GeoInformatica
https://doi.org/10.1007/s10707-022-00470-5

joinTree: A novel join-oriented multivariate operator for spatio-temporal data management in Flink

Hangxu Ji[1] · Gang Wu[1] · Yuhai Zhao[1] · Shiye Wang[2] · Guoren Wang[2] · George Y. Yuan[3]

Received: 1 March 2022 / Revised: 8 April 2022 / Accepted: 13 July 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract
In the era of intelligent Internet, the management and analysis of massive spatio-temporal data is one of the important links to realize intelligent applications and build smart cities, in which the interaction of multi-source data is the basis of realizing spatio-temporal data management and analysis. As an important carrier to achieve the interactive calculation of massive data, Flink provides the advanced Operator Join to facilitate user program development. In a Flink job with multi-source data connection operations, the selection of join sequences and the data communication in the repartition phase are both key factors that affect the efficiency of the job. However, Flink does not provide any optimization mechanism for the two factors, which in turn leads to low job efficiency. If the enumeration method is used to find the optimal join sequence, the result will not be obtained in polynomial time, so the optimization effect cannot be achieved. We investigate the above problems, design and implement a more advanced Operator joinTree that can support multi-source data connection in Flink, and introduce two optimization strategies into the Operator. In summary, the advantages of our work are highlighted as follows: (1) the Operator enables Flink to support multi-source data connection operation, and reduces the amount of calculation and data communication by introducing lightweight optimization strategies to improve job efficiency; (2) with the optimization strategy for join sequence, the total running time can be reduced by 29% and the data communication can be reduced by 34% compared with traditional sequential execution; (3) the optimization strategy for data repartition can further enable the job to bring 35% performance improvement, and in the average case can reduce the data communication by 43%.

Keywords Flink · Spatio-temporal data management · Data connection · Join sequence · Data repartition

✉ Gang Wu
wugang@mail.neu.edu.cn

Extended author information available on the last page of the article

1 Introduction

The construction of smart cities has become an irreversible historical trend in today's world, and also reflects the advanced stage of urban informatization. Smart sensor network is the basis of smart cities, which will generate massive spatio-temporal data. Therefore, interactive processing and intelligent analysis of massive multi-source spatio-temporal data is the core of smart city construction. At the level of data processing, data connection operation is one of the most frequently used operations in data interaction. For example, in intelligent transportation systems, a large amount of spatio-temporal data needs to be connected to predict road congestion [1, 2]; in the e-commerce system, spatio-temporal data need to be connected to achieve product recommendation and smart logistics [3]. Besides, the interaction of spatio-temporal data is also used in intelligent applications such as road network query [4] and route planning [5, 6]. Since data connection is a computationally expensive operation, and the above-mentioned applications are processed with massive amounts of data, these applications must use large-scale data centers and distributed systems as computing carriers.

With the development of distributed computing systems, Apache Flink [7] has gradually replaced Hadoop [8] and Spark [9] as the most popular system, which is characterized by batch-stream integration and higher efficiency. Compared with the traditional Hadoop MapReduce programming model, Flink has a variety of advanced Operators, which enable users to complete logically more complex big data jobs with less code. Among these advanced Operators, Join is the most frequently used. Join can connect two data sources according to defined rules, and output the connected result through a user-defined function, and finally output the intermediate data set to the next transform Operator or Sink Operator.

However, in the data connection operation, although the difference of join sequence will not affect the calculation result, the difference of calculation amount and network communication data amount caused by the join sequence will greatly affect the execution time of the job. In general, since users do not know which data join sequence will result in the least total running time, they will in most cases use the data source number as the fixed join sequence. If a multi-source data connection optimization algorithm is added to the job to reduce the running time with the best join sequence, the time of the optimization algorithm will increase exponentially when the number of data sources increases since the optimization problem is NP-hard [10]. To solve this, part of the solution strategy finds the optimal solution from the limited space of join sequences [21, 22], but the optimal join sequence is likely not in this space. There are also some studies to add index-based and genetic-based optimal join sequence solving mechanisms [11, 12, 16, 17] to jobs. Unfortunately, the above solutions are all adding optimization strategies to specific jobs, which are not universal and cannot solve the inconvenience caused by users calling the Join Operator multiple times. In addition, even if the optimal join sequence is obtained, the data repartition strategy in the Join Operator will still bring a huge amount of data communication between computing nodes, which further leads to low efficiency. Therefore, it is necessary to implement a lightweight optimization mechanism in Flink that dynamically adjusts the data join sequence, and an optimization mechanism in Flink that reduces the amount of repartition data for each join node, thereby reducing the job running time by using the shorter-running optimization mechanism. Furthermore, it is also urgent to combine the proposed optimization mechanisms to form a more advanced Operator in Flink for multi-source data connection operations.

Based on the in-depth study of the importance of data interaction in spatio-temporal data management and problems in the existing optimization strategies of multi-source data connection, combined with the defects of Join Operator in Flink, this paper proposes a novel join-oriented multivariate Operator with two key optimization strategies. This technology dynamically adjusts the join sequence to reduce the amount of calculation in the multi-source data connection operation, and further improves the job efficiency through the data repartition optimization strategy. The main contributions of this paper are summarized as follows:

(1) We propose a groundbreaking Operator that can support multi-source data connection operation for spatio-temporal data management. It first receives and decomposes the initial Flink Plan generated by the user program, then obtains the optimal join sequence through optimization techniques, and adds the repartition optimization mechanism, and finally reconstruct the new Flink Plan according to the optimization results;
(2) In order to reduce the amount of calculation in the multi-source data connection operation, we propose a dynamic adjustment strategy for the join sequence, by designing a computational cost model and introducing an ant colony algorithm to solve the optimal join sequence. Experimental evaluation verifies that this strategy can improve the job efficiency;
(3) In order to reduce the amount of data communication between computing nodes, we propose a repartition strategy based on data compression, by introducing a BloomFilter to construct global shared information to avoid data communication that does not meet the connection conditions. Experimental results show that this strategy can further reduce the running time of data connection operation.

The remainder of this paper is organized into 6 sections. Section 2 introduces the data connection framework in Flink and some data connection optimization technologies in distributed systems. Section 3 introduces the framework of joinTree. Section 4 explains the design and execution process of the optimization strategy for the join sequence. Section 5 describes the further optimization strategy for data communication between computing nodes. Section 6 presents the performance evaluation with respect to the running time and data transfer volume. Section 7 gives a brief conclusion.

2 Background and related work

In this section, we first summarized some of the implementation principles in Flink, including the principle of Join Operator for Equijoin, and the generation process of a Flink job with multi-source data connection operation, to verify the feasibility of our work. Then, the related work of data connection optimization technologies in distributed systems are explained, and the advantages and deficiencies of existing work are pointed out.

2.1 The principle of join operator for Equijoin in Flink

The Join Operator in Flink can support multiple types of data connection operations, of which Equijoin is the most common. Join needs to perform two steps when executing Equijoin operation: Ship Strategy and Local Strategy. Ship Strategy mainly includes two strategies: Repartition-Repartition (RR) and Broadcast-Forward (BF). RR assigns each partition

to a parallel join instance and all data of the partition is sent to its associated instance. This ensures that all elements that share the same join key are shipped to the same parallel instance and can be joined locally. BF will send a certain data source to a parallel instance that has another data source partition. After the data is transmitted to all parallel join instances using the RR or BF, each instance runs the local join algorithm. Join Operator has two strategies to perform local join: Sort-Merge (SM) and Hybrid-Hash (HH). The principle of SM is to sort the join attributes of two input data sources. If the local partition of the data sources is small enough, the sorting is done in memory, otherwise the external merge sorting is done. HH loads a smaller data source into the hash table, and then traverses another data source to pair according to the join key. If the amount of data in the smaller data source cannot be loaded into the memory, sharding the data source, and then perform a simple hash join on each shard. In the case of sufficient memory, the data is always stored in the memory.

