Bala-Join: Balancing the Workload in Distributed Hash Join

Wenlong Song
Hui li*
Jinxin Yang
Xidian University
Xi'an, China
songwl@stu.xidian.edu.cn
hli@xidian.edu.cn
yangjx@stu.xidian.edu.cn

Pinghui Wang
MOE KLINNS Lab, Xi'an Jiaotong University
Xi'an, China
phwang@mail.xjtu.edu.cn

Yingfan Liu
Jiangtao Cui
Xidian University
Xi'an, China
liuyingfan@xidian.edu.cn
cuijt@xidian.edu.cn

ABSTRACT

Distributed Hash Join (Dist-HJ) is fundamental to distributed databases. Under Dist-HJ, the build and probe table shall be first redistributed to multiple compute nodes, each of which then performs a local HASH JOIN afterward. However, the prevalent use of hash partitioning in the former phase often leads to uneven load distribution if data is skewed, resulting in inferior join performance. In this paper, we introduce *Bala-Join*, a comprehensive and adaptive solution to balance the load in Dist-HJ execution.

Our approach consists of a novel redistribution strategy, namely Balanced Partition and Partial Replication (BPPR) algorithm and a distributed online skewed value detector. BPPR achieves balanced redistribution of skewed data through a multi-cast mechanism to improve computational performance and reduce network overhead. The online detector is tailored to the BPPR algorithm and provides real-time skewed value information to BPPR in an imperceptible manner while minimizing additional overhead. The detector only accounts for 2%-6% extra cost compared to BPPR, which is superior to existing solutions. Our experimental study shows that Bala-Join, compared to the latest Dist-HJ solutions, exhibits much better throughput *i.e.,* by 25%-61%.

PVLDB Reference Format:
Wenlong Song, Hui li, Jinxin Yang, Pinghui Wang, Yingfan Liu, and Jiangtao Cui. Bala-Join: Balancing the Workload in Distributed Hash Join. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

1 INTRODUCTION

2 INTRODUCTION

Traditional relational database management systems (RDBMSs) compile a given SQL query and generate the corresponding query execution plan (QEP) via a query optimizer. QEP is a tree structure consisting of a set of physical operators showing how a given SQL query will be processed step by step. Among dozens of predefined operators, HASH JOIN is one of the fundamental operators that have been extensively studied and used [17, 22, 31]. Given two relations, R and S, HASH JOIN will be executed as follows: (1) Use the smaller relation, say R, to build a hash table in memory. (2) Scan each element in S, check the elements in the hash table that match S with respect to the join condition. At this point, R and S are called the **build table** and the **probe table**, respectively.

In the majority of Shared-Nothing [30] distributed RDBMS (*e.g.,* CockroachDB [32], TiDB [15], Spanner [6]), HASH JOIN must be executed in a distributed manner, *i.e.,* Distributed Hash Join (Dist-HJ). In these environments, both storage and computation significantly differ from single-node environment: 1) From the data *storage* perspective, each table tends to be distributed across multiple *data* nodes. 2) From a *computational* aspect, the distributed environment comprises multiple compute nodes, and thus the execution of Dist-HJ needs to be re-designed. For the convenience of description, we functionally abstract the nodes in the distributed environment into *data*, *compute*, and *response* nodes. *Data* nodes temporarily store parts of R and S, which could be either raw tables or intermediate ones. The concatenation of all the *data* nodes constitutes the complete R and S. *Compute* nodes are responsible for performing HASH JOIN computation over the records acquired from *data* nodes. *Response* nodes are not complimentary and are applicable only when there is no subsequent operator along the QEP. The overall process of the three types of nodes collaborating to perform Dist-HJ can be summarized in the following three steps:

- **Redistribution:** When the Dist-HJ process is initiated, the nodes where R and S (as well as the intermediate tables) reside have already been determined, so the data need to be redistributed to multiple *compute* nodes for HASH JOIN task.
- **Computation**: Each *compute* node performs local HASH JOIN computations over the records shares it receives.
- **Aggregation:** This step only applies when HASH JOIN is the final step in the QEP, results produced by each *compute* node need to be aggregated to the response node. Otherwise, *i.e.,* the QEP contains subsequent operations after HASH JOIN, the results produced by the *compute* nodes becomes intermediate tables, and these *compute* nodes subsequently transform into *data* nodes, serving as a data source to provide input for subsequent operations, *e.g.,* another HASH JOIN.

*Hui Li is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

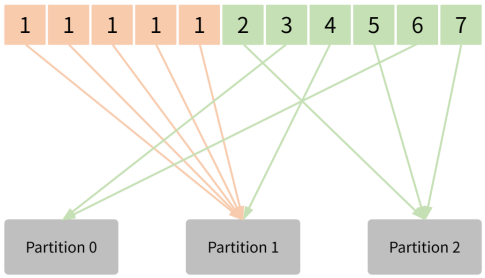


Figure 1: Hash partitioning under data skew

Notably, the redistribution phase is fundamental in a Dist-HJ solution. It is essential to consider both the task load balancing and the network overhead simultaneously, as both factors can affect the execution time of Dist-HJ. Shared-Nothing distributed database management systems (DDBMSs) often employ hash partitioning to redistribute data from different *data* nodes to *compute* nodes over the network based on the hash values derived from the join columns. When there is skewed data in the join columns, which is a universal phenomenon in practice [10, 27, 37], hash partitioning will lead to an excessive unbalance among nodes, degrading the performance of Dist-HJ significantly [2].

Figure 1 shows the process of hash partitioning following a hash function $hash(x) = x \mod 3$, values 1-7 are distributed among Partition 0, 1, and 2 according to the hash results. For simplicity, suppose a node is in charge of a partition. Obviously, value 1 is skewed, which is hash distributed to Partition 1. As a result, the node where Partition 1 resides will undertake much more computational task than the other nodes. A study by Barthels et al. [2] shows that in an 8-server cluster, the time cost for a Dist-HJ algorithm using hash partitioning is 3.3 times larger than in a skewed dataset (Zipf distribution, factor of 1.2), comparing with non-skewed data with the same volume. Prior studies [2, 24, 27] have extensively examined the challenges of the traditional redistribution with skewed data, identifying three main limitations.

- Hash partitioning of skewed data will result in uneven distribution of data, leading to certain hotspot nodes. Those nodes bear too much computational load, and this imbalance will make some other nodes idle and wait, resulting in the "long-tail effect";
- When a large amount of skewed data is sent to some of the hotspot nodes, it may lead to a network congestion towards these nodes;
- From the perspective of the whole cluster, when there is skewed data, the new nodes cannot disperse the burden on the hotspot nodes, thus limiting the horizontal scalability of the cluster and the space for performance improvement.

Therefore, addressing the skewed issue within Dist-HJ solutions is crucial for enhancing the performance of DDBMS clusters. This entails two main challenges: handling skewed data and detecting skewed values, the latter acts as a prerequisite for the former.

Handling skewed data aims to distribute the data as evenly as possible across *compute* nodes during the redistribution phase. The common idea [26, 27, 37] is to treat skewed data separately. While finer granularity can improve balance, it adds extra computational and network overhead. Current algorithms (*i.e.,* PnR [38] and PRPD [37]) typically focus on either balance or network overhead, thus limiting their overall performance.

In a distributed environment, detecting skewed values can be viewed as a global streaming frequent-item mining task, presenting real-time responsiveness and accuracy challenges. The common practice [3, 16] is to deploy detectors on each node, aggregating the results to form a consensus on skewed values. Both the deployment and aggregation phases introduce additional storage and network costs.

Generally, algorithms for handling skewed data and strategies for detecting skewness are optimized separately rather than deeply integrated, limiting the optimization potential. Furthermore, due to the diverse architectures of DDBMSs, the adaptability of solutions is another issue that need to address.

In this paper, we propose a comprehensive and adaptable technical solution, namely Bala-Join (Dist-HJ with near-balanced load and network overhead), to improve the performance of Dist-HJ in data-skewed environments. Bala-Join consists of a runtime skewed values detector, as well as a novel redistribution strategy, namely Balanced Partition and Partial Replication (BPPR). The former, can identify skewed values within intermediate join tables at runtime, while the latter is adaptable towards different fine-grained skew scenarios and consistently performs well. Specifically, our technical contribution in this work can be summarized as follows:

- We propose a novel algorithm for handling skewed data: the Balanced Partition and Partial Replication (BPPR) algorithm. The Balanced Partition mechanism of the algorithm balances overall throughput with network overhead, demonstrating high adaptability and stability. We also theoretically prove the correctness of BPPR and the balance of load it preserves.
- We present a distributed runtime skewed value detector that is tailored for BPPR. The strategy with Active-Signaling and Asynchronous-Pulling (ASAP) mechanism provides real-time skewed value information to BPPR imperceptibly while minimizing additional overhead costs.
- We conduct a series of experimental study to evaluate the performance of BPPR, the distributed detector, and the overall solution Bala-Join.

The rest of this paper is organized as follows. Section 3 reviews related works. In Section 4, we discuss the limitations of PRPD and PnR, and then propose the BPPR algorithm. We introduce the distributed skewness detector and the ASAP mechanism in Section 5. Section 6 covers the details of experimental setups and shows the evaluation results. Section 7 concludes the paper.

3 RELATED WORKS

3.1 Skewed data detection strategies

Skewed data detection strategies can be classified into two categories: static strategies and dynamic ones. Static detection typically takes place during the preparatory phase before a Join operation. Since static data inherently contains statistical information, it can be preprocessed by analyzing table or column statistics and sampling [10, 20, 26, 33]. Obviously, the static solution can not work when the HASH JOIN is not the first operator in the QEP.

Dynamic detection typically occurs at runtime, with uncertain data sources and scales. Skewed value information can only be obtained based on historical data that has already been received. This issue can be categorized under the problem of frequent itemset mining in data streams. Dynamic detection algorithms can be broadly classified into three categories: sampling-based algorithms, counter-based algorithms, and sketch-based algorithms.

Sampling-based algorithms [8, 12, 23] obtain data samples by randomly sampling or taking regular interval samples from the data stream.

