Bala-Join: An Adaptive Hash Join for Balancing Communication and Computation in Geo-Distributed SQL Databases

***Abstract*—Shared-nothing geo-distributed SQL databases, such as CockroachDB, are increasingly vital for enterprise applications requiring data resilience and locality. However, our joint database R&D team with Inspur encountered significant performance degradation in a real-world customer deployment spanning multiple data centers over a Wide Area Network (WAN). We determined that this issue is rooted in the performance of the Distributed Hash Join (Dist-HJ) algorithm, which is contingent upon a crucial balance between communication overhead and computational load. This balance is severely disrupted when processing skewed data from real-world customer workloads, leading to the observed performance decline.**

To tackle this specific industrial challenge, we introduce Bala-Join, an adaptive solution to balance the computation and network load in Dist-HJ execution. Our approach consists of the Balanced Partition and Partial Replication (BPPR) algorithm and a distributed online skewed join key detector. The former achieves balanced redistribution of skewed data through a multicast mechanism to improve computational performance and reduce network overhead. The latter provides real-time skewed join key information tailored to BPPR. Empirical study shows that Bala-Join outperforms the popular Dist-HJ solutions, increasing throughput by 25%-61%.

***Index Terms*—distributed hash join, shared-nothing, data skew**

I. Introduction

Shared-nothing [1] DBMSs such as CockroachDB [2], TiDB [3], Spanner [4] and OceanBase [5] are increasingly deployed to support geo-distributed applications that allocate data across various servers or data centers. In customer deployments built upon CockroachDB, our joint database research and development team with Shandong Inspur Database Technology Company Ltd. has encountered a significant challenge involving distributed query processing across wide-area networks (WANs). Specifically, for complex analytical queries involving distributed hash joins (Dist-HJ) that span data centers in different cities, the combination of real-world data skew and the high latency/limited bandwidth of the WAN creates a critical performance bottleneck.

This challenge is a direct consequence of the distributed hash join's execution model. As an essential method to implement the join operator, HASH JOIN relies on the hash table to efficiently match rows of two relations. The smaller relation is built into a hash table, while the bigger one is probed to match the tuples. At this point, the two relations are called the **build table** and the **probe table**, respectively. A general distributed hash join typically involves the following three steps:

Redistribution: While the build and probe tables reside on various *data* nodes, the tuples need to be redistributed to multiple *compute* nodes for HASH JOIN tasks.

Computation: Each *compute* node performs local HASH JOIN computations on the received shares of tuples.

Aggregation: *Response* node aggregates the results of each *compute* node while there is no follow-up task, *e.g.,* another HASH JOIN. Otherwise, the *compute* nodes subsequently transform into *data* nodes, providing input for subsequent operations.

The default hash partitioning strategies used by the underlying CockroachDB foundation, while effective for local networks, often fail to adequately balance the trade-off between network communication load and overall query response time, especially in the presence of data skew. Consider a typical join query within the TPC-H benchmark shown below, the $customer$ table and the $orders$ table are joined on the customer key (*CUSTKEY*). Orders from the same customer are distributed to the same *compute* node. If some customers hold a substantial number of orders, *i.e.,* data skew, the corresponding compute nodes will undertake much more computational tasks than the other nodes.

```
SELECT COUNT(*)
FROM CUSTOMER JOIN ORDERS
ON O_CUSTKEY = C_CUSTKEY;
```

Fig. 1 provides a real example executing the above query in CockroachDB after introducing a 10% data skew in the join between the CUSTOMER and ORDERS tables. The execution times of the three compute nodes (HashJoiners) are *72ms, 9.1s, and 69ms*, respectively. The data skew results in an unbalanced distribution of computational load across nodes, with one node bearing significantly higher computational overhead, thereby becoming the performance bottleneck of the overall query execution.

Plenty of studies [6]–[8] have justified the aforementioned problem as a universal phenomenon in practice [8]–[10]. Several efforts [8], [10]–[12] have been conducted to address this issue. The common idea is to employ different redistribution strategies for skewed and non-skewed data, respectively. This relies on the perception of skewed data, either through pre-collected statistics or online detection. Additionally, large-scale data transmission across nodes over a high-latency, limited-bandwidth WAN can lead to performance degradation [13], particularly when using the broadcast strategy com-

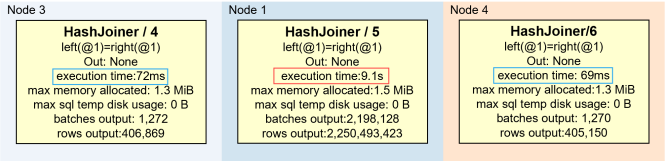


Fig. 1: Impact of data skew on distributed hash join

monly employed by existing solutions. Despite their promising progress in alleviating the load imbalance, the overall performance is still subject to the original data distribution across nodes [10] or favorable network conditions [12]. For instance, there exist many queries joining tables spanned over Beijing and Shanghai (with a distance larger than $1000km$) in Inspur group, a giant Chinese company providing high-performance computational infrastructure. We observe that in these cross-region joins, network overhead remains a critical factor for the join performance even if we replace the default Dist-HJ implementation with a series of alternatives tailored to load balance [8], [10]–[12]. The reason is these solutions (*i.e.,* PRPD [10] and PnR [12]) typically focus on either network overhead or balance, thus limiting their overall performance.

In this paper, we propose Bala-Join, which consists of a novel redistribution strategy and an online skewed join key detector. The former is adaptable to various skew scenarios and consistently performs well, while the latter can identify skewed join keys within intermediate join tables at runtime. In summary, our contributions are as follows.

- We present a novel redistribution strategy (BPPR) for Dist-HJ, with theoretically guaranteed balance. By introducing a balance factor and a multicast mechanism, it ensures a minimized network overhead with guaranteed balance load (Section 3),
- We present a distributed mechanism for skew detection, while minimizing additional overhead. The detection mechanism is further deeply integrated with BPPR. (Section 4),
- Empirical study demonstrates the performance of our solution in balancing the load and network overhead. (Section 5).

II. Preliminary and Motivation

A. Shared-nothing DBMSs

The shared-nothing architecture is widely adopted for distributed DBMSs, where each node operates independently with standalone CPU, memory, and storage. In these DBMSs, Dist-HJ is executed in parallel. Since data shards reside on different nodes, tuples with the same join key need to be redistributed to the same compute node. Each compute node then performs a local hash join, after which the results are aggregated by the response node. Although hash partitioning is the most commonly used redistribution strategy in practice, it can lead to significant imbalance under data skew. In such cases, newly added nodes cannot share the skewed data from the hot nodes, thus limiting the scalability. For instance, as a representative shared-nothing DBMS, CockroachDB [2] organizes data into ranges that are distributed and replicated throughout the cluster. When executing Dist-HJ, CockroachDB dynamically selects range replicas to generate the physical execution plan. Thus, the participating nodes for a given join query are not predetermined, and the volume of data contributed by each node may vary substantially. Moreover, the default Dist-HJ in CockroachDB employs hash partitioning as the exclusive redistribution strategy. Therefore, when data skew occurs, it can cause a load imbalance across the cluster.

B. Distributed Hash Join

This part explores the implementation of distributed hash join, focusing on several essential algorithms that optimize performance under different data distribution scenarios.

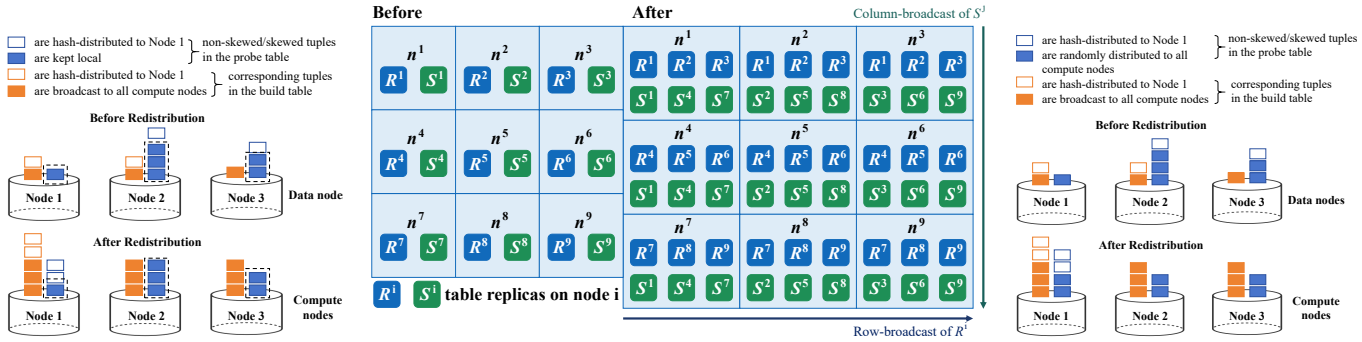
Grace Hash Join: Grace Hash Join (GraHJ) [14] is a widely used implementation for distributed hash join in shared-nothing architectures. For join tables R (build table) and S (probe table), GraHJ employs a hash function to partition both R and S into identical numbers of partitions, ensuring that each partition of R can fit in memory for the construction of the hash table. Using the same hash function for R and S, the tuples that match in R and S end up in the same partition, ensuring the correctness of the result. Afterwards, each partition contains a subset of R and S. The partitions are independent, and each must only perform its own HASH JOIN calculation. The union of the results from all partitions constitutes the final output.

