EdgeFrame: Worst-Case Optimal Joins for Graph-Pattern Matching in Spark

Per Fuchs[*]
per.fuchs@cs.tum.edu
Technische Universität München
München, Germany

Peter Boncz
peter.boncz@cwi.nl
CWI
Amsterdam, the Netherlands

Bogdan Ghit
bogdan.ghit@databricks.com
Databricks Inc.
Amsterdam, the Netherlands

ABSTRACT

We describe the design and implementation of EdgeFrame: a graph-specialized Spark DataFrame that caches the edges of a graph in compressed form on all worker nodes of a cluster, and provides a fast and scalable Worst-Case-Optimal Join (WCOJ) that is especially useful for matching of complex and cyclical patterns in large graphs. Our choice to forego shuffle- or communication-based WCOJ is motivated by our analysis of the Shares algorithm for distributed WCOJ, that was proven communication-optimal, but which we show to quickly deteriorate to a full broadcast of all data already with moderately complex graph patterns. Our work shows that specializing WCOJ to a multi-way self-join, and leveraging compressed storage, provides a significant opportunity for better WCOJ performance. Finally, we investigate WCOJ parallelization and load-balancing strategies and show that fine-grained dynamic load-balancing with work-stealing is to be preferred, creating interesting insights and challenges for the future evolution of the Spark scheduler.

CCS CONCEPTS

- **Information systems → Join algorithms**; **MapReduce-based systems**.

ACM Reference Format:
Per Fuchs, Peter Boncz, and Bogdan Ghit. 2020. EdgeFrame: Worst-Case Optimal Joins for Graph-Pattern Matching in Spark. In *3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'20), June 14, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3398682.3399162

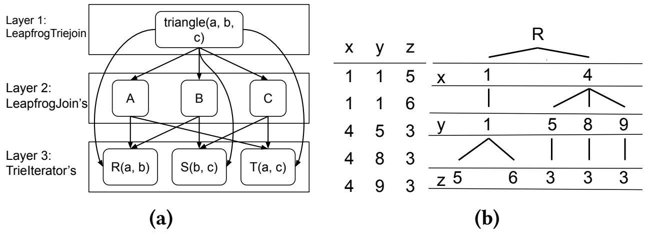
1 INTRODUCTION

Pattern-matching is an important use-case of graph query processing [7, 21, 38]. In essence, such queries evaluate a join graph, shaped by the graph-pattern over one table holding the edges of the graph. Each edge in the graph pattern corresponds to one join in the join graph. Such multi-join queries are traditionally executed by operators that compute one join at-a-time (binary joins: a join between two tables); of course, preceded by query optimization that determines an efficient order of their computation. However, important – mostly theoretical – work in the past years has shown that so-called, *worst-case optimal join* (WCOJ) [32] algorithms, such as the Leapfrog TrieJoin (LFTJ) [40] can perform multi-way joins with lower time complexity than running multiple binary joins [10]. This finding has had little impact on data warehousing and OLAP systems where the most popular join patterns are *primary-foreign-key* (PK-FK) with tree or snowflake shapes which are acyclic. Since such joins have linear complexity in the input size, binary joins such as hash joins are indeed worst-case optimal. In contrast, pattern-matching in graphs typically employs *foreign-foreign-key* (FK-FK) joins (i.e. over an edge table), and are likely to include cycles. These two conditions tend to cause binary-joins to generate intermediate results that can be much larger than the final result, i.e. under these circumstances they are very far from worst-case optimal. Examples of cyclic graph-processing queries include the diamond query pattern used by Twitter [20] to recommend people to follow, the detection of cycles for fraud detection [12, 36, 37], and the community detection in networks [15, 30].

In this paper, we present the design and implementation of GraphWCOJ, a worst-case optimal join specialized for graph-processing queries. To put these algorithms in the hands of practitioners who analyze large graph data, we developed GraphWCOJ inside the Spark framework, which is by far the most widely used scalable data science processing framework and is often used for large graph data. For example, GraphFrames [14], GraphX [18] (a Pregel [27] implementation) or graph query languages such as G-CORE [6] and openCypher with Cypher for Apache Spark or CAPS [35] all aim to ease graph processing on Spark – and CAPS became

[*] This work has been completed while working at CWI in the Netherlands

GRADES-NDA'20, June 14, 2020, Portland, OR, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8021-8/20/06…$15.00
https://doi.org/10.1145/3398682.3399162



x	y	z
1	1	5
1	1	6
4	5	3
4	8	3
4	9	3

Figure 1: An overview of the three layers in a Leapfrog TrieJoin algorithm for the triangle query. The arrows show the dependencies among components (a). A 3-ary relationship as table and trie (b).

standard part of Apache Spark 3.0. However, pattern matching operations with all these interfaces are still executed using multiple binary join operators, which – as we argued – is suboptimal.

Our work focused on two research questions. Firstly, existing WCOJ algorithms are designed for *n*-ary joins between *n* different tables. In contrast, graph pattern matching performs these joins over the same relationship (the edge table), making this an *n*-ary *self*-join. Our first research question thus is: how can WCOJ algorithms be specialized for graphs with such self-joins, and what performance benefits does this bring? Secondly, the scaling of previous attempts to run WCOJ algorithms on clusters (e.g., using MapReduce and Spark) has been relatively poor. A state-of-the-art implementation – called Shares – in the distributed system Myria [13] achieved a speedup of only 8 on 64 workers. Although there are theoretical results on communication costs of WCOJ in a distributed setting [2, 3, 11, 23] and the Shares algorithm was proven communication-optimal [13], communication costs are observed to be significant specifically in highly-connected graph pattern matching queries. Therefore, designing a distributed WCOJ with both high performance and good scalability remains an unsolved challenge, which we try to address in our work.