Counter-based algorithms [23, 24] use counters or counting structures to track the occurrences of different items. Specifically, the Space Saving algorithm [24] use a collection of k counters to track the top-k most frequent elements. If an element is in the collection, its counter is incremented. If the collection is not full, the element is added with a counter initialized to 1. When the collection is full, the algorithm finds the element with the smallest counter, increments its counter, and replaces it with the new element.

Sketch-based algorithms [4, 7] use data sketches to estimate frequent items. Specifically, Count Sketch [4] employs a hash table and two matrices to estimate the occurrence frequency of elements in a data stream. Count-Min Sketch [7] uses a hash table composed of multiple hash functions to perform estimation.

There also exist several efforts in distributed dynamic detection. Parallel Spacing Saving [3] introduces a distributed parallel version of Space-Saving and proves its accuracy. LD-Sketch [16] is a sketch-based algorithm that maintains multiple frequent items and their counters within each bucket of the sketch. The distributed version of LD-Sketch reports elements that are frequent on a selected set of nodes as frequent items.

3.2 Algorithms for handling skewed data

Previous research [10] defined four types of data skew, namely Tuple Placement Skew (TPS), Selectivity Skew (SS), Redistribution Skew (RS), and Join Product Skew (JPS). In the majority of DDBMS, effective data distribution techniques are commonly employed, and database administrators typically select appropriate partitioning columns for data distribution. Consequently, TPS is exceptionally rare in practical scenarios. Even if it does occur, a well-performing hash function can be chosen to mitigate it. SS is induced by query predicates and is closely related to query relevance; research in this area is limited. RS and JPS are two prominent skew types, with research in this domain typically classified into three categories.

The first category of algorithms models the data skew as a traditional task scheduling problem. In the case of Dist-HJ, the join tables are divided into smaller partitions (*e.g.,* range partitions [10] or hash buckets [1, 9, 14, 20, 34–36]) and then distributed to all processing units for individual local computation. Since task scheduling problems are NP-complete, various well-known heuristic algorithms (*e.g.,* LPT [13] and MULTIFIT [5]) have been employed in such algorithms [1, 10, 14, 20, 34–36]. These algorithms propose different approaches based on distinct cost models. They estimate the Join cost using the number of tuples on each processing unit, aiming to balance the number of tuples from join tables on each processing unit to avoid Redistribution Skew (RS).

The second category of algorithms employs advanced models [1, 9, 10, 19, 29, 39, 40]. Instead of pre-estimating execution time like the first type, it adapts in real-time, adjusting workloads among processing units. If one unit is overloaded, its tasks shift to less busy units. [10] introduces the Virtual Processor Scheduling (VPS) for this, using $|R|+|S|+|R \bowtie S|$ to gauge Join costs. [1] employs output and work functions for cost estimation. [9] considers systems with uneven processing unit configurations. [29] relies on shared virtual memory to shift tasks away from busy units.[19] observes join tables processing speeds. [40] offers single-stage and two-stage scheduling, the latter involving a central processor managing tasks. Fastjoin [39] periodically reallocates workloads to address stream processing imbalances.

The third category of algorithms takes a different approach by focusing on micro-level adjustments to balance computational loads rather than handle data skew on a macro-scale. PRPD [37] assumes data to be randomly distributed across units. On each unit, skewed tuples in one join table remain local, while corresponding tuples from another table are broadcasted to all units, thereby achieving workload balance. Flow-Join [27] refines PRPD by applying Symmetric Fragment Replicate (SFR) for skewed values in both join tables, which performs well in large high-speed networks. [26] offers Track Join, an algorithm that reduces network traffic by optimizing data transmission plans for each unique join value. [11] introduces FGSD, a fine-grained method to improve load balancing in MapReduce flows, thus reducing task times. Inspired by prior work in [27, 37], the PnR algorithm is presented in [38]. Departing from the traditional focus on minimizing network data transfer, PnR targets near-perfect workload balancing across processing units and excels in high-speed network conditions.

4 BALANCED PARTITION AND PARTIAL REPLICATION

4.1 Limitation of PRPD and PnR

In this section, we will provide a detailed description of the existing solutions, including PRPD and PnR, and highlight their limitations.

Grace Hash Join: Grace Hash Join (GraHJ) [21] is a widely used implementation for Dist-HJ in Shared-Nothing architectures. For join tables R (build table) and S (probe table), GraHJ employs a hash function to partition both R and S into an equal number of partitions, ensuring that each partition of R can fit in memory for hash table construction. By using the same hash function for R and S, matching elements in R and S end up in the same partition, ensuring the correctness of the result. After partitioning, each partition contains a subset of both R and S. The partitions are independent, and each only needs to perform its own HASH JOIN calculation. The union of the results from all partitions constitutes the final output. Figure 2 illustrates the partitioning mechanism of GraHJ.

PRPD: While GraHJ performs well in most distributed scenarios, its efficiency can decline when data is skewed, increasing network data transmission. PRPD [37], optimized for skewed data, enhances performance without affecting regular data. During the redistribution phase, PRPD categorizes elements in table S as either non-skewed or skewed: (a) Non-skewed elements follow GraHJ's redistribution logic. (b) Skewed elements are kept locally. Corresponding elements in table R are then duplicated across all nodes.

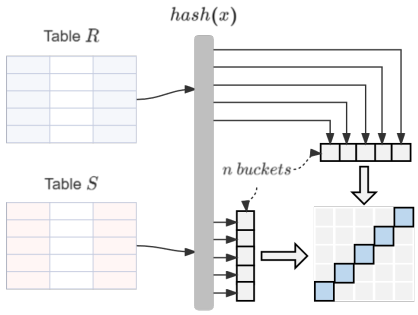


Figure 2: Partitioning of join tables in GraHJ.

This strategy balances workload and minimizes network transmission, especially when skewed elements are uniformly distributed across nodes. However, PRPD introduces an element of uncertainty due to the unpredictable distribution of skewed data, making it less stable than other load-balancing approaches.

Figure 3(a) illustrates the core idea of the PRPD algorithm. In the given example, *node* 0 and *node* 1 act as both *data* nodes and *compute* nodes. S contains 10 elements, among which the elements of value 1 is skewed. The skewed elements in S on both *node* 0 and *node* 1 are retained locally, while the skewed elements of value 1 from table R is replicated to both *node* 0 and *node* 1. The remaining elements in tables R and S are redistributed according to the strategy of GraHJ.

PnR: PnR selects to evenly redistribute skewed elements to achieve a workload balance across all *compute* nodes. As shown in Figure 3(b), all of the skewed elements of value 1 in table S are initially located on *node* 0. PnR divides these elements evenly, keeping one partition on *node* 0 while redistributing the other to *node* 1. In implementation, PnR achieves an even distribution by randomly sending skewed elements to *compute* nodes. The corresponding skewed elements of value 1 in R are also distributed across both nodes. The figure only shows the handling of skewed elements in S. The treatment of skewed elements in table R is similar to that of S.

PRPD focuses on optimizing network overhead, while PnR emphasizes load balancing. Both algorithms have their merits in specific scenarios but may face limitations in more generalized situations.

Figure 3(c) shows how PRPD performs under different data distributions. Here, the vertical bars indicates the count of specific values in the probe and build tables, showcasing three types of data distribution scenarios and their limitations. When the difference of the volume between the build table and the probe one is not too large (*e.g.,* with identical magnitude), it results in substantial network overhead.

(a) **Optimal Conditions:** In this scenario, elements in the probe table are evenly distributed across three *data* nodes. For illustration, we assume that elements in the build table are also balanced. After redistribution, the elements in both tables are balanced across all *compute* nodes.

(b) **Uneven Probe Table Distribution:** In this case, the data volume on *node* 2 significantly exceeds that on nodes 0 and 1. Following PRPD's redistribution strategy, *node* 2 will suffer from a much heavier load, while *node* 0 has a lighter one, causing a decline in the overall performance.

(c) **Large Build Table:** Here, elements in both the probe and build tables are evenly distributed across the three *data* nodes, but their volumes are nearly identical. Broadcasting the build table in this scenario results in substantial network overhead, leading to performance degradation.

These scenarios indicate that PRPD's strategy may not be sufficiently nuanced to provide optimal performance in every case.

Compared to PRPD, PnR has significant advantages in the computation phase. The average redistribution strategy ensures an absolute balance of the workload of each *compute* node and has no requirements for the distribution of the data itself, resulting in better stability. However, in the redistribution phase, PnR's strategy is theoretically more time-consuming than PRPD for the following reasons:

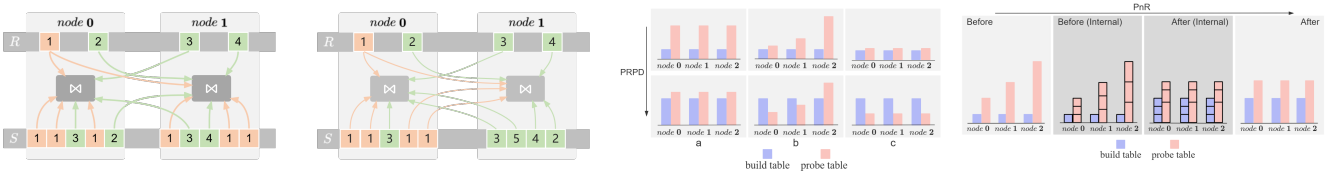
- **Massive Network Transmission:** The average redistribution strategy of PnR leads to the transmission of a large number of skewed elements into the network, which can cause large network overhead.
- **Uneven Redistribution Task Among Nodes:** Redistribution tasks involve data splitting and distribution, leading to uneven redistribution workloads among nodes. As shown in Figure 3(d), it is evident that *node* 2 faces a higher redistribution workload compared to *node* 0 and *node* 1. This disparity depends not only on the computational time for processing data but also on the network conditions: *node* 2 sends more data, which is more likely to cause network congestion than *node* 1.
- **Large Build Table:** In PnR, the handling of the build table involves broadcasting the elements corresponding to the skewed values of the probe table. When the sizes of the two tables are similar, the network data traffic caused by broadcasting the build table's data may even exceed that of the probe table.

The limitations stem from their different motivations. PRPD excels in poor network conditions by reducing network overhead, while PnR aims for optimal use of computational resources and performs best in good network environments.

Each of PRPD and PnR has weakness in either the computational or the redistribution phase. In light of that, we present a novel solution, Balanced Partition and Partial Replication (BPPR), in the following to achieve satisfactory performance in both computation and redistribution phases.

