Framework-Agnostic Optimization of Repeated Skewed Joins at Massive Scale

Andrea Nardelli, Vladimir Vlassov, Amir H. Payberah
KTH Royal Institute of Technology, Sweden
{andnar,vladv,payberah}@kth.se

***Abstract*—The join operation is a common but expensive operation in many data analysis processes. Despite lots of work on optimizing distributed joins in data-parallel processing platforms, there is still room to improve its performance, especially in repeated join operations and skewed data. Repeated join is an operation used in many services, where the same join operation is repeated multiple times by different users. Moreover, due to skewed data, joins may create uneven load distribution over the workers that harms the performance. Our goal is to improve the performance of repeated distributed joins on massive data with respect to execution time and network traffic. To this end, we present SMBJOIN, a framework-agnostic join operation that reduces execution time and network traffic by 50% and 17%, respectively, compared to the distributed hash-join operation after five joins. We also present SMBJOINSKEWED, a variant of SMBJOIN, which is particularly robust for different degrees of skewness, and show that it outperforms the distributed hash-join operation. For example, for a medium degree of skewness, SMBJOINSKEWED reduces execution time and network traffic by 34% and 5%, respectively, compared to the distributed hash-join operation after five joins.**

***Index Terms*—Join, Skewed Data, Repeated Join, Distributed Join, Apache Beam**

I. INTRODUCTION

Join is a prominent operation in almost all data analysis applications in data-parallel processing platforms, such as Spark [1], Flink [2], and Google Cloud Dataflow [3]. To execute a join operation in a cluster of computers (workers), each worker holds a certain fraction of data, and by communicating with each other and *shuffling* data, they can produce the join results. Shuffling data dominates the processing time in a distributed implementation of joins. This is mainly due to sending high network traffic and serializing/deserializing data to be transferred over the network.

One use case of join operations is *repeated joins* that naturally arises when multiple users need to compute the same join operation. For example, consider a company that stores the encrypted data of its users and the decryption keys in different tables to comply with the users' privacy regulations. To decrypt the data and access users' information, it is required to join the two tables based on users' ids as the join-keys to find the decryption key of each encrypted record. However, since it is impossible to store the decrypted data, the join operation should be repeated whenever the users' information is required. Given that joining is an expensive task, it is desired to reduce this operation's cost as much as possible. Note that tackling the privacy issue of such join operations is out of the scope of this paper.

Distributed join operations also suffer from *skewed* data, when a large fraction of records in a dataset have a small set of join-keys. In distributed joins, all records with the same join-key should be collected at the same worker so that each worker can perform the join operation locally. However, data skewness can create uneven load distribution among the workers [4]. If data is skewed, then collecting records with similar join-keys on workers can overload a few of them (which are responsible for skewed join-keys) and consequently degrades their performance. These slow workers (*stragglers*) can significantly delay the joint operation's completion time.

In this work, we present the SMBJOIN operation that improves the performance of repeated joins in data-parallel computing platforms, with respect to execution time and network traffic. We also present SMBJOINSKEWED, a variant of SMBJOIN, to address skewed data challenges in repeated joins. We evaluate SMBJOIN and SMBJOINSKEWED on two different types of datasets. First, we test them on real big data collected from a music streaming company's event delivery infrastructure. Compared to the distributed hash-join operation, the results show that SMBJOIN executes faster after two sequences of joins and shuffles fewer data after four repetitions. For example, if the join operation is repeated five times, SMBJOIN reduces execution time and network traffic by 50% and 17%, respectively, compared to the hash-join.

The first dataset, however, does not suffer from skewness under normal operating conditions. Therefore, we synthetically generate a similar dataset with increasing skewness degrees as the second type of dataset. While the performance of hash-join and SMBJOIN degrade quickly with a medium degree of skewness, SMBJOINSKEWED proves to be effective even when a high degree of skewness is present. In such high skewness scenarios, SMBJOINSKEWED performs better than distributed hash-joins even for a single join operation. We implemented our algorithms using Apache Beam [5] over Google Cloud Dataflow [3]; however, our algorithms are not tied to this execution platform and are compatible with any other data-parallel processing platforms, such as Spark [1] or Flink [2].

II. BACKGROUND

In this section, we briefly present the join operation in data-parallel processing platforms, explain the skewness and its impact on joins, and finally describe Apache Beam as the platform we used for the implementations.

A. Join operation

Hash-join is a popular way to implement the join operation that combines two tables R and S into one table, based on matching join-keys. It first creates a hash-table from one of the

tables that maps each join-key to all the records containing it, then scans the second table, and finds matching records in the hash-table. An alternative way to implement joins is the *sort merge-join*, which is a two-step process: (i) sort both tables in the same order, and (ii) merge-join the tables by scanning them in an interleaved fashion while outputting records that satisfy the join condition. If both tables R and S are sorted, the merge-join is the fastest join; otherwise, hash-join shows a better performance.

There are different ways to implement joins in data-parallel processing platforms, such as Spark [1], Flink [2], and Google Cloud Dataflow [3]. The main idea in all of these approaches is to forward records with the same join-key to the same worker and then perform a local join (e.g., hash-join or sort merge-join) algorithm on each worker. Two popular ways to implement distributed joins are *map-side join* and *reduce-side join*. The assumption in the map-side join is that one of the tables is small enough to be loaded in each worker's memory. In this case, the bigger table is split among the workers, while the smaller one is loaded into the memory of all of them. In the reduce-side join, both tables are partitioned according to the join-key using a hash function, and are shuffled over the network, such that all records with the same join-key will end up at the same worker.

