Scaling Equi-Joins

Ahmed Metwally
ametwally@uber.com
Uber, Inc.
Sunnyvale, CA, USA

ABSTRACT

This paper proposes Adaptive-Multistage-Join (AM-Join) for scalable and fast equi-joins in distributed shared-nothing architectures. AM-Join utilizes (a) Tree-Join, a novel algorithm that scales well when the joined tables share hot keys, and (b) Broadcast-Join, the fastest-known when joining keys that are hot in only one table.

Unlike the state-of-the-art algorithms, AM-Join (a) holistically solves the join-key skew problem by achieving load balancing throughout the join execution, and (b) supports all outer-join variants without record deduplication or custom table partitioning. For the best AM-Join outer-join performance, we propose Index-Broadcast-Join (IB-Join) for Small-Large outer-joins, where one table fits in memory and the other is orders of magnitude larger. IB-Join improves on the state-of-the-art outer-join algorithms.

The proposed algorithms can be adopted in any shared-nothing architecture. We implemented a MapReduce version using Spark. Our evaluation shows the proposed algorithms execute significantly faster and scale to more skewed and orders-of-magnitude bigger tables when compared to the state-of-the-art algorithms.

CCS CONCEPTS

• **Information systems → Join algorithms**.

KEYWORDS

Big Data; Data skew; Distributed Algorithms; Load balancing

ACM Reference Format:
Ahmed Metwally. 2022. Scaling Equi-Joins. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526042

1 INTRODUCTION

Retrieval of information from two database tables is critical for data processing, and impacts the computational cost and the response time of queries. In the most general case, this operation entails carrying out a cross join of the two tables. The more common case is computing an equi-join, where two records in the two tables are joined if and only if equality holds between their join attributes. The algorithms for equi-joins have been optimized regularly since the inception of the database community [14, 25, 31, 32, 43, 63, 65].

Significant research has been done to enhance the sequential equi-join algorithms on multi-core processors [4, 8–10, 12, 16, 41, 64] and on GPUs [20, 36, 38, 40, 53, 54, 60, 66]. However, the proliferation of data collection and analysis poses a challenge to sequential join algorithms that are limited by the number of threads supported by the processing units. Scaling equi-joins had to progress through distributed architectures, as reviewed in § 3.

In addition to the industry-wide applications of database joins, this work is motivated by joining large skewed datasets, which is an integral operation in the similarity-based join algorithms [50] used for identity-fraud detection. The state-of-the-art algorithms failed to scale to our datasets. This motivated us to develop Adaptive-Multistage-Join (AM-Join). AM-Join is fast, efficient, scalable, and built using the basic MapReduce primitives, and is hence deployable in any distributed shared-nothing architecture. AM-Join utilizes Tree-Join, a novel algorithm that scales well by distributing the load of joining a key that is hot in both tables to multiple executors. Such keys are the scalability bottleneck of the state-of-the-art distributed algorithms. AM-Join also utilizes Broadcast-Joins to reduce the network load when joining keys that are hot in only one table.

AM-Join, presented in § 4, extends to all the outer-join variants elegantly without record deduplication or custom table partitioning, unlike the state-of-the-art industry-scale algorithms [19]. For the fastest execution of AM-Join outer-joins, we propose Index-Broadcast-Join (IB-Join) for Small-Large outer-joins, where one table fits in memory and the other is orders of magnitude larger. IB-Join improves on the state-of-the-art algorithms [24, 72].

While the proposed algorithms can be adopted in any shared-nothing architecture, we implemented a MapReduce version using Spark [76]. Our evaluation in § 5 highlights the improved performance and scalability of AM-Join. When compared to the sate-of-the-art algorithms [19, 33], AM-Join executed comparably fast on weakly-skewed synthetic tables and can join more-skewed or orders-of-magnitude bigger tables, including our real-data tables. The proposed Broadcast outer-join algorithm executed much faster than the ones in [24, 72]. We summarize our contributions in § 6.

2 FORMALIZATION

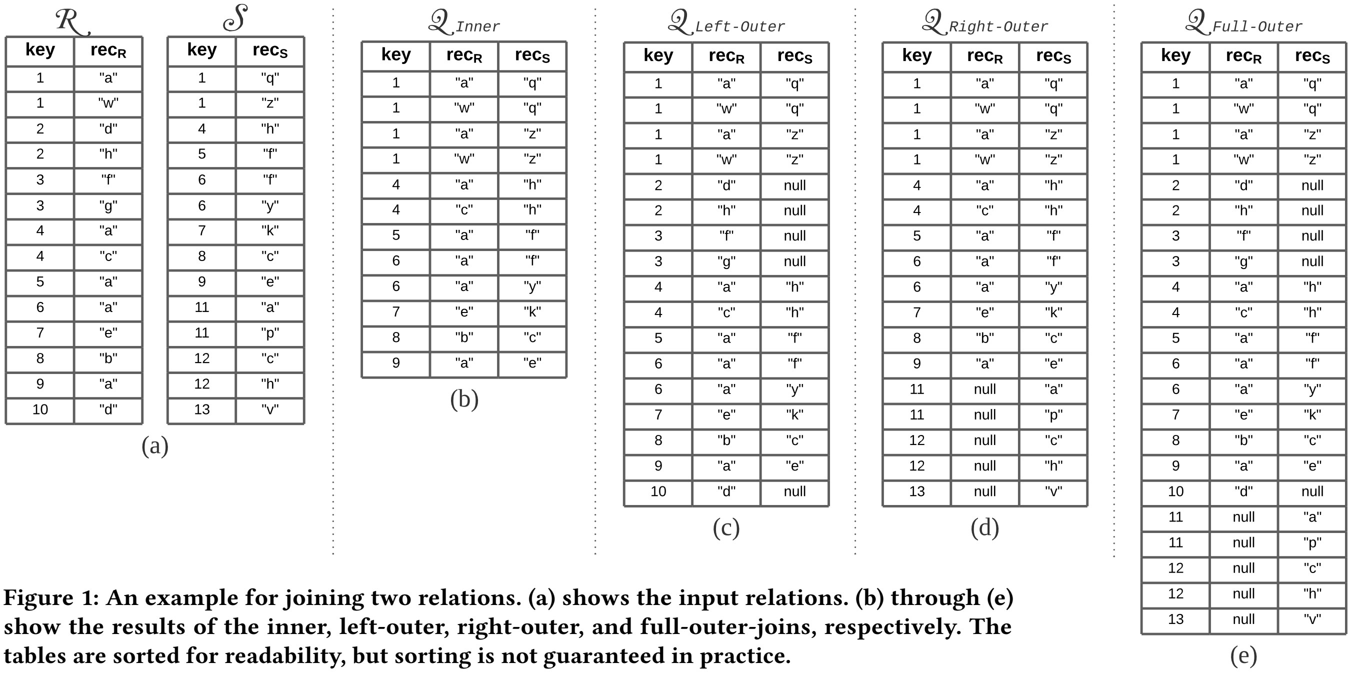
We now introduce the concepts used. Table. 1 shows the symbols used for the readers to refer to as they progress through the paper.

2.1 Equi-Joins and Variants

The equi-join operation combines columns from multiple tables, a.k.a. relations in the relational database community, based on the equality of the join attribute(s), a.k.a., join key(s). We focus on the case where two relations are joined. Other work, e.g., [1, 2, 26, 51], focused on extending equi-joins to multiple relations.

Given two relations $\mathcal{R}$ and $\mathcal{S}$, where $\mathcal{R} = (a, b)$ and $\mathcal{S} = (b, c)$, the equi-join, $\mathcal{R} \bowtie_b \mathcal{S}$, results in a new relation $Q = (a, b, c)$, where

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA.
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526042



$\mathcal{R}$

key	rec_R
1	"a"
1	"w"
2	"d"
2	"h"
3	"f"
3	"g"
4	"a"
4	"c"
5	"a"
6	"a"
7	"e"
8	"b"
9	"a"
10	"d"

$\mathcal{S}$

key	rec_S
1	"q"
1	"z"
4	"h"
5	"f"
6	"f"
6	"y"
7	"k"
8	"c"
9	"e"
11	"a"
11	"p"
12	"c"
12	"h"
13	"v"

(a)

$\mathcal{Q}_{\texttt{Inner}}$

key	rec_R	rec_S
1	"a"	"q"
1	"w"	"q"
1	"a"	"z"
1	"w"	"z"
4	"a"	"h"
4	"c"	"h"
5	"a"	"f"
6	"a"	"f"
6	"a"	"y"
7	"e"	"k"
8	"b"	"c"
9	"a"	"e"

(b)

$\mathcal{Q}_{\texttt{Left-Outer}}$

key	rec_R	rec_S
1	"a"	"q"
1	"w"	"q"
1	"a"	"z"
1	"w"	"z"
2	"d"	null
2	"h"	null
3	"f"	null
3	"g"	null
4	"a"	"h"
4	"c"	"h"
5	"a"	"f"
6	"a"	"f"
6	"a"	"y"
7	"e"	"k"
8	"b"	"c"
9	"a"	"e"
10	"d"	null

(c)

$\mathcal{Q}_{\texttt{Right-Outer}}$

key	rec_R	rec_S
1	"a"	"q"
1	"w"	"q"
1	"a"	"z"
1	"w"	"z"
4	"a"	"h"
4	"c"	"h"
5	"a"	"f"
6	"a"	"f"
6	"a"	"y"
7	"e"	"k"
8	"b"	"c"
9	"a"	"e"
11	null	"a"
11	null	"p"
12	null	"c"
12	null	"h"
13	null	"v"

(d)

$\mathcal{Q}_{\texttt{Full-Outer}}$

key	rec_R	rec_S
1	"a"	"q"
1	"w"	"q"
1	"a"	"z"
1	"w"	"z"
2	"d"	null
2	"h"	null
3	"f"	null
3	"g"	null
4	"a"	"h"
4	"c"	"h"
5	"a"	"f"
6	"a"	"f"
6	"a"	"y"
7	"e"	"k"
8	"b"	"c"
9	"a"	"e"
10	"d"	null
11	null	"a"
11	null	"p"
12	null	"c"
12	null	"h"
13	null	"v"

(e)

Figure 1: An example for joining two relations. (a) shows the input relations. (b) through (e) show the results of the inner, left-outer, right-outer, and full-outer-joins, respectively. The tables are sorted for readability, but sorting is not guaranteed in practice.

Symbol	Meaning
$\mathcal{R}, \mathcal{S}$	The joined relations/tables.
Q	The relation/table storing the join results.
b	The join keys/attributes.
ℓ	The number of records with a specific key in either $\mathcal{R}$ or $\mathcal{S}$.
ℓ_{max}	The maximum value of ℓ.
λ	The relative cost/runtime of sending data over the network vs. its IO from/to a local disk. $\lambda \geq 0$.
ℓ'	For a given key, the number of records a subsequent Tree-Join executor receives, where $\ell' = \ell^p$, for some p, s.t. $0 \leq p \leq 1$.
$\delta(\ell)$	The number of sub-lists produced by the splitter in Tree-Join for a list of length ℓ.
t	The number of iterations of Tree-Join.
$\kappa_{\mathcal{R}}, \kappa_{\mathcal{S}}$	The hot keys in $\mathcal{R}$ and $\mathcal{S}$.
$\lvert\kappa_{\mathcal{R}}\rvert_{max}, \lvert\kappa_{\mathcal{S}}\rvert_{max}.$	The maximum numbers of hot keys collected for $\mathcal{R}$ and $\mathcal{S}$.
M	The available memory per executor.
m_{id}	The size of a record identifier.
m_b	The average size of the join key, b.
$m_{\mathcal{R}}, m_{\mathcal{S}}$	The average size of records in $\mathcal{R}$ and $\mathcal{S}$.
$\Delta_{operation}$	The expected runtime of *operation.*
n	The number of executors.
e_i	A specific executor.
$\mathcal{R}_i, \mathcal{S}_i$	The partitions of $\mathcal{R}$ and $\mathcal{S}$ on e_i.

Table 1: The symbols used in the paper.

b is the join attribute(s), and a and c are the remaining attribute(s) from $\mathcal{R}$ and $\mathcal{S}$, respectively. Any pair of records from $\mathcal{R}$ and $\mathcal{S}$ whose b attribute(s) have equal values is output to Q.

In case a record in $\mathcal{R}$ or $\mathcal{S}$ is output to Q if and only if its b attribute(s) do have a match on the other side of the join, this is called an *inner-join*. If a record in $\mathcal{R}$ ($\mathcal{S}$) is output to Q whether or not it has a match on its b attribute(s) in $\mathcal{S}$ ($\mathcal{R}$), this is called a *left-outer-join* (*right-outer-join*). If any record in $\mathcal{R}$ or $\mathcal{S}$ is output to Q even if its b attribute(s) do not have a match on the other side of the join, this is called a *full-outer-join*. Fig. 1 shows an example of joining the two relations, $\mathcal{R}$ and $\mathcal{S}$, shown in Fig. 1(a). The results of the inner, left-outer, right-outer, and full-outer-joins are shown in Fig. 1(b) through Fig. 1(e), respectively.

