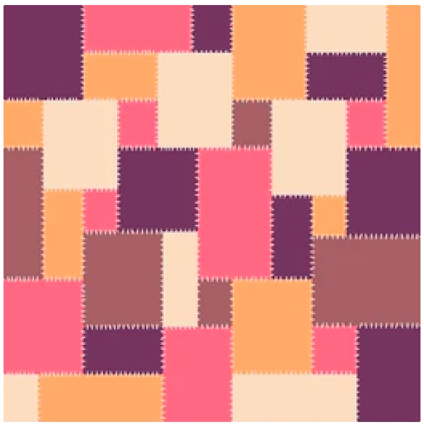
National Technical University of Athens
School of Electrical and Computer Engineering
MSc Data Science and Machine Learning

Load-centric data shuffling with a patch-based repartitioning algorithm exploiting the data placement and distribution

Diploma Thesis

of

EVDOKIA KASSELA



Supervisor: Nektarios Koziris
Professor

Athens, October 2023

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
MSC DATA SCIENCE AND MACHINE LEARNING

Load-centric data shuffling with a patch-based repartitioning algorithm exploiting the data placement and distribution

DIPLOMA THESIS

of

EVDOKIA KASSELA

Supervisor: Nektarios Koziris
Professor

Approved by the examination committee on 31st October 2023.

(Signature)	*(Signature)*	*(Signature)*
....	....	....
Nektarios Koziris	Ioannis Konstantinou	Georgios Goumas
Professor	Assistant Professor	Associate Professor

Athens, October 2023

National Technical University of Athens
School of Electrical and Computer Engineering
MSc Data Science and Machine Learning

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

(Signature)

....
Evdokia Kassela
31st October 2023

Abstract

This diploma thesis aims to the development of a novel data repartitioning algorithm that can be used to process unordered skewed data in distributed environments. In workloads involving joins and aggregations, the presence of skew in the join/group by attribute values typically causes load balancing issues in such environments, where one worker becomes a straggler. To address this problem, solutions that employ a subset-replicate partitioning methodology and rely on cost models have been applied in research, however they are based on custom-build execution engines or custom hardware. Typical general-purpose distributed processing engines which are widely used in industry, try to address the problem through dynamic task monitoring and partition resizing based on user-defined limits. Similarly, code-based user solutions such as key-salting can be used for creating more equally sized partitions. The aforementioned industry-based approaches tackle the problem of uneven load balancing based on task management and rely on user-defined thresholds, yet ignoring the network i/o related overheads that are induced by the replication of the unskewed side. Our goal is to develop an algorithm that can be easily integrated with any distributed processing system without involving cost-models or user-defined parameters and addresses both the load balancing and replication-related network overheads that arise under the presence of data skew for reduce-side operations. Our implementation uses information regarding the data distribution and placement on the workers and tries to minimize data movements by creating prioritized local-based partitions that can be locally processed with zero or minimal data movement while maintaining an even load on the workers. We evaluate our algorithm for different levels of skew in comparison with the hash-based partitioning algorithm using three different evaluation parameters; the size of data that were transferred over the network, the load on the workers, and the estimated execution time using a linear model. The experimental results confirm that the load balancing performed with our algorithm is always perfectly even, and that the increased network traffic which occurs with our algorithm due to data replication incurs a minimal overhead in the execution time in case of a low level of skew. At moderate to high skew levels the overall performance of our algorithm is superior as it is proved to be mainly affected by the worker load and less by the network i/o overheads.

Keywords

distributed execution, hash join, reduce-side, skew, partitioning, shuffling, load balancing, data placement, key-salting, subset-replicate

Περίληψη

Αυτή η διπλωματική εργασία στοχεύει στην ανάπτυξη ενός νέου αλγορίθμου ανακατανομής δεδομένων που μπορεί να χρησιμοποιηθεί για την επεξεργασία λοξών δεδομένων σε κατανεμημένα περιβάλλοντα. Σε φορτία εργασίας που περιλαμβάνουν συνενώσεις και συναθροίσεις, η παρουσία της λοξότητας στις τιμές του χαρακτηριστικού συνένωσης/ συνάθροισης προκαλεί ανισορροπία στον φόρτο εργασίας σε τέτοια περιβάλλοντα, όπου κάποιοι κόμβοι γίνονται πολύ αργοί. Για να αντιμετωπιστεί αυτό το πρόβλημα, έχουν αναπτυχθεί ερευνητικές λύσεις που χρησιμοποιούν τη μεθοδολογία ανακατανομής subset/replicate και βασίζονται σε μοντέλα κόστους. Ωστόσο, αυτές οι λύσεις βασίζονται σε εξατομικευμένες μηχανές εκτέλεσης ή εξατομικευμένο υλικό. Οι γενικού σκοπού κατανεμημένες μηχανές επεξεργασίας που χρησιμοποιούνται ευρέως στη βιομηχανία προσπαθούν να αντιμετωπίσουν το πρόβλημα μέσω δυναμικής παρακολούθησης των εργασιών και ανακατανομής των δεδομένων βάσει ορίων που ορίζει ο χρήστης. Επίσης, λύσεις που βασίζονται σε υλοποίηση του ίδιου του χρήστη, όπως το key-salting, μπορούν να χρησιμοποιηθούν για τη δημιουργία πιο ισομεγεθών τμημάτων δεδομένων προς επεξεργασία. Οι προαναφερθείσες προσεγγίσεις στον κλάδο της βιομηχανίας αντιμετωπίζουν το πρόβλημα της ανισορροπίας του φόρτου εργασίας με βάση τη διαχείριση εργασιών και βασίζονται σε όρια που καθορίζει ο χρήστης, αγνοώντας την αυξημένη δικτυακή κίνηση που προκαλείται από την ανακατανομή τμημάτων των δεδομένων. Στόχος μας είναι η ανάπτυξη ενός αλγορίθμου που μπορεί να ενσωματωθεί εύκολα σε οποιοδήποτε σύστημα κατανεμημένης επεξεργασίας χωρίς τη χρήση μοντέλων κόστους ή παραμέτρων που καθορίζονται από τον χρήστη. Ο αλγόριθμος στοχεύει στο να αντιμετωπίσει τόσο την ανισορροπία του φόρτου εργασίας όσο και την αυξημένη δικτυακή κίνηση που σχετίζεται με την ανακατανομή τμημάτων δεδομένων. Η υλοποίησή μας χρησιμοποιεί πληροφορίες σχετικά με τη κατανομή και την τοποθέτηση των δεδομένων στους κόμβους και προσπαθεί να ελαχιστοποιήσει τις μετακινήσεις των δεδομένων δημιουργώντας κατά προτεραιότητα τοπικά τμήματα δεδομένων που μπορούν να επεξεργαστούν τοπικά με μηδενική ή ελάχιστη μετακίνηση δεδομένων, διατηρώντας παράλληλα ένα ομοιόμορφο φόρτο εργασίας στους κόμβους. Αξιολογούμε τον αλγόριθμό μας για διάφορα επίπεδα λοξότητας σε σύγκριση με τον αλγόριθμο ανακατανομής βάσει κατακερματισμού (hash) χρησιμοποιώντας τρεις διαφορετικές παραμέτρους αξιολόγησης: το μέγεθος των μεταφερόμενων δεδομένων στο δίκτυο, τον φόρτο εργασίας στους κόμβους και τον εκτιμώμενο χρόνο εκτέλεσης χρησιμοποιώντας ένα γραμμικό μοντέλο. Τα πειραματικά αποτελέσματα επιβεβαιώνουν ότι ο φόρτος εργασίας που επιτυγχάνεται με τον αλγόριθμό μας είναι απόλυτα εξισορροπημένος και ότι η αύξηση της δικτυακής κίνησης που προκύπτει από τον αλγόριθμό μας λόγω της ανακατανομής των δεδομένων επηρεάζει ελάχιστα τον χρόνο εκτέλεσης σε περιπτώσεις χαμηλού επιπέδου λοξότητας. Σε περιπτώσεις μέτριας έως υψηλής λοξότητας, η συνολική απόδοση του αλγορίθμου μας είναι καλύτερη, καθώς αποδεικνύεται ότι ο χρόνος εκτέλεσης επηρεάζεται κυρίως από το φόρτο εργασίας και λιγότερο από τη δικτυακή κίνηση.

Λέξεις Κλειδιά

κατανεμημένη εκτέλεση, συνένωση, λοξότητα δεδομένων, τμηματοποίηση, ανακατανομή, φόρτος εργασίας, key-salting, subset-replicate, hash

to my parents

Acknowledgements

I would like to start by thanking professor Nectarios Koziris for supervising this thesis and for the opportunity he gave me to complete it in the Computing Systems Laboratory. I would also like to express my special thanks to Dr. Ioannis Konstantinou for his guidance and the exceptional cooperation we had. Thanks also to my fiance Kostas, for putting up with me being sat in a desk for hours on end, and for providing guidance and a sounding board when required. Finally, I would like to thank my parents for their guidance and moral support they provided me throughout these years.

Athens, October 2023

Evdokia Kassela

Table of Contents

List of Figures

List of Images

List of Tables

Preface

This work is presented as a diploma thesis for the Master of Data Science and Machine Learning offered by the National Technical University of Athens's ECE School. The development took place in Athens, Greece in the premises of the Computing Systems Laboratory of the ECE School.

Chapter 1

Introduction

1.1 Subject of the work

A typical organization/institute nowadays keeps many hundred GBs of data distributed in various datastores depending on their source, format and processing capabilities. 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, SQL, etc. In many of these cases, data residing in different datastores also need to be combined (i.e. joined in SQL semantics) in order to perform complex/multi-domain Big Data analytics.

SQL analytics is generally a very popular domain, which is commonly included in complex analysis scenarios, as it is used for joining and filtering datasets. Many research and production systems have been created in an effort to address the problem of performing such complex analytics with various data sources efficiently. The most prominent examples are Apache Spark [2] and Presto [4] which are autonomous distributed processing engines, however polystore systems such as BigDawg [11] also exist, which on the contrary exploit the sophisticated processing engines of the underlying datastores. Irrespective of owning a processing engine, the most important components of each of these systems for achieving the best performance is, however, the query planner/optimizer and the task scheduler.

In the context of SQL analytics when trying to efficiently join two or more datasets, most of the existing optimizers mainly focus on exploiting data locality profits (to minimize network traffic) by choosing the appropriate operator while the task scheduler aims to achieve a fine load balancing for the workers by distributing tasks to them. Especially when using different data sources however, the performance is greatly affected by the correct data management as a large amount of data may need to be transferred between/from different datastores inevitably.

Although a common optimizer considers these data transfer overheads before performing a join as we stated, it unfortunately lends no weight to the data skew which is a particularly important aspect in data management. For example, the hash-based repartition join algorithm which is the standard choice of optimizers when joining two large datasets, can not achieve good performance by definition when a dataset presents a high skew. In such cases, a large part of the data will be placed in a few partitions only by the underlying hash-based shuffling mechanism. On the other hand, the task scheduler which is responsible to select the worker that each data partition will be assigned, will blindly assign some large partition to a worker without considering the size of it rendering this worker overloaded.

Another aspect that is worth considering regarding the data transfer minimization is the exploitation of the data placement. In theory an optimizer is trying to schedule as many map-side operations as possible and avoid the network-related overheads induced by data shuffling operations. For example, under certain conditions an optimizer would select a broadcast join that is executed on the map-side instead of the the hash-based repartition join if less data movement is required for it. However, in some cases the hash-based repartition join would be preferred as the better option. The shuffling is therefore inevitable in some cases for performing reduce-side operations such as joins and aggregations, however the placement of data is completely ignored in this case.

Recent research works have been trying to develop new algorithms to address the problem of joining large skewed datasets over multiple nodes or engines. In all of the existing efforts the developed join algorithm is skew insensitive, meaning that it is used to perform any join irrespective of the level of skew (even with zero skew). In these cases, the optimizer simply provides existing data statistics to the join algorithm (if they are needed) for managing the data partitions, and does not participate in the join optimization process. However, the proposed algorithms lack extensive evaluation over large clusters and non-skewed datasets ([11],[14]) in order to prove their generality, come with autonomous engines ([11],[16]) or rely on specific hardware and protocols for their operation [16].

On the other hand, in release 3.0 of the Apache Spark engine the Adaptive Execution framework [1] has been introduced, which is able to handle skewed datasets by managing the size of data partitions during execution using some user-defined thresholds. However, the applied methodology is very simple as it uses no optimization rules and it 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. A similar but more generic approach was used in SkewTune, which makes on-the-fly cost-based decisions for repartitioning data of MapReduce tasks whenever a slot becomes available in the cluster using collected data statistics. It is clear that these approaches try to tackle the skew handling problem through the system itself, using continuous task monitoring and rescheduling. Although this methodology could be sufficient for load imbalances that appear in a cluster (related to skew or not), it is a system-centric approach that requires the modification of the internal operation of each engine.

In brief, solutions that rely on cost models have been applied in research using custom-build execution engines ([11],[14]) to address the problem of partitioning a skewed dataset. Also, typical distributed general-purpose engines which are widely used in industry, try to address the problem through task monitoring and dynamic rescheduling focusing on the load balancing aspect of the problem only. Moreover, code-based custom solutions such as key-salting which can be used with any engine, also deal only with the load balancing and ignore network-related overheads. Our goal is to develop an algorithm that can be used with any distributed processing system without involving cost-models and addresses both the load balancing and duplication-related network overheads that arise under the presence of data skew.

Our work aims to 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. Our algorithm can be used as a shuffling mechanism in place of the hash-based shuffling implementation which is commonly used at present. The key features of our developed 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, and the opposite.

- Locality-aware partitioning of input data: data statistics will be used to repartition the skewed parts of the dataset in a way that ensures minimum data movements.

- Priority to local processing: to ensure maximum exploitation of data locality, we split the partitioning 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.

1.2 Content organization

The rest of the document is organized as follows: in Section 2, we discuss the state-of-art partitioning techniques and present the related work. In Section 3, we present our methodology including examples and implementation details. The experimental evaluation is presented in Section 4 and we conclude our findings in Section 5.

Chapter 2

State-of-the-art partitioning techniques

In this section, we provide an overview of the subgroup partitioning strategy which is most commonly used (with slight variations) for splitting the join groups of an equi-join in the presence of skew.

2.1 Problem description

The most common algorithm which is used by distributed execution engines for joining two large datasets relies on hash-based partitioning. With this basic partitioning, the different values of the join attribute are assigned to different workers using a simple hash function, however if these values are highly skewed certain workers will inevitably present a significantly higher load. The impact of the load imbalance in such cases, is observed in the increased overall execution time.

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 partitioning. 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.

2.1.1 Skew examination

In order to identify the groups that need to be partitioned, 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. 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. Moreover, sampling techniques can be used in both cases to avoid examining the whole dataset(s). Such simple *count*-style calculations are necessary for determining skewed values and are part of the initial skew examination phase in all of the existing research works.

2.1.2 Subset-replicate partitioning

When a decision has been made to split a large group in 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 in 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 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 each 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 partitioning 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 partitioning for the remaining records to be joined, depending on the latest data statistics.

2.1.3 Optimization targets

The most important decision during the partitioning procedure is whether or not to further partition. 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 particular 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.

2.2 Related Work

BigDawg [11] is, to the best of our knowledge, the first polystore engine among many, that aims to optimize the execution of cross-engine shuffle-joins taking into account the data skew. In this polystore system, the shuffle-join framework of SciDB [10] 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 [14] a novel algorithm for executing hash-joins with large skewed datasets is presented. The aim of this work is to determine the best partitioning for heavy-hitters in order to balance the worker load as evenly as possible. A greedy algorithm that performs rectangular sub-group partitioning 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. In order 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 the aforementioned partitioning 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 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 partitioning, 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 partitioning with the lowest running time is selected. The performance of this algorithm with non-skewed datasets is not properly studied however.