The Join Operator uses a cost-based optimizer that automatically selects the optimal execution strategy for join operations, which helps the optimizer reason about existing data attributes by providing semantic information in user-defined functions. Therefore, the above optimization strategies can complete the data join operation with a smaller communication cost and computational cost, but its effect depends on the technical level of the programmers and their understanding of the details of the data sources, which will cause the optimization effect to be unstable.

2.2 Generation process of Flink job containing multi-source data connection operation

When a job containing multi-source data connection operation is submitted to the Flink cluster, OperatorTranslator will translate the program and start the reverse traversal with the Sink Operator to establish the initial Operators and the relationship between Operators. Then, various attributes in the multi-source data connection operation are encapsulated in the form of DataFlow. All the information saved above is called Plan in Flink, which is the initial representation of Flink job. Plan will be accepted by the OperatorOptimizer, and use Sink as the starting point to reversely traverse the Operators in Plan, and then generate OptimizerNode for each Operator. Then, connect these OptimizerNodes to generate the initial directed acyclic graph (DAG). After that, traverse the DAG and obtain the data information and attribute information of each OptimizerNode to estimate the cost of all the optional execution plans, and perform pruning operations based on this to generate the final OptimizedPlan. Finally, Optimizer abstracts OptimizedPlan into a JobGraph, in which Operators and intermediate data sets are abstracted into JobVertices, and data flows are abstracted into JobEdges.

Therefore, the core of this paper is to add multi-source data connection optimization strategies to the generation process of JobGraph to generate a new OptimizedPlan, and then generate an optimized JobGraph.

2.3 Data connection optimization technologies in distributed systems

The early MapReduce programming framework did not provide special Operators for data connection, so it is necessary to use basic programming interfaces to implement connection operations. The above method not only make programming difficult, but also the algorithm efficiency is low, so many data connection optimization technologies based on MapReduce

have been born, such as index-based optimization technology [11, 12], multi-source data connection optimization technology based on fixed number of data sources [13], star join optimization technology [14], etc.. Spark and Flink both implement Join Operator to support data connection, but both are 2-variate Operators. Therefore, the call sequence of Join Operators in a multi-source data connection operation has a critical impact on job efficiency [15]. At present, there have been many studies devoted to finding the optimal sequence of multi-source data connection, among which genetic algorithms and ant colony algorithms are more recognized [16, 17]. However, all of the above optimization techniques add corresponding optimization modules to jobs that contain data connection operations, which will significantly increase the amount of code in the programming process and bring difficulties to application development.

In the optimization of distributed data connection for data communication, optimization technologies based on information sharing are considered to be the most effective improvement strategy. The most representative one is the ComMapReduce [18] framework, which adds a coordination node responsible for receiving and distributing global shared information to the traditional MapReduce. The coordination node filters the data that does not match the data connection conditions through BloomFilter to reduce the amount of data in Shuffle, which has been proved to be an effective method long ago [19]. In short, a large number of studies have proved that the introduction of BloomFilter can effectively reduce the amount of communication data in distributed data connection operations.

3 Framework of joinTree

We propose joinTree, which can improve the efficiency of Flink jobs that contain multi-source data connection operation. joinTree firstly extracts and decomposes the original Flink Plan according to the user's programming logic, then solves the optimal join sequence through the dynamic cost estimation strategy, and reduces the communication cost through the repartition data filtering strategy, and finally reconstructs the Flink Plan according to the above optimization method. This section mainly introduces the overall structure of joinTree and the decomposition and reconstruction methods of Flink Plan. The specific implementation process of the two optimization strategies will be described in detail in Sections 4 and 5 respectively.

3.1 Overall structure

The essence of joinTree is a higher-level abstraction of multiple connection operations oriented to two data sources, that is, a higher-level abstraction of the Join Operator that is continuously called. Besides, we propose two optimization strategies in joinTree, among which the optimal join sequence solving strategy belongs to the optimization strategy between Join operations, and the repartition data filtering strategy belongs to the optimization strategy within Join operations. The overall structure of joinTree is shown in Fig. 1, which is mainly composed of four parts: Receiver, Decomposer, Optimizer and Reconstructor.

- **Receiver**: The Receiver is mainly responsible for receiving two parts of information: One is all the information in the initial Flink Plan generated by user programming logic (the generation process has been described in Section 2.2), including data node information, join node information, join attributes, etc., which are encapsulated

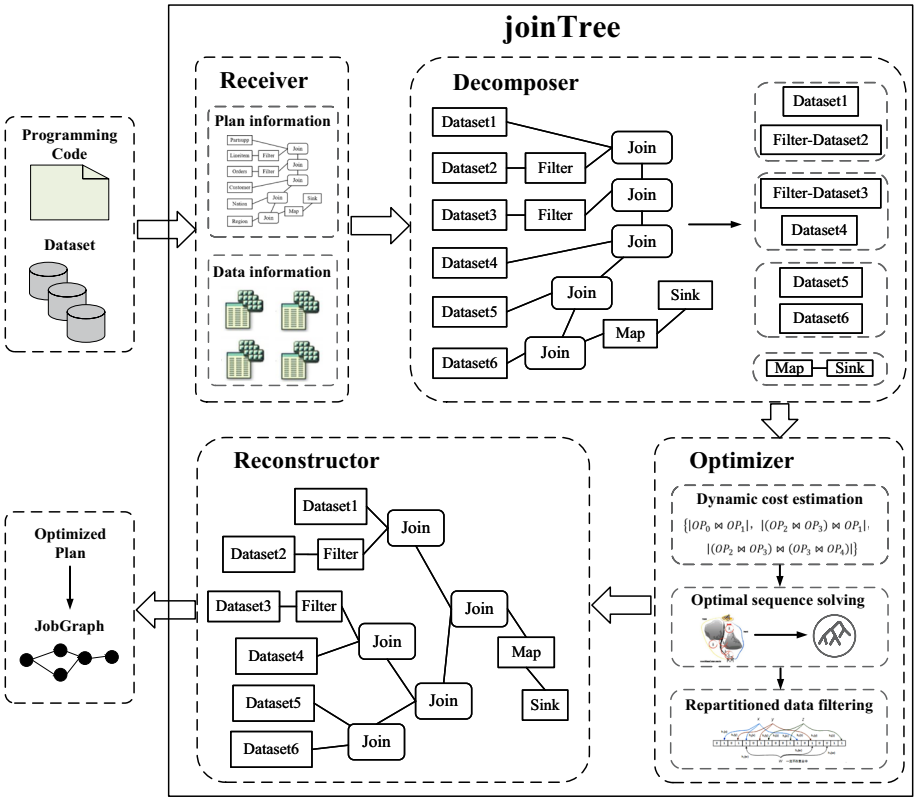


Fig. 1 Overall Structure of joinTree

in the .jar files submitted by the user. Since there are a large number of composite nested attributes in the join node, the Receiver also needs to serialize them into a byte stream that can be recognized by the downstream Decomposer. The other is the information in the input data sources, including the basic data information such as the number of records in the data sources and the size of the data sources.

- **Decomposer**: Since there may be other operations such as filtering, basic mathematical operations, etc. in a multi-source data connection job before join nodes, the function of the Decomposer is to decompose the "join" part and the "non-join" part of the received initial Flink Plan to generate different multi-source data connection strategies. Then, the decomposed elements are combined according to the join key of each join node to generate a large-scale solution space, which is finally sent to the Optimizer.
- **Optimizer**: The Optimizer first combines the decomposed packet data with the designed cost model, and dynamically estimates the cost of multi-source data connection operation under different join sequences. Then, according to the ant colony algorithm, the optimal execution sequence of multi-source data operation is solved. After that, BloomFilter is introduced into the calculation process of each 2-source connection operation, and the data communication between computing nodes in the process of repartition is reduced by filtering the data that does not meet the connec-

tion conditions. Finally, the join node information with the repartition data filtering strategy and the obtained optimal join sequence are sent to the downstream Reconstructor.