4.2 Algorithm details of BPPR

Before presenting the details of BPPR, we shall introduce some notions first. Assuming Dist-HJ is employed in a n-nodes cluster to accomplish: $R \underset{R.a=S.b}{\bowtie} S$, where R and S are two tables, either intermediate or raw. Each of R and S may be distributed across multiple *data* nodes. The data residing on node i is represented as R^i and S^i, which are subsets of R and S, respectively. The relationships between these subsets and their parent tables are governed by: $R = \bigcup_0^{n-1} R^i, S = \bigcup_0^{n-1} S^i$.



(a) Redistribution of PRPD (b) Redistribution of PnR (c) PRPD under different scenarios (d) PnR under different scenarios

Figure 3: Strategy and limitation of PRPD and PnR

Table 1: Notations

Symbol	Description
R, S	build table and probe table in Dist-HJ
R^i, S^i	partition of R, S on node i
$\widehat{R}, \widehat{S}$	skewed data in R, S
$\widetilde{R}, \widetilde{S}$	non-skewed data in R, S
R_b^i	data in R^i that intersects with $\widehat{S}$
R_h^i	data in R^i that does not intersect with $\widehat{S}$
$\widetilde{S^i}$	partition of $\widetilde{S}$ on node i
$\widehat{S^i}$	partition of $\widehat{S}$ on node i
$U(x)$ ($resp.$, $U^i(x)$)	set of nodes to which element x ($resp.$, on node i) can be distributed during the execution of BPPR
$U_{loc}(x)$	set of nodes where element x is kept local
$C(x, U(x))$	node selection function to choose a node from $U(x)$
$\mathcal{S}^i_{hash}$	data from $\widetilde{S}$ that node i receives through hash distribution
$\mathcal{S}^i_{bal}$	data from $\widehat{S}$ that node i receives through balanced distribution
$\mathcal{R}^i_{hash}$	data from $R_h^0, R_h^1, \ldots, R_h^{n-1}$ that node i receives through hash distribution
$\mathcal{R}^i_{bal}$	data from $R_b^0, R_b^1, \ldots, R_b^{n-1}$ that that node i receives through partial replication
$R(x)$	number of element x in R
$\widehat{S}^{ij}$	data distributed to node j from $\widehat{S^i}$

R and S may contain skewed tuples, represented as $\widehat{R}$ and $\widehat{S}$. For instance, $\widehat{R}$ is a set of frequently recurring skewed values in table R. Skewed values are identified by a frequency threshold f_C. If a value x in the $R.a$ column appears with a frequency at least as high as f_C, it is considered skewed and belongs to $\widehat{R}.a$. The function $f(x)$ denotes the frequency of x.

$$f(x \mid x \in \widehat{R}.a) \geq f_C, \quad f(x \mid x \in \widehat{S}.b) \geq f_C \tag{1}$$

Conversely, tables R and S also contain non-skewed data, donated as $\widetilde{R}$ and $\widetilde{S}$, respectively. Obviously, $R = \widetilde{R} \bigcup \widehat{R}, S = \widetilde{S} \bigcup \widehat{S}$

BPPR follows the general workflow of Dist-HJ, but with a different redistribution phase, containing an extra partitioning sub-step. The purpose of this step is to partition the subsets of tables S and R on each node into skewed and non-skewed parts, such that the skewed data can be processed correctly in the subsequent phases.

Step 1, Partition. In a distributed computing environment, it is typical for each *data* node to hold a subset of tables R and S, denoted as R^i and S^i for node i.

Partitioning S^i: S^i on each node is divided into two partitions, $\widetilde{S^i}$ and $\widehat{S^i}$.

Partitioning R^i: The partition of R^i at each node is passively determined by $\widehat{S^i}$. The details can be described as follows:

$$R_b^i = \{t \mid t.a \in \widehat{S}.b,\ t.a \in R^i.a\} \tag{2}$$

$$R_h^i = \{t \mid t.a \notin \widehat{S}.b,\ t.a \in R^i.a\} \tag{3}$$

Step 2, Redistribution. During the redistribution phase, R^i and S^i on each node are redistributed. After partitioning, R^i and S^i are divided into four partitions: $\widetilde{S^i}$, $\widehat{S^i}$, R_b^i and R_h^i. These partitions are redistributed as follows:

$\widetilde{S^i}$: This partition contains the non-skewed data from S on node i. The elements are hash-distributed, achieving nearly even distribution across n nodes.

$\widehat{S^i}$: This partition contains the skewed data from S on node i. The elements are distributed in a balanced manner, with each element of the skewed value targeted to a specific subset of nodes. The selection of these target nodes can be flexible, potentially encompassing some, or all n nodes, based on the application of a *sequence generator*, which is employed to generate the same sequences for identical values on each node used for updating the set of target nodes. It will be introduced in detail in Section 4.4. For any given skewed value x, the candidate set of target nodes is denoted as $U(x)$, with a choice function $C(x, U(x))$ determining the final destination. $U(x)$ and $C(x, U(x))$ satisfy the following relationship:

$$U(x) \in \{0, 1, 2, \ldots, n-1\},\ x \in \widehat{S} \tag{4}$$

$$C(x, U(x)) = i,\ i \in U(x) \tag{5}$$

R_b^i: According to Equation (2), all elements in R_b^i correspond with those in $\widehat{S}$. Since $\widehat{S^i}$ on each *data* node is balanced-redistributed, each element $x \in R_b^i$ must be sent to nodes specified by $U(x)$, which we refer to as *Partial Replication*.

R_h^i: Based on Equations (2) and 3, R_b^i and R_h^i are complementary sets. Specifically, R_b^i is the set related to S^i, while R_h^i is essentially the set unrelated to S^i. Elements in R_h^i undergo hash distribution.

Step 3, Computation. During the computation phase, each *compute* node receives data redistributed from the *data* nodes according to **Step 2**. Correspondingly, each *compute* node will receive four sets of data (shown in Figure 4):

$\mathcal{S}^i_{hash}$: This set is composed of partial data from $\widetilde{S^i}$ across n *data* nodes. During the redistribution phase, $\widetilde{S^i}$ on each node is hash-distributed to multiple *compute* nodes. $\mathcal{S}^i_{hash}$ thus consists of the data from $\widetilde{S^0}, \widetilde{S^1}, \ldots, \widetilde{S^{n-1}}$ that has been hash-distributed to node i. In fact, it is the subset of $\widetilde{S}$ that has been hash-distributed to node i. $\mathcal{S}^i_{hash}$ can be represented by the following formula:

$$\mathcal{S}^i_{hash} = \{t \mid (hash(t.b) \mod n) = i,\ t \in \bigcup_{0}^{n-1} \widetilde{S^i}\} \tag{6}$$

$\mathcal{S}^i_{bal}$: This set is derived from $\widehat{S^i}$ sent by n *data* nodes, where $\widehat{S^i}$ represents the skewed elements of S on node i. During the redistribution phase, element $t \in \widehat{S^i}$ is sent to the node $C(t.b, U(t.b))$. Specifically, for *compute* node i, $\mathcal{S}^i_{bal}$ contains element t when $C(t.b, U(t.b)) = i$. Therefore, $\mathcal{S}^i_{bal}$ is the data set of elements from $\widehat{S^0}, \widehat{S^1}, \ldots, \widehat{S^{n-1}}$ where $C(t.b, U(t.b)) = i$. $\mathcal{S}^i_{bal}$ can be represented by the following formula:

$$\mathcal{S}^i_{bal} = \{t \mid C(t.b, U(t.b)) = i,\ t \in \bigcup_{0}^{n-1} \widehat{S^i}\} \tag{7}$$

$\mathcal{R}^i_{hash}$: This set consists of partial data from R^i_h across n *data* nodes. R^i_h is the set that does not intersect with the skewed values of S and is hash-distributed during the redistribution phase. Therefore, $\mathcal{R}^i_{hash}$ is similar to $\mathcal{S}^i_{hash}$ and represents the data hashed to node i from $R^0_h, R^1_h, \ldots, R^{n-1}_h$. $\mathcal{R}^i_{hash}$ can be represented as:

$$\mathcal{R}^i_{hash} = \{t \mid (hash(t.a) \mod n) = i,\ t \in \bigcup_{0}^{n-1} R^i_h\} \tag{8}$$

$\mathcal{R}^i_{bal}$: This set includes elements from R^i_b. During redistribution, each R^i_b element is replicated to nodes in $U(t.a)$. $\mathcal{R}^i_{bal}$ may contain duplicate elements to ensure accurate HASH JOIN calculations. It consists of elements from $R^0_b, R^1_b, \ldots, R^{n-1}_b$ where $U(t.a)$ contains node i. It can be formally defined as follows:

$$\mathcal{R}^i_{bal} = \{t \mid i \in U(t.a),\ t \in \bigcup_{0}^{n-1} R^i_b\} \tag{9}$$

Upon receiving the above four data sets, each *compute* node must first perform local HASH JOIN as follows:

$$\mathcal{R}^i_{bal} \underset{a=b}{\bowtie} \mathcal{S}^i_{bal} \tag{10}$$

$$\mathcal{R}^i_{hash} \underset{a=b}{\bowtie} \mathcal{S}^i_{hash} \tag{11}$$

The local result for *compute* node i can be represented by the following equation:

$$R^i \underset{a=b}{\bowtie} S^i = (\mathcal{R}^i_{bal} \underset{a=b}{\bowtie} \mathcal{S}^i_{bal}) \bigcup (\mathcal{R}^i_{hash} \underset{a=b}{\bowtie} \mathcal{S}^i_{hash}) \tag{12}$$

Step 4, Union. After the computation phase, each *compute* node obtains local results. To aggregate these into a final result, a coordinating node is required to integrate the outputs from all *compute* nodes. The mathematical representation of this aggregation and integration process is as follows:

$$R \underset{R.a=S.b}{\bowtie} S = \bigcup_{0}^{n-1} (R^i \underset{R.a=S.b}{\bowtie} S^i) \tag{13}$$

Unlike PRPD and PnR, BPPR handles data skew from a macro perspective. While it does not optimize individual skewed values at the micro-level, the overall performance improves. In the case of PnR, for example, it ensures elements from the probe table S are evenly distributed across all nodes. However, in BPPR, these elements are distributed only to the node set $U(x)$ and might not be evenly distributed. Although the distribution of single-valued elements under BPPR is uneven, the overall distribution is balanced, which shall be justified in our empirical study in Section 6.