B. Skewness

Data skew refers to having a small set of join-keys shared by a large fraction of the records in a data table. This type of skew is applicable for `reduce`-like operations [6], where the records are assigned to workers based on hash partitioning of the join-keys. In data-parallel processing platforms, data skew can lead to uneven distribution of records of a table among workers [4], and consequently overload some of those workers, which are responsible for skewed join-keys. Such situations can severely downgrade the performance of queries, especially in join operations.

C. Apache Beam

Apache Beam [5] is a unified programming model for batch and streaming data processing. The Beam is *execution engine-agnostic*, meaning that jobs written with Beam can be used in any supported *runner*, which is the back-end system to execute that job. The runner is used to abstract away complicated details that are dependent on the underlying engine. The Beam API is reminiscent of FlumeJava [7], a Java library designed to simplify the task of writing MapReduce jobs [6]. Beam jobs are expressed as *pipelines* that represent a Directed Acyclic Graph of steps, i.e., a series of operations (`PTransforms`) on distributed collections (`PCollections`). Data-parallel operations equivalent to `map` in MapReduce [6] are expressed through `ParDo` in Beam, while `reduce`-like operations are expressed through `GroupByKey`. In Beam, the hash-join can be implemented through a `CoGroupByKey` operation, which uses the `GroupByKey` primitive.

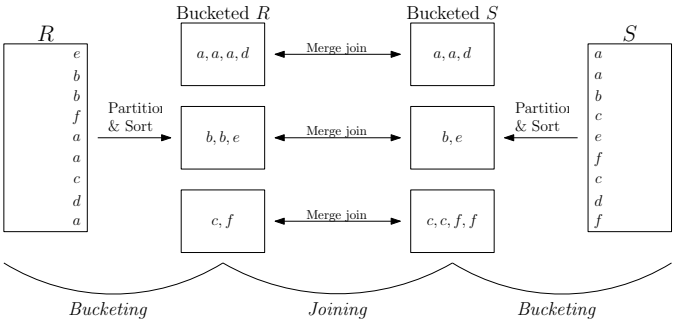


Fig. 1. High level overview of SMBJOIN.

III. SORT MERGE BUCKETS (SMB)

Here, we first explain SMBJOIN, a solution to improve the performance of repeated joins, and then describe SMBJOINSKEWED, a variant of SMBJOIN, to handle skewed data.

A. SMBJOIN

We aim to improve the performance of repeated join operations in data-parallel processing platforms with respect to execution time and network traffic. To this end, we present SMBJOIN, a distributed implementation of the sort merge-join. SMBJOIN breaks the join operation into two phases: (i) the *bucketing* phase to structure the data in such a way that the expensive shuffling step can be avoided by dividing the datasets into several buckets and sorting records in each bucket, and (ii) the *joining* phase to apply the merge-join on the prepared data in the previous phase. The bucketing phase is executed only once for a repeating join; thus, although it adds some overhead to the system, its cost will be amortized after a certain number of joins repetitions.

Figure 1 demonstrates a high-level overview of SMBJOIN. As it shows, two tables R and S are first partitioned and sorted on their join-keys, and then corresponding buckets are joined through a merge-join operation. The letters in Figure 1 represent the values of the join-key. Below, we present these phases in detail.

Bucketing phase. The bucketing phase is composed of two steps: *partitioning* and *sorting*. First, data is partitioned, such that each partition (*bucket*) can be loaded in the memory of each worker, and then the data will be sorted by the join-key. Achieving a global ordering on data is very expensive due to its massive size and data distribution over workers. Therefore, we relax the global ordering requirement by performing a local sorting of records in each worker's buckets. Listing 1 shows the bucketing phase of SMBJOIN, written in Beam.

1) Partitioning (lines 3, 4): here, we use `ParDo` to apply a hash function h on each record of the input table to partition them into B buckets. The bucket-key k_x of each record x is determined as $k_x = h(x.key) \bmod B$, where $x.key$ refers to the join-key of the record x. The partitioning process works by first extracting each record's bucket-key and then grouping records by their

Listing 1: Bucketing phase for SMBJOIN.

```
1 def BucketingPhase(input, B):
      Data: input is a PCollection⟨V⟩, with V denoting the type of the
            records inside the PCollection. B is the number of buckets.
      Result: PCollection⟨Bucket⟨V⟩⟩
2     return input
3         .apply(ParDo.of(ExtractBucketKeyFn(B)))
          /* Returns a PCollection⟨K, V⟩, where the key K is
             the bucket-key determined by applying the
             hash function h on records' keys.          */
4         .apply(GroupByKey.create())
          /* Returns a PCollection⟨K, Iterable⟨V⟩⟩       */
5         .apply(ParDo.of(SortBucketFns()))
          /* Locally sorts the records in an iterable by
             their join-key, creating a Bucket⟨V⟩ and
             returning a PCollection⟨Bucket⟨V⟩⟩.         */
```

Listing 2: Joining phase for SMBJOIN.

```
1 def JoiningPhase(Path_R, Path_S):
      Data: Path_R and Path_S represent the location where the bucketed
            data of R and S is stored.
      Result: PCollection⟨V_R, V_S⟩, with V_R and V_S represent the records of R
              and S, respectively.
2     return PBegin
3         .apply(ParDo.of(ResolveBucketingFn(Path_R, Path_S)))
          /* Returns a PCollection⟨Bucket⟨V_R⟩, Bucket⟨V_S⟩⟩,
             with each tuple represents a matching pair
             of buckets. Two buckets match, if they have
             the same bucket-key. The records inside each
             bucket are not read yet.                   */
4         .apply(ParDo.of(MergeJoinFn()))
          /* Performs a merge-join over each pair of
             buckets, returning a PCollection⟨V_R, V_S⟩.  */
```

bucket-keys into an iterable, through a `GroupByKey`. The `GroupByKey` operation in this step is the only shuffle operation (that transfers data over the network) in the whole SMBJOIN pipeline.