PRPD: GraHJ can experience degradation of efficiency under data skew. PRPD deals with skewed tuples and non-skewed tuples individually; the former is kept local while the latter follows a hash partitioning strategy. The corresponding tuples with skewed join keys in the build table would be broadcast to all nodes in the cluster. PRPD is designed to minimize network overhead and achieve a balanced load. Its optimal performance is based on the assumption that the skewed tuples are distributed uniformly across the nodes. However, due to the unpredictable distribution of skewed tuples, it is difficult to consistently achieve the expected favorable performance.

SFR: SFR is the redistribution strategy adopted by Flow-Join [8]. For tuples that are skewed only in the build table or the probe table, Flow-Join adopts the PRPD strategy. For tuples that are skewed in both tables, Flow-Join employs the Symmetric Fragment Replicate (SFR) strategy, where the nodes in the cluster are logically arranged in a matrix. The tuples from the build table are replicated across the matrix (node) horizontally, while the corresponding tuples from the probe table are replicated to nodes vertically.

PnR: PnR [12] behaves similarly to Flow-Join, but selects to distribute the skewed tuples randomly or in a round-robin manner across all nodes in the cluster. The corresponding tuples from the probe table are broadcast across the cluster.

Figure 2 shows an overview of the redistribution strategies designed for data skew. In the following, we shall discuss the applicable scenarios and limitations of these approaches, thereby clarifying our objectives.



(a) PRPD algorithm (b) SFR algorithm (c) PnR algorithm

Fig. 2: Redistribution strategies overview

C. Observations and Targets

PRPD and SFR focus on optimizing network overhead, while PnR emphasizes load balance. Both groups have their advantages in specific scenarios, but may suffer from limitations in more generalized situations.

As PRPD keeps the skewed tuples local, some nodes still bear a higher load than others when the original distribution of the skewed tuples is not uniform across nodes. Similarly, in SFR, when a row (*resp.,* column) in the matrix contains many more skewed tuples than other rows (*resp.,* column), the nodes on that row (*resp.,* column) will be overloaded. That is, PRPD and SFR are not suitable for scenarios where the original distribution of skewed tuples is excessively imbalanced.

PnR achieves balanced redistribution across various scenarios at the cost of massive network overhead. Nodes with more skewed tuples bear a heavier load during the redistribution phase. Furthermore, PnR suffers from the same problem as PRPD, especially when the volumes of both tables are nearly identical.

PRPD, Flow-Join and PnR all suffer from limitations in either the computation or the redistribution phase. In summary, previous strategies face the following challenges:

Chal. ❶ *Dependence on the original distribution*: Any strategy keeping skewed tuples local would experience performance degradation when the original distribution of skewed tuples is non-uniform across nodes.

Chal. ❷ *Dependence on statistics*: Previous approaches require the identification of skewed tuples. Moreover, it is also essential for Flow-Join and PnR to determine the number of skewed tuples in both the build and the probe table. For joins over intermediate tables, tuples are received as data streams, which implies that redistribution strategies can only be applied after all tuples have arrived and the statistical information has been obtained.

Chal. ❸ *Non-adaptive strategies*: Given a tuple, once the necessary statistical information is acquired, the redistribution will follow a specific strategy. Therefore, a dynamic load balance within the cluster is not achievable. In streaming scenarios, tuples determined to be non-skewed at the beginning may eventually become skewed as the data stream progresses. Therefore, the correctness of the distributed join is challenged.

TABLE I: Notations

Symbol	Description
R, S	build table and probe table
R^i, S^i	data shards of R, S at node i
R_{skew}, S_{skew}	skewed tuples in R, S
R_{non}, S_{non}	non-skewed tuples in R, S
R^i_{Sskew}, R^i_{Snon}	tuples in R^i that intersect with S_{skew} or not
S^i_{skew}	all skewed tuples of S^i
$U(x)$, $U^i(x)$	set of nodes to dispatch tuple t (on node i) with join key $\boldsymbol{x}$
$C(t)$	function to choose a node from $U(t)$
$\mathcal{S}^i_{hash}$	tuples from S_{non} that node i receives through hash distribution
$\mathcal{S}^i_{bal}$	tuples from S_{skew} that node i receives through a balanced distribution
$\mathcal{R}^i_{hash}$	tuples from $R^0_{Snon}, R^1_{Snon}, \ldots, R^{n-1}_{Snon}$ that node i receives through hash distribution
$\mathcal{R}^i_{bal}$	tuples from $R^0_{Snon}, R^1_{Snon}, \ldots, R^{n-1}_{Snon}$ that node i receives through multicast
$\widehat{S}^j_{skew}$	skewed tuples received by node j
$\widehat{S}^{ij}_{skew}$	tuples distributed to node j from S^i_{skew}

In order to address the challenges above, we present Bala-Join, which dynamically detects and deals with skewed tuples. Bala-Join immediately distributes tuples upon arrival, without waiting for the complete statistical information. BPPR algorithm of Bala-Join ensures dynamic balance in the cluster that is independent of the original data distribution. Tuples with the same join key, whether determined to be skewed or not, will always be distributed to the same subset of nodes. The integration of BPPR with the distributed detector and the ASAP mechanism achieves real-time detection and distribution, significantly improving the performance of the distributed hash join.

III. BALA-JOIN

A. Notations

Before presenting the details, we shall first introduce some notation. Assuming that distributed hash join is implemented in a n-nodes cluster to accomplish $R \underset{R.a=S.b}{\bowtie} S$, suppose R and

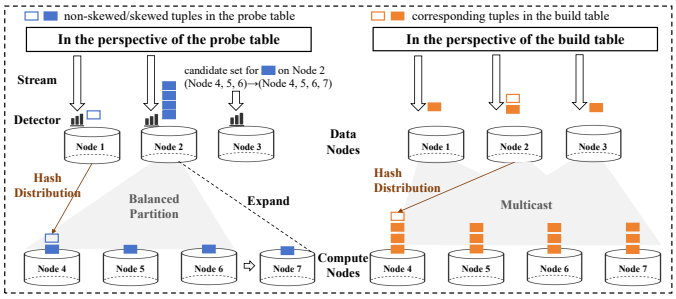


Fig. 3: Overview of the Bala-Join

S represent the build table and the probe table, which can be either intermediate results or original tables. Both R and S can be distributed across multiple data nodes. The tuples residing at node i are denoted as R_i and S_i, each corresponding to a subset of R and S, respectively.

R and S may contain skewed tuples denoted R_{skew} and S_{skew}. For instance, R_{skew} is the set of skewed tuples with the frequent join keys that occur in R. Skewed tuples are defined on the basis of a frequency threshold, where $f(\cdot)$ denotes the relative frequency.

$$R_{skew} = \{t | t \in R, f(t.a) \geq \lambda\}, S_{skew} = \{t | t \in S, f(t.b) \geq \lambda\} \quad (1)$$

Similarly, non-skewed data are denoted as R_{non} and S_{non}, respectively. Obviously, $R = R_{non} \bigcup R_{skew}, S = S_{non} \bigcup S_{skew}$. Other key symbols in this work are summarized in Table I.

B. System Overview

Bala-Join consists of a novel redistribution strategy, BPPR, and an online skewed join key detector. The build and probe tables participate in the join as data shards or data streams across data nodes. Each tuple is tested to be skewed or not by distributed detectors deployed on the nodes and then dispatched following BPPR.

To ensure the correctness of the join, when a tuple from one table is distributed to a certain node, all tuples with the same join key in the other table must be dispatched to that node. Since the probe table is typically much larger than the build one, Bala-Join distributes the former in a balanced manner while multicasts the latter to the corresponding nodes. In order to achieve load balance in the cluster, Bala-Join disperses skewed tuples in the probe table across a set of nodes, of which the cardinality should be minimized to reduce the network overhead caused by muticasting the corresponding tuples.

In BPPR, each data node maintains a node set $U^i(x)$ for each join key $\boldsymbol{x}$ of the skewed tuples. The newly arrived tuple will be distributed to the proper node to meet the balance factor defined in Section III-C. When none of the nodes are adequate, the node set will be expanded. The tuples with the same join key in the build table will soon be multicast to the node set afterwards. Figure 3 shows an overview of Bala-Join.

C. Balanced Partition and Partial Replication

A. BPPR Workflow

BPPR follows the general workflow of Dist-HJ, but contains a tuple-level partitioning step, which aims to further divide the skewed tuples into finer-grained partitions to achieve better balance. Given the join task of $R \underset{R.a=S.b}{\bowtie} S$, it works as follows.