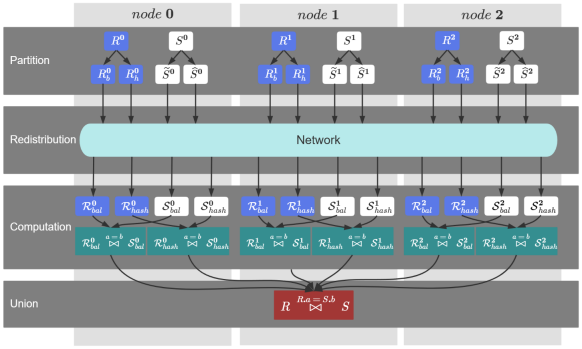


Figure 4: 3-steps illustration of BPPR (3 Nodes)

4.3 Correctness study of BPPR

Section 4.2 outlined the execution process of BPPR, including partitioning and computation. This section delves into the correctness of BPPR. As discussed in Section 4.1, for elements with the same values in the build and probe table, we can observe different combinations of distribution strategies, referred to as *pairing strategy*. Figure 3(a) shows two such pairing strategies: 1) Elements of non-skewed values 2, 3, 4 in the probe table are hash-distributed, and the corresponding elements in the build table are also hash-distributed. 2) Elements of skewed value 1 in the probe table are kept local, while the corresponding elements in the build table are broadcast to all nodes.

In summary, there are four distribution strategies (*i.e., hash*, *broadcast*, *keep local*, *random*) and three types of pairing strategies (*i.e., hash-hash*, *local-broadcast*, and *random-broadcast*) in PRPD and PnR. From the perspective of each of these strategies, we can infer:

Theorem 4.1. *In the computation of HASH JOIN, suppose the actual result is denoted by Z. When skewed values x in the probe table are distributed across multiple nodes while values x in the build table are replicated to all nodes, then the result Z' from the operation $R \bowtie_{R.a=S.b} S$ matches Z.*

This theory summarizes the *local-broadcast* and *random-broadcast* pairing strategies. The difference lies in the final distribution of elements from the probe table. Next, we will study the correctness of the three pairing strategies in sequence. Let $R(x)$ (*resp.*, $S(x)$) denote the count of tuples corresponding to x in R (*resp.*, S), after performing a HASH JOIN, the number of tuples with respect to x in the correct result should be $S(x) \times R(x)$.

(1) *hash-hash*: HASH JOIN is essentially an intersection operation. Only when a value x appears in both the probe and build table will there be a calculation result related to x. In the *hash-hash* strategy, no matter where the value x is distributed among the *data* nodes, using the same hash function will always send it to the same *compute* node. Therefore, this node will have $S(x)$ elements from the probe table and $R(x)$ elements from the build table. After a local HASH JOIN, the result count is $S(x) \times R(x)$, ensuring the correctness of this strategy for handling single value results.

(2) *local-broadcast*: In the *hash-hash* strategy, computation result generation for any value is always on a single node. However, with the *local-broadcast* strategy, these calculations spread across multiple nodes. Suppose value x from

the probe table distributes in set $U_{loc}(x)$, where $U_{loc}(x) \in \{0, 1, \ldots, n-1\}$, then the nodes involved are those in $U_{loc}(x)$. While the elements corresponding to x in S are kept local, the count of such elements at node i is $S^i(x)$, $i \in U_{loc}(x)$, satisfying the following:

$$S(x) = \sum_{i=0}^{n-1} S^i(x),\ i \in U_{loc}(x) \tag{14}$$

Elements corresponding to x in R are broadcasted to all nodes, therefore, the count of such elements at node i is $R^i(x)$, which can be represented as:

$$R^i(x) = R(x),\ i \in \{0, 1, \ldots, n-1\} \tag{15}$$

Then the count of results generated on node i is $S^i(x) \cdot R^i(x)$ and the final result count is:

$$\sum_{i=0}^{n-1} S^i(x) \cdot R^i(x),\ i \in U_{loc}(x) \tag{16}$$

By merging Equations (14), 15, and 16, we derive:

$$\sum_{i=0}^{n-1} S^i(x) \cdot R^i(x) = \sum_{i=0}^{n-1} S^i(x) \cdot R(x) = S(x) \cdot R(x) \tag{17}$$

At this point, the correctness of single-value pairing under the *local-broadcast* strategy has been proven.

(3) *random-broadcast*: The *random-broadcast* strategy is similar to the *local-broadcast* strategy. The difference is that calculations and result generation for a single value under the *random-broadcast* strategy are performed on all *compute* nodes. Regardless of how x in the probe table S distributes across *data* nodes originally, each *compute* node ends up with the same count of x elements. The number of x elements that node i obtains from S is denoted as $S^i(x)$.

$$S^i(x) = \frac{S(x)}{n} \tag{18}$$

Elements of value x in R are broadcasted to all nodes. The count of such elements on node i is $R^i(x)$, with the same representation as Equation (15). The final count of results is:

$$\sum_{i=0}^{n-1} S^i(x) \cdot R^i(x) \tag{19}$$

The above equation, combined with Equations (14) and 18, can be simplified to:

$$\sum_{i=0}^{n-1} S^i(x) \cdot R^i(x) = \sum_{i=0}^{n-1} \frac{S(x)}{n} \cdot R(x) = S(x) \cdot R(x) \tag{20}$$

This equation proves the correctness of single-value pairing under the *random-broadcast* strategy.

Although Theorem 4.1 guarantees the correctness of BPPR, it may contain redundant computation, especially in the *local-broadcast* pairing strategy. Specifically, when a value x in the probe table distributes across the node set $U_{loc}(x)$, broadcasting its corresponding elements from the build table to all nodes is excessive. In fact, broadcasting elements of the build table beyond $U_{loc}(x)$ does not affect the validity equation. Therefore, we can refine Therom 4.1 into the following:

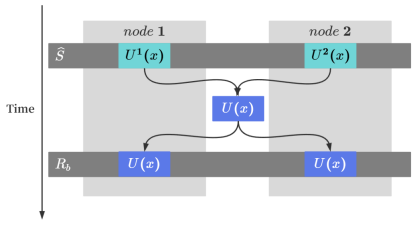


Figure 5: Consensus process for forming the node set $U(x)$ (2 nodes)

Theorem 4.2. *In a HASH JOIN operation, suppose the actual result is denoted by Z. If skewed values x from the probe table are distributed to the node set $U(x)$, and values x from the build table are replicated to the same node set $U(x)$, then the result Z' from the operation $R \bowtie_{R.a=S.b} S$ is equivalent to Z.*

During the redistribution phase of BPPR, there are two pairing strategies, *i.e.,* the *hash-hash* pairing strategy of $\widetilde{S^i}$ and R_h^i, and BPPR's pairing strategy of $\widehat{S^i}$ and R_b^i. Their computation phases correspond to Equations (10) and 11, respectively. The correctness of the results for the *hash-hash* pairing strategy has been proven by Theorem 4.1, and that for the BPPR pairing strategy follows Theorem 4.2. In all, the correctness of the BPPR algorithm is proven.

4.4 Sequence generator

During the redistribution phase of BPPR, $\widehat{S^i}$ on each *data* node is distributed in a balanced manner. It is mentioned that the set of nodes to which $\widehat{S^i}$ is sent is not fixed, and depends on the usage of the sequence generator. In the BPPR algorithm, the design purpose of the sequence generator is to hope that a consensus can be formed imperceptibly among *data* nodes.

Without a sequence generator, BPPR pairing strategy requires two synchronization steps. As shown in Figure 5, each *data* node generates a node set $U^i(x)$ when processing partition data $\widehat{S^i}$. Since $U^i(x)$ of each *data* node is different and does not satisfy Theorem 4.2, this leads to the first synchronization process.

$$U(x) = \bigcup_{0}^{n-1} U^i(x) \tag{21}$$

Afterwards, $U(x)$ has to be synchronized with the R_b^i partition of the data nodes, informing them to which nodes the data should be distributed. This means that R_b^i on the *data* node i must wait to generate $U^i(x)$. In BPPR, $U^i(x)$ continuously grows until $\widehat{S^i}$ is fully processed.

The sequence generator is designed to achieve the first synchronization. It generates a fixed sequence for each value, with both the sequence elements and length being related to the number of nodes n. The expansion of $U^i(x)$ follows this sequence from start to finish.

Algorithm 1 illustrates the generation of the sequence. Lines 5-12 detail the sequence generation process, with line 6 generating a temporary string to calculate the hash value. Line 7 generates a sequence element *number* by performing a hash modulus operation on the temporary string. Lines 8-9 conditionally add the element to the sequence. After iterations, the final sequence is returned.

Algorithm 1 Sequence Generation Algorithm

Input: value to generate the sequence$\boldsymbol{x}$, number of nodes $\boldsymbol{n}$
Output: ***sequence***

```
1:  procedure GenerateSequence(x, n)
2:      sequence ← []
3:      sequence.add(hash(x) mod n)
4:      epoch ← 0
5:      while sequence.size < n do
6:          hashString ← String(x) + String(epoch)
7:          number ← Hash(hashString) mod n
8:          if number ∉ sequence then
9:              sequence.add(number)
10:         end if
11:         epoch++
12:     end while
13:     return sequence
14: end procedure
```

Algorithm 2 Node Set Update Algorithm

Input: value to update the node set $\boldsymbol{x}$, node set to be updated $\boldsymbol{U(x)}$, generated sequence ***sequence***

```
1: procedure UpdateU(x, U(x), sequence)
2:     size ← U(x).size
3:     if size == sequence.size then
4:         return U(x)
5:     end if
6:     U(x).add(sequence[size])
7:     return
8: end procedure
```

Algorithm 2 details the process for updating the node set $U(x)$. It adds an element to the node set $U(x)$. Lines 3-5 determine whether the size of the node set $U(x)$ is equal to the *sequence*. If equal, no more elements are added. Line 6 adds an element from the sequence to the node set. Line 7 returns the updated node set $U(x)$.