2) Sorting (line 5): the previous step's output is a `PCollection` of buckets. Each bucket in the `PCollection` is located only on one worker; hence, it is possible to perform a local sort of each bucket in parallel. Here, we use the merge-sort [8] to sort records of each bucket.

At the end of the bucketing phase, each bucket's content is written to a file with its metadata that includes the bucket-key, the total number of buckets, the hash function h, and the join-keys.

Joining phase. In the second phase, SMBJOIN joins the two bucketed-sorted tables. Two bucketed tables can be joined if they are *compatible*. Two tables are compatible if (i) they use the same join-key for bucketing and sorting the records, (ii) they use the same hash function to ensure the same join-keys are hashed to the same bucket, and (iii) they both have the same number of buckets to ensure that the bucket-keys of two records with the same join-key in different tables are the same. The joining phase, which is presented in Listing 2, also consists of two steps: *bucketing resolution* and *merge-join*:

1) Bucketing resolution (line 3): this step ensures that the input tables are compatible, and it also generates the corresponding pairs of buckets. It iterates through the lists of metadata of two tables, R and S, to determine their compatibility and find pairs of matching buckets.
2) Merge-join (line 4): each pair of buckets are joined using a merge-join operation. When two tables are compatible, the result of the merge-join will be correct. This is because records that have the same join-key are assigned to the buckets with the same bucket-key.

One of the important parameters in SMBJOIN is the number of buckets B. If $B = 1$, then all the records will be grouped in a single bucket and on a single worker, which is unfeasible for big datasets that do not fit in memory. A practical approach to determine the number of buckets is to pick a number such that each bucket fits in the memory of one worker. However, determining the number of buckets presents some issues:

- It is necessary to know the size of the input data in advance. This can be estimated or can be computed as another Dataflow job.
- To determine if a bucket fits in workers' memory, runner-specific information such as the number of workers and their available memory is needed. However, any particular awareness of these configurations should be seen as a failure from the perspective of determining a framework-agnostic solution.
- An approach that simply divides the total dataset size over the available memory assumes that buckets contain an equal share of records. However, such a distribution is unlikely due to skewness in datasets.

B. SMBJOINSKEWED

Among the aforementioned challenges, skewness is a major problem in data-parallel processing platforms. In particular, when data is skewed, the bucketing phase of SMBJOIN is affected due to the `GroupByKey` operation. In addition to skewness, an additional issue lies in the notion of dataset compatibility, which limits the flexibility of joining two tables. To overcome these challenges, we present SMBJOINSKEWED, a variant of SMBJOIN, that adjusts the bucketing and joining phases as follows.

Bucketing phase. Unlike SMBJOIN that requires the number of buckets, in SMBJOINSKEWED we set the size of the buckets b. The number of buckets is calculated as the size of a table R divided by the bucket size b, e.g., $B_R = \frac{|R|}{b}$, where $|.|$ indicates the size of a table. When creating buckets, if the size of the records in a bucket exceeds the size of buckets, b, then the bucket is *skewed*. In this case, the bucket should be divided into multiple parts, called *shards*, such that each shard has the maximum size b. The records of the original skewed bucket are then distributed uniformly among the new shards (e.g., by using a round-robin strategy).

According to these changes, we modify the bucketing phase by first computing the number of buckets (given the bucket size b) and then computing the number of shards for each bucket. To prevent the skewness in `GroupByKey` operation, instead of using bucket-key as the key, we put shard-id together with

Listing 3: Bucketing phase for skew-adjusted SMB.

```
1  def BucketingPhase(input, b):
       Data: input is a PCollection⟨V⟩, with V denoting the type of the
             records inside the PCollection. b is the target bucket size.
       Result: PCollection⟨Shard⟨V⟩⟩
2      B ← input
3          .apply(ParDo.of(ComputeSizeFn()))
4          .apply(Sum.globally())
5          .apply(ParDo.of(ComputeNumBucketsFn(b)))
           /* Returns the number of buckets in B          */
6      M ← input
7          .apply(ParDo.of(ComputeSizeWithBucketKeyFn(B)))
8          .apply(Sum.perKey())
9          .apply(ParDo.of(ComputeNumShardsFn(b)))
           /* Returns the number of shards in M           */
10     return input
11         .apply(ParDo.of(ExtractShardedBucketKeyFn(B, M)))
           /* Returns a PCollection⟨K, V⟩, where the key K is
              a tuple of (bucket-key, shard-id).          */
12         .apply(GroupByKey.create())
           /* Returns a PCollection⟨K, Iterable⟨V⟩⟩        */
13         .apply(ParDo.of(SortBucketFns()))
           /* Locally sorts the records in an iterable by
              their join-key, creating a Shard⟨V⟩ and
              returning a PCollection⟨Shard⟨V⟩⟩.           */
```

bucket-key and use a tuple-key (bucket-key, shard-id) instead. This allows the `GroupByKey` operation and sorting to not suffer from skewness. As before, each shard is written out as a single file with all its metadata, including its (bucket-key, shard-id) tuple. Due to sharding, a bucket may span multiple files.

Listing 3 contains the updated code of the bucketing phase for SMBJOINSKEWED. Some boilerplate code has been removed for brevity: in particular, the number of buckets B and shard map M, which contains the number of shards for each bucket, are broadcasted to all workers.

