Spark-Based Two-Table Equi-Join Process Optimization Postprint

Authors: Zhang Zidong, Zheng Yanbin

Date: 2018-05-20T00:00:00+00:00

Abstract

In statistical data analysis queries, equi-join between tables is one of the commonly used operations, but it is costly. In big data environments, the efficiency of equi-joins between large tables is even lower. To address this problem, an optimization method for the two-table equi-join process based on Spark is proposed. First, a Bloom Filter is constructed based on data value density characteristics to complete table filtering operations; second, combining the advantages of both Simi-Join and Partition Join, the filtered single-side table is split using a greedy algorithm; finally, the split subsets are joined, thereby converting the join process of two large tables into a phased join of two small tables. Cost analysis and experimental results demonstrate that compared with existing Spark-based join operations, this algorithm not only achieves performance improvement but also has less impact on algorithm efficiency when data skew occurs.

Full Text

Optimization of Two-Table Equi-Join Process Based on Spark

Zhang Zidong[1], Zheng Yanbin[2] [1]College of Computer Engineering, Jimei University, Xiamen, Fujian 361021, China [2]College of Computer & Information Technology, Henan Normal University, Xinxiang, Henan 453007, China

Abstract

Equi-join between tables is a commonly used but expensive operation in statistical data analysis queries, and its efficiency is even lower for large-scale tables in big data environments. To address this problem, this paper proposes an optimization method for two-table equi-join processes based on Spark. First, we construct a Bloom Filter based on the low value density characteristic of data to complete table filtering operations. Second, combining the advantages

of both Semi-Join and Partition Join, we split the filtered unilateral table using a greedy algorithm. Finally, we perform joins on the resulting subsets, thereby transforming the join process between two large tables into a staged join process between smaller table pairs. Cost analysis and experimental results demonstrate that compared with existing Spark-based join operations, the proposed algorithm not only achieves performance improvements but also exhibits minimal efficiency degradation when data skew occurs.

Keywords: Spark; equi-join; big data; optimization; splitting

0 Introduction

In the Spark computing framework, table equi-join operations are implemented as RDD actions. Spark SQL primarily employs three join strategies: Broadcast Join, Shuffle Hash Join, and Sort Merge Join. Broadcast Join broadcasts one table to all cluster nodes via broadcast variables, enabling each executor to perform the join locally. While this avoids shuffle operations, it consumes memory space on each node to store the broadcasted data. Consequently, Broadcast Join is only suitable for joins between a large table and a small table, or between small tables. Other join methods such as Shuffle Hash Join and Sort Merge Join repartition data based on join keys, inevitably involving shuffle operations. When data volumes are large, shuffle introduces substantial network communication and disk I/O overhead, and the data distribution is often unpredictable. Data skew can cause certain nodes to experience prolonged processing times and heavy computational loads, leading to overall job delays, out-of-memory errors, and resource waste.

Given that big data is characterized by low value density, the two tables involved in join operations typically contain numerous tuples that do not satisfy the join conditions. Common join algorithms in Spark/MapReduce programming include broadcast join (map-side join) and repartition join (reduce-side join), both of which have limitations. Xu et al. proposed Semi-Join for MapReduce, which extracts join keys from a small table R into a small file F that can be cached in memory. During the map phase, F is distributed to all TaskTrackers via DistributedCache, and tuples from the other table Q with keys not in F are filtered out before the reduce phase. Ramesh et al. proposed a Bloom Filter-based equi-join algorithm that reduces shuffle data volume through filtering. However, this approach is based on MapReduce, which stores intermediate results on disk, making it fundamentally slower than Spark' s in-memory computation. Moreover, it does not address data skew. Zhou et al. identified data skew as a consequence of Hadoop' s inability to perceive data distribution at the mapper output, causing reducer load imbalance. Gufler et al. proposed two methods to determine partitioning functions based on sampling results to ensure reducer load balancing. By analyzing intermediate results to understand data distribution, adaptive partitioning and data distribution mechanisms can be de-

signed to balance reducer loads. Bian et al. proposed a Spark-based equi-join optimization that first obtains the distinct join key set from the Fact table with record positions, joins it with the Dim table to generate JoinedUK, and finally assembles the result with the Fact table. However, this method only works for large-small table joins and assumes all tables can be cached in memory, which is often impractical.

To overcome these limitations, this paper proposes a Spark-based optimization algorithm for two-table equi-join operations. The algorithm first constructs a standard Bloom Filter for table filtering. Second, it combines the strengths of Semi-Join and Partition Join, using the partition information of keys from one table and data skew analysis of keys from the other table as heuristic guidance for a greedy strategy. Finally, it splits the filtered unilateral table using a greedy algorithm and performs joins on the resulting subsets, transforming the two-large-table join into staged joins between smaller table pairs. Cost analysis and experiments demonstrate performance improvements over existing Spark-based joins with minimal impact from data skew.