With the sequence generator, every *data* node can use Algorithm 1 to generate a sequence for value x. Even without any communication between nodes, the generated sequences are identical. Although the sequence generation algorithm has linear time complexity, the sequences can be created once and reused, hence reducing the overhead. Once every node imperceptibly obtains the same sequence for x, we can use it as the node set $U(x)$ to achieve the first synchronization. However, this method is not flexible enough. Subsequent sections introduce the balanced partition algorithm. In combination with the sequence generator, it exhibits better performance.

4.5 Balanced partition algorithm

The primary task of balanced partitioning is to partition the probe table S. Specifically, at the *data* node level, balanced partitioning is a sub-task under the BPPR pairing strategy, focusing on the $\widehat{S^i}$ partition of S^i. At the *data* node, $\widehat{S^i}$ will undergo further partitioning (balanced repartitioning), and each sub-partition after this will be sent to a *compute* node. Here, the sub-partitions are denoted as $\widehat{S^{ij}}$, $j \in \{0, 1, \ldots, n-1\}$, indicating that this sub-partition data originates from $\widehat{S^i}$ and will be sent to the j^{th} *compute* node. To ensure that the data in $\widehat{S^i}$ is balanced at the macro level after repartitioning, a balance factor B^i needs to be defined:

$$B^i = (\max_j \widehat{S^{ij}} - \min_j \widehat{S^{ij}}) / \max_j \widehat{S^{ij}} \tag{22}$$

The balance factor reflects the degree of data balance among the sub-partitions. The smaller the value, the more balanced the sub-partitions are. When repartitioning $\widehat{S^i}$ for balance, a threshold B_c can be set to control the balance factor during the repartitioning process, that is, $B^i \leq B_c$.

After all *data* nodes complete the balanced repartitioning of $\widehat{S^i}$ and send the sub-partitions, the data volume received by *compute* node j is $\widehat{S^j}$:

$$\widehat{S^j} = \sum_{i=0}^{n-1} \widehat{S^{ij}} \tag{23}$$

Ultimately, the balance factor B of $\widehat{S}$ among the *compute* nodes can be calculated as:

$$B = (\max_j \widehat{S^j} - \min_j \widehat{S^j}) / \max_j \widehat{S^j} \tag{24}$$

Theorem 4.3. *In the case of balanced partitioning, if the balance factor B^i among sub-partitions for each data node is below the threshold B_c, it follows that the overall balance factor B among compute nodes after redistribution will also be below B_c.*

Proof. This statement can be proven through the transformation of inequalities. Transform Equation (22) into the following inequality:

$$\max_j \widehat{S^{ij}} - \min_j \widehat{S^{ij}} \leq B_c \cdot \max_j \widehat{S^{ij}} \tag{25}$$

Apply summation transformation to the inequality:

$$\sum_{i=0}^{n-1} (\max_j \widehat{S^{ij}} - \min_j \widehat{S^{ij}}) \leq \sum_{i=0}^{n-1} (B_c \cdot \max_j \widehat{S^{ij}}) \tag{26}$$

Shifting the summation gives:

$$\max_j (\sum_{i=0}^{n-1} \widehat{S^{ij}}) - \min_j (\sum_{i=0}^{n-1} \widehat{S^{ij}}) \leq B_c \cdot \max_j (\sum_{i=0}^{n-1} \widehat{S^{ij}}) \tag{27}$$

Divide both sides by $\max_j (\sum_{i=0}^{n-1} \widehat{S^{ij}})$:

$$\frac{\max_j (\sum_{i=0}^{n-1} \widehat{S^{ij}}) - \min_j (\sum_{i=0}^{n-1} \widehat{S^{ij}})}{\max_j (\sum_{i=0}^{n-1} \widehat{S^{ij}})} \leq B_c \tag{28}$$

Substitute into the inequality using Equation (23) to obtain:

$$\frac{\max_j \widehat{S^j} - \min_j \widehat{S^j}}{\max_j \widehat{S^j}} \leq B_c \tag{29}$$

By combining this with Equation (24), we obtain:

$$B \leq B_c \tag{30}$$

□

Algorithm 3 shows the detailed process of the balanced partitioning algorithm. According to Theorem 4.3, we only need to ensure that the balance factor B^i of each *data* node after partitioning is below the threshold B_c. Upon inputting the partition data $\widehat{S^i}$, the algorithm will ultimately return a set of sub-partitions, denoted as *partition*, that satisfy the condition $B^i \leq B_c$.

Lines 1-15 calculate the balance factor based on Equation (22). The calculation of the balance factor has linear time complexity. Nevertheless, in the actual implementation, this computation can be optimized to constant time complexity. As shown in line 23, each sub-partition only adds one element at a time. Thus, by maintaining the $maxVal$, $minVal$, and the element count for each sub-partition, we can simply update the $maxVal$ and $minVal$ when a new element is added to a sub-partition.

Lines 16-19 initialize the partition set as n empty sub-partitions. Lines 20-35 iterate over each element in $\widehat{S^i}$ and balance the partition. Lines 23-25 perform a test partition by inserting element t into the previously selected sub-partition, calculating the resulting balance factor, and then removing it. The balance factor later serves as the basis for strategic decisions.

Lines 27-33 aim to re-select a sub-partition ID when $balanceFactor > B_c$. Line 28 picks a sub-partition from the set $U(t)$, choosing the one with the fewest elements. If the sub-partition chosen in Line 28 is the same as the last, it would still exceed the balance factor threshold. In this case, Lines 30-31 update $U(t)$ and then re-select a sub-partition. Line 34 identifies the ultimately chosen sub-partition ID, which satisfies the balance factor threshold, and adds element t to it. Finally, Line 36 returns the sub-partition set *partition*.

Algorithm 3 Balanced Partition Algorithm

Input: partition of the current node $\widehat{S^i}$, thresholds for the balance factor B_c, number of nodes n
Output: set of the sub partitions after the balanced partitioning *partition*

```
1:  procedure CaclulateBalanceFactor(partition)
2:      maxVal ← partition[0].size
3:      minVal ← partition[0].size
4:      for array in partition do
5:          size ← array.size
6:          if size > maxVal then
7:              maxVal ← size
8:          end if
9:          if size < minVal then
10:             minVal ← size
11:         end if
12:     end for
13:     balanceFactor ← (maxVal − minVal) / maxVal
14:     return balanceFactor
15: end procedure
16: partition ← []
17: for i = 0 to n − 1 do
18:     partition.add([])
19: end for
20: for t in Ŝ^i do
21:     U ←set of partition IDs corresponding to t i.e., U(t)
22:     lastID ←partition ID last selected by t
23:     partition[lastID].add(t)
24:     balanceFactor ← CaclulateBalanceFactor(partition)
25:     partition[lastID].delete(t)
26:     selectID ← lastID
27:     if balanceFactor > B_c then
28:         selectID ←ID of the sub-partition in set U with the
    fewest elements
29:         if selectID == lastID then
30:             Expanding the ID set U using Algorithm 2.
31:             selectID ←ID of the sub-partition in set U with the
    fewest elements
32:         end if
33:     end if
34:     partition[selectID].add(t)
35: end for
36: return partition
```

5 DETECTION OF SKEWED ELEMENTS

Recall that our ultimate goal is to propose a universal Dist-HJ solution for skewed data scenarios, while the effectiveness of the BPPR algorithm depends on skewness information. That is, there still exist a gap between BPPR and a universal Dist-HJ solution, namely the detection of skewed values at runtime.

In this section, we will further introduce a dynamic distributed strategy for detecting skewed values. This strategy is based on the Space-Saving Algorithm [24] and is inherently integrated with BPPR to maximize its performance.

5.1 Detector in single-node environment

In distributed databases, as queries and computations involve multiple nodes coordinating, the input data for each operation is typically in the form of data streams. Therefore, we start with a skewness detector from data streams in a single-node environment, and then expand it to distributed settings. As discussed in Section 3, the existing skewness detecting strategies can be classified into: counter-based techniques and sketch-based techniques.

The Space-Saving Algorithm relies on a counter-based technique, serving as the building block in the distributed detection strategy of this paper. The core idea of Space-Saving is to maintain frequency information about a subset of the elements in data stream. Specifically, it preserves the frequencies of k elements using k counters. Each counter is updated to estimate the frequency of the corresponding element. Lightweight data structures are employed to sort these elements based on their estimated frequencies.

Algorithm 4 outlines the process of the Space-Saving Algorithm. Notably, in line 14, when the *monitorList* is at its capacity, and a new element e is to replace e_{min}, the *counter* retains the frequency information of the former element e_{min}. This counter is directly assigned to element e and added to the *monitorList*.

The primary challenge of the Space-Saving Algorithm lies in efficiently updating elements and removing elements corresponding to the minimum count, as indicated in lines 4-5 and 11-14 of Algorithm 4. Following the suggestion by [27], we adopt the Space-Saving Algorithm based on a hash table and an ordered array as the detector for single-node environment.

Algorithm 4 Space Saving Algorithm

Input: number of monitored elements $\boldsymbol{k}$, data stream ***Stream***
Output monitor list *monitorList*

```
1: monitorList ← []
2: for element e in Stream do
3:     if e in monitorList then
4:         counter ← counter corresponding to e
5:         counter + +
6:     else
7:         if monitorList.size < k then
8:             counter ← 1
9:             monitorList add e and counter
10:        else
11:            e_min ← element with the lowest frequency among the monitored elements
12:            counter ←counter corresponding to e_min
13:            monitorList remove e_min and its corresponding counter
14:            monitorList add e and counter
15:        end if
16:    end if
17: end for
18: return monitorList
```

5.2 Integration of BPPR and distributed detectors

We are now moving forward to integrate the detector into a distributed environment. A straightforward approach is to conduct local detection on each standalone node, aggregate the results, and then redistribute elements based on the obtained skewed values. However, this approach poses several challenges:

(1) **Double Traversals of S:** Each *data* node must traverse S^i table twice (once for local detection and again for redistribution). In distributed databases, *data* nodes often receive data in the form of data streams. Repeatedly traversing S^i can result in significant memory and I/O overhead.
(2) **Independent Standalone Detection:** In a distributed environment, independently executing detection on each standalone node can lead to varying execution times. Faster nodes may become idle after completing their tasks, while slower nodes can significantly impede the overall detection progress.
(3) **Communication Overhead Due to Consensus:** Achieving consensus on local skewness information from all *data* nodes is essential. However, this requires unavoidable communication overhead between *data* nodes.

