IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 17 December 2024, accepted 18 February 2025, date of publication 3 March 2025, date of current version 11 March 2025.

*Digital Object Identifier 10.1109/ACCESS.2025.3547623*

RESEARCH ARTICLE

# Accelerating Distributed Repartition Joins on Skewed Datasets via Patch-Based Shuffling

**EVDOKIA KASSELA[1], IOANNIS KONSTANTINOU[2], AND NECTARIOS KOZIRIS[1], (Member, IEEE)**
[1]School of Electrical and Computer Engineering, National Technical University of Athens, 115 27 Athens, Greece
[2]Department of Informatics and Telecommunications, University of Thessaly, 351 00 Lamia, Greece

Corresponding author: Evdokia Kassela (evie@cslab.ece.ntua.gr)

This work was supported by European Union and Greek National Funds through the Operational Program "Competitiveness, Entrepreneurship and Innovation," under the Call RESEARCH—CREATE—INNOVATE under Project T1EDK-04605.

**ABSTRACT** In distributed workloads involving joins and aggregations, skewed attribute values often cause load balancing issues, leading to stragglers and increased execution times. Existing solutions often rely on cost-based models, require extensive parameter tuning, or necessitate modifications to distributed execution engines, limiting their usability and generality. To address this challenge, we present a novel patch-based repartitioning algorithm that eliminates load imbalances while minimizing network overhead. Our approach extends the subset-replicate technique by leveraging data distribution and location statistics to optimize data locality and reduce unnecessary data movement. Unlike traditional hash-based methods, our technique is skew-insensitive, requires no parameter tuning, and integrates seamlessly into existing distributed execution engines as a drop-in replacement for the shuffle mechanism. The proposed method operates in three distinct stages: (1) statistics collection and load threshold computation, (2) patch-based subgroup assignment to ensure optimal load balancing with minimal replication, and (3) informed data shuffling and join execution. This structured process ensures even workload distribution across workers while reducing network I/O. Theoretical analysis proves the scalability, skew robustness, and load-balancing guarantees of our approach, establishing bounds on maximum worker load and network data movement. Experimental evaluations demonstrate that our method achieves perfectly balanced workloads and reduces execution time by up to 81% compared to conventional hash-based joins under moderate to high skew, while introducing negligible overhead at low skew levels. These improvements are attributed to reduced data movement, optimized worker utilization, and the algorithm's robust theoretical foundation. Our research provides a versatile and practical solution for skewed data processing, significantly advancing the efficiency of distributed data management systems.

**INDEX TERMS** Load balancing, repartition join, shuffling, subset-replicate.

## I. INTRODUCTION

A typical organization/institute nowadays keeps many hundreds of GBs of data distributed in various datastores depending on their source (e.g. IoT devices [1], [2] or transactional systems [3]), format (e.g. XML, CSV) and processing capabilities [4], [5]. The most common cases include limited structured data stored in relational databases that are used to perform simple analytics, and large-scale semi-structured or unstructured data stored in NoSQL databases and distributed file systems on top of which big data analytics is performed, such as machine learning pipelines [6], [7], log processing [8], federated learning [9], SQL, etc. Regardless of the data scale and structure, processing engines exist that allow users to interact with them using typical SQL data management programming APIs. The most prominent examples of such engines are Apache Spark [10], Apache Hive [11], Presto [12] and its newer alternative Trino [13] which are autonomous distributed processing engines.

The associate editor coordinating the review of this manuscript and approving it for publication was Dominik Strzalka.

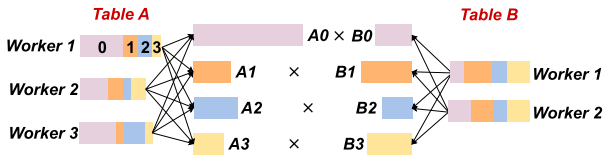



**FIGURE 1.** Approach 1: Standard repartition join and the effect of skew presence in a single key (pink).

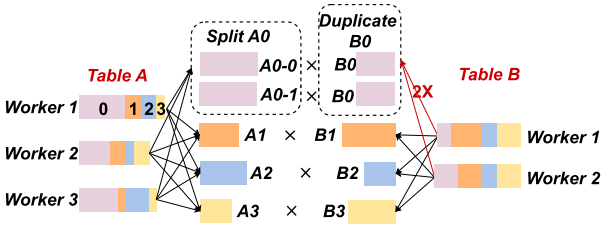



**FIGURE 2.** Approach 2: Subset-replicate join splits and replicates the pink key.

In the context of SQL analytics, when trying to efficiently join two or more datasets, most of the existing SQL engines focus on exploiting data locality profits to minimize network traffic. They achieve this by allowing the query optimizer to choose the appropriate join operator while the task scheduler aims to achieve a fine load balancing for the workers by distributing tasks to them.

The most common algorithm used by distributed execution engines for joining two large datasets (equi-join) is the standard repartition join (Alg. 3.1 in [14]) which is based on the typical MapReduce paradigm [15]. Repartitioning on both tables will happen based on the joining attribute using a hash function, therefore called hash-based repartitioning. With this repartitioning methodology, the different values of the join attribute are assigned to different workers. However, if some values are highly skewed, certain workers will inevitably present a significantly higher load. Considering a join group as the set of input records with a specific join attribute value, a group that corresponds to a value with higher frequency will have a significantly larger size. Such a group will be assigned to a single worker using the default hash-based repartitioning which will overload this worker. The impact of the load imbalance in such cases is observed in the increased overall execution time.

### A. FIRST APPROACH USING STANDARD REPARTITION JOIN

To visualize the impact of skew in such cases, we present in Fig. 1 an example where two relations must be combined, tables A and B, and the distinct colors represent data that belong to a different group. Data is split into chunks and stored in a cluster that consists of storage and compute nodes (Workers 1-3 in the figure). Table 1 is stored in Workers 1-3 whereas Table B is stored in Workers 1-2. Each group represents a part of the data that is related to a specific join/aggregation attribute value and must be processed independently. The groups are organized in a distributed manner during the intermediate shuffle phase of the MapReduce algorithm [15]. In this phase, every worker extracts a part of the local group data, and the framework assigns one worker as the sole responsible to aggregate these parts from both Tables and perform the entire calculation per group, i.e., execute the actual join in the "reduce" function. For instance, in Fig. 1 the shuffle phase in denoted with the black arrows and the aggregation and join calculation for the orange group is presented as a combination of the partitions A1 and B1 which will happen inside a specific worker (among workers 1-3). The rest of the groups will also randomly be assigned to some worker (either the same or different). If some value appears with higher frequency ('0' in Fig. 1) the corresponding group will contain more data in relation with others (pink) and our data will be skewed, leading to a bigger processing time for the unfortunate worker that will perform the specific calculation. Skew is usually present in one side of the joined/aggregated relations (Table A in Fig. 1), but in general both relations could present skew. Given that each group is assigned to a single worker for processing it, i.e. partitions A0 and B0 of Fig. 1 will be sent to the same worker, this unfortunate worker will receive more load compared with others.

Therefore, the common hash-based repartition join algorithm cannot achieve satisfactory performance by definition when a dataset presents a high skew. In such cases, a large part of the data will be placed by the underlying hash-based shuffling mechanism in a few partitions only (this is also depicted in Fig.5 in [14]) leading to considerably larger total execution times caused by worker imbalances: the task scheduler which is responsible to select the worker that each data partition will be assigned, will randomly assign some large partition to a worker without considering the size of it rendering this worker overloaded. The simplest approach to address the increased worker load caused by a large group is to split any large join group into subgroups that will be distributed to multiple workers, known as the subset-replicate method.

### B. SECOND APPROACH USING SUBSET-REPLICATE JOIN

The state-of-the-art subset-replicate methodology, which is used to unload workers in such cases, is to split the large partition A0 which is created from the skewed relation (Table A) into two subsets A0-0 and A0-1 as presented in Fig. 2. Each of these subsets will then be sent to a different worker in order to achieve a more even load balance. However, each of these workers will need a copy of partition B0 to perform the computation, hence the name subset-replicate. Partitions A0-0 and B0 would therefore form one subgroup, and partitions A0-1 and B0 a second subgroup which are created from the original group related to key 0. Every time we choose to split/repartition a single group we share the load between two or more workers and at the same time we increase the network traffic due to the duplication of input records in different subgroups.

The subset-replicate methodology is used by all SQL engines that aim to efficiently process skewed datasets. However, each engine follows a different approach, with some engines letting the task scheduler implement this logic on-the-fly while others implement an entirely new join algorithm that is based on this method. More information regarding the subset-replicate methodology and variations is provided in Section II. Moreover, code-based custom solutions such as key-salting, which can be used with any engine, deal only with the load balancing and ignore input duplication overheads.

Various research works ([16], [17], [18]) have been trying to develop new join algorithms to address the problem of joining large skewed datasets over multiple nodes or engines employing the subset-replicate methodology. Most of these implementations rely on cost-based models [19] for the repartitioning decisions and target the minimization of the estimated execution time. However, these algorithms are not easily reusable due to the integrated cost models, lack extensive evaluation over large clusters and non-skewed datasets ([16], [17]) to prove their generality, come with autonomous custom-build engines ([16], [18]) or rely on specific hardware and protocols for their operation [18]. Moreover, some solutions that implement new join operators without the use of cost models are presented in [20] and [21], however they are also built on top of custom systems.

On the other hand, popular distributed general-purpose engines try to address the problem through task monitoring and dynamic rescheduling focusing on the load balancing aspect of the problem. For example, Apache Spark implements this logic with the adaptive execution framework [22] using some user-defined thresholds for the size of data partitions which are provided as system configurations (refer to related work Section III for a detailed discussion on user-defined parameters). However, the applied methodology can hardly be considered skew insensitive, as its operation and performance can vary significantly depending on the defined thresholds which may differ per use case/query. A similar but more generic approach is used in SkewTune [23], which makes cost-based decisions for repartitioning data of MapReduce tasks. Although these approaches tackle load imbalances that appear in a cluster, they ignore network overheads, and they are system centric as they require the modification of the internal operation of each engine. Although the topic of skew in distributed data processing has received a lot of attention from the research community, we have identified a number of gaps in existing approaches (refer to Section III) which we wish to address.

In any case, traditional repartitioning techniques, while effective in general cases, fail to account for skew-induced load imbalances, leading to suboptimal performance. Existing solutions often involve intricate parameter tuning, manual intervention, or require modifications to the underlying distributed execution engine, limiting their usability in diverse real-world scenarios.

Our goal is to develop an algorithm that can be used with any distributed processing system without involving cost-models and address both the load balancing and duplication-related network overheads that arise under the presence of data skew. We present a new skew-insensitive repartitioning algorithm that can be integrated in any system in the category of distributed SQL analytics engines to efficiently execute large-scale joins or any other aggregation of unordered skewed or unskewed datasets. Since our implementation is skew-insensitive it can be effectively used in any case and the user is not required to know beforehand if the data is skewed. Our algorithm includes a novel patch-based repartitioning methodology and approximation techniques. It can be used as a shuffling mechanism in place of the hash-based shuffling implementation which is commonly used at present. Our initial, partially developed algorithm, is presented in [24]. The key features[1] of our technique are:

- *Operation insensitive to data skew*: using our algorithm we eliminate the impact of the reduce-side skew and achieve optimal load-balancing, while the performance with unskewed datasets is minimally affected. The run-time overhead of our algorithm is minimal irrespective of the level of skew and it requires zero parameterization.
- *Integration with common distributed SQL engines*: irrespective of the internal operation of each engine, our algorithm could be integrated as-is in any MapReduce-type engine interacting with its task scheduler. Also, its operation is not affected by the decisions of the engine's query optimizer, or the opposite.
- *Locality-aware repartitioning of input data*: data statistics will be used to repartition the skewed parts of the dataset in a way that ensures minimum data movements, by placing specific partitions to specific workers.
- *Priority to local processing*: to ensure maximum exploitation of data locality, we split the repartitioning procedure in two phases; first we create and assign as many 'local' partitions as possible (which can be processed with the least required data transfer) and then randomly process the rest of the partitions.

The rest of the document is organized as follows: in Section II, we discuss the state-of-art repartitioning techniques. In Section III we present the related work. In Section IV, we present our methodology including examples and implementation details. The experimental evaluation is presented in Section V and we conclude our findings in Section VI.

## II. SUBSET-REPLICATE PARTITIONING METHODOLOGY OVERVIEW

In this section, we provide an overview of the subgroup repartitioning strategy which is most commonly used (with

[1] This work extends [24] by providing: the fully developed prototype with a clear methodology description and discussion about its generic applicability, an extensive evaluation section with extra experiments, a theoretical analysis regarding performance and scalability and an expanded related work section.

slight variations) for splitting the join groups of an equi-join in the presence of skew, known as subset-replicate partitioning. In Section II-A we present the calculations required before applying the partitioning, in Section II-B we describe the exact partitioning methodology, and in Section II-C we discuss about the main optimization goals.

### A. SKEW EXAMINATION

In order to identify the groups that need to be repartitioned, the sizes of the various join groups must be first determined. The frequencies of the join attribute values in each dataset must be calculated for this purpose, and this information is gathered through the cardinality estimation mechanism of typical databases (for a comprehensive survey of cardinality estimation mechanisms please refer to [19]). Although any existing data statistics can also be re-used, in general these simple calculations can be quickly performed either before the actual join or dynamically during its execution [18], [22], [23], [25]. Moreover, sampling techniques can be used in both cases to avoid examining the whole dataset(s) [16], [18], [26]. Such simple *count*-style calculations are necessary for determining skewed values and are part of the initial skew examination phase in all the existing research works.

### B. SUBSET-REPLICATE PARTITIONING

When a decision has been made to split a large group into two subgroups, the records belonging to each dataset are considered as two different sets which are handled in a different way. Depending on the number of records in each set, the largest set is usually split into two subsets forming two different subgroups and the records of the other set must be replicated in both subgroups. This methodology is called *subset-replicate* partitioning [27] and is executed iteratively to split the largest join groups into smaller subgroups. It is also known as rectangular partitioning, since a join group can be represented as a rectangle whose side has the size of a single set and it is consecutively split in smaller rectangles. It is important to mention that the replication of a set of records in different subgroups means that the same records will be sent to multiple workers instead of a single one, known as input duplication.

The repartitioning of groups into subgroups can happen once before the join execution in an offline manner or it can be a dynamic procedure that constantly calculates the optimal repartitioning for the remaining records to be joined, depending on the latest data statistics.

### C. OPTIMIZATION TARGETS

The most important decision during the repartitioning procedure is whether or not to further split a subgroup. The question can take many different forms: Will the overall execution time profit? Will we benefit more from splitting this set of the group? Will we have better load balancing? Will the replication of records cause network-related overheads? This problem can be formulated in many different ways and various cost models with different optimization rules have been evaluated in the past. However, there are two common targets in all existing efforts: a) even load balancing and b) minimal network traffic. Even the most complicated execution time models have been created along the same lines after carefully modelling the worker processing time and the network transfer time in terms of the number of records processed/transferred.

## III. RELATED WORK

Various strategies have been proposed to address data skew in distributed joins, each employing different mechanisms for skew detection, load balancing, and data replication. To provide a structured comparison of these approaches, we present Table 1, which summarizes key attributes of existing methods and highlights the distinguishing features of our proposed solution.

BigDawg [16] is, to the best of our knowledge, the first polystore engine among many, that aims to optimize the execution of cross-engine shuffle-joins considering the data skew. In this polystore system, the shuffle-join framework of SciDB [35] is integrated and modified to operate on simple relational data instead of multi-dimensional arrays. Initially, to calculate the data skew, a similar histogram is populated for each table (engine) by taking random data samples of the join attribute and the histogram buckets form the join-units. The final assignment of join-units to engines is produced after two steps; first each of the join-units is assigned to the engine that has most data locally to minimize data transfer, and then an algorithm called Tabu Search is used to unload certain engines by reassigning join-units to engines with lower cost. Both the data migration cost and the actual join cost are considered for each engine, modeled as simple quadratic functions of the number of tuples. The performance of the used algorithm is satisfactory even for non-skewed datasets. However, it is untested with a large number of nodes and distributed relational engines.

In [17] a novel algorithm for executing hash-joins with large, skewed datasets is presented. The aim of this work is to determine the best repartitioning for heavy hitters to balance the worker load as evenly as possible. A greedy algorithm that performs rectangular sub-group repartitioning is used (each side of the rectangle represents a table). This algorithm gradually increases the number of partitions on each join side by splitting its largest partition, choosing the side that produces the greatest benefit in each iteration. To quantify this benefit, the load expectation and variance of a worker are defined (load is a linear function of the number of input and output tuples) and the chosen partitioning is the one that causes the greatest variance reduction. Although this repartitioning strategy is considered near-optimal for hash-joins and does not require any knowledge of the heavy hitters beforehand, it causes input duplication each time a partition is split in two which means higher worker load expectation as the number of partitions increases. The best balance between load expectation and variance must therefore be determined

**TABLE 1.** Comparison of related work on handling data skew in distributed joins.

| Approach | Load Balancing | Data Replication | Locality-aware | Generalizability | Automation |
|---|---|---|---|---|---|
| BigDawg [16] | No | No | Yes | No | Partial |
| Hash-Join Optimization [17] | Yes | Yes | No | Yes | Partial |
| Apache Spark AE [10], [22], [28] | Yes | Yes | No | No | Partial |
| Fangorn [25] | Yes | Yes | No | No | Yes |
| SkewTune [23] | Yes | Yes | No | No | Yes |
| Flow-Join [18] | Yes | Yes | Yes | No | Partial |
| Partial Shuffle [29] | No | No | Yes | Yes | No |
| SkewJoin Strategies [20] | Yes | Yes | No | No | No |
| PRPD [21] | Yes | Yes | Yes | No | Partial |
| Nereus [30] | Yes | Yes | No | No | Yes |
| Distributed Subtrajectory Join [31] | Yes | Yes | Yes | No | No |
| Spatial Join Optimization [32] | Yes | No | Yes | No | No |
| RelJoin [33] | Yes | No | No | No | Yes |
| AlCo [34] | Yes | No | Yes | No | No |
| Mimir Extension [26] | Yes | Yes | No | No | Yes |
| **Our Patch-Based Approach** | **Yes** | **Yes** | **Yes** | **Yes** | **Yes** |

such that the running time is minimized. Another linear model, similar to the worker load, is used for the running time which includes both the network transfer time and the join execution time. For each repartitioning decision, the algorithm uses deterministic assignment of partitions to workers and then the running time is estimated based on the number of shuffled tuples and join I/O tuples. The algorithm terminates when the running time does not improve significantly and the repartitioning with the lowest running time is selected. However, the performance of this algorithm with non-skewed datasets is not properly studied.

