Fast, scalable WCOJ graph-pattern matching on in-memory graphs in Spark

Master thesis draft iteration 2.0

Per Fuchs

April 2019

Abstract

Graph pattern matching with their vast number of cyclic foreign-key joins is a new challenge for data processing systems, like Spark, because their intermediary results grow over linear with regards to the inputs and are materialized by the traditionally used binary joins. Worst-case optimal join algorithms, WCOJ's, are a natural match to tackle this challenge because they do not materialize the aforementioned large intermediary results. We investigate two major open questions regarding WCOJ's. First, we develop a WCOJ specialized to graph-pattern matching, namely self-joins on a relationship with two attributes, and compare its performance with a general WCOJ. Second, we propose a novel method to distribute WCOJ's. We show in our proposal that current methods to distribute WCOJ's, suitable for Spark, do not scale well to bigger graph pattern (five vertices and more). Based on this result we propose to keep the edge relationship cached on all workers but distribute the computation using a logical partitioning. Along the line of argumentation in COST [38], we aim to provide a fast WCOJ implementation in Spark which scales well in use of computational resources by trading off scalability in memory usage. This is a reasonable trade-off as most graphs today fit in main memory [36]. Our thesis provides the first distributed, open-source implementation of a WCOJ in a data processing system widely used in industry, namely Spark.

Acknowledgements

The Spark integration of this work has been inspired by the IndexedDataframes [**indexed-dataframes**] work of Alex Uta and the supervisors of this thesis.

Contents

1 Introduction

Newly developed worst-case optimal join (WCOJ) algorithms, e.g. Leapfrog Triejoin, turned conventional thinking about join processing on its head because these multi-join algorithms have provably lower complexity than classical binary joins, i.e. join algorithms that join just two tables at-a-time. In the areas of data warehousing and OLAP, this finding does not have much impact, though, since the join patterns most commonly encountered are primary-foreign-key joins, which normally take the form of a tree or snowflake and contain no cycles. The computational complexity of FK-PK joins is by definition linear in size of the inputs. In these *conventional* cases, binary joins work fine, e.g. hash joins.

However, analytical graph queries often use foreign-foreign-key joins which can grow over linearly in the size of their inputs, and often contain cycles. For these use-cases, binary joins often exhibit highly suboptimal run-times because they generate a rapidly increasing set of intermediary results, e.g. when navigating a social graph with an out-degree in the hundreds. Many of these intermediary results are eliminated in later joins, e.g. a join that closes a cycle. Hence, an algorithm which avoids generating these results in the first case is to perform much better and closer to the optimal possible performance given by the output size. Worst-case optimal joins many of the intermediary results and are guaranteed to reach the best possible run time in terms of the output size.

WCOJ's avoid large result materialization and hence promise to be orders of magnitude faster than binary joins. Therefore, we believe that worst-case optimal join algorithms could be a useful addition to (analytical) graph database systems. We aim to integrate a scalable, WCOJ algorithm in Spark which is used by some modern graph engines [45, 8, 21]. Due to time constraints, we built our system for Spark local mode only. This mode allows to run queries in parallel using all cores of one machine but does not allow to distribute queries over multiple machines. We address the extensions to Spark's cluster modes in future work (section 9.1).

The rest of our introduction is structures as follows. Section 1.1 defines the term graph pattern matching, its translation into datalog and relational queries and two examples of cyclic graph pattern used in practice. We aim to give the reader an intuitive understanding of why WCOJ's are superior to binary joins for graph pattern matching in section 1.2. Section 1.3 motivates our choice to use Spark as the base for our thesis. Next, we state our research questions and contributions in section 1.4. Then in section 1.5, we outline the main ideas behind the thesis and their connections. The most important results are summarized in section 1.6. Finally, we outline the structure of the whole thesis.

1.1 Graph pattern matching

Graph pattern matching is the problem of finding all instances of a specific subgraph in a graph. The subgraph to find is described as a pattern or query. In this thesis, we use datalog queries to define subgraph queries.