Therefore, carrying out a distributed detector is not a trivial task. In response to these challenges, distributed skewness detection must be tightly integrated with the BPPR algorithm. We propose Bala-Join (Balanced Hash Join) by embeding distributed skewness detection into the redistribution and computation phases of BPPR without altering its time complexity, adding only a few necessary steps. In the follows, we will describe the Bala-Join redistribution and computation phases with distributed skewness detector embedded.

Algorithm 5 Redistribution on Data Nodes

Input: data stream $\boldsymbol{S^i}$, $\boldsymbol{R^i}$, number of nodes $\boldsymbol{n}$

```
1: for element e in S^i do
2:     Space Saving Algorithm Sampling Element e
3:     if e in monitor list then
4:         selectNode ← select a node from U(e)
5:         send e to selectNode
6:         signal selectNode that e is a skewed element
7:     else
8:         selectNode ← hash(e) mod n
9:         send element e to selectNode
10:    end if
11: end for
12: for element e in R^i do
13:    selectNode ← hash(e) mod n
14:    send element e to selectNode
15: end for
16: return
```

Redistribution In the original BPPR algorithm steps, both S^i and R^i on *data* node i are partitioned into four parts for redistribution. However, the partitioning process relies on the knowledge of skewed values, making it impossible to complete at this point. Therefore, the way how S^i and R^i are processed on a single *data* node differs from the original BPPR approach.

S^i: S^i takes the form of a data stream, with its length unknown. Space-Saving deployed on S^i can process each element one-after-another in the data stream.
Skewed Elements: elements detected (by Space-Saving) to be skewed will be sent to a sub-partition, which is selected based on the logic in lines 21-34 of the Algorithm 3. A signal will be sent to the node that the element is skewed.
Non-skewed Elements: elements found to be non-skewed will be hash-distributed.
Indeed, after the redistribution phase, S^i has been effectively partitioned into $\widetilde{S^i}$ and $\widehat{S^i}$. Therefore, this process accomplishes three critical objectives for the data stream S^i: local skewness detection, partitioning of S^i, and redistribution of S^i.

R^i: R^i is also a data stream, and no special treatment is applied to it. Instead, all its elements are directly hash-distributed. Following the redistribution phase outlined in Section 4.2, R^i needs to be partitioned into R^i_b and R^i_h. At this stage, all elements in the data stream are temporarily treated as part of R^i_h partition. Consequently, the partitioning of R^i is not completed at this point.

Algorithm 5 outlines the pipeline of the Bala-Join redistribution phase integrated with local skewness detection.

Computation In the original computation phase of BPPR, each *compute* node receives four data partitions. With the introduction of distributed skewness detection, each *compute* node continues to receive these four partitions, but the sources of these data partitions are slightly different.

Algorithm 6 Generation of $\mathcal{R}^i_{bal}$

Input: set $\boldsymbol{M^i}$, current node $\boldsymbol{i}$, number of the nodes $\boldsymbol{n}$
Output: set $\mathcal{R}^i_{bal}$

```
R^i_bal ← []
for element t in M^i do
    fetchNode ← hash(t) mod n
    if fetchNode == i then
        continue
    end if
    tSet ← all element t pulled from fetchNode
    R^i_bal.add(tSet)
end for
return R^i_bal
```

$\mathcal{S}^i_{hash}$: $\mathcal{S}^i_{hash}$ is composed of elements hash-distributed from the $\widehat{S}^i$ partitions of various *data* nodes to the current node.

$\mathcal{S}^i_{bal}$: $\mathcal{S}^i_{bal}$ has the same source of data as the original BPPR. It comprises skewed elements detected in various *data* nodes, which are sent to the current node based on the balanced partitioning.

$\mathcal{R}^i_{hash}$: $\mathcal{R}^i_{hash}$ consists of elements hash-distributed from R to the current node. The union of $\mathcal{R}^i_{hash}$ from all *compute* nodes forms R, which is expressed as $R = \sum_{i=0}^{n-1} \mathcal{R}^i_{hash}$.

$\mathcal{R}^i_{bal}$: After redistribution, each *compute* node receives a set of skewed elements, denoted as M^i. M^i consists of skewed elements informed by *data* nodes to node i. Let $q = hash(t) \bmod n,\ q \neq i$. FoSr each element t in M^i, all elements corresponding to t in R are pulled from *compute* node q. $\mathcal{R}^i_{bal}$ is composed of all these elements and can be represented as:

$$\mathcal{R}^i_{bal} = \{t \mid t \in M^i,\ t \in R\} \tag{31}$$

Algorithm 6 shows how $\mathcal{R}^i_{bal}$ is derived.

The *compute* node can calculate the local join result based on Equations (10) and 11 after obtaining data from the four partitions.

5.3 ASAP mechanism

To ensure the proper execution of HASH JOINs, computing nodes need to be aware of skewness information to fetch the required elements, which corresponds to the second synchronization process mentioned in Section 4.4. However, in a streaming environment, waiting for all elements to be detected and aggregated in order to obtain the global skewness values is impractical. The Bala-Join algorithm, enhanced with distributed skewness detection, imperceptibly establishes a consensus on global skewness information through the **Active-Signaling and Asynchronous-Pulling (ASAP)** mechanism. Specifically, concerning a skewed element x in S, the node receiving x is informed during the redistribution phase that element x is skewed. Then, in the computation phase, the node receiving element x asynchronously pulls all element x from R.

Figure 6 illustrates the specific flow of the skewed element x under the ASAP mechanism. The procedure of Active-Signaling and Asynchronous-Pulling ensures that it does not disrupt the original

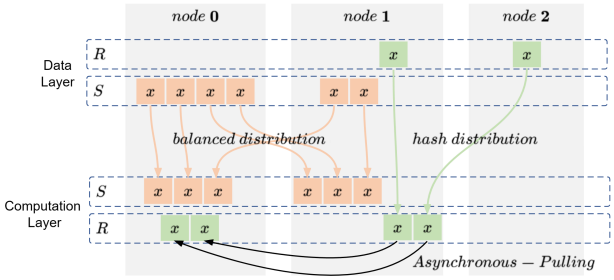


Figure 6: Flow of skewed element x under ASAP

redistribution process. Both active signaling and asynchronous pulling can be completed in a constant time.

Based on Theorem 4.2, the ASAP mechanism ensures correctness when an element x is detected as skewed by the Space-Saving algorithm. Next, we discuss the correctness of the results when the element x is falsely recognized as non-skewed on some data nodes. According to BPPR algorithm, when a *data* node determines that the element x is not skewed, let $q = hash(x) \bmod n$, it will hash distribute this element to node q and will not inform node q that the element x is skewed. On other *data* nodes, when element x is determined as skewed, it will be balanced-distributed to the node set $U(x)$, and these nodes will be notified that element x is a skewed element. Here, the node set $U(x)$ must include node q, which is determined by Algorithm 1. Therefore, the relevant nodes can still form a consensus. The Bala-Join algorithm, equipped with ASAP, has a robust tolerance for skewed value detection. Regardless of whether the skewed value detection is correct or not, it will only affect the performance of the algorithm rather than its correctness.

6 EVALUATION

6.1 Experimental setup

In this experiment, a total of approximately 12000 lines of experimental code were implemented using C++ programming language, supported with C++ standard library and the Google open source framework GRPC. On the experimental environment, Huawei cloud server were employed, with a model of c6s. xlarge. 2. The configuration includes a 4-core CPU, 8GB of memory, a general-purpose SSD, and the Ubuntu 18.04 operating system. The network supports an adjustable bandwidth of 1-300Mbit/s, and three cloud servers in Beijing, Shanghai, and Guiyang were used. The use of these three cloud servers is geographically cross regional, running multiple database nodes on each cloud server can simulate three data centers in different regions for experiments, restoring the execution environment of the real distributed database as much as possible.

In terms of experimental data, the Zipf [**?**] distribution dataset was used, which was widely adopted in many existing works [16, 20, 27–29] for skewed data. Assuming a dataset composed of n elements, the n elements are sorted in descending order of their frequency of occurrence in the dataset. The factor z of Zipf is used to represent the skewness of the dataset, and r corresponds to its sorting level. $r = 1$ is the element with the highest frequency of occurrence, and so on. According to the Zipf distribution, the frequency of the occurrence of elements with $r = 1$ is $x = \frac{1}{H(n,z)}$, where $H(n, z) =$

$\Sigma_{i=1}^{n} \frac{1}{i^z}$ is the sum of the first n terms of the corresponding sequence of p-series, and the frequency of the remaining $r > 1$ elements is $\frac{x}{r^z}$. In our implementation, we adopt Python numpy library to generate a dataset of Zipf distributions.

We evaluate the performance of different redistribution strategies (including PRPD [37], PnR [38] and BPPR) specifically designed for skewed data. Afterwards, we compare the performance of Bala-Join with other Dist-HJ baselines (with skewness detector) listed as follows.

- GraHJ: The popular Dist-HJ implementation adopted by many off-the-shelf DDBMSs.
- Flow-Join: A latest Dist-HJ solution [27] with runtime skew detector.

The performance of the compared baselines are evaluated from two major perspectives: efficiency and network overhead. Efficiency is measured by throughput, representing the number of result records produced every second. This offers a concise view of performance without delving into details like table sizes or execution time. On the other hand, network overhead reflects how an algorithm performs with respect to the network environment; higher transmission suggests more network strain.

6.2 Performance of BPPR

Firstly, we conduct a group of experiments to test the distributed execution strategies in face of different skewness settings. Hereby the skewness information is directly provided for each algorithm as an input knowledge.

Figure 7(a) shows the performance of four algorithms at different bandwidths from 10-300 Mbit/s. GraHJ has the lowest network transmission, while BPPR's transmission is 21% less than PnR but 14% more than PRPD. BPPR performs the best at medium bandwidths and is competitive at high bandwidths, especially when compared to GraHJ.

Figure 7(b) and Figure 7(c) explore the optimal value for BPPR's balance factor. As the balance factor increases, the throughput of BPPR increases in the beginning and arrives the largest value at a balance factor of 0.3. After that, the throughput drops down and eventually becomes similar with GraHJ at 0.7.