A similar, but much simpler, approach is used by Apache Spark ([2],[17]) for handling data skew with the Adaptive Execution framework [1] that is included since release 3.0. With Adaptive Execution, if a partition is much larger than the median partition size and a preconfigured threshold value, it is split into smaller partitions that have the average size of non-skewed partitions or a preconfigured size. 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 actually be hurt depending on the configuration and the use case. For example, the user must carefully consider the fact that when both join sides are skewed the join could become a cartesian product. Moreover, the standard shuffle mechanism is still used to randomly assign partitions to workers and locally available data are not exploited properly in order to reduce the amount of shuffle data which is greatly increased due to the previously mentioned replication.

In a recently developed general-purpose framework [9] 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, similar with the Spark framework, and locally available data are not exploited too.

Another system-centric approach is presented with SkewTune [13]. 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 partitioning process, SkewTune collects a compressed summary of the input data.

In Flow-Join [16] 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 partitioned 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.

Finally, 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 [15] propose a partial instead of global shuffle of the data aiming to maximize local processing and minimize data movement. This approach is highly applicable to other domains too except from the deep learning domain, such as large-scale relational processing.

Chapter 3

Methodology

In this section, we formulate the problem and describe our methodology to repartition skewed data. Our approach is based on the fact that the state-of-the-art partitioning methodology which relies on cost models has been mainly applied in research using custom-build execution engines. On the other hand, typical distributed general-purpose engines which are widely used try to address the problem through task monitoring and dynamic rescheduling focusing on the load balancing aspect of the problem. Moreover, code-based custom solutions such as key-salting which can be used with any engine, also deal only with the load balancing and ignore input duplication overheads. Our goal is to develop an algorithm that can be easily implemented in any distributed processing system without involving cost-models and addresses both the load balancing and duplication-related network issues that arise under the presence of data skew.

3.1 Approach overview

The proper partitioning of data groups is of crucial importance to effectively improve the performance of a highly skewed reduce-side operation. To visualize the impact of skew in such cases, we present in Image 3.1 an example where two relations must be combined, tables A and B, and the different colors represent data that belong to a different group. Each group represents a part of the data that are related to a specific join/aggregation attribute value and must be processed independently. If some value appears with higher frequency (0 in Image 3.1) the corresponding group will contain more data in relation with others (yellow) and our data will be skewed. Skew is usually present in one side of the joined/aggregated relations (Table A in Image 3.1), but in general both relations could present skew. Assuming that each group is usually assigned to a different worker for processing it, i.e. partitions A0 and B0 of Image 3.1 will be sent to the same worker, this worker will obviously receive more load compared with others.

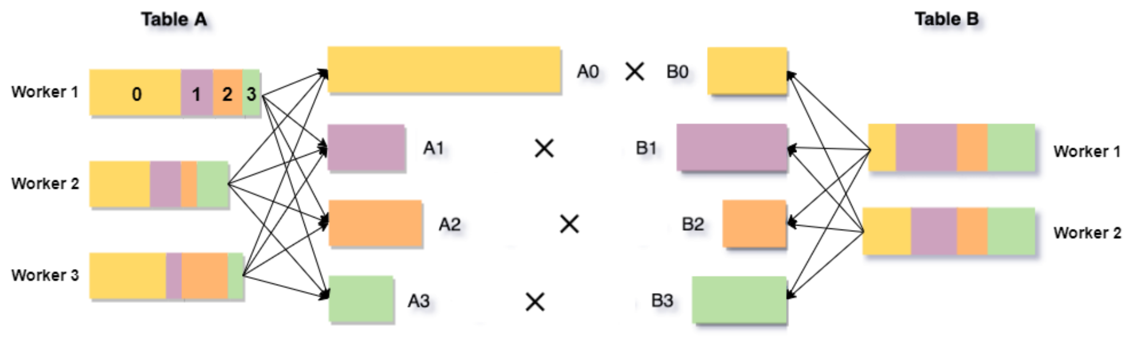


Image 3.1: *Effect of skew presence in a single key (yellow)*

The state-of-the-art subset-replicate methodology which is used to unload workers in such cases, is two 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 Image 3.2. Each of this subsets will then be sent to a different worker in order to achieve a more even load balancing. However, each of these workers will need a copy of partition B0 in order 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.

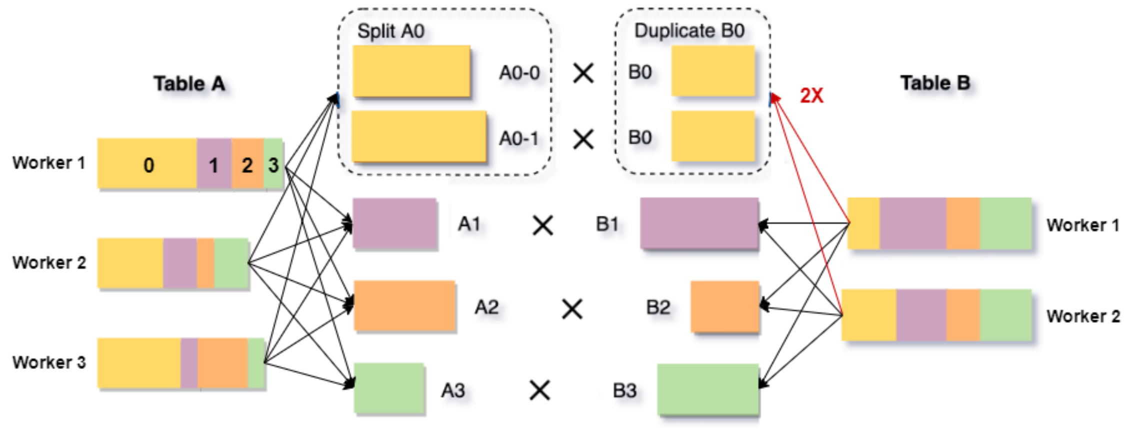


Image 3.2: *Subset-replicate methodology to handle skewed data*

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. Following the flow, we choose to focus on the load balancing and network traffic aspects too, however we introduce a new approach which is solely based on the assignment of subgroups to workers without the need for developing any cost models. We argue that through the guided initial creation and assignment of subgroups to workers one can tackle both issues:

3.1.1 Load balancing

To achieve an even load balancing, each subgroup must be assigned to a worker taking into account 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 3.4) 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 3.3.

3.1.2 Network transfer

Instead of attempting to minimize the data transfer that is caused by every splitting/-partitioning decision, we use an equivalent approach in which we try to ensure maximum data locality when creating and assigning subgroups. In particular, when we examine how a specific group should be partitioned 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 can not 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 we present the final developed algorithm for executing a highly skewed reduce-side operation.

3.2 Local and global skew examination

As in all previous works, before we start partitioning the groups we must first compute their sizes with a skew examination phase. Each group corresponds to a specific attribute 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 partitioning 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 similar to the group size but using only its local data. The notation of the variables that we will use is presented in detail in Table 3.1.

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 3.2, 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:

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

Table 3.1. *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	$P_a = \{p \in S_a \cup T_a\}$
$s_a(n), t_a(n)$	$s_a(n)$: 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 3.2. *Number of records in each set and the computed local weights per worker for group P_a.*

Worker	s_a size	t_a size	l_a weight
0	10	20	200
1	41	10	410
2	0	0	0
3	0	0	0

The total number of S_a records across all workers is $|S_a| = \sum_{n=1}^{N} s_a(n) = 51$. Respectively $|T_a| = \sum_{n=1}^{N} t_a(n) = 30$. $|S_a|$ and $|T_a|$ are considered as the group dimensions and the group size is computed as $g_a = |S_a| \times |T_a| = 1530$.

In essence, 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 partitioning algorithm to ensure an even load balancing.

3.3 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:

$$L = \sum_{v \in V} g_v$$

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

$$C_{max} = L/N + 1$$

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 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.

3.4 Two phase partitioning algorithm

Before we start partitioning 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 mainly 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 partitioning algorithm is presented in Algorithm 1 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 the optimal load balancing.

Algorithm 1: patchPartitionAndAssign

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

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.

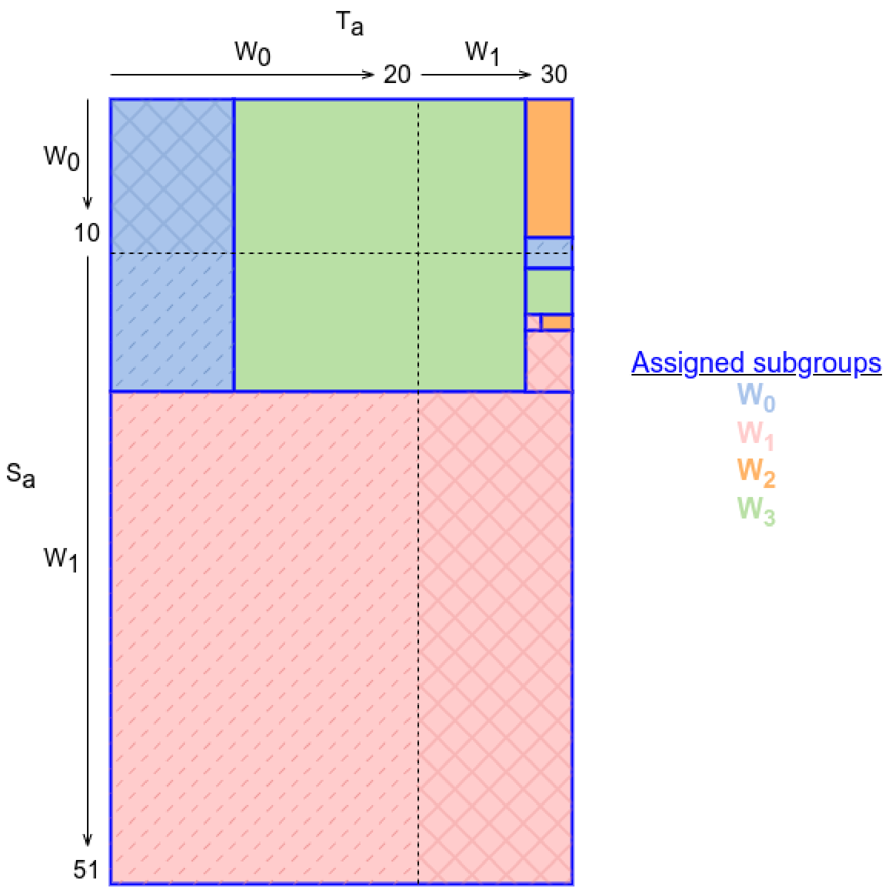


Image 3.3: *Patch-based partitioning of the join group P_a. Each subgroup is denoted as a blue rectangle and is colored differently depending on the worker it is assigned to. The cross-hatched areas can eventually be computed using only local records of the worker, while the dashed areas use partially local records.*

3.5 Patch-based group partitioning

Continuing from our previous example in subsection 3.2, we can visualize group P_a and the subgroups it is broken into in Figure 3.3. The size of each side of the whole 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 different 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 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 shown as the largest pink area in Fig. 3.3.

To explain this subgroup selection, first we try to assign the whole group to the worker. If its size exceeds the worker's remaining capacity as in this case ($g_a = 1530 > 973$), we split the largest set between S_a and T_a which is S_a. In this case, we attempt to use only the local S_a records and all of the T_a records 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 aforementioned procedure is depicted in Algorithm 2 and is used during the first phase of our algorithm.

```
Algorithm 2: computeMaxLocalitySubgroup
  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 of group $P_v$ for worker $w$ with
          the maximum data locality
  $ext_s \leftarrow \emptyset, ext_t \leftarrow \emptyset$
  /* consider whole group initially */
  $r_t = |T_v|, r_s = |S_v|$
  if $g_v > cap(w)$ then
      /* split largest of $S_v$ and $T_v$ */
      if $|T_v| > |S_v|$ and $|S_v| < cap(w)$ then
          $r_t = t_v(w)$
          if $r_s \times r_t > cap(w)$ then
              $r_t = cap(w) \div |S_v|$
      else if $|T_v| < cap(w)$ then
          $r_s = s_v(w)$
          if $r_s \times r_t > cap(w)$ then
              $r_s = cap(w) \div |T_v|$
      if $r_s \times r_t > cap(w)$ then
          return ()
  if $r_s > s_v(w)$ then
      $ext_s \leftarrow$fromOtherWorkers($r_s - s_v(w)$)
      $r_s = s_v(w)$
  if $r_t > t_v(w)$ then
      $ext_t \leftarrow$fromOtherWorkers($r_t - t_v(w)$)
      $r_t = t_v(w)$
  update $cap(w), |S_v|, |T_v|, s_v(w), t_v(w), l_v(w), g_v$
  return $(w, (v, r_s, ext_s, r_t, ext_t))$
```

With the predescribed approach it is clear that we do not split the largest set exactly in half or use predefined dimensions like previous works which consecutively divide in half the set or create subsets that do not exceed a specific size limit (Figure 3.4). Instead we aggressively try to exploit the data locality keeping as many records locally as possible. This is the reason why we call our partitioning 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.

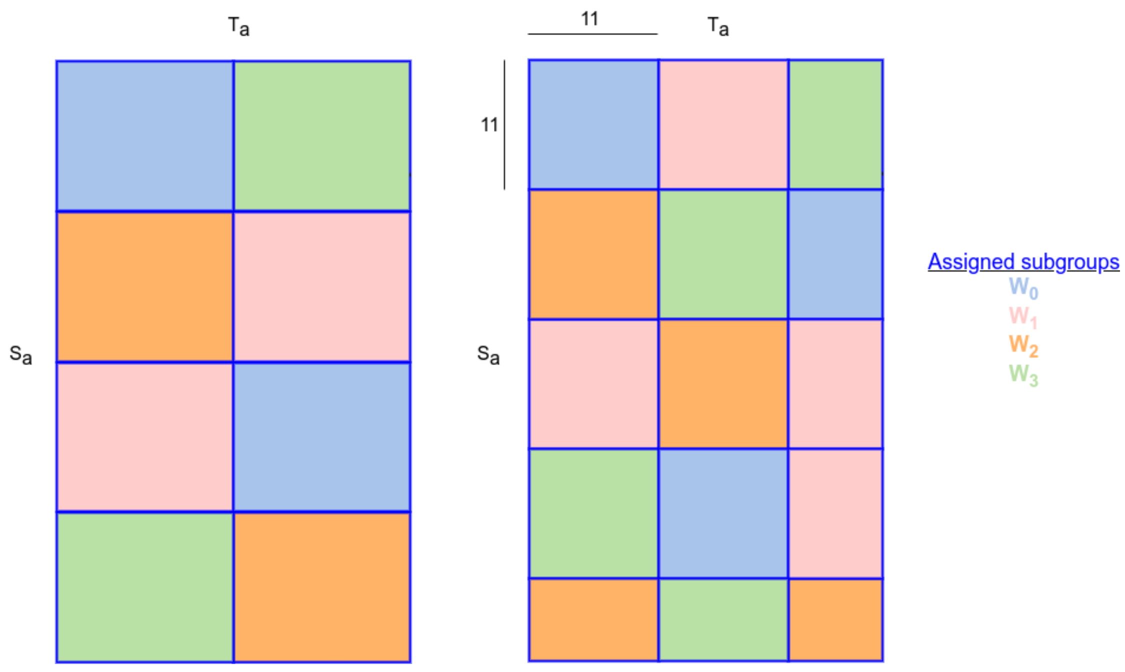


Image 3.4: *Partitioning based on consecutive divisions (left) and size limit (right) ignoring the data placement*

Regarding the data locality in the created subgroup, all the 32 S_a records can be found locally on worker 1 (see table 3.2), 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 2). 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 in order 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 they will not be needed again by other workers. 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 2, new subgroups are examined for creation depending on the updated local weights and the remanining 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. The rest of the subgroups appearing in Figure 3.3 for group P_a will therefore be created at future iterations, and not necessarily successively as other groups are also being partitioned in between.