Step 1, Redistribution. In a distributed environment, each data node typically holds data fragments of the build table R and the probe table S, denoted R^i and S^i for node i. Before redistribution, we introduce an operator $\Gamma : \mathbb{R} \mapsto \mathbb{R}^2$, where $\mathbb{R}$ denotes the field of relations. In particular, we define $\Gamma(S^i) = (S^i_{skew}, S^i \setminus S^i_{skew})$ and $\Gamma(R^i) = (R^i_{Sskew}, R^i \setminus R^i_{Sskew})$. Specifically,

$$R^i_{Sskew} = \{t \mid t \in R^i,\ t.a \in S_{skew}.b\} \quad (2)$$

For ease of discussion, we denote $S^i \setminus S^i_{skew}$ as S^i_{non} and $R^i \setminus R^i_{Sskew}$ as R^i_{Snon}, respectively.

Given the above, R^i and S^i are divided into four parts: S^i_{non}, S^i_{skew}, R^i_{Snon}, and R^i_{Sskew}, respectively. These parts are further partitioned and redistributed as follows:

S^i_{non}: This part contains the non-skewed tuples from S^i . The tuples are hash-partitioned and distributed, achieving nearly even distribution across n nodes.

S^i_{skew} : This part contains the skewed data from S^i, and each tuple $t \in {S_{skew}}^i$ is sent to a specific node, denoted as $C(t)$. For skewed tuples with the same join key $\boldsymbol{x}$, BPPR distributes them to a node set $U(x)$, and finally achieves a balance redistribution. For ease of discussion, we formally introduce *balance factor*, referred to as B, to quantify the degree of balance among compute nodes.

$$B = (\max_j |\widehat{S}^j_{skew}| - \min_j |\widehat{S}^j_{skew}|)/\max_j |\widehat{S}^j_{skew}| \quad (3)$$

$\widehat{S}^j_{skew}$ represents the skewed tuples received by *compute* node j. Naturally, a balanced redistribution of skewed tuples should satisfy $B \leq \epsilon$. where ϵ is a predefined threshold. For each skewed tuple at arrival, $C(t)$ attempts to select a target node from the set of nodes and test whether it is balanced enough. If none of the nodes in $U(x)$ meet the requirement, $U(x)$ is expanded to include additional nodes for selection (shall be discussed later in Algorithm 3).

R^i_{Sskew}: According to Eq. (2), each tuple in R_{Sskew} can be matched by a group of tuples in S_{skew}. Since S_{skew} is redistributed across nodes, each tuple in R_{Sskew} with the join key $\boldsymbol{x}$ must be multicast to the nodes specified by $U(x)$, which we refer to as *Partial Replication*.

R^i_{Snon}: The dispatch of tuples in R^i_{Snon} is independent of S_{skew}. Therefore, R^i_{Snon} are partitioned and distributed by hash.

Step 2, Computation. During the computation phase, each compute node receives the redistributed tuples from the data nodes according to **Step 1**. Consequently, each compute node will receive four sets of tuples:

$\mathcal{S}^i_{hash}$: This set is composed of partial tuples from S^i_{non} across n data nodes. During the redistribution phase, S^i_{non} on each node is hash-distributed to multiple compute nodes.

$\mathcal{S}^i_{hash}$ thus consists of the tuples from $S^0_{non}, S^1_{non}, \ldots, S^{n-1}_{non}$ that have been hash-distributed to node i. Hence, $\mathcal{S}^i_{hash}$ can be represented as:

$$\mathcal{S}^i_{hash} = \{t \mid (hash(t.b) \mod n) = i,\ t \in \bigcup_{i=0}^{n-1} S^i_{non}\} \quad (4)$$

$\mathcal{S}^i_{bal}$: This set is derived from the S^i_{skew} sent by the n data nodes. During the redistribution phase, the tuple $t \in S^i_{skew}$ is sent to the node $C(t)$. Therefore, $\mathcal{S}^i_{bal}$ is the set of tuples of $S^0_{skew}, S^1_{skew}, \ldots, S^{n-1}_{skew}$ where $C(t) = i$. $\mathcal{S}^i_{bal}$ can be represented as follows:

$$\mathcal{S}^i_{bal} = \{t \mid C(t) = i,\ t \in \bigcup_{i=0}^{n-1} S^i_{skew}\} \quad (5)$$

$\mathcal{R}^i_{hash}$: This set consists of partial tuples from R^i_{Snon} across n data nodes. $\mathcal{R}^i_{hash}$ is similar to $\mathcal{S}^i_{hash}$ and represents the tuples hashed to node i from $R^0_{Snon}, \ldots, R^{n-1}_{Snon}$. $\mathcal{R}^i_{hash}$ can be represented as:

$$\mathcal{R}^i_{hash} = \{t \mid (hash(t.a) \mod n) = i,\ t \in \bigcup_{i=0}^{n-1} R^i_{Snon}\} \quad (6)$$

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

$$\mathcal{R}^i_{bal} = \{t \mid i \in U(t.a),\ t \in \bigcup_{i=0}^{n-1} R^i_{Sskew}\} \quad (7)$$

Upon receiving the above four tuple sets, each compute node first performs a local HASH JOIN as follows:

$$\mathcal{R}^i_{bal} \underset{a=b}{\bowtie} \mathcal{S}^i_{bal}, \qquad \mathcal{R}^i_{hash} \underset{a=b}{\bowtie} \mathcal{S}^i_{hash} \quad (8)$$

The local result for compute node i is given by the following:

$$R^i \underset{a=b}{\bowtie} S^i = (\mathcal{R}^i_{bal} \underset{a=b}{\bowtie} \mathcal{S}^i_{bal}) \bigcup (\mathcal{R}^i_{hash} \underset{a=b}{\bowtie} \mathcal{S}^i_{hash}) \quad (9)$$

Step 3, Union. After the computation phase, each compute node obtains local results. To aggregate them into a final result, a coordinating node is required to merge the outputs from all compute nodes. The aggregation can be described as follows::

$$R \underset{R.a=S.b}{\bowtie} S = \bigcup_{i=0}^{n-1} (R^i \underset{R.a=S.b}{\bowtie} S^i) \quad (10)$$

In general, instead of dispersing the tuples as evenly as possible for each skewed join key (*e.g.,* PRPD and PnR), BPPR considers the load balance from the perspective of the whole cluster. For instance, PnR distributes skewed tuples with the join key $\boldsymbol{x}$ uniformly across the nodes, while BPPR only dispatches them to $U(x)$. Although the tuples with the join key $\boldsymbol{x}$ may not achieve a balanced distribution in BPPR, the overall distribution of all the skewed tuples is balanced. which shall be justified in our empirical study in Section V.

B. Balanced Partition Algorithm

Intuitively, according to the above strategy, for skewed tuples with a join key $\boldsymbol{x}$, each data node should send them to the same set of nodes, thus reducing the network overhead caused by multicasting the build table. However, in a distributed environment, this would require significant network communication to achieve consensus. When the join tables are intermediate results or data streams, real-time synchronization becomes more critical. In order to solve this problem, we propose the *Balanced Partition* algorithm based on a sequence generator, which achieves consensus without perception while ensuring the load balance in the cluster.

Without a sequence generator, we assume that each data node maintains an individual node set $U^i(x)$ for each skewed join key $\boldsymbol{x}$. This presents two requirements:

(1) a global $U(x) = \bigcup_{i=0}^{n-1} U^i(x)$ must be obtained for tuples from R_{Sskew} to be multicast;

(2) $|U(x)|$ must be minimized to reduce network overhead.

Requirement (1) will be ensured by the ASAP mechanism that we shall present in Section IV-B. To ensure (2), the sets $U^i(x)$ for $i = 0, \ldots, n-1$ should be nested, *i.e.,* each set contained by the larger set. To fulfill that, we present a sequence generator algorithm, *i.e.,* Algorithm 1. For a skewed join key $\boldsymbol{x}$, each data node would generate the same sequence following the algorithm. Each element in the sequence is assigned to a node. Lines 2-3 initialize an empty sequence. Lines 4-11 detail the sequence generation process. Line 6 generates 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. Note that when $epoch$ is 0, the first element in the sequence is hash(x) mod n, which corresponds to the target node for the tuple with the join key $\boldsymbol{x}$ under the hash distribution. This means that when $\boldsymbol{x}$ is first identified as non-skewed and later identified as skewed, its target node will always remain in $U(x)$. This resolves the issue raised in **Chal. ❸**.

Algorithm 1 ensures consistency of sequences generated for the same skewed join key across various nodes. Based on this, each data node can expand $U^i(x)$ following the same sequence. Algorithm 2 demonstrates how to update $U^i(x)$, *i.e.,* as long as $|U^i(x)| < n$, the next node in the sequence is added to $U^i(x)$ each time.

Given that each data node has achieved a consensus on the expansion of $U^i(x)$, we move on to chase for a load balance within the cluster. Recall in Eq. (3), S^i_{skew} shall be partitioned in a balanced manner, and each partition $\widehat{S}^{ij}_{skew}$ will be sent to compute node j. To ensure that the tuples in S^i_{skew} are balanced after partitioning, we define a balance factor B^i as follows:

$$B^i = (\max_j |\widehat{S}^{ij}_{skew}| - \min_j |\widehat{S}^{ij}_{skew}|)/\max_j |\widehat{S}^{ij}_{skew}| \quad (11)$$

The balance factor measures the degree of balance in the data volume between partitions, *e.g.,* a smaller B^i implies more balanced partitions. When partitioning S^i_{skew} for balance, a threshold ϵ can be set to regulate the balance factor, that is, $B^i \leq \epsilon$.