The results of our effort is the EdgeFrame, a special Spark DataFrame that incorporates GraphWCOJ and caches the edge structure of a graph in compressed form on all workers in the cluster. This is a practical approach that circumvents the inherent communication bottlenecks of WCOJ on distributed graphs, and maximally profits from specializing the algorithm to graph workloads. We analyze the performance of EdgeFrame with analytical graph queries on real-world datasets and release it in open source.

Our work makes the following contributions:

(1) We find that the Shares algorithm for distributed WCOJ, which has proven optimal communication cost, must replicate over 50% of all edge tuples on each worker even for simple graph patterns (Section 3), effectively causing close to worst case communication cost and memory consumption. This inherently leads to poor scalability. Hence, we propose EdgeFrame, which caches in compressed form the edge relationship on all workers, to enable a shuffle-free WCOJ for arbitrary queries afterwards.
(2) We design and implement GraphWCOJ, which is a worst-case optimal join specialized to graph pattern matching (section 4.1). We demonstrate a speedup of up to 11 times over a non-specialized LFTJ implementation.
(3) We then focus on effectively parallelizing GraphWCOJ on an EdgeFrame, that is cached on all workers. Since, most real world graphs today have a heavily skewed degree distribution, we propose to use dynamic work-stealing to partition the work (section 5). This provides good results when applied between the executors on each worker. However, our results suggest that to scale well on many workers, the Spark scheduler should evolve to facilitate more dynamic task creation and better handle jobs consisting of many tasks.

2 BACKGROUND

The worst-case time-complexity of a WCOJ algorithm matches the theoretical bound on the join output size, given by taking the join input size to the power of the fractional edge cover derived from the join query shape's hypergraph [10, 33]. This bound can be asymptotically lower than the complexity achieved by traditional, binary join plans, where each such binary join has a worst-case complexity that is quadratic to its input size. The advantage of this worst-case bound surfaces in practice when join connectivity is skewed, but also in multi-join queries which cyclical join shape, where the individual joins have a high out-degree.

Next, we explain the `Leapfrog TrieJoin`, a worst-case optimal join which was implemented in the LogicBlox commercial database system. The `Leapfrog TrieJoin` (LFTJ) is a variable-oriented join which requires an order over the variables of the join query to produce the result. As an example, consider the triangle query $triangles(a, b, c) \leftarrow R(a, b), S(b, c), T(a, c)$, with the variable ordering as a, b, c. Executing an LFTJ requires the input relationships to be sorted lexicographically in increasing order by the given variable ordering. For instance, R needs to be sorted by a and b respectively. We first fix a possible binding for variable a, while further setting bindings for b given a, and finally c given both a and b. We call the algorithm *variable-oriented* as it allows to enumerate the query result by means of a backtracking depth-first search for possible bindings which requires no intermediate results. Finding the best variable ordering for a WCOJ is important [8, 13, 28] and somewhat analogous to join ordering in query optimization.

We now explain how LFTJ works in case of the triangle query shown in Figure 1a. At a high-level, a LFTJ employs three layered components, each of which may have multiple instances. The first layer is the TrieIterator which is instantiated for each input relationship. The TrieIterator interface represents the input relationship as a trie data structure with all values for the first attribute on the first level, the values for the second attribute on the second level and so on, as we show in Figure 1b. As each level stores all possible values for its attribute, we can enumerate all tuples of the relationship by a depth-first traversal of the trie. The TrieIterator component offers the ability to traverse each edge of the trie, the sequence of all siblings and an upper-bound search over the values of all siblings in at most $O(\log N)$, e.g. by employing binary search over sorted arrays [13]. The second layer incorporates a LeapfrogJoin for each variable referenced in the join. To generate the bindings for a join variable, a LeapfrogJoin uses the underlying TrieIterators to *intersect* the possible values for all input relationships that contain the variable. Figure 1a shows three LeapfrogJoin instances for variables a, b, and c, each of which employ two TrieIterators. Finally, a single Leapfrog TrieJoin instance interacts with the previous layers to generate all possible bindings for the join. To do so, we can acquire one binding for each variable using the underlying LeapfrogJoins. After setting a binding for the current variable, we move the TrieIterators that contain the variable to the next level. Finally, the LFTJ operator emits a tuple when all variables have a binding. We repeat this process to find the next possible binding through backtracking.

Besides the LFTJ, there are two other WCOJ algorithms: NPRR [31] and Generic Join [33]. We use the LFTJ as the base for our work because it is implemented by two commercial products, i.e., LogicBlox [8] and RelationalAI [1]. Furthermore, the performance of the LFTJ is well understood by prior work [13, 34] and it has multiple open-source implementations [13, 39].

3 DISTRIBUTED WCOJ ANALYSIS

Myria is a MapReduce-based system with a typical multi-worker architecture which incorporates a distributed WCOJ implementation for graph pattern matching based on the Shares algorithm [13]. The Shares shuffle algorithm [3] enables Myria to solve n-ary joins in a single shuffle round. Nevertheless, shuffling leads to a significant amount of duplicated work done by multiple worker nodes. As a consequence, the scalability of the Shares algorithm is rather limited, with a speedup of 8 on 64 workers relatively to the execution on 2 workers. In this section, we present the operation of Shares and we analyze its scaling properties.

Pattern	Edges	64 workers	128 workers
Triangle	3	0.18	0.12
House	5	0.42	0.32
4-clique	6	0.59	0.44
Diamond	8	0.76	0.67
5-clique	10	0.90	0.82

Table 1: Fraction of all edges assigned to every worker by the optimal Shares algorithm. A replication free approach with linear scaling would require $1/\#workers$ edges per worker.

