Future Generation Computer Systems 141 (2023) 67–80

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

BS-Join: A novel and efficient mixed batch-stream join method for spatiotemporal data management in Flink

Hangxu Ji [a], Su Jiang [a], Yuhai Zhao [a,*], Gang Wu [a], Guoren Wang [b], George Y. Yuan [c]

[a] School of Computer Science and Engineering, Northeastern University, No. 3-11, Wenhua Road, Heping District, Shenyang, 110819, PR China
[b] School of Computer Science and Technology, Beijing Institute of Technology, No. 5, South Street, Zhongguancun, Haidian District, Beijing, 100081, PR China
[c] Thinvent Digital Technology Co.,Ltd., No. 681 Torch Avenue, High-Tech Development Zone, Nanchang, 410000, PR China

ARTICLE INFO

Article history:
Received 7 September 2022
Received in revised form 30 October 2022
Accepted 11 November 2022
Available online 15 November 2022

Keywords:
Flink
Mixed batch-stream data join
Cache
Hotspot awareness
Skip list

ABSTRACT

The new computing model, mixed batch-stream data processing, plays a crucial role in big spatiotemporal data managements. As the core of the above computing method, mixed batch-stream data join has high requirements on the throughput and latency due to the coexistence of two types of data sources. Apache Flink is the most suitable distributed system for mixed batch-stream data join, with lower latency than the join calculation model based on Hadoop and Spark, and it simulates remote real-time reading of batch data sources and completes join calculation with the DataStream API. However, as the degree of parallelism increases, frequent remote data reads will cause huge disk and communication pressure, thereby reducing the job efficiency and scalability. To make things trickier, the above effects are further amplified when simulating complex operations such as range joins. Aiming at the above difficulties and the characteristics of mixed batch-stream data join, a cache-based framework supporting mixed batch-stream join computing natively is proposed, which increases the search speed in the process of data join by building indexes in batch data sources. Meanwhile, for equijoin and range join, an optimization mechanism based on hotspot awareness and an optimization mechanism based on skip list are proposed respectively to further improve the job efficiency. In summary, the advantages of our work are highlighted as follows: (1) The proposed framework enables Flink to natively support mixed batch-stream data join, and can improve throughput by 5 times and speedup by 4 times; (2) The optimization mechanism based on hotspot awareness can further improve the efficiency of equijoin; (3) Compared with range queries by traditional Operators in Flink, the throughput can be increased by 6 times while the latency is reduced by 45%.

1. Introduction

In big data applications in all walks of life, the data range spans real-time data and historical data, which requires both batch processing of massive offline data to ensure its high throughput and stream processing of massive online data to ensure its low latency. Furthermore, with the increasing complexity of big spatiotemporal data applications, a new mode of interactive computing between offline data and online data has emerged, called mixed batch-stream data processing. For example, in e-commerce systems, it is necessary to join the real-time online data of the products purchased by the users with the offline spatiotemporal data such as user's geographical location and the time law of browsing goods for product recommendation [1,2]; In smart city systems, real-time road traffic event online spatiotemporal data and offline road traffic information are joined to calculate various intersection indicators [3–5]. The above scenarios take equijoin and range join as the basic operations, and put forward higher requirements for distributed computing systems.

In mainstream distributed computing systems, Hadoop [6] and Spark [7] are not suitable as the carriers of mixed batch-stream data join. This is because they are computing systems with batch data processing as the underlying engine, and their real-time processing ability of stream data is weak, so they cannot guarantee the low latency in mixed batch-stream data join. Therefore, the current mainstream join optimization methods based on parallel computing and distributed computing [8–12] are mostly oriented to batch data computing scenarios. To this end, Flink [13] is generally selected as the distributed system for mixed batch-stream data join, since it supports both batch data processing and stream data processing natively, and has the advantages of high throughput and low latency.

* Corresponding author.
E-mail addresses: jhx0223@gmail.com (H. Ji), 598757400@qq.com (S. Jiang), zhaoyuhai@mail.neu.edu.cn (Y. Zhao), wugang@mail.neu.edu.cn (G. Wu), wanggrbit@126.com (G. Wang), yuanye@thinvent.com (G.Y. Yuan).

https://doi.org/10.1016/j.future.2022.11.016
0167-739X/

However, Flink does not provide any native model and interface for mixed batch-stream join. It is necessary to use the stream data processing model and interface (DataStream API) to simulate the access and processing of batch data, and model the batch data as a "bounded stream" and centralize it locally for storage and reading, which brings great inconvenience to users in developing upper-layer applications. Moreover, in the above data processing mode, the stream data needs to be interactively calculated in real time and in parallel with the batch data stored centrally. Although this method of data reading and computing will not cause too much impact on the latency of result output, when the parallelism increases, it will inevitably cause calculation blocking due to frequent remote data acquisition. In other words, when computing resources are expanded, the scalability and speed-up ratio of jobs will be severely reduced, which is contrary to the purpose of distributed computing. If all batch data is loaded into the memory of each parallel computing instance, it will bring great pressure on computing resources and may cause the risk of memory overflow. Worse, when faced with complex queries such as range join, the data will be read and written more frequently and the amount of calculation will be higher, and the above bad effects will be further amplified. To address the above challenges, it is urgent to implement a computing model in Flink that natively supports mixed batch-stream join for spatiotemporal data management, and establish a lightweight and efficient data reading mechanism in this model to solve the blocking problem in high concurrency state. In addition, for the proposed mixed batch-stream join model and remote data acquisition optimization technology, it is necessary to further seek an optimization mechanism for mixed batch-stream join to improve job efficiency.

Based on the in-depth study of the importance of mixed batch-stream join model in big spatiotemporal data interaction processing and problems in the distributed computing of mixed batch-stream data processing, combined with the defects of join method in Flink, this paper proposes a novel and efficient cache-based mixed batch-stream data join framework in Flink for equi-join and range join. For different data join calculations, this technology introduces different types of incremental caches into Flink's parallel instances, and adds optimization strategies such as data hotspot detection and improved skip list. It can significantly improve the efficiency of mixed batch-stream join jobs without increasing too much pressure on memory resources.

The main contributions of this paper are summarized as follows:

(1) We propose a cache-based computing model in Flink that can support mixed batch-stream data join for spatiotemporal data management. By loading offline data into each parallel instance of the job in an incremental cache, it avoids frequent reading of local offline data by stream Operators, thereby significantly improving job efficiency;

(2) In order to solve the unstable system load caused by the skew of stream data, we propose a join method based on hotspot awareness, which makes the cluster achieve load balance through hotspot detection and data partition optimization mechanism. Experimental evaluation verifies that this method can reduce the latency, and improve the throughput compared with cache technology based on traditional hash table;

(3) In order to deal with more complex mixed batch-stream data join, we propose a join method oriented to range query, which improves efficiency by introducing cache technology based on skip list. Experimental results show that this method can significantly reduce the running time of range queries, and the sorting-based skip list can further improve the job efficiency.

The remainder of this paper is organized into 6 sections. Section 2 introduces the mixed batch-stream data processing model and data join model in Flink, and cache optimization technologies in distributed systems. Section 3 introduces the proposed cache-based mixed batch-stream data join framework. Section 4 explains the proposed join method based on hotspot awareness. Section 5 describes the proposed join method for range query. Section 6 presents the performance evaluation. Section 7 gives a brief conclusion.

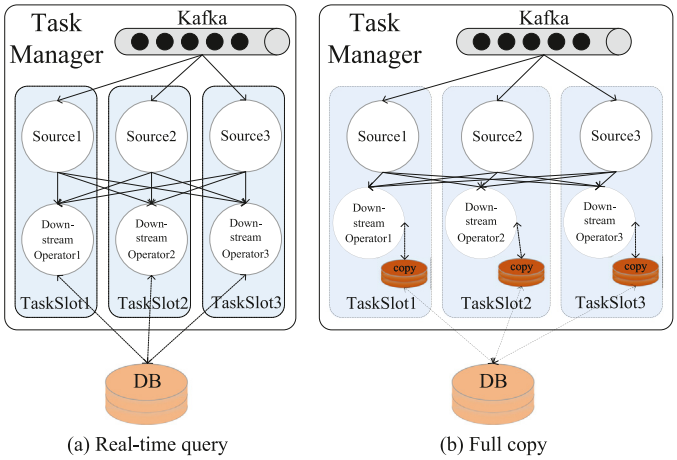


Fig. 1. Mixed batch-stream data processing methods.

2. Background and related work

In this section, we first summarize some of the implementation principles in Flink, including the mixed batch-stream data processing model, and the data join model in Flink, to verify the feasibility of our work. Then, the related work of cache optimization technologies in distributed systems are explained, and the advantages and deficiencies of existing work are pointed out.

2.1. Mixed batch-stream data processing model in Flink

Since Flink does not provide any native mixed batch-stream data processing model for users, it is first necessary to set the execution environment of the job as stream data processing environment, and then model the offline data into "bounded stream" data using the DataStream API, which bring great difficulties to users to write programs. At the data computing level, in order to complete the interactive processing of mixed batch-stream data, the following two methods are generally adopted, as shown in Fig. 1.

In general mixed batch-stream data interactive computing in Flink, offline data is centrally stored in the database. The stream Operator responsible for the calculation receives the stream data sent by the upstream Operator in each parallel instance (in Flink, each parallel instance is a TaskSlot), pulls the batch data from the centralized database, and finally completes the interactive computing, as shown in Fig. 1(a). The disadvantage of the above computing model is that when the degree of parallelism increases, frequent remote data reading will cause communication blockage, and thus the performance improvement cannot be brought about by the expansion of computing nodes. In order to improve computing performance, entire batch data can be copied to each parallel instance, as shown in Fig. 1(b), but this computing method is extremely expensive, puts great pressure on memory resources, and has a high risk of memory overflow. Therefore, it is necessary not only to implement a native framework supporting mixed batch-stream data join for users' convenience, but also to study a lightweight data reading strategy, which can avoid data blocking without obviously increasing hardware pressure.