Figure 7(d) and Figure 7(e) compare algorithms under different Zipf factors, which control the degree of skewness of data. BPPR outperforms all the other algorithms in the aspect of throughput and ranked 2nd in the aspect of network overhead.

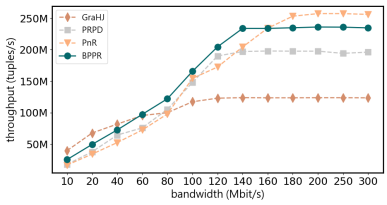
Figure 7(f) and Figure 7(g) test the performance for different $|R|/|S|$ ratios. BPPR remains stable, slightly inferior to PRPD when $|R|/|S|$ is less than 0.1.

Table 2 summarizes the average throughput and rankings of the four sets of experiments. It can be seen that BPPR has a stronger adaptability compared to PRPD and PnR, and it ranks the best in all the experimental settings.

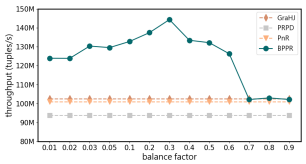
6.3 Performance of distributed detector

In this part, we test the performance for the distributed skewed element detector proposed in Section 5.2. Since the strategy is deeply integrated with BPPR and results in Bala-Join, for fairness, we compare it with two other schemes: BPPR (without skewness detector), which directly provide skewed values to the BPPR algorithm, allowing us to observe any overheads introduced by our distributed runtime detector. BPPR with an independent skewness detector, referred to as BPPR+Det: which employs an independent Space-Saving detection algorithm [3] on BPPR.

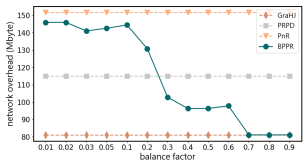
By comparing the three schemes, we can evaluate the additional overhead introduced by the distributed detector. Since all use BPPR for skewed data processing, they exhibit no difference in terms of network overhead. The focus here is solely on comparing the throughput of these methods. In the subsequent experiments, they are respectively abbreviated as BPPR, BPPR+Det, and Bala-Join, with the latter being the strategy introduced in this paper.



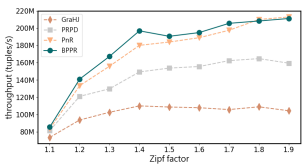
(a) Throughput vs. bandwidths



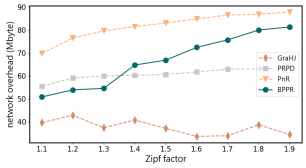
(b) Throughput vs. balance factor



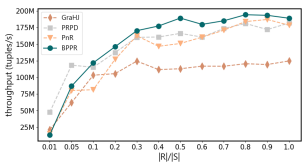
(c) Network overhead vs. balance factor



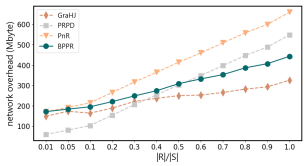
(d) Throughput vs. Zipf factor



(e) Network overhead vs. Zipf factor



(f) Throughput vs. $|R|/|S|$

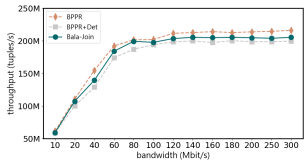


(g) Network overhead vs. $|R|/|S|$

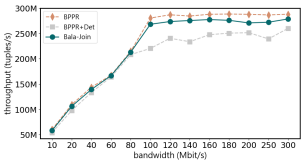
Figure 7: Comparison of distribution strategies in various scenarios

Table 2: Average throughput and ranking of different algorithms over all experiments in Section 6.2

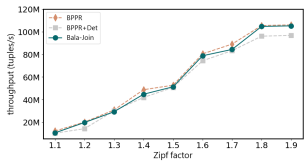
	Bandwidth	Balance Factor	Zipf Factor	$\|R\|/\|S\|$	Comprehensive Ranking
BPPR	2144M (1)	1622M (1)	1602M (1)	1848M (1)	7216M (1)
PnR	2064M (2)	1312M (3)	1546M (2)	1646M (3)	6568M (2)
PRPD	1820M (3)	1218M (4)	1273M (3)	1776M (2)	6087M (3)
GraHJ	1367M (4)	1326M (2)	909M (4)	1239M (4)	4841M (4)



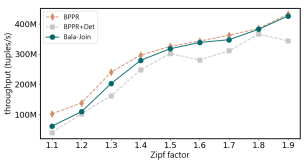
(a) Throughput vs. bandwidth (3 Nodes)



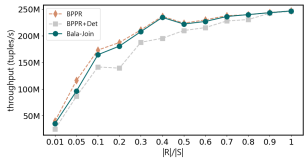
(b) Throughput vs. bandwidth (6 Nodes)



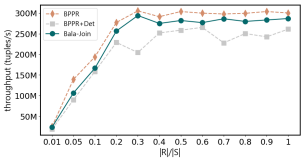
(c) Throughput vs. Zipf factor (3 Nodes)



(d) Throughput vs. Zipf factor (6 Nodes)



(e) Throughput vs. $|R|/|S|$ (3 Nodes)



(f) Throughput vs. $|R|/|S|$ (6 Nodes)

Figure 8: Cost of the runtime skewness detector

Figure 8(a) and Figure 8(b) show the throughput of different strategies in 3-node and 6-node clusters across various bandwidths. In both settings, the performance of Bala-Join lies between BPPR and BPPR+Det.

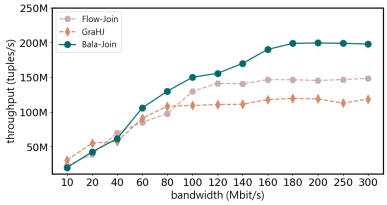
Figure 8(c)-8(f) present the throughput of these strategies under different Zipf factors and $|R|/|S|$ ratios. Consistently, the performance of Bala-Join ranks between the other two strategies.

Throughout all the experiments, the Bala-Join curve is consistently between BPPR and BPPR+Det. This indicates that the distributed detector in Bala-Join introduces some overhead, but is obviously better than simply putting a Space-Saving detetor before BPPR, *i.e.,* BPPR+Det, justifying the challenges as mentioned in Section 5.2. Overall, the overhead of the proposed detector is about 5%, comparing with BPPR. The 6-node setup often incurs more overhead than the 3-node.

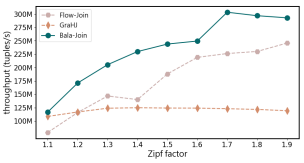
Table 3 summarizes the average throughput per second for the three strategies in each scenario.

Table 3: Average throughput and ranking of different algorithms over all experiments in Section 6.3

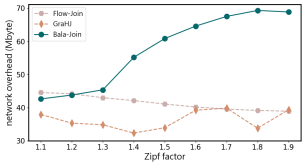
	Bala-Join	BPPR+Det	BPPR
Bandwidth (3 nodes)	$2319M$	$2232M(+4\%)$	$2415M(-4\%)$
Zipf Factor (3 nodes)	$524M$	$500M(+5\%)$	$548M(-4\%)$
$\|R\|/\|S\|$ (3 nodes)	$2327M$	$2141M(+9\%)$	$2377M(-2\%)$
Bandwidth (6 nodes)	$2868M$	$2602M(+10\%)$	$2985M(-4\%)$
Zipf Factor (6 nodes)	$2468M$	$2151M(+15\%)$	$2628M(-6\%)$
$\|R\|/\|S\|$ (6 nodes)	$2811M$	$2451M(+15\%)$	$3030M(-6\%)$



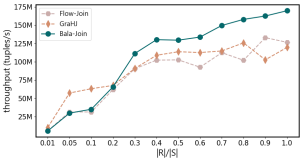
(a) Throughput vs. bandwidths (3 Nodes)



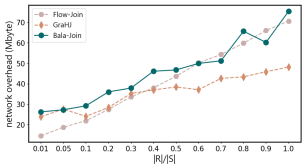
(b) Throughput vs. Zipf factor)



(c) Network overhead vs. Zipf factor (3 Nodes)



(d) Throughput vs $|R|/|S|$ (3 Nodes)



(e) Network overhead vs. $|R|/|S|$ (3 Nodes)

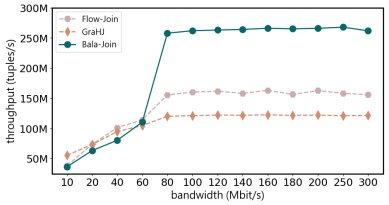
Figure 9: Comparison of Dist-HJ solutions in various scenarios (3 Nodes)

6.4 Performance of comprehensive Dist-HJ solutions

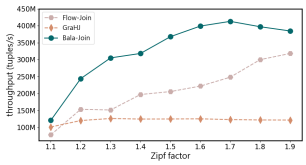
Finally, we evaluate the comprehensive Dist-HJ solutions (the whole pipeline including skewness detection, redistribution, computation). Figure 9 and Figure 10 show the comparison of solutions under different scenarios for 3-node and 6-node clusters, respectively. Table 4 summarizes the average throughput per second of each solution in various scenarios, as well as the improvement of our solution relative to Flow-Join and GraHJ.

We also evaluated the Dist-HJ solutions on a modified Star Schema Benchmark [25] (SSB-skew[18]) with a scale factor of 10, focusing on join operations across 24 nodes with varying skew levels in the `lineorder` table columns (`LO_PARTKEY`, `LO_ORDERDATE`, `sLO_CUSTKEY`, `LO_SUPPKEY`). Figure 11 shows that as skew increases, BPPR progressively outperforms GraHJ. Notably, even at lower skew levels, BPPR and Flow Join showed performance differences. This occurs because tuples with identical values are placed next to each other during data splitting, leading the detector to potentially classify the initially received tuples as skewed.