Given an n-ary join and all input relationships, the Shares shuffle overestimates the tuples that can be joined and ensures they are co-located on at least one worker. Then, we join the local tuples on each worker and so, we obtain the n-ary join result as the union of each worker output. To achieve such a partitioning, we organize the cluster workers in a multi-dimensional *hypercube* with one dimension per join attribute. In each dimension we set a number of workers p_i such that $p_0 \cdot p_1 \ldots p_k \leq w$, where k and w are the number of dimensions and the cluster size, respectively. We employ the same method proposed by the Myria system [13] to build a hypercube in which each worker is assigned a multi-dimensional coordinate. We employ a hash function to map the join attributes in each tuple to multiple of these coordinates and assign tuples to the matching workers. Hence, we replicate the tuples of a relationship across each dimension of the variables which are unbounded. In other words, we need to replicate the tuples of each variable that is unbound in a relationship across all workers in the corresponding dimension. Because graph pattern matching is an n-ary self-join with two attributes in each relationship, the Shares shuffle scales poorly with the number of variables because tuples are replicated along all but two dimensions. Furthermore, scaling with the cluster size is also rather limited because the dimension sizes increase with the number of workers.

We analyze Shares scalability by giving a closed-form formula of the fraction of tuples of the edge relationship that are allocated on each worker. We model the likelihood of a tuple being assigned to a worker with a Poisson binomial distribution. Given n independent binary trials and the probability u_i of the i^{th} trial succeeding, we set the probability that k out of n trials succeed to $\Pr(n, k, u_0 \ldots u_n)$. Let R be the number of relationships (edges) in the join. Further, we denote the sizes of the two dimensions bound by the attributes of the i^{th} relationship by $size_1(r_i)$ and $size_2(r_i)$. Thus, we estimate the probability that a tuple is not assigned to an arbitrary, fixed worker by setting $k = 0$, $n = R$, and $u_i = \frac{1}{size_1(r_i) * size_2(r_i)}$. In this way we can predict the number of tuples assigned to each worker by $|E| \times (1 - \Pr(|R|, 0, u_0, \ldots, u_{|R|})$, where E represents the edge relationship.

Table 1 shows the fractions of all edge tuples per worker for typical graph patterns. Two things stand out. Firstly, even for relatively small patterns like the 4-clique and diamond, each of the 64 workers holds **more than half** of all tuples. Secondly, doubling the size of the cluster to 128 workers is not an efficient way of reducing the number of replicated tuples per worker. From this analysis and the fact that Shares was proven to be **optimal** in communication cost [11], we conclude that for graph pattern matching, WCOJ algorithms that re-partition the edge relationship among nodes in a cluster by definition deteriorate to replicating the entire graph everywhere, causing maximal communication cost and memory usage.

This analysis motivated EdgeFrame: it is a Spark DataFrame in a broadcast variable that is cached on all Spark workers; and it stores the graph in a compact, compressed, format to limit memory consumption. Graph pattern matching on EdgeFrame therefore does not need to move any graph edges over the network, its GraphWCOJ method can exploit the compressed format to make WCOJ even faster, and the main challenge is how to parallelize its computational work evenly over all workers and CPU cores.

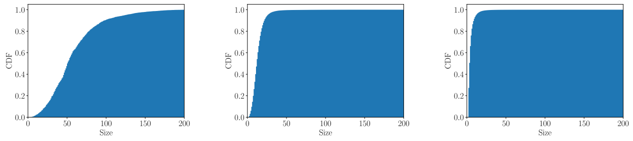
4 EDGEFRAME DESIGN

This section presents the design and implementation of our Spark EdgeFrame. First, we describe the specialization of a generic LFTJ to graphs by the use of CSR and a simpler intersection algorithm in Section 4.1. We expose GraphWCOJ in Spark [16] through an API resembling the GraphFrames API described in Section 4.2. Further, we detail the integration into Catalyst, Spark's SQL optimizer, in Section 4.3.

4.1 Graph-Specific Optimizations

Using CSR is a common approach of representing static graphs in a compact format. For simplicity, we assume that the graph's vertices are identified by non-negative integers from 0 to $|V| - 1$. However, we support arbitrary node identifiers by storing a translation in an additional array of size $|V|$. In particular, CSR uses two arrays to represent the topology of a graph. The `Adjacency Lists` array of $|E|$ elements is a projection of the edge relationship on the *dst* attribute. The `Indices` array of $|V| + 1$ elements stores the indices in the `AdjacencyLists` array. Therefore, we can find all vertices directly reachable from a given source vertex *src* by retrieving the index stored in `Indices` at position *src* to access the `AdjacencyLists` array.

We use the CSR of the graph as the underlying data structure of the `TrieIterators` of the LFTJ algorithm. This has a major advantage over using a sorted, columnar representation as in previous work [13, 40]. The CSR acts as an index for the first level of the `TrieIterators` allowing us to find the upper bound of any value by single array access instead of using a binary search which is the most expensive operation in the LFTJ algorithm [13]. In contrast, searching on the second level of the `TrieIterators` is fast because most graphs have low out-degrees.



(a) Smallest iterator **(b) Biggest intersection** **(c) Total intersection**

Figure 2: Metrics characterizing the LFTJ intersection when running the `5-clique` on the SNB dataset of scale factor 1.

We found the *n*-way intersection algorithm used by the `LeapfrogJoin` operators to be inefficient for graph pattern matching. The algorithm repeatedly searches the upper-bound for the largest value in the iterator at the lowest value until all iterators point to the same value. This is done to *leap* over many values in the lowest iterator which is inefficient for lists with huge gaps between the values, i.e. for graph adjacency lists. Besides, it requires the iterators are sorted on initialization. Instead, we propose a simpler algorithm which we describe in next.