Joining phase. After making shards, we should join the corresponding pairs. Recall that two bucketed tables can be joined if they are compatible, i.e., have the same number of buckets. However, the bucketing phase of SMBJOINSKEWED may violate the compatibility condition. Assume two tables R and S with different sizes, and the bucket size b, then the tables have $B_R = \frac{|R|}{b}$ and $B_S \frac{|S|}{b}$ buckets, respectively. As the two tables have different numbers of buckets, they are not compatible, and therefore there is no correspondence between bucket-keys of the two tables.

A naive approach to join tables with different numbers of buckets is to compute the merge-join between all pairs of buckets. Although the result of this join is correct (as all pairs of buckets are joined, and hence all matching records would be joined), it is an expensive approach. This is because each bucket of a table S is replicated for each bucket of the other table R, meaning that in total, it computes $B_R \times B_S$ merge-joins, which is not efficient.

However, it is not necessary to compute the merge-join for all pairs of buckets. Recall that for each record x with the join-key $x.key$, its bucket-key is $k_x = h(x.key) \bmod B$, where B is the number of buckets. Moreover, considering the *distributive property* of the modulo operation, for integers

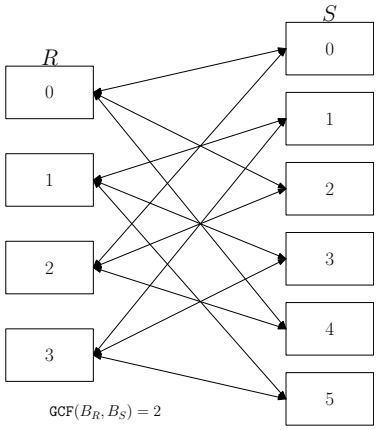


Fig. 2. Joining two tables with different numbers of buckets ($c = 2$, $h_R = 2$, and $h_S = 3$).

x, i, j we have $(x \bmod ij) \bmod i = x \bmod i$. Therefore, if we pick integers $c, h \in \mathbb{N}$ such that $B = c \times h$, then by using the distributive property, we have:

$$\begin{aligned} h(x.key) \bmod B &= k_x \\ (h(x.key) \bmod B) \bmod c &= k_x \bmod c \\ (h(x.key) \bmod ch) \bmod c &= k_x \bmod c \\ h(x.key) \bmod c &= k_x \bmod c. \end{aligned}$$

In other words, we can treat the tables as having been bucketed in c buckets, each of which is composed of h of the original B buckets, without actually having to re-bucket the data. This property can be used to significantly reduce the number of merge-joins by picking c such that $B_R = c \times h_R$ and $B_S = c \times h_S$, and emitting pairs whose bucket keys modulo c are the same. In this case, the total number of merge-joins computed is $\frac{B_R \times B_S}{c}$. If $c = 1$, this ends up computing the join over all pairs of buckets, and if the two tables have the same number of buckets, i.e., $c = B_R = B_S$, no replication is required. If we pick c as the greatest common factor (GCF) of B_R and B_S, then the number of computed joins will be minimum. With this adjustment, we do not need to define the number of buckets as we did in SMBJOIN.

As mentioned before, the size of buckets b should be selected such that each bucket fits in the memory of a worker. Given such a bucket size b, then the number of buckets of two tables R and S will be $B_R = \frac{|R|}{b}$ and $B_S = \frac{|S|}{b}$, respectively. The amount of replicated data for joining R and S, then depends on $c = \texttt{GCF}(B_R, B_S)$. The lower is c, the higher is the cost of replication. Figure 2 shows an example, where $c = \texttt{GCF}(4, 6) = 2$, $h_R = 2$, and $h_S = 3$.

After replicating the buckets (including their shards), we can join the two tables. If a bucket is sharded, to join two buckets, a merge-join between all pairs of underlying shards is computed instead. Therefore, the `ResolveBucketingFn` (in Listing 2) may end up with creating more join operations as a result of sharding. Each shard contains its bucket-key and is joined with all other shards with a matching bucket-key (modulo c if the number of buckets is different). The joining phase code is

then unchanged apart from the described changes inside the `ResolveBucketingFn` function.

IV. Experiments

This section presents the performance evaluation of SMBJoin and SMBJoinSkewed operations and compares them with the distributed hash-join operation. First, we describe the experimental setup and then present the comparison results for a pipeline scenario using two data sets: (i) a real non-skewed dataset collected from a music streaming company, and (ii) a synthetic data set with different skewness degrees.

A. Experimental Setup

We used Apache Beam [5] for the development and conducted the experiments on Google Cloud Dataflow [3] (as a runner of the Beam). We also use clusters of $\texttt{n1} - \texttt{standard} - 4$ workers, where each worker has four CPUs and 15 GB of memory. We define the following three metrics to compare the performance of the join algorithms (implemented as Beam pipelines):

1) Wall clock time: the total time to run a pipeline.
2) CPU hours: the number of CPU hours allocated to a pipeline. Since the wall clock is dependent on the number of workers, we define the CPU hours that shows the computing cost of running a pipeline.
3) Network traffic: the amount of shuffling data in GB.

We evaluate the above metrics in the following three algorithms:

- Hash-join: this is the standard join operation in Beam, which is implemented as a reduce-side join.
- SMBJoin: this is a distributed implementation of sort-merge join explained in Section III.
- SMBJoinSkewed: similar to SMBJoin, but takes into account the skewness cases.

To evaluate the algorithms, we consider the use case of the decryption of user data generated in user behavior experiments of a music streaming company. In this scenario, each interaction of users with the player application generates an event. There are two tables:

1) `Events`: each record of this table describes a particular event happening in a user experiment. The event information is stored encrypted in this table to keep the privacy of the users. Each event is identified with a user id, which is used as the two tables' join-key.
2) `Keys`: this table includes all the keys to decrypt the encrypted records of the `Events` table. All the `Events` table's records have a matching join-key in the `Keys` table.