2.2 Popular Distributed Frameworks

MapReduce [29] is a popular framework with built-in fault tolerance. It allows developers, with minimal effort, to scale data-processing algorithms to shared-nothing clusters, as long as the algorithms can be expressed in terms of the functional programing primitives *mapRec* and *reduceRecs.*

$mapRec : \langle key_1, value_1 \rangle \rightarrow \langle key_2, value_2 \rangle^*$

$reduceRecs : \langle key_2, value_2^* \rangle \rightarrow value_3^*$

The input dataset is processed using *executors* that are orchestrated by the *driver* machine. Each record in the dataset is represented as a tuple, $\langle key_1, value_1 \rangle$. Initially, the dataset is partitioned among the *mappers* that execute the map operation. Each mapper applies *mapRec* to each input record to produce a potentially-empty list of the form $\langle key_2, value_2 \rangle^*$. Then, the *shufflers* group the output of the mappers by the key. Next, each *reducer* is fed a tuple of the form $\langle key_2, value_2^* \rangle$, where $value_2^*$, the reduce_value_list, contains all the $value_2$'s that were output by any mapper with the key_2. Each reducer applies *reduceRecs* on the $\langle key_2, value_2^* \rangle$ tuple to produce a potentially-empty list of the form $value_3^*$. Any MapReduce job can be expressed as the lambda expression below.

$MapReduce(dataset)$:

$dataset.map(mapRec).groupByKey.reduce(reduceRecs)$

In addition to key_2, the mapper can optionally output tuples by a secondary key. The reduce_value_list would also be sorted by the secondary key in that case[1]. Partial reducing can happen at the mappers, which is known as *combining* to reduce the network load. For more flexibility, the MapReduce framework allows for loading external data when mapping or reducing. However, to preserve the determinism and the purity of the *mapRec* and *reduceRecs* functions, loading is restricted to the beginning of each operation.

The Spark framework [76] is currently the *de facto* industry standard for distributing data processing on shared-nothing clusters. Spark offers the functionality of MapReduce[2], as well as convenience utilities that are not part of MapReduce, but can be built using MapReduce. One example is performing tree-like aggregation of all the records in a dataset [44]. The *treeAggregate* operation can be implemented using a series of MapReduce stages, where each stage aggregates the records in a set of data partitions, and the aggregates are then aggregated further in the next stage, and so on, until the final aggregate is computed and sent to the driver.

Spark supports other functionalities that cannot be implemented using the map and reduce operations. One such functionality is doing hash-like lookups on the records across the network. These Spark-specific functionalities perform well under the assumption the dataset partitions fit in the memory of the executors.

3 RELATED WORK

We review the distributed algorithms that are generally applicable to equi-joins, and those targeting Broadcast outer-joins. We only review equi-joins in homogenous shared-nothing systems, where the processing powers of the executors are comparable. We neither review the algorithms devised for heterogenous architectures (e.g., [69]) nor in streaming systems (e.g., [28, 35, 46]), nor the approximate equi-join algorithms (e.g., [57]).

3.1 General Distributed Equi-Joins

The work most relevant to ours is that of distributing equi-joins on MapReduce [29] and on general shared-nothing [67] architectures.

3.1.1 MapReduce Equi-Joins. The seminal work in [17] explained *Map-Side* join (a.k.a., *Broadcast* or *Duplication* join in the distributed-architecture community), which is discussed in more details in § 4. This work also extended the Sort-Merge join [18] to the MapReduce framework. It also proposed using secondary keys to alleviate the bottleneck of loading the records from both joined relations in memory. This Sort-Merge join is considered a *Reduce-Side* join (a.k.a., *Shuffle* join in the distributed-architecture community). However, the work in [17] overlooks the skew in key popularity.

The SAND Join [7] extends the Reduce-Side joins by partitioning data across reducers based on a sample of data collected prior to partitioning. The work in [52] improves this key-range partitioning using quantile computation. It extends key-range partitioning to combinations of keys in the join results, and maps combinations of keys to specific reducers to achieve load-balancing. The algorithms in [33, 70] use sampling to build the cross–relation histogram, and balance the load between the executors based on the sizes of the relations and the estimated size of the join results.

There are major drawbacks with these key-range-division approaches [7, 33, 52, 70]. In the pre-join step, the key-range computation and communicated over the network to the mappers incurs significant computational [21] and communication cost [75]. The computational and network bottlenecks are more pronounced when computing key-ranges for combinations of keys [52][3]. These approaches implicitly assume that the keys are distributed evenly within each key range, which is not always the case for highly-diverse key spaces, highly-skewed data, or practical bucket sizes.

3.1.2 Shared-Nothing Equi-Joins. The algorithm in [37] is similar to that in [52] discussed above. It allows for grouping the join keys into evenly distributed groups that are executed on individual executors. However, it suffers from the same hefty overhead.

Map-Reduce-Merge [74] extends MapReduce with a *merge* function to facilitate expressing the join operation. Map-Join-Reduce [39] adds a join phase between the map and the reduce phases. These extensions cannot leverage the MapReduce built-in fault tolerance, and are implemented using complicated custom logic.

The Partial Redistribution & Partial Duplication (PRPD) algorithm in [73] is a hybrid between hash-based [30, 42] (as opposed to Sort-Merge [9, 13, 34]) and duplication-based join algorithms [31]. The PRPD algorithm collects $\kappa_{\mathcal{R}}$, and $\kappa_{\mathcal{S}}$, the high-frequency (a.k.a. hot) keys in $\mathcal{R}$ and $\mathcal{S}$, respectively. For the correctness of the algorithm, a key that is hot in both $\mathcal{R}$ and $\mathcal{S}$ is only assigned to either $\kappa_{\mathcal{R}}$ or to $\kappa_{\mathcal{S}}$. PRPD splits $\mathcal{S}$ into (a) $\mathcal{S}_{high}$ with keys in $\kappa_{\mathcal{S}}$, whose records are not distributed to the executors, (b) $\mathcal{S}_{dup}$ with keys in $\kappa_{\mathcal{R}}$, whose records are broadcasted to all the executors containing $\mathcal{R}_{high}$ records, and (c) $\mathcal{S}_{hash}$ the remaining records that are distributed among executors using hashing. $\mathcal{R}$ is split similarly into $\mathcal{R}_{high}$, $\mathcal{R}_{dup}$, and $\mathcal{R}_{hash}$. $\mathcal{S}_{hash}$ is joined with $\mathcal{R}_{hash}$ on the executors they were hashed into, and $\mathcal{S}_{high}$ ($\mathcal{R}_{high}$) is joined with $\mathcal{R}_{dup}$ ($\mathcal{S}_{dup}$) on the executors containing $\mathcal{S}_{high}$ ($\mathcal{R}_{high}$). The union of the three joins comprise the final join results.

Track-Join [55] aims at minimizing the network load using a framework similar to PRPD. For any given key, Track-Join migrates its records from one relation to a few executors [56], and *selectively broadcasts*, i.e., multicasts, the records from the other relation to these executors. This record migration process is expensive. Moreover, identifying the executors on which the join takes place is done in a preprocessing *track* phase, which is a separate distributed join that is sensitive to skew in key popularity.

The PRPD algorithm is developed further into the SkewJoin algorithm in [19]. Three flavors of SkewJoin are proposed. The Broadcast-SkewJoin (BSJ) is the same as PRPD. On the other end of the spectrum, Full-SkewJoin (FSJ) distributes $\mathcal{S}_{high}$ and $\mathcal{R}_{high}$ in a round-robin fashion on the executors using a specialized partitioner. This offers better load balancing at the cost of higher distribution overhead. The Hybrid-SkewJoin (HSJ) is a middle ground that retains the original data partitioning on the non-skewed side.

The PRPD algorithm is extended by [23] to use distributed lookup servers[4]. However, the algorithm in [23] splits $\mathcal{S}$ into $\mathcal{S}_{hash}$ and

[1] Secondary keys are not supported by the public version of MapReduce, Hadoop [6]. Two ways to support secondary keys were proposed in [47]. The first entails loading the entire reduce_value_list in the reducer memory, and the second entails rewriting the shuffler. The second solution is more scalable but incurs higher engineering cost.

[2] While Spark uses the same nomenclature of MapReduce, the MapReduce map function is called *flatMap* in Spark. We use the MapReduce notation, introduced in [29].

[3] This problem was the focus of [5] in the context of estimating join results size.

[4] Utilization lookup servers has been proposed before by the same research group [22].

$\mathcal{S}_{high}$. $\mathcal{S}_{hash}$, along with the distinct keys of $\mathcal{S}_{high}$, are hashed into the executors. All $\mathcal{R}$ is hashed into the executors. On each executor, e_i, $\mathcal{R}_i$ is joined with $\mathcal{S}_{hash_i}$, and with $\mathcal{S}_{high_i}$. The results of joining $\mathcal{R}_i$ with $\mathcal{S}_{hash_i}$ are reported as part of the final results. The results of joining $\mathcal{R}_i$ with $\mathcal{S}_{high_i}$ are augmented with the right $\mathcal{S}$ records using the distributed lookup servers.

Flow-Join [58] is similar to PRPD, but it detects the hot keys while performing the cross-network join. Flow-Join utilizes (a) *Remote Direct Memory Access* (RDMA) [45] over high-speed network [15, 59, 61] and (b) work-stealing across cores and Non-Uniform Memory Access (NUMA) [59] regions for local processing to perform well on rack-scale systems.

3.1.3 Comments on the General Distributed Equi-Joins. MapReduce is more restrictive than Spark that supports distributed lookups, for example. Spark is more restrictive than the general shared-nothing architecture that supports multicasting data, for example. An algorithm proposed for a more restrictive environment can be adopted for a less restrictive one, but the opposite is not true.

All the above algorithms that mix Broadcast-Joins and Shuffle-Joins [19, 23, 55, 56, 58, 73] do not extend smoothly to the outer-join variants. Mixing these two join techniques in the PRPD fashion results in having *dangling tuples* (tuples that do not join on the other side) of the same key distributed across the network. Deduplicating and counting dangling tuples across the network is a non-trivial operation, and was only discussed in [19, 58].

The strongest bottleneck in the Shuffle-Join (Hash or Sort-Merge) algorithms is they assume the Cartesian product of the records of any given key is computed using one executor. Hence, they fail to handle highly skewed data, where one key can bottleneck an entire executor. Consequently, all the other executors remain under-utilized while waiting for the bottleneck executor to finish [68].

On the other hand, the strongest drawback of all the algorithms that use Broadcast-Join is low scalability when there exists keys that are hot in both of the joined relations. In such cases, the broadcasted keys are hot, and their records may not fit in memory. Moreover, the computational load of joining hot keys is only balanced if the non-duplicated hot keys are evenly distributed among the executors.

3.2 Small-Large Outer-Joins

We now discuss the outer-joins when one relation can fit in memory and the other is orders of magnitude larger. Broadcast-Joins are the fastest-known for these *Small-Large join cases.* A solution to this problem is essential for extending AM-Join to outer-joins.

For a left-outer-join, the Duplication and Efficient Redistribution (DER) algorithm [72] broadcasts the small relation, $\mathcal{S}$, to all n executors. On each executor, e_i, $\mathcal{S}$ is inner joined with $\mathcal{R}_i$. The joined records are output, and the ids of the *unjoined* records of $\mathcal{S}$ are distributed to the n executors based on their hash. If an executor receives n copies of a record id, then this $\mathcal{S}$ id is *unjoinable* with $\mathcal{R}$ (since it was unjoined on the n executors). The unjoinable ids are inner Hash-Joined with $\mathcal{S}$ and *null*-padded. The left-outer-join result is the union of the two inner join results.

The Duplication and Direct Redistribution (DDR) algorithm [24] is similar to DER, but for the records failing the first inner join, each executor hashes the entire unjoined record, instead of only its id, over the network. Because the entire records, instead of the ids, are hash-distributed, (a) the first join can be executed as an out-of-the-box left-outer-join, and (b) the need for the second inner join is obviated. Extending DER and DDR to full-outer-joins is achieved by changing the first join to output the unjoined records on $\mathcal{R}_i$.

4 THE AM-JOIN ALGORITHM

AM-Join can be implemented with the basic MapReduce primitives, and hence can be implemented on any shared-nothing architecture. We first propose Tree-Join, a novel algorithm that scales well even in the existence of keys that are hot in both of the joined relations (§ 4.1). Then, we discuss Small-Large joins. We devise Index-Broadcast-Join (IB-Join) and show analytically it is more efficient than the state-of-the-art algorithms [24, 72] (§ 4.2). The Tree-Join, and the Broadcast-Join algorithms are the building blocks of AM-Join (§ 4.3). AM-Join achieves (a) high scalability by utilizing Tree-Join that distributes the load of joining a key that is hot in both relations to multiple executors, and (b) fast execution by utilizing the Broadcast-Join algorithms that reduce the network load when joining keys that are hot in only one relation.