To illustrate the motivation for our intersection algorithm, we use the `5-clique` query on the SNB dataset and gather runtime statistics that characterize the intersection. Figure 2a shows that for roughly 80% of the intersections the smallest iterator has a size that is lower than 80 elements. This distribution is different than the typical long-tail distribution encountered in power-law graphs because the smallest iterator is selected from a total of 5 iterators. We find that in 80% of the intersections the largest intersection of the smallest iterator with other iterators contains less than 21 elements (Figure 2b) and the size of the total intersection is less than 5 (Figure 2c). Given these observations, we propose to use multiple pair-wise intersections starting with the intersection between the smallest iterator and any other second-level iterator. Also, we defer intersection with all first-level iterators towards the end because they are unlikely to limit the intersection. Given the fast convergence to the final size of the intersection, this approach is faster than the original algorithm because it limits the searches on iterators that are intersected later. Furthermore, instead of sorting the iterators upfront, we only need to find the smallest iterator.

4.2 The EdgeFrame API

The API of our WCOJ implementation is inspired by the existing Spark GraphFrames API [14]. The user can define a pattern, so that each edge is expressed as `(a) - [] -> (b)`, where *a* and *b* represent the source and the destination, respectively. A pattern may consist of multiple edges separated by semicolons. We use standard *homomorphism* semantics: a variable in a pattern is not guaranteed to be a distinct element in the graph. For example, the pattern `(a) - [] -> (b); (b) - [] -> (c)` may be either a linear path of length two or a circle between *a* and *b*. The WCOJ is invoked by calling the `findPattern` method on the input DataFrame which typically has two columns, one for source and the other for the destination. The method takes the graph pattern and the variable ordering as input parameters.

```
// Load the edge relationships
val df = spark.read.csv("edge_relationships")
// Set of node identifiers
val nodes = Seq("a", "b", "c")
// Triangle pattern
val pattern = """
| (a) - [] -> (b);
| (b) - [] -> (c);
| (a) - [] -> (c)
""".stripMargin
// Invoke WCOJ to find triangles
df.findPattern(pattern, nodes).show()
```

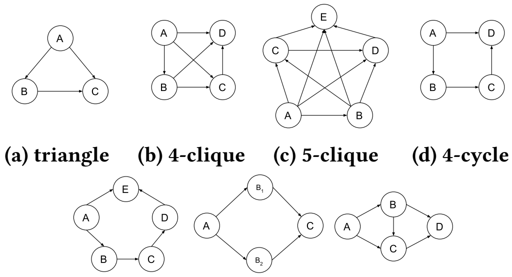
Listing 1: Finding triangles in graph with `EdgeFrame`

4.3 Catalyst Integration

In this section, we present the details of integrating `GraphWCOJ` into Catalyst [9]. To this end, we created new *operators* for the logical and physical plans and we implemented a *strategy* for performing the WCOJ execution planning.

At the logical plan level, we introduce a new *logical WCOJ* operator. The logical operator has three children, the first two of which represent the input relationship, whereas the remaining one is an empty RDD with as many partitions as the desired level of parallelism. Our Catalyst strategy converts the logical operator into multiple physical operators. The strategy replaces the two logical operators denoting the input relationship with two physical operators which either represent formerly build and broadcasted CSRs of the graph or built fresh CSRs if the edge relationship is queried for the first time. For each partition of the empty RDD, we execute the WCOJ backing its `TrieIterator` with the broadcasted CSRs while partitioning the data logically using one of the schemes presented in Section 5.

The physical operator to materialize an edge relationship into one CSR per edge direction builds both data structures from the two row-wise iterators over the edge relationship sorted by *(src, dst)* and *(dst, src)*, respectively. We build the CSRs in tandem because some vertices might have no outgoing or incoming edges which would lead to an incompatible CSRs if they were built separately. We do so by zipping the row-wise input iterators and aligning them on their first attribute. Scanning and sorting the input relationships as well as broadcasting the result is supported by existing Spark operators.



(a) triangle (b) 4-clique (c) 5-clique (d) 4-cycle

(e) 5-cycle (f) Diamond (g) Kite

Figure 3: An overview of the graph query patterns used in our experiments.

We only compute these CSRs once when an edge relationship is queried for the first time and we cache them after. The caching is backed by *broadcast variables* which allow data reuse over multiple tasks in Spark and guarantee that the data is shipped at most once to each worker. Whenever we translate a *logical WCOJ* plan into a physical one, we check if the CSRs for the edge relationship has already been broadcasted and can be reused.

5 LOAD-BALANCING STRATEGY

Our design decision to replicate the graph on all workers requires us to partition the work into multiple Spark tasks. First, experiments with various static partitioning schemes show that it is very challenging to find a scalable trade-off between duplicated work and skew-resilience on the skewed degree distribution of real-world graphs (see appendix A). Therefore, we design a dynamic work-stealing approach to actively fight skew while replicating no work. We start by describing a work-stealing scheme for a single-machine execution (*local mode*) which we further extend to a cluster execution (*distributed mode*).

Local Mode. Our work-stealing strategy aims to organize the work in a large number of small tasks which can be executed by any worker in the system. Thus, we place the tasks in a queue that is accessible to all workers to poll tasks whenever they are idle. Therefore, tasks are naturally load-balanced across workers, while the maximum skew is roughly the size of the smallest task. To avoid duplicated work, we define a task as the work required to find all possible tuples for a single binding of the first join variable. In local mode, the Spark master and workers run in the same JVM process. So, a simple thread-safe queue can be used to implement the scheme. Tasks are assigned in batches to avoid contention. Integration within the LFTJ algorithm is done by a LeapfrogJoin component which only returns first attribute values that have been polled from the queue.

This way of work-stealing has two main drawbacks which we discuss in turn. Firstly, our work-stealing scheme leads to an unpredictable partitioning of the results: we cannot guarantee the bindings per partitions nor the partition sizes. However, the user can repartition the results afterwards. Secondly, our implementation is not fault-tolerant. This can be overcome either by restarting the computation upon a single task failure or by making the master aware of the set of values assigned to each worker and re-add them to the queue when failures occur.