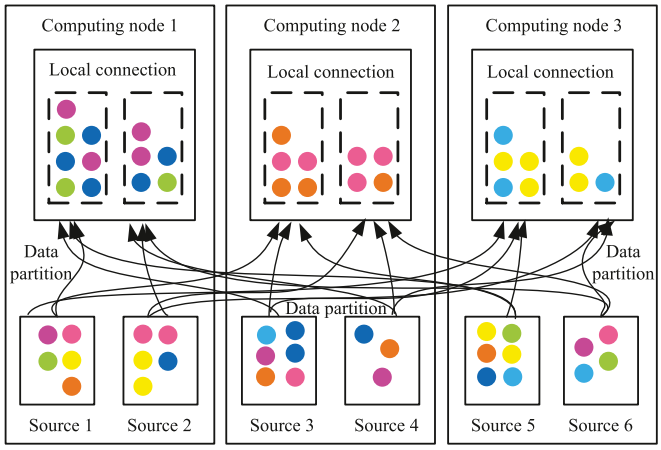


Fig. 2. Data join process in Flink.

2.2. Data join model in Flink

Join Operator is the basis for multi-source data interactive computing in Flink, and Flink provides join operations in both batch data computing and stream data computing. In addition to the join calculation in traditional offline data scenarios, in order to cope with the current rich real-time calculation applications, Flink provides time-based and window-based join calculation models for two stream data sources in stream processing mode, and uses "State" to cache real-time stream data waiting for join.

In the process of data processing, both data partition and local connection calculation should be carried out in turn for data join in batch processing and stream processing scenarios. As shown in Fig. 2, it first assigns the data records from the two connected data sources to different parallel instances, which is called Ship Strategy. Flink provides us with two data partition methods during the Ship Strategy: Broadcast-Forward and Repartition–Repartition. The former sends all one data source to a parallel instance with another data source partition, which is suitable for the situation of sufficient computing resources or small data sources; The latter uses the same partition function for the same attributes on the associated keys of two data sources. Although the data shuffling process has certain computational cost, it occupies less space and is more universal. When the data is partitioned, the local connection calculation is performed, which is called Local Strategy. Flink also provides us with Sort-Merge-Join and Hybrid-Hash-Join local connection computation methods, which are suitable for different data source sizes and available working memory space. When faced with the new mixed batch-stream data join computing model, the core of the research on improving job efficiency is still to seek optimization mechanism for data partition and local connection computing.

2.3. Cache optimization technologies in distributed systems

Indexing and caching mechanism are necessary to overcome the problems of insufficient memory resources and high complexity of query time and space in massive data processing [14–16]. Over the past decade, indexing techniques for organizing data have emerged, some of which are specifically designed for disk characteristics, such as B+ trees [17]. In order to improve cache utilization, CSB+ tree, a variant of B+ tree, is proposed in [18]. Compared with B+ tree, CSB+ tree only retains the pointer address of the first child node in each internal node, and the remaining child nodes are accessed by offset. There are also many memory-oriented indexing technologies, such as Red–Black Trees [19], skip lists [20], etc., The common memory index technology used for join is hash table, which has query performance close to O(1) complexity, but does not support range query. The query complexity of skip lists is equivalent to that of tree index, which can support range query and is simple to implement. Even the variant supporting high concurrency can be easily implemented through optimistic locking mechanism. The above studies either put forward parallel solutions for stand-alone environment, or put forward solutions for distributed environment, without considering the ubiquitous data hotspots in the real world. Data hotspots refer to highly inclined workloads that make only a small part of data frequently accessed [21], which often leads to further data skew in distributed systems. In the scenario of data skew, improper data partition will lead to unbalanced load of computing nodes, resulting in performance bottlenecks [22].

With the maturity of caching and indexing technologies, complex queries and applications based on them have emerged in large numbers. For example, in the storage and query of complex large graphs with extremely high time and space complexity, caching and indexing mechanism are important prerequisites for ensuring new social networks, road traffic networks and other applications [23–26]. At the same time, distributed caching and indexing mechanism is also applied to new spatiotemporal data management applications such as travel planning [27,28], trajectory mining [29–31] and keyword queries [32,33]. Therefore, for the new distributed computing system and the new mixed batch-stream data join computing, the above key technologies are still the research hotspots in the future.

3. Cache-based mixed batch-stream join model

We propose a novel framework in Flink, which can natively support the interactive processing of batch data and stream data, and provide an efficient cache-based mixed batch-stream join method for spatiotemporal data management in this framework, called BS-Join. This section mainly introduces the overall structure of BS-Join, and the specific implementation process of the optimization strategies will be described in detail in Section 4 and Section 5 respectively.

3.1. Mixed batch-stream data processing framework

Flink unifies batch processing and stream processing in the computing engine layer, but it still needs to create different execution environments for different data sources in the interface layer. When faced with mixed batch-stream data processing, it is necessary to set the execution environment as stream processing, and then access the batch data source as a "bounded stream" through the stream Operators, which is extremely inconvenient for users.

In order to overcome the above difficulties, we provide a new execution environment for batch data and stream data interactive processing in Flink, so that users can use batch Operator, stream Operator and mixed batch-stream Operator respectively in this execution environment. If there are Operators with duplicate names, it can be judged according to the type of input data source. Fig. 3 shows an example of mixed batch-stream data processing. In this environment, the stream Operators are still consistent with the logic of traditional stream computing, and perform real-time computing with data inflow; By modifying the data distribution and computation logic in new computing environment, batch Operators achieve the same effect as traditional batch computing; Mixed Operator is used for mixed computing of stream data and batch data, and its upstream Operators are stream Operator and batch Operator respectively. It blocks the stream calculation until the batch data arrives completely, and starts the calculation after

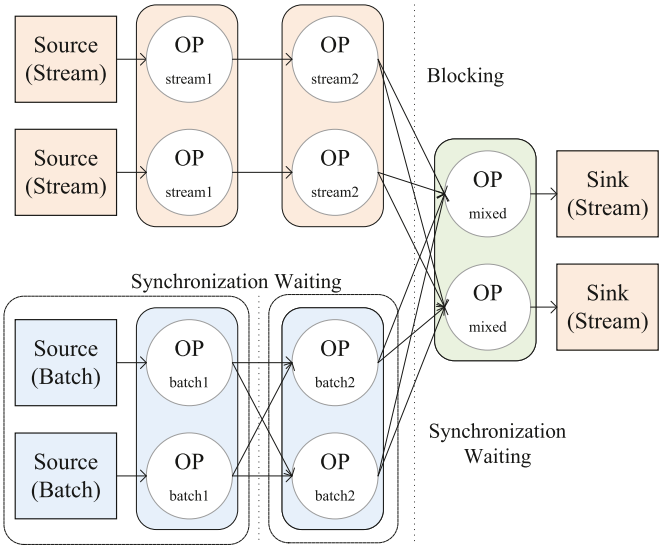


Fig. 3. An example of mixed batch-stream data processing.

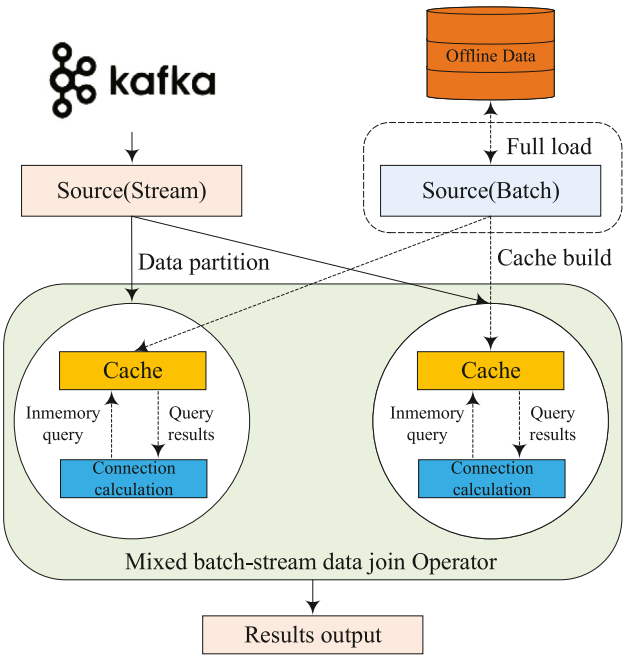


Fig. 4. Mixed batch-stream data join framework.

the batch data distribution is completed. The calculation process of mixed batch-stream data processing can be divided into the following steps:

- **Initialization**: First, the information needed to query the batch data source is imported into the computing nodes through a predefined data source format. Taking MySql-oriented query as an example, it needs to import database address, database password, database name, data table name and other information. Then, activate the dimension data query module built in the computing node to build the access and prepare for the query.
- **Data query**: Notify JobManager in Flink to distribute data. In the process of continuously reading stream data, every arriving stream data record is processed to obtain the information needed by query, and then query instructions are constructed and query results are obtained from batch data sources.
- **Data calculation**: After the data query is finished, data calculation is carried out, and the stream data is connected with the queried dimension data. In this process, if the query result is null, according to the specific design of calculation logic, choose to send the connection to the downstream computing nodes or abandon the data directly.