4.1 The Tree-Join Algorithm

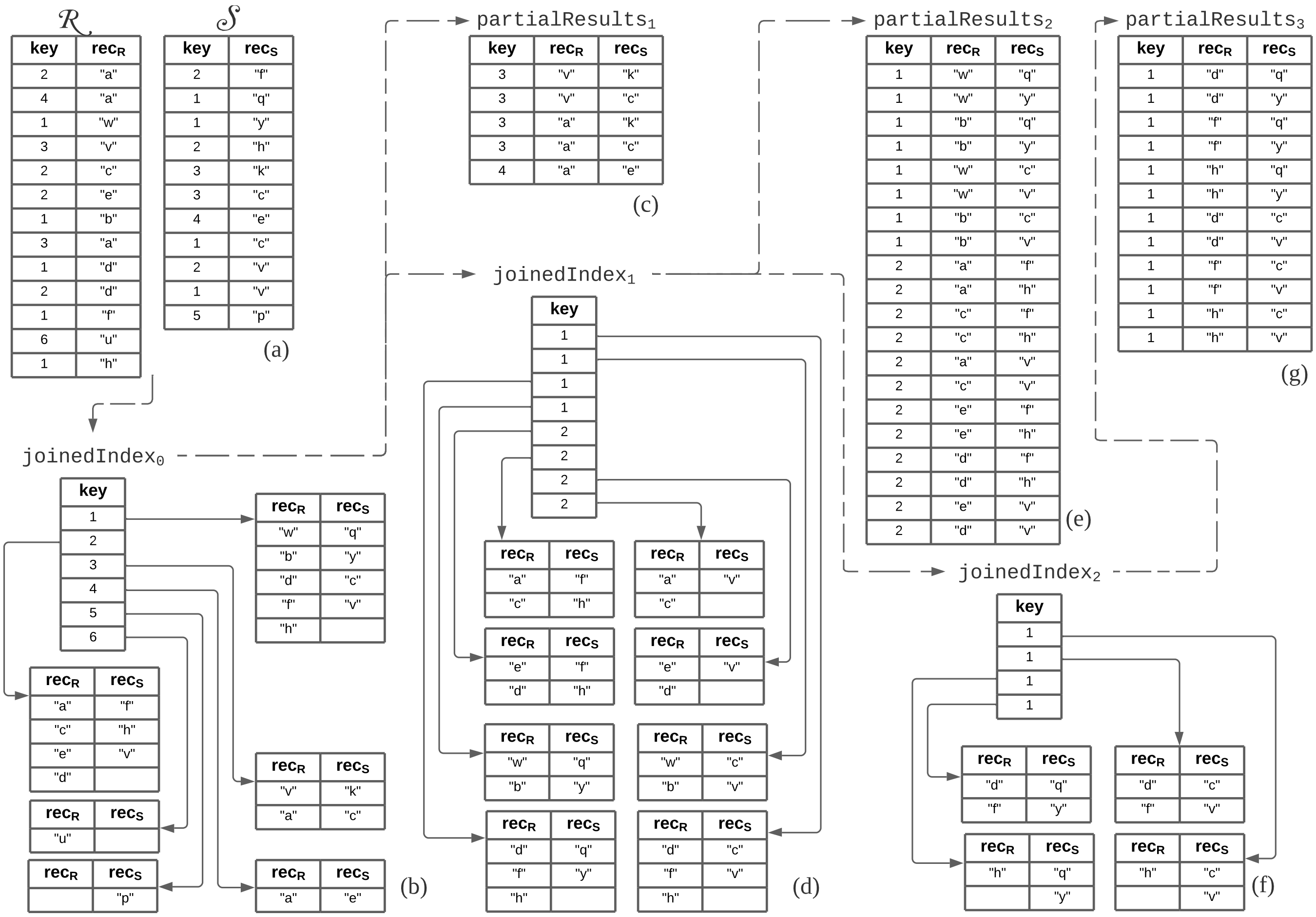
Tree-Join can execute over multiple stages, each corresponding to a MapReduce Job. Tree-Join scales well when joining relations sharing hot keys. If a Shuffle-Join (Hash or Sort-Merge) is executed, a bottleneck happens because a single executor has to process and output a disproportionately large number of pairs of records that have the same hot key. Tree-Join, on the other hand, alleviates this bottleneck by utilizing the idle executors without doing any duplicate work, while adding minimal overhead.

Algorithm 1 *treeJoin*($\mathcal{R}$, $\mathcal{S}$)

```
Input: Two relations to be joined.
Output: The join results.
$joined\_index = buildJoinedIndex(\mathcal{R}, \mathcal{S})$
$Q$ = empty Dataset
while $joined\_index$.nonEmpty do
    $\langle partial\_join, new\_index \rangle = treeJoinIteration(joined\_index)$
    $Q = Q \cup partial\_join$
    $joined\_index = new\_index$.randomShuffle
end while
Return $Q$
```

The inner variant of Tree-Join (Alg. 1) is explained below. The outer-join variants are straightforward extensions. The algorithm starts by building a distributed *joined index* (Alg. 2). This is done by hashing the two relations to the executors using the same hash function. Then, each executor groups the relations by the join keys.

Each key in the joined index has two *joined lists* of records. The keys are joined independently, and the union of their join results constitutes the join results of $\mathcal{R}$ and $\mathcal{S}$. This *key-independence* observation simplifies the analysis. The two joined lists of each key are joined in stages, i.e., iteratively. In each iteration (Alg. 3), if the joined lists are short enough (based on the analysis in § 4.1.2), the join results of this key are obtained by outputting the key with all pairs of records from both relations. However, if the record lists are long, i.e., the key is hot, each of the two lists is chunked into sub-lists, and the executor outputs all pairs of sub-lists. This dataset of pairs of sub-lists constitutes the joined index to be processed in the next iteration. In the next iteration, the keys and their pairs of joined lists of the new joined index are assigned to random partitions to distribute the load among the executors.



(a) ℛ

key	recR
2	"a"
4	"a"
1	"w"
3	"v"
2	"c"
2	"e"
1	"b"
3	"a"
1	"d"
2	"d"
1	"f"
6	"u"
1	"h"

(a) 𝒮

key	recS
2	"f"
1	"q"
1	"y"
2	"h"
3	"k"
3	"c"
4	"e"
1	"c"
2	"v"
1	"v"
5	"p"

(b) joinedIndex₀

key
1
2
3
4
5
6

recR	recS
"w"	"q"
"b"	"y"
"d"	"c"
"f"	"v"
"h"	

recR	recS
"a"	"f"
"c"	"h"
"e"	"v"
"d"	

recR	recS
"v"	"k"
"a"	"c"

recR	recS
"a"	"e"

recR	recS
	"p"

recR	recS
"u"	

(c) partialResults₁

key	recR	recS
3	"v"	"k"
3	"v"	"c"
3	"a"	"k"
3	"a"	"c"
4	"a"	"e"

(d) joinedIndex₁

key
1
1
1
1
2
2
2
2

recR	recS
"a"	"f"
"c"	"h"

recR	recS
"a"	"v"
"c"	

recR	recS
"e"	"f"
"d"	"h"

recR	recS
"e"	"v"
"d"	

recR	recS
"w"	"q"
"b"	"y"

recR	recS
"w"	"c"
"b"	"v"

recR	recS
"d"	"q"
"f"	"y"
"h"	

recR	recS
"d"	"c"
"f"	"v"
"h"	

(e) partialResults₂

key	recR	recS
1	"w"	"q"
1	"w"	"y"
1	"b"	"q"
1	"b"	"y"
1	"w"	"c"
1	"w"	"v"
1	"b"	"c"
1	"b"	"v"
2	"a"	"f"
2	"a"	"h"
2	"c"	"f"
2	"c"	"h"
2	"a"	"v"
2	"c"	"v"
2	"e"	"f"
2	"e"	"h"
2	"d"	"f"
2	"d"	"h"
2	"e"	"v"
2	"d"	"v"

(f) joinedIndex₂

key
1
1
1
1

recR	recS
"d"	"q"
"f"	"y"

recR	recS
"d"	"c"
"f"	"v"

recR	recS
"h"	"q"
	"y"

recR	recS
"h"	"c"
	"v"

(g) partialResults₃

key	recR	recS
1	"d"	"q"
1	"d"	"y"
1	"f"	"q"
1	"f"	"y"
1	"h"	"q"
1	"h"	"y"
1	"d"	"c"
1	"d"	"v"
1	"f"	"c"
1	"f"	"v"
1	"h"	"c"
1	"h"	"v"

Figure 2: An example of inner-joining two relations using Tree-Join. (a) The input relations. (b) The initial index built from the input relations in (a). The initial index is used to produce (c) the first partial results, and (d) the first joined index. The first joined index is used to produce (e) the second partial results, and (f) the second joined index. The second joined index is used to produce (g) the third partial results. The union of the partial results in (c), (e), and (g) constitutes the inner-join results.

Fig. 2 shows an example of Tree-Join. The *buildJoinedIndex* algorithm (Alg. 2) is invoked to build the initial joined index, $joinedIndex_0$ (Fig. 2(b)), on the input relations in Fig. 2(a). Notice that all the keys in this initial index are distinct. Assuming that any key whose joined lists has 2 or fewer records in both joined relations is not considered hot, the joined lists of keys 3 and 4 are used to produce the first partial results, $partialResults_1$ (Fig. 2(c)) and the joined lists of keys 1 and 2 are chunked into the first joined index, $joinedIndex_1$ (Fig. 2(d)), and the joined lists of keys 5 and 6 are discarded, since they have records in only one relation.

Algorithm 2 *buildJoinedIndex*($\mathcal{R}$, $\mathcal{S}$)

```
Input: Two relations to be joined.
Output: A joined index mapping
        each join key to the pair of lists of records.
$keyed_r = \mathcal{R}$.map(map$_{buildJoinedIndex}$)
$keyed_s = \mathcal{S}$.map(map$_{buildJoinedIndex}$)
$joined\_index = keyed_r$
    .union($keyed_s$)
    .groupByKey
    .reduce(reduce$_{buildJoinedIndex}$)
Return $joined\_index$
```

map$_{buildJoinedIndex}$:

$$r_i \longrightarrow \langle key_{r_i}, \langle 0, r_i \rangle \rangle$$

$$s_j \longrightarrow \langle key_{s_j}, \langle 1, s_j \rangle \rangle$$

reduce$_{buildJoinedIndex}$:

$$\langle key, (\langle 0, r_i \rangle \textbf{ OR } \langle 1, s_j \rangle)^* \rangle \longrightarrow \langle key, r_i^*, s_j^* \rangle$$

Notice that the keys in $joinedIndex_1$ are repeated. Line 6 in Alg. 1 distributes these keys randomly among the executors for load balancing. *treeJoinIteration* (Alg. 3) uses the keys that have 2 or fewer records in both joined relations in $joinedIndex_1$ to produce the second partial results, $partialResults_2$ (Fig. 2(e)) and uses the remaining keys to produce the second joined index, $joinedIndex_2$ (Fig. 2(f)). While all the keys in $joinedIndex_2$ have the same value, they are distributed among the executors for load balancing. All the keys in $joinedIndex_2$ have 2 or fewer records in both joined relations. Hence, no more iterations are executed, and $joinedIndex_2$ is used to produce the third and final partial result, $partialResults_3$ (Fig. 2(g)). The union of the three partial results constitutes the inner-join results of the input relations in Fig. 2(a).

Algorithm 3 *treeJoinIteration*($joined\_index$)

```
Input: A joined index of two relations to be joined.
Output: Join results of some keys,
        and the joined index (of the remaining keys)
        for next iteration.
⟨$cold\_index, hot\_index$⟩ = $joined\_index$.split($isHotKey$)
$partial\_join$ = $cold\_index$.map(map$_{getAllValuePairs}$)
$new\_index$ = $hot\_index$
    .map(map$_{chunkPairOfLists}$)
    .map(map$_{getAllValuePairs}$)
Return ⟨$partial\_join, new\_index$⟩
```

Algorithm 4 $map_{getAllValuePairs}(\langle key, \mathcal{L}_1, \mathcal{L}_2\rangle)$

```
Input: A tuple of the join key, and two lists.
Output: A list of triplets,
        each triplet has $key$ and a pair from $\mathcal{L}_1$ and $\mathcal{L}_2$.
        All pairs from $\mathcal{L}_1$ and $\mathcal{L}_2$ are output.
u = empty Buffer
for all $v_i \in \mathcal{L}_1, v_j \in \mathcal{L}_2$ do
    u.append(⟨$key, v_i, v_j$⟩)
end for
Return u
```

We next analyze Tree-Join. We discuss chunking of lists for optimal load balancing in § 4.1.1, identifying hot keys in § 4.1.2, and establish the stages, i.e., the iterations, are very limited in number in § 4.1.3. For simplicity, we assume both lists of a joined key have the same length, ℓ, and λ is the relative cost of sending data over the network vs. its IO from/to a local disk, and $\lambda > 0$. While the analysis ignores the overhead of allocating the executors for the stages, it of course considers the processing cost in each stage.