A similar, but much simpler approach is used by Apache Spark ([10], [28]) for handling data skew with the adaptive execution framework [22] that is included since release 3.0. If a partition is much larger than the median partition size (arbitrarily preconfigured to five times bigger than the median partition size) and a preconfigured user-defined threshold value in MB (arbitrarily set to 256 MB), it is split into smaller partitions that have the average size of non-skewed partitions or a preconfigured size. Details on configuring those parameters are provided in Spark's performance tuning manual[2] in the "Optimizing skew join" section. In this case the matching partition on the other side of the join needs to be replicated. Although this methodology can successfully address the skew problem in specific simple use cases, it relies on user-defined thresholds and the performance may be hurt depending on the configuration and the use case. While Spark developers come with some suggested values that may work in the general case, they cannot cover every different dataset, cluster, and query combinations. In cases when those preconfigured values are not optimal, the user must detect them considering complex system-specific operations that may not be easily understood by a non-experienced user. For example, the user must carefully consider the fact that when both join sides are skewed, the join could become a cartesian product with a quadratic blow in the execution complexity. Moreover, the standard shuffle mechanism is still used to randomly assign partitions to workers and locally available data are not exploited properly to reduce the amount of shuffle data which is increased due to the previously mentioned replication. The main difference between Spark's adaptive execution framework and our patch-based approach in terms of user-defined configurable parameters is that we do not leave the user decide any parameter; instead, our approach automatically detects the optimal key population and replication to mitigate skew.

In Fangorn [25], a recently developed general-purpose framework that can be used with various workloads employing multiple execution engines in shared clusters, the authors use automatic run-time skew detection and dynamic plan adjustment by gathering statistics during execution. This approach is based on task management, like the Spark framework, and locally available data are not exploited too.

Another system-centric approach is presented with SkewTune [23]. Being implemented as an extension for MapReduce-type engines, it monitors task execution and reacts whenever a slot in the cluster becomes available to rebalance the load. In order to do so, it stops the task with the maximum estimated completion time and repartitions its input data to new tasks using a heuristic algorithm. For the repartitioning process, SkewTune collects a compressed summary of the input data. SkewTune addresses different types of skew that can occur when some keys are more expensive to process than others, because, for instance, they are larger in size, or when the hashing partitioning algorithm does not produce even distributions. The algorithm identifies straggling tasks by constantly examining their completion time; when it identifies that the completion time exceeds a user-defined threshold compared to the mean completion time, it then splits the load of this task into two different tasks and dispatches them in idle resources. Although both SkewTune and our patch based algorithm addresses load imbalances, our approach targets specifically join operations. Moreover, our approach uses both load migration (by moving keys between nodes) and replication (by replicating hot areas

[2]https://spark.apache.org/docs/latest/sql-performance-tuning.html

into more than one node) to distribute the join load among the cluster, whereas SkewTune uses only load migration.

In Flow-Join [18] the authors focus on fast skew detection when executing hash-joins over modern high-speed networks. A novel algorithm is presented, which detects heavy hitters and repartitions the data at runtime using small approximate histograms. Each worker maintains its own histogram with local heavy hitters, and constantly updates it using the space-saving algorithm while performing the join. After processing only 1% of the probe input, each worker can start using his histogram to decide on its local heavy hitters based on a fixed skew threshold value. The probe tuples are normally sent to the build side if no skew is detected, otherwise they are kept local, and the worker asks to receive the build side tuples from other workers asynchronously. This method is called Selective Broadcast and aims to minimize network transfers. For a more general approach, a global histogram is built for each of the input sides and in case the skew is detected in both sides the heavy hitter tuples are redistributed using the symmetric fragment replicate shuffling scheme. With this scheme a grid is created that repartitions data to the servers, in order to avoid the excessive network I/O and load imbalance that the Selective Broadcast would cause. The grid shape depends on the relative frequency of a heavy hitter in both inputs, i.e., the heavy hitters are repartitioned using a square grid in case skew is similar in both input sides. The final algorithm uses lazy tuple materialization while building the two global histograms and uses a pipelined probing approach for deciding how each tuple will be joined.

Addressing the increased network traffic that incurs with the hash-based shuffle mechanism which is used not only in hash-joins but in the deep learning domain too, the authors in [29] propose a partial instead of global shuffle of the data aiming to maximize local processing and minimize data movement. This approach is also applicable to other domains too except from the deep learning domain, such as large-scale relational processing.

In [20] the authors implement three different SkewJoin strategies to mitigate different data skew scenarios in inner- and outer- joins. The three strategies present trade-offs between performance and reliability, and the evaluation confirms that although they improve the performance by balancing the workload, each strategy performs best in a different scenario and there is no single one that outperforms the others in all scenarios. There are pros and cons in the usage of each strategy which the user should carefully consider.

In [21] the authors propose a new join geography called PRPD (partial redistribution and partial duplication) which can be used by SQL optimizers when generating a query plan. The implementation is based on the reasoning that skewed values are kept locally which saves redistribution cost and decreases execution time. PRPD not only deals with cases where only one relation is skewed, but also cases where both relations are skewed.

In [30] the authors present Nereus, a distributed stream join processing scheme that is based on adaptive partitioning and migrations to achieve load balancing between participating peers. They measure observed load among nodes and when they detect imbalances, they perform load migrations. Nevetheless, their work is focused only on streaming data, and they do not consider also the subset replicate approach that we follow in our methodology.

In [31] the authors deal with the problem of joining spatial subtrajectories using MapReduce. They employ a repartitioning mechanism that achieves load balancing and collocation of temporally adjacent data called subjtrajectory join with repartitioning in which they initially sample the dataset to create equally sized partitions. Nevertheless, their approach does not consider a subset replicate approach, and they do not consider data movement costs during the partition calculation.

In [32] the authors also deal with the problem of joining spatial datasets by utilizing a distributed processing engine like Apache Spark. They propose a spatial partitioner that divides the joining datasets while maximizing resource utilization, decrease shuffling and reduce execution time. Although their approach is like our adaptive repartitioning scheme, they do not consider a subset replicate approach during partition calculation.

In RelJoin [33] the authors present distributed joins that consider skewed datasets and implement their approach over Apache Spark by employing a cost model that considers transfer and execution time. They study the joins presented in [14] on skewed datasets. Nevertheless, their main contribution lies in the optimization of the produced query plan by performing, among others, operand reordering without dealing with subset replicate or locality-aware data placement approaches.

In AlCo [34] the authors propose a fragments allocation (i.e., partition allocation) method to perform joins in a distributed database by issuing a genetic algorithm. They implement their approach in oceanbase [36], an RDBMS developed by Alibaba and utilize the TPC-H benchmark to experimentally evaluate their findings. During partition allocation the authors consider data locality and movement cost in a similar to our Patch-Based approach, nevertheless they do not investigate how a subset replicate approach could be employed.

In [26] the authors extend Mimir [37], their previous work, and present a skew tolerant MapReduce methodology. They employ dynamic repartition optimizations to balance memory usage across workers over an MPI setting. They devise data-skew strategies based on dynamic repartitions and ''superkey'' split strategies for in-memory workloads. Repartitions are performed in an iterative manner by utilizing a repartition and bin table that through sampling collects information about the dataset skew. Consecutive repartitioning is performed until a load-balancing threshold is reached. Nevertheless, during repartitioning the authors do not consider existing data locality. When the repartitioning

phase is complete, the superkey split strategy is employed. The superkey split strategy is in essence a subset replicate approach that splits ''superkeys'' values (i.e., specific keys that contain a large number of values) during the reduce-side processing into different partitions. This information is kept on a hash table called ''split'' table. Nevertheless, their approach also does not consider data locality during subset replicate and placement, whereas it merely tries to only minimize load imbalances without considering imposed transfer costs.

An orthogonal but yet relevant approach to deal with the aspect of privacy in distributed cloud-enabled is followed in [5], [38], and [39]. In these works, the authors provide efficient indexing structures that can be used in distributed data stores to maintain the data privacy while allowing query execution.

In overall, we believe the following gaps exist in the literature which we aim to address:

### A. SKEW-AGNOSTIC DESIGN

Unlike many existing solutions requiring heuristic tuning of skew parameters, our patch-based algorithm is skew-insensitive and does not require any heuristic tuning. It automatically adapts to varying data distributions without manual intervention.

### B. GENERALIZABILITY

Most existing techniques are tailored for specific applications or data types. The proposed algorithm is designed to integrate seamlessly into any distributed execution engine, offering a general-purpose solution for unordered datasets.

### C. LOCALITY-AWARE OPTIMIZATION

Traditional hash-based shuffles are network-intensive, as they indiscriminately redistribute data. The patching mechanism leverages data locality to minimize network overhead while balancing load.

## IV. METHODOLOGY

In this section, we formulate the problem and describe our methodology to repartition skewed data. In IV-A we describe in brief our approach and optimization goals. In IV-B and IV-C, we present the types of statistics that must be calculated initially, which will be used by our algorithm to extract the optimal load balancing and minimize data movement. We proceed in IV-D where we describe the high-level logic of our repartitioning algorithm, while in IV-E we present in detail the patch-based repartitioning. In IV-F the final shuffle and join algorithm is presented, which is based on the results of our patch-based repartitioning algorithm. In the Appendix section B we provide a full example of the execution of our repartitioning algorithm and compare the results with other common implementations. In the Appendix section C we present the exact Algorithm implementations in pseudo-code. Finally, in IV-G we describe how our approach can be integrated with existing MPP Databases.

### A. APPROACH OVERVIEW

The proper repartitioning of data groups is of crucial importance to effectively improve the performance of a highly skewed reduce-side operation. Following the flow, we choose to focus on the load balancing and network traffic aspects too, however we introduce a novel approach which is solely based on the assignment of subgroups to workers without the need for developing any cost models. Unlike subset-replicate partitioning, which relies on iterative splitting and replication of records, our patch-based approach uses informed subgroup assignment to minimize replication while maintaining load balance.

Our approach forms a novel shuffling algorithm which can be used *before performing any reduce-side operation*, i.e., before performing any actual key movement or join computation. The pseudo-code that presents in high-level the logic of our novel patch-based shuffling algorithm and how it is used for performing reduce-side operations such as a join is presented in Algorithm 1. The entire process is split into three major stages, one involving only statistics and thresholds calculation, one including the calculation of the patch-based partitions and the other for the actual data transfer and join computation, depicted as a diagram in Fig. 3:

- The first stage only calculates the necessary statistics. This stage consists of the first two steps of Fig. 3, it is presented in high-level in lines 1-3 of Algorithm 1 and in more detail in Section IV-B and IV-C.
  - In lines 1-2 of Algorithm 1 the group sizes are extracted and the data location is identified for every dataset. All workers participate in the calculation of these statistics. This procedure is the first step of Fig. 3 and it is described in detail in Section IV-B.
  - In line 3 of Algorithm 1 one worker calculates the actual join load and a per-worker threshold to ensure load balancing. This procedure is the second step of Fig. 3 and it is described in detail in Section IV-C.
- The second stage performs the patch-based data repartitioning. The algorithm that computes the ''patches'' is executed in a single worker and consists of two steps as depicted in Fig. 3 and in more detail in Section IV-D and IV-E.
  - In lines 4-20 of Algorithm 1 we are ready to form the groups that will be joined together. This procedure consists of the third and fourth steps of Fig. 3 and it is described in brief in Section IV-D and in Algorithm 2.
  - First, we aggressively try to identify groups that can be joined locally, i.e., without performing network data transfers (step 3 of Fig. 3, lines 4-12 of Algorithm 2: we extracted the functionality of line 7 into a separate procedure described in Section IV-E and Algorithm 3 for clarity).

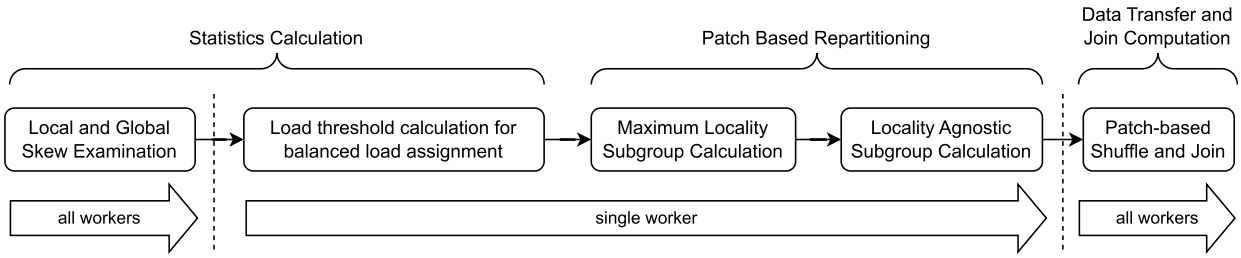



**FIGURE 3.** Pictorial representation of our patch-based algorithm split into its main stages.

  - Second, we identify groups that involve data transfer from one worker to the other (step 4 of Fig. 3, lines 13-20 of Algorithm 2: the functionality of line 17 is described in Section IV-E and Algorithm 4 for clarity).
- The third stage performs the actual join after taking informed decisions calculated in the second stage that both minimize data transfer and perform load-balancing. It consists of the last step of Fig. 3 and it is presented in high-level in lines 21-27 of Algorithm 1 and in more detail in Section IV-F. An exact algorithmic presentation of the joining process is described in Algorithm 5.

We argue that through the guided initial creation and assignment of subgroups to workers one can tackle both issues:

#### 1) LOAD BALANCING

To achieve an even load balancing, each subgroup must be assigned to a worker considering its size and the existing assignments/load of the worker. We consider the load of a worker as the aggregate size of the subgroups assigned to it or as the aggregate number of computed output tuples. When a new subgroup is created, we select a worker for it (based on different criteria that we explain in subsection IV-D) under the condition that its total load will not exceed a threshold which is common for all workers. We therefore prefer to follow an over-partitioning approach when creating subgroups to ensure that we have minimal load variance between workers. We provide more details about this in subsection IV-C.

#### 2) NETWORK TRANSFER

Instead of attempting to minimize the data transfer that is caused by every splitting/repartitioning decision, we use an equivalent approach in which we try to ensure maximum data locality when creating and assigning subgroups. When we examine how a specific group should be split, we first consider the workers that have most of the group data stored locally and we create and assign to them the largest possible subgroups without overloading them. For the remaining data of the group, for which we cannot ensure locality, we iteratively assign the largest possible subgroup to the worker with the least load. In the following paragraphs we describe in more detail our methodology and present the developed algorithms which are combined to execute a highly skewed reduce-side operation (Algorithms 2-5). The exact algorithm definitions can be found in the Appendix C.

**Algorithm 1** Patch-Based Shuffle and Join High-Level Overview

**Input** Input datasets, number of workers
**Output** Result

```
1:  computeGroupSizes()
2:  computeWorkerLocalRecordStats()
3:  computeWorkerLoadThreshold()
    /* compute subgroups and their assignment to workers */
4:  repeat
5:    for worker in getWorkersWithMostLocalData() do
6:      if workerLoadLessThanThreshold() then
7:        computeLocalSubgroupUsingThreshold()
8:        assignSubgroupToWorker()
9:        updateRecordStatsRemovingSubgroupData()
10:     end if
11:   end for
12: until no local subgroups can be assigned
13: repeat
14:   for group in getLargestRemainingGroups() do
15:     getWorkerWithLeastLoad()
16:     computeSubgroupBasedOnThreshold()
17:     assignSubgroupToWorker()
18:     updateRecordStatsRemovingSubgroup()
19:   end for
20: until all remaining data are assigned
    /* shuffle data using subgroups assignments */
21: for record in input datasets do
22:   getInitialRecordGroupAndLocation()
23:   getComputedTargetSubgroupsForRecord()
24:   tagRecordWithSubgroupId()
25:   sendRecordToTargetWorkerOrKeepLocally()
26: end for
27: joinDataOnSubgroupIds()
```

### B. LOCAL AND GLOBAL SKEW EXAMINATION

As in all previous works, before we start repartitioning the groups we must first compute their sizes with a skew examination phase. Each group corresponds to a specific attribute

**TABLE 2. Variables notation.**

| Variable | Meaning |
|---|---|
| $N$ | Number of workers in the cluster |
| $S, T$ | Input datasets |
| $A$ | Join attribute |
| $V$ | Set of distinct join attribute values |
| $S_a, T_a$ | $S_a = \{s \in S : s.A = a\}, T_a = \{t \in T : t.A = a\}$ |
| $P_a$ | $P_a = \{p \in S_a \cup T_a\}$ |
| $P_a^{lo}$ | A partition (subgroup) of $P_a$ assigned to a specific worker $n$ than can be processed mainly **locally** |
| $P_a^{mov}$ | A partition (subgroup) of $P_a$ assigned to a specific worker $n$ that **Requires global network data transfer** |
| $s_a(n), t_a(n)$ | $s_a(n)$: number of $S_a$ records in worker $n$, $t_a(n)$: number of $T_a$ records in worker $n$ |
| $l_a(n)$ | Local weight of worker $n$ for join group $P_a$ |
| $g_a$ | Size of join group $P_a$ |
| $C_{max}$ | Maximum allowed load on a worker |

**TABLE 3. Number of records in each set and the computed local weights per worker for group $P_a$.**

| Worker $n$ | $s_a(n)$ | $t_a(n)$ | $l_a(n)$ |
|---|---|---|---|
| 0 | 10 | 20 | 200 |
| 1 | 41 | 10 | 410 |
| 2 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 |

value and the data inside it must be joined/aggregated in the reduce side. The size of each group is calculated as the product of the frequencies of this specific value in the two combined datasets. We perform these calculations once before we execute our repartitioning algorithm and we store a few different metrics for each group $P_x$: a local weight $l_x(n)$ per worker $n$ and its total size $g_x$. The local weight is computed for each worker like the group size but using only its local data. The notation of the variables that we will use is presented in detail in Table 2.