During the second phase of our algorithm a slightly different procedure is used for creating subgroups, which is presented in Algorithm 3. The difference from Algorithm 2 is that the local weights are no longer used, and the algorithm will simply create the largest possible subgroup for the worker.

```
Algorithm 3: computeLocalityAgnosticSubgroup
  Input: attribute value $v$, worker $w$, capacity $cap(w)$,
         $|S_v|, |T_v|, g_v$
  Output: A subgroup of group $P_v$ for worker $w$
          ignoring data locality
  $ext_s \leftarrow \emptyset, ext_t \leftarrow \emptyset$
  $r_t = |T_v|, r_s = |S_v|$
  if $g_v > cap(w)$ then
      if $|T_v| > |S_v|$ then
          $r_t = cap(w) \div |S_v|$
      else
          $r_s = cap(w) \div |T_v|$
  $ext_s \leftarrow$fromOtherWorkers($r_s$)
  $ext_t \leftarrow$fromOtherWorkers($r_t$)
  update $cap(w), |S_v|, |T_v|, g_v$
  return $(w, (v, r_s, ext_s, r_t, ext_t))$
```

The complexity of Algorithms 2 and 3 is O(1). In the worst case scenario, Algorithm 1 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 1 in such case is L, where L is the total load defined in section 3.3, and is equal to the total number of output tuples that will be computed. Algorithm 1 has a worst case complexity of O(L).

3.6 Reduce-side processing

After the subgroups and their assignments are computed using Algorithm 1, the join or any other reduce-side operation can be executed. In order to perform the required operation, the records must first placed into the subgroups by tagging them with the corresponding subgroup ids. During this tagging procedure the necessary records are also transferred to specific workers and then the actual 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 particular attribute value (noted in their assigned subgroups) the worker also sends the record with the appropriate tag(s) to the required workers. A record is not necessarily placed in all the relative subgroups that are assigned to the worker. The subgroup size is considered for this purpose along with some more recorded metrics which are constantly updated during the tagging procedure. Tagging each record with multiple tags generally 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 tagging procedure for group P_a that was presented in the previous subsection, is presented in Image 3.5. The black allows represent records that will only be tagged and kept locally, and the red allows represent records that will be tagged and sent to other workers. In case of record s_5, it will be sent to worker 3, but it will be included in only in one of the two subgroups assigned to this worker. On the other hand, record s_30 will be included in both subgroups of worker 3. The record will not be emitted twice in this case, instead two tags will be sent together with this record.

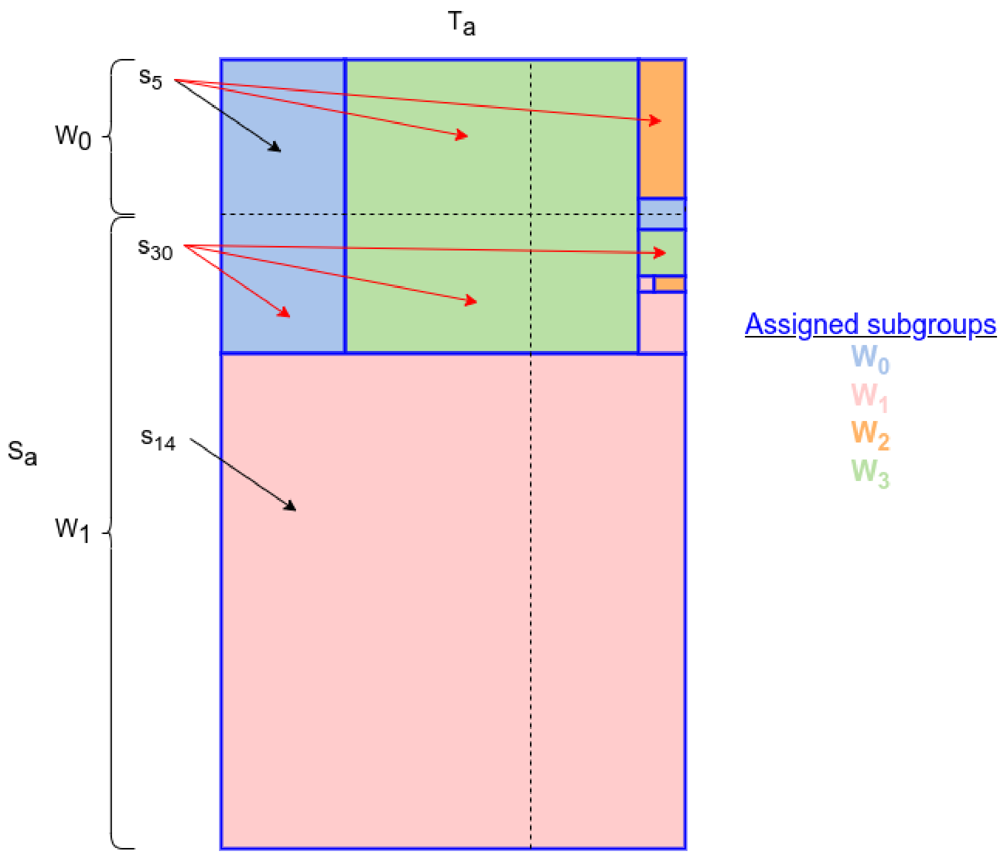


Image 3.5: *Records placement in the subgroups created for group P_a.*

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.

The selective tagging procedure that we previously described is not yet fully implemented. In particular, the algorithm that performs the tagging before the data is transferred on the reduce-side is currently being designed. The selection of the subgroups that each record will be placed is a variation of the subset sum problem [6] which is an NP-hard problem. For the subset sum problem there exist pseudo-polynomial time dynamic programming solutions and polynomial time approximation solutions which will be studied for the final implementation. Another approach that could be used is based on the two-dimensional bin packing problem [3] or the rectangle packing problem [5], which are also NP-hard.

In the next chapter we will evaluate the resulting worker load and amount of data that must be transferred to each worker based on the subgroups assignments that our patch-based partitioning algorithm produces for various skewed datasets. We will also compare the results with the corresponding load and data transfer expected when using the common hash-based shuffling procedure.

3.7 Proof of concept

In this section we will present in detail a simple example of the execution of Algorithm 1 and analyze the outcome and profit compared with the common hash-based partitioning used for shuffling.

We create two datasets, with the following characteristics:

Table 3.3. *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

Table S presents 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 partitioning 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 Image 3.6.

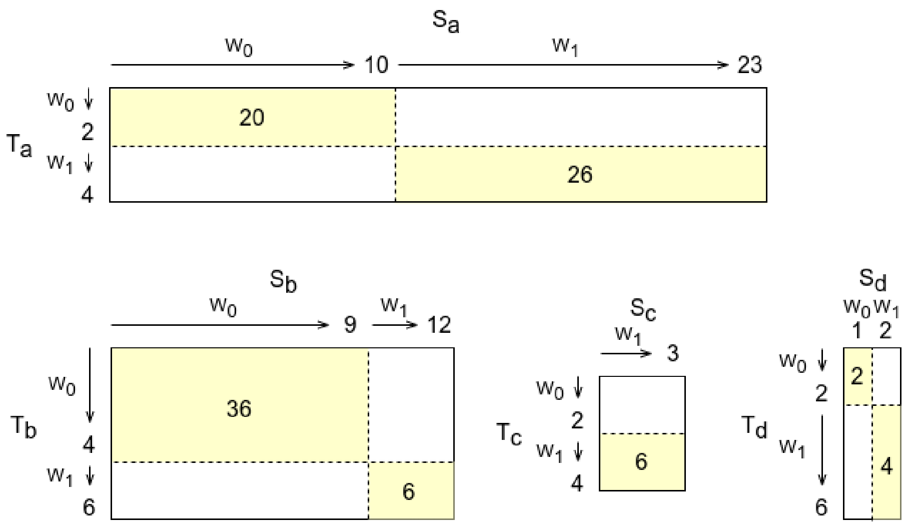


Image 3.6: *Representation of the initial four groups and their sizes, as well as the data placement in the two workers*

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 Image 3.7 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:

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

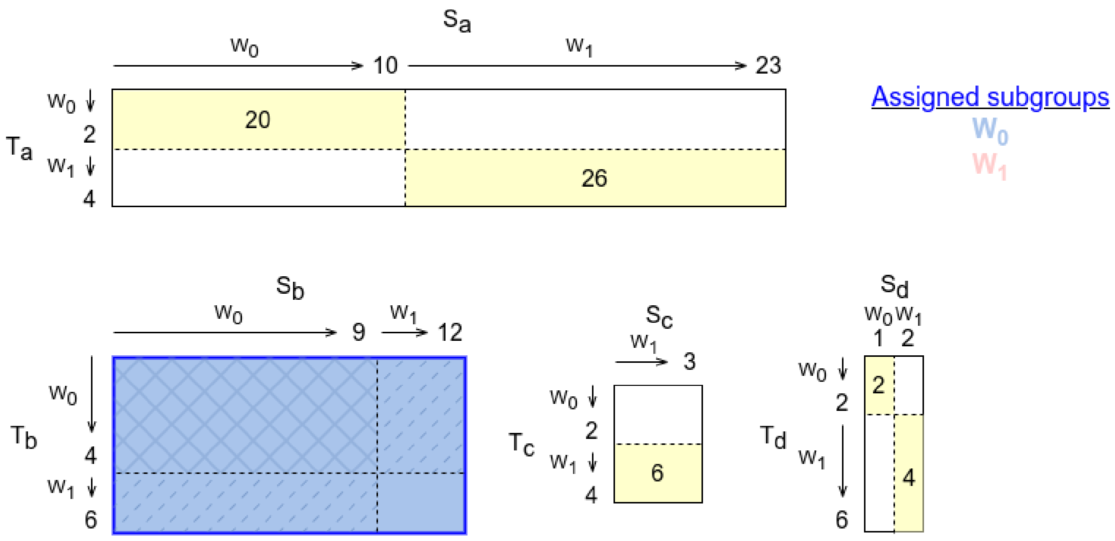


Image 3.7: *The first subgroup created for group P_b, assigned to worker 0*

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 Image 3.8 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:

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

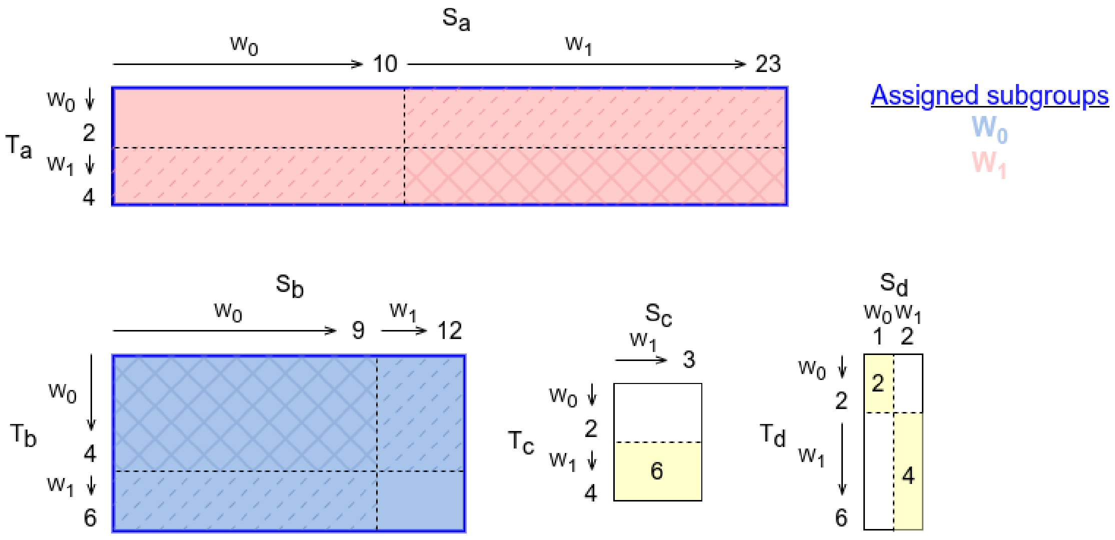


Image 3.8: *The second subgroup created for group P_a, assigned to worker 1*

The third subgroup created will be related to $l_c(1) = 6$ which is the first element in the queue, and will be assigned 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 Image 3.9 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:

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

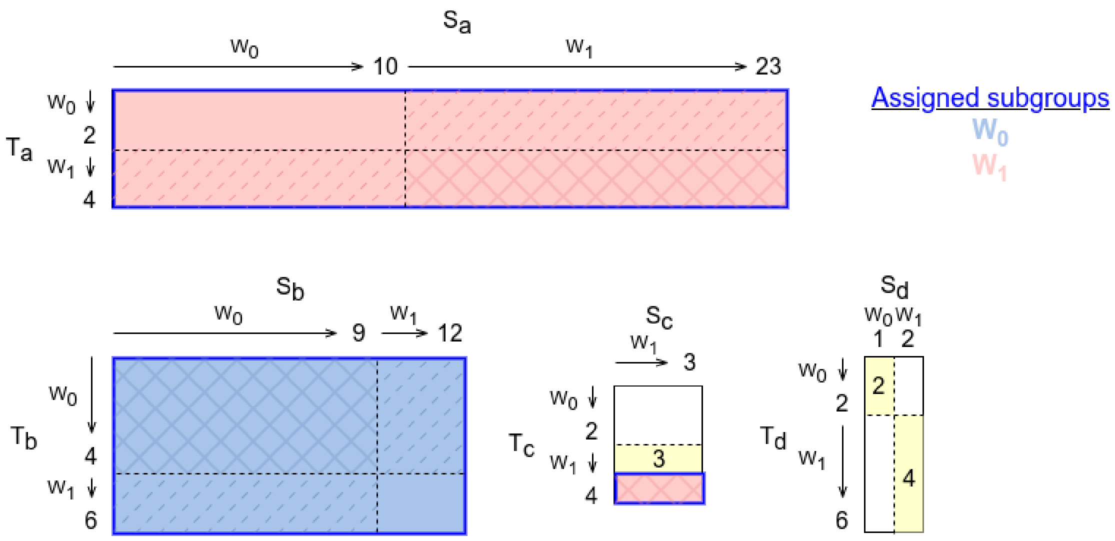


Image 3.9: *The third subgroup created for group P_c, assigned to worker 1*

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 (Image 3.10).