Theorem 1. *In the case of balanced partitioning, if the balance factor $B^i \leq \epsilon$, it follows that the overall balance factor B between the compute nodes after redistribution will also satisfy $B \leq \epsilon$.*

Proof. The relationship between $\widehat{S}^{ij}_{skew}$ and $\widehat{S}^{j}_{skew}$ is:

$$|\widehat{S}^{j}_{skew}| = \sum_{i=0}^{n-1} |\widehat{S}^{ij}_{skew}| \tag{12}$$

Let $B_c = \epsilon$, transform Eq. (11) into the following inequality:

$$\max_j |\widehat{S}^{ij}_{skew}| - \min_j |\widehat{S}^{ij}_{skew}| \leq B_c \cdot \max_j |\widehat{S}^{ij}_{skew}| \tag{13}$$

Apply a summation transformation to the inequality:

$$\sum_{i=0}^{n-1}(\max_j |\widehat{S}^{ij}_{skew}| - \min_j |\widehat{S}^{ij}_{skew}|) \leq \sum_{i=0}^{n-1}(B_c \cdot \max_j |\widehat{S}^{ij}_{skew}|) \tag{14}$$

Rearrange the summation:

$$\max_j(\sum_{i=0}^{n-1} |\widehat{S}^{ij}_{skew}|) - \min_j(\sum_{i=0}^{n-1} |\widehat{S}^{ij}_{skew}|) \leq B_c \cdot \max_j(\sum_{i=0}^{n-1} |\widehat{S}^{ij}_{skew}|) \tag{15}$$

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

$$\frac{\max_j(\sum_{i=0}^{n-1} |\widehat{S}^{ij}_{skew}|) - \min_j(\sum_{i=0}^{n-1} |\widehat{S}^{ij}_{skew}|)}{\max_j(\sum_{i=0}^{n-1} |\widehat{S}^{ij}_{skew}|)} \leq B_c \tag{16}$$

Substitute this into the inequality using Eq. (12):

$$\frac{\max_j |\widehat{S}^{j}| - \min_j |\widehat{S}^{j}|}{\max_j |\widehat{S}^{j}|} \leq B_c \tag{17}$$

Combining this with Eq. (3), we obtain:

$$B \leq B_c \tag{18}$$

□

Therefore, to ensure the overall balance requirement in *BPPR Workflow*, *i.e.,* $B \leq \epsilon$, what we need to do is to make sure that each data node satisfies $B^i le \epsilon$. To achieve that, we present a new algorithm, namely Balanced Partition, shown in Algorithm 3. For each skewed tuple with join key $\boldsymbol{x}$, *Balanced Partition* assumes it would be dispatched to $C(t)$, where t is the last tuple with $t.b = x$. If $B^i \leq \epsilon$ is not satisfied, the node with the fewest tuples in $U(x)$ will be selected. Otherwise, if $C(t)$ is already the node with the fewest tuples, $U(x)$ will expand according to Algorithm 2.

In Algorithm 3, Lines 1-14 calculate the balance factor based on Eq. (11). The calculation of the balance factor has a linear time complexity. In practice, it can be further improved to achieve constant time complexity. As demonstrated in Line 27, each partition adds only one tuple at a time. Therefore, by keeping track of $maxVal$, $minVal$, and the size of each partition, we can efficiently update $maxVal$ and $minVal$ whenever a new tuple is added to a partition.

Lines 15-18 initialize an empty partition set. Lines 19-40 assign partitions for each tuple in S^i_{skew} to achieve a balanced distribution. Lines 22-24 dispatch the tuple with a newly identified skewed join key to the partition specified by the first element in the sequence. Lines 26-29 attempt to assign t to the partition previously selected for the tuple with the join key $\boldsymbol{x} = t.b$. Line 31 reselects candidate nodes from $U^i(x)$ if the balance factor $B^i > \epsilon$. If none of the nodes in $U^i(x)$ meet the requirement, Lines 33-34 expand the $U^i(x)$ to contain a new node. Finally, Line 40 returns the partition set *i.e., partition*.

The *Balanced Partition* algorithm improves the load balance in the cluster by partitioning the skewed tuples in a fine-grained manner. Additionally, the carefully designed algorithm to expand the node set minimizes the network overhead, further enhancing the performance of the distributed hash join. The effectiveness of the *Balanced Partition* algorithm does not depend on a specific distribution of tuples, allowing for the dynamic distribution of skewed tuples to achieve cluster balance, thus addressing the **Chal. ❶** and **Chal. ❸** listed in Section II-C.

Algorithm 1 Sequence Generation Algorithm

Input: join key $\boldsymbol{x}$ of the skewed tuple t , number of nodes $\boldsymbol{n}$
Output: $\boldsymbol{sequence}$

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

D. Algorithm Complexity

The BPPR algorithm consists of the *Balanced Partition* algorithm and the multicast mechanism. The former dispatches skewed tuples from the probe table to various nodes, while the latter replicates the tuples with the same join key in the build table to corresponding nodes. In a n-node cluster, skewed tuples on node i are denoted as S^i_{skew}, with the number of tuples represented as $m_p = |S^i_{skew}|$. The number of tuples from R_{Sskew} is denoted as m_b. Define $D(S^i_{skew})$ as the deduplication operation, where the number of distinct skewed join keys on each node after deduplication is denoted as $k = |D(S^i_{skew})|$.

Algorithm 2 Node Set Update Algorithm

Input: join key $\boldsymbol{x}$ of the skewed tuple t, node set $\boldsymbol{U^i(x)}$
Output: updated $\boldsymbol{U^i(x)}$

procedure UPDATEU($\boldsymbol{x}, \boldsymbol{U^i(x)}, \boldsymbol{sequence}$)
  $size \leftarrow \boldsymbol{U^i(x)}.size$
  $\boldsymbol{sequence} \leftarrow$ GENERATESEQUENCE($\boldsymbol{x}, \boldsymbol{n}$)
  if $size == \boldsymbol{sequence}.size$ **then**
    return $\boldsymbol{U^i(x)}$
  end if
  $\boldsymbol{U^i(x)}.add(\boldsymbol{sequence}[size])$
  return
end procedure

TABLE II: Time and Space Complexity of BPPR

	Balanced Partition	Multicast	BPPR
time	$O(m_p)$	$O(m_b \cdot n)$	$O(\max(m_p, m_b \cdot n))$
space	$O(k \cdot n)$	$O(1)$	$O(k \cdot n)$

In the *Balanced Partition* algorithm, skewed tuples are traversed with time complexity $O(m_p)$. The optimized computation of the balance factor has a constant time complexity $O(1)$, the same as expanding the set of nodes. Therefore, the time complexity of the *Balanced Partition* algorithm is $O(m_p)$. The algorithm *Balanced Partition* records the m tuples and $U^i(x)$ for each skewed join key $\boldsymbol{x}$, the size of which is at most n and at least 1. The sequence for each skewed join key is kept with the size n. Hence, the space complexity of the *Balanced Partition* algorithm is $O(m_p + k \cdot n)$. Since tuples are directly dispatched in practice, the space complexity is $O(k \cdot n)$.

For multicast, each tuple with the join key $\boldsymbol{x}$ in R_{Sskew} is replicated to $U^i(x)$. Then the time complexity is at least $O(m_b)$ and at most $O(m_b \cdot n)$. The space complexity is $O(1)$.

For each data node, the time and space complexity is listed in Table II.

IV. DISTRIBUTED DETECTOR

In this section, we start with a skew detector for data streams on a single node and then extend it to distributed settings, as a component of Bala-Join.

Any strategy to address data skew must first identify skewed join keys. For joins on raw tables, statistics about skewed join keys can be obtained easily. However, for join tasks over intermediate tables, skewed join keys cannot be directly identified from the statistics in the database. Therefore, it is necessary to deploy skew detectors on the nodes to identify and address data skew in real time. A distributed detector should meet the following requirements:

(1) Runtime detection on skewed join keys as per Eq. (1);

(2) Global consensus on the skewed join keys to partition the tuples on each data node into S^i_{non}, S^i_{skew}, R^i_{Snon}, and R^i_{Sskew};

(3) Limited time and space cost.