For example, let us assume two datasets $S$ and $T$ that need to be joined on a single attribute $A$, and a join group $P_a$ which is for the attribute value $a$. We also assume that there are four workers available, of which only two contain data related to the attribute value $a$. In Table 3, we can see the number of records in each worker where this value appears (i.e. its frequency) and the local weights which are computed as

$$l_a(n) = s_a(n) \times t_a(n), n = \{1, .., N\}. \tag{1}$$

Using (2), the total number of $S_a$ records across all workers is $|S_a| = 51$. Respectively, according to (3), $|T_a| = 30$. $|S_a|$ and $|T_a|$ are considered as the group dimensions and the group size is computed using (4) as $g_a = 1530$.

$$|S_a| = \sum_{n=1}^{N} s_a(n) \tag{2}$$

$$|T_a| = \sum_{n=1}^{N} t_a(n) \tag{3}$$

$$g_a = |S_a| \times |T_a| \tag{4}$$

The size of a join group indicates the total load that this group will produce in our execution environment, in terms of the number of comparisons that will be performed (using a cartesian product) or the number of final output tuples. The local weights, on the other hand, indicate the load that will be put on each worker iff its local group data are joined locally. Most importantly, the local weights are used in our case to measure the benefit of selecting a 'local' join i.e. creating a subgroup that contains its local data and assigning it to this particular worker. The highest the local weight of a worker is for a particular attribute value the least network transfer will be required for the group given that we attempt to assign the largest possible subgroup to this worker. Based on our example, worker 1 is the first candidate to be assigned with the largest possible subgroup and then worker 0. Workers 2 and 3 will only be used at the end in case the two first workers have been overloaded.

At the end of the skew examination phase, the local weights, the group dimensions, and the size for each group have been computed. Next, we calculate the maximum load that can be assigned to any worker by our repartitioning algorithm to ensure an even load balancing.

### C. LOAD THRESHOLD CALCULATION FOR BALANCED LOAD ASSIGNMENT

Let $V$ be the set of all the distinct values of the join attribute for a specific join operation. Each element in $V$ corresponds to a different join group. By adding the sizes of all the join groups that are computed in the skew examination phase, we can estimate the total load $L$ induced by this join operation as

$$L = \sum_{v \in V} g_v. \tag{5}$$

Assuming an optimal even load balancing between workers, the maximum load of a worker, which we will refer to as maximum capacity, is

$$C_{max} = L/N + 1. \tag{6}$$

The maximum worker capacity will indicate in our case the maximum allowed load of a worker and will be used as a strict threshold to avoid overloading any single worker at any time.

In practice, when we examine a specific worker that has the most data of a particular group locally, we will try to assign the largest possible subgroup to it. If the worker already has a great amount of load from previous assignments of other groups and therefore only a little capacity left until reaching $C_{max}$ value, we force the creation of even the smallest subgroup that can be assigned to it without violating this threshold to exploit any existing data locality. Although this may lead to an aggressive over-partitioning that we should avoid since it causes input duplication, it tends to happen more during the last decisions taken by our algorithm. At this point, both the remaining capacity of workers and the remaining records are limited and the benefits from constantly trying to schedule as much local joins as possible prove to be much more important in overall.

### D. TWO PHASE REPARTITIONING ALGORITHM

Before we start repartitioning groups and assigning subgroups to workers, the maximum worker capacity is computed and is initialized to this value for all workers and all the computed local weights are gathered and sorted in descending order creating a queue. Then starting from the largest weight, we identify the worker and group that it refers to and compute the largest possible subgroup that can be assigned to the worker *without exceeding* its remaining capacity. When we reach the end of the queue, we update the local weights of the workers by excluding any records that will not be used again, and we rebuild the queue using the new local weights. Then the procedure is repeated with the updated queue.

When we fail to create and assign a new subgroup after traversing the whole queue (which means that no worker has enough capacity left) the first phase of our algorithm, that tries to schedule *local* computations using the local weights, is completed.

A second phase follows, for which we create a new queue by ordering the remaining group data in descending size. We traverse this queue only once and for each group we repeatedly assign the largest possible subgroup to the worker with the most remaining capacity without again exceeding the maximum allowed load. We do not proceed to the next group in the queue until all the current group data have been aggressively assigned to workers. During this second phase, we therefore randomly schedule reduce-side computations ignoring the initial data location.

Our repartitioning algorithm is presented in Algorithm 2 and includes the two aforementioned distinct phases. In brief, in the first phase we create subgroups trying to exploit the data locality based on the queue that we update constantly, and the second phase simply handles the remaining group data ignoring data locality. During both phases, the worker maximum capacity is never exceeded, leading to optimal load balancing.

More formally, we first apply local assignment and then we proceed to global assignment:

#### 1) LOCAL ASSIGNMENT

The assignment of $P_a^{lo} \subseteq P_a$ to workers with high local weight $l_a(n)$, ensuring:

$$|P_a^{lo}| \leq \min(l_a(n), C_{max}).$$

#### 2) GLOBAL ASSIGNMENT

The assignment of remaining data $P_a^{mov}$ to workers based on their remaining capacity, ensuring:

$$|P_a^{mov}| \leq C_{max}.$$

We also define the locality factor for a specific join group $P_a$

$$\text{Locality factor} = \frac{\sum_n l_a(n)}{\sum_n (s_a(n) + t_a(n))}.$$

The numerator measures the amount of data processed locally, while the denominator represents the total data associated with the group $P_a$. A locality factor of 1 means all records are processed locally without requiring data transfer, therefore this factor approaches 1 under optimal locality.

The computational complexity of Phase 1 depends on the number of subgroups created, while Phase 2 runs with a worst-case complexity of $O(L)$, where $L$ is the total load as defined earlier. In the Appendix section we give a detailed theoretical analysis and convergence proof of our approach.

In the next subsection, we present a detailed example of how a single subgroup is created from a group for a specific worker during the first phase and the different algorithms that are used in each phase for the subgroup creation.

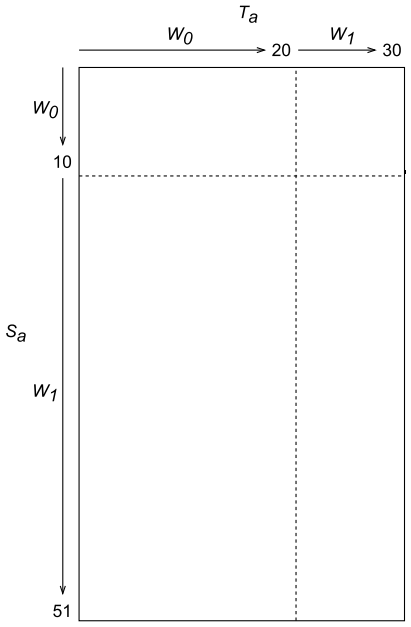



**FIGURE 4. Visualization of join group $P_a$ including the number of records in each table and the initial placement of records in the workers. The delimited cross is used to differentiate the location of the data.**

### E. PATCH-BASED GROUP REPARTITIONING

Continuing from our previous example in subsection IV-B, we can visualize group $P_a$ in Fig. 4. The size of each side of the rectangle is equal to the number of records on each dataset i.e. $|S_a| = 51$ and $|T_a| = 30$, and the total area is equal to the group size. We also note next to each side the part of records stored in each worker and try to visualize the separate locations of the data with the delimited black cross. Assuming that worker 1 is the one with the largest local weight (410) in the queue for group $P_a$ and assuming that its remaining capacity until reaching $C_{max}$ value is $cap(1) = 973$, the first subgroup that we create and assign to worker 1 includes 32 records from $S_a$ and all the 30 records of $T_a$, and is denoted as the pink area in Fig. 5.

To explain this subgroup selection, first we try to assign the whole group (whole rectangle) to worker 1. If its size exceeds the worker's remaining capacity as in this case ($g_a = 1530 > cap(1) = 973$), we try to split the largest

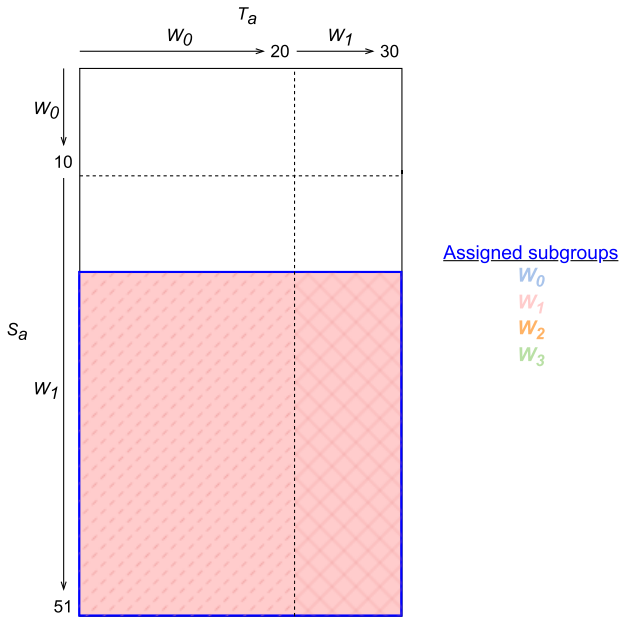



**FIGURE 5.** First subgroup created from the join group $P_a$, assigned to worker 1. The subgroup is denoted as a blue rectangle and is colored differently depending on the worker it is assigned to. The cross-hatched areas of the subgroup can eventually be computed using only local records of the worker, while the dashed areas use partially local records.

set between $S_a$ and $T_a$ which is $S_a$. Initially we will attempt to select worker's 1 local $S_a$ records (41) and all of the $T_a$ records (30) which will create a subgroup of size $41 \times 30 = 1230$. As this number still exceeds the available capacity, we finally decide to select as much local $S_a$ records as possible given the remaining capacity i.e. we use $973 \div 30 = 32$ records from $S_a$ in the subgroup that we create. The final subgroup size, which is the new load assigned to worker 1, is $32 \times 30 = 960$, leaving a remaining capacity of 13 to the worker. The procedure is depicted in Algorithm 3 and is used during the first phase of our algorithm.

Regarding the data locality in the created subgroup, all the 32 $S_a$ records can be found locally on worker 1 (see table 3), however only the 10 out of 30 $T_a$ records are local and the remaining 20 will be transferred from worker 0 to 1. The number of records that need to be transferred from each remote worker is also stored when creating a subgroup (variables *ext* in Algorithm 3). It is important to notice that although we try to use as much local data as possible, the subgroup contains records transferred from other workers too, for example in this case the 20 $T_a$ records that we mentioned. This is necessary to ensure that the 32 local $S_a$ records will not be used again as they are joined with all the corresponding $T_a$ records. Therefore, we have fully exploited the locality for these 32 records since other workers will not need them. The values of the variables $s_a, t_a, l_a, g_a$ are updated at the end, considering that the 32 $S_a$ records of the subgroup will not be used again.

In the next iterations of Algorithm 3, new subgroups are examined for creation from various join groups depending on the updated local weights and the remaining capacity on the workers. The first and largest local weight that appears on the queue indicates the group that will be examined next and the worker that it will be assigned.

During the second phase of our algorithm a slightly different procedure is used for creating subgroups, which is presented in Algorithm 4. The difference from Algorithm 3 is that the local weights are no longer used, and the algorithm will simply create the largest possible subgroup for the worker.

When Algorithm 2 is complete and all subgroups have been created, group $P_a$ and the subgroups it is broken into are presented in Fig. 6. The subgroups appearing in Fig. 6 for group $P_a$ will therefore be created at random future iterations, and not necessarily successively as other groups are also being repartitioned in between.

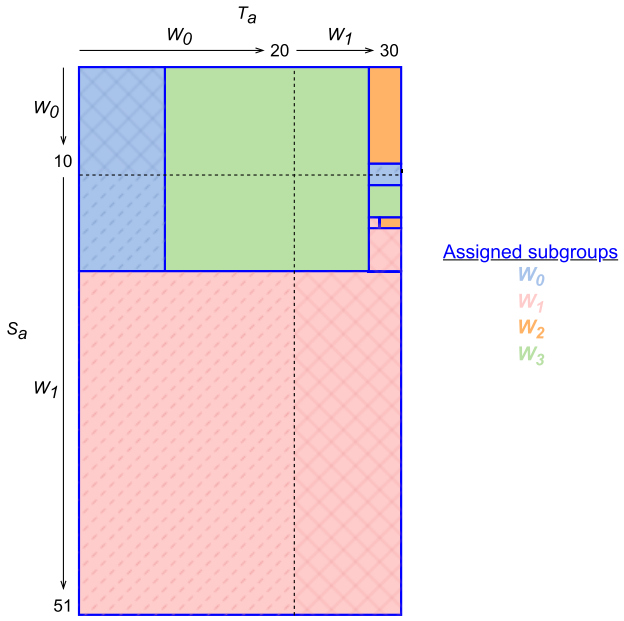



**FIGURE 6.** Patch-based repartitioning of the join group $P_a$. Each subgroup is denoted as a blue rectangle and is colored differently depending on the worker it is assigned to. The cross-hatched areas can eventually be computed using only local records of the worker, while the dashed areas use partially local records. The simply colored areas are computed only with remote records that are transferred from other workers.

The complexity of Algorithms 2 and 3 is O(1). In the worst-case scenario, Algorithm 2 will assign to all the created subgroups 1 S record and 1 T record therefore turning the calculation of the output into a cartesian product. The number of subgroups that will be computed from Algorithm 2 in such case is L, where L is the total load defined in section III-C and is equal to the total number of output tuples that will be computed. Algorithm 2 has a worst-case complexity of O(L).

With the described approach it is clear that we do not split the largest set exactly in half or use predefined dimensions

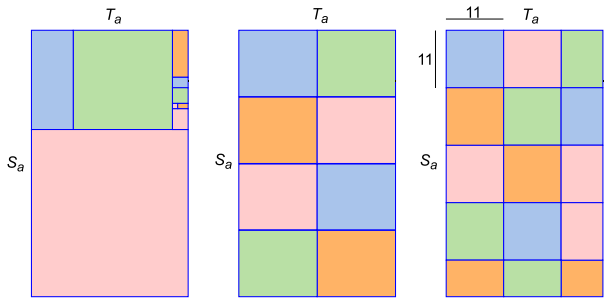



**FIGURE 7.** Repartitioning based on patches (left), consecutive divisions (middle) and size limit (right).

like previous works which consecutively divide in half the set or create subsets that do not exceed a specific size limit (Fig. 7). Instead, we aggressively try to exploit the data locality keeping as many records locally as possible. This is the reason why we call our repartitioning method **patch-based**, since we create subgroups with random dimensions that resemble patches instead of being subdivisions of the initial group dimensions. The choice of splitting the largest set of the two is based on the simple reasoning that if we split the smallest one then the large set must inevitably be replicated and used again in the future by other workers transferring a largest amount of data over the network.

### F. FINAL SHUFFLE AND JOIN ALGORITHM

After the subgroups and their assignments are computed using Algorithm 2, the join or any other reduce-side operation can be executed.

To perform the required operation, the records must first be placed into the subgroups by tagging them with the corresponding subgroup ids. This tagging procedure is equivalent to a common shuffle as records are transferred to specific workers and then the actual join operation can be executed.

More specifically, if for a record's attribute value there are relative subgroup(s) that are assigned to the worker itself, the record will be kept locally and be tagged (with an extra field) with some subgroup id(s). If there are other workers that must receive records for this attribute value (noted in their assigned subgroups) the worker also sends the record with the appropriate tag(s) to the required workers. Tagging each record with multiple tags helps us avoid duplicating a record for each subgroup inside a worker and therefore reduces the memory footprint of our algorithm.

A graphical representation of the tags that are created for three $S_a$ records of group $P_a$ which was presented in the previous subsection, is presented in Fig. 8. The black tagged entries represent records that will only be tagged and kept locally, and the red tagged entries represent records that will be tagged and sent to other workers. In case of record $s_6$ which is initially located in worker 1, it will be assigned to only one subgroup (large pink) which contains all $T_a$ records and it is assigned to worker 1 as we can see in Fig. 8. On the other hand, record $s_{39}$ will be included in three different subgroups to be joined with all the $T_a$ records, one subgroup which is assigned to worker 0 (large blue) and two other subgroups assigned to worker 3 (large and small green). The record will not be emitted twice to worker 3 in this case, instead two tags will be sent together with this record.