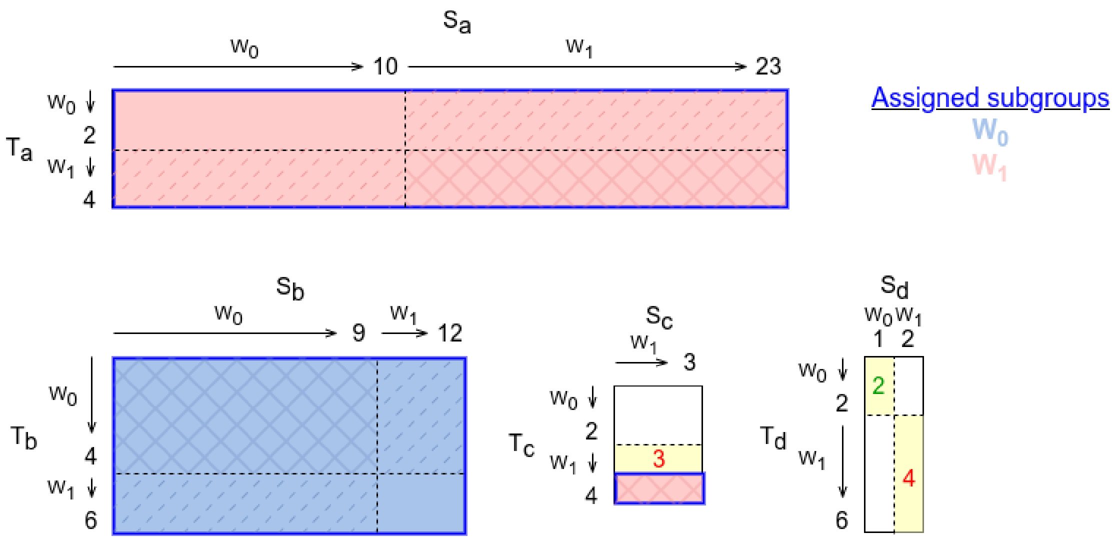


Image 3.10: *The areas noted with '4' and '3' cannot be assigned to worker 1 to exploit the data locality, therefore the area noted with '2' will be next studied for worker 0*

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 Image 3.11 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:

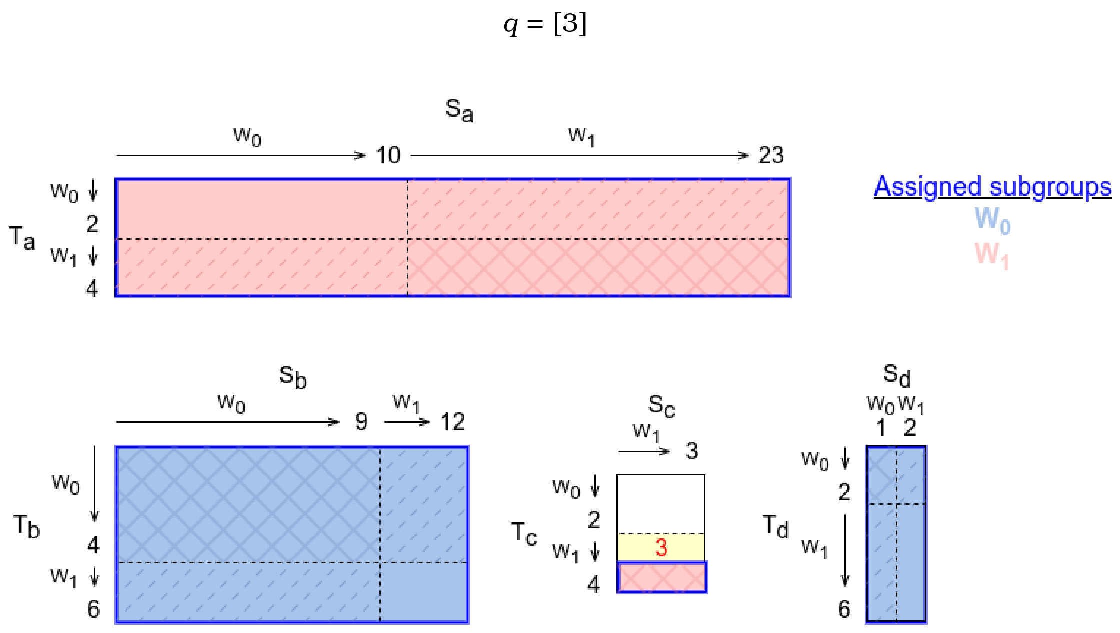


Image 3.11: *The fourth subgroup created for group P_d, assigned to worker 0. The first phase is complete as no more locality-based subgroups can be created.*

At this point we enter the second phase of the partitioning 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 $11 - 3 \cdot 3 = 2$.

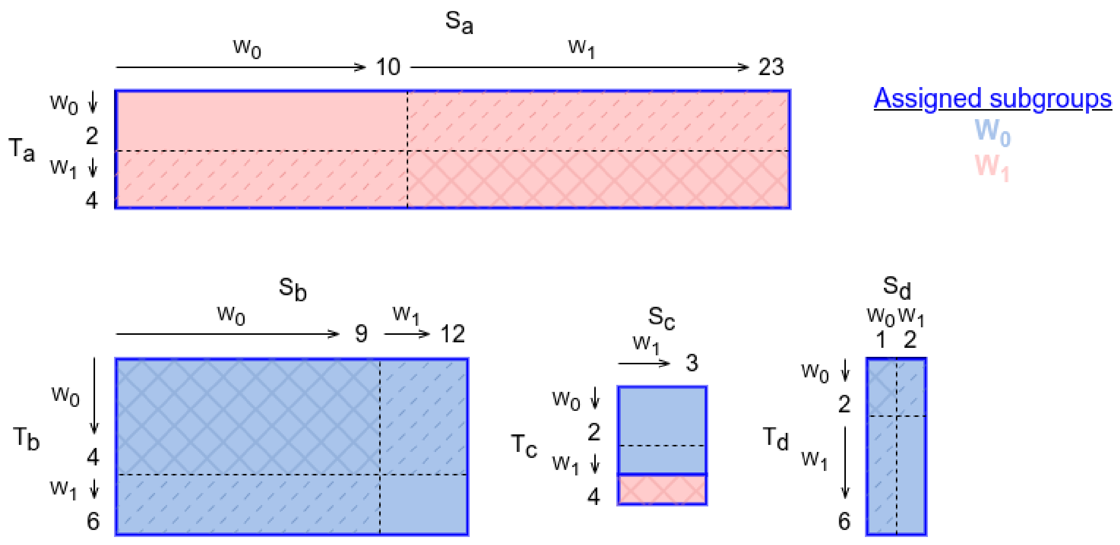


Image 3.12: *The fifth and last subgroup is created for group P_c without considering data locality, and is assigned to worker 0.*

The final load of the workers is 95 and 93, which means that they are practically perfectly balanced. With the simple hash-based partitioning 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 partitioning 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.

To demonstrate the differences between our approach in comparison with the hash-based partitioning as well as the original subset-replicate method that uses subdivisions of the initial group size, we constructed a diagram presenting the subgroup assignments in each case. In the first diagram in Image 3.13 we present the worst case naive hash-based partitioning in which groups P_a and P_b are randomly assigned to worker 1, leading to a serious load imbalance.

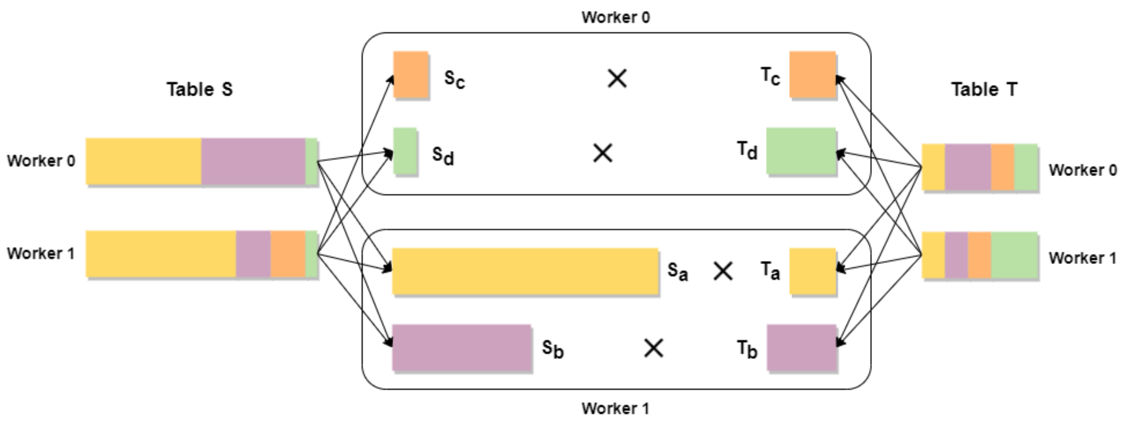


Image 3.13: *Groups assignments with the hash-based partitioning (worst case)*

In the second diagram, appearing in Image 3.14, 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 Image 3.14 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.

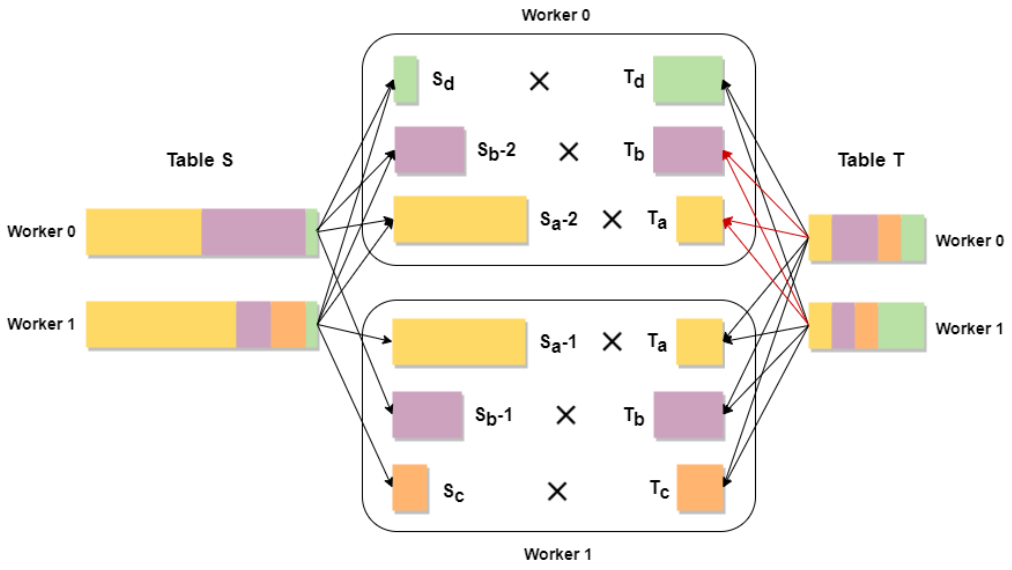


Image 3.14: *Subset-replicate based partitioning of subgroups*

Finally, our patch-based method is presented similarly in Image 3.15. As we can see, the subgroups created with our algorithm are completely different from the common subset-replicate partitioning for two reasons; 1. Our algorithm does not necessarily split skewed groups if the examined worker has enough capacity to receive it without being overloaded. 2. Our algorithm tries to exploit data locality by performing specific assignments to workers (instead of random) and this leads to less data movement between workers. Even the order that the subgroups are created and assigned is selected specifically to maximize data locality and minimize data transfer. In particular, the dashed arrows in Image 3.15 are movements that will not be performed with our algorithm and instead locally available data will be used (denoted as cross-hatched areas). Only set S_c will be replicated in this case which corresponds to only 5% of the initial total number of records, while the common subset replicate methodology replicated 17% as we previously mentioned and achieved the same optimal load balancing. Also, the total size of shuffle data is 27% less than the worst case hash-based partitioning, and 8% more than the best case hash-based as we previously mentioned however we achieve a perfectly even load balancing. Therefore, our methodology seems to present better results than both the common subset-replicate and hash-based partitioning approaches.

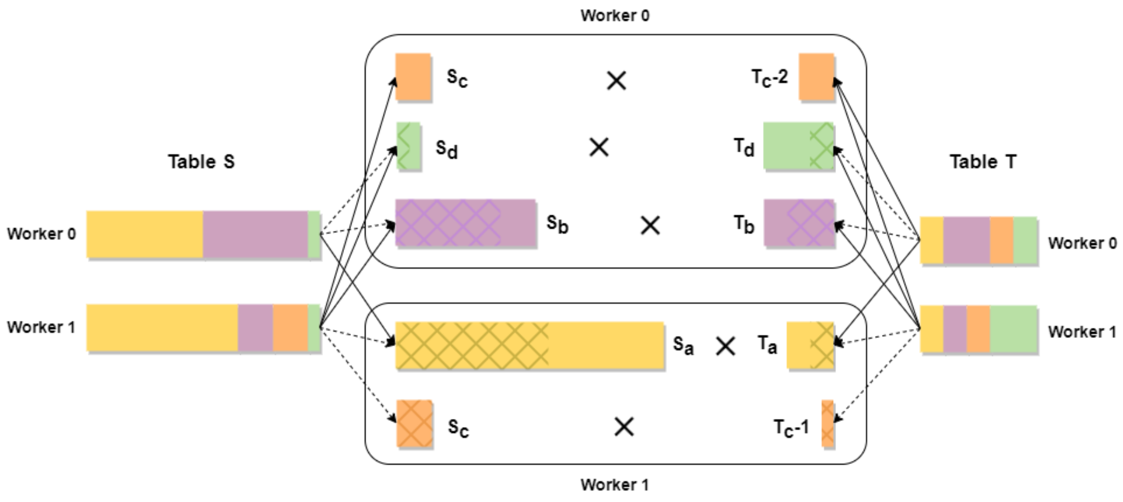


Image 3.15: *Patch-based partitioning of subgroups exploiting data locality*

3.8 Implementation

3.8.1 Patch-based partitioning algoritm

We choose to implement Algorithm 1 and all underlying methods using Java. As we focus our efforts in optimizing the execution on distributed environments, we use multiple workers which are implemented as separate threads and each worker initially loads in his memory a part of the input data. Various thread-safe memory access mechanisms are used in our implementation. Considering the initial random placement of data on the workers of distributed systems, we focus on performing as much local processing of the data as possible and at the same time avoid expensive data movements. This approach is fully compatible with the logic of our algorithm that tries to schedule as much *local* subgroup assignments as possible, considering the initial location of the in-memory data instead of the physical data location.

The code is still under development and many future optimizations are scheduled for implementation, therefore it is not yet publicly available.

3.8.2 Data generation

Two datasets with a predefined number of records are generated on-the-fly in the beginning each experiment. The number of distinct values for the attribute (or key) used to combine the datasets is initially defined, and each dataset is generated using a Zipfian distribution for this attribute values. The distribution exponent (∂ parameter) is predefined and different for each dataset as it controls the level of skew that will appear in the keys of each dataset. The Java code that performs the aforementioned procedure is included in Appendix A.

When the generation is complete, each dataset's records are uniformly divided to N equal sized parts, where N is the defined number of workers. Each worker will later use his in-memory part of the data. This approach is similar with big data concepts where a distributed filesystem is used and the workers can access in their local disk an equally sized part of the physical data which will be loaded in memory for further processing. The Java code that performs the aforementioned procedure is included in Appendix B.

3.8.3 Statistics for the data skew

Similar to previous works that require the group sizes to be previously known/computed, we first need to compute similar statistics using a multi-threaded task. First we perform some partial aggregation on the workers and compute the frequency of the join attribute values on each worker. We assume for the workers, that each one contains multiple different keys and there is no sorting of the join key. Then a global aggregation is performed to compute the local weights, by summing up the results per worker. Finally, a computation of the groups dimensions and sizes is performed based on the local weights. The Java code that performs the aforementioned procedure is included in Appendix C.

Chapter 4

Experimental evaluation

A virtual machine running Ubuntu 20.04 with 40VCPUs and a RAM of size 200GB is used for the experiments. For each experiment the following parameters are initially defined: the S and T table sizes, the number of unique keys, the number of workers and the distribution exponent ∂ for each table which we will refer to as skew factor. Each experiment includes two consecutive executions on the same initially generated datasets: the patch-based partitioning algorithm and a random case of the hash-based shuffling method. Each worker is implemented as a single thread, as we have previously mentioned.

To visualize the change of the skew factor in a dataset, assuming that we have 32 unique keys and 200000 records, we present in Figure 4.1 the ratio of records that correspond to each key. Each line represents a dataset with a different skew factor in the range $\partial : [0, \ldots 3]$.