For example, eq. (1) shows the datalog query describing a triangle.

$$triangle(a,\ b,\ c) \leftarrow R(a,\ b),\ S(b,\ c),\ T(c,\ a) \tag{1}$$

Here we join three atoms S, R and T with two attributes each (a, b), (b, c) and (a, c) respectively. The task of enumerating all triangles within the three atoms can be also be described as finding all possible bindings for the join variables a, b and c within them.

The translation from datalog queries to graph patterns is straightforward. An attribute or a variable refers to a vertice in a graph and an atom to an edge. A depiction of the subgraph

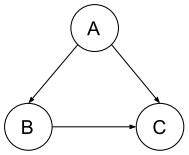


Figure 1: Depiction of the triangle subgraph query.

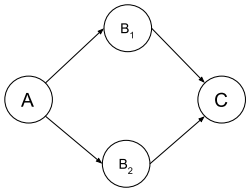


Figure 2: The dimaond query is used by Twitter. The vertices are users and the edges follower relationships. In the example, they could recommend A to follow D because A follows B_1 and B_2 which both follow C.

pattern described by eq. (1) is shown in fig. 1.

In relational terms, a graph pattern matching query is an n-ary, conjunctive, self-equijoin on the edge relationship of the graph. In this thesis, all join queries discussed belong to this subcategory of possible join queries. Other join queries can be useful to describe more complex graph patterns, e.g. disjunction for two edges of which only one needs to exist or negation to exclude instances that have too many connections. Some techniques used in this work can be extended to cover these cases, we mention related literature but do not focus our efforts on these extensions.

Graph pattern matching is fundamental to analytical graph analysis workloads [**twitter-diamonds**, 46, 24, 14, 41]. We show two graph patterns which are used in practice below and explain the use-cases.

Figure 2 shows the diamond query which is used by Twitter to recommend their users new people to follow. The idea is that if a user a is following multiple accounts $c_1, \ldots, c_k$ who all follow a person b then it is likely that b would be interesting to follow for a as well. In the figure, we see the diamond query for $k = 2$. This is the diamond query as discussed in most papers in academia [44, 18, 39], although, Twitter uses $k = 3$ in production [26].

Our second concrete use-case example is the n-cycle. As explained in [46], cycles can be used to detect bank fraud. A typical bank-fraud often involves so-called *fraud-rings*. These are two or more people who combine their legitimate contact information in new ways to craft multiple false identities. For example, two people share real phone numbers and addresses to craft four fake identities; all combinations possible with two pieces of information. They open accounts under wrong names with real contact information, use these accounts normally to build trust with the bank and build up bigger credit lines. At a certain date, they max out all credit lines and disappear. The phone numbers are dropped and the actual people living at the addresses deny ever knowing the identities that opened the accounts.

This scheme can be detected using graph pattern matching. Let us assume, we have a graph database in which customer of the bank, their addresses and phone numbers are all vertices and the relationship of an address or phone number belonging to a customer are edges. Then, the

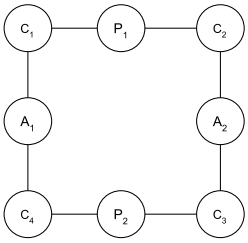


Figure 3: Schematics of a bank fraud ring. 2 fraudsters share their phone numbers and addresses (labelled P and A) to create four fake customers (C vertices).

case described above forms an 8-cycle of 4 persons (fake identities) connected by the shared use of phone numbers and addresses. The imagined cycle is shown in fig. 3.

1.2 Binary joins vs WCOJ's: an intuitive example

We introduce the triangle query and possible binary join plans. Then we point out the general problem of binary join plans on this query and the idea of how WCOJ's can improve the situation. Next, we give a concrete example of a database instance to illustrate the aforementioned problem. We conclude our motivation to use worst-case optimal joins by reporting multiple papers that show that these joins are highly beneficial to graph pattern matching queries in practice.

The simplest example of a cyclical join query enumerates all triangles in a graph. It is shown in eq. (1) and fig. 1.