$\text{map}_{chunkPairOfLists}$:
$\langle key, \mathcal{L}_1, \mathcal{L}_2\rangle \rightarrow \langle key, chunkList(\mathcal{L}_1), chunkList(\mathcal{L}_2)\rangle$

4.1.1 Chunking Lists of Hot Keys.

Computing the optimal number of sub-lists for a list of records is crucial for load balancing between the executors and across iterations. Chunking a pair of joined lists into numerous pairs of sub-lists adds overhead to the *splitter*, the executor assigned the original pair of lists. Conversely, chunking the pair of joined lists into few pairs of sub-lists adds overhead to the executors that handle these pairs in the subsequent iteration.

We choose to balance the load between the *splitter* and the subsequent executors. Ignoring the key size, for each of the joined lists, the splitter produces $\delta(\ell) = \ell/\ell^p = \ell^{1-p}$ sub-lists, each is of size $\ell' = \ell^p$, for some p, where $0 \leq p \leq 1$. The splitter outputs $\Theta((\ell^{1-p})^2)$ pairs of sub-lists. Hence, the output of this executor is $\Theta(\ell^{2-2p})$ pairs of sub-lists, each containing ℓ' records. Therefore, the splitter work is $\Theta(\ell^{2-p})$ records. Each subsequent executor receiving a pair of sub-lists outputs $\Theta((\ell')^2)$ pairs of records. Hence, the output of each subsequent executor is $\Theta(\ell^{2p})$ records. To achieve decent load balancing between the splitter and each subsequent executor, the loads should be comparable, yielding $2 - p = 2p$.

$$p = \frac{2}{3} \tag{1}$$

From Eqn. 1, to achieve the load-balancing goal, $\delta(\ell)$, the number of sub-lists is $\sqrt[3]{\ell}$, and each has $\ell^{\frac{2}{3}}$ records.

To illustrate using an example, assuming two joined lists each has 10^5 records. A naïve Hash-Join would output ${10^5}^2$ pairs of records, i.e., $2{\times}10^{10}$ IO, using one executor. Meanwhile, the Tree-Join splitter would chunk each list into 47 sub-lists, and would output $47 \times 47 \times (2127 + 2127)$ records, which is $\approx 9.4 \times 10^6$ IO. Each one among the 2209 subsequent executors would output $2174 \times 2174 \times 2 \approx 9.5 \times 10^6$ IO. Notice the only overhead for utilizing 2209 executors to join this key is producing $\approx 9.4 \times 10^6$ IO by the splitter, and sending this data over the network. Since $\frac{9.4{\times}10^6}{2{\times}10^{10}} < 0.05\%$, this overhead is $\approx (1 + \lambda) \times 0.05\%$. If the lists instead had 10^4 (10^3) records, the load can be distributed on 441 (100) workers, and the overhead would only increase to $(1 + \lambda) \times 0.2\%$ ($(1 + \lambda) \times 1\%$).

Algorithm 5 *chunkList*($\mathcal{L}$)

```
Input: A list of records.
Output: A list of sub-lists of records.
$\ell = |\mathcal{L}|$ // The length of $\mathcal{L}$.
$\delta(\ell) = \left\lceil \sqrt[3]{\ell} \right\rceil$ // The number of sub-lists.
$\ell' = \left\lfloor \frac{\ell}{\delta(\ell)} \right\rfloor$ // The length of a sub-list.
Return $[\mathcal{L}_0 \dots \mathcal{L}_{\ell'-1}], [\mathcal{L}_{\ell'} \dots \mathcal{L}_{2\times\ell'-1}], \dots, [\mathcal{L}_{(\delta(\ell)-1)\times\ell'} \cdots \mathcal{L}_C]$
```

4.1.2 Defining Hot Keys.

From the key-independence observation, keys can be considered in isolation. If the joined lists are not chunked, the time to produce the join results on 1 executor using Hash-Join is $\Delta_{ShuffleJoin} \approx \Theta(\ell^2)$. Meanwhile, the end-to-end Tree-Join processing time, $\Delta_{treeJoin}$, is the time taken by the splitter, the time to randomly shuffle the pairs of sub-lists over the network, and the time taken by the $\left\lceil \ell^{\frac{2}{3}} \right\rceil$ subsequent executors in the next iteration. A key is considered hot and is worth chunking if $\Delta_{ShuffleJoin} > \Delta_{treeJoin}$ using at least a single Tree-Join stage. Given n available executors, this condition is expressed in Rel. 2.

$$\ell^2 > \left((1+\lambda) + \left\lceil \frac{\ell^{\frac{2}{3}}}{n} \right\rceil\right) \times \ell^{\frac{2}{3}} \tag{2}$$

One (non-tight) value for ℓ that provably satisfies Ineq. 2 for any $\lambda > 0$ and any $n \geq 1$, is $\left(1 + \sqrt{2+\lambda}\right)^{\frac{3}{2}}$. This value is used in the rest of the paper to define the minimum frequency for a key to be hot.

Algorithm 6 *isHotKey*($\langle key, \mathcal{L}_1, \mathcal{L}_2\rangle$)

```
Input: A tuple of the join key, and two joined lists of records.
Output: Whether the joined lists should be chunked.
Constant: $\lambda$ the ratio of network to disk costs.
$\ell_1 = |\mathcal{L}|$ // The length of $\mathcal{L}_1$.
$\ell_2 = |\mathcal{L}|$ // The length of $\mathcal{L}_2$.
$\ell = \sqrt{\ell_1 \times \ell_2}$ // The effective length if $\ell_1 = \ell_2$.
Return $\ell > \left(1 + \sqrt{2+\lambda}\right)^{\frac{3}{2}}$
```

4.1.3 The Number of Iterations.

Since the overhead of allocating the executors for a stage was ignored, we make the case the number of iterations in this multistage join is very small.

From the key-independence observation, the join is concluded when the results for the last key are computed. Assuming all other factors are equal, the last key to compute has the longest pair of joined lists, ℓ_{max}. After the first stage, each of the subsequent executors of the next iteration inherits a pair of lists whose lengths are $\ell_{max}^{\frac{2}{3}}$. This chunking continues for t times as long as the lengths exceed $\left(1 + \sqrt{2+\lambda}\right)^{\frac{3}{2}}$. Hence, the following inequality holds for t.

$$\left(1 + \sqrt{2+\lambda}\right)^{\frac{3}{2}} < \ell_{max}^{\overbrace{\frac{2}{3}^{\frac{2}{3}\cdots\frac{2}{3}}}^{t \text{ times}}}$$

Hence, $\left(1+\sqrt{2+\lambda}\right)^{\frac{3}{2}} < \ell_{max}^{(\frac{2}{3})^t}$. Raising both sides to the $(\frac{3}{2})^t$ power yields $\left(1+\sqrt{2+\lambda}\right)^{(\frac{3}{2})^{t+1}} < \ell_{max}$. Taking the log with base $1+\sqrt{2+\lambda}$ on both sides yields the following.

$$\left(\frac{3}{2}\right)^{t+1} < \log_{1+\sqrt{2+\lambda}}(\ell_{max})$$

Taking the log with base $\frac{3}{2}$ yields Rel. 3.

$$t < \log_{\frac{3}{2}}\left(\log_{1+\sqrt{2+\lambda}}(\ell_{max})\right) - 1 \tag{3}$$

From Ineq. 3, the number of iterations is $O(\log(\log(\ell_{max})))$, which grows very slowly with ℓ_{max}.

4.2 The Small-Large Outer-Join Algorithms

Assuming $\mathcal{R} \gg \mathcal{S}$, an index is built for the small relation, $\mathcal{S}$, the driver collects the index locally, and broadcasts it to all n executors performing the join (Alg. 7). Broadcasting the index can be done in time that is logarithmic in n [11, 62], and can hence be faster than shuffling the large relation, $\mathcal{R}$, across the network.

map$_{buildIndex}$:

$rec_i \longrightarrow \langle key_{rec_i}, rec_i \rangle$

Algorithm 7 *broadcastJoin*($\mathcal{R}$, $\mathcal{S}$)

```
Input: Two relations to be joined.
Output: The join results.
Assumes: $\mathcal{S}$ fits in memory. $\mathcal{R} \gg \mathcal{S}$.
index = $\mathcal{S}$.map(map$_{buildIndex}$).groupByKey
broadcast index to all executors
$Q$ = $\mathcal{R}$.map(map$_{broadcastJoin}$)
Return $Q$
```

Assuming each executor, e_i, joins can accommodate the index of $\mathcal{S}$ in memory. For each record in its local partition, $\mathcal{R}_i$, the executor extracts the join key, probes the local index of $\mathcal{S}$ with that key, and produces the join results. The local join is shown in Alg. 8.

Algorithm 8 $map_{broadcastJoin}(rec_i)$

```
Local: index, a map from each join key in the replicated relation
       to all the records with that key.
Input: A record from the non-replicated relation.
Output: A list of triplets representing the join of $rec_i$ with index.
u = empty Buffer
$key = getKey(rec_i)$
for all $rec_j \in index[key]$ do
    u.append($\langle key, rec_i, rec_j \rangle$)
end for
Return u
```

AM-Join utilizes Broadcast-Joins for keys that are hot in only one relation. Extending AM-Join to outer-joins entails performing Small-Large outer-joins. We propose Index-Broadcast-Join (IB-Join). The full-outer-join (IB-FO-Join) algorithm is in Alg. 9. The left-outer-join and the right-outer-join are mere simplifications.

Like the inner-join, IB-FO-Join builds an index on the small relation, collects it at the driver, and broadcasts it to all the executors. The results of the left-outer-join are then computed. The $map_{broadcastLeftOuterJoin}$ function (Alg. 10) is similar to Alg. 8, but also produces *unjoined* records from the non-replicated relation.

To produce the full-outer-join, IB-FO-Join needs to also produce the right-anti join results, i.e., the *unjoinable* records in the small replicated relation. The driver finds the unjoinable keys in the replicated relation by utilizing the already replicated relation. The $map_{getRightJoinableKey}$ (Alg. 11) produces a set for each record in the larg non-replicated relation. The set either has a single key from the replicated relation if the key is joinable with the non-replicated relation, or is empty otherwise. The driver then unions these sets, first on each executor, and then across the network. The union constitutes the set of joined keys in the replicated relation. The index keys not in this union constitute the unjoinable keys.

Algorithm 9 *indexBroadcastFullOuterJoin*($\mathcal{R}$, $\mathcal{S}$)

```
Input: Two relations to be joined.
Output: The join results.
Assumes: $\mathcal{S}$ fits in memory. $\mathcal{R} \gg \mathcal{S}$.
 1: index = $\mathcal{S}$.map(map$_{buildIndex}$).groupByKey
 2: broadcast index to all executors
 3: $Q_{leftOuter}$ = $\mathcal{R}$.map(map$_{broadcastLeftOuterJoin}$)
 4: $keys_{joined}$ = $\mathcal{R}$
 5:     .map(map$_{getRightJoinableKey}$)
 6:     .combine(unionSets)
 7:     .treeAggregate(unionSets)
 8: $keys_{unjoinable}$ = index.keys - $keys_{joined}$
 9: broadcast $keys_{unjoinable}$ to all executors
10: $Q_{rightAnti}$ = $\mathcal{S}$.map(map$_{rightAntiJoin}$)
    Return $Q_{leftOuter} \cup Q_{rightAnti}$
```

Algorithm 10 $map_{broadcastLeftOuterJoin}(rec_i)$

```
Local: index, a map from each join key in the replicated relation
       to all the records with that key.
Input: A record from the non-replicated relation.
Output: A list of triplets representing the
        left-outer-join of $rec_i$ with index.
u = empty Buffer
$key = getKey(rec_i)$
if key in index then
    for all $rec_j \in index[key]$ do
        u.append($\langle key, rec_i, rec_j \rangle$)
    end for
else
    u.append($\langle key, rec_i, null \rangle$)
end if
Return u
```

Algorithm 11 $map_{getRightJoinableKey}(rec_i)$

```
Local: index, a map from each join key in the replicated relation
       to all the records with that key.
Input: A record from the non-replicated relation.
Output: The key of $rec_i$ if it is joinable with index.
u = empty Set
$key = getKey(rec_i)$
if $key \in index$.keys then
    u.append(key)
end if
Return u
```

The driver then broadcasts these unjoinable keys back to the executors, and maps the original small relation using $map_{rightAntiJoin}$ (Alg. 12). Each executor scans its local partition of the small relation, and outputs only the records whose keys are unjoinable. The left-outer-join results are unioned with the right-anti join results to produce the full-outer-join results.

4.2.1 *Optimizations.*

Broadcast-Joins can be optimized as follows.

(1) The driver should send the joinable keys over the network if they are fewer than the unjoinable keys (Line 9 of Alg. 9). In that case, the condition in Line 2 of Alg. 12 has to be reversed.
(2) Lines 3 and 5 in Alg. 9 can be combined into a single *mapRec* function to reduce the number of scans on the large relation.
(3) In a shared-nothing architecture where an executor, e_i, can scan its data partition, Lines 5 and 6 in Alg. 9 can be combined. A single set can be allocated and populated as $\mathcal{R}_i$ is scanned.