3.2. Mixed batch-stream data join method

Data join operation is the basis of interactive computing of multiple data sources, so we provide a native data join Operator in the framework of mixed batch-stream data processing. In order to improve the query efficiency of join computing, two different cache optimization mechanisms, full cache and incremental cache, are added to the Operator, which are suitable for different behavior patterns and applicable scenarios of mixed batch-stream data join computing.

Fig. 4 shows the mixed batch-stream data join method based on caching mechanism. In the full cache mode, the job has two types of data sources at the same time. The stream data source can be connected to streaming message middleware systems such as Kafka to consume real-time events, while the batch data source is usually connected to offline storage systems to load the full amount of data in specified data tables and distribute them to Operators responsible for executing mixed batch-stream data join. The mixed batch-stream data join Operator will construct these partition data into hash table cache. When the stream data is ingested, it can directly access the hash table cache for local memory query, which greatly reduces the communication and transmission cost of remote query. It is not difficult to find that this working mode is mainly restricted by the overall memory resources of the cluster, and is suitable for scenarios with small batch data scale.

The mixed batch-stream data join method based on incremental cache takes into account the scenario of huge batch data scale, and does not load the whole batch data into memory, that is, in Fig. 4, there is no Full Load procedure, and there is no need to use the Source Operator for batch data reading. The Flink job only creates a stream data source, when the mixed batch-stream data join Operator receives the stream data, it will first try to query the logically associated batch data from the local memory. If the data cannot be queried, the Operator will interact directly with the remote database system, and then add the query results to the incremental cache. If there are repeated queries in the future, it can achieve the same purpose as the full cache mode. The above method is naturally suitable for data skew scenarios, because if the stream data is evenly distributed, batch data will be frequently and alternately added to the incremental cache, so most queries will still be remote queries. However, in the scenario of data skew, the data in incremental cache will be relatively fixed due to frequent access, so most queries are local memory queries, which brings considerable computational efficiency.

In addition to the choice of cache type, the cached data format is also important. The cache improves query efficiency only if it is effectively hit, but it cannot be hit if the information required by the stream data never exists in the dimension data, which is easy to happen in incremental cache mode. If the reason for the above problems is that the queried data does not exist in the offline data source at all, frequent remote queries will occur, thus reducing the job efficiency. To this end, the proposed framework also caches the data that has not been hit, thus reducing the extra overhead caused by frequent queries. Algorithm 1 describes the specific process of cache building for mixed batch-stream data join. First, the primary key of the cache is calculated (line 1),

and the hit condition of the cache is judged. If the cache hits, the addition of dimension data is determined according to the validity of the judged cache (lines 2–7). If the cache is not hit and the offline data query result is empty, an empty query cache will still be created (lines 8–12), and if the query result is obtained, a query cache will be created (lines 13–15). The cache build process continues until the end of the job (line 16).

Algorithm 1: Cache building for mixed batch-stream data join

```
   Input: Stream data source: streamData; Data cache: cache
   Output: Dimension data: batchData
1  cacheKey = buileKey(streamData);
2  if cache.containsKey(cacheKey) then
3      queryData = cache.get(cacheKey);
4      if queryData.isEmptyCache() then
5          batchData.add(NULL);
6      else
7          batchData.add(queryData.getResult);
8  else
9      resultData = Query(cacheKey);
10     if resultData.isEmpty() then
11         cache = buildEmptyCache(cacheKey);
12         batchData.add(NULL);
13     else
14         cache = buildCache(cacheKey, resultData);
15         batchData.add(resultData);
16 return batchData
```

4. Join method based on hotspot awareness

On the basis of cache construction, this section focuses on the problem of data skew in mixed batch-stream data join computing and studies a data partitioning method based on hotspot awareness of stream data.

4.1. Problem description

In the field of big data processing, the occurrence of hotspot will cause data skew, and the problem directly caused by data skew is that the load of the cluster is unbalanced, so that the job cannot make full use of the cluster resources to achieve optimal computing performance. A known typical solution to data skew is to split the detected hotspot data into different computing resources, which requires the distribution of the data to be determined before running the job. Unfortunately, in the case of mixed batch-stream data processing, the stream data comes continuously after the job is started, which leads to the unknown distribution of the calculated data. Aiming at the above problems, a solution based on range partitioning is designed, which dynamically adjusts the partitioning rules according to the change of stream data skew. By sampling and analyzing the stream data currently being processed, the distribution of the next wave of stream data is predicted, the partition rules are adjusted, and the computing load is rebalanced.

Fig. 5 describes the mixed batch-stream data join framework based on hotspot awareness, which divides the stream data into different cycles, and predicts the distribution of data in the next cycle by analyzing the data in the current cycle. The framework

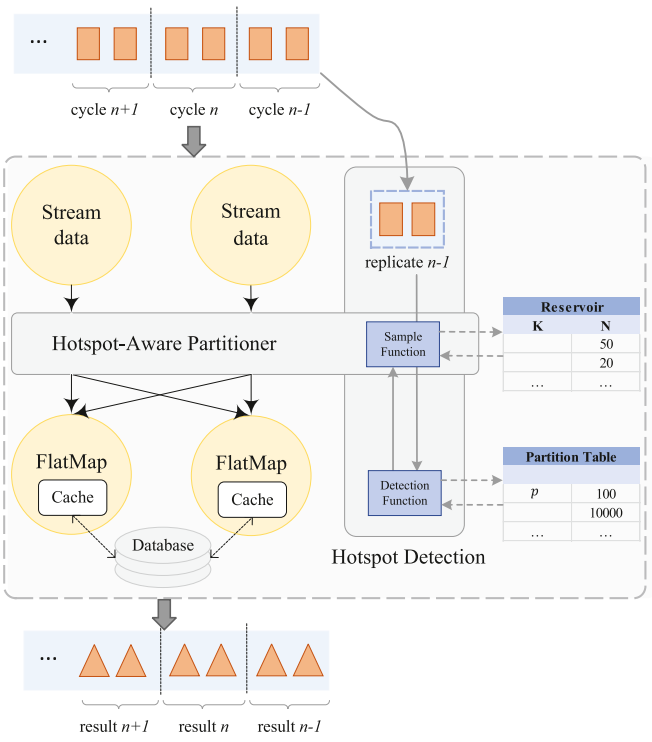


Fig. 5. Mixed batch-stream data join framework based on hotspot awareness.

includes three functions: (1) Hotspot detection, which effectively calculate the workload of each partition, as a basis for judging whether the data hotspot has changed, and guide the partitioner to start the sampling logic to adjust the partitioning rules; (2) Data partition, which is used to sample the stream data of each cycle and form a new partitioning rule. If it is judged that the data has been skewed, the sample function will be triggered to sample the data in $cycle_{n-1}$, and the applicable statistics will be calculated. Use the current quantiles array as the new partitioning rule, and consider that the partitioning rule also applies to the data in $cycle_n$; (3) Join calculation, which is responsible for performing mixed batch-stream data join.

4.2. Hotspot detection model

In order to predict the distribution of the incoming stream data, the stream data is logically divided into different cycles, and the data distribution of $cycle_n$ is predicted by analyzing the data of $cycle_{n-1}$. The framework we proposed divides cycles according to the amount of stream data, which is equivalent to setting a tumble count window. At the same time, in order not to affect the job performance when processing mainline pipeline data, $cycle_{n-1}$ will be copied to $replicate_{n-1}$ for analysis and sampling. As the core of the hotspot detection module, the detection function is mainly responsible for collecting relevant information from $replicate_{n-1}$ to determine whether the data hotspot has changed, and the basis for the determination is the degree of data skew between partitions. To describe this problem more concretely, use PS_j to denote the number of key–value pairs contained in partition j, as shown in Formula (1):

$$PS_j = \sum_{x=1}^{l_j} n_{k_x^j} \tag{1}$$

where $n_{k_x^j}$ represents the number of key–value pairs sharing k_x in partition j. Therefore, the average of the number of key–value

pairs contained in all partitions can be expressed as $\overline{MEAN}$, as shown in Formula (2):

$$\overline{MEAN} = \frac{\Sigma_{x=1}^{m} n_{k_x}}{n_p} = \frac{\Sigma_{x=1}^{n_p} PS_x}{n_p} \tag{2}$$

where n_p represents the number of partitions. Finally, the slope of the partitioned data *DS* can be defined, as shown in Formula (3):

$$DS = \frac{\sqrt{\frac{\Sigma_{x=1}^{n_p}(PS_x - \overline{MEAN})^2}{n_p - 1}}}{\overline{MEAN}} \tag{3}$$

It can be easily found that the molecule is a standard deviation calculation method, which can effectively measure the distribution difference of key–value pairs in different data partitions, while the average number of key–value pairs in the denominator can eliminate the influence of the number of key–value pairs on the data inclination in the cycle. It has been proved that the data hotspot detection strategy based on the number of key–value pairs [21] and the stream data hotspot detection strategy based on cycle [34] are authoritative in solving the load balancing problem. In order to calculate *DS*, the hotspot detection module will create a partition table, the data in the table is stored in the form of $< P, PS_j >$ pairs, and the total amount of data allocated in each partition in $replicate_{(n-1)}$ is recorded, where the rules for partitioning follow the current cycle's quantiles array. At the beginning of the cycle, a copy of each stream data in $replicate_{(n-1)}$ will be sent to the hotspot detection module, and then the detection function will continue to analyze the copy data and record the relevant status values in the partition table. At the end of the cycle, the *DS* of $cycle_{n-1}$ is calculated according to Formula (3). If the *DS* exceeds the threshold, it is determined that the data hotspot has changed, and then the next work is transferred to the partitioner module. The threshold needs to be set in consideration of specific application scenarios and tolerance for data skew.