Distributed Mode. In this section, we describe a simple, yet effective way of integrating work-stealing in Spark running in distributed mode. The main challenge is that Spark workers cannot communication with each other. Therefore, we seek a communication-free approach: we statically partition the total amount of work across multiple workers and enable work sharing only between threads on the same machine. We implement this using *barrier scheduling* execution mode for the worst-case optimal join operator. In this mode, all tasks of a stage are scheduled simultaneously with access to the location of other tasks for this stage. Hence, we can employ the local mode as explained above by partitioning the task queue in a round-robin fashion.

This approach is limited to mitigate skew at the level of individual workers, skew between workers may still exist. Furthermore, the *barrier mode* requires enough free resources in the cluster to schedule all tasks at once which may be difficult in a cluster used by multiple users.

6 EXPERIMENTAL RESULTS

In this section, we introduce our experimental setup. Then, we compare the performance of the GraphWCOJ with vanilla Spark and with different flavors of the LFTJ algorithm (Section 6.1). Finally, we assess the scaling properties of GraphWCOJ both in a single worker and multi-worker distributed setting (Section 6.2).

We run our experiments on the SciLens cluster of the CWI Database Architecture research group [19]. Each cluster machine is equipped with 4 Intel Xeon E5-4657Lv2 processors with a total of 48 cores. Whereas each core has 32 KB on the first two levels of cache, the third cache level provides 30 MB which is shared across the 12 physical cores. The main memory consists of 1 TB of RAM DDR-3 memory. The machines run a Fedora version 30 Linux system. In our experiments, we use a standard deployment of Spark 2.4.0 with Scala 2.11.12 on Java OpenJDK 1.8.

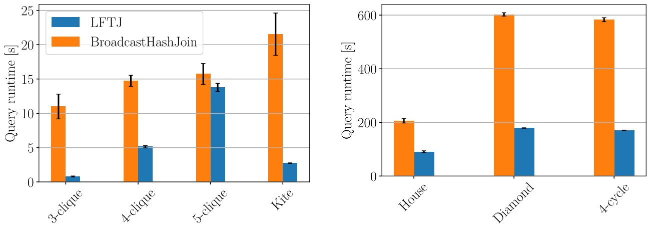


Figure 4: The query runtime of the basic LFTJ algorithm versus vanilla Spark.

Figure 3 depicts the graph patterns used in our experiments. We extend the clique queries with inequality filters, while for the others we require distinct bindings. We use the SNB [25], LiveJournal, and Orkut [26] datasets which represent social network friendship graphs. We generated SNB with default parameters and the scale factor set to 1, keeping only the friendship relationship in the SNB benchmark, thus obtaining 10k vertices and 453k edges. The LiveJournal dataset consists of roughly 5 million vertices and 69 million edges. Similarly, Orkut has roughly 3 million vertices and 117 million edges.

We report the run-time of the algorithms without setup costs, e.g. sorting of the input relationships or creation and broadcasting of the CSR data structure. This is because the system is designed to do this once at start up and reuse the built graph for all queries. After, start up no further sorting or input preparation is necessary for any query.

6.1 Performance of GraphWCOJ

In this section, we compare the performance of our LFTJ algorithm relatively to vanilla Spark on a single-worker cluster using one thread and no parallelism. Furthermore, we assess the performance of the graph-specialized GraphWCOJ algorithm relatively to the WCOJ+CSR implementation, having the original LFTJ algorithm as a baseline. We use count queries for different patterns as shown in Figure 3 on the SNB dataset.

Spark employs broadcast hash join operators to execute the queries. The broadcast hash join avoids shuffling and so, it is the fastest join implementation available in Spark when run on a single worker. Moreover, Spark also generates Java byte-code for its SQL operators [4, 29] which our LFTJ implementation doesn't support yet. Despite the code generation optimization in Spark, Figure 4 shows that our generic LFTJ outperforms vanilla Spark on all query patterns. Whereas LFTJ performs at its best on the 3-clique with a speedups of 13.9, for a 5-clique query it obtains a relatively low speedup of 1.1. The reason of this result is that an n-clique query typically has a large number of semi-joins which decrease the size of intermediate results, and so Spark can execute them relatively fast as well. Indeed, we find that

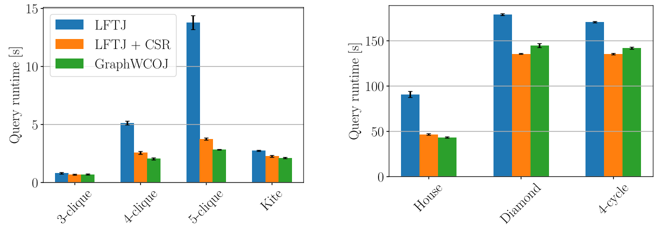


Figure 5: The query runtime with LFTJ+CSR and GraphWCOJ versus the LFTJ baseline.

for the 5-clique query only 3 out of 9 joins in Spark's plan lead to larger intermediate results.

Figure 5 depicts the comparison of the different WCOJ implementations. We first analyze the performance gain from using the CSR data structure in the LFTJ+CSR algorithm over vanilla LFTJ. We find that all queries run faster with the LFTJ+CSR algorithm when compared to the basic LFTJ baseline. In particular, we observe that the higher the clique size, the larger speedup of the LFTJ+CSR is. In contrast, LFTJ+CSR provides lower improvements on the house and kite query patterns . Whereas the original LFTJ implementation uses a column-based implementation, CSR enables searches on the first level of the TrieIterators as a two-array reads instead of a binary search. Therefore, the denser the query, the more first-level TrieIterator accesses are needed. For example, the 5-clique query employs 10 TrieIterators with 10 first-levels to iterate on, while the 4-cycle and diamond patterns only need 4 TrieIterators.