To access the values of the `Events` table, we should join it with the `Keys` table to find their decryption keys. However, data cannot be stored unencrypted on disk, and since the decryption key cannot be kept along with the encrypted data, the tables should be joined before accessing their data. It means that the join operation should be repeated for each access to the values of events.

TABLE I
Comparison of joins on the decryption pipeline.

Baseline	Wall Clock	CPU Hours	Shuffled Data GB
$\texttt{Events} \bowtie \texttt{Keys}$	0:45:53	349.42	6576
SMBJoin	**Wall Clock**	**CPU Hours**	**Shuffled Data GB**
Bucketing `Keys`	0:15:00	23.644	845
Bucketing `Events`	0:53:57	423.159	11810
$\texttt{Events} \bowtie^{\text{SMBJoin}} \texttt{Keys}$	0:19:16	72.767	2974

B. Real Non-skewed Data

Here, we executed the experiments on 128 workers, with 512 CPUs and 1880 GB of memory, and used a real data set collected at a music streaming company. In this case, the `Events` and `Keys` tables are as below:

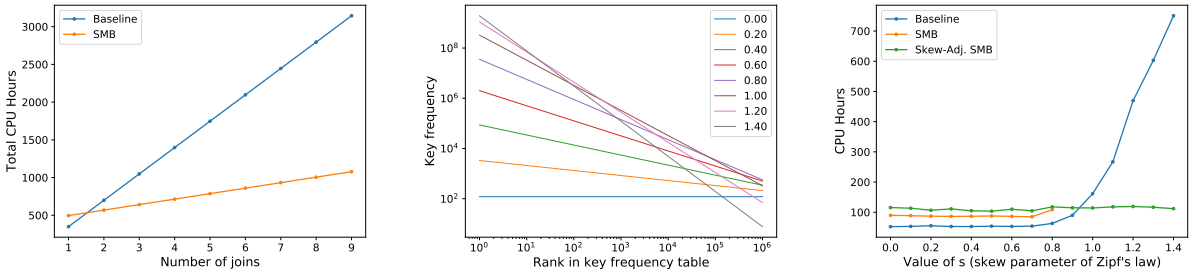
- `Events`: 6.7×10^9 records for a total serialized size of 2.86 TB. Each record describes a particular event happening in a user experiment.
- `Keys`: 1.15×10^9 records for a total serialized size of 194.87 GB. Each record contains an encryption key for a specific user.

In SMBJoin, we create 8096 buckets of approximately 1 GB each. As each worker has four CPUs and 15 GB memory, considering 1 GB for each bucket is reasonable to load four buckets in memory simultaneously (each bucket is assigned to one CPU). To pick the number of buckets, we consider $1.4\times$ overhead for deserializing data and $2\times$ overhead for sorting in `PTransform`. Given the size of the serialized `Events` (i.e., 2.86 TB), we need around 8000 GB (i.e., $2.86 \times 1.4 \times 2$) for the deserialized data in memory. That is why we selected 8096 buckets.

The first block in Table I shows the result of executing the distributed hash-join in Beam over Google Cloud Dataflow. The amount of shuffled data corresponds to reading the data and performing a `CoGroupByKey` operation. The results of SMBJoin are represented in the second block of Table I. These values are shown for the bucketing and the joining phases separately. For the bucketing phase, the shuffled data includes reading data, creating buckets (using `GroupByKey`), and writing the buckets. In the join phase, the operation reads each pair of buckets, joins them, and decrypts the user field. As expected, the join phase of SMBJoin is faster than the hash-join as no shuffling is required. However, the bucketing phase is an expensive task, but this phase happens only once for any number of joins.

We can compute the number of joins n after which SMBJoin has a better performance than hash-joins as $\texttt{Bucketing} + \texttt{MergeJoin} \times n \leq \texttt{HashJoin} \times n$, where `Bucketing` and `MergeJoin` show the cost of bucketing and joining phases on SMBJoin for a particular metric (e.g., CPU Hours), respectively, and `HashJoin` shows the hash-join cost for the same metric. So, we can define

$$n \geq \left\lceil \frac{\texttt{Bucketing}}{\texttt{Join} - \texttt{SMBJoin}} \right\rceil. \tag{1}$$



(a) (b) (c)

Fig. 3. (a) the total CPU hours after n joins for hash-join and SMBJOIN, (b) the frequency distribution for the first million most frequent keys in generated `Event` for different values of s, and (c) the comparison of CPU hours of SMBJOIN (in orange, bucketing and joining), SMBJOINSKEWED (in green, bucketing and joining), and distributed hash-join (in blue) for different values of s.

By solving it for CPU hours, we obtain $n = 2$, meaning that after the second SMBJOIN, the cost of bucketing has been amortized. Figure 3(a) illustrates it. The same applies to the amount of shuffled data, which becomes less after $n = 4$ joins.

C. Synthetic Skewed Data

In order to evaluate SMBJOINSKEWED, we repeat the analysis for synthetic datasets with different degrees of skewness. To do so, we generate data with join-keys following Zipf's law. This Zipf's law states that the frequency of a join-key is inversely proportional to its rank, i.e., the ith key has frequency $\frac{1}{i^s}$ times the most frequent key, for some shape parameter s. The higher is s, the more skewed are the frequencies.