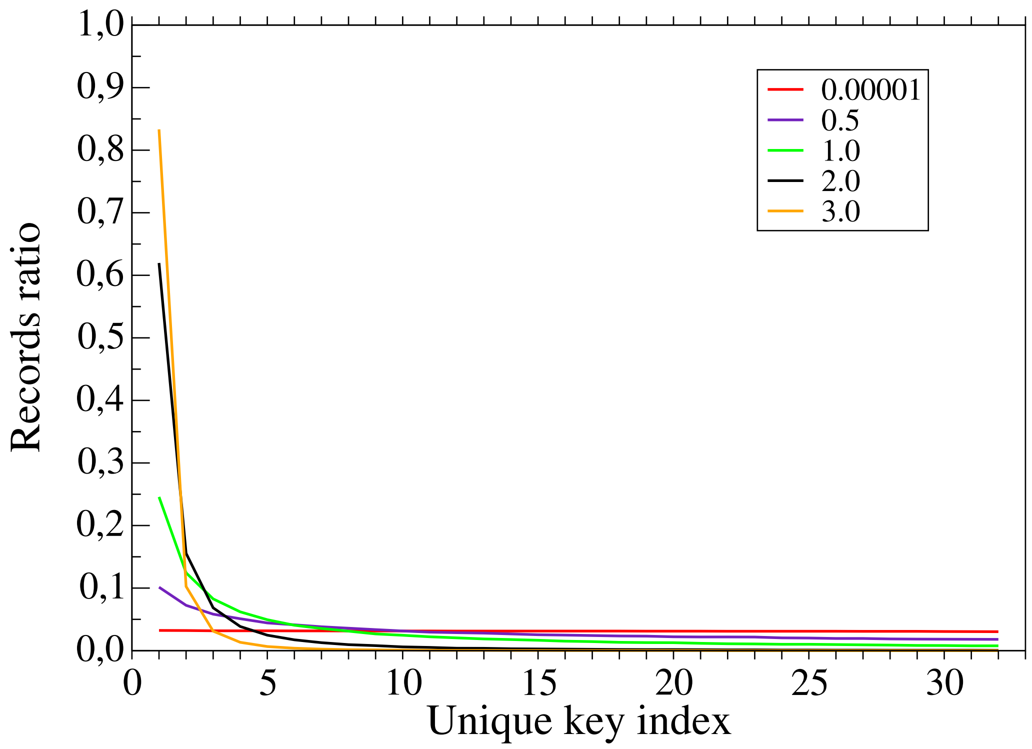


Figure 4.1. *Records distribution in the keys for different skew factors*

In Figure 4.1, the red line corresponds to zero skew (uniform distribution), the purple line which corresponds to skew factor $\partial = 0.5$ has 10% of the records in one key, the green line which corresponds to skew factor 1.0 has 25% of the records in one key, while the orange line which corresponds to skew factor 3.0 has 83% of the records in one key. In real datasets we usually find skew factors less that 1.0, however we will present some of the results for skew factors greater than 1.0 to exhibit the behavior of our algorithm under extreme cases.

Three sets of experiments are used for evaluating the performance of our algorithm. The first set of experiments examines the load balancing that occurs with our repartitioning algorithm compared with the 'naive' hash-based repartitioning using varying skew factors for the datasets. The second set focuses on 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 hash-based technique. For this set we use varying skew factor in both datasets and the number of unique keys and workers. The third set of experiments is used to evaluate the overall performance of our implementation compared to the simple hash-based partitioning. 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.

4.1 Load balancing

In the first set of experiments we will compare the minimum and maximum load that appeared among the workers, using our implementation and the 'naive' hash-based repartitioning. In order to compare our implementation with the hash-based technique, we choose a number of workers which is equal to the number of unique keys. This setup will allow us to better present the differences using a simple self-explained base case in which the hash-based technique would simply assign one whole group to each worker.

Assuming that table T has zero skew in the keys ($\partial_T \approxeq 0$), we vary the skew factor in table S using values in the range $\partial_S : [0, \ldots 3]$. Both tables contain multiple records for each attribute value in this case, using a different distribution however, resembling a *foreign-foreign* key join scenario. The number of workers and the parameters used for creating the datasets are presented in Table 4.1.

Table 4.1. *Parameters used for the experiments in Figures 4.2 and 4.3*

N	32
$\lvert V\rvert$	32
$\lvert S\rvert$	200000
$\lvert T\rvert$	200000
∂_T	0.00001

We create two diagrams with the results. The first diagram (Figure 4.2) presents the maximum expected worker load as the factor in table S increases for the two algorithms. The load is normalized using the average worker load value.

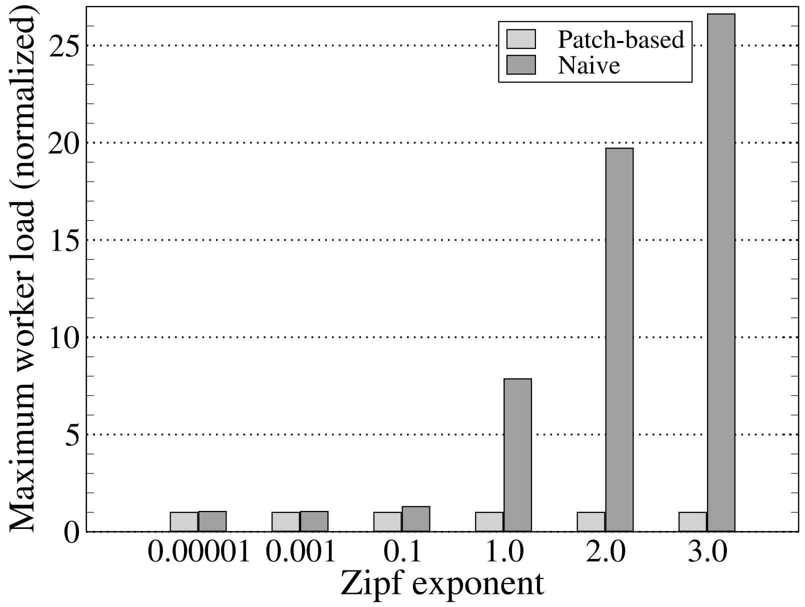


Figure 4.2. *Maximum worker load with the patch-based and naive hash-based algorithms for different skew factors in table S (∂_S)*

As we can observe in Figure 4.2, our patch-based implementation presents a maximum worker load equal to the average value in all skew factors, therefore performing perfect load balancing under all circumstances. On the contrary, the hash-based technique is of course affected by the skew in table S and as the skew factor increases to values greater than 0.1 the maximum worker load increases significantly up to 26 times more than the average value.

The second diagram (Figure 4.3) presents similarly the normalized minimum expected worker load as the factor in table S increases for the two algorithms.

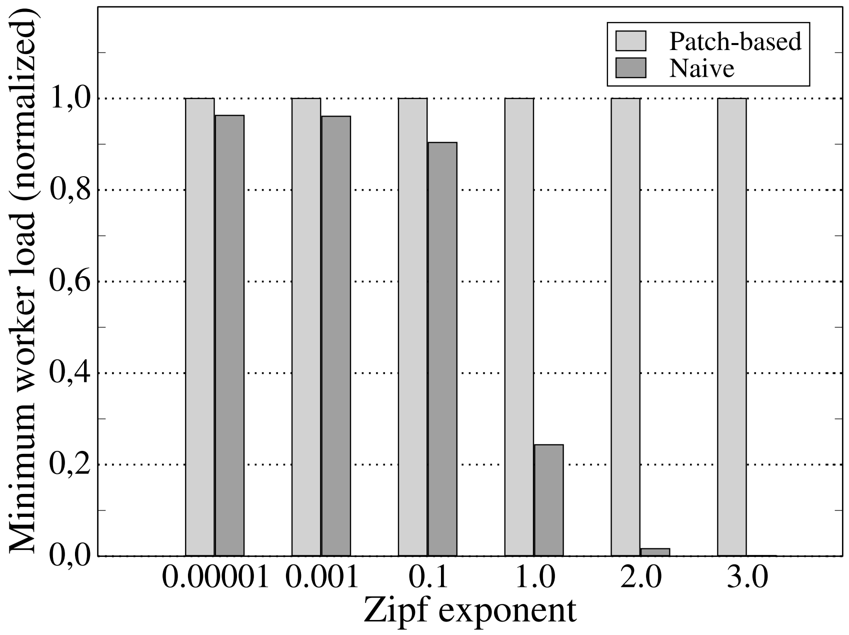


Figure 4.3. *Minimum worker load with the patch-based and naive hash-based algorithms for different skew factors in table S (∂_S)*

The results from Figure 4.3 are in agreement with Figure 4.2, and present the perfect load balancing performed by our algorithm and how the hash-based technique is again affected by the skew leading to zero minimum load in some worker(s) as the skew factor increases.

Next, assuming that both tables S and T present the same amount of skew, we vary the skew factor using values in the range $\partial_S : [0, \ldots 1]$. Both tables contain multiple records for each attribute value, resembling a *foreign-foreign* key join scenario. The number of workers and the parameters used for creating the datasets are presented in Table 4.2.

Table 4.2. *Parameters used for the experiments in Figures 4.4 and 4.5*

N	32
$\|V\|$	32
$\|S\|$	200000
$\|T\|$	200000

Like previously we create two diagrams, one presenting the normalized maximum worker load (Figure 4.4) and one presenting the minimum worker load (Figure 4.5) for the two algorithms as the skew factor increases.

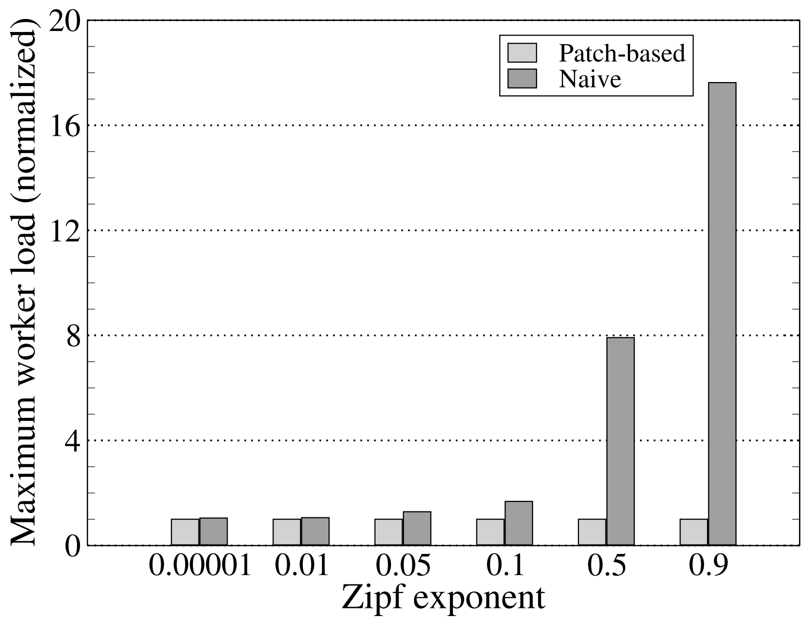


Figure 4.4. *Maximum worker load with the patch-based and naive hash-based algorithms for various skew factors applied in both tables S and T*

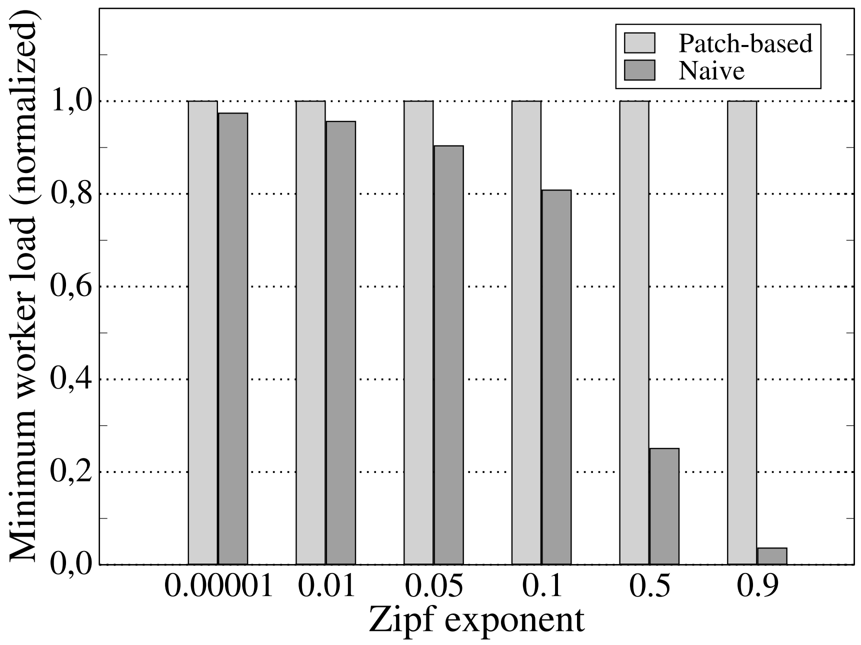


Figure 4.5. *Minimum worker load with the patch-based and naive hash-based algorithms for various skew factors applied in both tables S and T*

The inductions are similar with Figures 4.2 and 4.3. In this case we observe for the naive hash-based technique that the maximum load increases to much larger values and the minimum drops to even lower values due to the presence of skew in both tables.

Finally, assuming that table T has zero skew in the keys ($\partial_T \approxeq 0$), we vary the skew factor in table S using values in the range $\partial_S : [0, \ldots 3]$. In this case table T contains a unique appearance of each key in its records, resembling a *primary-foreign* key join scenario. The number of workers and the parameters used for creating the datasets are presented in Table 4.3.

Table 4.3. *Parameters used for the experiments in Figures 4.6 and 4.7*

N	32
$\lvert V \rvert$	32
$\lvert S \rvert$	200000
$\lvert T \rvert$	32
∂_T	0.00001

Like previously we create two diagrams, one presenting the normalized maximum worker load (Figure 4.6) and one presenting the minimum worker load (Figure 4.7) for the two algorithms as the skew factor increases. The inductions are identical with the ones for Figures 4.2 and 4.3.

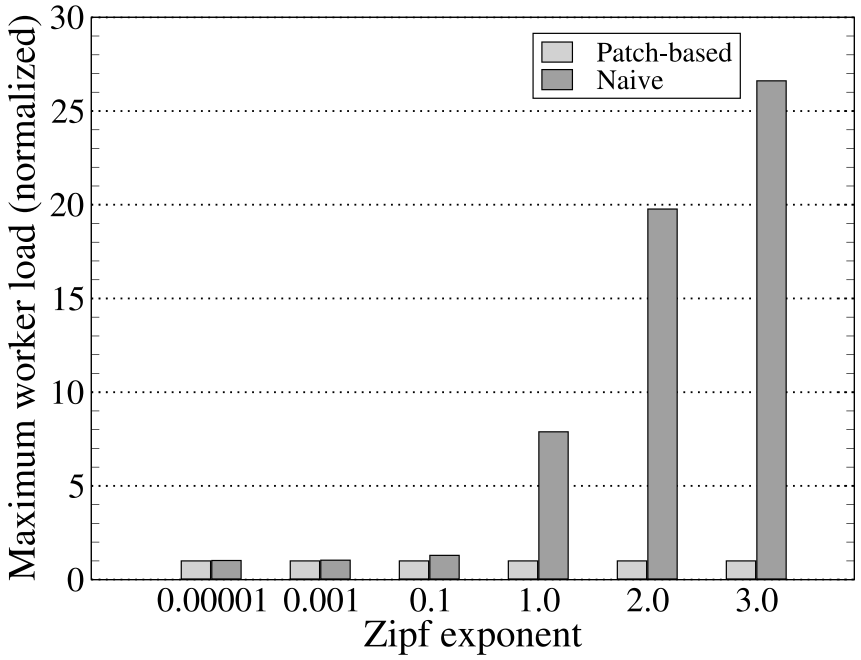


Figure 4.6. *Maximum worker load with the patch-based and naive hash-based algorithms for different skew factors in table S (ϑ_S) having unique keys in table T*

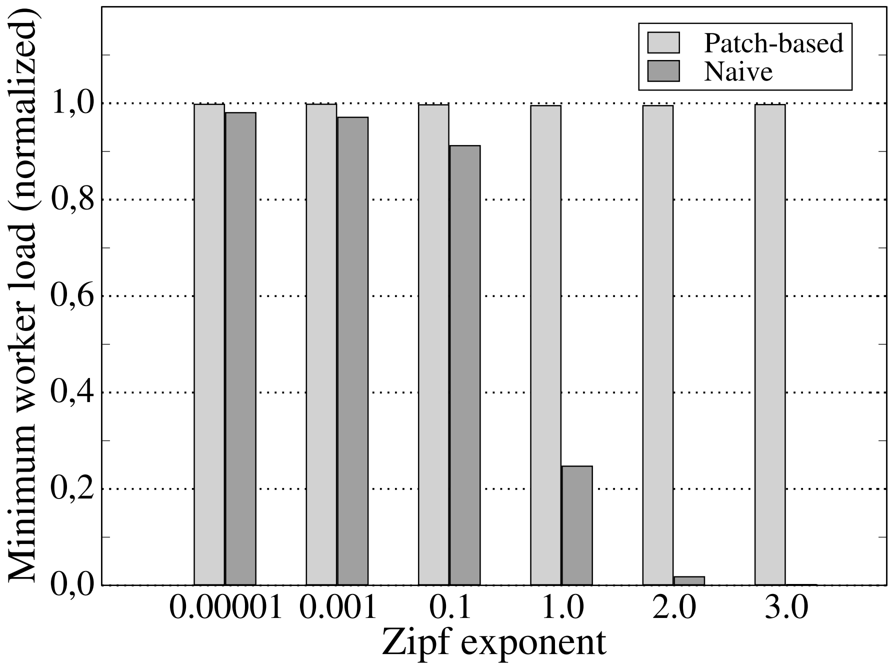


Figure 4.7. *Minimum worker load with the patch-based and naive hash-based algorithms for different skew factors in table S (ϑ_S) having unique keys in table T*

4.2 Data movement

The second set of experiments includes two different evaluation targets. Initially, we will compare the total number of records transferred over the network using our patch-based partitioning algorithm and the 'naive' hash-based partitioning. Then we will study in a more large scale setup the impact that the number of workers and the number of unique keys have in the produced shuffle data size.