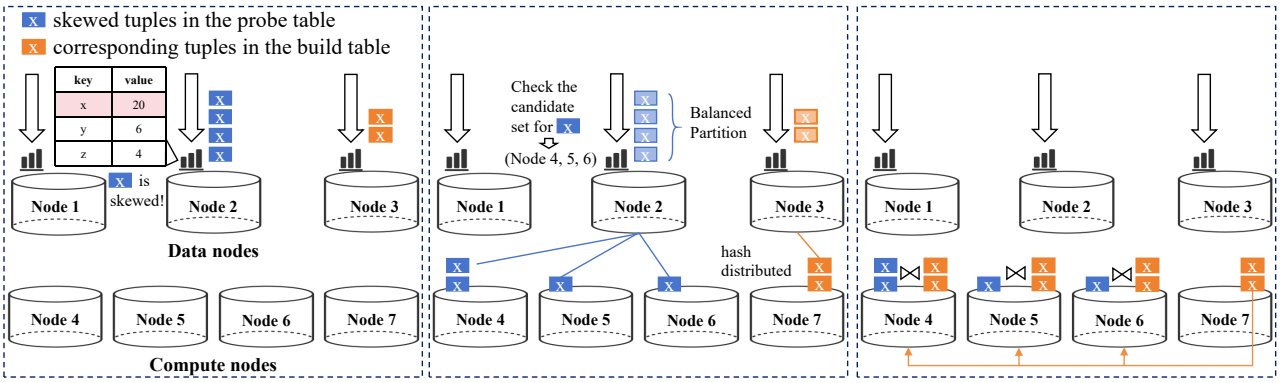
Based on the above considerations, we employ the Space Saving algorithm [6], a counter-based technique, as the building block of our detector. The core idea of the Space Saving algorithm is to maintain frequency information about a subset of the elements in the data stream with limited space. Specifically, it preserves the frequencies of the k elements using a list of k counters. Each counter is updated to estimate the frequency of the corresponding element. Lightweight data structures are used to sort these elements according to their

Algorithm 3 Balanced Partition Algorithm

Input: skewed tuples $\boldsymbol{S^i_{skew}}$, threshold $\boldsymbol{\epsilon}$, number of nodes $\boldsymbol{n}$
Output: set of the balanced subpartitions

procedure CALCBALANCEFACTOR($partition$)
  $maxVal \leftarrow partition[0].size$
  $minVal \leftarrow partition[0].size$
  for $subpartition$ in $partition$ **do**
    if $size > maxVal$ **then**
      $maxVal \leftarrow size$
    end if
    if $size < minVal$ **then**
      $minVal \leftarrow size$
    end if
  end for
  $balanceFactor \leftarrow (maxVal - minVal)\ /\ maxVal$
  return $balanceFactor$
end procedure

$partition \leftarrow []$
for $i = 0$ to $\boldsymbol{n} - 1$ **do**
  $partition.add([])$
end for
for t in $\boldsymbol{S^i_{skew}}$ **do**
  $\boldsymbol{x} \leftarrow t.b$
  $\boldsymbol{sequence} \leftarrow$ GENERATESEQUENCE($\boldsymbol{x}, \boldsymbol{n}$)
  if $U^i(t) = \emptyset$ **then**
    $U^i(t) \leftarrow \boldsymbol{sequence}[0]$
    $partition[ID].add(t)$
  else
    $lastID \leftarrow$ subpartition ID last selected by $\boldsymbol{x}$
    $partition[ID].add(t)$
    $balanceFactor \leftarrow$ CALCBALANCEFACTOR($partition$)
    $partition[lastID].delete(t)$
    if $balanceFactor > \boldsymbol{\epsilon}$ **then**
      $ID \leftarrow$ min_subpartition ID
      if $ID == lastID$ **then**
        UPDATEU($\boldsymbol{x}, U^i(t)$)
        $ID \leftarrow newID$
      end if
    end if
  end if
  $partition[selectID].add(t)$
end for
return $partition$



(a) Skew detection (b) Redistribution (c) Asynchronous Pulling

Fig. 4: ASAP mechanism

estimated frequencies.

Whenever a new element arrives, it tests whether it can be found in the list (the detector table in Fig. 4(a)); the corresponding counter is incremented if found successfully. Otherwise, if the list is not full, the element is added with a counter initialized to 1. When the list is full, the algorithm finds the element with the smallest counter, increases its counter, and replaces it with the new element.

The primary challenge of the Space Saving algorithm lies in efficiently updating elements and removing elements corresponding to the minimum count. Following the suggestion of [8], we adopt the Space Saving algorithm based on a hash table and an ordered array as the detector for the single-node environment, therefore achieving minimized time and space cost.

A. Distributed Skew Detector for BPPR

We are now moving forward to integrate the detector into a distributed environment. A straightforward approach [15], [16] is to perform local detection on each standalone node and aggregate the lists at each node to a global one, then redistribute the tuples based on the skewed join keys obtained. However, this approach poses three problems as follows:

Double Traversals of S: Each data node must traverse the table S^i twice (once for local detection and again for redistribution). In distributed databases, data nodes often receive tuples in the form of data streams, requiring a complete traversal to identify skewed join keys. Repeatedly traversing S^i can result in significant memory and I/O overhead.

Independent Standalone Detection: In a distributed environment, independent detection on each standalone node can result in varying execution times. Faster nodes may become idle after completing their tasks, while slower ones can hinder overall detection progress.

Communication Overhead for Consensus: Achieving a globally recognized list over all data nodes is essential. However, this requires unavoidable communication overhead between data nodes.

Therefore, implementing a distributed detector is not a trivial task. In response to the above problems, the distributed skew detector must be tightly integrated with the BPPR algorithm. As a solution, Bala-Join is employed with the distributed detector embedded into the redistribution and computation phases of BPPR. In the following, we will describe the Bala-Join redistribution and computation phases with the distributed skew detector. Different from Section III-C, which reports our general strategy in terms of an offline setting for ease of understanding, in the following we move on a step forward and present our detailed solution in a practical execution environment, where the intermediate tuples to be joined construct a data stream.

Redistribution: According to the BPPR algorithm, both S^i and R^i on the data node i are divided into four parts for redistribution. However, the partitioning process depends on the knowledge of the skewed join keys, making it impossible to complete for S^i and R^i in the form of data streams. Therefore, the way in which S^i and R^i are processed on a single data node differs from the basic strategy we reported in Section III-C.

S^i : Space Saving detector deployed on S^i can process each tuple one after another in the data stream, identifies them as skewed or non-skewed tuples.
Skewed tuples will be dispatched to a partition decided by Algorithm 3 (Lines 20-38). A signal will be sent to the node that the tuple is skewed.
Non-skewed tuples will be hash-distributed.
At this stage, the tuples in S^i are detected and distributed simultaneously. That is, a single traversal of the data stream achieves three critical objectives for S^i: detection of local data skew, partitioning of S^i, and redistribution of S^i.

R^i : Instead of sending R^i_{Snon} and R^i_{Sskew} separately in the basic BPPR version, all tuples from R^i will be hash partitioned and distributed. The partitioning for R^i into R^i_{Snon} and R^i_{Sskew} will be deferred to the computation phase.

Computation: In the online version of Bala-Join, each compute node is still waiting to receive these four partitions, but

in a different way compared to the basic version shown in Section III-C.

$\mathcal{S}^i_{hash}$: it now consists of tuples that are hash distributed from the S^i_{non} partitions across various data nodes.

$\mathcal{S}^i_{bal}$: it has the same source as the basic version. It contains skewed tuples detected in various data nodes, which are sent to node i based on balanced partitioning.

$\mathcal{R}^i_{hash}$: It now consists of hash-distributed tuples from R to node i. The union of $\mathcal{R}^i_{hash}$ from all compute nodes forms R, which is denoted as $R = \bigcup_{i=0}^{n-1} \mathcal{R}^i_{hash}$.

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

$$\mathcal{R}^i_{bal} = \{t' \mid t \in M^i,\ t.b = t'.a,\ t' \in R\} \tag{19}$$

Then the compute node can perform the local join result based on Eq. (8) after obtaining tuples from the four partitions.

The distributed detector provides real-time detection of skewed join keys, which addresses **Chal. ❷**. Its combination with BPPR, referred to as Bala-Join, extends BPPR to data streams, enhancing the performance of distributed hash joins across a wider range of scenarios.

B. ASAP Mechanism

To guarantee the correctness of HASH JOINs, data nodes across the cluster must reach two forms of consensus:

Consensus 1: A global consensus on skewed keys for data nodes to handle skewed tuples from S in a balanced manner.

Consensus 2: A global consensus on $U(x)$ for multicasting tuples with the join key $\boldsymbol{x}$ from R.

We achieve consensus through the **Active-Signaling and Asyn-chronous-Pulling (ASAP)** mechanism. Specifically, consider a skewed tuple t with the join key $\boldsymbol{x}$ from the probe table S. During the redistribution phase, the node receiving t is notified that the join key $\boldsymbol{x}$ is skewed, while all tuples from the build table R are hash distributed. In the computation phase, the node holding the tuple t asynchronously pulls all the tuples with the join key $\boldsymbol{x}$ from R located on node i, where $i = hash(x)$. Fig. 4 shows how the ASAP mechanism works. (a) Space Saving detectors deployed on each data node perform real-time assessments to identify whether incoming tuples from the probe table are skewed. A join key x is flagged as skewed when its observed frequency in the detector's counting structure exceeds a predetermined threshold (*e.g.,* 5% of total tuples). (b) Skewed tuples with the join key x from the probe table are distributed to a set of compute nodes following the BPPR algorithm. These compute nodes are notified that the tuples are skewed. Non-skewed tuples and build table entries with the join key x are hash-distributed to compute nodes. (c) Notified compute nodes asynchronously pull the tuples with the skewed join key x from the node to which they have been hash-distributed. Therefore, tuples with the join key x can be accurately joined on each compute nodes.