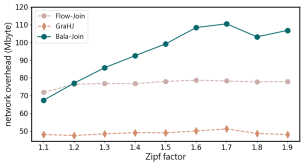
Among the above experimental scenarios, our comprehensive solution consistently outperforms Flow-Join in terms of throughput. However, it did not have an advantage over Flow-Join in the aspect of network overhead. Furthermore, Flow-Join, relying on its SFR strategy in the skewness processing, exhibited a trend of decreasing network overhead while the number of nodes in the cluster increases from 3 to 6. Nevertheless, the results from our test scenarios still showcase how our solution strikes a better balance between



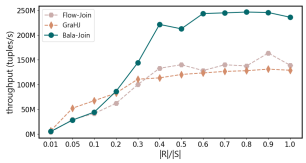
(a) Throughput vs. bandwidths (6 Nodes)



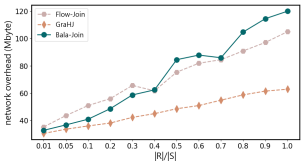
(b) Throughput vs. Zipf factor (6 Nodes)



(c) Network overhead vs. Zipf factor (6 Nodes)

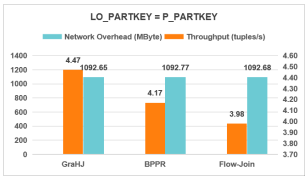


(d) Throughput vs $|R|/|S|$ (6 Nodes)

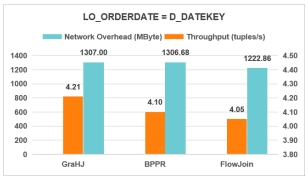


(e) Network overhead vs. $|R|/|S|$ (6 Nodes)

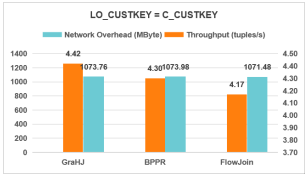
Figure 10: Comparison of Dist-HJ solutions in various scenarios (6 Nodes)



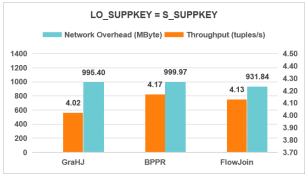
(a) LO_PARTKEY = P_PARTKEY



(b) LO_ORDERDATE = D_DATEKEY



(c) LO_CUSTKEY = C_CUSTKEY



(d) LO_SUPPKEY = S_SUPPKEY

Figure 11: Performance of Dist-HJ solutions on SSB-skew

network and computational performance. Despite higher network overhead, our algorithm displays superior execution efficiency.

7 CONCLUSIONS

Distributed Hash Join is a fundamental operator in distributed databases, the implementation of which in most DDMSs is based on the principle of hash partitioning. While this approach is straightforward to develop and maintain, the system experiences significant throughput degradation when confronted with data skewness. In this paper, we propose BPPR algorithm to handle skewed data with high adaptability and outstanding performance. We also theoretically proved the correctness for the algorithm. Subsequently, we propose a distributed skewness detection strategy tailored to the BPPR algorithm. It deeply couples with BPPR algorithm and adopts the Active-Signaling and Asynchronous-Pulling (ASAP) mechanism. The mechanism imperceptibly enables the real-time provision of skewness information to BPPR, limiting the additional overhead of skewed value detection. Experiments results justifies that our comprehensive Dist-HJ solution, namely Bala-Join, effectively balances network and computational performance, achieving superior overall efficiency.

Table 4: Average throughput and ranking of different Dist-HJ implementations over all experiment in Section 6.4

	Bala-Join	Flow-Join	GraHJ
Bandwidth (3 nodes)	$1818M$	$1460M(+25\%)$	$1261M(+44\%)$
Zipf Factor (3 nodes)	$2106M$	$1594M(+32\%)$	$1082M(+95\%)$
$\|R\|/\|S\|$ (3 nodes)	$1281M$	$994M(+29\%)$	$1087M(+18\%)$
Bandwidth (6 nodes)	$2659M$	$1761M(+51\%)$	$1423M(+89\%)$
Zipf Factor (6 nodes)	$2953M$	$1870M(+58\%)$	$1091M(+171\%)$
$\|R\|/\|S\|$ (6 nodes)	$1957M$	$1217M(+61\%)$	$1192M(+65\%)$

REFERENCES

[1] Khaled Alsabti and Sanjay Ranka. 2001. Skew-insensitive parallel algorithms for relational join. *Journal of King Saud University-Computer and Information Sciences* 13 (2001), 79–110.

[2] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-scale in-memory join processing using RDMA. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1463–1475.

[3] Massimo Cafaro, Marco Pulimeno, and Piergiulio Tempesta. 2016. A parallel space saving algorithm for frequent items and the hurwitz zeta distribution. *Information Sciences* 329 (2016), 1–19.

[4] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2004. Finding frequent items in data streams. *Theoretical Computer Science* 312, 1 (2004), 3–15.

[5] Edward G Coffman, Jr, Michael R Garey, and David S Johnson. 1978. An application of bin-packing to multiprocessor scheduling. *SIAM J. Comput.* 7, 1 (1978), 1–17.

[6] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[7] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[8] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. 2002. Frequency estimation of internet packet streams with limited space. In *Esa*, Vol. 2. Citeseer, 348–360.

[9] Hasanat M Dewan, Kui W Mok, Mauricio Hernández, and Salvatore J Stolfo. 1994. Predictive dynamic load balancing of parallel hash-joins over heterogeneous processors in the presence of data skew. In *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*. IEEE, 40–49.

[10] David J DeWitt, Jeffrey F Naughton, Donovan A Schneider, and Srinivasan Seshadri. 1992. *Practical skew handling in parallel joins.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[11] Elaheh Gavagsaz, Ali Rezaee, and Hamid Haj Seyyed Javadi. 2019. Load balancing in join algorithms for skewed data in MapReduce systems. *The Journal of Supercomputing* 75 (2019), 228–254.

[12] Phillip B Gibbons and Yossi Matias. 1998. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 331–342.

[13] Ronald L. Graham. 1969. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics* 17, 2 (1969), 416–429.

[14] Kien A Hua and Chiang Lee. 1991. Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning.. In *VLDB*, Vol. 91. 525–535.

[15] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[16] Qun Huang and Patrick PC Lee. 2014. Ld-sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 1420–1428.

[17] Wentao Huang, Yunhong Ji, Xuan Zhou, Bingsheng He, and Kian-Lee Tan. 2023. A Design Space Exploration and Evaluation for Main-Memory Hash Joins in Storage Class Memory. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1249–1263.

[18] David Justen, Daniel Ritter, Campbell Fraser, Andrew Lamb, Nga Tran, Allison Lee, Thomas Bodner, Mhd Yamen Haddad, Steffen Zeuch, Volker Markl, et al. [n.d.]. POLAR: Adaptive and Non-invasive Join Order Selection via Plans of Least Resistance. ([n. d.]).

[19] Masaru Kitsuregawa. 1995. Dynamic join product skew handling for hash-joins in shared-nothing database systems. In *Proceedings Of The Fourth International Conference on Database Systems For Advanced Applications*, Vol. 5. World Scientific, 246.

[20] Masaru Kitsuregawa and Yasushi Ogawa. 1990. Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC).. In *VLDB*. 210–221.

[21] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. 1983. Application of hash to data base machine and its architecture. *New Generation Computing* 1, 1 (1983), 63–74.

[22] Robert Lasch, Mehdi Moghaddamfar, Norman May, Süleyman Sirri Demirsoy, Christian Färber, and Kai-Uwe Sattler. 2022. Bandwidth-optimal relational joins on FPGAs. In *EDBT*. 1–27.

[23] Gurmeet Singh Manku and Rajeev Motwani. 2002. Approximate frequency counts over data streams. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 346–357.

[24] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *Database Theory-ICDT 2005: 10th International Conference, Edinburgh, UK, January 5-7, 2005. Proceedings 10*. Springer, 398–412.

[25] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. 2007. The star schema benchmark (SSB). *Pat* 200, 0 (2007), 50.

[26] Orestis Polychroniou, Rajkumar Sen, and Kenneth A Ross. 2014. Track join: distributed joins with minimal network traffic. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1483–1494.

[27] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. 2016. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 1194–1205.

[28] Wolf Rödiger, Tobias Mühlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2014. Locality-sensitive operators for parallel main-memory database clusters. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 592–603.

[29] Ambuj Shatdal and Jeffrey F Naughton. 1993. Using shared virtual memory for parallel join processing. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. 119–128.

[30] Michael Stonebraker. 1986. The case for shared nothing. *IEEE Database Eng. Bull.* 9, 1 (1986), 4–9.

[31] Wenbo Sun, Asterios Katsifodimos, and Rihan Hai. 2023. An Empirical Performance Comparison between Matrix Multiplication Join and Hash Join on GPUs. In *2023 IEEE 39th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 184–190.

[32] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.

[33] Aleksandar Vitorovic, Mohammed Elseidy, and Christoph Koch. 2016. Load balancing and skew resilience for parallel joins. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. Ieee, 313–324.

[34] Joel L Wolf, Daniel M Dias, and Philip S. Yu. 1991. An effective algorithm for parallelizing hash joins in the presence of data skew. In *Proceedings. Seventh International Conference on Data Engineering*. IEEE Computer Society, 200–201.

[35] Joel L Wolf, Daniel M Dias, and Philip S. Yu. 1993. A parallel sort merge join algorithm for managing data skew. *IEEE Transactions on Parallel and Distributed Systems* 4, 1 (1993), 70–86.

[36] Joel L. Wolf, Daniel M. Dias, Philip S. Yu, and John Turek. 1994. New algorithms for parallelizing relational database joins in the presence of data skew. *IEEE Transactions on Knowledge and Data Engineering* 6, 6 (1994), 990–997.

[37] Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. 2008. Handling data skew in parallel joins in shared-nothing systems. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 1043–1052.

[38] Jinxin Yang, Hui Li, Yiming Si, Hui Zhang, Kankan Zhao, Kewei Wei, Wenlong Song, Yingfan Liu, and Jiangtao Cui. 2024. One Size Cannot Fit All: a Self-Adaptive Dispatcher for Skewed Hash Join in Shared-nothing RDBMSs. In *Proceedings Of The 29th International Conference on Database Systems For Advanced Applications*.

[39] Shunjie Zhou, Fan Zhang, Hanhua Chen, Hai Jin, and Bing Bing Zhou. 2019. Fastjoin: A skewness-aware distributed stream join system. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1042–1052.

[40] Xiaofang Zhou and Maria E Orlowska. 1995. Handling data skew in parallel hash join computation using two-phase scheduling. In *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, Vol. 2. IEEE, 527–536.