4.2.1 Comparison with hash-based shuffle

In order to compare our implementation with the hash-based technique, we choose a number of workers which is equal to the number of unique keys. This setup will allow us to better present the differences using a simple self-explained base case in which the hash-based technique would simply assign one whole group to each worker.

Assuming that table T has zero skew in the keys ($\partial_T \approxeq 0$), we vary the skew factor in table S using values in the range $\partial_S : [0, \ldots 3]$. Both tables contain multiple records for each attribute value in this case, using a different distribution however, resembling a *foreign-foreign* key join scenario. The number of workers and the parameters used for creating the datasets are presented in Table 4.4.

Table 4.4. *Parameters used for the experiments in Figures 4.8 and 4.9*

N	32
$\|V\|$	32
$\|S\|$	200000
$\|T\|$	200000
∂_T	0.00001

We create two diagrams with the results. The first diagram (Figure 4.8) presents the total number of records that need to be transferred as the skew factor in table S increases for the two algorithms. The number of records is normalized using the initial number of records which is $|S| + |T|$.

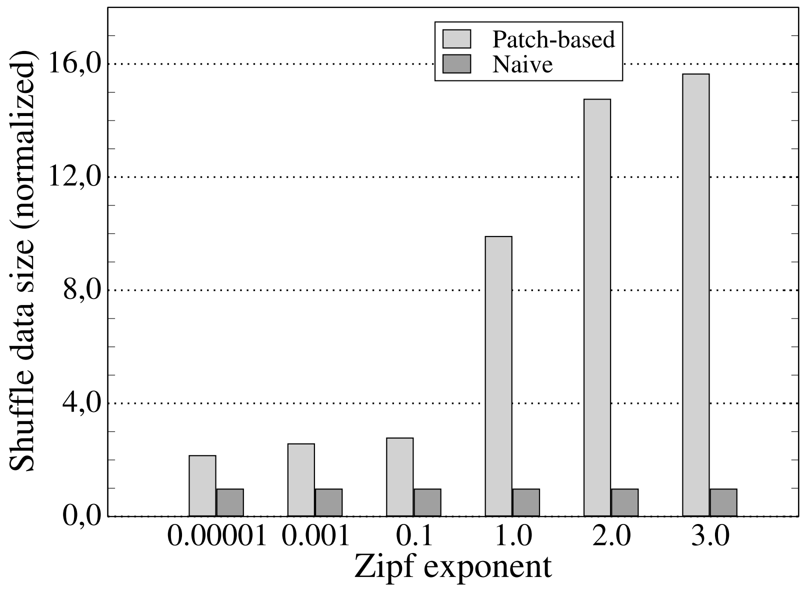


Figure 4.8. *Produced shuffle data size of the patch-based and naive hash-based algorithms for different skew factors in table S (∂_S)*

In the case of the naive hash-based algorithm, the total number of records is expected to be less or equal to the initial number of records as each record is sent exactly once to the relative worker. With our algorithm, the total number of records is higher, as expected, due to the replication that happens. As the skew increases, we can notice that the shuffle data size increases significantly, and it is 10 times larger in the case of $\partial_S = 1.0$ and almost 16 times larger in case of $\partial_S = 3.0$.

The second diagram graphically presents the statistical dispersion of the received number of records in each worker using the Gini index. A Gini 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 Figure 4.9 we present the Gini index value as the skew factor ∂_S increases for the two algorithms. The results show that initially, when there is little skew in the data, the naive algorithm has better distribution of the records to workers and our patch-based algorithm performs worse having constantly a Gini value of 0.25. However, as the skew factor increases to values greater than 0.1 we can see an opposite result as the Gini index constantly drops for our algorithm while the naive technique is seriously affected from the skew and reaches a value of 0.49 for skew factor 3.0. It is expected that our algorithm will not present the optimal behavior in terms of data exchange, 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 can not be avoided.

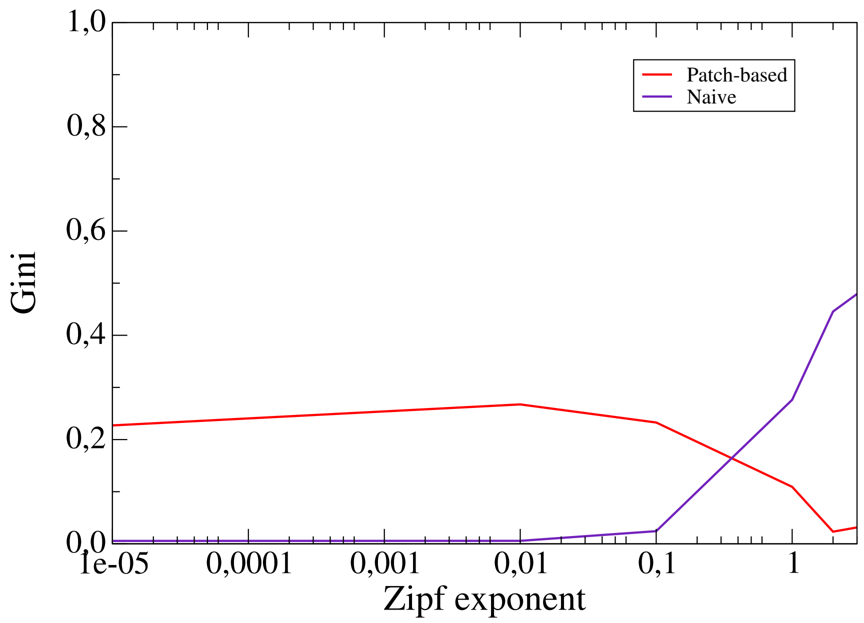


Figure 4.9. *Gini index for the shuffle data size sent to each worker with the patch-based and naive hash-based algorithms for different skew factors in table S (∂_S)*

In more detail, at large skew factors our algorithm tends to sequentially create subgroups for the largest group which are assigned to all the workers in turn leading to almost equally sized subgroups being distributed to the workers. The naive technique is on the contrary better with no to little skew as it simply distributes the almost equally sized groups to the workers. 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.

Combining the results from Figures 4.8 and 4.9 an important observation is derived. Although with our algorithm at large skew factors the total size of shuffle data is increasing up to 15.6X 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 in one worker with out algorithm, therefore the observed impact of the increased size will be significantly less visible. This observation will be further established with the results of section 4.3.

Next, assuming that both tables S and T present the same amount of skew, we vary the skew factor using values in the range $\partial_S : [0, \ldots 1]$. Both tables contain multiple records for each attribute value, resembling a *foreign-foreign* key join scenario. The number of workers and the parameters used for creating the datasets are presented in Table 4.5.

Table 4.5. *Parameters used for the experiments in Figures 4.10 and 4.11*

N	32
$\|V\|$	32
$\|S\|$	200000
$\|T\|$	200000

Like previously we create two diagrams, one presenting the normalized total number of records that need to be transferred as the skew factor increases (Figure 4.10) and one with the Gini index (Figure 4.11) for both algorithms.

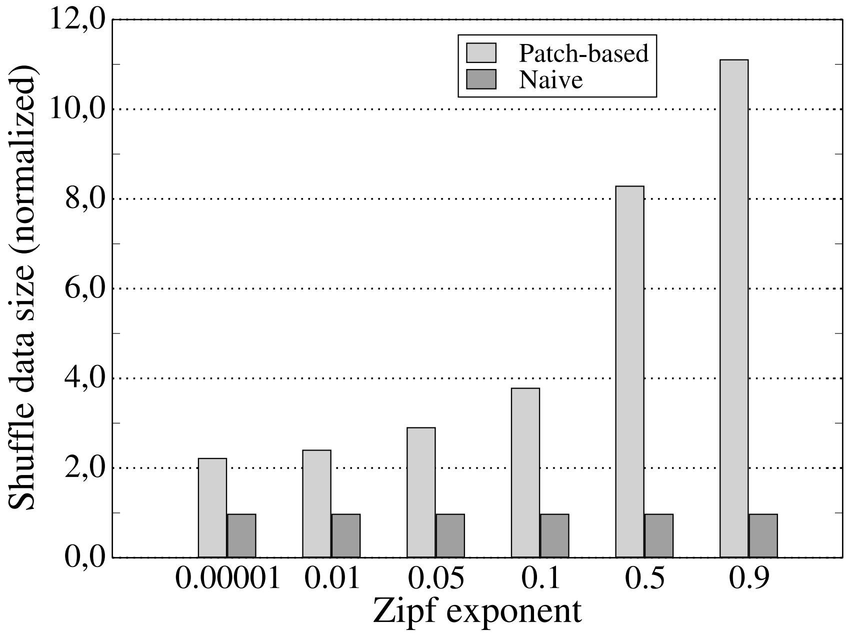


Figure 4.10. *Produced shuffle data size of the patch-based and naive hash-based algorithms for various skew factors applied in both tables S and T*

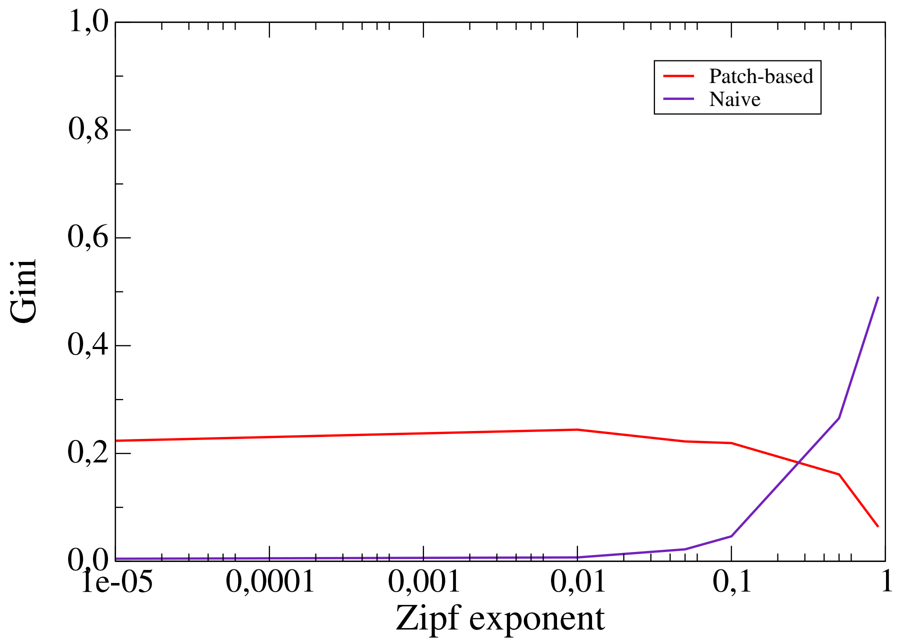


Figure 4.11. *Gini index for the shuffle data size sent to each worker with the patch-based and naive hash-based algorithms for various skew factors applied in both tables S and T*

The inductions are similar with those for Figures 4.8 and 4.9. Comparing with Figure 4.8, in Figure 4.10 there is a very small increase in the total shuffle data size in this case that the skew is present in both tables. Regarding the Gini index, the naive algorithm presents an even higher value as the skew factor increases, i.e. greater imbalance, which is normal. The observed results for the performance of our patch-based algorithm do not change with the appearance of skew in both tables.

Finally, assuming that table T has zero skew in the keys ($\partial_T \approxeq 0$), we vary the skew factor in table S using values in the range $\partial_S : [0, \ldots 3]$. In this case table T contains a unique appearance of each key in its records, resembling a *primary-foreign* key join scenario. The number of workers and the parameters used for creating the datasets are presented in Table 4.6.

Table 4.6. *Parameters used for the experiments in Figures 4.12 and 4.13*

N	32
$\|V\|$	32
$\|S\|$	200000
$\|T\|$	32
∂_T	0.00001

Like previously we create two diagrams, one presenting the normalized total number of records that need to be transferred as the skew factor increases (Figure 4.12) and one with the Gini index (Figure 4.13) for both algorithms.

We can observe in both Figures a significant differentiation from the previous cases. In Figure 4.12 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 1.0. 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.

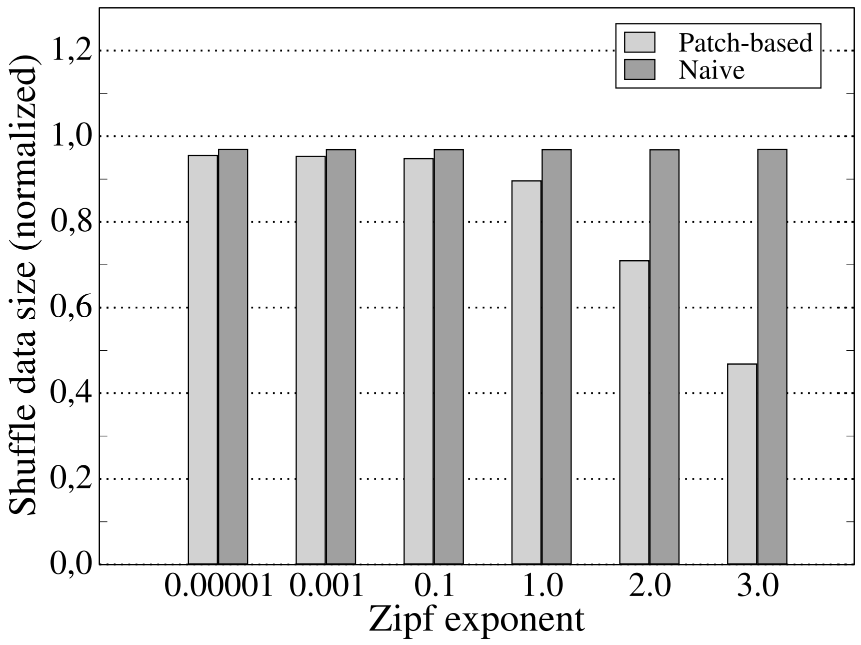


Figure 4.12. *Produced shuffle data size of the patch-based and naive hash-based algorithms for different skew factors in table S (∂_S) having unique keys in table T*

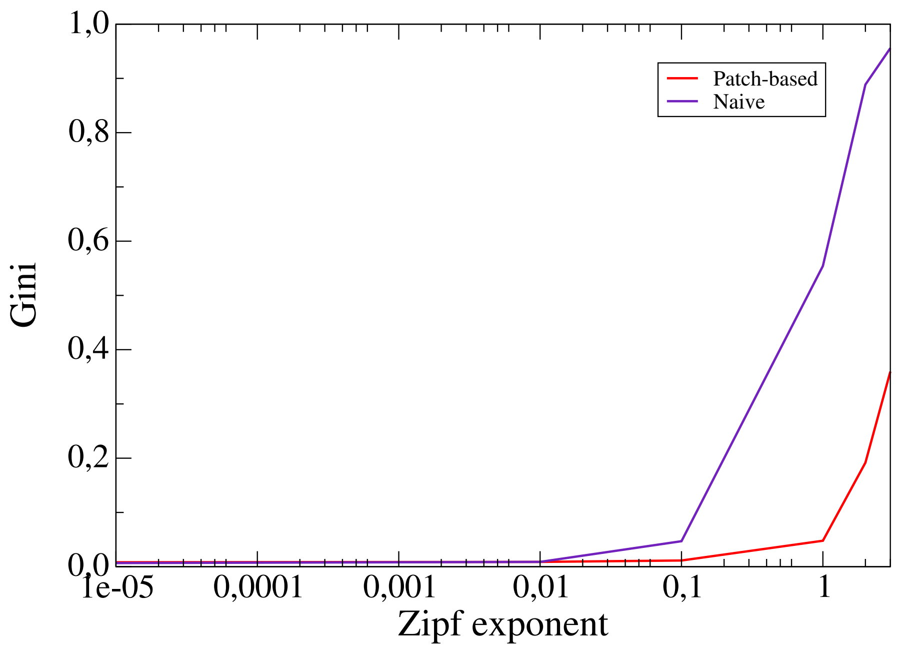


Figure 4.13. *Gini index for the shuffle data size sent to each worker with the patch-based and naive hash-based algorithms for different skew factors in table S (∂_S) having unique keys in table T*

In Figure 4.13 we also confirm that with zero to little skew our algorithm is similar with the naive solution in terms of the data transferred to each worker, which are almost equal in size. As the skew factor increases to values larger than 0.1 the naive solution is greatly affected by the skew in table S, as expected, since almost 85% of the data will be placed in one worker. Our algorithm, on the contrary, presents lower Gini index values due to the better load balancing that it tries to perform. However, compared to Figures 4.9 and 4.11 we can see that in this case we have larger values of the Gini index for increased values of the skew factor. 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. The total size of local data exploited in the first phase is around 30% based on the distribution of 4.1 and the number of workers. The local S data of the single largest group in each worker are therefore fully exploited only by one worker, and randomly a few more may be exploited in the second phase.