Here, `Keys` is approximately 50 GB, and the `Events` is approximately 640 GB. In order to test the effect of skewness, we generate the `Event` tables 15 times with different degrees of skewness, by choosing $s \in [0.0, 1.4]$. Figure 3(b) plots the frequency of the first million most frequent keys for a subset of the values s. When $s = 0$ all keys have an equal frequency of 120. For $s = 0.2$, the first five keys have frequencies $[3327, 2896, 2670, 2521, 2411]$. As s increases, the first key has a frequency of 86482 for $s = 0.4$, 1999427 (approximately two millions) for $s = 0.6$, 35534777 (approximately 36 millions) for $s = 0.8$, up to 1933322357 (approximately two billions, roughly one third of the dataset) for $s = 1.4$.

To do this experiment, we executed the join pipelines on 32 workers with a total of 128 CPUs and 480 GB of memory. Here, we evaluate the performance of SMBJOIN and SMBJOINSKEWED against the hash-join for different join-key distributions. As Table II shows, the hash-join performance appears to be approximately constant until $s > 0.7$. As the data becomes more and more skewed, we observe a steep increase in the time spent for executing the pipeline's job. By inspecting the Google Cloud Dataflow graph, we figure out the reason for slowing down the pipeline process is appearing a straggler (i.e., an overloaded worker). Out of the 10^9 unique join keys, a worker is stuck in the `GroupByKey` grouping all the data for the last skewed key, whereas the remaining 999999999 keys have already been processed. The amount of shuffled data remains approximately constant. GBK in the table refers to the traffic generated by `GroupByKey`.

TABLE II
HASH-JOIN BETWEEN `Keys` AND `Events` FOR DIFFERENT VALUES OF s.

`Events` ⋈ `Keys` **Repartition Join** for diff. values of s	**Wall Clock**	**CPU Hours**	**Shuffled Data GB** (Read + GBK)
0.0	0 : 27 : 12	52.343	1485.22
0.1	0 : 27 : 27	53.175	1484.15
0.2	0 : 28 : 30	55.356	1483.98
0.3	0 : 27 : 33	52.790	1483.48
0.4	0 : 27 : 24	52.739	1482.78
0.5	0 : 27 : 56	53.845	1481.83
0.6	0 : 27 : 26	53.006	1480.49
0.7	0 : 28 : 12	54.059	1478.47
0.8	0 : 32 : 16	63.061	1475.36
0.9	0 : 44 : 19	90.093	1465.99
1.0	1 : 18 : 00	161.499	1459.73
1.1	2 : 09 : 00	267.030	1442.35
1.2	3 : 44 : 00	469.501	1436.95
1.3	4 : 47 : 00	602.886	1435.39
1.4	5 : 58 : 00	750.614	1429.68

Table III and Table IV show the performance of SMBJOIN for the bucketing and joining phases, respectively. The pipelines for $s = 0.9$ and $s = 1.0$ fail in the bucketing step due to lack of progress when sorting. The workers responsible for sorting the skewed buckets are straggling and cannot output any records for over 30 minutes, which causes a failure in Google Cloud Dataflow. Figure 3(c) plots the total CPU hours of a single hash-join (in Table II) in blue and a single SMBJOIN in orange.

We observe that SMBJOIN is more advantageous than hash-join in terms of CPU hours starting with the second SMBJOIN ($n = 2$), for all values of s. Note that in this scenario we do not include the bucketing step for `Keys`, as that happens only once for all 15 values of s. It can also happen in parallel as bucketing of the `Events`, in addition to requiring less than one 10th of the time. Repeating the same analysis for the amount of network traffic, we observe that SMBJOIN shuffles fewer data after the fourth join ($n = 4$) for all s. In summary, this means that SMBJOIN has better performance than hash-join after performing two joins and shuffles fewer data after four joins. However, this join is not robust to higher degrees of skewness as some tested scenarios fail.

In the last experiment, we measure the performance of SMBJOINSKEWED. Here, instead of picking the number of

TABLE III
BUCKETING PHASE OF SMBJOIN FOR KEYS AND EVENTS.

Bucketing `Keys`	**Wall Clock**	**CPU Hours**	**Shuffled Data GB** (Read + GBK + Write)
n/a	0 : 16 : 13	7.271	205.33
Bucketing `Events` for diff. values of s	**Wall Clock**	**CPU Hours**	**Shuffled Data GB** (Read + GBK + Write)
0.0	0 : 40 : 04	77.496	2595.42
0.1	0 : 39 : 25	76.100	2593.58
0.2	0 : 38 : 42	74.846	2593.50
0.3	0 : 38 : 56	74.683	2592.87
0.4	0 : 38 : 53	75.697	2591.87
0.5	0 : 39 : 03	76.454	2590.48
0.6	0 : 38 : 21	74.901	2588.48
0.7	0 : 38 : 08	73.871	2585.38
0.8	0 : 48 : 50	96.901	2580.68
0.9	*failed*	*failed*	*failed*
1.0	*failed*	*failed*	*failed*

TABLE IV
JOINING PHASE OF SMBJOIN FOR KEYS AND EVENTS.

`Events` ⋈ `Keys` **SMB Join** for diff. values of s	**Wall Clock**	**CPU Hours**	**Shuffled Data GB** (Read)
0.0	0 : 08 : 41	12.368	620.31
0.1	0 : 09 : 07	12.482	619.84
0.2	0 : 08 : 50	12.148	619.75
0.3	0 : 08 : 34	11.692	619.52
0.4	0 : 08 : 08	10.944	619.18
0.5	0 : 08 : 12	11.204	618.71
0.6	0 : 08 : 00	11.531	618.05
0.7	0 : 08 : 05	11.056	617.01
0.8	0 : 08 : 28	11.711	615.46
0.9	*n/a*	*n/a*	*n/a*
1.0	*n/a*	*n/a*	*n/a*

buckets, we set a bucket size of 300 MB in order for each worker to have ample memory to sort it. If the size of a bucket exceeds 300 MB, it will be split into several shards, such that the size of each shard does not go beyond 300 MB. Table V and Table VI show the performance of bucketing and joining of SMBJOINSKEWED, respectively. Note that the additional column in Table V represents the total number of shards written out, and the last column in Table VI represents the value of n (Formula 1) for CPU hours and shuffled data, respectively.