The ASAP mechanism ensures correctness when the join keys are accurately identified as skewed or non-skewed. Next, we discuss the correctness of the results when the skewed join key $\boldsymbol{x}$ is falsely recognized as non-skewed on some data nodes. According to the BPPR algorithm, if a data node determines that the join key $\boldsymbol{x}$ is not skewed, it computes $q = \text{hash}(x) \bmod n$ and hash-distributes $\boldsymbol{x}$ to node q without notifying node q that x is skewed. On other data nodes, if $\boldsymbol{x}$ is recognized as skewed, it will be balancedly distributed throughout the set of nodes $U(x)$, and these nodes will be informed that $\boldsymbol{x}$ is skewed. Importantly, the node set $U(x)$ must include node q, as determined by Algorithm 1. Therefore, the relevant nodes can still form a consensus.

The Bala-Join algorithm, equipped with ASAP, exhibits robust tolerance to local skew detection biases. Whether or not the skew is locally detected correctly at a single node, it affects only the *efficiency* of the algorithm, but not the *correctness*.

V. EVALUATION

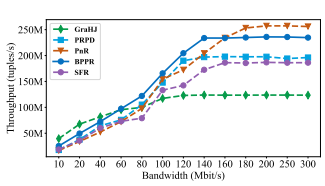
A. Experiment Setup

Our experimental setup was configured to faithfully replicate the real-world, cross-region topologies from our industrial projects with Inspur. This establishes a realistic testbed for evaluating performance over a WAN, allowing us to assess the solution's robustness by systematically varying the network bandwidth from congested to high-performance levels. We adopt $1 \sim 8$ Huawei cloud servers deployed across Beijing, Shanghai and Guiyang respectively, which together form a cluster of 3 to 24 nodes. Each node is equipped with a 4-core CPU, 8GB of RAM, a general-purpose SSD, and operates on the Ubuntu 18.04 operating system.

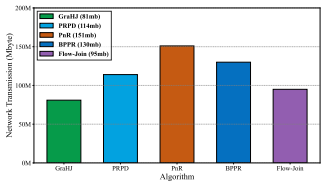
We evaluate the performance of various redistribution strategies specifically designed for skewed data, including GraHJ [14], PRPD [10], PnR [12], SFR [17], and BPPR. Afterward, we compare the performance of Bala-Join with other distributed hash join baselines that employ a skew detector, as listed below.

- GraHJ: A strategy that only uses hash partitioning
- PRPD: A strategy that keeps skewed tuples locally
- PnR: A strategy that distributes skewed tuples evenly
- SFR: A strategy that symmetrically replicates skewed tuples to nodes on rows and columns across the grid
- Flow-Join: A comprehensive solution with the SFR strategy and a detector based on the Space Saving algorithm

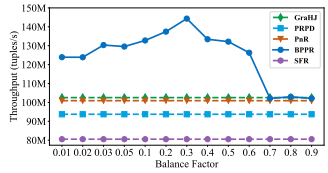
For experimental data, we use a synthetic dataset with Zipf [18] distribution, which is widely used in previous works [8], [16], [19]–[21]. The degree of skewness of the data exhibits a positive correlation with the Zipf factor z. In addition, we adopt the SSB-skew dataset [22] with the scale factor set to 10, which is a variation of the Star Schema Benchmark [23]. We focus on the join operations in the queries and execute them through the distributed hash join.



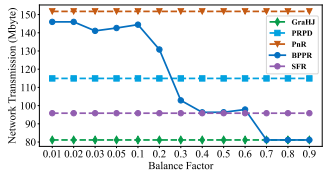
(a) Throughput vs. bandwidths



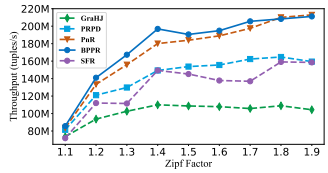
(b) Network overhead vs. bandwidths



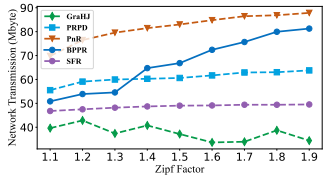
(c) Throughput vs. balance factor



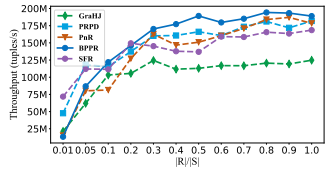
(d) Network overhead vs. balance factor



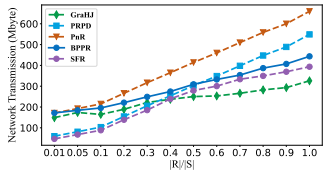
(e) Throughput vs. Zipf factor



(f) Network overhead vs. Zipf factor



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



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

Fig. 5: Comparison of distribution strategies in various scenarios

TABLE III: Average throughput and ranking of different algorithms over all experiments in Section V-B

	Bandwidth	Balance Factor	Zipf Factor	$\|R\|/\|S\|$	Comprehensive Ranking
BPPR	$165M$ (1)	$125M$ (1)	$178M$ (1)	$154M$ (1)	$622M$ (1)
PnR	$159M$ (2)	$101M$ (3)	$172M$ (2)	$137M$ (4)	$569M$ (2)
PRPD	$140M$ (3)	$94M$ (4)	$142M$ (3)	$148M$ (2)	$524M$ (3)
SFR	$127M$ (4)	$81M$ (5)	$131M$ (4)	$140M$ (3)	$470M$ (4)
GraHJ	$105M$ (5)	$103M$ (2)	$102M$ (5)	$103M$ (5)	$413M$ (5)

Performance is measured by throughput and network transmission. Throughput refers to the number of result tuples generated per second by the cluster, while network transmission represents the amount of data (in bytes) transmitted over the network. The former reflects the processing efficiency of the system, while the latter indicates the network overhead.

B. Redistribution Strategy

First, we conduct a series of experiments to evaluate the redistribution strategies. In these experiments, each algorithm is provided with the skew information directly. Default settings: Bandwidth = 100 Mbit/s, Balance Factor = 0.2, Zipf Factor = 1.25, and $|R|/|S|$ = 2/3.

Figure 5(a) and 5(b) show the performance of five strategies at different bandwidths of 10-300 Mbit/s. GraHJ has the lowest network transmission, while BPPR's transmission is 21% less than PnR but 14% more than PRPD and exceeds SFR. BPPR performs best at medium bandwidths and is competitive at high bandwidths, especially when compared to GraHJ.

Figure 5(c) and 5(d) explore the optimal value of the BPPR balance factor. As the balance factor increases, the throughput of BPPR increases in the beginning and achieves the highest value at a balance factor of 0.3. After that, the throughput drops and eventually becomes similar to GraHJ at 0.7.

Figure 5(e) and 5(f) compare algorithms under different Zipf factors, which control the degree of data skew. BPPR outperforms all other algorithms in the aspect of throughput and ranks 2^{nd} in the aspect of network overhead.

Figures 5(g) and 5(h) evaluate performance in different ratios $|R|/|S|$. BPPR remains stable, performing slightly worse than PRPD when $|R|/|S|$ is less than 0.1.

Table III summarizes the average throughput and rankings in the four sets of experiments. BPPR demonstrates greater adaptability compared to PRPD, SFR, and PnR, consistently achieving the highest rank in all experimental settings.

C. Distributed Skew Detector

In this section, we evaluate the performance of the distributed data skew detector proposed in Section IV. Since this strategy is tightly integrated with BPPR, forming Bala-Join, we compare it with two other schemes for fairness:

BPPR (without skewness detector): This version directly provides skewed values to the BPPR, allowing us to observe any overhead introduced by the distributed runtime detector.

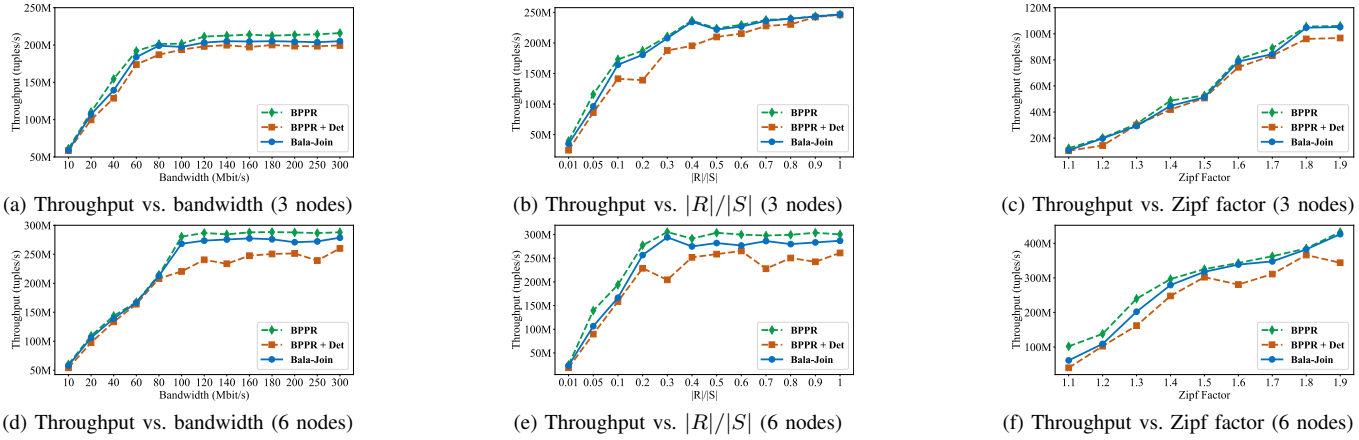
BPPR with an independent data skew detector (BPPR+Det): This employs the independent Space Saving detection algorithm [15] alongside BPPR.