4.2.2 Impact of number of keys and workers

In this set of experiments we will evaluate how the total shuffle data size is affected in setups that resemble real large scale environments. We will study two different parameters, the number of distinct keys and the number of workers. The initial data size is not evaluated for large scale setups too, as it is expected that the shuffle data size will increase as the initial data size increases.

Assuming that table T has zero skew in the keys ($\partial_T \cong 0$), we vary the number of distinct keys using values in the range $|V| : [100, 10000]$. We perform the same experiment for different skew factors in table S using values in the range $\partial_S : [0, \ldots 1]$. Both tables contain multiple records for each attribute value in this case, using a different distribution however, resembling a *foreign-foreign* key join scenario. The number of workers and the parameters used for creating the datasets are presented in Table 4.7.

Table 4.7. *Parameters used for the experiments in Figure 4.14*

N	32
$\lvert S\rvert$	1000000
$\lvert T\rvert$	150000
∂_T	0.00001

Figure 4.14 presents the normalized total number of records that need to be transferred as the number of distinct keys increases for different skew factors in table S.

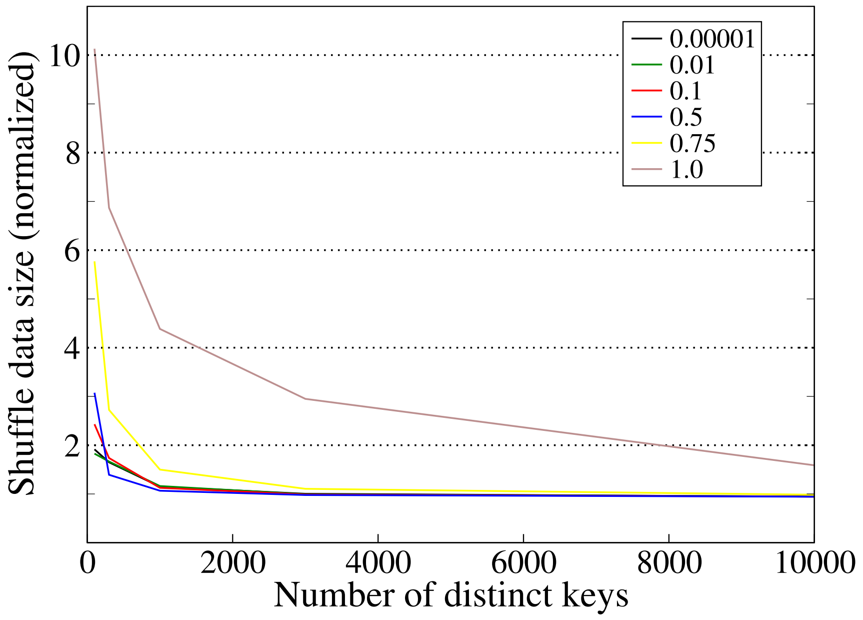


Figure 4.14. *Produced shuffle data size of the patch-based algorithm as the number of distinct keys $|V|$ increases for different skew factors in tables S (∂_S)*

In Figure 4.14, we can observe that independent of the skew factor there is an initial decrease in the normalized number of records that are sent over the network as the number of distinct keys increases. This is expected as with more distinct keys the group sizes become smaller therefore less subgroups will be created per group and less data replication will take place. The decrease seems to be exponential as the number of distinct keys increases. The shuffle data size is initially ($|V| = 100$) bigger for higher skew factor values, and the higher the skew factor is the higher seems to be the rate parameter (λ) of the exponential distribution.

Next, we perform the same set of experiments by setting the number of distinct keys to a stable value and varying the number of workers using values in the range $N : [16, 256]$. We repeat each experiment using different skew factors in table S with values in the range $\partial_S : [0, \ldots 1]$. The number of distinct keys and the rest of the parameters used for creating the datasets are presented in Table 4.8.

Table 4.8. *Parameters used for the experiments in Figure 4.15*

$\|V\|$	3000
$\|S\|$	1000000
$\|T\|$	150000
∂_T	0.00001

Figure 4.15 presents the normalized total number of records that need to be transferred as the number of workers increases for different skew factors in table S.

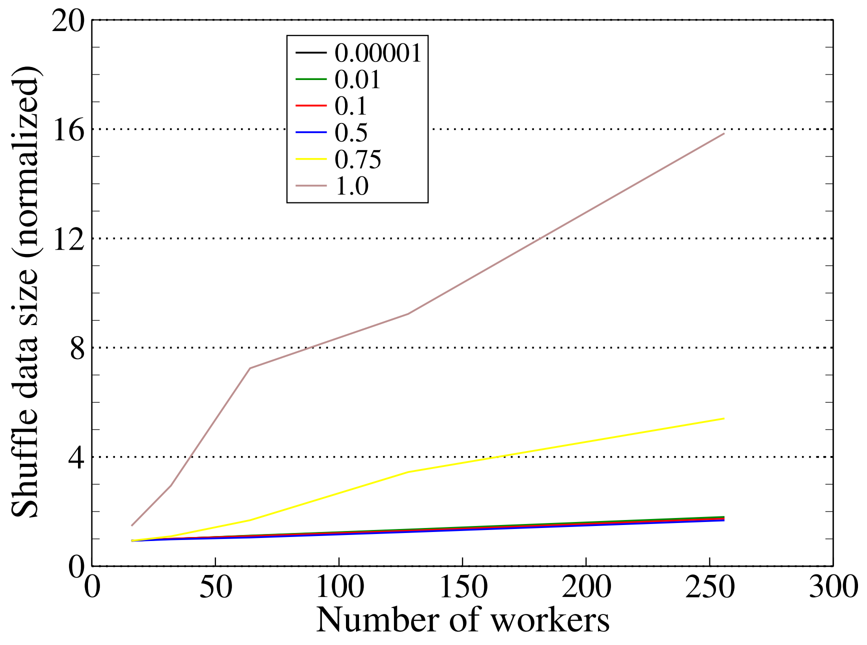


Figure 4.15. *Produced shuffle data size of the patch-based algorithm as the number of workers N increases for different skew factors in tables S (∂_S)*

As we can observe in Figure 4.15, the shuffle data size seems to be increasing as the number of workers increases for all skew factors. Since the same initial data are distributed to more workers it is expected that more subgroups and more data replication will take place. The rate that the size increases is different however depending on the skew factor. For small skew factors ($\partial \leq 0.5$) the rate is very low, however for larger skew factors the rate increases significantly and therefore the shuffle data size increases faster as the number of workers increases.

4.3 Execution time

In the third set of experiments we focus on the overall evaluation of the data shuffling and reduce-side execution as the main phases that differentiate between the patch-based and hash-based algorithms and can affect the final observed execution time. The overhead inserted from the initial data loading from disk into memory is common in both algorithms, therefore omitted from our analysis, while the overhead of the execution of our patch-based algorithm is very small (milliseconds) and could not be visible in the following graphs, therefore it is omitted too.

Initially we assume a distributed cluster setup where each worker has 4 cores with a clock speed of 2.2GHz, therefore in total 8.8GHz, and a simple 1Gbit ethernet network with transfer rate 125MB/s. A single record of either table is considered to have a size of 1KB. Various studies [12, 7, 8] around the performance of the hash-join and sort-merge-join algorithms have indicated that the cycles required in hash-based implementations per join output tuple are 30 (independent of the record size). It is safe to use this amount for our experimental evaluation assuming that the reduce-side operation is using key matching techniques to join/aggregate the records.

The final observed execution time is of course dependent on the slowest worker, and the model for the time estimation is created as follows:

$$t(x, y) = \max_{w}\left(30 \cdot x(w) \div 8.8 \cdot 10^{-9} + 1000 \cdot y(w) \div 125 \cdot 10^{-6}\right)$$

where x is a list containing the number of output tuples that each worker w calculated, and y is a list containing the number of records that were sent to each worker w.

Assuming that table T has zero skew in the keys ($\partial_T \cong 0$), we vary the skew factor in table S using values in the range $\partial_S : [0, \ldots 3]$. Both tables contain multiple records for each attribute value in this case, using a different distribution however. The number of workers and the parameters used for creating the datasets are presented in Table 4.9.

Table 4.9. *Parameters used for the experiments in Figure 4.16*

N	32
$\lvert V\rvert$	32
$\lvert S\rvert$	200000
$\lvert T\rvert$	200000
∂_T	0.00001

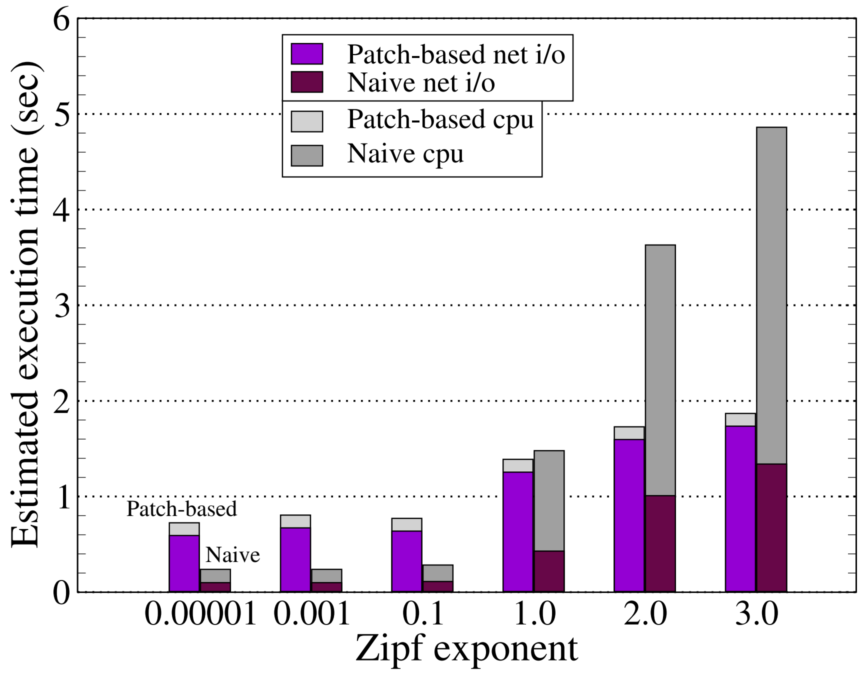


Figure 4.16. *Estimated execution time split into shuffling and execution phases with the patch-based and naive hash-based algorithms for different skew factors in table S (∂_S)*

The diagram presenting the total estimated execution time broken into the data shuffling (net i/o) and reduce-side processing (cpu) phases appears in Figure 4.16 for both algorithms. A we can observe in this Figure, for smaller values of the skew factor our algorithm spends more time in transferring the bigger amount of replicated data that it produces in one worker. Our implementation is 3X slower with no to little skew, as it uses a methodology which is not profitable in this case compared with the hash-based partitioning. However, for skew factors $\partial_S \geq 1.0$ our implementation is faster than the hash-based partitioning which is seriously affected by the skew. Although our algorithm's performance is generally affected only by the data shuffling phase, we can observe that the hash-based technique is affected in both the shuffling and execution phases by the presence of skew that struggles one worker. Our algorithm is 1.1X faster for $\partial = 1.0$ and 2.6X faster for $\partial = 3.0$.

As a general observation, the data shuffling phase in our algorithm is proportionally constantly dropping as the skew factor increases compared with the hash-based technique. Although the total size of shuffle data becomes 15.6 times more than the hash-based for $\partial = 3.0$ as we have previously seen, these data are equally distributed among the 32 workers therefore a single worker will receive $15.6 \cdot 400000 \div 32 = 195K$ records. On the contrary, with the hash-based approach for $\partial_S = 3.0$ 84% of the S records and 3% of the T records i.e. 174K records will be sent to a single worker which will of course be the slowest. These numbers fully explain the difference that we observe in Figure 4.16 when $\partial_S = 3.0$ for the net i/o. Our algorithm greatly profits from the uniform distribution of data at large skew factors, and the impact of the increased size of shuffle data becomes negligible.

Next, assuming that both tables S and T present the same amount of skew, we vary the skew factor using values in the range $\partial_S : [0, \ldots 1]$. Both tables contain multiple records for each attribute value, resembling a *foreign-foreign* key join scenario. The number of workers and the parameters used for creating the datasets are presented in Table 4.10.

Table 4.10. *Parameters used for the experiments in Figure 4.17*

N	32
$\|V\|$	32
$\|S\|$	200000
$\|T\|$	200000

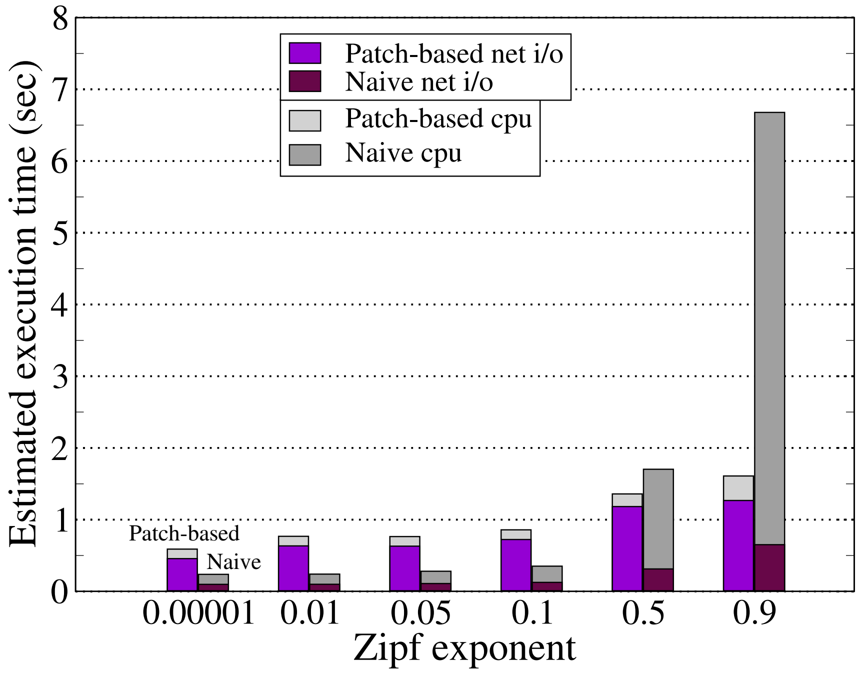


Figure 4.17. *Estimated execution time split into shuffling and execution phases with the patch-based and naive hash-based algorithms for various skew factors applied in both tables S and T*

The diagram presenting the total estimated execution time broken into the data shuffling (net i/o) and execution (cpu) phases appears in Figure 4.17 for both algorithms. The inductions are similar with the ones for Figure 4.16. As we can observe in this Figure, our algorithm is 2.47X slower for zero skew but it becomes 1.25X faster for $\partial = 0.5$ and 4.2X faster for $\partial = 0.9$.