(4) In a shared-nothing architecture that supports both multicasting and broadcasting, instead of broadcasting the index to all the executors (Line 2 in Alg. 7 and Alg. 9), the driver may broadcast the unique keys only. Instead of the local joins (Line 3 in Alg. 7 and Alg. 9), each executor, e_i, joins the $\mathcal{S}$ keys with its $\mathcal{R}_i$ and sends the joinable keys to the driver. The driver multicasts each $\mathcal{S}$ record only to its joinable executors. Each e_i joins the received records with its $\mathcal{R}_i$. The driver need not compute the unjoinable $\mathcal{S}$ keys (Lines 4–8 in Alg. 9), since they are the keys not multicasted at all.

```
Algorithm 12 $map_{rightAntiJoin}(rec_j)$
Local: $keys_{unjoinable}$, a set of unjoinable keys.
Input: A record from the replicated relation.
Output: The join results of $rec_j$ if its key is in $keys_{unjoinable}$.
key = getKey(rec_j)
if $key \in keys_{unjoinable}$ then
    Return $\langle key, null, rec_j \rangle$
end if
```

4.2.2 Comparison with the State-of-the-Art Algorithms. One main difference between the proposed IB-Join (Alg. 9), DER [72] and DDR [24] is subtle but effective. To flag the unjoined records in the n executors, IB-Join utilizes the unique join keys, instead of the DBMS-assigned record ids in case of DER[5] or entire records in case of DDR. Assuming the join key is of the same size as the record id, and is significantly smaller than the record, sending unique join keys reduces the network load, and scales better with a skewed $\mathcal{S}$.

The other main difference is how the unjoinable records are identified after the first Broadcast-Join. DER hashes the unjoined ids from all executors over the network, and performs an inner Hash-Join. This entails hashing the records of $\mathcal{R}$, each of size $m_{\mathcal{R}}$, and the unjoinable ids of $\mathcal{S}$, each of size m_{id} over the network. The communication cost of this step is $(n+1)\times|\mathcal{S}|\times m_{id}+|\mathcal{R}|\times m_{\mathcal{R}}$. DDR hashes the $\mathcal{S}$ records, each of size $m_{\mathcal{S}}$, from all executors, incurring a communication cost of $n \times |\mathcal{S}| \times m_{\mathcal{S}}$, which is as costly as the first Broadcast-Join itself. IB-Join collects and broadcasts the unique keys, each of size m_b (Lines 4 – 7 and 9 of Alg. 9, respectively), with a communication cost of $2n \times |\mathcal{S}| \times m_b$. This is much more efficient than DER and DDR, even without considering that broadcasting the unique keys is done in time logarithmic in n.

4.3 The AM-Join Algorithm

```
Algorithm 13 $amJoin(\mathcal{R}, \mathcal{S})$
Input: Two relations to be joined.
Output: The join results.
1: $\kappa_{\mathcal{R}} = getHotKeys(\mathcal{R})$
2: $\kappa_{\mathcal{S}} = getHotKeys(\mathcal{S})$
3: $\langle \mathcal{R}_{HH}, \mathcal{R}_{HC}, \mathcal{R}_{CH}, \mathcal{R}_{CC} \rangle = splitRelation(\mathcal{R}, \kappa_{\mathcal{R}}, \kappa_{\mathcal{S}})$
4: $\langle \mathcal{S}_{HH}, \mathcal{S}_{HC}, \mathcal{S}_{CH}, \mathcal{S}_{CC} \rangle = splitRelation(\mathcal{S}, \kappa_{\mathcal{S}}, \kappa_{\mathcal{R}})$
5: $Q = treeJoin(\mathcal{R}_{HH}, \mathcal{S}_{HH})$
6:     $\cup\ broadcastJoin(\mathcal{R}_{HC}, \mathcal{S}_{CH})$
7:     $\cup\ broadcastJoin(\mathcal{S}_{HC}, \mathcal{R}_{CH})$.map(map$_{swapJoinedRecords}$)
8:     $\cup\ shuffleJoin(\mathcal{R}_{CC}, \mathcal{S}_{CC})$
    Return $Q$
```

AM-Join employs the above algorithms to maximize the executors' utilization, and minimize the network cost. AM-Join starts by collecting a list of hot keys for both relations, $\kappa_{\mathcal{R}}$ and $\kappa_{\mathcal{S}}$. Then, it splits each relation into four sub-relations containing the keys that are hot in both sides, the keys that are hot in its side but cold in the other side, the keys that are hot in the other side but cold in its side, and the keys that are cold in both sides, respectively. Therefore, $\mathcal{R}$ is split into $\mathcal{R}_{HH}$, $\mathcal{R}_{HC}$, $\mathcal{R}_{CH}$ and $\mathcal{R}_{CC}$, and $\mathcal{S}$ is split similarly. Irrespective of the keys in $\kappa_{\mathcal{R}}$ and $\kappa_{\mathcal{S}}$, it is provable that the join results are the union of the four joins in Eqn. 4.

$$\begin{aligned} Q = \mathcal{R}_{HH} &\bowtie \mathcal{S}_{HH} \\ \cup\, \mathcal{R}_{HC} &\bowtie \mathcal{S}_{CH} \\ \cup\, \mathcal{R}_{CH} &\bowtie \mathcal{S}_{HC} \\ \cup\, \mathcal{R}_{CC} &\bowtie \mathcal{S}_{CC} \end{aligned} \tag{4}$$

For the example of the two relations in Fig. 1(a), assuming the hot keys on either $\mathcal{R}$ and $\mathcal{S}$ are those occurring multiple times, then $\kappa_{\mathcal{R}}$ and $\kappa_{\mathcal{S}}$ are the sets $\{1, 2, 3, 4\}$ and $\{1, 6, 11, 12\}$, respectively. Fig. 3(b) shows the sub-relations of $\mathcal{R}$ and $\mathcal{S}$ paired together to be joined, according to Eqn. 4. If all the joins in Eqn. 4 are inner-joins, then the inner-join results of $\mathcal{R}$ and $\mathcal{S}$ are given in Fig. 1(b).

Since the first join involves keys that are hot in both the joined relations, Tree-Join (Alg. 1) is used. For the second and the third joins in Eqn. 4, the cold side is likely to have few keys, since they are a subset of the hot keys on the hot side of the join, each with few records. Hence, these two joins are good candidates for Broadcast-Join. Each record resulting from the third join is swapped so the a attributes from $\mathcal{R}$ precede the c attributes from $\mathcal{S}$, where a and c are their non-join-key attribute(s), respectively. Finally, the fourth join involves keys that are hot on neither side, and is hence performed using a Shuffle-Join. The algorithm is shown in Alg. 13.

While the AM-Join algorithm bears some resemblance to the PRPD family, it deviates in a key aspect. PRPD assigns a key as hot to the relation that has more records for that key. AM-Join independently collects the hot keys for $\mathcal{R}$ and $\mathcal{S}$, resulting in the simple and elegant Eqn. 4. Eqn. 4 leads to (a) AM-Join utilizing the scalable Tree-Join to join the keys that are hot in both $\mathcal{R}$ and $\mathcal{S}$, and (b) extending AM-Join smoothly to all outer-join variants (§ 4.3.5).

4.3.1 Collecting Hot Keys. The *getHotKeys* algorithm (Lines 1 and 2 in Alg. 13) collects the hot keys for both $\mathcal{R}$ and $\mathcal{S}$. The maximum numbers of hot keys collected for $\mathcal{R}$ and $\mathcal{S}$, $|\kappa_{\mathcal{R}}|_{max}$ and $|\kappa_{\mathcal{S}}|_{max}$, respectively, are not shown in Alg. 13 for simplicity, but are discussed in § 4.3.2. We only discuss $|\kappa_{\mathcal{R}}|_{max}$, but the logic applies to $|\kappa_{\mathcal{S}}|_{max}$. We assume $\mathcal{R}$ is evenly distributed on the n executors.

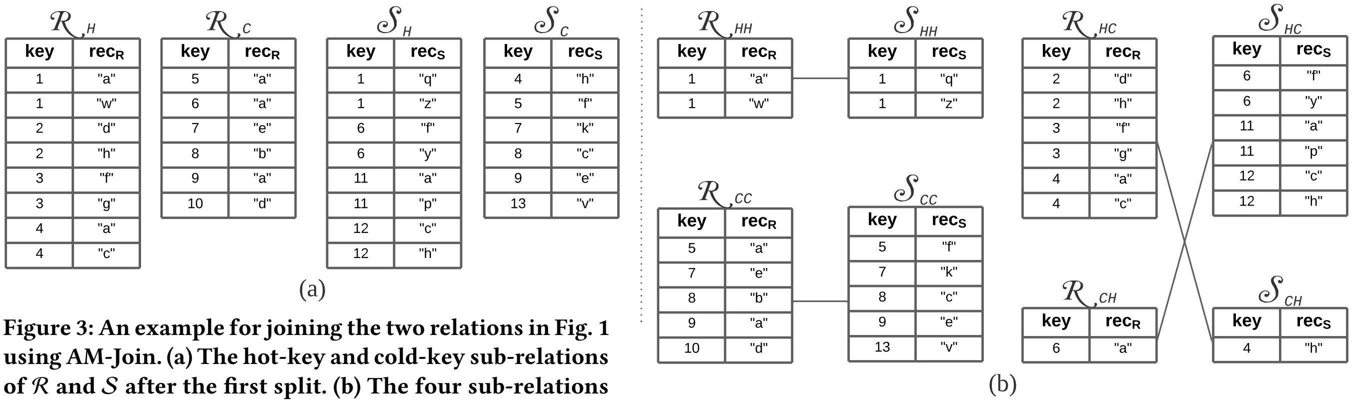
Hot keys are collected using the approximate distributed heavy-hitters algorithm in [3]. This algorithm runs the local Space-Saving algorithms [49] on individual partitions, and then merges the individual results over the network in a tree-like manner using a priority queue of bounded size, $|\kappa_{\mathcal{R}}|_{max}$. The cost of collecting the hot keys is the cost of scanning the local $\mathcal{R}_i$ partitions in parallel, and the cost of merging the hot keys over the network in a tree-like manner. Given m_b and $m_{\mathcal{R}}$, the average sizes of the join key and the $\mathcal{R}$ records, respectively, the total cost is given by Rel. 5.

$$\Delta_{getHotKeys} = \frac{|\mathcal{R}| \times m_{\mathcal{R}}}{n} + |\kappa_{\mathcal{R}}|_{max} \times m_b \times \lambda \times \log(n) \tag{5}$$

If $\mathcal{R}$ is already partitioned by the join key, all records of any key reside on one partition, and its local and global frequencies are equal. The above technique would then find the hot keys exactly.

We like to contrast collecting the hot keys using a distributed heavy-hitters algorithm to histogramming the relation using a distributed quantiles algorithm [33, 52, 70]. Given $\mathcal{R}$ is distributed

[5] A minor advantage of IB-Join is abolishing the dependency on DBMS-specific functionality. This facilitates its adoption in any shared-nothing architecture.



$\mathcal{R}_H$

key	rec_R
1	"a"
1	"w"
2	"d"
2	"h"
3	"f"
3	"g"
4	"a"
4	"c"

$\mathcal{R}_C$

key	rec_R
5	"a"
6	"a"
7	"e"
8	"b"
9	"a"
10	"d"

$\mathcal{S}_H$

key	rec_S
1	"q"
1	"z"
6	"f"
6	"y"
11	"a"
11	"p"
12	"c"
12	"h"

$\mathcal{S}_C$

key	rec_S
4	"h"
5	"f"
7	"k"
8	"c"
9	"e"
13	"v"

(a)

$\mathcal{R}_{HH}$

key	rec_R
1	"a"
1	"w"

$\mathcal{S}_{HH}$

key	rec_S
1	"q"
1	"z"

$\mathcal{R}_{CC}$

key	rec_R
5	"a"
7	"e"
8	"b"
9	"a"
10	"d"

$\mathcal{S}_{CC}$

key	rec_S
5	"f"
7	"k"
8	"c"
9	"e"
13	"v"

$\mathcal{R}_{HC}$

key	rec_R
2	"d"
2	"h"
3	"f"
3	"g"
4	"a"
4	"c"

$\mathcal{S}_{HC}$

key	rec_S
6	"f"
6	"y"
11	"a"
11	"p"
12	"c"
12	"h"

$\mathcal{R}_{CH}$

key	rec_R
6	"a"

$\mathcal{S}_{CH}$

key	rec_S
4	"h"

(b)

Figure 3: An example for joining the two relations in Fig. 1 using AM-Join. (a) The hot-key and cold-key sub-relations of $\mathcal{R}$ and $\mathcal{S}$ after the first split. (b) The four sub-relations for each of the two input relations after the second split, and their joins based on Eqn. 4.

evenly between the n executors. and an error rate, ϵ, the local heavy-hitters algorithm uses $\Theta(\frac{1}{\epsilon})$ space [49], while the local quantiles algorithm uses $O(\frac{1}{\epsilon}\log(\epsilon\frac{\mathcal{R}}{n}))$ space [27][6]. Collecting heavy hitters is more precise given the same memory, and incurs less communication when merging local statistics over the network.