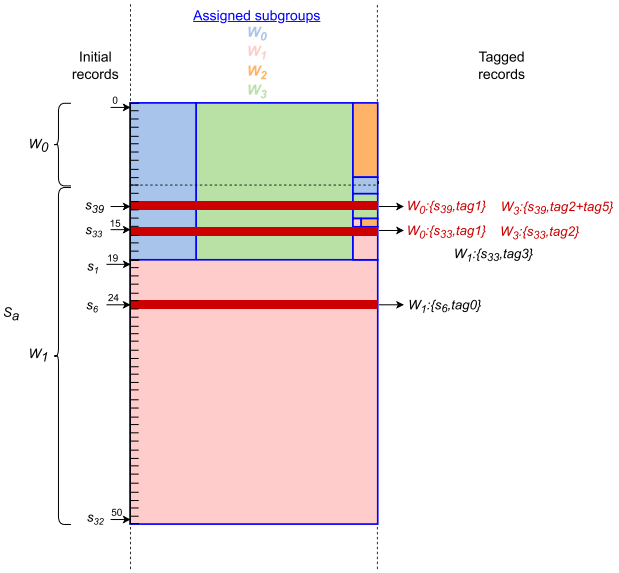



**FIGURE 8.** $S_a$ records placement in the subgroups created for group $P_a$.

When the record tagging and transfer is finished all the workers can start processing their local subgroup data. The records from each dataset are **joined/aggregated on the selected attribute for each of their common tags** i.e. for each subgroup that they are collocated.

Although Fig. 8 presents tagging to be a simple procedure, the selection of the subgroups that each record will be placed is neither straightforward nor a trivial decision. The complexity of this problem derives from the fact that the output of Algorithm 2 for each group is a list of subgroups with their sizes and the number of records required from other workers. Using this information only, the selection could be considered as a variation of the subset-sum problem [40] which is an NP-complete problem. Another approach that could be used is based on the exact placement of the subgroups in the larger group rectangle as visualized in Fig. 8, and then each record will be placed in exactly one row which automatically indicates the combination of subgroups. The placement of the subgroups could be based on the two-dimensional bin packing problem [41] or the rectangle packing problem ([42], [43]), which are also NP-complete. However, these solutions, except from a high complexity, pack the subgroups in a random way ignoring the initial and final data placement which in our case is specific. To preserve the locality which is selected by the patch-based repartitioning algorithm, we therefore implement a custom approximate method for the record tagging procedure, which

aims to apply the results of Algorithm 2 as close as possible while reducing the complexity of the problem.

The approximate tagging procedure is presented in Algorithm 5 as part of the final patch-based join algorithm and is based on the following:

- The patch-based repartitioning algorithm preserves for each subgroup two indices which indicate the position of the top left corner of each subgroup inside the group rectangle. The indices are constantly updated with every subgroup creation and this information is stored inside the subgroup information. This is omitted in Algorithms 3 and 4 for ease of readability.
- Since the placement of subgroups is therefore predefined, the selection of subgroups for a single record becomes equivalent to selecting a row as shown in Fig. 8. The row contents indicate the subgroups that this record will be placed.
- To select the row, the subgroups are examined with their order of creation, which helps in preserving data locality as the first groups are created with maximal locality. When a subgroup is selected its indices function as a placeholder which indicates the selected row, and this row is marked appropriately and never reused.
- The approximate tagging procedure first tries to assign the local record of a worker using as a placeholder for the row a subgroup assigned to the same worker. If this fails, then it uses a subgroup assigned to another worker that requires data from this worker (indicated in the subgroup information). In case both steps fail because no row is available, the row is randomly selected.
- The subgroup information is not updated during the tagging procedure to reduce the complexity of the problem (except from marking the used rows). This is the reason our algorithm is approximate as it does not strictly count the records transferred from worker to worker and may have some deviation from the theoretical values at the end.
- In the case of S records, a row must be selected similar to Fig. 8, whereas for T records a column in selected respectively.

To better understand the row selection procedure and how the row indices are determined, we will present an example based on Fig. 8. Initially, during the subgroup creation, the size, and indices of the top left corner of each subgroup is stored as we have previously mentioned. In case of the group that appears in Fig. 8 some of the subgroups would for example be represented with the following information: $[\{subgroup\_id = 1, assigned\_worker = 1, s\_records = 32, t\_records = 30, external\_s = \{\}, external\_t = \{w_0 = 20\}, s\_idx = 19, t\_idx = 0\}, \{subgroup\_id = 2, assigned\_worker = 0, s\_records = 19, t\_records = 8, external\_s = \{w_1 = 9\}, external\_t = \{\}, s\_idx = 0, t\_idx = 0\}, ..]$. When we examine in which subgroup(s) should record $s_6$ of group $P_a$ be placed, we start by checking the initial location of the record which is worker 1. We therefore try to locate the first subgroup which was assigned to worker 1 which has id 1 and is the largest pink area in Fig. 8. The value of $s\_idx = 19$ is the placeholder which indicates the starting row that we will begin searching for an available (not previously marked) row in this subgroup and the value $s\_idx + s\_records = 51$ indicates the row index that we will stop searching. If rows with $s\_idx \in [19, 23]$ are already marked as used by other records, we serially increase the starting index value by one until we locate an available row which would be row with $s\_idx = 24$ in this case. This row indicates the subgroups that this record will be placed, which is only subgroup with id 1 and then this row is marked as used. In case of record $s_{33}$ which is initially located in worker 1 too, we start by examining subgroup with id 1, but supposing that all rows are used by the previous 32 records there is no available row in this subgroup. In this case we then look for the next subgroup which was created and assigned to worker 1 (the second largest ping area in Fig. 8) which indicates row with $s\_idx = 15$ as the starting row for our search. Assuming that this row is available it indicates that the record must be placed in three different subgroups, one of which is only assigned to worker 1. It is worth noting that the three different subgroup ids are directly retrieved from a structure where we store during the subgroup creation the subgroup ids that exist in each row.

The complexity of the tagging procedure for a single record is O(z), where z is the number of subgroups created for the relevant group. As we can see in Algorithm 5 for each record the list of subgroups is iterated three times at most.

In terms of memory pressure, the patch-based algorithm is primarily related to the data structures required for patch computation. Specifically: Each cluster node maintains only queues or boundary structures for calculating patches. These structures are lightweight and do not require significant memory resources. After patch computation, the join operation is executed using the underlying distributed processing engine's standard strategies (e.g., repartition join).

### G. GENERIC APPLICABILITY

In this section we describe how our approach can be integrated with existing MPP (i.e., Massively Parallel Processing) Databases. In Fig. 9 we present a typical software and hardware stack of an MPP database. We begin by briefly describing each component in the architecture and we then discuss the required changes so that our Patch-Based approach can be integrated.

The stack moves from the infrastructural bottom layers up to the API layers where the entire functionality is being "hidden" behind a typical SQL client library (upper figure layer). Therefore, both MPP and legacy centralized RDBMs seem the same: a client/server approach is used where users utilize a client library like JDBC or ODBC to connect with the existing server backend by providing a url-like address which includes the server's IP along with other connection details. When the connection is successful, clients can send SQL queries to the server and retrieve back the results.

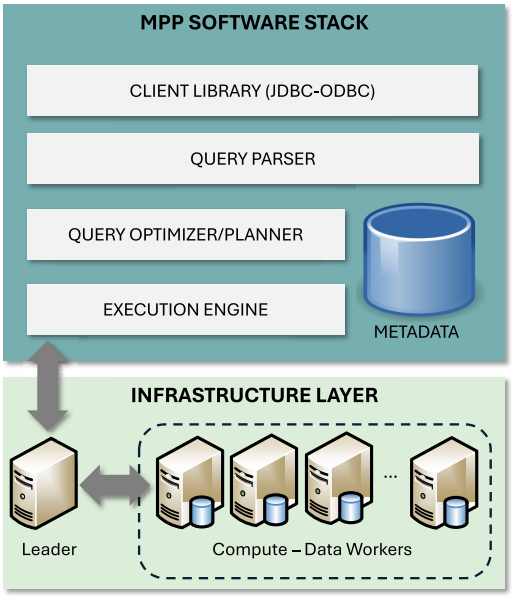



**FIGURE 9.** Typical software and infrastructure stack of an MPP engine.

The second layer presents the query parser module. It retrieves the user SQL and translates it in an intermediate tree-like structure using relational algebra. In the tree, nodes represent abstract calculations (i.e., filters, joins, selections, projections, etc.) and links depict calculation dependencies between parent and children nodes. In this layer, SQL is transformed into an abstract syntax tree (AST) that includes the necessary calculations to execute the SQL statement (often refereed to as the ''logical query plan'').

In the third layer, the logical query plan from the second layer is being transformed into an exact execution plan using the query optimization engine (often referred to as the ''physical query plan''). The query optimizer performs a cost-based optimization in which it identifies a set of possible different execution plans using both well-known algebraic properties (e.g., commutative and associative properties of join or select operators, etc.) and different algorithm implementations (e.g., broadcast vs repartition join, etc. [14]). The optimizer exploits a metadata DB that contains qualitative information about existing data (the DB right to the optimizer in the Figure). Typical information includes table structure, existing indices, key cardinality, min/max/median values, etc. An example of such a metadata repository is Apache Hive's MetaStore [11]. At the end, an exact computation workflow (i.e., the physical query plan) is being compiled and is ready to be executed by the underlying hardware.

In the fourth layer the actual computation takes place. The execution engine interacts with the underlying hardware resources in the OS level and executes the physical query plan. In a typical centralized single node approach the execution engine creates and executes the necessary threads through the operating system scheduler. In an MPP approach the engine interacts with the cluster scheduler that typically runs in a leader-worker setting (lower level of Fig. 9). The scheduler is executed in the leader node of a computing cluster and assigns pieces of work into the available worker nodes according to different policies. Typical cluster schedulers are MESOS [44], Yarn [45], Google's Kubernetes,[3] Apache Spark's Fair Scheduler,[4] etc.

Our Patch-Based algorithm can be easily integrated with existing MPP databases that follow this architecture by performing targeted changes to the query optimizer and minimal changes to the query parser in order to allow the selection of our shuffling mechanism by the user. The execution engine, client libraries and communication with existing schedulers remain intact. In the first stages of the execution of the Patch-Based algorithm where the data statistics and subgroups are calculated, i.e. Algorithms 2, 3 and 4, some changes are required in the query optimizer to include in the physical plan as a map-reduce task the calculation of statistics followed by the execution of our patch-based algorithm inside the reducer. The shuffling procedure, which is included in Algorithm 5, will use the results of this task and should be implemented as a new shuffle operator to be used in the physical plan in place of the existing shuffle mechanism that exists between map and reduce tasks in repartition joins. The optimizer can be configured to add the initial map-reduce task and select our shuffling operator in the produced physical plan using a query hint provided by the user.

Our approach can be highly applicable to a number of different domains such as:

#### 1) IOT ANALYTICS

IoT systems generate massive amounts of heterogeneous and skewed data streams [1], [2]. Efficient processing of this data often involves distributed systems that suffer from load imbalances. Our patch-based algorithm can ensure uniform worker utilization, optimizing operations like aggregations and joins on IoT data.

#### 2) LARGE-SCALE SQL OPERATIONS

In database management systems, particularly in massively parallel processing (MPP) architectures [3], [10], [11], [13], [22], operations such as joins, group-by, and aggregations are sensitive to skew. The proposed algorithm offers a robust alternative to traditional shuffle mechanisms, ensuring efficient query execution under skewed workloads.

#### 3) MACHINE LEARNING PIPELINES

Distributed training or preprocessing pipelines often encounter skewed datasets [6], [7]. The patch-based technique minimizes training delays by ensuring an even distribution of data batches across nodes. Moreover, while the current implementation is tailored for join operations, the underlying principles could inspire optimizations in

[3]https://kubernetes.io/
[4]https://spark.apache.org/docs/latest/job-scheduling.html

other areas, such as gradient aggregation in distributed deep learning which follows a similar computational approach [46].

#### 4) LOG PROCESSING AND REAL-TIME ANALYTICS

Distributed systems processing application logs, clickstream data, or telemetry often face skew in partitioned data [8], [47], [48]. The proposed method reduces bottlenecks, improving the latency and throughput of these systems.

## V. EVALUATION

In this section we perform an extensive experimental evaluation of our solution. In V-A we describe the datasets we used and in V-B our experimental setup. In V-C we evaluate our patch-based repartitioning algorithm in terms of the quality of the results as well as the execution speed. Regarding the join operation, in V-D we examine the load balancing that occurs with our patch-based join algorithm compared with the 'naive' hash-based repartition join using varying skew factors for the datasets. In V-E we also examine the size of the data that were transferred over the network, which we will refer to as shuffle data in the rest of this section, comparing our algorithm with the simple hash-based technique using varying skew factors too. In V-F we evaluate the overall performance of our implementation compared to the simple hash-based repartition join. For this purpose, a linear model is used to simulate the total time required for the data exchange and reduce-side processing in a distributed cluster. In V-G we examine the scalability of our patch-based join algorithm compared with the hash-based in terms of load and data transfer as the number of workers increases using varying skew factors. Finally, in V-H we summarize the results and discuss edge case performance.

### A. DATASETS AND STATISTICS

Two datasets with a predefined number of records are generated at the start of each experiment. The number of distinct key values is defined beforehand, and the datasets are created using a Zipfian distribution, with the skew controlled by the distribution's exponent ($\theta$). Each dataset is then uniformly divided among $N$ workers, with each worker handling an equal share in memory, mimicking distributed systems where local disk access is utilized.

Table 4 provides the datasets' characteristics, with the Zipfian distribution implemented using the *ZipfDistribution* class from *org.apache.commons.math3* (v3.4.1). The sample() method retrieves indices for key selection based on the parameters $|V|, \theta_S$ and $|V|, \theta_T$ for datasets $S$ and $T$, respectively.

Post-generation, key frequencies are computed locally and globally to determine group sizes and dimensions. Partial aggregation on workers identifies local key distributions, which are summed globally for weight computation. This process resembles methods in related studies requiring pre-computed group statistics.

**TABLE 4. Number of workers (N) and dataset specifications for the three experimental scenarios.**

| | Join type | $N$ | $\|V\|$ | $\|S\|$ | $\|T\|$ | $\theta_S$ | $\theta_T$ |
|---|---|---|---|---|---|---|---|
| **Scenario 1** | foreign-foreign (skew in one table) | 32 | 200 | 400K | 400K | $[0, \ldots 2]$ | 0.00001 |
| **Scenario 2** | foreign-foreign (skew in both tables) | 32 | 200 | 200K | 200K | $[0, \ldots 1]$ | $[0, \ldots 1]$ |
| **Scenario 3** | primary T - foreign S (skew in one table) | 32 | 200 | 400K | 200 | $[0, \ldots 2]$ | - |

**TABLE 5. Final data movement using the approximate tagging procedure.**

| | Additional shuffle data for various skew levels | | | | | | |
|---|---|---|---|---|---|---|---|
| **Scenario 1** | +0.79% | +0.72% | +0.81% | +0.14% | +0.06% | +0.02% | +0.39% |
| **Scenario 2** | +0.71% | +0.84% | +0.89% | +0.56% | +1.53% | +2.23% | +2.71% |
| **Scenario 3** | +0.02% | -0.09% | +0.04% | -0.18% | -0.57% | -0.17% | -6.54% |

**TABLE 6. Final worker load using the approximate tagging procedure.**

| | Maximum additional worker load for various skew levels | | | | | | |
|---|---|---|---|---|---|---|---|
| **Scenario 1** | 0.07% | 0.08% | 0.06% | 0.09% | <0.01% | <0.01% | 0.16% |
| **Scenario 2** | 0.11% | 0.2% | 0.15% | 0.1% | 0.33% | 0.29% | 0.14% |
| **Scenario 3** | 0% | 0% | 0% | 0% | 0% | 0% | 0% |

**TABLE 7. Patch-based repartitioning algorithm execution time.**

| | Execution time overhead for various skew levels | | | | | | |
|---|---|---|---|---|---|---|---|
| **Scenario 1** | 10.89% | 9.58% | 9.09% | 9.09% | 11.39% | 13.38% | 11.20% |
| **Scenario 2** | 9.67% | 9.50% | 9.72% | 10.95% | 9.52% | 8.98% | 10.84% |
| **Scenario 3** | 36.83% | 21.68% | 25.32% | 24.46% | 27.86% | 23.78% | 29.46% |

Fig. 10 illustrates how skew changes with varying $\theta$. For 32 unique keys and 200K records, a uniform distribution ($\theta = 0$) spreads records evenly, while increasing $\theta$ concentrates records on fewer keys. For example, at $\theta = 2.0$, 62% of records belong to a single key. Though real-world datasets typically exhibit $\theta < 1.0$, we include extreme values to highlight algorithm performance under significant skew.

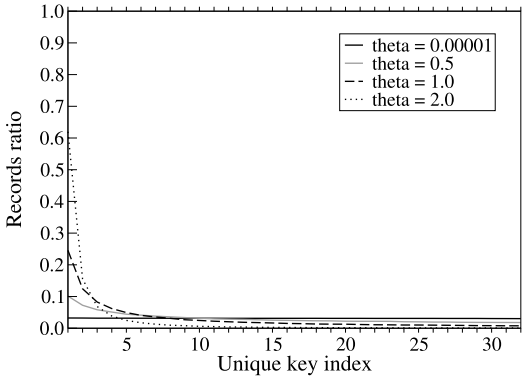



**FIGURE 10. Records distribution in the keys for various skew levels.**

### B. EXPERIMENTAL SETUP

Experiments were conducted on a host with an AMD EPYC 7443P processor, 48 threads, 256 GB DDR4 RAM, and a RAID 10 array of 22 TB storage. Each experiment defined

parameters for dataset size, number of unique keys, workers, and skew factor ($\theta$) for the zipfian-distributed datasets $S$ and $T$. Algorithms were implemented in Java, with workers represented as threads, each processing a subset of the input data in memory.

Three join scenarios (Table 4) were simulated, inspired by entity relationships described in [49]. The first scenario varied the skew in $S$ ($\theta_S : [0, \ldots, 2]$) while $T$ had no skew, representing a *foreign-foreign* key join. The second scenario applied equal skew to both tables ($\theta : [0, \ldots, 1]$), also modeling a *foreign-foreign* key join. The third scenario varied the skew in $S$ ($\theta_S : [0, \ldots, 2]$) while $T$ had unique keys, forming a *primary-foreign* key join.