- **Reconstructor**: The Reconstructor combines the optimal join sequence solved by the Optimizer with the elements decomposed by the Decomposer to re-establish the new Flink Plan based on the construction method of binary tree. Then, as described in Section 2.2, the new Flink Plan generates an OptimizedPlan from further cost estimates of Flink and is eventually abstracted into a JobGraph.

3.2 Decomposition and reconstruction of Flink Plan

Because Plan is the initial representation of Flink programming logic, in order to complete the adjustment of join sequence, it first need to decompose the original Plan, and find the optimal join sequence based on these decomposed elements, and finally reconstruct these elements according to the optimal join sequence.

The process of Plan decomposition mainly includes two parts: disassembly and grouping. The disassembly strategy has been described in Section 3.1, which mainly disassembles the Plan into "join" part and the "non-join" part. Later optimizers do not need to estimate the cost based on all combinations of disassembled elements, because elements with the same join key can adjust the join sequence. Therefore, it is necessary to group the disassembled elements according to the corresponding relationship of join keys.

The core of joinTree is the high-level abstraction and encapsulation of multiple 2-source data connection operations, so it is necessary to find the join keys corresponding to each 2-source data connection operation first, and then trace the source based on these join keys to find all the corresponding input data sources. Figure 2 depicts the specific process of traceability and data grouping based on join keys. Depth-first traversal with Join5 as the root node. First, Join0 node is found in the post-traversal, and the Dataset0 and Dataset1 corresponding to their JoinKey(a) are found, and their corresponding codes are stored in the SubSet(a). Similarly, the data sources corresponding to their JoinKey(b) and JoinKey(c) can be found. Then, when the Join2 node is traversed and the Join2's join key and Join0's join key are found to be a, the SubSet(a) indexed by a is obtained, which is merged with the code of another data source corresponding to the JoinKey(a) in Join2, and the set SubSet(a) corresponding to the JoinKey(a) in the mapping set is updated. Finally, the above process is repeated until all the grouping sets indexed by the join keys are found.

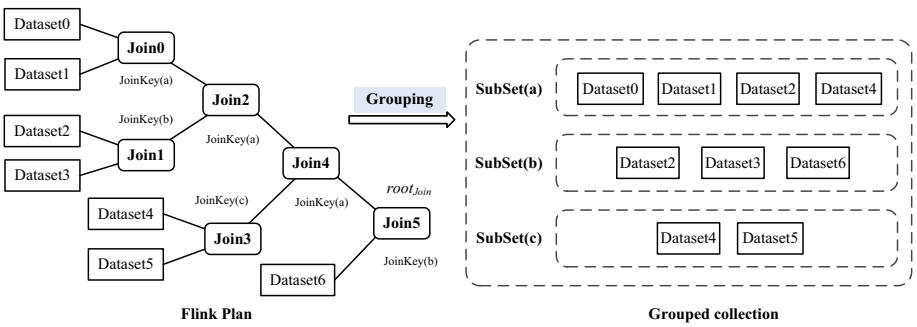


Fig. 2 Traceability and data grouping process based on join key

After solving the optimal join sequence and introducing the optimization strategy for repartition data communication cost, it is necessary to reconstruct the decomposed elements into a new Plan that can be recognized by Flink. Because the process of multi-source data connection can be modeled as a binary tree, the new Plan is reconstructed based on the decomposed elements, combined with the solved join sequence and binary tree construction strategy.

Algorithm 1: Decomposition and Reconstruction of Flink Plan.

Input: The original Flink Plan; Input data source information
Output: Reconstructed Flink Plan

```
1  i = -1; E ← ∅;
2  while traversal is not ended do
3      traversal.preVisit(V);
4      while traversal in postVisit(V) do
5          if The current node V is a Join node then
6              for k = 0; k ≤ 1; k++ do
7                  V_pre ← V.getInputNode;
8                  if The V_pre is not a Join node && joinKey ∈ V_pre then
9                      subSet[++i].add(V_pre)
10                 else if The V_pre is a Join node then
11                     if V.joinKey == V_pre.joinKey then
12                         subSet[i] ← subSet[i] ∪ getsubSet(joinKey);
13                     else
14                         subSet[i] ← subSet[i] ∪ trace(V_pre);
15 Solution of optimal join sequence J = {J_1, J_2, ···, J_m} ;
16 for n = 1; n ⩽ m; n++ do
17     if ∄ J_n.input ∈ E then
18         joinTree.establish(J_n, J_n.input_1);
19         joinTree.establish(J_n, J_n.input_2);
20     else
21         joinTree.establish(J_n, J_n.input_1);
22         joinTree.establish(J_n, J_n.input_2);
23         E.remove(J_n.input);
24     E ← J_n;
25 joinTree is converted into a new Flink Plan;
```

Algorithm 1 describes the decomposition and reconstruction process of Flink Plan in detail, which is implemented based on Flink's Visitor Pattern. In the Plan decomposition phase, first perform the initialization operation (line 1), then traverse the join node and the data source node in the initial Flink Plan (line 2), and use preVisit() to indicate that node V was initially visited (line 3). When traversing to the end to leave node V (postVist(V)), determine the data source group corresponding to the current join node in a bottom-up manner (lines 4-9), and then determine the data source group corresponding to the parent join node of the current join node (lines 10-14). If the join key of the parent join node is the same as the join key of the current join node, the data sources corresponding to the two nodes are merged. The reconstruction process of Plan is based on the optimal join sequence (line 15) and the construction method of the binary tree (lines 16-24). By traversing the join sequence and judging the relationship between the current join node and its corresponding data sources, a binary tree is constructed to generate the join tree of the multi-source data connection operation part. Finally, generate a new Flink Plan according to the generated join tree and the construction method mentioned in Section 2.2 (line 25).

4 Optimization strategy based on join sequence adjustment

From the perspective of reducing the amount of calculation in data connection operation, this section studies an optimization strategy based on dynamic adjustment of the join sequence, and improves the efficiency of multi-source data connection operation by finding the join sequence with the least total calculation amount.

4.1 Problem description

Since the multi-source data connection operation satisfies the commutative law and the associative law, the operation can be performed in different join sequences, but it does not affect the final output result. We first formally define the problem studied in this section, the multi-source data connection optimization for join sequence can be described as follows:

Definition 1 (optimal join sequence). Given a multi-source data connection operation, construct all its possible join sequence set S=$\{ s_1, s_2, \cdots, s_n \}$, calculate the cost of each element in S through cost estimation method C(s_i)($1 \le i \le n$), and finally find the join sequence C$(s_i)_{min}$ that meets the minimum cost.

Unfortunately, as the number of data sources increases, the number of elements in the set S will increase exponentially. Assuming that the number of data sources is N, then the total number of internal connection operations is (N-1), and the number of elements in the join sequence set Size(S) can be expressed as Formula (1) [20]:

$$Size(S) = \frac{(2N-2)!}{(N-1)!} + 2 \times N! \tag{1}$$

Although a better join sequence can significantly reduce the amount of calculation, the process of optimizing the optimal sequence will still incur huge costs. Therefore, most studies divide elements in set S into two categories, one is the linear tree set(LTS) containing $(2 \times N!)$ elements, the other is the multi-path tree set(MTS) containing $\frac{(2N-2)!}{(N-1)!}$ elements, and then only search for the optimal connection sequence in the LTS [21, 22]. However, the number of elements in the MTS is much greater than the number of elements in the LTS, so the optimal join sequence is more likely to appear in the MTS. In summary, it is necessary to search for a lightweight optimization algorithm in a larger space to find the optimal join sequence.

4.2 Design of computational cost model

Before solving the optimal join sequence, it is necessary to design a cost model based on the relevant information of the input data sources and the specific process performed by the multi-source data connection operation in Flink as the basis for the solution.