In the specific process of hotspot detection. First, traverse the replicate data copy, and determine the partition number of the partition to which each piece of data belongs in the current cycle based on the working principle of the range partition, so as to increment the corresponding data item in the partition table. Then, according to the capacity of replicate and the number of data in partition table, the MEAN of the amount of data contained in all partitions is calculated, so as to calculate the data skew *DS* between partitions according to Formula (3). Finally, by comparing the *DS* and the threshold, it is judged whether hotspot migration has occurred, and the judgment result is returned.

4.3. Data partition optimization

The partitioner module mainly undertakes two functions: one is to sample the data of the previous cycle and count the distribution of keys in the sample data; the other is to calculate the partitioning rules applicable to the data of the next cycle according to the distribution, namely quantiles array. The scene of mixed batch-stream processing is dominated by massive real time data calculation, so the reservoir sampling algorithm is adopted in the sampling algorithm of this module, which can reduce the time complexity to O(n) even when processing unknown stream data.

In order to further improve the efficiency of data partition decision, the hotspot detection and sampling process are executed simultaneously, and create a reservoir table and collect sample data through the reservoir sampling. When $replicate_{(n-1)}$ is handed over to the detection function for analysis, the sample function in the partition module has entered the ready state. In addition, the data in the table exists in the form of $< K, N >$, K represents the unique key, and *N* represents the corresponding number, because the complete information of the sample data does not need to be used when calculating the quantiles array. According to the above strategy, this process not only eliminate the time cost of the sampling process itself, but also save the space overhead of saving data copies.

Algorithm 2: Data partitioning based on hotspot awareness

```
   Input: Streaming data: stream; Initial quantile array:
          initialQuantiles; Partition skew threshold:
          threshold
   Output: Partition result: partition
1  quantiles = initialQuantiles;
2  for cycle in stream do
3      replicate = cycle;
4      reservoir = sample(replicate);
5      for record in cycle do
6          partitionNo = determinePartition(record.k,
            quantiles);
7          partitions[partitionNo].collect(record);

8      if detection(replicate, quantiles, threshold) then
9          sort(reservoir);
10         for i = 1 to reservoir.length do
11             sum += reservoir[i].N;

12         mean = sum/partitions.length;
13         for i = 1 to reservoir.length do
14             count += reservoir[i].N;
15             if count ≥ mean then
16                 nextQuantiles[j++] = reservoir[i].K;
17                 count = 0;


18         quantiles = nextQuantiles;
```

As shown in Algorithm 2, in order for the task to run normally at the beginning of startup, an initial initialQuantiles array must be allocated first (line 1). For the replica data replicate in each cycle, sample the sample data set reserve, and distribute each stream data in the current cycle according to hotspot detection model to maintain normal data processing (lines 2–4). At the end of the current cycle, use hotspot detection model to determine whether hotspot migration occurs (lines 5–7). If hotspot migration occurs, sort the sample data (lines 8–9), and then calculate the average value of the sample data for each partition to obtain all the keywords located at the boundary (lines 8–12), and fill them into nextQuantiles array (lines 13–17), and an array of quantiles as the data partitioning rules for the next cycle (line 18).

In the aspect of cache construction, in order to reduce the time overhead and space overhead at the same time, incremental cache based on LRU algorithm is adopted, which can reduce the performance consumption caused by frequent data replacement and further improve the query efficiency. In addition, the working method of LRU is suitable for mixed batch-stream data join in data skew scenarios, because high stream data skew is accompanied by high-frequency access to batch data cache, and the cache maintained by each computing node will clear cold data and retain hot data. For each stream data record, first perform a memory-based search in the LRU cache if the search result (batch data record) is empty, then execute a remote query in the database, and add the query result to the LRU cache. If the search

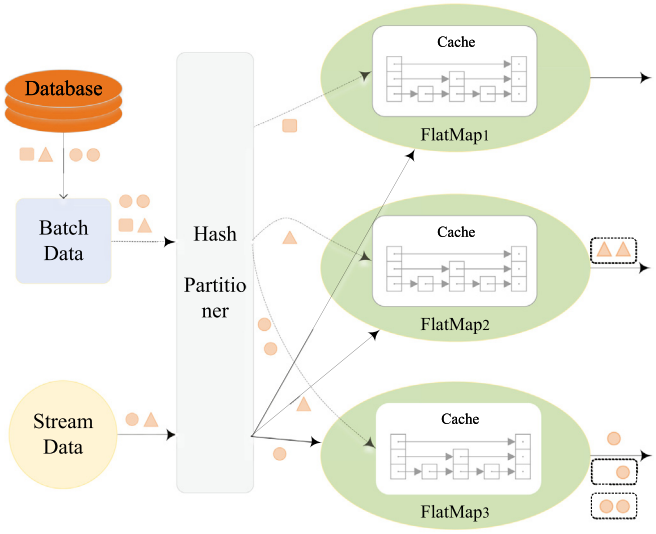


Fig. 6. Mixed batch-stream join computing model for range query.

result in the database is not empty, connect stream data record and batch data record and output, otherwise it is determined as an empty join.

5. Join method for range query

In this section, the mixed batch-stream data join operation is extended to more complex scenarios, and a join framework and optimization techniques for range query are studied.

5.1. Problem description

In complex applications such as OLAP, the connection conditions between data are rich and varied, and there will be a large number of storage requirements for data with interval attributes, and the connection of these data requires the execution of range query according to the fields with interval attributes. However, the above requirements cannot be fulfilled by relying only on the proposed equijoin operation. In addition, range queries also urgently need an efficient and functional index organization.

For range queries of the mixed batch-stream data join model by introducing support range queries skip list indexing technology as a cache to build solutions, and revolves around the series of range query oriented mixed batch-stream data join method is designed, and then in order to reduce latency and increase throughput as the goal, according to the characteristics of Flink and the skip list index to optimize query performance of cache.

Fig. 6 illustrates the mixed batch-stream data join computing framework for range query in detail. The batch data source ingests offline data from the database and distributes it to the FlatMap Operator to form subtasks and build caches, and the stream data source partitions the data in the same way. When the FlatMap Operator receives the stream data source, it performs range queries from the cache organized by the skip list index. When the data that meet the conditions are found, they are connected to the stream data in turn and output. This preloading method requires a certain time cost to block the stream data, but after the loading is completed, Flink no longer needs to frequently access the external database system during the calculation process, which not only saves the network communication overhead, but also increases the query efficiency. Therefore, it has huge advantages for stream computing scenarios that focus on real-time performance. In addition, the batch data source acts as a data partitioner, making full use of the overall resources of the cluster and breaking up the batch data without putting pressure on the memory.

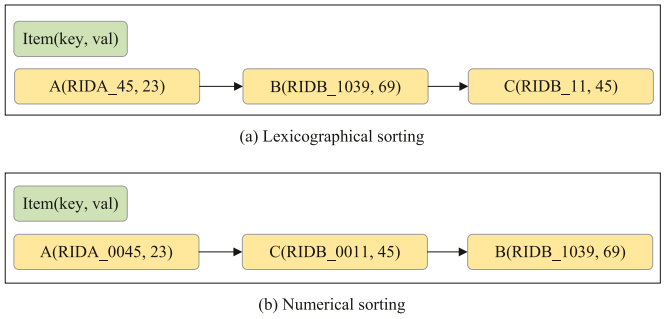


Fig. 7. Differences between lexicographic sorting and numerical sorting.

5.2. Cache build strategy

In the framework of mixed batch-stream data join oriented to range query, skip list is used as the cache structure of batch data, because skip list natively supports interval data connection. Moreover, from the performance point of view, the general mixed batch-stream data join job only requires high query performance, and does not frequently modify data records in batch data sources, which is very consistent with the characteristics of skip list. However, in a distributed environment, building a range query-oriented cache brings two problems: First, the traditional data partition method is only for a single keyword, which is not suitable for range query scenarios, and the skip list is an in-memory key–value storage structure, nor is it suitable for organizing data with multiple keywords. Multiple fields can be integrated into one keyword by string concatenation, and then partitioned by hash function. Unfortunately, in the mixed batch-stream join computing oriented to range queries, this partition method will cause the target batch data searched to be scattered on each remote node, resulting in a large amount of empty connections. If the broadcast mode is used to distribute stream data to each Operator, although empty connections can be avoided, it will undoubtedly increase the burden on the system and lead to low efficiency. Second, the skip list is a logically ordered index structure, and the keywords must be of comparable types. Therefore, the difference in the comparison method between lexicographical sorting and numerical sorting is also a problem that has to be considered. As shown in Fig. 7(a), the ASCII code corresponding to “0” in the keyword of data record B is smaller than the ASCII code of “1” in the corresponding position of the keyword of data record C, which will cause data record B to be ranked in front of data record C, resulting in illegal search result.

For problems with query fields integrity, the keywords extracted are firstly distinguished into partition keywords (partitionKey) and cache keywords (cacheKey) in the process of cache building. Partition keywords are used for data partition, and cache keywords are used for cache building. For problems with query fields comparability, the solution adopted is to complete the keywords, as shown in Fig. 7(b). First, convert the value in the keyword into a string of uniform length (the high-order bits are filled with “0”), and then spliced to the back of the equivalent query field, so as to avoid illegal index order.