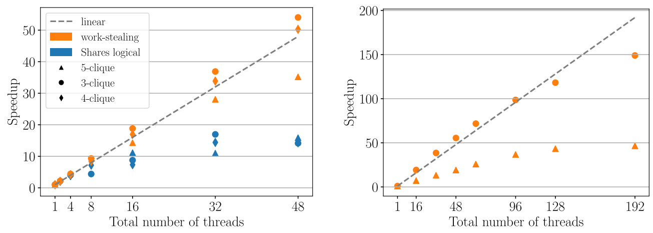
Next, we assess the impact of the graph-specialized intersection algorithm which we incorporated in our GraphWCOJ implementation. Overall, we find that it provides a relatively low improvement over the LFTJ+CSR algorithm. While this technique achieves the largest speedup on dense queries such as the 5-clique, it slows down sparse cyclic queries by a small margin. In Table 2 we show the number of upper bound searches with and without the specialized intersection algorithm enabled. We observe that the improvement on dense queries can be explained by the larger number of adjacency lists that are intersected. In contrast, the diamond and the 4-cycle queries intersect at most two such lists. The slow-down is incurred by our implementation which builds the complete intersection at once by copying the result to a new array, this fails to pay off for sparser queries and could be improved with little engineering effort.

6.2 GraphWCOJ Scalability

In this section we assess the vertical scalability of the GraphWCOJ. In particular, we show the benefit of the work-stealing scheme relatively to the best static partitioning scheme Shares (details see Appendix A) and we assess the impact of the query size on the GraphWCOJ performance. We run GraphWCOJ with work-stealing on a single machine with one executor with up to 48 threads. Furthermore, we consider a distributed setting with 4 machines having a total of 192 threads. In these experiments we employ the 3-clique and 5-clique queries on the Orkut dataset.

Query	LFTJ	GraphWCOJ	Difference [%]
3-clique	34.739.080	33.526.024	3.4
4-clique	118.451.741	99.402.372	16.1
5-clique	262.304.687	192.296.784	26.7
Kite	346.636.041	272.840.747	21.2
4-cycle	4.591.408.924	4.402.790.869	4.1
Diamond	10.230.067.028	9.680.437.365	5.3
House	5985.294.145	5.550.487.243	7.2

Table 2: Difference in upper bound searches for LFTJ versus GraphWCOJ due to the specialized intersection algorithm.



(a) Single worker

(b) Four workers

Figure 6: The speedup achieved by the GraphWCOJ algorithm on n-clique queries with a single worker (a) and 4 workers (b).

Figure 6a depicts the speedup achieved by each partitioning scheme on a single worker when increasing the number of threads from 1 to 48. We observe that work-stealing outperforms the Shares logical partitioning scheme for all queries irrespective the level of parallelism. When we execute the 4-clique query on 96 threads, our work-stealing scheme achieves a speedup of 42.5 which is significantly higher than the speedup of 25.4 obtained by Shares. Figure 6a also shows the Shares logical partitioning is insensitive to the query size. However, for work-stealing, we notice that increasing the query size leads to worse scaling behavior. The reason for this result is that in our work-stealing scheme we set the task size to a single first variable binding. Having tasks which increase with the query size can lead to significant skew. In the single worker setup, we measure the skew as the ratio between the average and the slowest tasks. While the skew for the 3-clique query is below 3%, the skew caused by the 4-clique and 5-clique increases with the number of total threads in the cluster and can be up to 16.3% and 32.3%, respectively. As a consequence of the increasingly large skew with the number of utilized threads, we observe the flattening of the speedup curves for those queries. An extension of work-stealing on any variable of the join is described in the thesis accompanying this paper [17].

Figure 6b shows the performance of GraphWCOJ in a distributed setting with 4 workers. We find that the 3-clique query scales significantly better in the distributed setting than on a single worker. However, the 5-clique scales less well in a distributed setting. For the distributed setting, we measure the skew as the ratio between the wall-clock execution time of the average and slowest executor. The skew of the 3-clique query is negligible in the distributed setting. Furthermore, for the 5-clique query the skew decreases from 62.7% to 37.7% when the number of threads increases from 16 to 192.

Finally, we compare the execution of the 3-clique query on the LiveJournal dataset with Spark and GraphWCOJ. We find that our GraphWCOJ is roughly 100 times as fast as Spark when using a distributed setting with 4 workers. Spark has relatively poor scalability as it takes 424 s and 467.2 s to execute the query on 192 and 384 threads, respectively. In contrast, GraphWCOJ requires only 4.3 s and 3.5 s when running on 192 and 384 threads, respectively. As such, we believe our EdgeFrame offers significant performance advantages to systems and APIs that perform graph pattern matching on Spark.

7 RELATED WORK

In this section, we report on alternative implementations of distributed WCOJs. Prior work proposes a distributed worst-case optimal join based on three algorithms which incorporate a Generic Join [33] in the Timely Dataflow system [5]. These algorithms run in multiple rounds, each of which is binding a single variable in the join query and uses the prefixes from the previous round as input. This requires finding the relationship that provides the smallest set of possible bindings for the current variable for each prefix. The join is executed in multiple batches of work to avoid high memory usage when storing all prefixes. Overall, this mechanism requires $2R \times V$ steps of communication for a query with R relations and V variables. We argue that this approach is not suitable for Spark because it requires a modality of incremental query execution that is not well-supported in Spark since each communication step triggers a shuffle operation which is relatively expensive in Spark (which e.g. materializes all shuffle output locally, rather than streaming it). The straightforward approach based on binary joins requires only $R - 1$ shuffle operations.

GraphWCOJ is computational- and communication-optimal with respect to the AGM bound, thus providing the same theoretical guarantees as the Timely Dataflow system. Whereas Timely Dataflow only partially attempts to balance work, GraphWCOJ provides a dynamic load-balancing mechanism through work-stealing. In addition, Timely Dataflow can handle graphs which exceed the memory of a single machine by distributing the graph edge relationship across all workers.