To ensure the statistical significance of our experimental results, each experiment was executed a sufficient number of times (typically 10-20 runs, depending on the observed variance). The presented results represent the average performance across these runs. Furthermore, we conducted statistical tests, such as calculating the standard deviation and confidence intervals, to verify the consistency and reliability of our findings. In all cases, the variations observed across multiple runs were minimal, and the results consistently adhered to the trends reported in the paper, confirming the robustness of our conclusions.

### C. PATCH-BASED REPARTITIONING ALGORITHM

The patch-based repartitioning algorithm was evaluated for its performance in approximate tagging, focusing on result quality and execution time. Quality was assessed by comparing the final subgroup record placement with the theoretical optimal, using metrics such as additional shuffle data size (Table 5) and maximum worker load (Table 6). Divergences were minimal, with shuffle data size differences mostly below 1% and a maximum of 2.71%. In some cases, particularly in scenario 3, less data was shuffled than theoretically expected. Worker load variations were even smaller, with a maximum increase of 0.33%.

Execution time was benchmarked as a percentage of the reduce-side join operation's duration, with overhead averaging 10% for scenarios 1 and 2, peaking at 13.38%. Scenario 3 exhibited higher percentages due to its shorter join execution times. This overhead is considered a reasonable tradeoff for mitigating skew-induced delays, a benefit confirmed in Section V-F, where overall execution time comparisons with the skew-sensitive hash-based join are provided.

The next sections analyze worker performance during the join process, contrasting the patch-based and hash-based approaches.

### D. LOAD BALANCING

This section compares the load distribution across workers in the patch-based and hash-based join implementations as the skew factor increases.

The maximum observed worker load, normalized to the average load in each scenario, demonstrates that the patch-based approach achieves perfect load balancing across all skew factors, as shown in Figs. 11a to 11c. In contrast, the hash-based join experiences substantial imbalance, with the maximum worker load rising sharply to as much as 19.4 times the average under high skew conditions.

The median worker load, similarly normalized, reveals further advantages of the patch-based implementation. As shown in Figs. 12a to 12c, the patch-based method consistently maintains balanced median loads. Conversely, the hash-based join suffers from a skew-induced concentration of load, which causes the median to decrease as fewer workers handle most of the data while others remain idle. Even under minimal skew, the hash-based median is suboptimal compared to the patch-based implementation.

The consistency of these results across all scenarios highlights the effectiveness of the patch-based algorithm in mitigating the effects of skew.

### E. DATA SHUFFLING

In this section we present the total number of records transferred over the network and their dispersion to the workers, comparing our implementation and the 'naive' hash-based join as the skew factor increases.

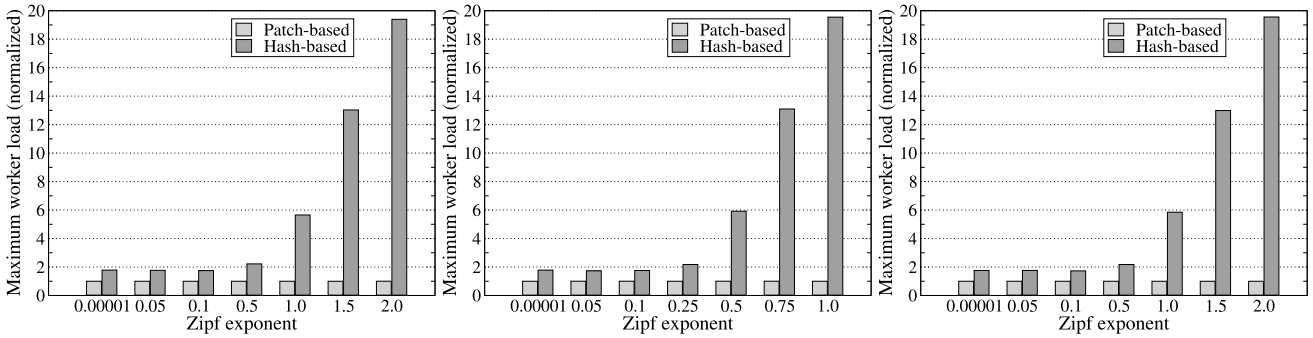
We start by examining the total number of records that need to be transferred as the skew factor increases for the two algorithms. Figs. 13a to 13c present the results for scenarios 1, 2 and 3 respectively. The number of records is normalized using the initial number of records which is $|S| + |T|$.

As we can observe in Fig. 13a, in the case of the hash-based join, the total number of records is always less but close to the initial number of records. This is expected as each record is sent exactly once to the relative worker (only in case it is not already available locally). With our algorithm, the total number of records is always higher, as expected, due to the replication that happens.

Comparing with Fig. 13a, in Fig. 13b there is an important increase in the total shuffle data size for scenario 2 where skew is present in both tables. As the skew level increases, we can notice that the shuffle data size increases significantly, and it is almost 6 times larger in the case of $\theta = 1.0$ whereas the maximum increase in scenario 1 is 1.6 times.

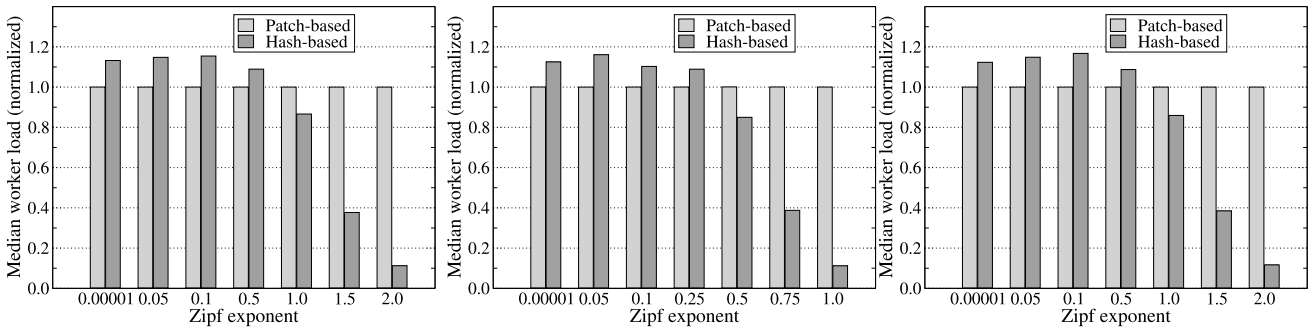
Finally, in Fig. 13c we observe a significant differentiation from the previous scenarios as the total size of records transferred is always smaller compared with the naive technique. When there is no skew in table S our algorithm transfers almost the same amount as the naive technique, and as the skew increases in table S our algorithm tends to transfer even less data especially for skew factors larger than 0.5. Our algorithm profits from the exploitation of data locality for table S in this case which combined with the small size of T records that will be replicated, leads to a significant reduction in the amount of data transferred.

Next, we examine the statistical dispersion of the received number of records in each worker using the Gini index. A Gini



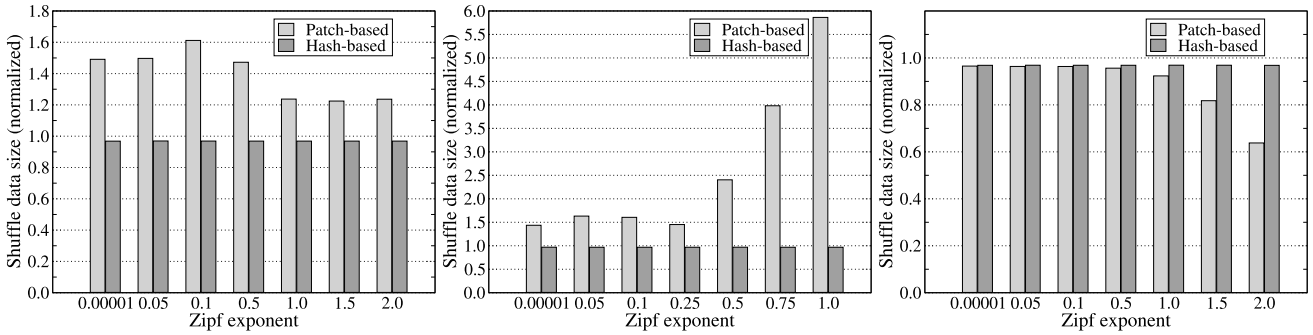

**a.** Table S ($\theta_S$) (scenario 1). **b.** Both tables S and T (scenario 2). **c.** S($\theta_S$) with unique keys in T (scenario 3)

**FIGURE 11.** Maximum worker load with the patch-based and hash-based join algorithms for different skew factors.




**a.** Table S ($\theta_S$) (scenario 1) **b.** Tables S and T (scenario 2) **c.** Table S ($\theta_S$), unique keys in T (scenario 3)

**FIGURE 12.** Median worker load with the patch-based and hash-based join algorithms for different skew factors.



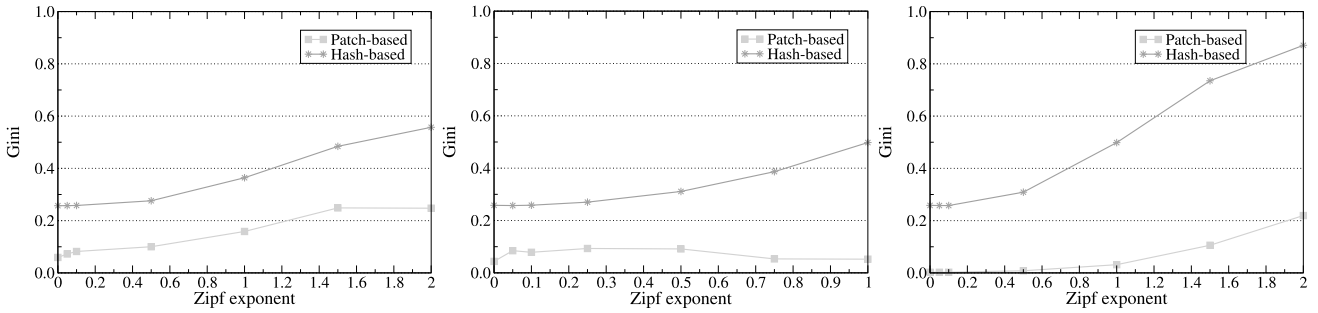

**a.** Table S ($\theta_S$) (scenario 1). **b.** Both tables S and T (scenario 2). **c.** S($\theta_S$) with unique keys in T (scenario 3).

**FIGURE 13.** Produced shuffle data size of the patch-based and hash-based join algorithms for various skew factors applied.

value close to zero represents an exactly equal number of records sent to each worker, while higher Gini values indicate greater inequality in the number of records per worker.

In Fig. 14a we present the Gini index value as the skew factor $\theta_S$ increases for scenario 1. The results show that our patch-based join algorithm distributes data more evenly to workers irrespective of the level of skew compared with the hash-based join, therefore presenting a constantly lower Gini index value. As the skew factor increases the Gini index increases for both algorithms for several reasons.

In more detail, at large skew factors our algorithm tends to sequentially create subgroups for the largest group (by splitting the S skewed side) which are assigned to all the workers in turn leading to more equally sized subgroups being distributed to the workers. Our algorithm does not present the optimal distribution however, as the main goal is the even load balancing and the exploitation of data locality to reduce the total amount of data transferred, given that the replication costs cannot be avoided. The hash-based join, on the other hand, randomly distributes the initial groups to




**a** Table S ($\theta$ ) (scenario 1). **b** Both tables S and T (scenario 2). **c** S($\theta$ ), unique keys in table T (scenario 3)

**FIGURE 14.** Gini index for the shuffle data size sent to each worker with the patch-based and hash-based join algorithms for various skew factors applied.

the workers, and as the skew factor increases these groups are more unequally sized leading to more data imbalances.

At small skew factors, our algorithm is suboptimal in terms of data distribution due to the priority given to local processing which leads to over-partitioning and over-replication. The hash-based join is also suboptimal at small skew factors due to the random distribution of the almost equally sized groups to the workers using a hash function.

The results are similar for scenario 2 in Fig. 14b. The main difference from Fig. 14a is that we observe an even better Gini index value with our algorithm for large skew factors since equally sized subgroups of the largest group are created (by splitting the S and T sides in turn) and are distributed to all the workers.

Regarding scenario 3, in Fig. 14c, the hash-based join presents an even higher Gini index value as the skew factor increases to values larger than 0.1, i.e. greater imbalance, which is normal. For example, when $\theta_S = 2.0$, at least 62% of the total data will be placed in one worker based on the distribution of Fig. 10 (T records are negligible). Our algorithm, on the contrary, presents lower Gini index values. This is because the unique keys of table T are placed one in each worker, and this leads to an initial creation of exactly one subgroup in each of these workers for his unique key during the first phase of our algorithm. This leads to a perfect load balancing for low skew levels as equally sized groups will be evenly distributed to the workers sequentially.

Combining the results of this section, an important observation is derived. Although with our algorithm at large skew factors the total size of shuffle data is increasing up to 6X compared with the hash-based technique, these data are more equally distributed to the workers compared with the hash-based technique where a significant amount of data will be sent to a single worker. In other words, the network traffic will not be concentrated on one worker with our algorithm, therefore the observed impact of the increased size will be significantly less visible. This observation will be further established with the results of the following section (V-F).

### F. EXECUTION TIME

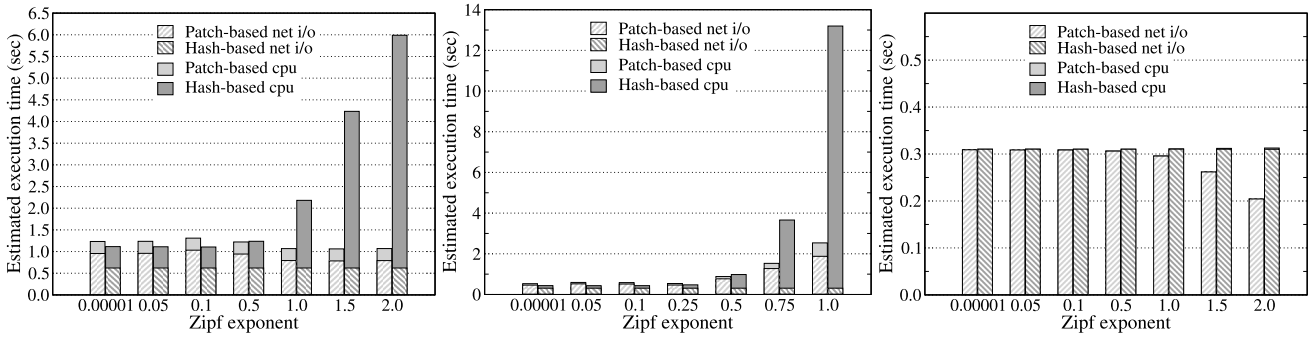
The analysis of execution time focuses on the critical phases of data shuffling and reduce-side execution, which distinguish the patch-based and hash-based join algorithms. Common overheads, such as in-memory data loading and initial statistics calculation, are excluded from the analysis due to their minimal impact. The study assumes a distributed cluster setup with single-threaded 2.7GHz cores, a 10Gbit Ethernet network, and 1KB record sizes. Prior studies [50], [51], [52] indicate that hash-based joins require 30 CPU cycles per join output tuple. A linear model estimates execution time by combining network transfer time and the processing time of the slowest worker, as given by Equation 7. The model considers data size and processing rates to quantify the phases affecting execution time.

$$t(x, y) = \frac{1000 \cdot y}{1.25 \cdot 10^9} + \max_w \left( \frac{30 \cdot x(w)}{2.7 \cdot 10^9} \right), \tag{7}$$

$x$ is a list containing the number of output tuples that each worker $w$ calculated, and $y$ is the total number of records that were sent over the network. The first part of the equation estimates the network transfer time that happens during the shuffle phase, i.e., during the copying of the remote group data. The time depends on the data size calculated in the numerator (the total number of records multiplied with the record size) divided by the link speed set to 1.25GB/s. The second part of the equation estimates the time needed to perform the reduce-side calculations: it depends on the worker with the biggest output to produce. In this worker the time depends on the number of output tuples multiplied with the processing time per tuple (i.e., around 30 clock cycles) divided by the CPU clock speed, i.e., 2.7GHz.

In Scenario 1 (Fig. 15a), the patch-based algorithm shows a slower data shuffling phase due to increased data replication but benefits from better load balancing, being only 1.1 times slower than the hash-based algorithm for minimal skew. For higher skew factors ($\theta_S \geq 0.5$), the patch-based approach surpasses the hash-based method, with improvements of 51% for $\theta = 1.0$ and 82% for $\theta = 2.0$. The overhead from the patch-based repartitioning algorithm remains negligible, as seen in Fig. 15a, with CPU times below 0.5 seconds.

In Scenario 2 (Fig. 15b), the findings align with those of Scenario 1. The patch-based algorithm starts 1.2 times slower with minimal skew but achieves significant gains, being 58% faster for $\theta = 0.75$ and 81% faster for $\theta = 1.0$. Again, the




**a.** Table S ($\theta_S$) (scenario 1) **b.** Both tables S and T (scenario 2) **c.** S($\theta_S$) with unique keys in T (scenario 3)

**FIGURE 15.** Estimated execution time split into shuffling and execution phases with the patch-based and hash-based join algorithms for different skew factors.

**TABLE 8.** Number of workers ($N$) and dataset specifications for the scalability experiments.

| Join type | $N$ | $\|V\|$ | $\|S\|$ | $\|T\|$ | $\theta_S$ | $\theta_T$ |
|---|---|---|---|---|---|---|
| foreign-foreign (skew in one table) | $[16, \ldots 256]$ | 3000 | 1M | 150K | $[0, \ldots 1]$ | 0.00001 |

repartitioning overhead remains negligible, with CPU times consistently below one second.