[Figure 1: see original paper] Algorithm Phase 1: Filtering Join Tables
[Figure 2: see original paper] Algorithm Phase 2: Sampling and Statistical Distribution Analysis
[Figure 3: see original paper] Algorithm Phase 3: Staged Pre-join and Result Assembly

---

1 Spark-Based Two-Table Equi-Join Optimization Method

During the equi-join of two large tables, we first optimize by filtering out numerous tuples that do not satisfy join conditions to reduce overall data volume. To address the limitations of Spark' s Broadcast Join, Shuffle Hash Join, and Semi-Join, we propose an optimization method for equi-joins in Spark, which consists of three phases as shown in Figures 1, 2, and 3. The symbols used in the figures are explained in Table 1.

1.1 Filtering Join Tables

Due to their small memory footprint and fast computation, we employ standard Bloom Filters for data filtering in this phase. We extract the join attributes (join keys) from both tables (RDD_a and RDD_b) to obtain two new RDDs: Joinkey_a and Joinkey_b. We then apply Spark' s distinct operation to these RDDs to produce two duplicate-free datasets, Joinkey_a and Joinkey_b. Using Spark' s BloomFilter library, we compress these duplicate-free key sets to generate two bit arrays, BFA and BFB, and compute their AND operation to produce the final filter bit array BF. We then apply BF to filter both input tables, eliminating tuples that cannot participate in the join, resulting in two filtered tables RDD_a_f and RDD_b_f.

1.2 Sampling and Statistical Distribution Analysis

We extract join keys from the filtered partitions RDD_a_f and RDD_b_f to obtain JoinKey_a_f and JoinKey_b_f. Using Spark' s sample operator, we sample these RDDs at a rate of 0.2 (based on the Pareto principle, as larger sampling rates would impose significant computational overhead) to obtain two samples for statistical analysis. This analysis identifies the keys with the largest data volumes and assesses data skew in both tables.

1.2.1 Sampling Analysis Let the smaller table be RDD_S. We sample its join keys at rate to obtain the sample set SampleKeysRs and analyze key skew within it. Based on key frequency distribution and overall distribution, we partition keys into two disjoint sets: high-frequency keys are placed in set Sstew, while relatively uniform or low-frequency keys are placed in set Sdis. Keys in RDD_S not appearing in SampleKeysRs are considered extremely low-frequency based on statistical probability and are also placed in Sdis. We use Spark' s ReservoirSample algorithm (reservoir sampling) for this operation, which efficiently selects k samples from a set S of n items—particularly when n is large or unknown and cannot fit entirely in memory—thereby reducing memory consumption.

1.2.2 Statistical Distribution Analysis We perform statistical analysis and comparison on the two sampled key sets JoinKey_a_s and JoinKey_b_s to examine key distribution and skew in both RDDs. Leveraging Spark Statistics library methods such as summary statistics, stratified statistics, and kernel density estimation, we obtain: (a) the proportion Rk of each individual key in an RDD; (b) keys with severe skew in each RDD; (c) natural ordering of keys by volume in the sample; and (d) the proportion Rs of identical keys across both RDDs.

1.3 Staged Pre-join and Result Assembly

For the smaller table RDD_S, we sample its keys to obtain SampleKeysRs. Based on key skew in SampleKeysRs and partition information from RDD_B_UK, we split RDD_S using a greedy algorithm that dynamically divides it into n subsets RDD_Si. Each time we split out a subset RDD_Si, we immediately join it with RDD_B. This split-and-join process repeats until all data in RDD_S is processed. Since each RDD_Si is much smaller than RDD_S, it requires minimal memory and computation. Below we detail the unilateral table splitting method, pre-join process, and result assembly.

1.3.1 Unilateral Table Splitting Method For the pre-joined result Pre-Joined, we must repartition based on partition numbers contained in the keys. Therefore, the uniform distribution and integrity of partition numbers in Pre-Joined affect partitioning parallelism, while data skew of all keys in each partition directly impacts node load balancing during assembly. Our greedy strategy

considers three factors: (a) partition numbers across all keys in subset RDD_Si must be uniformly distributed, with roughly equal partition counts per key, and the overall partition number integrity should approximate that of RDD_B; (b) the data scale of all keys on any partition should be similar to other partitions; and (c) the greedy strategy uses the partition lists corresponding to keys in RDD_B_UK and the skew sets Sstew and Sdis from RDD_S sampling as heuristic guidance.

