearly know the RPC success rate of storage primary calls. Proxy should also monitor the number of items per single request for each business, making it easier to control the batch quantity for each request.

3. Conclusion

The recommendation index system designed in this paper is an efficient solution based on a microservice architecture, particularly suitable for high-concurrency, high-traffic application scenarios. Through a multi-level caching mechanism, this system effectively addresses the read amplification problem inherent in recommendation systems and is highly adaptable to business needs. By designing multi-level caches, the system resolves high-concurrency processing and read amplification issues, enabling it to withstand extremely high-traffic concurrent access, significantly improving system response speed and reducing direct access to backend storage systems, thus lowering the overall read overhead.

Through dynamic routing and flexible data parsing mechanisms, the system efficiently handles different business requirements, ensuring a high rate of successful data retrieval online, thereby enhancing the reliability and accuracy of recommendation results. The multi-level caching not only boosts system performance but also significantly reduces system costs. By employing a cache & SSD hybrid storage strategy, the system notably alleviates memory pressure while making use of low-cost SSD to store a large amount of cold data, balancing performance and cost.

The designed protocol-within-a-protocol mechanism and the use of Flatbuffers format minimize data parsing overhead, ensuring that data can be retrieved efficiently and economically.

References

[1] McDonald, D. W., & Ackerman, M. S. (2000, December). Expertise recommender: a flexible recommendation system and architecture. In Proceedings of the 2000 ACM conference on Computer supported cooperative work (pp. 231-240).

[2] Schafer, J. B., Frankowski, D., Herlocker, J., & Sen, S. (2007). Collaborative filtering recommender systems. In The adaptive web: methods and strategies of web personalization (pp. 291-324). Berlin, Heidelberg: Springer Berlin Heidelberg.

[3] Zhang, S., Yao, L., Sun, A., & Tay, Y. (2019). Deep learning based recommender system: A survey and new perspectives. ACM computing surveys (CSUR), 52(1), 1-38.

[4] Chunyan Mao, Shuaishuai Huang, Mingxiu Sui, Haowei Yang, Xueshe Wang, Analysis and Design of a Personalized Recommendation System Based on a Dynamic User Interest Model. Advances in Computer, Signals and Systems (2024) Vol. 8: 109-118.

[5] Zhizhong Wu, Xueshe Wang, Shuaishuai Huang, Haowei Yang, Danqing Ma, Research on Prediction Recommendation System Based on Improved Markov Model. Advances in Computer, Signals and Systems (2024) Vol. 8: 87-97.

[6] Liu, D., Waleffe, R., Jiang, M., & Venkataraman, S. (2024). GraphSnapShot: Graph Machine Learning Acceleration with Fast Storage and Retrieval. arXiv preprint arXiv:2406.17918.

[7] Tao, Y., Jia, Y., Wang, N., & Wang, H. (2019, July). The fact: Taming latent factor models for explainability with factorization trees. In Proceedings of the 42nd international ACM SIGIR conference on research and development in information retrieval (pp. 295-304).

[8] Yang, H., Zi, Y., Qin, H., Zheng, H., & Hu, Y. (2024). Advancing Emotional Analysis with Large Language Models. Journal of Computer Science and Software Applications, 4(3), 8-15.

[9] Zheng, H., Wang, B., Xiao, M., Qin, H., Wu, Z., & Tan, L. (2024). Adaptive Friction in Deep Learning: Enhancing Optimizers with Sigmoid and Tanh Function. arXiv preprint arXiv:2408.11839.

[10] Li, S., Dong, X., Ma, D., Dang, B., Zang, H., & Gong, Y. (2024). Utilizing the lightgbm algorithm for operator user credit assessment research. arXiv preprint arXiv:2403.14483

[11] Pradana, M. A., Rakhmatsyah, A., & Wardana, A. A. (2019, September). Flatbuffers implementation on mqtt

publish/subscribe communication as data delivery format. In 2019 6th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI) (pp. 142-146). IEEE.

[12] Li, S., Dong, X., Ma, D., Dang, B., Zang, H., & Gong, Y. (2024). Utilizing the lightgbm algorithm for operator user credit assessment research. arXiv preprint arXiv:2403.14483.

[13] Kara, K., & Alonso, G. (2016, August). Fast and robust hashing for database operators. In 2016 26th International Conference on Field Programmable Logic and Applications (FPL) (pp. 1-4). IEEE.

[14] Pan, C., Luo, Y., Wang, X., & Wang, Z. (2019, November). predis: Penalty and locality aware memory allocation in redis. In Proceedings of the ACM Symposium on Cloud Computing (pp. 193-205).

[15] Cao, Z., Dong, S., Vemuri, S., & Du, D. H. (2020). Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In 18th USENIX Conference on File and Storage Technologies (FAST 20) (pp. 209-223).