Algorithm 3 describes the specific construction process of the distributed cache for range query. First, construct the partition key, which mainly includes two stages: extracting the equijoin field (line 1) and building the partition key (lines 2–3). Then, the building of the cache keyword mainly includes three stages:

Algorithm 3: Construction of fields in cache

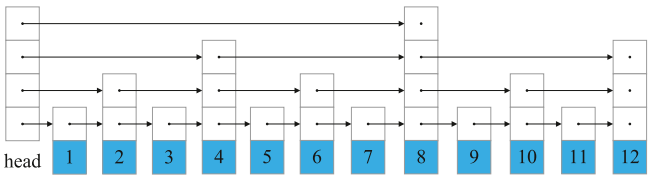
```
Input: Batch data record: batchRecord; Batch dataset:
       batchData
Output: Distributed cache: partitions
1  equalFields = extractEqualFields(batchRecord);
2  for equalField in equalFields do
3      partitionKey += equalField + "_";
4  rangeField = extractRangeField(batchRecord);
5  supplementedRangeField = supplement(rangeField);
6  cacheKey = partitionKey + supplementedRangeField ;
7  for batchRecord in batchData do
8      partitionNo = hash(partitionKey) % parallelism;
9      partitions[partitionNo].add(cacheKey, batchRecord);
10 return partitions
```

extracting the range query field (line 4), complementing the keyword (line 5) and constructing the cache keyword (line 6). The above two processes can effectively avoid the problems of empty connections and illegal query results in the connection calculation process. Finally, the building of the distributed cache includes two stages: determining the partition number (lines 7–8) and adding the constructed cache key to the partition cache (lines 9–10).

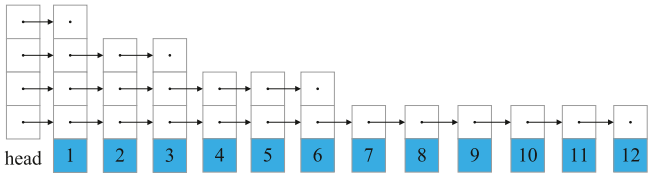
5.3. Query optimization

The typical skip list is implemented based on random algorithm, and its actual performance is still far from the theoretical value. Therefore, it is necessary to optimize the cache performance around the cache structure itself, combining the characteristics of Flink and skip list index. Fig. 8(a) illustrates the data query process with an ideal skip list with 12 nodes as an example. The ideal shape means that its index height distribution absolutely conforms to {1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3}, and only two comparisons are needed to locate the target when searching the record "12". However, the typical skip list uses random algorithm to generate index height for data, which cannot guarantee to build an ideal form. Therefore, it is an unstable data structure, and like binary search tree, it will degenerate into a form with the same search efficiency as singly linked list in extreme cases, as shown in Fig. 8(b). Through the above analysis, it is not difficult to find that there is the possibility of further optimization of the skip list. If the insertion position of data items can be predicted according to the data distribution of the data set, there will be a certain operating space to make the skip list close to the ideal shape, thus improving the stability of query efficiency.

If the computing framework can predict the position of each data record in the whole data set, it can accurately calculate the skip list cache that the data record should have and conform to the ideal index height. Skip list is essentially an ordered linked list, so the subscript of each data record can be determined by sorting. To construct an ideal skip list, it is necessary to determine the number of index layers according to the subscript position of data records, which means that the subscript position and the number of index layers are highly coupled. Once the subscript position changes, the established number of index layers is no longer applicable, and it is unrealistic to re-index all data records every time a new data is added. Therefore, the data set is first required to be invariant, that is, the data scale is determined and known. In the scenario of mixed batch-stream data join, batch data is often stored in offline database system, and it is hardly updated during job executing, which can be regarded as satisfying the above data invariance conditions. In addition, Flink



(a) Ideal situation



(b) Extreme situation

Fig. 8. Data query process under different forms of skip list.

can partition large data sets natively, and batch Operators not only read data from external data storage systems as data sources, but also act as partitioners. Therefore, it can make full use of the overall computing resources of the cluster and significantly relieve the CPU and memory pressure caused by sorting. The relationship between the subscript position of the data record and the number of index layers is shown in Formula (4):

$$Level(i) = \begin{cases} log_2(i+1), & (i+1) \in 2^k \\ Level[(i+1)\%2^{\lfloor log_2(i+1) \rfloor} - 1], & (i+1) \notin 2^k \end{cases} \tag{4}$$

where i is the subscript position of the data record. First, set the termination condition, and return the result directly when the subscript i is 0 or 1; When $(i+1)$ is an integer power of 2, the index height level is its logarithm based on $2+1$; Otherwise, as described in Formula (4), the index height decision method is called recursively, returning the final level.

Algorithm 4: Cache optimization for range join

```
Input: Initially constructed cache: partitions
Output: Distributed cache: partitionCaches
1  for partition in partitions do
2      sort(partition);
3      for i = 0 to partition.length do
4          batchRecord = partition[i];
5          level = determineLevel(i);
6          cacheKey = buildCacheKey(batchRecord);
7          skipLists[i].add(level, cacheKey, batchRecord);
8  partitionCaches = skipLists;
9  return partitionCaches
```

Algorithm 4 describes the specific building process of the distributed cache for range join. By traversing the batch data sets, the partition keyword partitionKey of each data record is constructed by using Algorithm 3, and the partition number partitionNo is determined by hashing modulus, and they are distributed to each partition. For each partition, firstly, the data in it is sorted to get an ordered subscript sequence (lines 1–2), then the index height of each data is determined by Formula (4) (lines 3–5), and then skip lists is added (lines 6–8), and finally it is returned as a distributed cache partitionCache (line 9).

After the cache build phase is complete, it can unblock the stream data source and execute the range query. However, before

constructing cache keywords, it is necessary to determine the upper bound and lower bound according to the range query interval, and finally join the query result set with the stream data in turn and output it. Therefore, when executing the range query, it only need to determine the location of the lower bound cache keywords, and then add the batch data in the search interval into the result set one by one by traversing the simple linked list until the upper bound cache keywords are accessed, which is the reason why use skip list to build the cache.

Algorithm 5: Range join calculation

```
   Input: Stream data source: stream; Batch data source:
          batch; Lower bound: lowerRange; Upper bound:
          upperRange
   Output: Join result: collector
 1 block(stream);
 2 Calculate partitionCaches according to batch data source;
 3 unblock(stream);
 4 for streamRecord in stream do
 5     partitionKey = partitions.partitionKey(streamRecord);
 6     partitionNo = hash(partitionKey) % parallelism;
 7     lowerField = extractRangeField(streamRecord) -
         lowerRange;
 8     upperField = extractRangeField(streamRecord) +
         upperRange;
 9     lowerCacheKey = partitions.cacheKey(lowerField);
10     upperCacheKey = partitions.cacheKey(upperField);
11     resultSet = getByRange(partitionCaches[partitionNo],
         lowerCacheKye, upperCacheKey);
12     if resultSet != null then
13         for batchRecord in resultSet do
14             collector.collect(join(streamRecord,
                 batchRecord));
15         else
16             collector.collect(streamRecord);
17 return collector
```

The specific query and join calculation process is shown in Algorithm 5: First, block the stream data source (line 1), and then unblock it after building the distributed partitionCaches according to batch data source through Algorithm 4 (lines 2–3). For each stream data streamRecord, the partition keyword and the partition number are obtained according to Algorithm 3 (lines 4–6), and the lower bound cache keyword lowerCacheKey and the upper bound cache keyword upperCacheKey of the stream data are calculated respectively according to lowerRange and upperRange according to Algorithm 3 (lines 7–10). Then execute the range query in the partition cache specified by partitionNo (according to Algorithm 3) to obtain the target result set resultSet (line 11). If the resultSet is not empty, traverse each batch data cache batchRecord in the resultSet, connect streamRecord and batchRecord into an output, and then collect it by the system and send it to the downstream, otherwise, it is deemed as an empty connection (lines 12–17).

6. Evaluation results

This section uses different Flink jobs with mixed batch-stream data join operation to experiment on different types of spatiotemporal data sets. The experiment firstly demonstrates the advantages of the proposed framework in throughput, latency and speed-up in the equijoin job, and verifies the effectiveness of the proposed optimization method in the data skew scenario. Then, the effectiveness of the proposed optimization mechanism is verified in range query jobs. Each group of experiments is run more than 10 times, and the records are taken after removing the maximum and minimum values.

6.1. Experimental setup

We run experiments on a 9-nodes OMNISKY cluster, and all nodes are connected with 10-Gigabit Ethernet. Each node has two Intel Xeon Silver 4110 CPUs @ 2.10 GHz (16 cores $\times$ 2 threads), 256 GB memory, and 1 TB SSD. The above hardware resources are virtualized into 129 computing nodes using docker + kubernetes, including 1 JobManager and 128 TaskManagers. Each TaskManager is assigned 1 CPU core, 8 GB memory, 20 GB SSD, and JobManager is assigned 8 CPU cores, 64 GB memory, 100 GB SSD. Flink version 1.8.0 (the most popular version) and 1.12.2 (the latest stable version) are chosen as the experimental environment, and the configuration files are configured according to the hardware environment as mentioned above.

Experiments use TPC Benchmark [35] and Ali Benchmark to verify the effectiveness of the proposed method. TPC is an authoritative data generation tool for testing the response time of complex queries of the system, and it can generate spatiotemporal datasets for commodity retailing. The dataset defines 8 tables of different types, each of which satisfies its own corresponding constraints, and there are connection conditions between each table. Ali Benchmark can generate massive spatiotemporal data sets for calculating various road traffic indicators in smart cities, including 7 data sources.

6.2. Test of equijoin

In order to intuitively verify the effectiveness of the proposed mixed batch-stream data join framework, the experiment first runs jobs composed of equijoin under the two Benchmarks. We first test the throughput and latency in TPC Benchmark jobs for different TaskManager numbers. Throughput refers to the number of calculation results output per second when the stream data is consumed at full speed, which can reflect the overall performance of the proposed computing framework. In the test of latency, the count window is set to 10,000, and the inflow speed of stream data is lowered to ensure that there is no extra data blocking. The calculation time used for outputting one stream data record is tested, which can reflect the performance of the proposed framework in response time.