According to the description in Section 2.1, combined with the characteristics of multi-source data connection operations in Flink, it is believed that the repartition Ship Strategy and the local Hybrid-Hash join strategy are more universal [23]. Therefore, we design the total cost of the multi-source data connection operation to be the sum of the cost of the repartition process of every two data connection and the sum of the cost of the local hash join, which can be expressed as Formula (2):

$$\begin{aligned} Cost_{Total} &= Cost_{Repartition} + Cost_{Hash} \\ &= Com_{join} \times w_{Com} + IO_{join} \times w_{IO} + CPU_{join} \times w_{CPU} \end{aligned} \tag{2}$$

where the cost of the repartition phase = $\mathrm{Com}_{join} = D1.\mathrm{Size} + D2.\mathrm{Size}$, the sum of the sizes of the two data sources. In order to ensure the robustness of the cost estimation strategy, it is assumed here that the number of computing nodes in the cluster is large enough, then the repartition phase will generate the transmission between the computing nodes of the entire data source, ignoring the case of local transmission.

The cost of the hash join phase can be decomposed into the sum of the cost of the Build process and the cost of the Probe process, assuming $D1.\mathrm{Size} < D2.\mathrm{Size}$. During the Build process, Flink loads two data sources $D1$ and $D2$ into memory, partitions them with a hash function for their join keys, and writes them to disk. During the Probe process, data is read by partition, and hash joins are made based on partitions in memory. In the case of sufficient memory, try to save the complete partition in memory as much as possible. In this way, after the Probe starts, the memory already has as many $D1$'s hash tables as possible, which ensures the maximum utilization of the memory cache. For example, keep the hash table of the first partition $P0$ of $D1$ and the data of $P0$ of $D2$ in the memory during the Build process, so that at the end of the Build, there is no need to load P0 from the disk to the memory again before proceeding to the next Probe. According to the above description, the space overhead of the intermediate result generated by this process is the sum of the read and write of the two data sources and the constructed hash table. In the Flink memory model, the memory for allocating intermediate result data is limited to 40 percent of total Flink

memory. Due to the complex dependencies of various components in Flink, we will not optimize the memory allocation ratio of each component here. Therefore, we use Formula (3) to estimate the cost of IO_{join} in the local hash join process (assuming that the memory is not large enough):

$$IO_{join} = (D1.Size + D2.Size) \times 2.5 - TotalFlinkMemory \times 0.4 \tag{3}$$

According to the time complexity of hash join O(m+n), we formulate the cost of CPU_{join} as Formula (4):

$$CPU_{join} = (D1.Size + D2.Size) \times factor_{cpu} \tag{4}$$

where $factor_{cpu}$ is the CPU calculation factor for hash join.

4.3 Solution of optimal join sequence

Finding the sequence with the lowest cost among a large number of feasible join sequences can be regarded as a special kind of traveling salesman problem (TSP), which takes join nodes as the vertices to calculate the optimal solution. In this paper, the maximum and minimum ant colony system (MMAS) [24] is used to solve the optimal connection sequence, which is easier to achieve the global optimal state. Unlike the traditional approach to solving the TSP, the cost between each two join nodes is unknown, and the cost between each join node and the other join nodes is determined each time a join node is visited. Therefore, it is necessary to design a dynamic cost estimation method combined with MMAS to solve the optimal join sequence.

According to the grouping method described in Section 3.2, the cost of the initial input data sources is first calculated. When there are only two data sources $D1$ and $D2$ in the current group G_i, calculate the cost directly according to Formula (2), and save the cost $Jcost$ and the mapping relationship between $Jcost$ and its data sources. When there are m ($2 < m$) data sources in the group G_i, adopt the greedy strategy and Formula (2) to calculate $m-1$ minimum costs, and save these costs $\{Jcost_0, Jcost_1, \cdots, Jcost_{m-2}\}$ and the mapping relationship between costs and data sources. Finally, the multi-source data connection operation generated by n data sources can generate a cost set containing $n-1$ elements $C = \{Jcost_0, Jcost_1, \cdots, Jcost_{n-1}\}$ and the data source relations corresponding to these elements $R = \{J_0^{D01,D02}, J_1^{D11,D12}, \cdots, J_{n-1}^{D(n-1)1,D(n-1)2}\}$, which are the initial input of the optimal join sequence solution.

Next, combine Algorithm 2 to introduce the further calculation of the dynamic cost between join nodes, that is, the dynamic weight between nodes of the MMAS. Assume that there is a join node set N_v that has been visited and a join node set N_w that has not been visited at a certain time. In the process of traversing all unvisited join nodes, if there is no join relationship between the current join node W_i and all the visited join nodes N_v, the cost of the current join node corresponding to its initial data sources is added to the cost set C_{dy} (lines 1-8). If W_i has a connection relationship with one join node V_i that has been visited, the total join cost of W_i and the initial data sources in V_i is calculated, and then added to C_{dy} (lines 9-11). If W_i has a connection relationship with two join nodes V_i and V_j that have been visited, the total join cost of V_i and V_j is calculated, and then added to C_{dy} (lines 12-13).

Algorithm 2: Calculation of Dynamic Costs.

Input: Visited join node set N_v; join node set waiting to be visited N_w
Output: Cost set for unvisited join nodes C_{dy}

```
for W_i ∈ N_w do
    for V_i ∈ N_v do
        if ∃ a join relation between W_i and V_i then
            T ← V_i;
    n = getSize(T);
    switch n do
        case 0: do
            C_dy ← Cost_Total(W_i);
        case 1: do
            D ← getDataset(T_1);
            C_dy ← Cost_Total(Cost_Total(W_i), D);
        case 2: do
            C_dy ← Cost_Total(Cost_Total(T_1), Cost_Total(T_2));
```

Due to the nature of the multi-source data connection operation, the cost of all the next reaching vertices changes dynamically when the ants are at a certain vertex. Therefore, it is necessary to combine the above dynamic cost estimation method to design the optimal join sequence solution strategy based on MMAS. Formula (5) describes the transition probability of an ant k at vertex i to vertex j:

$$\eta_{ij}(t) = \frac{1}{C_{dy}(j)}, \qquad j \in N_i^k$$
$$p_{ij}^k(t) = \begin{cases} \frac{[\tau_{ij}(t)]^\alpha [\eta_{ij}(t)]^\beta}{\Sigma_{j \in N_i^k} [\tau_{ij}(t)]^\alpha [\eta_{ij}(t)]^\beta}, & j \in N_i^k \\ 0, & j \notin N_i^k \end{cases} \tag{5}$$

where α is the factor of information heuristic, β is the factor of information expectation, $\eta_{ij}(t)$ is the inverse of the cost, and $\tau_{ij}(t)$ is the concentration of the pheromone confined between τ_{min} and τ_{max}. When all the ants walk all the paths, the ants on the path with the least cost will release pheromone, while other paths will volatilize pheromone. The calculation process is shown in Formula (6):

$$\tau_{ij}(t+n) = (1-\rho)\tau_{ij}(t) + \Delta\tau_{ij}^{best}$$
$$\Delta\tau_{ij}^{best} = \frac{Q}{L} \tag{6}$$

where ρ is the factor of volatilization, Q is the factor of release, $\Delta\tau_{ij}^{best}$ is the pheromone contributed by the lowest cost ant k on Edge_{ij}, and $L = \min(L_k) = \min(\Sigma_i^N Cost_{Total}(i))$ is the sum of all costs traversed by ant k.

The time complexity of dynamic cost calculation is $O(\frac{n(n-1)}{2})$, which can be obtained in Algorithm 2, where n is the number of join nodes. The time complexity of solving the optimal join sequence is $O(\frac{n(n-1)}{2}mt)$, where m is the number of ants and t is the number of iterations. Since the settings of m and t are generally linearly related to n, when the number of join nodes is large, the time complexity can be considered as $O(n^4)$.

Algorithm 3 describes the process of combining the dynamic cost estimation method with MMAS to solve the optimal join sequence. First initialize the parameters required by MMAS and the upper and lower bounds of the pheromone concentration (lines 1-2). Then combine the dynamic cost estimation method with MMAS to make the ants visit all the vertices and calculate the path with the lowest cost in this iteration process (lines 3-8). If no better join sequence is found after completing a certain number of iterations, it means that the algorithm has fallen into a local optimal solution, which requires restoring the pheromone concentration on each path to the initial value τ_{max} (lines 9-12). After each round of iteration is completed, the pheromone will be updated according to Formula (6). Until the maximum number of iterations is completed, the global optimal path will be generated, that is, the final optimal join sequence will be obtained (lines 13-15).