4.3.2 How many Hot Keys to Collect? We only discuss $|\kappa_{\mathcal{R}}|_{max}$, but the logic applies to $|\kappa_{\mathcal{S}}|_{max}$. Any key with $\left(1+\sqrt{2+\lambda}\right)^{\frac{3}{2}}$ or more records is hot (Ineq. 2). Three upper bounds apply to $|\kappa_{\mathcal{R}}|_{max}$.

(1) No more than $\frac{M}{m_b}$ hot keys can be collected, since no more join keys of size m_b can fit in a memory of size M.
(2) The number of hot keys in $\mathcal{R}$ cannot exceed $\frac{|\mathcal{R}|}{\left(1+\sqrt{2+\lambda}\right)^{\frac{3}{2}}}$.
(3) The second join in Eqn. 4 is executed using Broadcast-Join. Let $m_{\mathcal{S}}$ be the average record size in $\mathcal{S}$. $\mathcal{S}_{CH}$ is computed after *getHotKeys*, but is bounded to fit in memory by Rel. 6.

$$|\mathcal{S}_{CH}| \leq \frac{M}{m_{\mathcal{S}}} \tag{6}$$

The keys in $\mathcal{S}_{CH}$ are a subset of $\kappa_{\mathcal{R}}$. Each of these keys has frequency below $\left(1+\sqrt{2+\lambda}\right)^{\frac{3}{2}}$ in $\mathcal{S}_{CH}$. To ensure $\mathcal{S}_{CH}$ fits in memory, we limit $|\kappa_{\mathcal{R}}|$ by Rel. 7.

$$|\kappa_{\mathcal{R}}|_{max} \leq \frac{|\mathcal{S}_{CH}|}{\left(1+\sqrt{2+\lambda}\right)^{\frac{3}{2}}} \tag{7}$$

Substituting Rel. 6 in Rel. 7 gives an upper bound on $|\kappa_{\mathcal{R}}|_{max}$.

Given the three upper bounds above, $|\kappa_{\mathcal{R}}|_{max}$, is given by Eqn. 8.

$$|\kappa_{\mathcal{R}}|_{max} = \min\left(\frac{\min\left(|\mathcal{R}|, \frac{M}{m_{\mathcal{S}}}\right)}{\left(1+\sqrt{2+\lambda}\right)^{\frac{3}{2}}}, \frac{M}{m_b}\right) \tag{8}$$

4.3.3 Splitting the Relations. Once the hot keys for both $\mathcal{R}$ and $\mathcal{S}$ are collected, each relation is split into its four sub-relations. We only discuss $\mathcal{R}$, but the logic applies to $\mathcal{S}$. This *splitRelation* module (Alg. 14) proceeds in two rounds. It first splits $\mathcal{R}$ into $\mathcal{R}_H$, the sub-relation whose keys are in $\kappa_{\mathcal{R}}$, and $\mathcal{R}_C$, the cold-key records in $\mathcal{R}$. In the example in Fig. 1, the results of the first-round of splitting is in Fig. 3(a). In the second round, $\mathcal{R}_H$ is split into $\mathcal{R}_{HH}$, the sub-relation whose keys are in $\kappa_{\mathcal{S}}$, and $\mathcal{R}_{HC}$, the remaining records in $\mathcal{R}_H$; it also splits $\mathcal{R}_C$ into $\mathcal{R}_{CH}$ and $\mathcal{R}_{CC}$, similarly using $\kappa_{\mathcal{S}}$. In the example in Fig. 1, the second-round splits are in Fig. 3(b).

Algorithm 14 *splitRelation*($\mathcal{R}$, $\kappa_{\mathcal{R}}$, $\kappa_{\mathcal{S}}$)

```
Input: A relation to be split,
       its hot keys,
       and the hot keys of the other relation.
Output: The four splits of $\mathcal{R}$.
1: $\langle \mathcal{R}_H, \mathcal{R}_C \rangle = splitRelationPartitions(\mathcal{R}, rec \rightarrow rec \in \kappa_{\mathcal{R}})$
2: $\langle \mathcal{R}_{HH}, \mathcal{R}_{HC} \rangle = splitRelationPartitions(\mathcal{R}_H, rec \rightarrow rec \in \kappa_{\mathcal{S}})$
3: $\langle \mathcal{R}_{CH}, \mathcal{R}_{CC} \rangle = splitRelationPartitions(\mathcal{R}_C, rec \rightarrow rec \in \kappa_{\mathcal{S}})$
   Return $\langle \mathcal{R}_{HH}, \mathcal{R}_{HC}, \mathcal{R}_{CH}, \mathcal{R}_{CC} \rangle$
```

The splitting is done by each executor reading its local partition sequentially, checking if the record key exists in the hot-key set, and writing it to either the hot-key sub-relation or the cold-key sub-relation. The splitting involves no communication between the executors. The two rounds can be optimized into one.

4.3.4 When not to Perform Broadcast-Joins? The second and third joins in Eqn. 4 are executed using Broadcast-Joins. We only discuss the second join, but the logic applies to the third. For the Broadcast-Join assumption to hold, broadcasting the records of the small relation, $\mathcal{S}_{CH}$, over the network has to be faster than splitting the large relation, $\mathcal{R}_{HC}$, among the n executors over the network. From [11], the time to read $\mathcal{S}_{CH}$ and broadcast it follows the Big-Θ below.

$$\Delta_{broadcast_{\mathcal{S}_{CH}}} \approx \Theta\left(|\mathcal{S}_{CH}| \times m_{\mathcal{S}} \times \left(1 + \lambda \times \log_{\lambda+1}(n)\right)\right)$$

Let $m_{\mathcal{R}}$ be the average size of a record in $\mathcal{R}$. The time to read $\mathcal{R}_{HC}$ and split it among the n executors follows the Big-Θ below.

$$\Delta_{split_{\mathcal{R}_{HC}}} \approx \Theta\left(|\mathcal{R}_{HC}| \times m_{\mathcal{R}} \times (1+\lambda)\right)$$

At the time of executing the second join in Eqn. 4, $\mathcal{R}_{HC}$ and $\mathcal{S}_{CH}$ have already been computed. A Broadcast-Join is performed if $\Delta_{split_{\mathcal{R}_{HC}}} \geq \Delta_{broadcast_{\mathcal{S}_{CH}}}$. Otherwise, Tree-Join is performed.

4.3.5 The Outer Variants of AM-Join. Due to the AM-Join elegant design, the outer-join variants are minor modifications of the inner-join in Alg. 13. The only difference is that some of the four joins of Eqn. 4 are executed using outer-join algorithms. Table. 2 shows the

[6] This is the theoretical bound on any deterministic single-pass comparison-based quantiles algorithm, even if the algorithms in [33, 52, 70] do not necessarily use it.

	left-outer AM-Join	right-outer AM-Join	full-outer AM-Join
$\mathcal{R}_{HH} \bowtie \mathcal{S}_{HH}$	Tree-Join	Tree-Join	Tree-Join
$\mathcal{R}_{HC} \bowtie \mathcal{S}_{CH}$	Broadcast left-outer-join	Broadcast-Join	Broadcast left-outer-join
$\mathcal{S}_{HC} \bowtie \mathcal{R}_{CH}$	Broadcast-Join	Broadcast left-outer-join	Broadcast left-outer-join
$\mathcal{R}_{CC} \bowtie \mathcal{S}_{CC}$	Shuffle left-outer-join	Shuffle right-outer-join	Shuffle full-outer-join

Table 2: The algorithms for the four sub-joins of Eqn. 4 of the outer variants of AM-Join.

algorithm used for each of the four joins to achieve the outer-join variants of AM-Join. The first and the fourth joins are performed using Tree-Join and a Shuffle outer-join, respectively. The second and third joins are performed using a Broadcast-Join or Broadcast left-outer-join. Applying the outer-joins in Table. 2 to the joins in Fig. 3(b) yields the left-outer, right-outer, and full-outer-join results in Fig. 1(c), Fig. 1(d), and Fig. 1(e), respectively.

Unlike the state-of-the-art distributed industry-scale algorithms, the different flavors of SkewJoin [19] built on top of Microsoft SCOPE, AM-Join supports all variants of outer-joins without record deduplication or custom partitioning of the relations. Moreover, AM-Join does not require introducing a new outer-join operator variant that understands the semantics of *witness* tuples, a form of tuples introduced in [19] to extend SkewJoin to outer-joins.

5 EVALUATION RESULTS

We evaluated the scalability and the handling of skewed data of Tree-Join, and AM-Join against Hash-Join, Broadcast-Join, Full-SkewJoin [19] (the state-of-the-art industry-scale algorithm of the PRPD family), and Fine-Grained partitioning for Skew Data (FGSD-Join) [33] (the state-of-the-art of the key-range-division family). We also compared against the Spark v3.x joins[7], the most advanced open-source shared-nothing algorithm. The two main optimizations of Spark3-Join is (a) dynamically changing the execution plan from Shuffle-Join to Broadcast-Join when one relation can fit in memory, and (b) dynamically combining and splitting data partitions to reach almost equi-size partitions for load balancing. Multicasting is not supported by Spark, the framework we used for evaluation. Evaluating multicasting algorithms (e.g., [56]) is left for future work.

The performance of the algorithms on outer-joins was in line with the inner-joins, reported below. However, we could build the outer-join variants of neither Full-SkewJoin, since it depends on Microsoft SCOPE-specific deduplication, nor FGSD-Join, since it was missing in [33]. We compared the proposed IB-Join against DER [72] and DDR [24] on Small-Large outer-joins.

We conducted comprehensive experiments on reproducible synthetic realistic data, and on real proprietary data. All the experiments were executed on machines (driver and executors) with 8GB of memory, 20GB of disk space, and 1 core. Based on Ineq. 2 and Rel. 8, among the most popular 1000 keys, those that have 100 records or more were considered hot. These values agree with the guidelines of Full-SkewJoin. The sample for FGSD-Join was 10× the number of partitions, which is 3× to 5× the number of executors.

The algorithms code and the code for the dataset generation and experiment running was written in Scala v2.11.12 and executed using Spark v2.4.3, except for the Spark v3.x experiments that were run on Spark v3.0.2. All the code was peer reviewed, and tested for correctness.. The code will be available on https://github.com/uber.

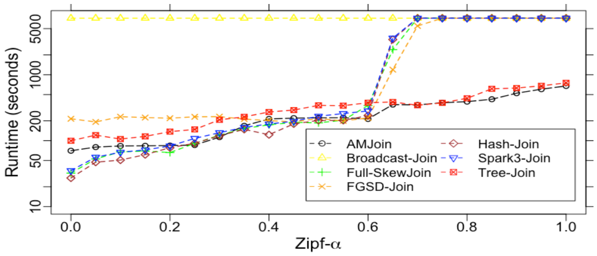


Figure 4: The runtime of the equi-join algorithms on 1000 executors for $D(\alpha, 100)$ while varying the Zipf-α.

5.1 Experiments on Synthetic Data

The synthetic data is a combination of random data generated using two processes. The first process generates a dataset of 10^9 records whose keys are uniformly selected from the positive range of the 32-bit Integer space. The second process generates a dataset with 10^7 records whose keys are drawn from a Zipf distribution with a skew-parameter α and a domain of 10^5 keys. The records of both processes have the same size, S.

Initially, the experiments were conducted using the Zipfian keys only. The Zipfian keys were generated using the Apache Commons math package, which uses the state-of-the-art inverse-CDF algorithm. Yet, generating skewed keys from a space of 10^5 was pushing the limits of the math package. The uniformly-distributed keys were eventually incorporated to increase the key space.

We refer to the combined dataset using the notation $D(\alpha, S)$. For instance, $D(0.5, 10^3)$ refers to a dataset whose records have a size of 10^3 Bytes, and has 10^9 records whose keys are uniformly selected from the positive Integer space, and 10^7 records whose keys are drawn from a Zipf-0.5 distribution with a range of 10^5.

Two sizes of records were experimented with, $S = 10^2$, and $S = 10^4$ Bytes, representing relatively small and large records. The Zipf-α was varied between 0.0 and 1.0, representing uniform to moderately skewed data. Higher values of α are common in many applications[8]. However, higher values of α are expected to favor algorithms that handle data skew better, namely AM-Join and Tree-Join. These two algorithms split the processing of each hot key among multiple executors. Finally, the number of executors used to execute the join was varied from 100 to 1000 executors to evaluate how the algorithms utilize more resources to scale to more skewed datasets. All algorithms were allowed 2 hours to execute the join.

5.1.1 Scalability with Data Skew. Fig. 4 shows the runtimes of the algorithms when joining two relations, each has records with a size of 10^2 Bytes, and has 10^9 records whose keys are uniformly selected from the positive Integer space, and 10^7 records whose Zipfian keys have a range of 10^5 while varying the Zipf-α.