Using TPC Benchmark to generate three offline data tables for product recall, which are click record table (10 M records, 12 GB), product table (10 M records, 0.1 GB) and user attribute table (10M records, 11 GB). Then access the real-time stream data source of consumer ID information, and connect with three offline tables in turn to calculate the product recall.

From the test results in Table 1, it can be seen that the proposed optimization method can significantly improve the throughput of mixed batch-stream data join operation. Under high TaskManager numbers, the improvement effect compared with Flink 1.8.0 can reach more than 4 times. It can be proved that the introduced caching mechanism can greatly reduce the extra overhead caused by remote data reading.

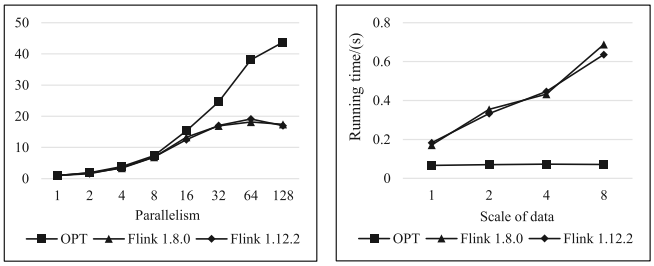
Table 2 shows the test results of latency under TPC Benchmark. It can be found that under the different number of TaskManager, our proposed optimization method can perform stably in terms of latency, and the effect is better than that of the two versions of Flink. When the number of TaskManagers increases, the latency of the Flink job increases obviously, while the latency of the job

Table 1
Throughput in TPC Benchmark for different TaskManager numbers.

TaskManagers	OPT	Flink-1.8.0	Increase percentage compared with Flink-1.8.0	Flink-1.12.2	Increase percentage compared with Flink-1.12.2
1	3267	1621	101.49%	1769	84.62%
2	6281	2940	113.58%	2858	119.72%
4	12587	5442	131.26%	6289	100.13%
8	24218	11219	115.86%	12313	96.68%
16	50154	21602	132.17%	22122	126.72%
32	80463	27437	193.26%	30158	166.80%
64	124622	29474	322.82%	33853	268.12%
128	143032	28232	406.62%	30069	375.68%

Table 2
Latency (ms) in TPC Benchmark for different TaskManager numbers.

TaskManagers	OPT	Flink-1.8.0	Increase percentage compared with Flink-1.8.0	Flink-1.12.2	Increase percentage compared with Flink-1.12.2
1	3.81	7.83	105.68%	7.12	87.02%
2	4.21	6.28	49.26%	6.45	53.37%
4	3.76	6.56	74.57%	6.08	61.97%
8	3.99	6.13	53.71%	6.16	54.62%
16	4.10	5.76	40.24%	6.07	48.01%
32	4.09	7.51	83.65%	7.62	86.53%
64	4.53	9.29	105.06%	9.72	114.58%
128	4.90	18.34	274.33%	19.91	306.40%



(a) Speed-up (b) Scalability

Fig. 9. Speed-up and Scalability in TPC Benchmark.

based on the proposed framework remains at a low level. It can be proved that the traditional mixed batch-stream data join method still has data read blocking even if only 10,000 stream data records are processed, and our proposed computing framework can perfectly solve the above problems.

In the test of speed-up and scalability, speed-up refers to the ratio of the throughput per unit time of a job running in N TaskManagers and a single TaskManager, as shown in Formula (5), and scalability refers to the ratio of job running time under unit batch data volume and n times batch data volume.

$$Speed-up = \frac{Throughput(N \quad TaskManagers)}{Throughput(1 \quad TaskManager)} \tag{5}$$

As can be seen from Fig. 9(a), when the number of TaskManagers is greater than 16, our proposed computing framework shows obvious advantages in speed-up, while when the number of TaskManagers reaches 128, the performance of jobs running in Flink decreases. The above situation shows that when the number of TaskManagers reaches 64, the job running in Flink has already appeared obvious data reading blocking, and when the number of TaskManagers continues to increase, the stability of Flink decreases due to more frequent remote data reading, thus reducing the overall efficiency of the job. On the contrary, our proposed computing framework has obvious advantages in speed-up. In the scalability experiment, the total calculation time per 10,000 stream data records for different batch data volume is recorded, as shown in Fig. 9(b). The running time of Flink jobs increase obviously with the increase of batch data volume, while the running time of the job running in the proposed computing framework is always in a short state, which is the advantage brought by the proposed cache mechanism, it obviously eliminates the influence caused by frequent remote reading of batch data.

Ali Benchmark is a mixed batch-stream computing of road event stream data and historical traffic batch data, and obtains the traffic capacity index of each turn of the road, including delay time, parking time and queue length. The batch data source consists of 6 tables (road traffic network data) with a total data volume of 63 GB.

The calculation logic for throughput and latency under Ali Benchmark is the same as the test in TPC Benchmark, and the results are shown in Table 3 and Table 4, respectively. It can be seen that the proposed computing framework still has obvious performance advantages under Ali Benchmark. In addition, when the number of TaskManagers is more than 16, the advantages of the proposed computing framework are more obvious than the test results in TPC Benchmark, because the amount of data in Ali Benchmark is larger and the computing logic is more complex, so the disadvantages of remote data reading in the traditional mixed batch-stream data join computing method are exposed earlier. Similarly, we show the experimental results of speed-up and scalability under Ali Benchmark in Fig. 10. It is not difficult to see that our proposed computing framework still shows strong advantages in speed-up and scalability in Ali Benchmark.

In order to describe the execution of different TaskManagers more clearly, we set the experimental environment as a real distributed environment composed of 7 computing nodes (1 JobManager and 6 TaskManagers) for the data skew scenario. Other configurations are the same as those described in Section 6.1. The experiment under the data skew scene is tested and analyzed with TPC Benchmark.

In the aspect of stream data source, the experiment uses a generator conforming to Zipfian distribution to generate data. Zipfian distribution can reflect the degree of data skew, so it is

Table 3
Throughput in Ali Benchmark for different TaskManager numbers.

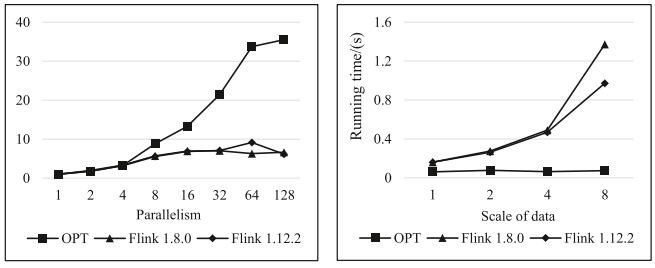
TaskManagers	OPT	Flink-1.8.0	Increase percentage compared with Flink-1.8.0	Flink-1.12.2	Increase percentage compared with Flink-1.12.2
1	30104	27746	8.50%	28122	7.05%
2	52890	54114	5.54%	51505	2.69%
4	97955	92352	6.07%	89643	9.27%
8	265821	158293	67.93%	156272	70.10%
16	400872	193026	107.68%	193030	107.67%
32	645143	195029	230.79%	199510	223.36%
64	1015615	174211	482.98%	257357	294.63%
128	1069050	184536	479.32%	172474	519.83%

Table 4
Latency (ms) in Ali Benchmark for different TaskManager numbers.

TaskManagers	OPT	Flink-1.8.0	Increase percentage compared with Flink-1.8.0	Flink-1.12.2	Increase percentage compared with Flink-1.12.2
1	33.97	36.11	6.28%	35.84	5.50%
2	39.04	41.63	6.64%	39.38	3.44%
4	42.48	44.32	4.33%	45.75	7.70%
8	33.90	76.37	125.28%	68.84	103.05%
16	26.49	115.74	336.90%	116.29	338.99%
32	31.78	213.13	570.72%	228.82	620.11%
64	41.10	581.39	1314.45%	558.56	1258.90%
128	47.22	1730.88	3507.01%	1605.18	3299.65%

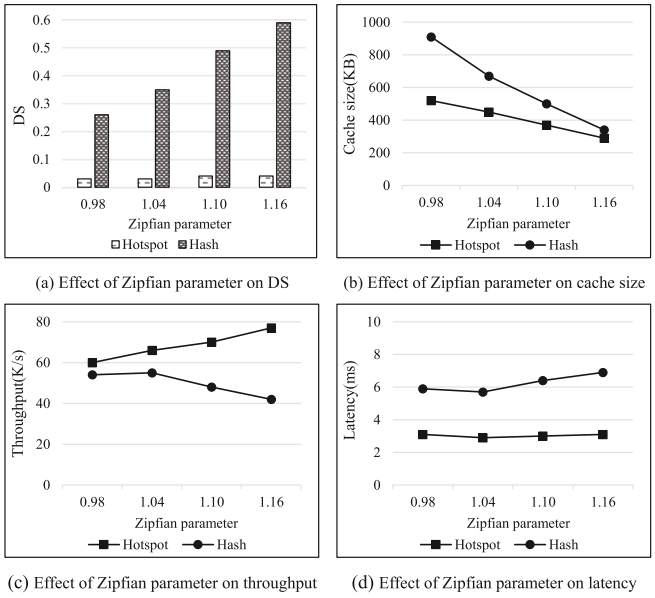
Table 5
Proportion of data access under different Zipfian parameters.