Algorithm 3: Solving the optimal join sequence.

Input: Cost set $C = \{Jcost_0, Jcost_1, \cdots, Jcost_{n-1}\}$, data source relations $R = \{J_0^{D01,D02}, J_1^{D11,D12}, \cdots, J_{n-1}^{D(n-1)1,D(n-1)2}\}$

Output: Optimal join sequence

1 Initialize parameters α, β, ρ and Q;
2 Initialize pheromones τ_{min} and τ_{max};
3 **for** *itr = 1 to MAX_ITR* **do**
4 **for** *ant = 1 to ANT_NO* **do**
5 Calculate the cost set for unvisited join nodes C_{dy};
6 Calculate the transition probability $p_{ij}^k(t)$;
7 Use the roulette algorithm to determine the next visited vertex j;
8 Record *path* with the least cost;
9 **if** *The cost of the current iteration is lower* **then**
10 Update current path;
11 **if** *Stuck in a local optimum* **then**
12 $pheromones \leftarrow \tau_{max}$;
13 **else**
14 Pheromone volatilization and release;
15 Control the pheromone between τ_{min} and τ_{max};

5 Optimization strategy based on repartition data compression

In this section, based on the optimization strategy oriented to the adjustment of the join sequence, the research is carried out from the aspect of the network communication cost in the distributed data connection operation, and the further optimization algorithm based on repartition data compression is introduced in detail.

5.1 Problem description

Between the two data transmission strategies introduced in Section 2.1, most data connection operations use the RR strategy because of its universal applicability, we first introduce the definition of repartition data, and the process of RR strategy is shown in Fig. 3.

Definition 2 (repartition data). Given a data connection operation, Join uses the same partition function to partition the join keys of the two data sources, and sends the partitioned results to each computing node through the network. The total amount of data received by each computing node is repartition data.

5.2 Repartition data compression and filtering

To minimize the transfer of data between compute nodes, this strategy first uses the smaller of the two data sources to compress its portion in each partition, and then combines the compressed information to generate final global shared information and send it to each partition. Finally, another data source is filtered based on the global shared

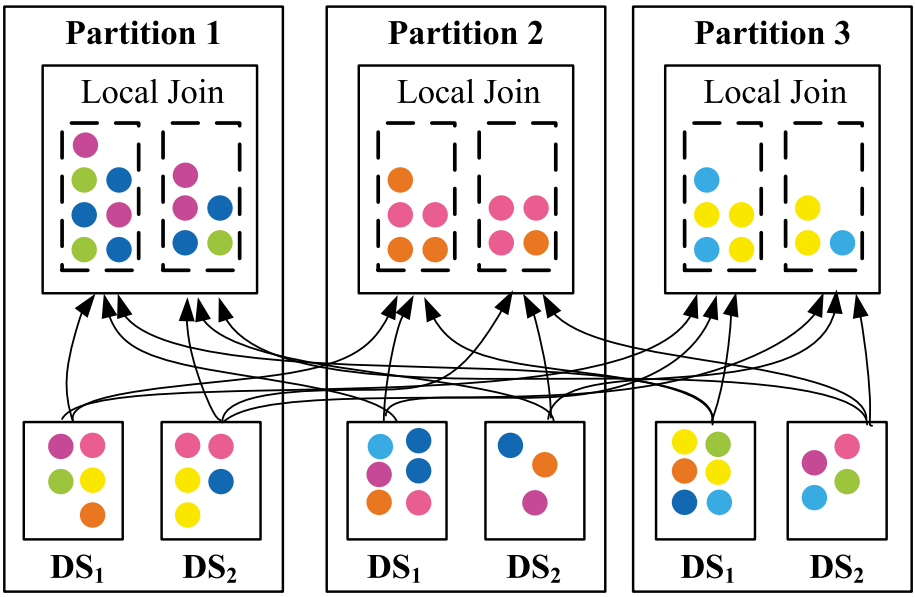


Fig. 3 The process of RR strategy

information to minimize repartition data that does not meet the join conditions. The overall framework of this optimization strategy is shown in Fig. 4:

However, in the process of data repartition, most of the transmitted data does not meet the join conditions, which will lead to a large amount of useless data transmission and calculations, thereby reducing the efficiency of the entire job. Therefore, the basic idea of the further optimization strategy described in this section is to introduce a data filtering mechanism before the join calculation, which can eliminate the higher time-consuming useless data communication and calculations through less time-consuming data filtering calculations.

During the repartition data compression, the strategy uses a BloomFilter to build the partition shared information. BloomFilter is a highly space-efficient data structure represented by a bit vector, which can efficiently determine whether an element belongs to a set at the cost of a certain False Positive. Therefore, users can set the appropriate bit vector size based on hardware resources to find the optimal balance between filter construction efficiency and data transmission volume, and ensure the final output is correct. The construction process of shared information is mainly divided into the following three phases:

(1) **Construction of BloomFilter:** Since the data to be connected is stored in partitions, it is necessary to perform BloomFilter compression in each partition, and the generation process of the bit vector is the core of building the BloomFilter. First create a bit vector of length m bits for each partition, and then generate k hash values $\{h_1, h_2, \cdots, h_k\}$ for the join key of each piece of data in the partition. We choose MurmurHash3 non-encrypted hash function, because for the join keys with strong regularity, the random distribution characteristics of MurmurHash are better than other hash functions, and thus have the characteristics of low collision rate. In addition, we adopt the idea of double hashing to simulate k independent hash functions to further reduce hash collisions. Use the MurmurHash algorithm to hash the input data to obtain a 128-bit hash value, obtain the lower 64 bits as the first hash value h_a and the upper 64 bits as the second hash value h_b, by $\mathrm{h}(x) = \mathrm{h}_a(x) + i \times \mathrm{h}_b(x), 0 \le i \le k-1$ (i is the step size), k hash functions can be simulated. After the k hash values are obtained modulo m, the

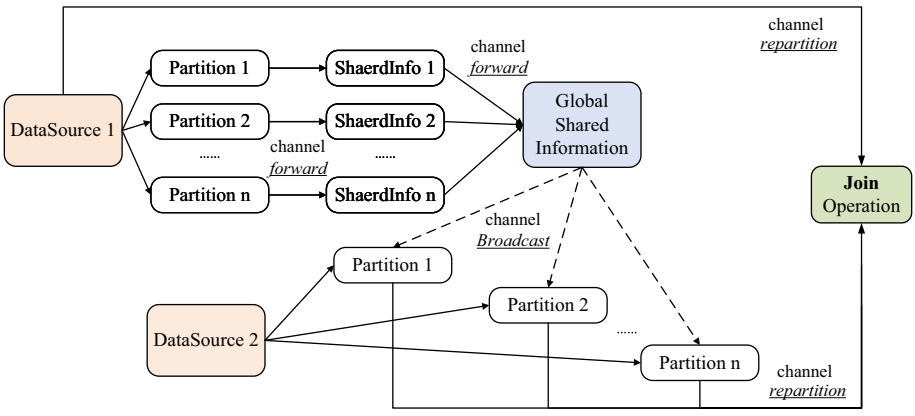


Fig. 4 Framework of data compression and filtering

corresponding bit of the bit array is set to 1. Perform the above operations on each piece of data separately, it can get a complete bit vector;

(2) **Merging of shared information:** Call the groupReduce method in Flink and set its parallelism to 1 to get the shared information in all partitions. Since the bit vectors of the BloomFilters generated by each partition have the same size and use the same number of hash functions, the global shared information can be obtained by splicing the shared information data in each partition;

(3) **Broadcast of shared information:** First, a broadcast channel is established, which connects the tail of the constructed global shared information and the header of another data source information, and then the compressed global shared information data will be output to the broadcast channel. At this time, the global shared information data will be received by each computing node, and finally obtained by different tasks on the node as a public broadcast variable.