The splitting process uses a greedy algorithm to dynamically divide RDD_S into n subsets RDD_Si, where 0 i < n. This involves n-1 operations, each splitting out RDD_Si from the current RDD_Sj, where 0 j < n. The relationship between RDD_Sj and RDD_Si satisfies: RDD_Sj = RDD_Si RDD_Sj+1, with i + j = n-1. Initially, RDD_Sj is set to RDD_S. After splitting out RDD_Si, we proceed to the join phase between RDD_B_UK and RDD_Si, which includes pre-join and result assembly. This process repeats until all keys in RDD_S are split out.

1.3.2 Pre-join We perform parallel joins between RDD_B_UK and each subset RDD_Si to obtain the pre-joined result PreJoined. Each element in PreJoined contains a key, the corresponding tuples from RDD_Si, and the list of partition numbers for that key in RDD_B. Since RDD_Si is much smaller than RDD_S and RDD_B_UK is also small, pre-join is fast and its results are distributed across compute nodes, reducing shuffle overhead.

1.3.3 Result Assembly We repartition PreJoined data based on partition numbers corresponding to keys, where partition numbers correspond one-to-one with those in RDD_B (the larger table). This ensures that the partitioned result sets and RDD_B are stored in identical partitions. We then use zipPartitions to quickly assemble RDD_B and PreJoined without transferring RDD_B data over the network. The greedy strategy selected earlier reduces data skew, balances node loads, and ensures uniform partition distribution, enabling full utilization of cluster parallelism and improving computation speed.

---

2 Cost Analysis

2.1 Network I/O Cost Analysis

In our algorithm, the duplicate elimination phase incurs no network overhead. Network I/O costs arise from broadcasting BloomFilters for filtering and from data repartitioning during the pre-join phase and result assembly, as shown in Equation (2):

$$NetCost = NetCost_{pj} + NetCost_{zp} + NetCost_{bf} \tag{2}$$

Although our algorithm splits the unilateral table, the total network I/O volume equals that of the unilateral table, with each split-and-join operation generating far less network pressure. $NetCost_{pj}$ represents the network I/O cost for pre-join data repartitioning, expressed in data volume as Equation (3):

$$NetCost_{pj} = \alpha \times (Size(RDD\_B\_UK) + Size(RDD\_S)) \tag{3}$$

Since the deduplicated RDD_B_UK contains fewer tuples than RDD_B and each tuple only includes a key and its partition number, $Size(RDD\_B\_UK) \ll Size(RDD\_B)$, i.e., $Size(RDD\_B\_UK) = \varepsilon \times Size(RDD\_B)$ where $0 < \varepsilon < 1$.

$NetCost_{zp}$ is the network I/O cost for result assembly. Because pre-joining effectively performs global deduplication on RDD_B_UK' s join attributes and eliminates non-joinable tuples, $NetCost_{zp}$ is typically much lower than $NetCost_{pj}$. $NetCost_{bf}$ is the network I/O cost for the compressed bit arrays used in BloomFilter filtering, which is also far smaller than $NetCost_{pj}$.

With N participating nodes, total communication is distributed across nodes, yielding the estimated network I/O cost in Equation (4):

$$NetCost \approx \frac{\alpha \times \varepsilon \times Size(RDD\_B) + \alpha \times Size(RDD\_S)}{N} \leq \frac{Size(RDD\_B) + Size(RDD\_S)}{N} \tag{4}$$

where $0 < \alpha, \varepsilon < 1$.

2.2 Memory Space Cost Analysis

The BloomFilter data structures are released after filtering and need not remain in memory. Throughout the splitting process, we cache the filtered keys with their partition numbers and the skew set Sstew, and the pre-join results also consume memory. Sstew contains skewed keys from RDD_S based on the sampling rate, so its memory footprint is much smaller than $Size(RDD\_S)$. Thus, the memory cost is given by Equation (5):

$$Cost_{mem} = 2\times Size(RDD\_B\_UK)+Size(RDD\_S)+Size(S_{stew}) \approx 2\times Size(RDD\_B\_UK)+Size(RDD\_S) \tag{5}$$

Since $Size(RDD\_B\_UK) = \alpha \times Size(RDD\_B)$, we have:

$$Cost_{mem} \approx 2\alpha \times Size(RDD\_B) + Size(RDD\_S) \tag{6}$$

2.3 Comparative Analysis

Since Spark Broadcast Join only applies when one table is small, we compare our algorithm against the widely-used Hash Join. Hash Join is a repartitioning join, so each node' s network communication volume is $(Size(RDD\_B) + Size(RDD\_S))/N$, requiring memory space of $Size(RDD\_B) + Size(RDD\_S)$. Table 2 compares the two algorithms.