In Scenario 3 (Fig. 15c), the patch-based algorithm consistently performs as well as or better than the hash-based approach. For minimal skew, performance is comparable, but for $\theta \geq 1.0$, the patch-based method achieves up to 35% faster execution for $\theta = 2.0$. Here, the shuffling phase dominates the overall execution time, as reduce-side processing times are minimal. The repartitioning overhead, approximately 30% of the reduce-side execution time, is negligible in this case as well.

The results illustrate the patch-based algorithm's ability to handle skew efficiently, outperforming the hash-based method in high-skew scenarios while maintaining minimal overhead across all scenarios.

### G. SCALABILITY

This section evaluates the scalability of the patch-based join algorithm by analyzing shuffled data distribution and load handling as the number of workers increases, comparing it to the hash-based join across varying skew factors in dataset $S$. A specialized dataset configuration, detailed in Table 8, facilitated the experiments.

The shuffle data distribution was analyzed using the Gini index, as shown in Fig. 16a, which plots the index across increasing worker counts and skew factors ($\theta_S$). For small skew ($\theta_S \leq 0.5$), the Gini index rises slowly and remains below 0.4 even at $N > 64$. For higher skew, while the index increases with worker count, the growth rate diminishes as $N$ exceeds 64. These results align with findings in Section V-E, demonstrating the algorithm's ability to maintain balanced shuffle data distribution across large worker pools.

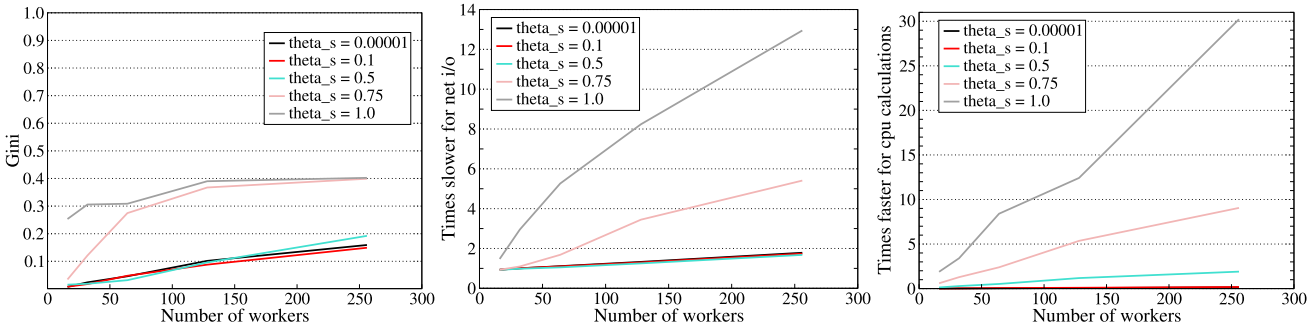
The execution time was analyzed for both data shuffling and reduce-side phases, with Fig. 16b showing the shuffling time ratio between the patch-based and hash-based algorithms. As worker count increases, the patch-based algorithm requires more time for data shuffling due to increased subgroups and data replication, especially for high skew factors ($\theta_S > 0.5$). For $\theta_S = 1.0$ and $N = 256$, it is up to 13 times slower than the hash-based approach in this phase. In contrast, Fig. 16c illustrates the reduce-side execution time ratio, where the patch-based algorithm consistently outperforms the hash-based join due to superior load balancing. This advantage is most pronounced at high skew factors, reaching a 30X improvement for $\theta_S = 1.0$ and $N = 256$. The disparity arises as the hash-based algorithm's load imbalance worsens with increasing worker counts, exacerbating bottlenecks.

Overall, the patch-based join algorithm scales effectively, handling larger worker counts while maintaining balanced data distribution. Although slower during data shuffling, it compensates with significantly faster reduce-side execution, as evident when comparing the y-axes of Figs. 16b and 16c. This trade-off makes the algorithm well-suited for environments with high worker counts and pronounced data skew.

### H. SUMMARY AND DISCUSSION

The final key points for the performance of our algorithm compared with the naive hash-based approach are the following:

- The results in Figs. 13a to 13c and 14a to 14c confirm that our algorithm automatically adapts to different skew levels by dynamically rebalancing data distribution across workers. The observed reduction in imbalance, even under severe skew conditions, highlights the effectiveness of our approach in handling varying data distributions without requiring manual parameter tuning.
- Our patch-based join algorithm is on average 45% faster than the hash-based for a moderate to high skew factor ($\theta = 1.0$). Moreover, it is 81% faster for an extremely




**a.** Gini index for the shuffle data size sent to each worker with the patch-based join algorithm

**b.** Time ratio required for data transfer with the patch-based join algorithm to the time required with the hash-based join

**c.** Time ratio required for reduce-side calculations with the hash-based join algorithm to the time required with the patch-based join

**FIGURE 16.** Shuffle data size Gini index and time ratios compared to the hash-based approach for different skew factors in tables S ($\theta_S$) as the number of workers $N$ increases.

high skew factor ($\theta = 2.0$) in a foreign-foreign key join scenario (scenario 1, Fig 15a).

- Our patch-based join algorithm is on average 10% slower than the hash-based approach when there is zero or little skew in the data (scenario 1, Fig 15a).
- The network-related overhead is, as expected, greater with our algorithm in foreign-foreign key join scenarios due to replication of records (scenarios 1 and 2, Fig 15a and Fig 15b respectively). However, it is smaller in primary-foreign key join scenarios (scenario 3, Fig. 15c).
- At high skew levels the overall performance is mainly affected by the load balancing on the workers and less affected by the network overheads in foreign-foreign key join scenarios (2 and 3 in Fig 15b and Fig 15c respectively).
- At high skew levels the skew-related load imbalance and execution delay is eliminated with our algorithm (right sides of Figs 15a, 15b and 15c).
- The time overhead that is incurred with the execution of our patch-based algorithm is negligible.
- Our algorithm can scale to a large number of both cluster size (experimentally tested up to 250 nodes, which is a reasonable data center cluster size) and dataset size (in the order of million keys), as described in Table V-G and observed in Figures 16a, 16b and 16c.

Regarding the performance of our algorithm in edge cases, there are two cases in which we evaluated our implementation.

*Edge case 1.* As we have previously mentioned, in case of zero skew the optimal performance is expected to be achieved by the standard hash-based repartitioning algorithm. Our approach is on the other hand built to manage optimally more heavily skewed datasets. We evaluate our algorithm's performance against the hash-based approach using zero or little skew in all the conducted experiments and our algorithm is on average 10% slower than the hash-based approach.

*Edge case 2.* The second edge case that we evaluate is the experimental Scenario 3 in Table 4, where we execute a primary-foreign key join. Such a join could be performed using a broadcast join (Alg. 3.2 in [14]) instead of a repartition join for optimal performance if the table containing the primary keys is adequately small. Although it is out of scope of this work to be compared with the broadcast join implementation, our results indicate that our patch-based repartitioning algorithm presents improved performance compared with the hash-based approach for skewed datasets. Therefore, in cases like scenario 3 where a broadcast join cannot be used, our algorithm can improve the performance of a repartition join.

### 1) STRAGGLER NODE AND RECOVERY MECHANISMS FOR FAULT TOLERANCE

The proposed patch-based shuffle and join algorithm assumes a well-maintained and healthy cluster environment, operating under typical data center conditions with high-speed, low-latency network connectivity (e.g., 10G links or similar). Handling stragglers or recovery of failed data transfers are a complex system-level challenge that is orthogonal to the core contribution of this work. These scenarios are already addressed by distributed processing engines such as Apache Spark, which incorporate speculative execution to mitigate the impact of slow or delayed nodes even in case of failures (`spark.speculation` configuration entry [53]). Our algorithm operates at the application logic level and relies on these system-level features for fault tolerance.

### 2) CROSS-RACK TRANSFERS AND NETWORK CONGESTION MANAGEMENT

Our algorithm is designed to minimize overall data movement, regardless of whether the network spans multiple racks. Network congestion and cross-rack transfer optimization are orthogonal concerns that are typically addressed at the network infrastructure level or by specialized distributed file systems and processing engines (see for example the configuration entry `spark.locality.wait` of Apache

Spark [53]). These considerations could complement our approach but are not required to demonstrate its efficacy in load balancing, network movement minimization, and skew handling.

#### 3) NETWORK BOTTLENECK QUANTIFICATION FOR LARGE-SCALE DEPLOYMENTS

We have carefully considered the potential for network bottlenecks in large-scale deployments and provide the following analysis: The algorithm minimizes data movement by prioritizing local processing, therefore utilizing mostly the hard disk speed and utilizing the network only for remote (global) patches $P_v^{mov}$.

CPU speed: Assuming a 2.7 GHz CPU and processing efficiency of approximately 30 cycles/tuple, the system can process approximately 30 million tuples per second per worker hardware thread. For 1 KB tuples, this equates to a throughput of roughly 90 GB/s per worker hardware thread. Disk R/W speeds: A standard HDD provides 100-200 MB/s, while an SSD provides 500 MB/s to 7 GB/s depending on the configuration. The ratio of CPUs to disks in the data center determines the true bottleneck in such scenarios. Even in a case where there is a one-to-one mapping of hardware threads to SSD disks, the limiting factor seems to be the SSD disk speed. Network speed: In a typical leaf-spine topology with 25 GbE Top-of-Rack (ToR) switches the network provides a theoretical throughput of 25 Gb/s (approximately 3.125 GB/s per link). This throughput is sufficient to handle the inter-node communication requirements without causing a bottleneck, since a) traffic will be mainly between CPU and local HDDs minimizing network communication and b) the limiting factor seems to be the HDD speed R/W. Given the above, our algorithm does not introduce network bottlenecks even in large-scale deployments. The network capacity in modern data centers significantly exceeds the processing and disk I/O limits, ensuring smooth data movement during patch redistribution.

## VI. CONCLUSION

In this work, we introduced a novel patch-based repartitioning technique designed to address the critical challenges of load imbalance and excessive network traffic in distributed join operations over skewed datasets. Our approach leverages detailed data distribution and location statistics, enabling informed subgroup assignment to workers based on local weight $l_a(n)$ and an adaptively calculated threshold $C_{max}$. This deterministic approach eliminates the need for iterative cost modeling or heuristic parameter tuning, dynamically adapting to varying levels of data skew.

### A. RESEARCH CONTRIBUTIONS

Our research makes several key contributions:

- **Novel Patch-based Shuffling Algorithm:** We developed a novel patch-based shuffling algorithm that operates **before** any reduce-side operation, effectively pre-processing data for optimal join execution. This algorithm consists of three distinct stages: statistic calculation, patch-based partitioning, and data transfer/join computation.
- **Two-Phased Subgroup Assignment:** We introduced a two-phased approach for subgroup assignment. The first phase prioritizes local processing by assigning subgroups to workers with the most local data. The second phase balances load globally by distributing remaining data to workers with the least load.
- **Formal Analysis and Guarantees:** We provided a rigorous theoretical framework, including proofs of convergence and scalability, demonstrating that our algorithm guarantees load balancing, limits network data movement, and effectively handles data skew. Specifically, we prove that the maximum per-worker load is bounded and that the total network cost is $O(\frac{|S|+|T|}{N})$.

### B. PRACTICAL IMPACT AND ADVANCEMENTS

Our patch-based repartitioning algorithm represents a significant advancement in distributed data processing by providing a practical, scalable, and theoretically grounded solution for handling skewed datasets. Unlike conventional hash-based approaches, which suffer from severe load imbalance and high network costs under extreme skew, our method dynamically adapts to data distribution, significantly improving performance and efficiency. Seamless integration into existing distributed frameworks and demonstrated performance gains of up to 81% faster execution highlight its potential for real-world adoption in large-scale data processing systems.

### C. RESEARCH LIMITATIONS

Despite its strengths, our approach has certain limitations. The current implementation assumes a well-maintained, high-speed cluster environment and relies on external fault-tolerance mechanisms (e.g., speculative execution) to handle stragglers or failures. Consequently, its performance may be less optimal in heterogeneous or unreliable network settings characterized by frequent network partitions or highly variable link latencies. Furthermore, while our design effectively minimizes inter-node data transfers, it does not explicitly optimize for cross-rack transfers or severe network congestion, leaving room for further system-level enhancements. Additionally, in scenarios with little or no data skew, our patch-based method incurs a modest overhead (approximately 10% slower) compared to traditional hash-based repartitioning.

### D. FUTURE RESEARCH DIRECTIONS

Our research lays the foundation for further exploration in distributed join optimization. Immediate next steps and promising research directions include:

- **Integration with Existing Distributed Systems:** Adapting our approach for use in widely adopted

distributed join frameworks such as Apache Spark and Trino. Additionally, assessing how easily it can be incorporated into different execution engines with minimal modifications and evaluating its performance on realistic and benchmark workloads such as TPC-DS and TPC-H to demonstrate its practical advantages.

- **Enhanced Fault Tolerance:** Investigating deeper integration of fault-tolerance mechanisms within the patch-based algorithm to improve robustness in heterogeneous or failure-prone environments.
- **Extension to Other Join Types:** Extending the patch-based approach to support a broader range of join operations (e.g., broadcast joins, outer joins and theta joins) and evaluating its effectiveness across diverse data distributions and workload scenarios.
- **Adaptive Hybrid Approaches:** Developing adaptive frameworks that dynamically select between patch-based and traditional hash-based methods based on real-time skew detection, thereby optimizing performance across all levels of data skew.
- **Cross-Rack and Congestion Optimization:** Exploring network-aware optimizations that explicitly address cross-rack transfers and congestion through topology-aware subgroup assignment and network-level integration.

### E. CONCLUSION AND BROADER IMPACT

In summary, our patch-based repartitioning technique presents a theoretically robust and practically efficient solution for accelerating distributed joins on skewed datasets. By addressing fundamental bottlenecks in skew handling, our work contributes to the broader field of distributed computing, enabling more adaptive, efficient, and scalable data processing. We believe this research opens up new directions in self-tuning distributed systems, paving the way for future innovations in large-scale data management, workload balancing, and network-efficient computation.

## APPENDIX A
## THEORETICAL ANALYSIS AND CONVERGENCE PROOF

Using the notation from Table 2 we provide a detailed analysis of the patch-based shuffle and join algorithm with respect to load balancing, network movement minimization, and skew handling. We conclude with the provision of theoretical guarantees regarding the algorithm's convergence and scalability.

### A. LOAD BALANCING

**Objective:** Distribute the workload across $N$ workers such that no worker's load exceeds the maximum capacity $C_{max}$.

**Analysis:** For each worker $n$, its total load is defined as:

$$\text{Load}_n = \sum_{a \in V} |S_a(n)| \cdot |T_a(n)|.$$

The algorithm ensures that:

$$\text{Load}_n \leq C_{max}, \quad \forall n \in \{1, \ldots, N\}.$$

The repartitioning operates in two phases:

- **Phase 1: Local assignment**
  For each $P_a$, the subgroup $P_a^{lo} \subseteq P_a$ is assigned to workers $n$ with the highest $l_a(n)$ (local weight), provided:
  $$|S_a(n)| \cdot |T_a(n)| \leq C_{max}.$$
  If $|P_a| > C_{max}$, the subgroup $P_a^{lo}$ is split into smaller chunks such that:
  $$|P_a^{lo}| = \min(l_a(n), C_{max}).$$
- **Phase 2: Global assignment**
  Remaining data $g_a$ (unassigned from $P_a$) is distributed to workers with the most remaining capacity:
  $$|P_a^{mov}| = \min(g_a, C_{max}).$$

**Convergence**: The iterative updates to $g_a$ and $C_{max}$ ensure all data is assigned:

$$g_a \to 0 \quad \text{as} \quad \sum_{n \in N} C_{max} \geq \sum_{a \in V} g_a.$$

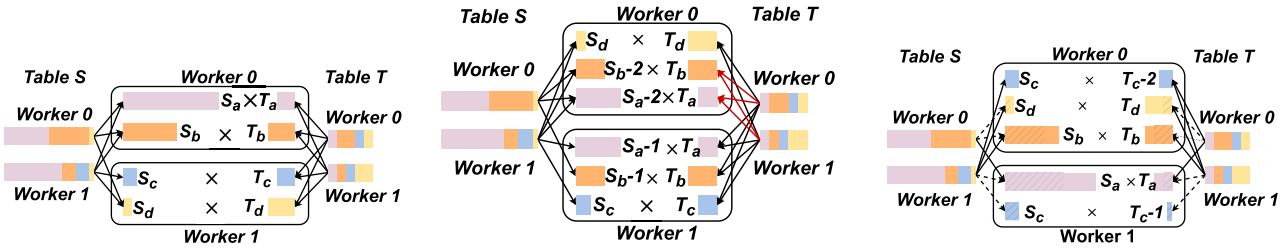
The total load $L = \sum_{a \in V} g_a$ represents the combined size of all join groups that need to be distributed across $N$ workers. The algorithm iteratively assigns parts of each join group $g_a$ to workers, prioritizing workers with available capacity. Each iteration reduces the unassigned portion of $g_a$. Specifically, in each step, the algorithm assigns a fraction of $g_a$ such that the total load on the selected worker does not exceed $C_{max}$. Since $\sum_{n \in N} C_{max} = N \cdot C_{max} = L + N$, the total capacity across all workers exceeds the total workload $L$. This guarantees that there is always sufficient room across the workers to assign all parts of $g_a$, even under skewed distributions. Convergence is ensured because $\sum_{n \in N} C_{max} \geq \sum_{a \in V} g_a$, meaning the system has the capacity to handle the entire workload. Thus, the condition $\sum_{n \in N} C_{max} \geq \sum_{a \in V} g_a$ ensures that the total workload $g_a$ is distributed across workers and $g_a \to 0$ as the algorithm progresses.