Z	1%	10%	20%	30%	40%	50%
0.98	69.01%	84.15%	88.85%	91.62%	93.61%	95.16%
1.04	78.71%	89.85%	93.00%	94.81%	96.07%	97.04%
1.10	86.42%	93.99%	95.95%	97.03%	97.78%	98.34%
1.16	91.89%	96.69%	97.82%	98.42%	98.83%	99.13%



(a) Speed-up (b) Scalability

Fig. 10. Speed-up and Scalability in Ali Benchmark.



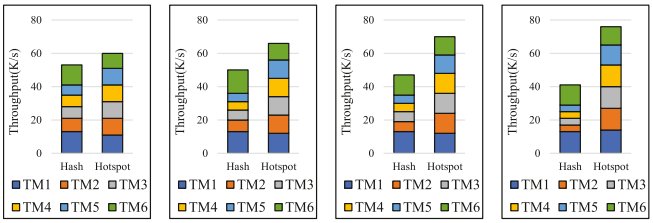
(a) Effect of Zipfian parameter on DS (b) Effect of Zipfian parameter on cache size

(c) Effect of Zipfian parameter on throughput (d) Effect of Zipfian parameter on latency

Fig. 11. Experimental results for different Zipfian parameters.

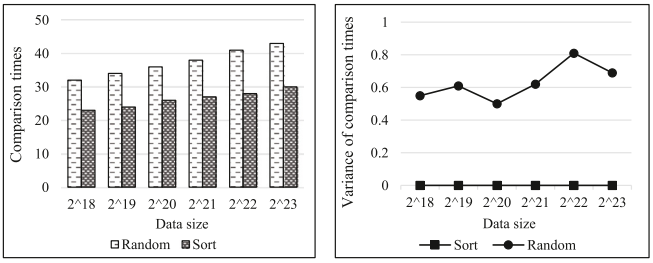
widely used in the research of data skew scene. Four different Zipfian parameters (Z), 0.98, 1.04, 1.10, and 1.16 are set to simulate different degree of data skew. Table 5 shows the probability of stream data being accessed under various Zipfian parameters and the proportion in the total data source. Among them, the first row represents the proportion of accessed data in the total data source, and the first column represents Zipfian parameter. Taking $Z = 1.16$ as an example, the data corresponding to the second column indicates that 91.89% of the accesses are concentrated in 1% of the total data source.

Fig. 11 shows the experimental results for different Zipfian parameters. As can be seen from Fig. 11(a), in the traditional Hash-based data partition mode, DS increases significantly with the increase of data skew, which will lead to unbalanced cluster load; In the proposed Hotspot-based data partition mode, DS is relatively stable and far lower than Hash partition under each Zipfian parameter, which means that the performance bottleneck caused by load imbalance between TaskManagers can be avoided. In terms of the influence of different Zipfian parameters on the cache size, with the increase of data skew, the cache size of jobs decreases obviously, as shown in Fig. 11(b). This is because the higher the data skew, the more concentrated the access to the batch data source, and therefore the higher the cache hit



(a) Z=0.98 (a) Z=1.04 (a) Z=1.10 (a) Z=1.16

Fig. 12. Workload distribution under different Zipfian parameters.



(a) Effect of data size on comparison times in skip list (b) Variance of comparisons times in skip list under different data size

Fig. 13. Performance of different types of skip lists.

ratio. In addition, the cache size of the proposed Hotspot-based data partition method is relatively low, which shows that this method can effectively avoid too many redundant copies, and is a lightweight and effective data partition strategy. In terms of throughput and latency, the proposed Hotspot-based data partition method can also bring obvious optimization effect, which is the positive impact of the proposed incremental cache mechanism, as shown in Fig. 11(c) and Fig. 11(d), respectively. When the data skew is higher, the data partition method based on Hash leads to the decrease of throughput, which just shows that the job efficiency will be greatly negatively affected under the condition of unbalanced load.

Fig. 12 shows the load distribution for each TaskManager (TM) with different Zipfian parameters, based on throughput. It can be found that with the increase of data skew, the throughput of each TaskManager in the traditional Hash partition method is obviously different. The throughput ratio of some TaskManagers is increasing, while the ratio of other TaskManagers is decreasing. This is because the Hash partition method cannot guarantee the uniform distribution of data to each TaskManager, which leads to obvious waste of computing resources in many TaskManagers. On the contrary, our proposed Hotspot-based data partition method can make each TaskManager have a balanced load and has significant advantages in throughput.

6.3. Test of range join

In the experiment of range join, the road congestion calculation module of Ali Benchmark is used, and the specific business is to calculate the traffic capacity index of signal intersection.

Road travel time table and turn travel time table are used for batch data, and real-time traffic information table is used for stream data. Because of the periodicity of road congestion, the correlation query based on time slice range is used in the analysis process, which is in line with the logic of mixed batch-stream data join calculation oriented to range query. In terms of data volume, the total amount of batch data is set to range from 2^{18} records to 2^{23} records, and the storage space is in the range of 132 MB to 3.96 GB.

Table 6
Throughput and latency of jobs based on FlatMap.

Data size	TaskManagers	Throughput (K/s)	Latency (ms)
2^{21}	8	16.32	301.40
	32	72.09	283.36
	128	187.28	372.55
2^{22}	8	14.59	311.98
	32	69.91	304.42
	128	176.08	397.42
2^{23}	8	12.43	351.11
	32	58.38	344.53
	128	168.77	420.08

Table 7
Throughput and latency of jobs based on random skip list.

Data size	TaskManagers	Throughput (K/s)	Latency (ms)
2^{21}	8	65.71	171.52
	32	186.19	164.01
	128	447.82	220.08
2^{22}	8	62.00	179.66
	32	173.45	169.98
	128	419.77	228.89
2^{23}	8	55.66	191.63
	32	161.40	178.92
	128	382.11	236.01

Firstly, the comparison times between the skip list based on random algorithm and the skip list based on sorting algorithm in a single query process are compared, and the comparison times means the memory access frequency, which can prove the superiority of the proposed optimization mechanism in query performance. Fig. 13(a) shows the comparison times of two kinds of skip lists in a single query under different data size. It can be seen that with the increase of data size, the average comparison times of the two kinds of skip lists increase steadily, and the average comparison times based on sorting skip list are lower, which can be reduced by about 25% compared with the average comparison times based on random skip list. Fig. 13(b) shows the variance of the number of single query comparison times obtained from 10 tests. Variance represents the degree of deviation between a set of data and its expected value. The larger the variance, the more unstable the query performance is. It can be found that the variance of single query comparison times based on random skip list is higher, because when the skip list is constructed by random algorithm, the shape of each query is inconsistent. The index height of the skip list based on sorting algorithm is determined according to the subscript value of ordered data, and it can meet the ideal shape every time it is constructed, so its variance is always 0, which shows that it is better than the skip list based on random algorithm in the stability of query performance.

Then, the performance of mixed batch-stream join for range query is demonstrated and analyzed in detail. The experiment uses FlatMap Operator in Flink to simulate range query as a baseline. In the calculation process, batch data in the database will be queried continuously according to the join key in the stream data source. Then the same logic is implemented using the proposed range query oriented mixed batch-stream join interface, including different implementations based on random skip list and sorting skip list, which are described in detail in Section 5.3. Throughput and latency are used to evaluate the computing performance.

Table 6, Table 7 and Table 8 show the experimental results of throughput and latency of the three implementations under different TaskManager numbers and different data scales respectively. It can be seen that under the condition of fixed

Table 8
Throughput and latency of jobs based on sorting skip list.

Data size	TaskManagers	Throughput (K/s)	Latency (ms)
2^{21}	8	87.33	149.63
	32	233.85	144.00
	128	506.02	194.63
2^{22}	8	81.59	156.34
	32	213.22	148.89
	128	476.40	200.88
2^{23}	8	74.71	165.32
	32	193.89	155.47
	128	442.76	204.43

TaskManager number and data size, the throughput and latency of the implementation based on the two skip lists are obviously better than those based on the traditional FlatMap in Flink. This is because when using FlatMap to join batch data and stream data, it will still query from the storage system on the remote host immediately and pull the data locally, which brings extra network overhead. On the contrary, the proposed implementation based on skip list eliminates the above overhead by establishing distributed local cache for batch data, thus bringing 2–6 times throughput improvement and reducing latency by more than 45%.

In terms of the effect of data size on query performance, the throughput of the three implementations decreases with the increase of data size, while the latency increases with the increase of data size. The performance of the implementation based on skip list is obviously better than that based on traditional FlatMap. This is not only because the cache mechanism introduced reduces the bottleneck of data communication, but also makes full use of the advantage of binary search in skip list index, which can reduce the complexity to O(log n). In the aspect of the influence of TaskManager numbers on query performance, the performance of the job based on two kinds of skip lists is obviously better than that based on traditional FlatMap, which proves once again that our proposed computing framework has great advantages in parallel and distributed computing.

7. Conclusion and discussion

We analyze the importance of mixed batch-stream data join in big spatiotemporal data applications, and the shortcomings of Flink in the above calculation model, then propose a new cache-based mixed batch-stream data join model that can support equijoin and range join, in which the optimization strategies mainly includes two aspects:

(1) We propose a join method based on hotspot awareness, which enables the cluster to achieve load balancing through hotspot detection and data partition optimization mechanism. Experimental results prove that the proposed optimization strategy can improve the efficiency of jobs involving mixed batch-stream equijoin;

(2) We propose a range query-oriented join method, which reduces the number of data reads and calculations by building a skiplist-based cache and further optimization. Experimental results show that the proposed optimization strategy can significantly improve the efficiency of mixed batch-stream range join.