Finally, assuming that table T has zero skew in the keys ($\partial_T \approxeq 0$), we vary the skew factor in table S using values in the range $\partial_S : [0, \ldots 3]$. In this case table T contains a unique appearance of each key in its records, resembling a *primary-foreign* key join scenario. The number of workers and the parameters used for creating the datasets are presented in Table 4.11.

Table 4.11. *Parameters used for the experiments in Figures 4.6 and 4.7*

N	32
$\|V\|$	32
$\|S\|$	200000
$\|T\|$	32
∂_T	0.00001

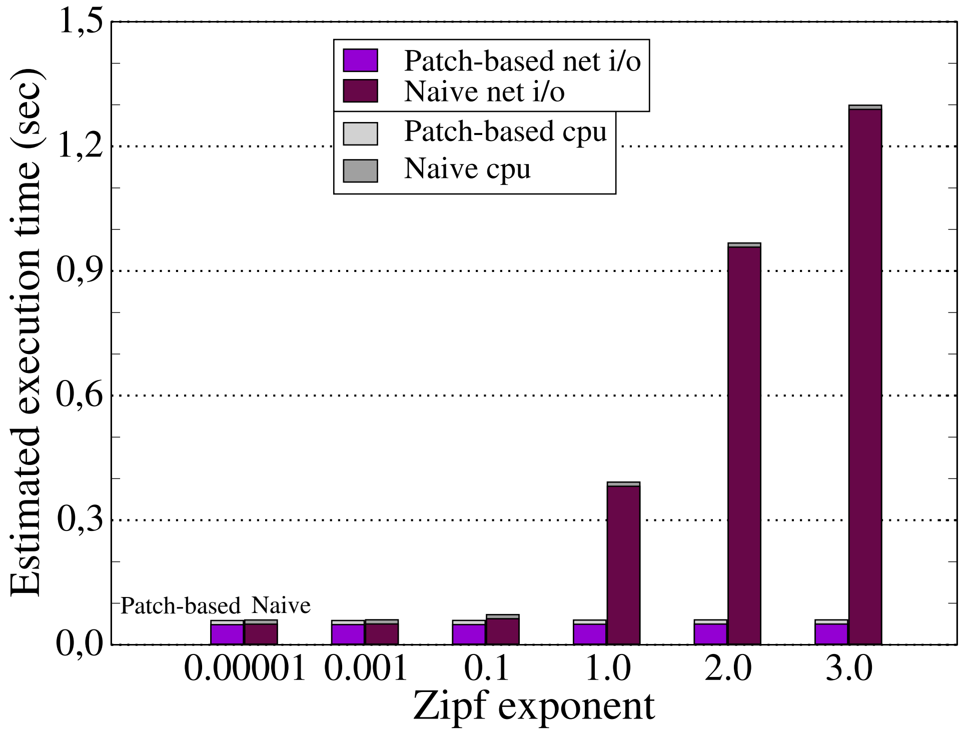


Figure 4.18. *Estimated execution time split into shuffling and execution phases with the patch-based and naive hash-based algorithms for different skew factors in table S (∂_S) having unique keys in table T*

The diagram presenting the total estimated execution time broken into the data shuffling (net i/o) and execution (cpu) phases appears in Figure 4.18 for both algorithms. As we can observe in this Figure, our algorithm is always performing the same or better compared the naive hash-based technique. In particular, for small amounts of skew the performance of our algorithm is similar with the naive hash-based, and becomes significantly faster for $\partial \geq 0.1$ being 25X faster for $\partial = 3.0$. For this particular case, the overall execution time is mainly affected by the shuffling phase as the reduce-side processing time appears to be negligible. The amount of data that is transferred in a single worker by the hash-based technique becomes therefore a major struggling point as the skew increases. On the contrary, for $\partial = 3.0$ our algorithm transfers in total nearly half of data of the hash-based technique and also distributes them more uniformly compared with the hash-based technique, as we have previously seen. This is the reason why we observe a huge difference in Figure 4.18 for the net i/o with $\partial = 3.0$, as approximately 84% of the records are sent to the slowest worker with the naive technique and only 3.4% using our algorithm.

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

- Our patch-based algorithm is on average 4.3X faster than the hash-based approach for a moderate to high skew factor ($\partial = 1.0$).
- Our patch-based algorithm is 25X faster than the hash-based approach for an extremely high skew factor ($\partial = 3.0$) in a primary-foreign key join scenario.

- Our patch-based algorithm is on average 2.1X slower than the hash-based approach when there is zero skew in the data.
- Our patch-based algorithm has the same performance with the hash-based approach when there is zero skew in the data in a primary-foreign key join scenario.
- The network-related overhead is greater with our algorithm in foreign-foreign key join scenarios due to replication of records.
- The network-related overhead is smaller with our algorithm in primary-foreign key join scenarios.
- At high skew levels the load-related overhead is eliminated with our algorithm.
- 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.

Chapter 5

Epilogue

5.1 Conclusions

In this diploma thesis we presented a novel partitioning algorithm for distributing skewed data to workers in order to eliminate load imbalances that incur for reduce-side operations when skew is present. The methodology used relies on the subset-replicate state-of-the-art partitioning technique which includes data replication leading to increased network traffic. Our algorithm uses statistics for the data distribution and data location and aims to facilitate as much local processing as possible in an effort to reduce network traffic. The implemented algorithm is skew-insensitive as it does not need any use case specific parameterization, and can be integrated in any distributed execution engine in place of the shuffle mechanism which is commonly used for operations on unordered datasets. The performance evaluation of our algorithm confirms that the load balancing performed with our algorithm is always perfectly even, and that the overall performance is superior compared with the typical hash-based partitioning as it is proved to be mainly affected by the worker load and less by the network i/o overheads.

5.2 Future work

The planned future work can be summarized as follows:

- Design and implementation of the record tagging procedure which is required for completing the network transfer before executing any reduce-side operations, which is an NP-hard problem.
- Design and evaluation of more sophisticated rules for the subgroup creation decisions in order to further decrease the network traffic.
- A more extensive performance evaluation with fully implemented join operations in large scale setups and compare with other implementations.
- Adapt the logic of the partitioning algorithm to present better performance for datasets with a low level of skew.
- Transform the statistics collection and record tagging procedures to be executed dynamically during execution.

Appendices

A

Code for data generation

```java
import java.util.*;
import java.util.stream.IntStream;
import org.apache.commons.math3.distribution.ZipfDistribution;

static String[][] tableS, tableT;

// Generate two datasets with custom skew based on the Zipfian distribution
// s_size, t_size: number of records in each table
// s_fact, t_fact: the Zipf exponent used for each dataset
// numKeys: number of distinct keys
// numWorkers: number of workers
static void generate_data(int s_size, int t_size, double s_fact, double
    t_fact, int numKeys, int numWorkers){
   tableS = new String[s_size][];
   tableT = new String[t_size][];
   System.out.println("Creating datasets..");
   ZipfDistribution zipfDistribution = new ZipfDistribution(numKeys, s_fact);
   ZipfDistribution finalZipfDistribution = zipfDistribution;
   IntStream.range(0, s_size)
      .parallel()
      .forEach(i -> {
         int sample = finalZipfDistribution.sample();
         tableS[i]=new String[]{"stuple"+i, "key"+sample};
      });

   zipfDistribution = new ZipfDistribution(numKeys, t_fact);
   ZipfDistribution finalZipfDistribution1 = zipfDistribution;
   IntStream.range(0, t_size)
      .parallel()
      .forEach(i -> {
         int sample = finalZipfDistribution1.sample();
         tableT[i] = new String[]{"key"+sample, i+"ttuple"};
      });
}
```

B

Code for data placement in workers

```java
import java.util.*;
import java.util.Map.Entry;
import java.util.concurrent.ConcurrentHashMap;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

static ConcurrentHashMap<Integer, List<String[]>> localTData, localSData;

// Split equally each table's data to the workers
static void compute_local_data(int numWorkers){
   int s_per_worker = tableS.length / numWorkers;
   int t_per_worker = tableT.length / numWorkers;

   System.out.println("Creating random worker input data..");
   localTData = new ConcurrentHashMap<>();
   localSData = new ConcurrentHashMap<>();

   IntStream.range(0, numWorkers)
      .parallel()
      .forEach(i -> {
         int extra=0;
         // In the last worker add the remainder number of records
         if (i==numWorkers-1)
            extra = tableT.length % numWorkers;
         localTData.put(i, Arrays.stream(Arrays.copyOfRange(tableT,
             i*t_per_worker, i*t_per_worker + t_per_worker +
             extra)).collect(Collectors.toList()));
      });

   IntStream.range(0, numWorkers)
      .parallel()
      .forEach(i -> {
         int extra=0;
         // In the last worker add the remainder number of records
         if (i==numWorkers-1)
            extra = tableS.length % numWorkers;
         localSData.put(i, Arrays.stream(Arrays.copyOfRange(tableS,
             i*s_per_worker, i*s_per_worker + s_per_worker +
             extra)).collect(Collectors.toList()));
```

```
        });
}
```

C

Code for statistics computation

```
static Map<String, Integer> tableSGroupSizes, tableTGroupSizes;
static Map<String, Long> groupSizes;
static Long totalTuples, Z;
static ConcurrentHashMap<Integer,Map<String, Integer>> SWeights, TWeights;

// Compute statitics regarding the distribution and location of data
static void compute_stats(){
   // Compute cardinality for each key in table S
   List<String[]> tableSGroupSizesList = Arrays.stream(tableS)
      .collect(Collectors.groupingBy(ints -> ints[1]))
      .entrySet().stream()
      .map(entry -> new String[]{
            entry.getKey(),
            String.valueOf((long) entry.getValue().size())
      })
      .collect(Collectors.toList());

   tableSGroupSizes = tableSGroupSizesList.stream()
      .collect(Collectors.toMap(x -> x[0], x->Integer.valueOf(x[1])));
   System.out.println("Table S cardinality: "+tableSGroupSizes.toString());

   // Compute cardinality for each key in table T
   List<String[]> tableTGroupSizesList = Arrays.stream(tableT)
      .collect(Collectors.groupingBy(ints -> ints[0]))
      .entrySet().stream()
      .map(entry -> new String[]{
            entry.getKey(),
            String.valueOf((long) entry.getValue().size())
      })
      .collect(Collectors.toList());
   tableTGroupSizes = tableTGroupSizesList.stream()
      .collect(Collectors.toMap(x -> x[0], x->Integer.valueOf(x[1])));
   System.out.println("Table T cardinality: "+tableTGroupSizes.toString());

   // Compute number of output tuples of each join group(key)
   tableSGroupSizesList.addAll(tableTGroupSizesList);
   List<String[]> groupsSizesList =
       Arrays.stream(tableSGroupSizesList.toArray(new String[0][]))
      .collect(Collectors.groupingBy(ints -> ints[0]))
```

```
        .entrySet().stream()
        .map(entry -> new String[]{
              entry.getKey(),
              String.valueOf(((long) entry.getValue().size() <2) ? 0 :
                    entry.getValue().stream().mapToInt(num->Integer.parseInt(num[1]))
                          .reduce(1, (a,b) -> (a+b)>1? a*b:0))
        })
        .collect(Collectors.toList());
    groupSizes = groupsSizesList.stream()
        .collect(Collectors.toMap(x -> x[0], x->Long.valueOf(x[1])));
    System.out.println("Group sizes: "+groupSizes.toString());

    // Total output tuples/load
    totalTuples = groupSizes.values().parallelStream().reduce(0L, Long::sum);
    System.out.println("Total load: "+totalTuples);

    // Average load computed per worker (initial capacity)
    Z = totalTuples/numWorkers+1;
    System.out.println("Initial maximum worker's capacity: "+Z);

    // Compute the number of local records per key in each worker
    TWeights = new ConcurrentHashMap<>();
    SWeights = new ConcurrentHashMap<>();
    IntStream.range(0, numWorkers)
        .parallel()
        .forEach(i -> {
           List<String[]> SworkerGroupSizesList = localSData.get(i).stream()
              .collect(Collectors.groupingBy(ints -> ints[idx1]))
              .entrySet().stream()
              .map(entry -> new String[]{
                 entry.getKey(),
                 String.valueOf((long) entry.getValue().size())
              })
              .collect(Collectors.toList());
           SWeights.put(i, SworkerGroupSizesList.stream()
              .collect(Collectors.toMap(x -> x[0], x->Integer.valueOf(x[1]))));

           List<String[]> TworkerGroupSizesList = localTData.get(i).stream()
              .collect(Collectors.groupingBy(ints -> ints[idx2]))
              .entrySet().stream()
              .map(entry -> new String[]{
                 entry.getKey(),
                 String.valueOf((long) entry.getValue().size())
              })
              .collect(Collectors.toList());
           TWeights.put(i, TworkerGroupSizesList.stream()
              .collect(Collectors.toMap(x -> x[0], x->Integer.valueOf(x[1]))));
        });
}
```

Bibliography

[1] Adaptive execution. https://www.databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html.

[2] Apache spark. https://spark.apache.org/.

[3] Bin packing problem. https://en.wikipedia.org/wiki/Bin_packing_problem.

[4] Presto db. https://prestodb.io/.

[5] Rectangle packing problem. https://en.wikipedia.org/wiki/Rectangle_packing.

[6] Subset sum problem. https://en.wikipedia.org/wiki/Subset_sum_problem.

[7] Balkesen, C., Alonso, G., Teubner, J., and Özsu, M. T. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow. 7*, 1 (sep 2013), 85–96.

[8] Balkesen, C., Teubner, J., Alonso, G., and Özsu, M. T. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)* (2013), pp. 362–373.

[9] Chen, Y., Wang, J., Lu, Y., Han, Y., Lv, Z., Min, X., Cai, H., Zhang, W., Fan, H., Li, C., Guan, T., Lin, W., Jia, Y., and Zhou, J. Fangorn: Adaptive execution framework for heterogeneous workloads on shared clusters. *Proc. VLDB Endow. 14*, 12 (jul 2021), 2972–2985.

[10] Duggan, J., Papaemmanouil, O., Battle, L., and Stonebraker, M. Skew-aware join optimization for array databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), SIGMOD '15, p. 123–135.

[11] Gupta, A. M., Gadepally, V., and Stonebraker, M. Cross-engine query execution in federated database systems. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)* (2016), pp. 1–6.

[12] Kim, C., Kaldewey, T., Lee, V. W., Sedlar, E., Nguyen, A. D., Satish, N., Chhugani, J., Di Blas, A., and Dubey, P. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proc. VLDB Endow. 2*, 2 (aug 2009), 1378–1389.

[13] KWON, Y., BALAZINSKA, M., HOWE, B., AND ROLIA, J. Skewtune: Mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), SIGMOD '12, p. 25–36.

[14] LI, R., RIEDEWALD, M., AND DENG, X. Submodularity of distributed join computation. In *Proceedings of the 2018 International Conference on Management of Data* (2018), SIGMOD '18, p. 1237–1252.

[15] NGUYEN, T. T., TRAHAY, F., DOMKE, J., DROZD, A., VATAI, E., LIAO, J., WAHIB, M., AND GEROFI, B. Why globally re-shuffle? revisiting data shuffling in large scale deep learning. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2022), pp. 1085–1096.

[16] RÖDIGER, W., IDICULA, S., KEMPER, A., AND NEUMANN, T. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)* (2016), pp. 1194–1205.

[17] ZAHARIA, M., XIN, R. S., WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, M. J., GHODSI, A., GONZALEZ, J., SHENKER, S., AND STOICA, I. Apache spark: A unified engine for big data processing. *Commun. ACM 59*, 11 (oct 2016), 56–65.