L. Lai et al. surveyed different strategies to distribute graph pattern matching in Timely Data Flow [24]. They include *BigJoin*, Shares, fully replicating the graph and compare to binary joins as a baseline. Additionally, they study the effect of triangle indexing and compression of the intermediary join results. They empirically find that fully replicating the graph is the best option because it beats all other approaches in total run-time, has the lowest memory footprint and shows the best scaling behavior. Our analysis using the Poisson binomial distribution complements this observation with analytical backing. Like us, they find that Shares does not scale for graph pattern matching and is overall the weakest strategy. Their implementation of a fully replicated approach differs from ours. First, the local algorithm they use is DualSim [22] which has been developed for an out-of-core memory setting and is not known to be worst-case optimal. Second, they logically partition the work by round-robin partitioning on the second variable. As we report in appendix A.1, this scheme does not offer good skew resilience compared to work-stealing and is likely to hinder their scalability in terms of number workers used as well as query size.

8 CONCLUSIONS

Providing a fast and scalable Worst-Case Optimal Join (WCOJ) is an attractive yet challenging target for data processing frameworks when matching complex and cyclical patterns in large graphs. Towards this end, we have presented EdgeFrame, a graph-specialized Spark DataFrame that caches the edges of a graph in compressed form on all cluster nodes, thus avoiding expensive shuffle operations. The design of EdgeFrame is based on our analysis of the state-of-the-art Shares algorithm for distributed WCOJ, which we show to degenerate into a full broadcast of all data already for moderately complex graph patterns. In order to store graph data compactly in compressed form, EdgeFrame is based on the well-known CSR representation. We show that the CSR representation maps naturally on the conceptual Trie data structure used by the Leapfrog TrieJoin, a well-studied WCOJ algorithm. EdgeFrame enables GraphWCOJ, our graph-specialized Leapfrog TrieJoin to provide a fast intersection algorithm and achieve a speedup of up to 11 relative to a non-graph-specific Leapfrog TrieJoin implementation. We also propose a work-stealing mechanism for efficient parallel task processing that avoids duplicate work and mitigates skew.

REFERENCES

[1] [n.d.]. RelationalAI. https://www.relational.ai.

[2] Foto N Afrati, Nikos Stasinopoulos, Jeffrey D Ullman, and Angelos Vassilakopoulos. 2018. SharesSkew: An Algorithm to Handle Skew for Joins in MapReduce. *Elsevier Information Systems* 77 (2018), 129–150.

[3] Foto N Afrati and Jeffrey D Ullman. 2011. Optimizing Multiway Joins in a Map-Reduce Environment. *IEEE TKDE* 23, 9 (2011), 1282–1298.

[4] Sameer Agarwal, Davies Liu, and Reynold Xin. 2016. *Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop, Deep Dive into the New Tungsten Execution Engine.* https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html

[5] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *VLDB Endowment* 11, 6 (2018), 691–704.

[6] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, et al. 2018. G-CORE: A Core for Future Graph Query Languages. *ACM SIGMOD* (2018).

[7] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–40.

[8] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and Implementation of the LogicBlox System. *ACM SIGMOD* (2015).

[9] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark SQL: Relational data Processing in Spark. *ACM SIGMOD* (2015).

[10] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size Bounds and Query Plans for Relational Joins. *IEEE FOCS* (2008).

[11] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in Parallel Query Processing. *ACM Symp. on Principles of Database Systems* (2014).

[12] Arezo Bodaghi and Babak Teimourpour. 2018. Automobile Insurance Fraud Detection Using Social Network Analysis.

[13] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. *ACM SIGMOD* (2015).

[14] Ankur Dave, Joseph Bradley, Tim Hunter, and Xiangrui Meng. 2016. *GraphFrame.* https://databricks.com/blog/2016/03/03/introducing-graphframes.html

[15] Gary William Flake, Steve Lawrence, C Lee Giles, and Frans M Coetzee. 2002. Self-Organization and Identification of Web Communities. *IEEE Computer* 3 (2002), 66–71.

[16] Per Fuchs. [n.d.]. EdgeFrames. https://github.com/cwida/edge-frames.

[17] Per Fuchs. 2019. Fast, scalable worst-case optimal joins for graph-pattern matching on in-memory graphs in Spark.

[18] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. *USENIX ODSI* (2014).

[19] CWI Research Group. [n.d.]. Scilens Cluster. https://projects.cwi.nl/scilens/.

[20] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *VLDB Endowment* 7, 13 (2014), 1379–1380.

[21] Tim Hegeman and Alexandru Iosup. 2018. Survey of Graph Analysis Applications. *arXiv:1807.00382* (2018).

[22] Hyeonji Kim, Juneyoung Lee, Sourav S Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath HA Jarrah. 2016. DUAL-SIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *SIGMOD*. ACM, 1231–1245.

[23] Paraschos Koutris, Paul Beame, and Dan Suciu. 2016. Worst-case Optimal Algorithms for Parallel Query Processing. *Int'l Conf. on Database Theory (ICDT)* (2016).

[24] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. Distributed Subgraph Matching on Timely Dataflow. *VLDB Endowment* 12, 10 (2019), 1099–1112.

[25] LDBC. 2017. *LDBC SNB Documentation.* https://github.com/ldbc/ldbc_snb_docs

[26] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[27] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. *ACM SIGMOD* (2010), 135–146.

[28] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *VLDB Endowment* 12, 11 (2019), 1692–1704.

[29] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *VLDB* (2011).

[30] Mark EJ Newman. 2004. Detecting Community Structure in Networks. *Springer European Physical Journal B* 38, 2 (2004), 321–330.

[31] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-Case Optimal Join Algorithms. *ACM Symposium on Principles of Database Systems* (2012).

[32] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2013. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *arXiv preprint arXiv:1310.3314* (2013).

[33] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *ACM SIGMOD* (2014).

[34] Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q Ngo, Christopher Ré, and Atri Rudra. 2015. Join Processing for Graph Patterns: An Old Dog with new Tricks. *ACM GRADES Workshop* (2015).