By comparing these three schemes, we can evaluate the overhead introduced by the distributed detector. Since all systems use BPPR for skewed data processing, there is no difference in network overhead, and the comparison focuses solely on throughput. For clarity, we abbreviate the schemes as BPPR, BPPR+Det, and Bala-Join (the solution introduced in this paper) in subsequent experiments.

Figures 6(a) and 6(d) show the throughput of different strategies in 3-node and 6-node clusters across various bandwidths. In both cases, Bala-Join performs between BPPR and BPPR+Det. We also examine performance across different Zipf factors (Figures 6(c) and 6(f)) and $|R|/|S|$ ratios (Figures 6(b) and 6(e)).

In all experiments, the Bala-Join curve consistently falls between BPPR and BPPR+Det, indicating that the distributed detector introduces some overhead but performs significantly better than the independent Space-Saving detector used in BPPR+Det. This supports the challenges discussed in Section IV. On average, the proposed detector adds about 5% overhead compared to BPPR, with the 6-node setup generally incurring more overhead than the 3-node setup.

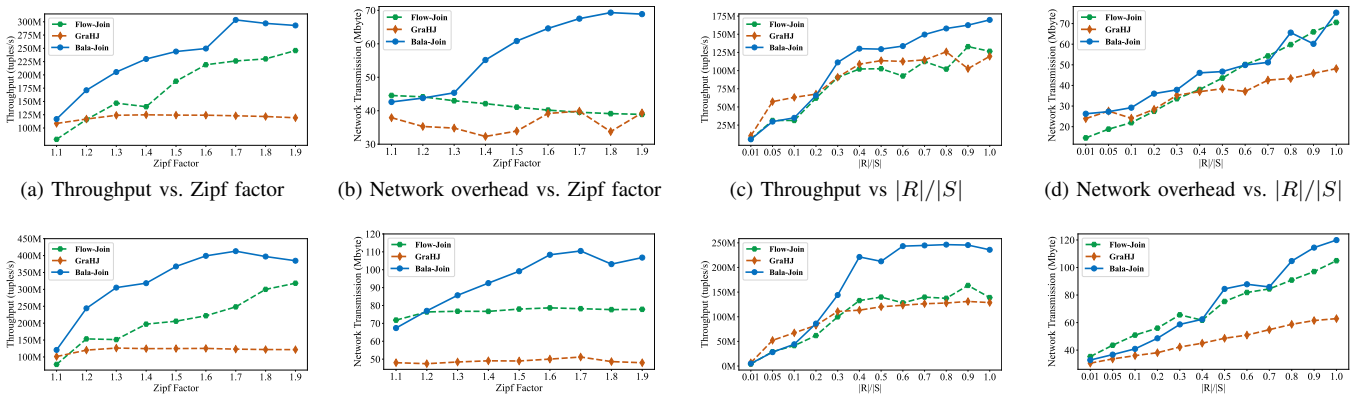
Table IV summarizes the average throughput per second for the three strategies across all scenarios.



(a) Throughput vs. bandwidth (3 nodes) (b) Throughput vs. $|R|/|S|$ (3 nodes) (c) Throughput vs. Zipf factor (3 nodes)

(d) Throughput vs. bandwidth (6 nodes) (e) Throughput vs. $|R|/|S|$ (6 nodes) (f) Throughput vs. Zipf factor (6 nodes)

Fig. 6: Cost of the runtime skew value detector



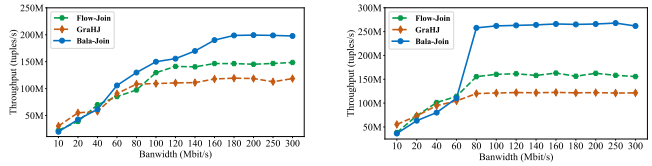
(a) Throughput vs. Zipf factor (b) Network overhead vs. Zipf factor (c) Throughput vs $|R|/|S|$ (d) Network overhead vs. $|R|/|S|$

(e) Throughput vs. Zipf factor (f) Network overhead vs. Zipf factor (g) Throughput vs $|R|/|S|$ (h) Network overhead vs. $|R|/|S|$

Fig. 7: Comparison of Dist-HJ solutions in different cluster settings: (a-d) 3 nodes; (e-h) 6 nodes

TABLE IV: Average throughput and comparison of different algorithms over all experiments in Section V-C

	Bala-Join	BPPR+Det	BPPR
Bandwidth (3 nodes)	$178M$	$172M(+433\%)$	$186M(-4\%)$
Zipf factor (3 nodes)	$59M$	$55M(+7\%)$	$61M(-8\%)$
$\|R\|/\|S\|$ (3 nodes)	$194M$	$179M(+8\%)$	$199M(-3\%)$
Bandwidth (6 nodes)	$221M$	$200M(+11\%)$	$230M(-4\%)$
Zipf factor (6 nodes)	$274M$	$240M(+14\%)$	$292M(-6\%)$
$\|R\|/\|S\|$ (6 nodes)	$235M$	$205M(+15\%)$	$253M(-7\%)$

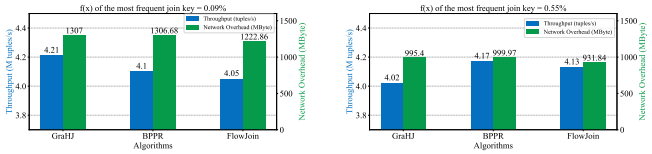


(a) Throughput vs. bandwidths (3 nodes) (b) Throughput vs. bandwidths (6 nodes)

Fig. 8: Throughput vs. bandwidth for Dist-HJ solutions

D. The overall Dist-HJ Solution

Finally, we evaluate the complete Dist-HJ solution, including the entire pipeline of data skew detection, redistribution, and computation. Figures 7 and 8 compare the performance of different solutions in 3-node and 6-node clusters, respectively. Table V summarizes the average throughput per second for each solution in various scenarios, together with the performance improvements of our solution compared to Flow-Join and GraHJ.



(a) LO_ORDERDATE = D_DATEKEY (b) LO_SUPPKEY = S_SUPPKEY

Fig. 9: Performance of Dist-HJ solutions on SSB-skew

Figure 9 presents the results for the top two join operations, which appear most frequently in the SSB-skew benchmark, evaluated on a 24-node cluster (similar trends were observed on other cluster scales).

Both join operations involve tables with varying degrees of data skew. For the join "LO_ORDERDATE = D_DATEKEY", the most frequent join key occurs at a rate of 0.09%, while for "LO_SUPPKEY = S_SUPPKEY", it occurs at 0.55%.

As the skew level increases, BPPR demonstrates superior performance compared to GraceHJ and Flow-Join. Notably, GraceHJ outperforms BPPR in case (a). This is because, in the SSB-skew benchmark, the same join key (LO_ORDERDATE) is clustered together. As a result, the detector may prematurely classify early-arriving tuples as skewed, leading to unnecessary partitioning. This insight is valuable for the practical deployment of data skew detection mechanisms.

In all experimental scenarios, our comprehensive solution consistently outperforms Flow-Join in terms of throughput. However, it does not have a clear advantage over Flow-Join in terms of network overhead. Furthermore, Flow-Join, with its SFR strategy for skewness processing, shows a trend of reducing network overhead as the number of nodes increases from 3 to 6. However, our results demonstrate that our solution strikes a better balance between the network and the computational performance. Despite the higher network overhead, our algorithm achieves significantly better execution efficiency.

TABLE V: Average throughput and comparison of different Dist-HJ implementations over all experiments in Section V-D

	Bala-Join	Flow-Join	GraHJ
Bandwidth (3 nodes)	$140M$	$112M(+25\%)$	$97M(+44\%)$
Zipf factor (3 nodes)	$235M$	$177M(+33\%)$	$121M(+94\%)$
$\|R\|/\|S\|$ (3 nodes)	$107M$	$83M(+29\%)$	$91M(+18\%)$
Bandwidth (6 nodes)	$205M$	$135M(+52\%)$	$109M(+88\%)$
Zipf factor (6 nodes)	$328M$	$208M(+58\%)$	$121M(+171\%)$
$\|R\|/\|S\|$ (6 nodes)	$163M$	$101M(+61\%)$	$99M(+65\%)$

VI. RELATED WORKS

A. Skewed data detection

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, they can be pre-processed by analyzing table or column statistics and sampling [9], [11], [19], [24]. The static solution cannot work when the HASH JOIN is used for intermediate results.