Cost Comparison of Two Equi-Join Algorithms

Algorithm	Network I/O Cost	Memory Space Cost
Hash Join	$(Size(RDD\_B) + Size(RDD\_S))/N$	$Size(RDD\_B) + Size(RDD\_S)$
Our Algorithm	$\leq (Size(RDD\_B) + Size(RDD\_S))/N$	$2\alpha \times Size(RDD\_B) + Size(RDD\_S)$

3 Experiments

3.1 Experimental Environment

We used two datasets, each containing two tables of different sizes (the first with lower skew than the second). With a sampling rate of 40%, we performed joins using both our algorithm and Spark' s Hash Join for comparison. The join attributes were C_RID and P_RID. Dataset sizes are shown in Table 3.

Dataset Sizes

The software environment is listed in Table 4.

Software Environment

Software	Version
JDK	jdk1.8.0_92
Hadoop	Apache Hadoop 2.7
Spark	Apache Spark 2.1.0
OS	CentOS 7 x64

Experiments were conducted on a VMware virtual cluster with 6 nodes, each configured as shown in Table 5.

Hardware Environment

3.2 Experimental Results

Execution times (averaged over 3 runs) are shown in Figure 4. Our algorithm consistently outperforms Spark SQL' s Hash Join by more than 2×. Comparing the two experimental groups with similar table sizes but different skew levels, Spark Hash Join shows significant performance degradation—4.8 minutes slower for the more skewed second dataset—while our algorithm' s execution time only increased by 0.7 minutes between the first and second groups. This demonstrates that our algorithm not only executes faster than Spark Hash Join but also maintains stable performance under data skew.

[Figure 4: see original paper] Join Execution Time Comparison

---

4 Conclusion

As Spark becomes ubiquitous in data analysis, interactive big data analytics demands continue to grow, with join performance remaining a critical bottleneck. This paper analyzes existing equi-join algorithms for Spark/MapReduce and proposes an optimization algorithm that leverages the low value density characteristic of big data and addresses data skew through filtering, deduplication, and staged splitting of the join process. Cost analysis and experiments demonstrate performance improvements over existing algorithms and robustness to data skew.

References

[1] Spark [EB/OL]. http://spark.apache.org/.

[2] Yu J, Xiang H, Dai Q, et al. Spark Core and Advanced Applications [M]. Beijing: Mechanical Industry Press, 2016.

[3] Armbrust M, Xin R S, Lian C, et al. Spark SQL: relational data processing in Spark [C]//Proc of ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2015: 1383-1394.

[4] Blanas S, Patel J M, Ercegovac V, et al. A comparison of join algorithms for log processing in MapReduce [C]//Proc of ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2010: 975-986.

[5] Xu Y, Zhou X, Chen L, et al. Handling data skew in parallel joins in shared-nothing systems [C]//Proc of ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2008: 1043-1052.

[6] Ramesh S, Papapetrou O, Siberski W. Optimizing distributed joins with bloom filters [C]//Proc of International Conference Distributed Computing and Internet Technology. 2008: 145-156.

[7] Zhou J, Wang Q, Gao J, et al. An approach for load balancing in MapReduce via dynamic partitioning [J]. Journal of Computer Research & Development, 2016.

[8] Gufler B, Augsten N, Reiser A, et al. Load balancing in mapreduce based on scalable cardinality estimates [C]//Proc of IEEE International Conference on Data Engineering. Washington DC: IEEE Computer Society, 2012: 522-533.

[9] Gufler B, Augsten N, Reiser A, et al. Handling data skew in MapReduce [C]//Proc of International Conference on Cloud Computing and Services Science. 2011: 574-583.

[10] Bian H, Chen Y, Du X, et al. Optimization of equi-join on Spark [J]. Journal of East China Normal University: Natural Science, 2014, 2014(5): 263-270.

[11] Hacigumus H, Iyer B, Mehrotra S. Providing database as a service [C]//Proc of International Conference on Data Engineering. 2002: 29-38.

[12] Wang Z, Chen Q, Li Z, et al. A MapReduce data skew handling method based on incremental partitioning strategy [J]. Chinese Journal of Computers, 2016, (1): 19-35.

[13] Nie C, Jiang J. Optimization of configurable greedy algorithm for covering array generation [J]. Journal of Software, 2013(7): 1469-1483.

Note: Figure translations are in progress. See original paper for figures.