After receiving the global shared information data, each partition needs to perform data selection on the data source 2, so as to retain only the data that meets the join conditions as much as possible. The filtering operation of data source 2 is implemented based on mapPartition method in Flink. Implement a user-defined function in mapPartition, which calculates each record in data source 2 to obtain k hash values $\{h_1, h_2, \cdots, h_k\}$ by using the k hash functions mentioned above. Perform modulo operation on the k hash values, and then check whether the values of the k positions corresponding to BloomFilter are all 1. If one of the k positions is not 1, The record is filtered. Finally, the qualified data is sent to each computing node according to the RR strategy, and the local hash join operation is performed.

For each data partition, the time complexity of BloomFilter construction and data filtering is O(nk), where k is the number of hash functions and n is the number of data records in each partition. The space complexity is O(m), where m is the length of the BloomFilter.

6 Evaluation results

This section uses different Flink jobs with multi-source data connection operation to experiment on different types of spatio-temporal data sets. The experiment first test the execution time of multi-source data connection jobs after optimization compared with the traditional sequential execution time, and then test the total amount of data communication between computing nodes, and finally show the independent execution time of the optimization algorithm. Each group of experiments is run more than 10 times, and the records were taken after removing the maximum and minimum values.

6.1 Experimental setup

We run experiments on a 7-nodes OMNISKY cluster (1 JobManager & 6 TaskManagers), and all nodes are connected with 10-Gigabit Ethernet. Each node has two Intel Xeon Silver 4210 CPUs @ 2.20GHz (10 cores × 2 threads), 128 GB memory, and 1 TB SSD. Hadoop version 2.7.0 (for storing data on HDFS) and Flink version 1.8.0 are chosen as

the experimental environment, and their configuration files are configured according to the hardware environment as mentioned above.

Experiments use TPC Benchmark [25] 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 spatio-temporal datasets for commodity retailing. The dataset defines different 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 spatio-temporal data sets for calculating various road traffic indicators in smart cities. According to the strict regulations on the amount of data in the TPC Benchmark and Ali Benchmark, the experiment takes 1 GB as the standard data total unit. In TPC Benchmark, a data unit contains 8 different data sources such as region, customer and orders etc., with sizes ranging from 1 KB to 700 MB, data record numbers ranging from 30 to 80,000, and column numbers ranging from 3 to 16. In Ali Benchmark, a data unit contains 7 different data sources such as road, signal and speed etc., with sizes ranging from 3 KB to 590 MB, data record numbers ranging from 50 to 64,000, and column numbers ranging from 5 to 23.

In terms of parameter settings, the network communication cost weight $w_{Com} = 100$, the IO cost weight $w_{IO} = 10$, the CPU cost weight $w_{CPU} = 0.01$, the CPU calculation factor of the hybrid hash join $factor_{cpu} = 4$, the pheromone heuristic factor $\alpha = 1$, the expectation heuristic factor $\beta = 3$, the pheromone volatility coefficient $\rho = 0.35$, and the number of ants is set to the number of Join Operators.

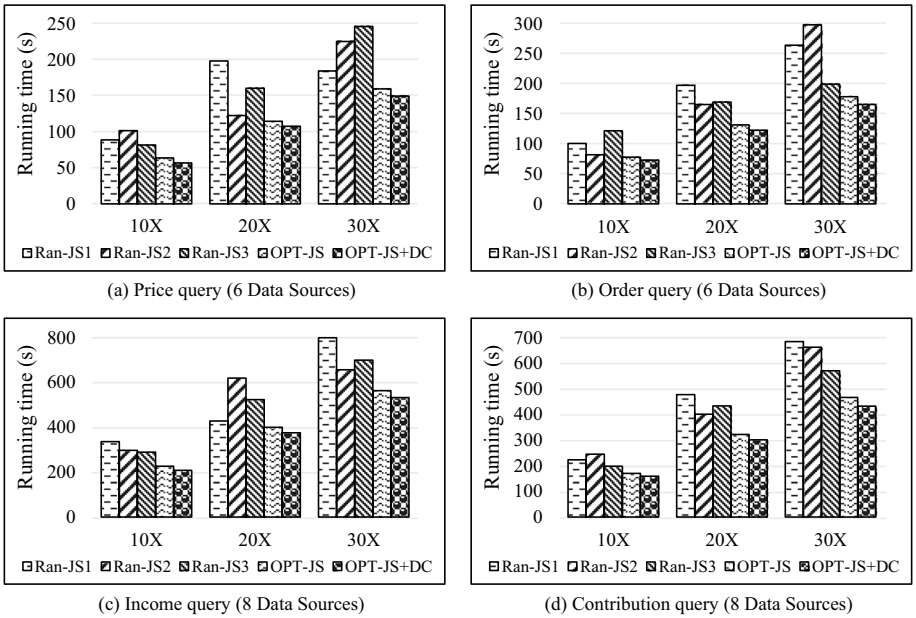
6.2 Test of job efficiency

In order to evaluate the direct effect of the two strategies proposed in this paper, the experiments first compare the execution time of multi-source data connection jobs. The experiments first randomly generates three different join sequences in the same Flink job according to the data connection conditions (Ran-JS1 to Ran-JS3), and then records their running time and compares them with the running time after adding the optimization strategy based on join sequence adjustment (OPT-JS) and further adding the optimization strategy based on repartition data compression (OPT-JS+DC).

Figure 5 shows the job running time of TPC Benchmark under different numbers of data sources, different queries, and different total data sizes. We expand standard data total unit to 10 times, 20 times and 30 times respectively to reach 10 GB, 20 GB and 30 GB, and then use 6 data sources and 8 data sources respectively to verify the effect of different queries. It can be seen that after using the optimization strategy based on join sequence adjustment, the job running time is significantly reduced compared with the three random join sequences, and the average can reach 27%. After adding optimization strategy based on repartition data compression, the execution time of the job has been further reduced, reaching an average of 33%.

Figure 6 depicts the comparison of job running time under Ali Benchmark for different numbers of data sources and different queries, where the number of data sources is 4 and 7 respectively, and the total data size is 10 GB, 20 GB and 30 GB respectively. It can be seen that the efficiency of multi-source data connection jobs after adding the optimization strategies is still significantly improved, and the job running time can be reduced by an average of 35% under the blessing of the two optimization strategies.

From the above two experimental results, it can be verified that different join sequences have a great impact on job efficiency, and can even more than double the



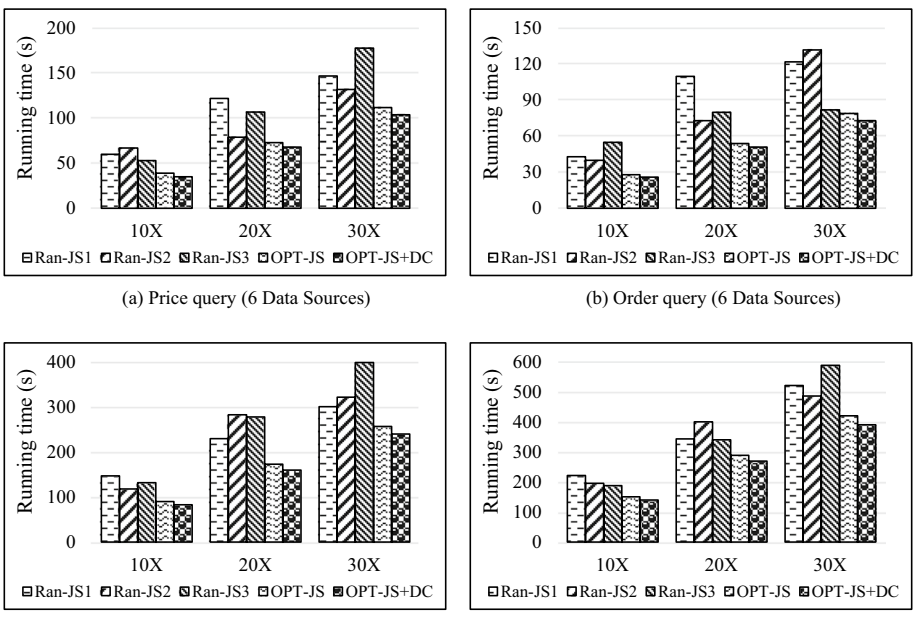
(a) Price query (6 Data Sources)

(b) Order query (6 Data Sources)

(c) Income query (8 Data Sources)

(d) Contribution query (8 Data Sources)

Fig. 5 Job running time comparison under TPC Benchmark



(a) Price query (6 Data Sources)

(b) Order query (6 Data Sources)

(c) Income query (8 Data Sources)

(d) Contribution query (8 Data Sources)

Fig. 6 Job running time comparison under Ali Benchmark

job running time. Due to the huge join sequences space, it is difficult for the randomly generated join sequences to achieve the optimal job efficiency. The two optimization strategies proposed in this paper can effectively reduce the total computational cost of the job (described in Section 4.2), thereby reducing the running time of the job.