For these relatively small records, the runtimes of all the algorithms were comparable until Zipf-α reached 0.6 with a slight edge

[7] https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html, https://docs.databricks.com/spark/latest/spark-sql/aqe.html

[8] One such application is the word frequency in Wikipedia [71].

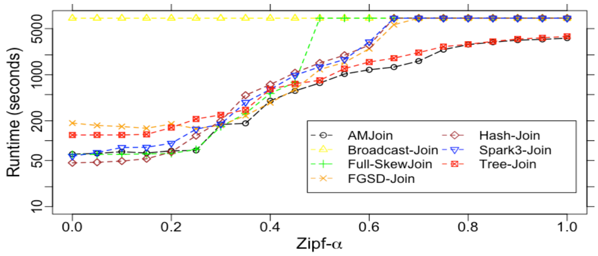


Figure 5: The runtime of the equi-join algorithms on 1000 **executors for** $D(\alpha, 10000)$ **while varying the Zipf-**α**.**

for Hash-Join until Zipf-α reached 0.15. For Zipf-α in the range $[0.2, 0.55]$, none of the algorithms was a consistent winner. AM-Join and Hash-Join had a clearer edge for Zipf-α at 0.65. Only they successfully executed the joins starting at 0.7.

The experiment was repeated with records of size 10^4 Bytes instead of 10^2 Bytes, and the results are reported in Fig. 5. For these relatively large records, the runtimes of all the algorithms were comparable until Zipf-α reached 0.3 with a slight edge for Hash-Join in the Zipf-α $[0.0, 0.15]$ range. For the Zipf-α $[0.2, 0.3]$ range, none of the algorithms was a consistent winner. AM-Join had the edge for Zipf-α above 0.35. Full-SkewJoin, Hash-Join, Spark3-Join and FGSD-Join could not finish within the deadline starting at Zipf-α of 0.5, 0.65, 0.65, and 0.7 respectively.

The runtime of all the algorithms increased as the data skew increased. Not only do some executors become more loaded than others, but also the size of the results increases. The only exception was FGSD-Join, whose runtime was very stable through the low to mid range of Zipf-α ($[0.0, 0.6]$ and $[0.0, 0.3]$ in Fig. 4 and Fig. 5, respectively). The sampling phase was costly, but useless in load-balancing when the data was not skewed. In the mid Zipf-α range, FGSD-Join allocated more executors to the hot keys than the simpler Hash-Join, and hence was faster. For higher Zipf-α, FGSD-Join was bottlenecked by the executor joining the hottest key.

AM-Join and Tree-Join scale almost linearly with the increased skewness, since they are able to distribute the load on the executors fairly evenly. From Ineq. 3, their runtime is expected to increase with ℓ_{max}, the frequency of the hottest key, which is impacted by Zipf-α. Meanwhile, Full-SkewJoin and Hash-Join perform relatively well for weakly skewed joins until both the joined relations become mildly skewed (at a Zipf-α of 0.6 and 0.4 in Fig. 4 and Fig. 5, respectively). For moderately skewed data, both algorithms failed to produce the results within the deadline, albeit for different reasons. Full-SkewJoin could not load the hot keys in the memory of all the executors, while the executors of Hash-Join were bottlenecked by the join results of the hottest keys.

The Hash-Join executes faster than the adaptive algorithms (AM-Join and Full-SkewJoin) for the majority of the weakly skewed range (Zipf-α $[0.0, 0.6]$ and $[0.0, 0.3]$ ranges in Fig. 4 and Fig. 5, respectively), since the adaptive algorithms are slowed down by computing the hot keys. However, the adaptive algorithms utilizing the Broadcast-Join to join the keys that are hot in only one of the joined relations pays off for more skewed data. For larger Zipf-α (Zipf-α of 0.65 and 0.45 in Fig. 4 and Fig. 5, respectively), Full-SkewJoin executed clearly faster than Hash-Join, and AM-Join executed significantly faster than both.

Since the data partitions are too big to fit in memory, and they are already of equal size, Spark3-Join performed as a basic Sort-Merge-Join. This was comparable, but slightly slower than Hash-Join.

Broadcast-Join was never able to execute successfully, regardless of the data skew, since both datasets were too large to load into the memory of the executors. Broadcast-Join is a fast algorithm, but is not a scalable one. This is clear from comparing the runtimes of Full-SkewJoin in Fig. 4 and Fig. 5, respectively. Since Full-SkewJoin employs Broadcast-Join for the hot keys, it was able to handle more skewed data when the records were smaller (until Zipf-α = 0.65 and 0.45 when the record sizes were 10^2 and 10^4, respectively).

For Tree-Join and AM-Join, there were regions of relative flattening of the runtime due to the stepwise increase in the number of iterations. This is clear in the α $[0.5, 0.7]$ and $[0.85, 1.0]$ ranges in Fig. 4, and in the α $[0.55, 0.7]$ and $[0.8, 1.0]$ ranges in Fig. 5. This is because the number of iterations is a function of ℓ_{max} which is impacted by Zipf-α.

5.1.2 *Exploring the Parameter Space.*

Similar results were obtained for different parameters, e.g., scaling the keys by a multiplier to influence the join selectivity, using 10^8 Zipfian records[9], using records of size 10^3 Bytes, or allocating 4 GB of memory per executor. For non-skewed data, Hash-Join was consistently the fastest and FGSD-Join was consistently the slowest. Full-SkewJoin was fast while it can fit the data in memory, and AM-Join and Tree-Join were fast and able to scale for various α values. The runtime of Spark3-Join closely tracked that of Hash-Join but was slightly slower.

We also experimented with Small-Large joins. The Broadcast-Join performed best, and so did Spark3-Join since it also morphed into a Broadcast-Join. Full-SkewJoin performed slightly (11% to 15%) better than AM-Join, Tree-Join, FGSD-Join, and Hash-Join. As the size of the small relation increased and reached the memory limit, the Spark3-Join did not execute a Broadcast-Join, and the Broadcast-Join cannot accommodate the small relation in memory, and the results were very similar to those in § 5.1.1.

5.1.3 *Scalability with Computational Resources.*

Fig. 6 shows the runtimes of the equi-join algorithms while varying the number of executors for two $D(0.65, 100)$ datasets. While the Zipf-α of 0.65 generates mildly skewed keys, this value was chosen, since this is the largest α where all the algorithms except Broadcast-Join were able to compute the join results.

All the algorithms, except Broadcast-Join, were able to scale with the increased number of resources. The algorithms showed a reduction in runtime as they were allowed more resources, albeit with various degrees. While FGSD-Join was the slowest at 100 executors, its runtime showed the biggest absolute drop. As the number of executors increased, the sample used by FGSD-Join to determine the key-ranges grew, allowing FGSD-Join to better distribute the load. The improvements of Hash-Join and Spark3-Join were the fastest to saturate, and their run-times almost did not improve starting at 700 executors, since the bottleneck was joining the hottest keys. Full-SkewJoin followed, and stopped having noticeable improvement at 800 executors. AM-Join and Tree-Join were able to improve their runtime as more executors were added, since they were able to split the load of joining the hottest keys among all the executors.

[9] Using 1000 executors, each dataset took over 10 hours to generate.

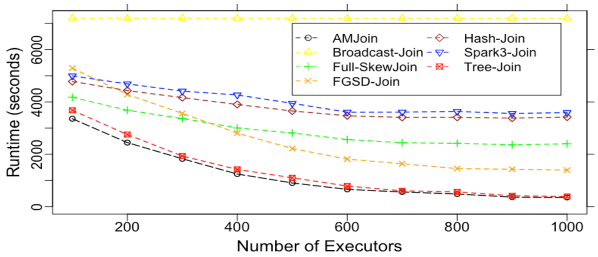


Figure 6: The runtime of the equi-join algorithms for $D(0.65, 100)$ while varying the number of executors.

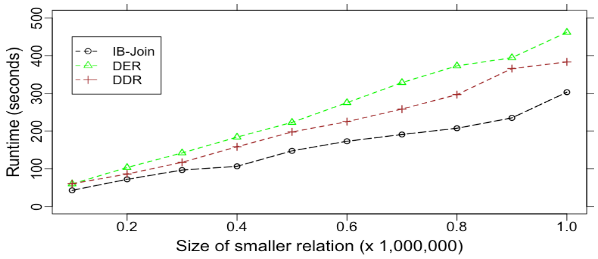


Figure 7: The runtime of the Small-Large outer-join algorithms while varying the size of the smaller relation.

There is another phenomenon that is worth highlighting. The difference in the runtime between AM-Join and Tree-Join diminished as more executors were used. The reason is the majority of the scalability come from organizing the equi-join execution in multiple stages, which is shared by both algorithms.

Again, the Broadcast-Join was never able to execute successfully, regardless of the number of executors used, since neither of the datasets could be fit in the memory of the executors.

5.1.4 Performance on Small-Large Outer-Joins. Fig. 7 shows the runtimes of IB-Join, DER [72] and DDR [24][10] when performing a right-outer join of two relations with records of size 10^2 Bytes, and keys uniformly selected in the range $[1, 2 \times 10^5]$. The larger relation has 10^8 records, while the size of the smaller relation was varied between 10^5 and 10^6 records. The keys in the smaller relation were all even to ensure a selectivity of 50%. This is the selectivity that least favors the optimizations of IB-Join in § 4.2.1.

Fig. 7 confirms the communication cost analysis done in § 4.2.2. DDR was consistently faster than DER with one exception, while IB-Join was significantly faster than both. In reality, DER performs worse than shown in Fig. 7, since the time for assigning unique ids to the records is not shown.

5.2 Experiments on Real Data

We ran two experiments on real data. The first experiment was a self-join on records collected from trips in a specific geo-location in October, 2020. The self-joined relation had 5.1×10^8 distinct keys, and each record was of size 36 Bytes. The total size of the data was 17.6 TB. Only AM-Join and Tree-Join were able to finish successfully. They both took 2.0 hours to finish. Full-SkewJoin, Hash-Join and Broadcast-Join did not finish in 48 hours.

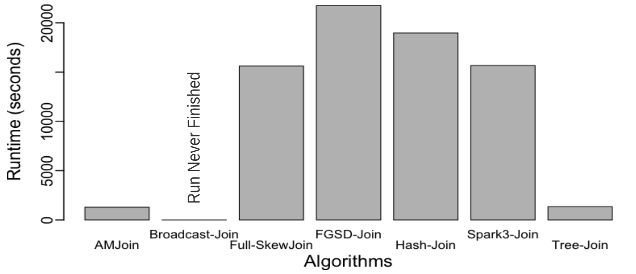


Figure 8: The runtime of the equi-join algorithms on 1000 executors on real data.

The second experiment was a join between two relations. The first relation is the one described above. The second relation represented data on a single day, October 14^{th}, 2020. The size of the second relation was 0.57 TB. The runtime of the equi-join algorithms on 1000 executors is shown in Fig. 8. All the algorithms, were able to finish successfully, except for Broadcast-Join since it could not fit the smaller relation in the memory of the executors. Thanks to organizing the equi-join execution in multiple stages, AM-Join and Tree-Join were able to execute an order of magnitude faster than all the other algorithms.

6 CONCLUSION

We propose Adaptive-Multistage-Join (AM-Join) a fast, efficient and scalable equi-join algorithm that is built using the basic MapReduce primitives, and is hence deployable in any distributed shared-nothing architecture. We started by proposing Tree-Join, a novel algorithm that organizes the equi-join execution in multiple stages. Tree-Join attains scalability by distributing the load of joining a key that is hot in both relations to multiple executors. Such keys are the scalability bottleneck of the state-of-the-art distributed algorithms. AM-Join utilizes Tree-Join for load balancing, high resource utilization, and scalability. Moreover, AM-Join utilizes Broadcast-Joins that reduce the network load when joining keys that are hot in only one relation. By utilizing Tree-Join and Broadcast-Join, AM-Join achieves scalability, speed and efficiency. AM-Join extends to all the outer-joins elegantly without record deduplication or custom table partitioning, unlike the state-of-the-art industry-scale algorithms [19]. For the fastest execution of AM-Join outer-joins, we proposed Index-Broadcast-Join (IB-Join) that improves on the state-of-the-art Small-Large outer-join algorithms [24, 72].

Our future directions focus on optimizing AM-Join for the general shared-nothing architecture that supports multicasting data, for NUMA machines connected by a high-bandwidth network, and learning from the RDMA and the work-stealing enhancements of [58]. Moreover, we plan to explore using Radix join [12, 48] that it is only bound by the memory bandwidth as the Shuffle-Join.

7 ACKNOWLEDGMENTS

We would like to express our appreciation to Sriram Padmanabhan, Vijayasaradhi Uppaluri, Gaurav Bansal, and Ryan Stentz from Uber for revising the manuscript and improving the presentation of the algorithms, and to Nicolas Bruno from Microsoft for discussing the SkewJoin algorithms. We are also indebted to the anonymous reviewers #1, #4, and #5 for critically reading this manuscript, and for providing suggestions that substantially improved its quality.