Dynamic detection typically occurs at runtime, with uncertain data sources and scales. Skewed value information can only be obtained based on historical data that have already been received. This issue can be categorized as the problem of frequent item-set 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 [25]–[27] obtain data samples by randomly sampling or taking regular interval samples from the data stream.

Counter-based algorithms [6], [26] use counters or counting structures to track the occurrences of different items.

Sketch-based algorithms [28], [29] use data sketches to estimate frequent items. Specifically, Count Sketch [28] uses a hash table and two matrices to estimate the frequency of occurrence of elements in a data stream. Count-Min Sketch [29] uses a hash table composed of multiple hash functions to perform estimation.

Several efforts have also been made in distributed dynamic detection. Parallel Spacing Saving [15] 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.

B. Joins over skewed data

Previous research [9] defines four types of data skew, namely the Tuple Placement Skew (TPS), the Selectivity Skew (SS), the Redistribution Skew (RS) and the Join Product Skew (JPS). In most 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. A well-performing hash function can be chosen to mitigate it, even if it does occur. 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 [9] or hash buckets [19], [30]–[35]) 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 [36] and MULTIFIT [37]) have been employed in such algorithms [9], [19], [30]–[34]. These algorithms propose different approaches based on distinct cost models. They estimate the JOIN cost using the number of tuples on each processing unit, with the aim of balancing the number of tuples from join tables on each processing unit to avoid Redistribution Skew (RS).

The second category of algorithms employs advanced models [9], [20], [31], [35], [38]–[40]. Instead of preestimating 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-crowded units. [9] introduces the Virtual Processor Scheduling (VPS) for this, using $|R| + |S| + |R \bowtie S|$ to gauge the join costs. [31] employs output and work functions for cost estimation. [35] considers systems with uneven processing unit configurations. [20] relies on shared virtual memory to shift tasks away from busy units. [38] observes the processing speeds of join tables. [39] offers single-stage and two-stage scheduling, the latter involving a central processor managing tasks. Fastjoin [40] periodically reallocates workloads to address stream processing imbalances.

The third category of algorithms takes a different approach by focusing on partitioning the join tables to balance computational loads. PRPD [10] assumes that the data is distributed randomly among 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 [8] refines PRPD by applying Symmetric Fragment Replicate (SFR) for skewed values in both join tables, which performs well in large high-speed networks. [11] offers Track Join, an algorithm that reduces network traffic by optimizing data transmission plans for each unique join value. [41] introduces FGSD, a fine-grained method to improve load balancing in MapReduce flows, thus reducing task times. Inspired by previous work in [8], [10], the PnR algorithm is presented in [12]. 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.

VII. CONCLUSION

The efficiency of Distributed Hash Join (Dist-HJ) in geo-distributed databases, such as CockroachDB , is severely impeded by data skew, particularly over Wide Area Networks (WANs), posing a critical industrial challenge. In this paper, we propose Bala-Join, an adaptive solution designed to mitigate this specific performance bottleneck. The core of our solution, the BPPR algorithm, employs a controlled multicast mechanism to judiciously balance computational load distribution against network overhead. This algorithm is tightly integrated with an online skew detector, which utilizes the Active-Signaling and Asynchronous-Pulling (ASAP) mechanism to facilitate efficient, real-time processing of data streams. Our empirical evaluation demonstrates that Bala-Join achieves a superior balance between network and computational performance, yielding throughput gains of 25%-61% and confirming its practical viability for real-world, geo-distributed applications.

REFERENCES

[1] M. Stonebraker, "The case for shared nothing," *IEEE Database Eng. Bull.*, vol. 9, no. 1, pp. 4–9, 1986.

[2] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss *et al.*, "Cockroachdb: The resilient geo-distributed sql database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1493–1509.

[3] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang *et al.*, "Tidb: a raft-based htap database," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3072–3084, 2020.

[4] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[5] Z. Yang, C. Yang, F. Han, M. Zhuang, B. Yang, Z. Yang, X. Cheng, Y. Zhao, W. Shi, H. Xi *et al.*, "Oceanbase: a 707 million tpmc distributed relational database system," *Proceedings of the VLDB Endowment*, vol. 15, no. 12, pp. 3385–3397, 2022.

[6] A. Metwally, D. Agrawal, and A. El Abbadi, "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, 2005, pp. 398–412.

[7] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann, "Rack-scale in-memory join processing using rdma," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 1463–1475.

[8] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann, "Flow-join: Adaptive skew handling for distributed joins over high-speed networks," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016, pp. 1194–1205.

[9] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1992.

[10] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, "Handling data skew in parallel joins in shared-nothing systems," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1043–1052.

[11] O. Polychroniou, R. Sen, and K. A. Ross, "Track join: distributed joins with minimal network traffic," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 2014, pp. 1483–1494.

[12] J. Yang, H. Li, Y. Si, H. Zhang, K. Zhao, K. Wei, W. Song, Y. Liu, and J. Cui, "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*, 2024.

[13] A. Pradhan, S. Karthik, and R. S, "Optimal query plans for geo-distributed data analytics at scale," in *Proceedings of the 7th Joint International Conference on Data Science & Management of Data (11th ACM IKDD CODS and 29th COMAD)*, 2024, pp. 247–251.

[14] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka, "Application of hash to data base machine and its architecture," *New Generation Computing*, vol. 1, no. 1, pp. 63–74, 1983.

[15] M. Cafaro, M. Pulimeno, and P. Tempesta, "A parallel space saving algorithm for frequent items and the hurwitz zeta distribution," *Information Sciences*, vol. 329, pp. 1–19, 2016.

[16] Q. Huang and P. P. Lee, "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, 2014, pp. 1420–1428.

[17] J. W. Stamos and H. C. Young, "A symmetric fragment and replicate algorithm for distributed joins," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 12, pp. 1345–1354, 1993.

[18] G. K. Zipf, *Human behavior and the principle of least effort: An introduction to human ecology*. Addison-Wesley, 1949.

[19] M. Kitsuregawa and Y. Ogawa, "Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (sdc)." in *VLDB*, 1990, pp. 210–221.

[20] A. Shatdal and J. F. Naughton, "Using shared virtual memory for parallel join processing," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, 1993, pp. 119–128.

[21] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann, "Locality-sensitive operators for parallel main-memory database clusters," in *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 2014, pp. 592–603.

[22] D. Justen, D. Ritter, C. Fraser, A. Lamb, N. Tran, A. Lee, T. Bodner, M. Y. Haddad, S. Zeuch, V. Markl *et al.*, "Polar: Adaptive and non-invasive join order selection via plans of least resistance."

[23] P. E. O'Neil, E. J. O'Neil, and X. Chen, "The star schema benchmark (ssb)," *Pat*, vol. 200, no. 0, p. 50, 2007.

[24] A. Vitorovic, M. Elseidy, and C. Koch, "Load balancing and skew resilience for parallel joins," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. Ieee, 2016, pp. 313–324.

[25] P. B. Gibbons and Y. Matias, "New sampling-based summary statistics for improving approximate query answers," in *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, 1998, pp. 331–342.

[26] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 2002, pp. 346–357.

[27] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Frequency estimation of internet packet streams with limited space," in *Esa*, vol. 2. Citeseer, 2002, pp. 348–360.

[28] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theoretical Computer Science*, vol. 312, no. 1, pp. 3–15, 2004.

[29] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[30] K. A. Hua and C. Lee, "Handling data skew in multiprocessor database computers using partition tuning." in *VLDB*, vol. 91, 1991, pp. 525–535.

[31] K. Alsabti and S. Ranka, "Skew-insensitive parallel algorithms for relational join," *Journal of King Saud University-Computer and Information Sciences*, vol. 13, pp. 79–110, 2001.

[32] J. L. Wolf, D. M. Dias, and P. S. Yu, "An effective algorithm for parallelizing hash joins in the presence of data skew," in *Proceedings. Seventh International Conference on Data Engineering*. IEEE Computer Society, 1991, pp. 200–201.

[33] ——, "A parallel sort merge join algorithm for managing data skew," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 70–86, 1993.

[34] J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek, "New algorithms for parallelizing relational database joins in the presence of data skew," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 6, pp. 990–997, 1994.

[35] H. M. Dewan, K. W. Mok, M. Hernández, and S. J. Stolfo, "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, 1994, pp. 40–49.

[36] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[37] E. G. Coffman, Jr, M. R. Garey, and D. S. Johnson, "An application of bin-packing to multiprocessor scheduling," *SIAM Journal on Computing*, vol. 7, no. 1, pp. 1–17, 1978.

[38] M. Kitsuregawa, "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, 1995, p. 246.

[39] X. Zhou and M. E. Orlowska, "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, 1995, pp. 527–536.

[40] S. Zhou, F. Zhang, H. Chen, H. Jin, and B. B. Zhou, "Fastjoin: A skewness-aware distributed stream join system," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 1042–1052.

[41] E. Gavagsaz, A. Rezaee, and H. Haj Seyyed Javadi, "Load balancing in join algorithms for skewed data in mapreduce systems," *The Journal of Supercomputing*, vol. 75, pp. 228–254, 2019.