Compared with hash-joins for $s < 0.8$, SMBJOINSKEWED shows a better performance in terms of CPU hours after three joins and shuffles less data after five. As skewness quickly increases, SMBJOINSKEWED uses fewer CPU hours than a single hash-join for $s \geq 1.0$, as can be seen in Figure 3(c). This advantage in processing time has a trade-off in amounts of data shuffled: as data becomes more skewed, SMBJOINSKEWED replicates more and more shards. For the highest degree of skew tested, it still shuffles fewer data after 10 joins.

When compared with SMBJOIN, for $s < 0.8$, the bucketing operation has an overhead of approximately 25%. For $s = 0.8$, the bucketing takes the same amount of CPU hours. For $s > 0.8$, unlike SMBJOIN, SMBJOINSKEWED handles all degrees of skewness tested. SMBJOINSKEWED also relaxes the notion of compatibility to allow joining with datasets with different numbers of buckets.

According to the results shown, we can combine SMBJOIN and SMBJOINSKEWED to optimize different join scenarios based on different degrees of skew:

- Low skew ($0 \leq s < 0.8$): if joining more than two times, SMBJOIN has a better performance than a hash-join.

TABLE V
BUCKETING PHASE OF SMBJOINSKEWED FOR KEYS AND EVENTS.

Bucketing `Keys`	**Wall Clock**	**CPU Hours**	**Shuffled Data GB** (Read + GBK + Write)	**Shards**
n/a	0 : 23 : 59	11.275	206.83	512
Bucketing `Events` for diff. values of s	**Wall Clock**	**CPU Hours**	**Shuffled Data GB** (Read + GBK + Write)	**Shards**
0.0	0 : 48 : 22	97.423	2606.94	4096
0.1	0 : 47 : 07	95.113	2605.09	4096
0.2	0 : 44 : 05	88.211	2605.04	4096
0.3	0 : 46 : 27	93.051	2604.38	4097
0.4	0 : 43 : 03	86.183	2603.39	4105
0.5	0 : 42 : 28	85.178	2602.03	4200
0.6	0 : 45 : 21	90.889	2600.34	4484
0.7	0 : 42 : 37	85.222	2597.00	4774
0.8	0 : 48 : 17	97.524	2591.98	4927
0.9	0 : 47 : 02	93.961	2581.75	5227
1.0	0 : 46 : 00	92.573	2572.81	5691
1.1	0 : 48 : 37	95.683	2563.15	6359
1.2	0 : 49 : 11	96.485	2539.17	6896
1.3	0 : 48 : 16	93.908	2538.45	7293
1.4	0 : 46 : 47	89.342	2540.40	7528

TABLE VI
JOINING PHASE OF SMBJOINSKEWED FOR KEYS AND EVENTS.

`Events` ⋈ `Keys` **Skew-Adj. SMB Join** for diff. values of s	**Wall Clock**	**CPU Hours**	**Shuffled Data GB** (Replicated Read)	n (CPU, IO)
0.0	0 : 12 : 14	18.194	899.75	(3, 5)
0.1	0 : 12 : 04	18.299	899.28	(3, 5)
0.2	0 : 12 : 11	18.354	899.19	(3, 5)
0.3	0 : 12 : 24	18.062	899.04	(3, 5)
0.4	0 : 12 : 21	18.741	899.32	(3, 5)
0.5	0 : 12 : 30	18.282	906.26	(3, 5)
0.6	0 : 13 : 05	19.209	927.74	(3, 5)
0.7	0 : 12 : 48	19.409	949.31	(3, 5)
0.8	0 : 13 : 27	20.293	959.69	(3, 6)
0.9	0 : 13 : 34	20.866	981.48	(2, 6)
1.0	0 : 13 : 49	21.654	1014.86	(1, 6)
1.1	0 : 18 : 28	22.171	1064.82	(1, 7)
1.2	0 : 17 : 36	22.527	1098.79	(1, 8)
1.3	0 : 17 : 26	22.777	1128.53	(1, 9)
1.4	0 : 17 : 48	22.522	1152.54	(1, 10)

- Medium skew ($0.8 \leq s < 1.0$): if joining more than three times, SMBJOINSKEWED has a better performance than a hash-join.
- High skew ($s \geq 1.0$): regardless of the number of joins, SMBJOINSKEWED has a better performance than the others.

V. RELATED WORK

There exists a rich study on computing joins in a distributed environment, both in practice and theory. For example, [9] presents a solution for scalable online joins, [10] demonstrates a multi-round distributed join algorithm, [11] presents distributed joins using MapReduce, and [12] describes distributed joins on multiple cores, while [13] shows a formal analysis of join processing in parallel systems, [14] analysis the worst-case optimal algorithms for parallel query processing, and [15] computes a worst-case optimal multi-round algorithm for parallel computation of conjunctive queries.

There are also many works on resolving the problem of skewed data in distributed joins. Early work by Lin et al. [16] shows that skewed data distributions lead to the creation of stragglers in parallel processing. They argue that this distribution can arise in input data and intermediate data, imposing a limit on the degree of parallelism of certain tasks, such as joining tables. To tackle this problem, Afrati and Ullman [17] present an optimizing join for skewed data. They propose a

mechanism based on replicating data and performing the join in a single MapReduce job.