### B. NETWORK MOVEMENT MINIMIZATION

**Objective**: Minimize the number of records $|S_a|$, $|T_a|$ transferred between workers during shuffling.

**Analysis:**

- **Phase 1: Local assignment**
  Prioritize workers with high local weight $l_a(n)$ for assigning subgroups. This minimizes the external data fetched $(s_a(w'), t_a(w'), w' \neq n)$
- **Phase 2: Global assignment**
  For $g_a > 0$, subgroups $P_a^g$ are assigned globally. While this increases the external data fetched, the two-phase approach ensures this represents a smaller fraction of the total:
  $$\text{Network cost} = \sum_n \sum_{a \in V} \left(s_a(w') + t_a(w')\right), \quad w' \neq n.$$




**a.** Hash-based repartitioning (worst case) **b.** Common subset-replicate repartitioning **c.** Patch-based repartitioning, data locality

**FIGURE 17.** Groups and subgroups assignments with different repartitioning algorithms.

**TABLE 9.** Parameters used for proof of concept.

| | |
|---|---|
| N | 2 |
| $\|V\|$ | 4 |
| $\|S\|$ | 40 |
| $\|T\|$ | 20 |
| $\|S_a\|, \|S_b\|, \|S_c\|, \|S_d\|$ | 23,12,3,2 |
| $\|T_a\|, \|T_b\|, \|T_c\|, \|T_d\|$ | 4,6,4,6 |
| $s_a, s_b, s_c, s_d$ | [10,13],[9,3],[0,3],[1,1] |
| $t_a, t_b, t_c, t_d$ | [2,2],[4,2],[2,2],[2,4] |
| $l_a, l_b, l_c, l_d$ | [20,26],[36,6],[0,6],[2,4] |
| $g_a, g_b, g_c, g_d$ | 92,72,12,12 |
| $L$ | 188 |
| $C_{max}$ | 95 |

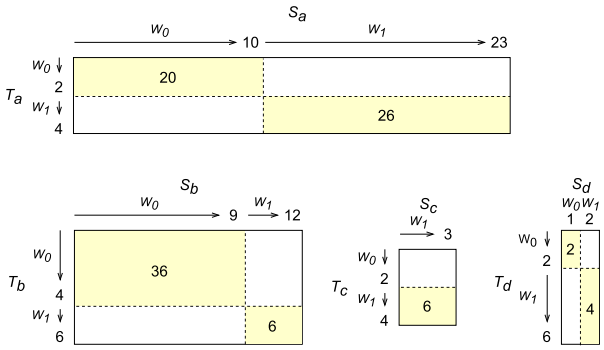



**FIGURE 18.** Representation of the initial four groups and their sizes, as well as the data placement in the two workers.

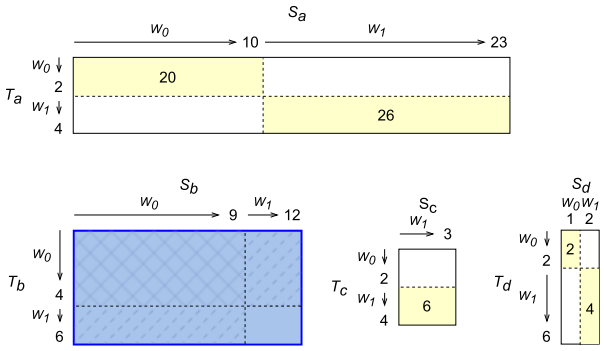



**FIGURE 19.** The first subgroup created for group $P_b$, assigned to worker 0.

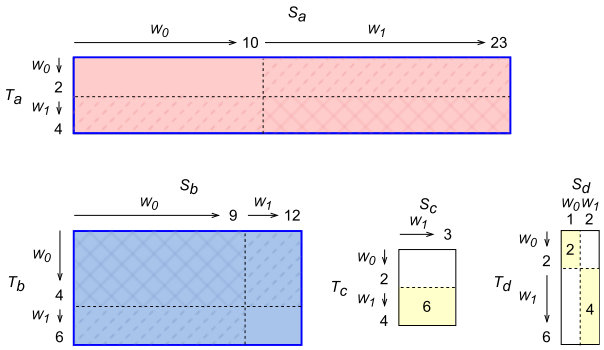



**FIGURE 20.** The second subgroup created for group $P_a$, assigned to worker 1.

Our algorithm ensures that the Locality Factor (defined in Section IV-D) approaches 1 by maximizing the usage of locally available data before fetching data from external sources.

$$\text{Locality Factor} = \frac{\sum_n l_a(n)}{\sum_n (s_a(n) + t_a(n))}.$$

The algorithm ensures that $\sum_n l_a(n)$ is maximized because workers with high local weights are prioritized for processing $P_a$ in the first phase.

External data $(s_a(w'), t_a(w'))$ is fetched only when strictly necessary, limiting the increase in $\sum_n (s_a(n) + t_a(n))$ in the denominator.

For skewed groups, the algorithm assigns smaller subgroups $p_{a,i}$ to multiple workers in such a way that each subgroup still processes as much local data as possible.

As the number of workers $(N)$ increases, the chance of finding a worker with a high local weight $l_a(n)$ also increases, further improving locality.

Therefore, under optimal conditions:

$$\text{Locality factor} \to 1 \quad \text{as} \quad l_a(n) \approx s_a(n) + t_a(n)\ \forall n.$$

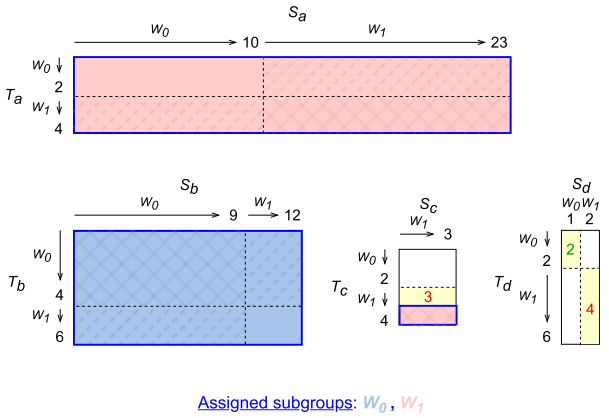



**FIGURE 21.** The third subgroup created for group $P_c$, assigned to worker 1. The areas noted with '4' and '3' cannot be assigned to worker 1 to exploit the data locality, therefore the area noted with '2' will be next studied for worker 0.

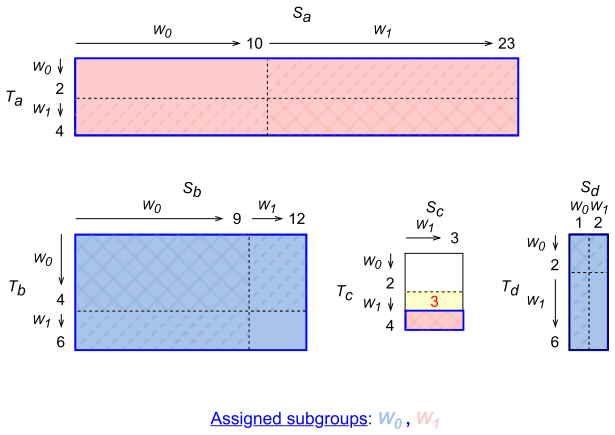



**FIGURE 22.** The fourth subgroup created for group $P_d$, assigned to worker 0. The first phase is complete as no more locality-based subgroups can be created.

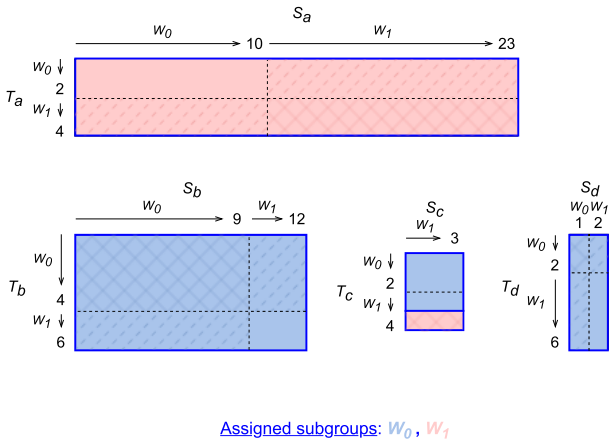



**FIGURE 23.** The fifth and last subgroup is created for group $P_c$ without considering data locality and is assigned to worker 0.

This implies that most of the data processing happens locally, with minimal external data movement.

**Key result**: Compared to hash-based shuffling (where all data is randomly distributed), patch-based repartitioning reduces data movement by maximizing locality. This happens because only the data required to balance workloads is moved, governed by the thresholds in Algorithms 3 and 4.

Specifically:

$$\text{Total bytes transferred (patch-based)} \approx O\left(\frac{|S|+|T|}{N}\right),$$

compared to $O(|S|+|T|)$ for hash-based.

$$\text{Locality factor} \to 1 \quad \text{as} \quad l_a(n) \approx s_a(n) + t_a(n)\ \forall n.$$

### C. HANDLING SKEW

**Objective:** Repartition skewed datasets $P_a$ without requiring pre-knowledge of $S_a, T_a$.

**Analysis:**

**Skewed distributions:**

Join group $P_a$ is skewed if:

$$g_a = |S_a| \cdot |T_a| \gg \frac{\sum_{a\in V} g_a}{|V|}.$$

Common skew patterns include: - Zipfian ($|S_a| \propto a^{-\alpha}$, $\alpha > 1$).

**Patch-based handling:** For each skewed group $P_a$, the algorithm ensures:

$$|P_a^{lo}| + |P_a^{mov}| = |P_a|, \quad |P_a^{lo}| \cdot |P_a^{mov}| \le C_{max}.$$

This divides $P_a$ among multiple workers, reducing the load imbalance caused by skew.

**Iterative refinement:**

The algorithm dynamically adjusts subgroup sizes ($|P_a^{lo}|, |P_a^{mov}|$) to balance the load:

$$\max_n \text{Load}_n = O\left(\frac{\sum_{a\in V} g_a}{N}\right).$$

### D. KEY RESULTS - THEORETICAL GUARANTEES

**Load balancing:**

$$\max_n \text{Load}_n = O\left(\frac{\sum_{a\in V} g_a}{N}\right) + O(1).$$

**Network efficiency:**

The total network cost is bounded by:

$$C_{network} = O\left(\frac{|S|+|T|}{N}\right). \tag{8}$$

**Skew robustness:**

The algorithm handles skewed datasets by distributing $g_a$ across multiple workers without exceeding $C_{max}$:

$$|P_a^{lo}| + |P_a^{mov}| = |P_a| \quad \text{with} \quad |P_a^{lo}| \cdot |P_a^{mov}| \le C_{max}.$$

**Scalability:**

The algorithm scales effectively with increasing $|S|, |T|$ and $N$, maintaining:

$$T_{execution} = O\left(\frac{|S|+|T|}{N}\right). \tag{9}$$

**Algorithm 2** Patch-Based Repartitioning

**Input** $|S_v|, |T_v|, s_v(n), t_v(n), l_v(n), g_v$ for each worker $n = \{1, .., N\}$ and join attribute value $v \in V$, maximum capacity $C_{max}$
**Output** Set $R$ containing worker-subgroup pairs
**Procedure** patchRepartitionAndAssign():

$R \leftarrow \emptyset$
$cap(n) \leftarrow C_{max}, n = \{1, .., N\}$
$q \leftarrow orderDescending(l_{v1}(1), .., l_{v1}(N), l_{v2}(1), ..)$
/* 1st phase */
**repeat**
  **for** $l_v(w)$ **in** q **do**
    **if** $cap(w) > |S_v|$ **or** $cap(w) > |T_v|$ **then**
      $r \leftarrow computeMaxLocalitySubgroup(v, w)$
      $R \leftarrow R \cup \{r\}$
    **end if**
  **end for**
  $q \leftarrow orderDescending(l_{v1}(1), .., l_{v1}(N), l_{v2}(1), ..)$
**until** q is unchanged
/* 2nd phase */
$q \leftarrow orderDescending(g_{v1}, g_{v2}, ..)$
**for** $g_v$**in**$q$ **do**
  **while** $g_v > 0$ **do**
    $w \leftarrow getWorkerWithMostCapacity()$
    $r \leftarrow computeLocalityAgnosticSubgroup(v, w)$
    $R \leftarrow R \cup \{r\}$
  **end while**
**end for**
**return** $R$

**Algorithm 3** Create Subgroup Exploiting Data Locality

**Input** attribute value $v$, worker $w$, worker capacity $cap(w)$, $|S_v|, |T_v|, s_v(n), t_v(n), l_v(n), g_v$ for each worker $n = \{1, .., N\}$
**Output** A subgroup $P_v^{lo}$ of group $P_v$ for worker $w$ with the maximum data locality
**Procedure** computeMaxLocalitySubgroup():

$ext_s \leftarrow \emptyset, ext_t \leftarrow \emptyset$
$r_t \leftarrow |T_v|, r_s \leftarrow |S_v|$
**if** $g_v > cap(w)$ **then**
  **if** $|T_v| > |S_v|$ **and** $|S_v| < cap(w)$ **then**
    $r_t \leftarrow t_v(w)$
    **if** $r_s \times r_t > cap(w)$ **then**
      $r_t \leftarrow cap(w) \div |S_v|$
    **end if**
  **else if** $|T_v| < cap(w)$ **then**
    $r_s \leftarrow s_v(w)$
    **if** $r_s \times r_t > cap(w)$ **then**
      $r_s \leftarrow cap(w) \div |T_v|$
    **end if**
  **end if**
  **if** $r_s \times r_t > cap(w)$ **then**
    **return**
  **end if**
**end if**
**if** $r_s > s_v(w)$ **then**
  $ext_s \leftarrow SamountFromOtherWorkers(v, w)$
**end if**
**if** $r_t > t_v(w)$ **then**
  $ext_t \leftarrow TamountFromOtherWorkers(v, w)$
**end if**
**update** $cap(w), |S_v|, |T_v|, s_v(w), t_v(w), l_v(w), g_v$
**return** $(v, (unique\_subgroup\_id, w, r_s, r_t, ext_s, ext_t))$

## APPENDIX B
## PROOF OF CONCEPT

In this section we will present in detail a simple example of the execution of Algorithm 2 and analyze the outcome and profit compared with the common hash-based repartitioning used for shuffling.

We create two datasets, with the characteristics presented in Table 9. As we can see in this table, dataset S presents a significant skew as 23 out of 40 records are referring to attribute value a, 12 to b and only 5 records to attribute values c and d. The initial queue is created from variables $l_a, l_b, l_c, l_d$ as:

$$q = [36, 26, 20, 6, 6, 4, 2]$$

To visualize the repartitioning procedure, which includes 6 iterations in total, we will present graphically the groups and subgroups after each iteration of the algorithm. The initial groups, with their sizes, data placement in each worker and the local weights that form the queue are presented in Fig. 18.

The first subgroup created will be related to $l_b(0) = 36$ which is the first element in the queue and will be assigned to worker 0. The subgroup will contain 6 $T_b$ records and 12 $S_b$ records, hence the whole group $P_b$ will be assigned to worker 0. The remaining capacity in worker 0 after this assignment is $95 - 6 \cdot 12 = 23$. As we can see in Fig. 19 a large part of the created subgroup will be computed with local records, but 2 $T_b$ and 3 $S_b$ records will need to be transferred from worker 1 to 0. The local weights become $l_b = [0, 0]$ and the queue is updated to the following:

$$q = [26, 20, 6, 4, 2]$$

The second subgroup created will be related to $l_a(1) = 26$ which is the first element in the queue and will be assigned to worker 1. The subgroup will contain 4 $T_a$ records and 23 $S_a$ records; hence the whole group $P_a$ will be assigned to worker 1. The remaining capacity in worker 1 after this assignment is $95 - 4 \cdot 23 = 3$. As we can see in Fig. 20 a part of the created subgroup will be computed with local records, but 2 $T_a$ and 10 $S_a$ records will need to be transferred from worker 0 to 1. The local weights become $l_a = [0, 0]$ and the queue is updated to the following:

$$q = [6, 4, 2]$$

The third subgroup created will be related to $l_c(1) = 6$ which is the first element in the queue and will be assigned

**Algorithm 4** Create Subgroup Ignoring Data Locality

**Input** attribute value $v$, worker $w$, capacity $cap(w)$, $|S_v|$, $|T_v|$, $g_v$

**Output** A subgroup $P_v^{mov}$ of group $P_v$ for worker $w$ ignoring data locality

**Procedure** computeLocalityAgnosticSubgroup():

```
ext_s ← ∅, ext_t ← ∅
r_t ← |T_v|, r_s ← |S_v|
if g_v > cap(w) then
    if |T_v| > |S_v| then
        r_t ← cap(w) ÷ |S_v|
    else
        r_s ← cap(w) ÷ |T_v|
    end if
end if
ext_s ← SamountFromOtherWorkers(v, w)
ext_t ← TamountFromOtherWorkers(v, w)
update cap(w), |S_v|, |T_v|, g_v
return (v, (unique_subgroup_id, w, r_s, r_t, ext_s, ext_t))
```

to worker 1. The subgroup will contain 1 $T_c$ and 3 $S_c$ records and the remaining capacity in worker 1 after this assignment is $3 - 1 \cdot 3 = 0$. As we can see in Fig. 21 only local records will be used in this subgroup. The local weights become $l_c = [0, 3]$ and the queue is updated to the following:

$$q = [4, 3, 2]$$

The first two elements in the queue are related to $l_d(1) = 4$ and $l_c(1) = 3$, however no subgroups can be created for them since worker 1 has no capacity left. The last element in the queue related to $l_d(0) = 2$, can however be used to create a subgroup for worker 0 which still has some capacity left (Fig. 21). The fourth subgroup created will therefore be related to $l_d(0) = 2$, and will be assigned to worker 0. The subgroup will contain 6 $T_d$ and 2 $S_d$ records, hence the whole group $P_d$ will be assigned to worker 0. The remaining capacity in worker 0 after this assignment is $23 - 6 \cdot 2 = 11$. As we can see in Fig. 22 a part of the created subgroup will be computed with local records, but 4 $T_d$ and 1 $S_d$ record will need to be transferred from worker 1 to 0. The local weights become $l_d = [0, 0]$ and the queue is updated to the following:

$$q = [3]$$

At this point we enter the second phase of the repartitioning algorithm where the remaining groups are simply greedily assigned to the remaining workers.