[35] openCypher Project. 2016. *CAPS: Cypher for Apache Spark.* https://github.com/opencypher/cypher-for-apache-spark

[36] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *VLDB Endowment* 11, 12 (2018), 1876–1888.

[37] Gorka Sadowski and Philip Rathle. 2014. Fraud Detection: Discovering Connections with Graph Databases. *White Paper-Neo Technology-Graphs are Everywhere* (2014).

[38] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2019. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey. *VLDB Journal* (2019), 1–24.

[39] Christian Schroeder dewitt. 2012. *Leapfrog Triejoin Implementation for 'Database Systems and Implementation' at Oxford University.* https://github.com/schroeder-dewitt/leapfrog-triejoin

[40] Todd L Veldhuizen. 2012. Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm. *arXiv preprint arXiv:1210.0481* (2012).

A STATIC PARTITIONING SCHEMES

In this section, we motivate why we decided dynamic work-stealing was needed, by describing our earlier attempts to parallelize GraphWCOJ using two static work partitioning approaches which failed to scale properly.

Baseline. Our baseline static partitioning is a single-variable partitioning scheme in which we partition the value of a single variable into as many ranges as the desired level of parallelism. The domain of each variable is equal to V, the number of vertices in the graph. Therefore, if we partition by the first variable, the first worker processes only the first V/w values for bindings of the first variable, where w is the number of workers.

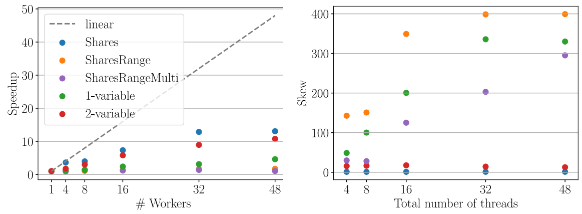
Logical Shares. We also extend the main idea behind Shares to a *logical* partitioning scheme. Instead of physically partitioning the graph before computing the join, we determine if a tuple should be considered by the task using the Shares logic. We do so by organizing the cluster workers in a hypercube in which we assign a coordinate to each worker. Similar to Shares, each worker processes only those tuples that match its coordinate.

Integrating Shares and LFTJ comes with two important design decisions. Firstly, we need to compute the optimal hypercube configuration, which we solve by employing the exhaustive search algorithm proposed by Myria [13]. Secondly, the LFTJ operates on a complete copy of the edge relationship and so, we need to filter out the values that do not match the coordinate of the worker. Using a hash-based filter directly on the `TrieIterators` is relatively expensive because it can only filter one value at a time, thus causing repeated searches. Instead, we filter the values after building the intersection by applying the filter over the set of `LeapfrogJoins`.

We improve the physical Shares scheme by using the same CSR data structure for all `TrieIterators`. As a consequence, we avoid materializing a prefiltered data structure for each `TrieIterator` which reduces both time and memory if the partitions become large for long-running queries.

Range Shares. Our first Shares implementation filters out values after building the intersections. In this section, we explain how to push it into the `TrieIterators` by applying relatively inexpensive rang e based filters instead of hashing.

The main idea behind Shares is to map each attribute from the value space to the hypercube coordinate space through hashing. However, this can also be achieved by using ranges so that a value matches a coordinate if it is in the corresponding range. For instance, in a three-dimensional hypercube with three workers per dimension, we can divide the value space into three ranges.



(a) Speedup

(b) Skew

Figure 7: Contrasting the speedup and the skew achieved by the static partitioning schemes using a 3-clique algorithm on the LiveJournal dataset.

Intuitively, we expect this way of mapping to scale better than a hash-based Shares because it requires less intersection work. However, it turns out that the same range can be assigned multiple times to the same worker for different attributes. As Figure 7b shows, this may result in significant skew, thus degrading the performance as compared to the standard hash-based Shares. To mitigate this issue, we divide the vertex identifiers into more ranges than the number of workers in the hypercube dimension corresponding to the attributes. We assign multiple ranges to each `TrieIterator` in such a way that the overlap of the first two attributes equals the overlap of the hash-based implementation. We further assign the later attributes randomly so that all combinations are covered. Nevertheless, we find that the search pattern instances are still unevenly distributed over the ranges of vertex identifiers. In particular, for the triangle query on the `LiveJournal` dataset, we find that the ratio between the number of triangles generated by the fastest and slowest worker is only 40%. Therefore, our experiments in Section A.1 show that the standard hash-based Shares outperforms the range-based version.

A.1 Experiments

In this section, we analyze the scalability and skew introduced by the static partitioning scheme introduced above when running a 3-clique query on the `LiveJournal` dataset. To assess scalability we run the query with 1 to 48 threads. We define skew of a scheme as the ratio between the time it takes to compute the largest and smaller partitions.

Figures 7a and 7b show the speedup and skew of the 3-clique query when increasing the number of threads. We find that all schemes scale sublinearly with the number of threads and that logical Shares outperforms the other schemes. Interestingly, partitioning on the second variable is slightly worse than Shares, while the other two schemes perform much worse. The reason for this result is that both logical Shares and partitioning on the second variable introduce the lowest skew, as we see in Figure 7b.

We observe that there is a strong correlation between the speedup and the skew depicted in Figures 7a and 7b, respectively. In all partitioning schemes except the *SharesRangeMulti*, the amount of skew relates directly to the scaling behavior. *SharesRangeMulti* is more skew resilient than both first variable partitioning and *SharesRange*, but fails to scale better. The reason for this result is that the first variable partitioning avoids replicating any work and so, scaling is better even with higher skew between workers.

Overall, static partitioning schemes have relatively poor scaling properties. Even partitioning schemes such as partitioning on the first variable which avoid replicating work are outperformed by logical Shares which is the best static policy in mitigating skew.