Kwon et al. [4] present a general overview of skewness in MapReduce. They suggest a series of techniques to mitigate skewness, such as pre-aggregating data through combiners after the map phase or collecting properties of the data in previous processing before MapReduce to use different partitioning strategies. With SkewTune [18], the authors present a drop-in replacement for the MapReduce implementation that automatically mitigates skew. At runtime, SkewTune determines stragglers and automatically repartitions data for which stragglers are responsible through range partitioning. Two other skewss-resilient join solutions are FastJoin [19] and ScaleJoin [20]. The former solution proposes an algorithm to find out the skewed keys and a tuple migration strategy to solve the load imbalance problem. The latter solution enables a deterministic and disjoint-parallel join on skewed data.

While the previous skew-handling solutions work at runtime by monitoring pipeline progress and mitigating stragglers, [21] proposes using proactive cloning, through which tasks are replicated similarly to the previous work, and the first available result for each clone group is used. In order to mitigate the problem, they introduce delay assignment as a hybrid approach, in which downstream consumers first wait a certain amount of time before reading intermediate data to get an exclusive copy. After that time, reading with contention is used. Note, however, it has some limitations: computing the cluster-wide straggler probability is not always possible, e.g., when using a managed service, and the solution modified the Hadoop implementation, which does not make it applicable in a broader scope. A similar approach to shard cloning is used in our SMBJOINSKEWED algorithm.

VI. CONCLUSION

In this work, we have presented SMBJOIN and SMBJOINSKEWED, two algorithms to optimize repeated and skewed joins at a massive scale. Both SMBJOIN and SMBJOINSKEWED consist of two steps: (i) bucketing step to reorganize the data, and (ii) joining step that exploits the structure of the data for efficient merge-join. Compared to reduce-side distributed hash-join, these joins have the bucketing step extra. However, since the joins operations considered in this work are repeated joins, our proposed solutions show better performance for joins repeated multiple times.

In addition to repeated joins, we tackle the skewed data problem, where a large fraction of data has a small set of join-keys. To support this kind of data, we propose SMBJOINSKEWED, a variant of SMBJOIN, robust to varying degrees of skewness. To mitigate the effects of skewed data, SMBJOINSKEWED splits skewed buckets into several shards and replicates them to have a broader versatility in joining tables that are split into different numbers of buckets.

When data is not skewed, SMBJOINSKEWED has more overhead than SMBJOIN, but it is still more compute-efficient than hash-join after a couple of join operations. Moreover, SMBJOINSKEWED is robust to fluctuating data size and skew and has a lower barrier of entry: the programmer does not need to estimate properties of the data or understand how the number of buckets ties into the inner workings of the procedure. We conducted the experiments using Apache Beam running on Google Cloud Dataflow. As no runner-specific details are used, this solution is broadly applicable to all of Beam's runners.

REFERENCES

[1] M. Zaharia et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 15–28.

[2] P. Carbone et al., "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[3] S. Krishnan et al., "Google cloud dataflow," in *Building Your Next Big Thing with Google Cloud Platform*. Springer, 2015, pp. 255–275.

[4] Y. Kwon et al., "A study of skew in mapreduce applications," *Open Cirrus Summit*, vol. 11, no. 8, 2011.

[5] *Apache Beam: An advanced unified programming model*, (accessed May 9, 2020). [Online]. Available: https://beam.apache.org

[6] J. Dean et al., "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] C. Chambers et al., "Flumejava: easy, efficient data-parallel pipelines," *ACM Sigplan Notices*, vol. 45, no. 6, pp. 363–375, 2010.

[8] R. Cole, "Parallel merge sort," *SIAM Journal on Computing*, vol. 17, no. 4, pp. 770–785, 1988.

[9] M. Elseidy et al., "Scalable and adaptive online joins." VLDB, 2014.

[10] F. Afrati et al., "Gym: A multiround distributed join algorithm," in *20th International Conference on Database Theory (ICDT 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[11] A. Okcan et al., "Processing theta-joins using mapreduce," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011, pp. 949–960.

[12] C. Barthels et al., "Distributed join algorithms on thousands of cores," *Proceedings of the VLDB Endowment*, vol. 10, no. 5, pp. 517–528, 2017.

[13] P. Koutris et al., "A guide to formal analysis of join processing in massively parallel systems," *ACM SIGMOD Record*, vol. 45, no. 4, pp. 18–27, 2017.

[14] ——, "Worst-case optimal algorithms for parallel query processing," in *19th International Conference on Database Theory (ICDT 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[15] B. Ketsman et al., "A worst-case optimal multi-round algorithm for parallel computation of conjunctive queries," in *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2017, pp. 417–428.

[16] J. Lin et al., "The curse of zipf and limits to parallelization: An look at the stragglers problem in mapreduce." in *LSDS-IR@ SIGIR*, 2009.

[17] F. Afrati et al., "Optimizing joins in a map-reduce environment," in *Proceedings of the 13th International Conference on Extending Database Technology*, 2010, pp. 99–110.

[18] Y. Kwon et al., "Skewtune: mitigating skew in mapreduce applications," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012, pp. 25–36.

[19] S. Zhou et al., "Fastjoin: A skewness-aware distributed stream join system," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 1042–1052.

[20] V. Gulisano et al., "Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join," *IEEE Transactions on Big Data*, 2016.

[21] G. Ananthanarayanan et al., "Effective straggler mitigation: Attack of the clones," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 185–198.