The fifth and final subgroup consisting of 3 $T_c$ and 3 $S_c$ records can be whole assigned to worker 0, which finally obtains a minimum capacity of $11 - 3 \cdot 3 = 2$.

The final load of the workers is 95 and 93, which means that they are perfectly balanced. With the simple hash-based repartitioning algorithm two groups would be assigned to worker 0, and the other two to worker 1. Assuming the best-case scenario, groups $P_a$ and $P_c$ would be assigned to worker 1 and $P_b$ and $P_d$ to worker 0, hence the load of the workers would be 104 and 84. In the worst-case however, groups $P_a$ and $P_b$ could be both assigned to worker 1 leading to load values of 164 and 24, which is a significant load imbalance as worker 1 would be 6.8X slower than worker 0. The reduce-side operation's execution time would be proportional to the maximum worker load, which is 95 with our repartitioning algorithm and 164 with the worst-case hash-based approach i.e. the default 'naive' approach would be 73% slower due to the presence of skew.

The records that were transferred are in total 26, 17 $S$ records and 9 $T$ records. The default hash-based approach would in the worst-case scenario transfer 36 records i.e. 38% more than our approach, and in the best-case scenario 24 records i.e. 8% less than our approach. However, in the base case scenario our approach is 9% faster. We argue that the benefits of better load balancing in the final measured execution time are much more important compared to network-related overheads, especially with the development of the last generation Ethernet and InfiniBand networks. We intend to verify this argument in the experimental evaluation section.

The differences between our approach, the hash-based repartitioning, and the original subset-replicate method using group subdivisions are illustrated through a diagram of subgroup assignments. Fig. 17a depicts the worst-case scenario for naive hash-based repartitioning, where groups $P_a$ and $P_b$ are randomly assigned to worker 1, causing a severe load imbalance.

In the second diagram, appearing in Fig. 17b, we can see that the subset-replicate methodology splits in half sets $S_a$ and $S_b$ which are significantly larger and achieves a perfectly even load balancing. However, with this method sets $T_a$ and $T_b$ are sent to both workers as we can see in Fig. 17b with the red arrows. This will increase the size of shuffle data by the amount $|T_a| + |T_b|$ which corresponds to $10 \div 60 = 0.166$ or 17% of the initial total number of records.

Our patch-based method is shown in Fig. 17c. The subgroups created with our algorithm differ significantly from the common subset-replicate repartitioning for two reasons: first, we do not always split skewed groups if the worker can handle them without being overloaded; second, we optimize data locality by assigning subgroups to workers in a specific, non-random manner, reducing data movement between workers. The order of subgroup creation and assignment is also carefully chosen to maximize data locality and minimize transfer. The dashed arrows in Fig. 17c represent movements that are avoided by our algorithm, instead using locally available data (denoted as hatched areas). Only set $S_c$ is replicated, accounting for 5% of the total records, compared to the 17% replication required by the common subset-replicate method to achieve the same load balancing. The total shuffle data size is 27% smaller than in the worst-case hash-based repartitioning, and 8% larger than in the best-case hash-based approach, yet we achieve perfectly even load balancing. Thus, our

**Algorithm 5** Patch-Based Shuffle and Join

**Input** $N, S, T$
**Output** Join result

```
1: compute(V, s, t, l, g, C_max)
2: R ← patchRepartitionAndAssign()
   /* tag each record */
3: for record r in S ∪ T do
4:    w ← getLocation(r), v ← r.A
5:    assigned ← False
6:    local ← True
7:    while not assigned do
8:       for targetSubgroup t in R.getAll(v) do
            /* first fill subgroups assigned to this worker */
            if local and t.getWorker! = w then
10:           continue /* next fill subgroups that expect data
              from this worker */
11:         else if not local and t.ext.get(w) == 0 then
12:            continue
13:         else if nextEmptyLineExists(t) then
14:            tags ← ∅
15:            for subgroup sb in t.getNextLine() do
16:               tags.add(sb.getId, sb.getWorker)
17:            end for
18:            t.fillNextLine()
19:            tagAndShuffle(r, tags)
20:            assigned ← True
21:            break
22:         end if
23:      end for
         /* finally fill random subgroups */
24:      if not assigned and not local then
25:         for targetSubgroup t in R.getAll(v) do
26:            if nextEmptyLineExists(t) then
27:               tags ← ∅
28:               for subgroup sb in t.getNextLine() do
29:                  tags.add(sb.getId, sb.getWorker)
30:               end for
31:               t.fillNextLine()
32:               tagAndShuffle(r, tags)
33:               assigned ← True
34:               break
35:            end if
36:         end for
37:      end if
38:      if not assigned and not local then
39:         local ← False
40:      end if
41:   end while
42: end for
43: joinDataForeachCommonTag()
```

methodology outperforms both the common subset-replicate and hash-based repartitioning methods.

## APPENDIX C
## ALGORITHM DEFINITIONS

See Algorithms 2–5.

## REFERENCES

[1] T. Um, G. Lee, and B.-G. Chun, "Pluto: High-performance IoT-aware stream processing," in *Proc. IEEE 41st Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2021, pp. 79–91.
[2] E. Mehmood and T. Anees, "Challenges and solutions for processing real-time big data stream: A systematic literature review," *IEEE Access*, vol. 8, pp. 119123–119143, 2020.
[3] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner, "The snowflake elastic data warehouse," in *Proc. Int. Conf. Manage. Data*, Jun. 2016, pp. 215–226.
[4] Z. Liu, L. Wan, J. Guo, F. Huang, X. Feng, L. Wang, and J. Ma, "PPRU: A privacy-preserving reputation updating scheme for cloud-assisted vehicular networks," *IEEE Trans. Veh. Technol.*, vol. 74, no. 2, pp. 1877–1892, Feb. 2025.
[5] Y. Miao, Y. Yang, X. Li, Z. Liu, H. Li, K. R. Choo, and R. H. Deng, "Efficient privacy-preserving spatial range query over outsourced encrypted data," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 3921–3933, 2023.
[6] B. Mahesh, "Machine learning algorithms—A review," *Int. J. Sci. Res. (IJSR)*, vol. 9, no. 1, pp. 381–386, Jan. 2020.
[7] D. Xin, H. Miao, A. Parameswaran, and N. Polyzotis, "Production machine learning pipelines: Empirical analysis and optimization opportunities," in *Proc. Int. Conf. Manage. Data*, Jun. 2021, pp. 2639–2652.
[8] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum, "In-situ MapReduce for log processing," in *Proc. USENIX Annu. Tech. Conf.*, 2011, pp. 115–129.
[9] J. Guo, Z. Liu, S. Tian, F. Huang, J. Li, X. Li, K. K. Igorevich, and J. Ma, "TFL-DT: A trust evaluation scheme for federated learning in digital twin for mobile networks," *IEEE J. Sel. Areas Commun.*, vol. 41, no. 11, pp. 3548–3560, Nov. 2023.
[10] Apache. (2025). *Apache Spark—Unified Engine for Large-Scale Data Analytics*. Accessed: Feb. 19, 2025. [Online]. Available: https://spark.apache.org/
[11] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009, doi: 10.14778/1687553.1687609.
[12] Presto. (2025). *Presto DB*. Accessed: Feb. 19, 2025. [Online]. Available: https://prestodb.io/
[13] M. Fuller, M. Moser, and M. Traverso, *Trino: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly Media, 2022.
[14] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in MaPreduce," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2010, pp. 975–986.
[15] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
[16] A. M. Gupta, V. Gadepally, and M. Stonebraker, "Cross-engine query execution in federated database systems," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2016, pp. 1–6.
[17] R. Li, M. Riedewald, and X. Deng, "Submodularity of distributed join computation," in *Proc. Int. Conf. Manage. Data*, May 2018, pp. 1237–1252.
[18] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann, "Flow-join: Adaptive skew handling for distributed joins over high-speed networks," in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 1194–1205.
[19] H. Lan, Z. Bao, and Y. Peng, "A survey on advancing the DBMS query optimizer: Cardinality estimation, cost model, and plan enumeration," *Data Sci. Eng.*, vol. 6, no. 1, pp. 86–101, Mar. 2021.
[20] N. Bruno, Y. Kwon, and M.-C. Wu, "Advanced join strategies for large-scale distributed computation," *Proc. VLDB Endowment*, vol. 7, no. 13, pp. 1484–1495, Aug. 2014.

[21] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, "Handling data skew in parallel joins in shared-nothing systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2008, pp. 1043–1052.

[22] Databricks. *Adaptive Execution*. Accessed: Feb. 19, 2025. [Online]. Available: https://www.databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html

[23] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "SkewTune: Mitigating skew in mapreduce applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2012, pp. 25–36.

[24] E. Kassela, "Load-centric data shuffling with a patch-based repartitioning algorithm exploiting the data placement and distribution," M.S. thesis, School Elect. Comput. Eng., Nature Tech. Univ. Athens (NTUA), Athens, Greece, 2023.

[25] Y. Chen, J. Wang, Y. Lu, Y. Han, Z. Lv, X. Min, H. Cai, W. Zhang, H. Fan, C. Li, T. Guan, W. Lin, Y. Jia, and J. Zhou, "Fangorn: Adaptive execution framework for heterogeneous workloads on shared clusters," *Proc. VLDB Endowment*, vol. 14, no. 12, pp. 2972–2985, Jul. 2021.

[26] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, and M. Taufer, "Memory-efficient and skew-tolerant MapReduce over MPI for supercomputing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 12, pp. 2734–2748, Dec. 2020.

[27] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," *Proc. VLDB Endowment*, pp. 27–40, Aug. 1992.

[28] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.

[29] T. T. Nguyen, F. Trahay, J. Domke, A. Drozd, E. Vatai, J. Liao, M. Wahib, and B. Gerofi, "Why globally re-shuffle? Revisiting data shuffling in large scale deep learning," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2022, pp. 1085–1096.

[30] S. Yu, H. Chen, and H. Jin, "Nereus: A distributed stream band join system with adaptive range partitioning," *IEEE Trans. Consum. Electron.*, vol. 69, no. 4, pp. 949–961, Nov. 2023.

[31] P. Tampakis, C. Doulkeridis, N. Pelekis, and Y. Theodoridis, "Distributed subtrajectory join on massive datasets," *ACM Trans. Spatial Algorithms Syst.*, vol. 6, no. 2, pp. 1–29, Jun. 2020.

[32] A. Zeidan and H. T. Vo, "Efficient spatial data partitioning for distributed kNN joins," *J. Big Data*, vol. 9, no. 1, pp. 1–42, Dec. 2022.

[33] F. Liang, F. C. M. Lau, H. Cui, Y. Li, B. Lin, C. Li, and X. Hu, "RelJoin: Relative-cost-based selection of distributed join methods for query plan optimization," *Inf. Sci.*, vol. 658, Feb. 2024, Art. no. 120022.

[34] J. Gao, W. Liu, Z. Li, J. Zhang, and L. Shen, "A general fragments allocation method for join query in distributed database," *Inf. Sci.*, vol. 512, pp. 1249–1263, Feb. 2020.

[35] J. Duggan, O. Papaemmanouil, L. Battle, and M. Stonebraker, "Skew-aware join optimization for array databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2015, pp. 123–135.

[36] Z. Yang, "The architecture of OceanBase relational database system," *J. East China Normal Univ.*, vol. 2014, no. 5, pp. 141–148, Sep. 2014.

[37] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, and M. Taufer, "Mimir: Memory-efficient and scalable MapReduce for large supercomputing systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2017, pp. 1098–1108.

[38] Y. Miao, F. Li, X. Li, Z. Liu, J. Ning, H. Li, K. R. Choo, and R. H. Deng, "Time-controllable keyword search scheme with efficient revocation in mobile E-health cloud," *IEEE Trans. Mobile Comput.*, vol. 23, no. 5, pp. 3650–3665, May 2024.

[39] Y. Miao, Y. Yang, X. Li, L. Wei, Z. Liu, and R. H. Deng, "Efficient privacy-preserving spatial data query in cloud computing," *IEEE Trans. Knowl. Data Eng.*, vol. 36, no. 1, pp. 122–136, Jan. 2024.

[40] C. P. Schnorr and M. Euchner, "Lattice basis reduction: Improved practical algorithms and solving subset sum problems," *Math. Program.*, vol. 66, nos. 1–3, pp. 181–199, Aug. 1994.

[41] H. I. Christensen, A. Khan, S. Pokutta, and P. Tetali, "Approximation and online algorithms for multidimensional bin packing: A survey," *Comput. Sci. Rev.*, vol. 24, pp. 63–79, May 2017.

[42] E. D. Demaine and M. L. Demaine, "Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity," *Graphs Combinatorics*, vol. 23, no. 1, pp. 195–208, Jun. 2007.

[43] A. Lodi, S. Martello, and M. Monaci, "Two-dimensional packing problems: A survey," *Eur. J. Oper. Res.*, vol. 141, no. 2, pp. 241–252, Sep. 2002.

[44] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2011.

[45] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, and B. Saha, "Apache Hadoop YARN: Yet another resource negotiator," in *Proc. 4th Annu. Symp. Cloud Comput.*, 2013, pp. 1–16.

[46] D. Jankov, B. Yuan, S. Luo, and C. Jermaine, "Distributed numerical and machine learning computations via two-phase execution of aggregated join trees," *Proc. VLDB Endowment*, vol. 14, no. 7, pp. 1228–1240, Mar. 2021.

[47] R. Tang, N. K. Aridas, and M. S. A. Talip, "Design of a data processing method for the farmland environmental monitoring based on improved spark components," *Frontiers Big Data*, vol. 6, pp. 1–9, Nov. 2023.

[48] Y. Jahnavi, Y. Pavan Kumar Reddy, V. S. K. Sindhura, V. Tiwari, and S. Srivastava, "A novel processing of scalable web log data using map reduce framework," in *Proc. Comput. Vis. Robot.*. Singapore: Springer, 2023, pp. 15–25.

[49] P. P.-S. Chen, "The entity-relationship model—Toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, no. 1, pp. 9–36, Mar. 1976, doi: 10.1145/320434.320440.

[50] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey, "Sort vs. Hash revisited: Fast join implementation on modern multi-core CPUs," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1378–1389, Aug. 2009.

[51] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," *Proc. VLDB Endowment*, vol. 7, no. 1, pp. 85–96, Sep. 2013.

[52] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu, "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware," in *Proc. IEEE 29th Int. Conf. Data Eng. (ICDE)*, Apr. 2013, pp. 362–373.

[53] Apache. (2025). *Apache Spark Configuration*. Accessed: Feb. 19, 2025. [Online]. Available: https://spark.apache.org/docs/latest/configuration.html

**EVDOKIA KASSELA** received the Master of Engineering degree in electrical and computer engineering and the Master of Science degree in data science and machine learning from the National Technical University of Athens (NTUA), in 2013 and 2023, respectively, where she is currently pursuing the Ph.D. degree with the Computing Systems Laboratory. She was a Teaching Assistant in various big-data related courses offered by the School of Electrical and Computer Engineering, NTUA. Currently, she also works as a Senior-level Data Engineer. Her research interests include big data topics and relational processing.

**IOANNIS KONSTANTINOU** received the Diploma degree in electrical and computer engineering and the M.Sc. degree in techno-economic systems from the National Technical University of Athens (NTUA), in 2004 and 2011, respectively. He is currently an Associate Professor with the Informatics and Telecommunications Department, University of Thessaly, where he regularly teaches operating systems and programming courses. He is also a Senior Researcher with the Computing Systems Laboratory, National Technical University of Athens, where he also teaches advanced topics in databases. He also serves as a member for the Board of Directors of KTP SA. His research interests include large scale distributed data management systems (cloud computing and big-data systems). He was a recipient of one Best Paper Award (IEEE CCGRID 2013) and one Best Paper Award Nomination (IEEE CCGRID 2015) for his work on large-scale distributed systems.

**NECTARIOS KOZIRIS** (Member, IEEE) is currently a Professor in computer science and the Former Dean of the School of Electrical and Computer Engineering, National Technical University of Athens. His research interests include parallel and distributed systems, interaction between compilers, OS and architectures, datacenter hyperconvergence, scalable data management, and large scale storage systems. He has co-authored more than 180 research articles with more than 6600 citations (H-index: 36). Since 1998, he has been involved in the organization of many international scientific conferences, including IPDPS, ICPP, SC, and SPAA. He has given many invited talks in conferences and universities. He was a recipient of two best paper awards for his research in parallel and distributed computing (IEEE/ACM IPDPS 2001 and CCGRID 2013) and had received honorary recognition from Intel, in 2015, for his research and insightful contributions in transactional memory (TSX synchronization extensions). He has participated as a partner or a consortium coordinator in several EU projects involving large-scale systems. He is a member of the IEEE Computer Society, a Senior Member of the ACM, and the elected Chair of the IEEE Greece Section and started the IEEE Computer Society Greece. To promote the open source software in Greece, he co-founded the Greek Free/Open Source Software Society (GFOSS-www.ellak.gr), in 2008, with members 29 Greek universities and research centers, where he is also serving as the Vice-Chair for the Board of Directors. For the last ten years (2004–2014), he has served as the Vice-Chair for the Greek Research and Technology Network-GRNET. He has been the Founder of the Okeanos Project, since 2012, a public cloud IaaS infrastructure, among the biggest ones in the European public sector (topping out beyond 10.000 active VMs), powered by the open source Synnefo software (www.synnefo.org). He is also serving as the National Delegate for European High Performance Joint Undertaking (EuroHPC JU), where he is also a member of the Governing Board. For more information visit the link (http://www.cslab.ece.ntua.gr/nkoziris).