6.3 Test of data communication

Next, the experimental results and analysis of data communication is carried out. As the specific calculation cost mentioned in Section 6.2 cannot be measured, the experiment on data communication can not only verify that the proposed optimization strategies can effectively reduce data communication pressure, but also verify the reduction of calculation cost brought by the proposed optimization strategies from the side. As in Section 6.2, the experiments also uses the randomly generated join sequences (Ran-JS1 to Ran-JS3) and adds the two proposed optimization strategies (OPT-JS, OPT-JS+DC) to compare the data communication.

In TPC Benchmark, the number of data sources is set to 5-8, and the total data volume is 5 GB, 10 GB and 15 GB, respectively, to verify the comparison of data communication under different numbers of data sources and different data sizes. Figure 7 shows the comparison results of data communication. It can be seen that after using the join sequence dynamic adjustment strategy, the total data communication between computing nodes has been significantly reduced compared to the three random join sequences, with an average of 32%. After using the communication data filtering optimization strategy, the data communication volume has been further significantly reduced, reaching an average of 42%.

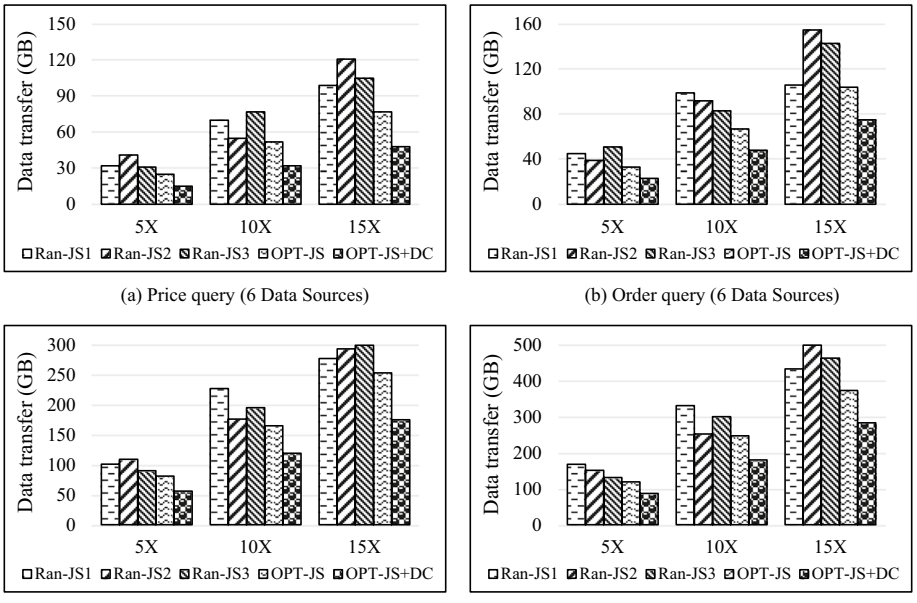


Fig. 7 Data communication volume comparison under TPC Benchmark

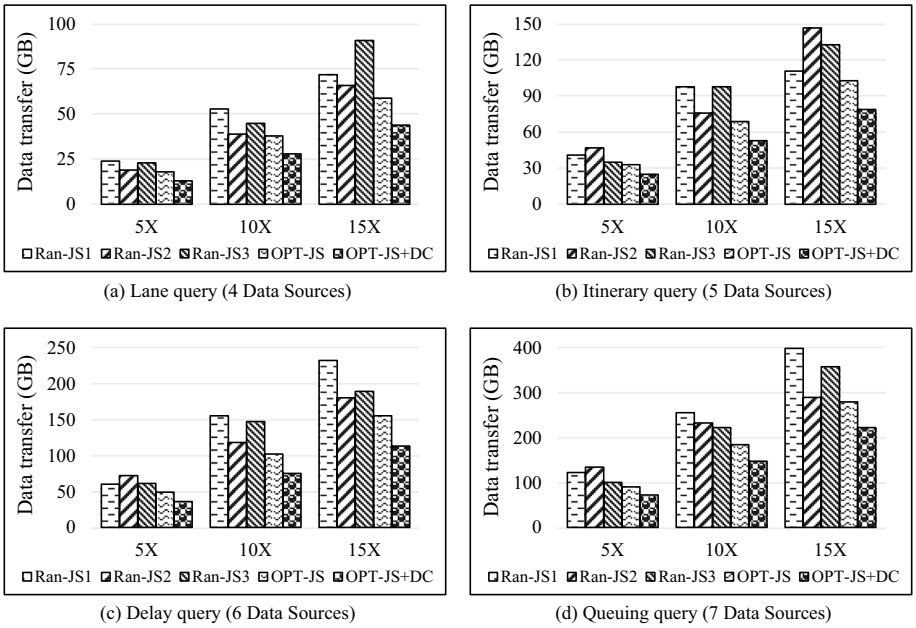


Fig. 8 Data communication volume comparison under Ali Benchmark

In Ali Benchmark, the number of data sources is set to 4-7, and the total data volume is 5 GB, 10 GB and 15 GB, respectively. In the experimental results depicted in Fig. 8, the two optimization strategies can achieve 34% and 43% reductions in data communication, respectively.

The reason for the above experimental results is that the cost estimation strategy and the dynamic adjustment strategy of the join sequence proposed in this paper are more inclined to execute the connection operations with a small amount of intermediate data result in advance, thereby reducing the total amount of data communication. Moreover, The size of the intermediate data result will affect the CPU calculation cost and IO cost, which in turn affects the job efficiency. In addition, the optimization strategy based on repartitioning data compression itself is an optimization strategy to reduce the data communication of intermediate results, so it can significantly reduce the data communication between computing nodes.

6.4 Test of optimize processing time

Finally, the experiment will verify the running time of the two optimization strategies algorithm itself. The total amount of data in both Benchmarks is set to 10 GB, where the number of data sources (NOD) for TPC Benchmark is set to 5-8, and the number of data sources for Ali Benchmark is set to 4-7. The query settings are the same as in Section 6.3.

Table 1 shows the processing time of join sequence adjustment strategy (JSA) and repartition data compression strategy (RDC) with different number of data sources. It can be seen that the execution time of join sequence adjustment strategy is extremely short even when dealing with large-scale data. Although the running

Table 1 Processing time of the optimization phase

NOD	TPC Benchmark			NOD	Ali Benchmark		
	Running Time				Running Time		
	JSA	RDC	Total		JSA	RDC	Total
5	0.52s	5.37s	5.89s	4	0.39s	3.86s	4.25s
6	0.69s	7.79s	8.48s	5	0.47s	5.02s	5.49s
7	0.93s	10.21s	11.14s	6	0.66s	7.58s	8.24s
8	1.38s	14.13s	15.51s	7	0.96s	10.88s	11.84s

time of repartition data compression strategy is longer, it can still bring about the improvement of job efficiency. The reason for the longer running time of repartition data compression strategy is that it scans the records in the data source, while join sequence adjustment strategy only uses the basic information of the data source. The goal of repartition data compression strategy is to use the constructed filter to filter out the transmission of data records that does not meet the association conditions between computing nodes before performing the connection operation. Therefore, even though the filter construction process and the data filtering process consume a certain amount of time, it is still less than the reduced data connection calculation time caused by the reduction of data communication.