Distributed systems still have many parts that need to be optimized and improved in the aspect of mixed batch-stream join calculation. The following issues can be studied in the future:

(1) The proposed BS-Join framework currently supports equijoin and range join, so it is necessary to explore a universal computing model and a variety of cache index mechanisms to deal with more complex join conditions;

(2) The join method based on hotspot awareness considers the load balancing on the number of connections, and does not distinguish between memory-based queries and external database-based queries. Therefore, it is necessary to improve the load skew evaluation algorithm and data partition method based on more abundant factors.

Funding

This research is supported by the National Key R&D Program of China under Grant No. 2019YFB1405302; and the NSFC, China under Grant No. 61872072, 62072087, 61972077, 61932004, 61732003, 62225203, U2001211, and U21A20516; and the Fundamental Research Funds for the Central Universities, China under Grant No. N2016009.

CRediT authorship contribution statement

Hangxu Ji: Conceptualization, Software, Methodology, Supervision, Validation, Writing – original draft. **Su Jiang:** Supervision, Validation. **Yuhai Zhao:** Methodology. **Gang Wu:** Writing – review & editing. **Guoren Wang:** Writing – review & editing. **George Y. Yuan:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The data that has been used is confidential.

References

[1] M. Delianidi, M. Salampasis, K. Diamantaras, T. Siomos, I. Karaveli, A graph-based method for session-based recommendations, 2021.
[2] Z. Zhu, S. Wang, F. Wang, Z. Tu, Recommendation networks of homogeneous products on an E-commerce platform: Measurement and competition effects, Expert Syst. Appl. 201 (2022) 117128.
[3] Y. Ye, L. Xiang, L. Chen, Y. Sun, G. Wang, RSkNN: kNN search on road networks by incorporating social influence, IEEE Trans. Knowl. Data Eng. 28 (6) (2016) 1575–1588.
[4] Y. Yuan, X. Lian, G. Wang, L. Chen, Y. Ma, Y. Wang, Weight-constrained route planning over time-dependent graphs, in: 2019 IEEE 35th International Conference on Data Engineering, ICDE, 2019.
[5] Y. Wang, Y. Yuan, H. Wang, X. Zhou, C. Mu, G. Wang, Constrained route planning over large multi-modal time-dependent networks, ICDE (2021) 313–324.
[6] H. Failure, H. Failure, S.D. Access, S.D. Access, L.D. Sets, L.D. Sets, S.C. Model, S.C. Model, M. Computation, M. Computation, The hadoop distributed file system: Architecture and design, Hadoop Proj. Website 11 (11) (2007) 1–10.
[7] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: Cluster computing with working sets, 2010.
[8] A. Phan, T. Phan, N. Trieu, A comparative study of join algorithms in spark, FDSE 12466 (2020) 185–198.
[9] S. Blanas, Y. Li, J.M. Patel, Design and evaluation of main memory hash join algorithms for multi-core CPUs, in: SIGMOD Conference, ACM, 2011, pp. 37–48.
[10] C. Balkesen, J. Teubner, G. Alonso, M.T. Özsu, Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware, 2013, pp. 362–373.
[11] S. Villarroya, J.R.R. Viqueira, J.M. Cotos, J.A. Taboada, Enabling efficient distributed spatial join on large scale vector-raster data lakes, IEEE Access 10 (2022) 29406–29418.

[12] E. Azhir, N.J. Navimipour, M. Hosseinzadeh, A. Sharifi, M. Unal, A.M. Darwesh, Join queries optimization in the distributed databases using a hybrid multi-objective algorithm, Clust. Comput. 25 (3) (2022) 2021–2036.
[13] P. Carbone, A. Katsifodimos, Kth, S. Sweden, K. Tzoumas, Apache flink : Stream and batch processing in a single engine, 2015.
[14] S. Adali, K.S. Candan, Y. Papakonstantinou, V.S. Subrahmanian, Query caching and optimization in distributed mediator systems, ACM SIGMOD Rec. 25 (2) (1996).
[15] M. Cai, A. Chervenak, M. Frank, A peer-to-peer replica location service based on a distributed hash table, in: Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference, 2004.
[16] H. Zhang, A. Goel, R. Govindan, Improving lookup latency in distributed hash table systems using random sampling, ACM (2003) 114.
[17] J. Rao, K.A. Ross, Making b^+-trees cache conscious in main memory, in: SIGMOD Conference, ACM, 2000, pp. 475–486.
[18] J. Boyar, K.S. Larsen, Efficient rebalancing of chromatic search trees, J. Comput. System Sci. 49 (3) (1994) 667–682.
[19] S. Hanke, The performance of concurrent red-black tree algorithms, WAE 1668 (1999) 287–301.
[20] W.W. Pugh, Skip lists: A probabilistic alternative to balanced trees, Commun. ACM 33 (6) (1990) 668–676.
[21] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, F. Li, HotRing: A hotspot-aware in-memory key-value store, FAST (2020) 239–252.
[22] S. Zhou, F. Zhang, H. Chen, H. Jin, B.B. Zhou, FastJoin: A skewness-aware distributed stream join system, IPDPS (2019) 1042–1052.
[23] Y. Yuan, X. Lian, L. Chen, G. Wang, J.X. Yu, Y. Wang, Y. Ma, GCache: Neighborhood-guided graph caching in a distributed environment, IEEE Trans. Parallel Distributed Syst. 30 (11) (2019) 2463–2477.
[24] Y. Cheng, Y. Yuan, L. Chen, G. Wang, C.G. Giraud-Carrier, Y. Sun, DistR: A distributed method for the reachability query over large uncertain graphs, IEEE Trans. Parallel Distributed Syst. 27 (11) (2016) 3172–3185.
[25] F. Guo, Y. Yuan, G. Wang, X. Zhao, H. Sun, Multi-attributed community search in road-social networks, ICDE (2021) 109–120.
[26] Y. Sun, Y. Yuan, G. Wang, An OS-ELM based distributed ensemble classification framework in P2P networks, Neurocomputing 74 (16) (2011) 2438–2443.
[27] S. Shang, L. Chen, Z. Wei, C.S. Jensen, J. Wen, P. Kalnis, Collective travel planning in spatial networks, IEEE Trans. Knowl. Data Eng. 28 (5) (2016) 1132–1146.
[28] Y. Li, F. Xiong, Z. Wang, Z. Chen, C. Xu, Y. Yin, L. Zhou, Spatial-temporal deep intention destination networks for online travel planning, IEEE Trans. Intell. Transp. Syst. 23 (3) (2022) 2688–2700.
[29] S. Shang, L. Chen, K. Zheng, C.S. Jensen, Z. Wei, P. Kalnis, Parallel trajectory-to-location join, IEEE Trans. Knowl. Data Eng. 31 (6) (2019) 1194–1207.
[30] S. Shang, L. Chen, Z. Wei, C.S. Jensen, K. Zheng, P. Kalnis, Parallel trajectory similarity joins in spatial networks, VLDB J. 27 (3) (2018) 395–420.
[31] S. Wang, X. Niu, P. Fournier-Viger, D. Zhou, F. Min, A graph based approach for mining significant places in trajectory data, Inform. Sci. 609 (2022) 172–194.
[32] K. Zheng, H. Su, B. Zheng, S. Shang, J. Xu, J. Liu, X. Zhou, Interactive top-k spatial keyword queries, ICDE (2015) 423–434.
[33] C. Luo, P. Wang, Y. Li, B. Zheng, G. Li, Efficient time-interval augmented spatial keyword queries on road networks, Inform. Sci. 593 (2022) 505–526.
[34] P. Lu, Y. Yue, L. Yuan, Y. Zhang, AutoFlow: Hotspot-aware, dynamic load balancing for distributed stream processing, Lecture Notes in Computer Science, 13157 (2021) 133–151.
[35] M. Barata, J. Bernardino, P. Furtado, An overview of decision support benchmarks: TPC-DS, TPC-H and SSB, in: WorldCIST, no 1, in: Advances in Intelligent Systems and Computing, vol. 353 (2015) 619–628.

Hangxu Ji received B.Sc. in Computer Science and Technology from the Northeastern University in 2013, and then received M.Sc. in Computer Technology and Theory in 2015. Currently, he is a Ph.D. candidate of Computer Science and Engineering College of Northeastern University. His research interests include distributed system, graph data management, and machine learning.

Su Jiang received his M.S. degree from the College of Computer Science and Engineering, Northeastern University, China, in 2022. His research interests include distributed system and machine learning.

Yuhai Zhao received the B.S., M.S., and Ph.D. degrees in computer science from Northeastern University in 1999, 2004, and 2007, respectively. He is currently a professor in the Department of Computer Science, Northeastern University, China. His research interests include data mining and bioinformatics.

Gang Wu received the B.S. and M.S. degrees in computer science from Northeastern University in 2000 and 2003, respectively, and then received Ph.D. degrees in computer science from Tsinghua University in 2008. He is currently an associate professor in the Department of Computer Science, Northeastern University, China. His research interests include knowledge graph, new database, and humanistic big data computing.

Guoren Wang received the B.Sc., M.Sc., and Ph.D. degrees from the Department of Computer Science, Northeastern University, China, in 1988, 1991, and 1996, respectively. Currently, he is a professor in the School of Computer Science and Technology, Beijing Institute of Technology, China. His research interests include XML data management, query processing and optimization, bioinformatics, high-dimensional indexing, parallel database systems, and P2P data management. He has published more than 100 research papers.

George Y. Yuan received the B.S., M.S., and Ph.D. degrees in computer science from Northeastern University in 2004, 2007, and 2011, respectively. He is currently a professor in the Department of Thinvent Digital Technology Co., Ltd., China. His research interests include probabilistic database, graph database, cloud database and data privacy.