[10] For DER and DDR, the Spark implementations provided by the authors of [24] at https://github.com/longcheng11/small-large were used in these experiments.

REFERENCES

[1] F. Afrati, N. Stasinopoulos, J. Ullman, and A. Vassilakopoulos. SharesSkew: An Algorithm to Handle Skew for Joins in MapReduce. *Information Systems*, 77:129–150, 2018.

[2] F. Afrati and J. Ullman. Optimizing Joins in a Map-Reduce Environment. In *EDBT International Conference on Extending Database Technology*, pages 99–110, 2010.

[3] P. Agarwal, G. Cormode, Z. Huang, J. Phillips, Z. Wei, and K. Yi. Mergeable Summaries. *TODS ACM Transactions on Database Systems*, 38(4):1–28, 2013.

[4] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *arXiv preprint arXiv:1207.0145*, 2012.

[5] K. Alway and A. Nica. Constructing Join Histograms from Histograms with q-error Guarantees. In *ACM SIGMOD International Conference on Management of Data*, pages 2245–2246, 2016.

[6] Apache Hadoop. http://hadoop.apache.org.

[7] F. Atta, S. Viglas, and S. Niazi. SAND Join — A Skew Handling Join Algorithm for Google's MapReduce Framework. In *IEEE INMIC International Multitopic Conference*, pages 170–175. IEEE, 2011.

[8] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[9] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *IEEE ICDE International Conference on Data Engineering*, pages 362–373, 2013.

[10] M. Bandle, J. Giceva, and T. Neumann. To Partition, or Not to Partition, That is the Join Question in a Real System. In *ACM SIGMOD International Conference on Management of Data*, pages 168–180, 2021.

[11] A. Bar-Noy and S. Kipnis. Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems. *Mathematical Systems Theory*, 27(5):431–452, 1994.

[12] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-Scale In-Memory Join Processing using RDMA. In *ACM SIGMOD International Conference on Management of Data*, pages 1463–1475, 2015.

[13] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. Distributed Join Algorithms on Thousands of Cores. *Proceedings of the VLDB Endowment*, 10(5):517–528, 2017.

[14] P. Bernstein, N. Goodman, E. Wong, C. Reeve, and J. Rothnie Jr. Query Processing in a System for Distributed Databases. *TODS ACM Transactions on Database Systems*, 6(4):602–625, 1981.

[15] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The End of Slow Networks: Itś Time for a Redesign. *Proceedings of the VLDB Endowment*, 9(7):528–539, 2016.

[16] S. Blanas, Y. Li, and J. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *ACM SIGMOD International Conference on Management of Data*, pages 37–48, 2011.

[17] S. Blanas, J. Patel, V. Ercegovac, J. Rao, E. Shekita, and Y. Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *ACM SIGMOD International Conference on Management of Data*, pages 975–986, 2010.

[18] M. Blasgen and K. Eswaran. Storage and Access in Relational Data Bases. *IBM Systems Journal*, 16(4):363–377, 1977.

[19] N. Bruno, Y. Kwon, and M.-C. Wu. Advanced Join Strategies for Large-Scale Distributed Computation. *Proceedings of the VLDB Endowment*, 7(13):1484–1495, 2014.

[20] R. Chen and V. Prasanna. Accelerating Equi-Join on a CPU-FPGA Heterogeneous Platform. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 212–219, 2016.

[21] Z. Chen and A. Zhang. A Survey of Approximate Quantile Computation on Large-Scale Data. *IEEE Access*, 8:34585–34597, 2020.

[22] L. Cheng, S. Kotoulas, T. Ward, and G. Theodoropoulos. QbDJ: A Novel Framework for Handling Skew in Parallel Join Processing on Distributed Memory. In *IEEE HPCC International Conference on High Performance Computing and Communications*, pages 1519–1527. IEEE, 2013.

[23] L. Cheng, S. Kotoulas, T. Ward, and G. Theodoropoulos. Robust and Skew-resistant Parallel Joins in Shared-Nothing Systems. In *ACM CIKM International Conference on Conference on Information and Knowledge Management*, pages 1399–1408, 2014.

[24] L. Cheng, I. Tachmazidis, S. Kotoulas, and G. Antoniou. Design and Evaluation of Small-Large Outer Joins in Cloud Computing Environments. *Journal of Parallel and Distributed Computing*, 110:2–15, 2017.

[25] T.-Y. Cheung. A Method for Equijoin Queries in Distributed Relational Databases. *IEEE TOC Transactions on Computers*, 100(8):746–751, 1982.

[26] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *ACM SIGMOD International Conference on Management of Data*, pages 63–78, 2015.

[27] G. Cormode and P. Veselỳ. A Tight Lower Bound for Comparison-Based Quantile Summaries. In *ACM PODS SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 81–93, 2020.

[28] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing over Data Streams. In *ACM SIGMOD International Conference on Management of Data*, pages 40–51, 2003.

[29] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[30] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE TKDE Transactions on Knowledge and Data Engineering*, 2(1):44–62, 1990.

[31] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1992.

[32] D. DeWitt, M. Smith, and H. Boral. A Single-User Performance Evaluation of the Teradata Database Machine. In *International Workshop on High Performance Transaction Systems*, pages 243–276. Springer, 1987.

[33] E. Gavagsaz, A. Rezaee, and H. Javadi. Load Balancing in Join Algorithms for Skewed Data in MapReduce Systems. *The Journal of Supercomputing*, 75(1):228–254, 2019.

[34] G. Graefe. Sort-Merge-Join: An Idea Whose Time Has(h) Passed? In *IEEE ICDE International Conference on Data Engineering*, pages 406–417. IEEE, 1994.

[35] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. ScaleJoin: a Deterministic, Disjoint-Parallel and Skew-Resilient Stream Join. *IEEE Transactions on Big Data*, 7(2):299–312, 2016.

[36] C. Guo, H. Chen, F. Zhang, and C. Li. Distributed Join Algorithms on Multi-CPU Clusters with GPUDirect RDMA. In *ICPP International Conference on Parallel Processing*, pages 1–10, 2019.

[37] M. Hassan and M. Bamha. An Efficient Parallel Algorithm for Evaluating Join Queries on Heterogeneous Distributed Systems. In *IEEE HiPC International Conference on High Performance Computing*, pages 350–358. IEEE, 2009.

[38] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *ACM SIGMOD International Conference on Management of Data*, pages 511–524, 2008.

[39] D. Jiang, A. Tung, and G. Chen. MAP-JOIN-REDUCE: Toward Scalable and Efficient Data Analysis on Large Clusters. *IEEE TKDE Transactions on Knowledge and Data Engineering*, 23(9):1299–1311, 2010.

[40] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU Join Processing Revisited. In *International Workshop on Data Management on New Hardware*, pages 55–62, 2012.

[41] C. Kim, T. Kaldewey, V. Lee, E. Sedlar, A. Nguyen, N. Satish, J. Chhugani, A. D. Blas, and P. Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009.

[42] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of Hash to Data Base Machine and its Architecture. *New Generation Computing*, 1(1):63–74, 1983.

[43] M. Lakshmi and P. Yu. Effectiveness of Parallel Joins. *IEEE Computer Architecture Letters*, 2(04):410–424, 1990.

[44] R. Lämmel. Google's MapReduce programming model — Revisited. *Science of Computer Programming*, 70(1):1–30, 2008.

[45] F. Li, S. Das, M. Syamala, and V. Narasayya. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *ACM SIGMOD International Conference on Management of Data*, pages 355–370, 2016.

[46] Q. Lin, B. Ooi, Z. Wang, and C. Yu. Scalable Distributed Stream Join Processing. In *ACM SIGMOD International Conference on Management of Data*, pages 811–825, 2015.

[47] J. Linn and C. Dyer. Data-Intensive Text Processing with MapReduce. *Synthesis Lectures on Human Language Technologies*, 3(1):1–177, 2010.

[48] S. Manegold, P. Boncz, and M. Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE TKDE Transactions on Knowledge and Data Engineering*, 14(4):709–730, 2002.

[49] A. Metwally, D. Agrawal, and A. E. Abbadi. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *ICDT International Conference on Database Theory*, pages 398–412. Springer, 2005.

[50] A. Metwally and C. Faloutsos. V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *Proceedings of the VLDB Endowment*, 5(8):704–715, 2012.

[51] A. Nica, I. Charlesworth, and M. Panju. Analyzing Query Optimization Process: Portraits of Join Enumeration Algorithms. In *IEEE ICDE International Conference on Data Engineering*, pages 1301–1304. IEEE, 2012.

[52] A. Okcan and M. Riedewald. Processing Theta-Joins using MapReduce. In *ACM SIGMOD International Conference on Management of Data*, pages 949–960, 2011.

[53] J. Paul, B. He, S. Lu, and C. Lau. Revisiting Hash Join on Graphics Processors: A Decade Later. *Distributed and Parallel Databases*, pages 1–23, 2020.

[54] J. Paul, S. Lu, B. He, and C. Lau. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *ACM SIGMOD International Conference on Management of Data*, pages 1413–1425, 2021.

[55] O. Polychroniou, W. Zhang, and K. Ross. Track Join: Distributed Joins with Minimal Network Traffic. In *ACM SIGMOD International Conference on Management of Data*, pages 1483–1494, 2014.

[56] O. Polychroniou, W. Zhang, and K. Ross. Distributed Joins and Data Placement for Minimal Network Traffic. *TODS ACM Transactions on Database Systems*, 43(3):1–45, 2018.

[57] D. Quoc, I. Akkus, P. Bhatotia, S. Blanas, R. Chen, C. Fetzer, and T. Strufe. ApproxJoin: Approximate Distributed Joins. In *ACM SoCC Symposium on Cloud Computing*, pages 426–438, 2018.

[58] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-Join: Adaptive Skew Handling for Distributed Joins over High-Speed Networks. In *IEEE ICDE International Conference on Data Engineering*, pages 1194–1205. IEEE, 2016.

[59] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-Speed Query Processing over High-Speed Networks. *Proceedings of the VLDB Endowment*, 9(4):228–239, 2015.

[60] R. Rui, H. Li, and Y.-C. Tu. Efficient Join Algorithms For Large Database Tables in a Multi-GPU Environment. *Proceedings of the VLDB Endowment*, 14(4):708–720, 2020.

[61] A. Salama, C. Binnig, T. Kraska, A. Scherp, and T. Ziegler. Rethinking Distributed Query Execution on High-Speed Networks. *IEEE Data Engineering Bulletin*, 40(1):27–37, 2017.

[62] P. Sanders, J. Speck, and J. Träff. Two-Tree Algorithms for Full Bandwidth Broadcast, Reduction and Scan. *Parallel Computing*, 35(12):581–594, 2009.

[63] D. Schneider and D. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. *ACM SIGMOD Record*, 18(2):110–121, 1989.

[64] S. Schuh, X. Chen, and J. Dittrich. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *ACM SIGMOD International Conference on Management of Data*, pages 1961–1976, 2016.

[65] D. Shasha and T.-L. Wang. Optimizing Equijoin Queries In Distributed Databases Where Relations Are Hash Partitioned. *TODS ACM Transactions on Database Systems*, 16(2):279–308, 1991.

[66] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hardware-conscious Hash-Joins on GPUs. In *IEEE ICDE International Conference on Data Engineering*, pages 698–709, 2019.

[67] M. Stonebraker. The Case for Shared Nothing. *IEEE Database Engineering Bulletin*, 9(1):4–9, 1986.

[68] S. Suri and S. Vassilvitskii. Counting Triangles and the Curse of the Last Reducer. In *WWW International Conference on World Wide Web*, pages 607–614, 2011.

[69] Y. Tian, F. Özcan, T. Zou, R. Goncalves, and H. Pirahesh. Building a Hybrid Warehouse: Efficient Joins Between Data Stored in HDFS and Enterprise Warehouse. *TODS ACM Transactions on Database Systems*, 41(4):1–38, 2016.

[70] A. Vitorovic, M. Elseidy, and C. Koch. Load Balancing and Skew Resilience for Parallel Joins. In *IEEE ICDE International Conference on Data Engineering*, pages 313–324. IEEE, 2016.

[71] Word frequency in Wikipedia (November 27, 2006). https://en.wikipedia.org/wiki/Zipf's_law.

[72] Y. Xu and P. Kostamaa. A New Algorithm for Small-Large Table Outer Joins in Parallel DBMS. In *IEEE ICDE International Conference on Data Engineering*, pages 1018–1024. IEEE, 2010.

[73] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling Data Skew in Parallel Joins in Shared-Nothing Systems. In *ACM SIGMOD International Conference on Management of Data*, pages 1043–1052, 2008.

[74] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *ACM SIGMOD International Conference on Management of Data*, pages 1029–1040, 2007.

[75] K. Yi and Q. Zhang. Optimal Tracking of Distributed Heavy Hitters and Quantiles. *Algorithmica*, 65(1):206–223, 2013.

[76] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. *HotCloud*, 10(10-10):95, 2010.