7 Conclusion and discussion

In this paper, we analyze the importance of data connection in spatio-temporal data management, and the shortcomings of Flink in multi-source data connection operation, and then propose a new Operator that can support multi-source data connection operation, in which the optimization strategy mainly includes two aspects:

(1) We propose a join sequence adjustment strategy, which reduces the amount of calculation by introducing a computational cost estimation model and MMAS algorithm to adjust the join sequence. Experimental results prove that the proposed optimization strategy can improve the efficiency of jobs involving multi-source data connection operation;
(2) We propose a repartitioned data compression strategy, which filters repartition data by using BloomFilter to construct shared information and reduce data communication between computing nodes. Experimental results show that the proposed optimization strategy can further improve the efficiency of data connection operation.

Distributed systems still have many parts that need to be optimized and improved at the level of multi-source data connection. The following issues can be studied in the future:

(1) The Optimizer considers the size of the data source and the number of records as basic information, and the distribution of the data source is also a key factor affecting the efficiency of distributed data connection. In future research, the connection data can

be preprocessed according to the distribution of data sources, and then optimization strategies can be introduced;

(2) The proposed optimization strategy is only motivated by different join sequences resulting in different calculation amounts, but different join modes will also result in different calculation amounts in multi-source data connection operation. Therefore, collaborative optimization strategies for join sequence and join mode can be considered in the future.

Author contributions Conceptualization, Hangxu Ji; software, Hangxu Ji; methodology, Hangxu Ji and Yuhai Zhao; supervision, Hangxu Ji and Shiye Wang; validation, Hangxu Ji and Shiye Wang; writing-original draft, Hangxu Ji; writing-review and editing, Gang Wu, George Y. Yuan, and Guoren Wang.

Funding This research was supported by the National Key R&D Program of China under Grant No. 2018YFB1004402; and the NSFC under Grant No. 61872072, 62072087, 61772124, 61932004, 61732003, and 61729201; and the Fundamental Research Funds for the Central Universities under Grant No. N2016009.

Data availability All data in the experiment is authoritative and available.

Code availability All the codes in this research are available.

Declarations

Ethics approval This article does not contain any studies involving human participants and/or animals by any of the authors.

Consent to participate All authors have agreed to participate in the research described in this manuscript.

Consent for publication All authors have read and agreed to the published version of the manuscript.

Conflict of interest The authors declare no conflict of interest.

References

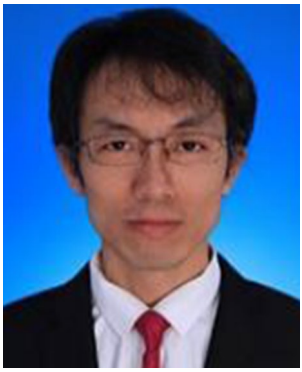
1. Isaksen ET, Johansen BG (2021) Congestion pricing, air pollution, and individual-level behavioral responses. Memorandum
2. Ye Y, Wang G, Chen L, Wang H (2015) Graph similarity search on large uncertain graph databases. Vldb Journal 24(2):271–296
3. Delianidi M, Salampasis M, Diamantaras K, Siomos, T, Karaveli I (2021) A graph-based method for session-based recommendations
4. Ye Y, Xiang L, Chen L, Sun Y, Wang G (2016) Rsknn: knn search on road networks by incorporating social influence. IEEE Transactions on Knowledge & Data Engineering 28(6):1575–1588
5. Yuan Y, Lian X, Wang G, Chen L, Ma Y, Wang Y (2019) Weight-constrained route planning over time-dependent graphs. 2019 IEEE 35th international conference on data engineering (ICDE)
6. Wang Y, Yuan Y, Wang H, Zhou X, Mu C, Wang G (2021) Constrained route planning over large multi-modal time-dependent networks. ICDE, 313–324
7. Carbone P, Katsifodimos A, Kth, Sweden S, Tzoumas K (2015) Apache flink : Stream and batch processing in a single engine
8. Failure H, Failure H, Access SD, Access SD, Sets LD, Sets LD, Model SC, Model SC, Computation M, Computation M (2007) The hadoop distributed file system: Architecture and design. Hadoop Project Website 11(11):1–10

9. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010) Spark: Cluster computing with working sets
10. Scheufele W, Moerkotte G, Seminargebaude A (1997) Constructing optimal bushy processing trees for join queries is np-hard (extended abstract)
11. Dittrich J, Quiané-Ruiz J, Jindal A, Kargin Y, Setty V, Schad J (2010) Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). Proc. VLDB Endow 3(1):518–529
12. Eltabakh MY, Tian Y, Özcan F, Gemulla R, Krettek A, McPherson J (2011) Cohadoop: Flexible data placement and its exploitation in hadoop. Proc. VLDB Endow 4(9):575–585
13. Kimmett B, Thomo A, Venkatesh S (2014) Three-way joins on mapreduce: An experimental study, 227–232
14. Afrati FN, Ullman JD (2011) Optimizing multiway joins in a map-reduce environment. IEEE Transactions on Knowledge and Data Engineering 23(9):1282–1298
15. Leis V, Radke B, Gubichev A, Mirchev A, Boncz PA, Kemper A, Neumann T (2018) Query optimization through the looking glass, and what we found running the join order benchmark. VLDB J 27(5):643–668
16. Li N, Liu Y, Dong Y, Gu J (2008) Application of ant colony optimization algorithm to multi-join query optimization 5370:189–197
17. Kadkhodaei H, Mahmoudi F (2011) A combination method for join ordering problem in relational databases using genetic algorithm and ant colony, 312–317
18. A LD, A GW, A JX, A XW, A SH, B RZ (2012) Commapreduce: An improvement of mapreduce with lightweight communication mechanisms. In: International conference on database systems for advanced applications, pp. 224–247
19. Michael L, Nejdl W, Papapetrou O, Siberski W (2007) Improving distributed join efficiency with extended bloom filter operations. In: 21st international conference on advanced information networking and applications (AINA 2007)
20. Selinger PG, Astrahan MM, Chamberlin DD, Lorie, RA, Price TG (1979) Access path selection in a relational database management system, 23–34
21. Vance B, Maier D (1996) Rapid bushy join-order optimization with cartesian products, 35–46
22. Ahmed R, Sen R, Poess M, Chakkappen S (2014) Of snowstorms and bushy trees. Proc. VLDB Endow 7(13):1452–1461
23. Blanas S, Li Y, Patel JM (2011) Design and evaluation of main memory hash join algorithms for multi-core cpus, 37–48
24. Stutzle T, Hoos H (1999) Improving the ant system: A detailed report on the max-min ant system
25. Barata M, Bernardino J, Furtado P (2015) An overview of decision support benchmarks: Tpc-ds. TPC-H and SSB 353:619–628

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

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.



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.

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.

ShiyeWang received her M.S. degree from the College of Computer Science and Engineering, Northeastern University, China, in 2020. Her research interests include distributed system and machine learning.

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.

Authors and Affiliations

Hangxu Ji[1] · Gang Wu[1] · Yuhai Zhao[1] · Shiye Wang[2] · Guoren Wang[2] · George Y. Yuan[3]

Yuhai Zhao
zhaoyuhai@mail.neu.edu.cn

Guoren Wang
wanggrbit@126.com

George Y. Yuan
yuanye@thinvent.com

[1] School of Computer Science and Engineering, Northeastern University, No. 3-11, Wenhua Road, Heping District, Shenyang 110819, China

[2] School of Computer Science and Technology, Beijing Institute of Technology, No. 5, South Street, Zhongguancun Street, Haidian District, Beijing 100081, China

[3] Thinvent Digital Technology Co., Ltd., No.681 Torch Avenue, High-Tech Development Zone, Nanchang 410000, China