Traditionally, this would be processed by using multiple binary joins:

$$R \bowtie S \bowtie T \tag{2}$$

The join above can be solved in 3 different orders: $(R \bowtie S) \bowtie T$, $(R \bowtie T) \bowtie S$ and $R \bowtie (T \bowtie S)$. Independent of the chosen order, database instances exist where the intermediary result size is in $\mathcal{O}(N^2)$ with $N = |R| = |S| = |T|$. However, it is provable that the output of this query is guaranteed to be in $\mathcal{O}(n^{3/2})$ [12, 43] for any database instance. Hence, binary joins materialize huge intermediary results after processing parts of the query, which are much bigger than the final result.

The described problem is a fundamental issue with traditional binary join plans [12, 43]. We call these plans also *join-at-a-time* approach because they process whole joins at the time.

Fortunately, worst-case optimal join algorithms can materialize cyclic joins with memory usage linear to their output size by solving the join *variable-at-a-time* which avoids materializing big intermediary results [51, 42].

In a variable-at-a-time the algorithm finds a binding for the first variable a, then one for b and finally one for c. After this, they emit the tuple as part of the output. Then they find further bindings via backtracking until they enumerated the whole join when all bindings for a have been explored.

A simple example graph database instance gives an idea of why a variable-at-a-time approach is beneficial for cyclic queries. In fig. 4, we see an edge relationship. It is repeated three times labelled with different attributes to ease the understanding of the following explanation; however, in a system's implementation, only one table exists and is used by all joins as input.

A binary join plan which joins R and S via b first produces $16 + 3$ intermediary results; 4 times 4 results for $b = 2$ and one for 6, 11, 12 each. The next join reduces these 16 results to the three triangle instances; all permutations of the set $\{6, 11, 12\}$.

a	b
1	2
2	7
2	8
2	9
2	10
3	2
4	2
5	2
6	11
11	12
12	6

b	c
1	2
2	7
2	8
2	9
2	10
3	2
4	2
5	2
6	11
11	12
12	6

c	a
1	2
2	7
2	8
2	9
2	10
3	2
4	2
5	2
6	11
11	12
12	6

Figure 4: Three aliases to an edge relationship which contains three triangles, the permutations of {6, 11, 12}, and one skewed value.

A variable-at-a-time approach finds 4 bindings for a, namely $2, 6, 11, 12$; the intersections of both columns labelled a.

Intersecting both columns of b values we notice $2, 6, 11, 12$ could be possible bindings for b. When we fix an a value these four possibilities are reduced to the b values which exist for this a value in the leftmost table. So once we fixed a binding for a, we find one possible binding for b each; except the binding $a = 2$ for which we cannot find a matching b value.

Finally, we find all three instances of the triangle by completing the three a, b bindings with the matching c binding; only one exists for each a, b binding.

We can drastically reduce the workload by formulating the join as a problem of finding variable bindings using information from all parts of the join, instead of, using only one constraint at the time and building it join-by-join.

We do not claim that the example above illustrates the generality of why binary join plans are provably worse than WCOJ's. Clearly, the example does not show an intermediary result of N^2 as $N = 11$ and the intermediary result has the size of 16. However, we note that even in such a simple example all possible binary join orders produce an intermediary result of size 16. While all possible variable orderings for a variable-at-a-time approach eliminate the skewed value (2) after finding no binding for the second variable. A more general but less concrete example is explained in [43].

In practice, these worst-case optimal join algorithms are highly beneficial for cyclic queries in analytical graph workloads in an optimized, single machine system [51, 44]. [44] compares a system using WCOJ's against multiple general-purpose database systems using binary joins and some graph pattern matching engines on 15 datasets and 7 queries and finds that worst-case optimal joins can beat all other systems in the vast majority of queries and datasets, often by the order of magnitudes or even being the only system to finish within 30 minutes.

Later worst-case optimal joins have been applied successfully to a distributed shared-nothing settings [18, 7]; we describe these systems in more detail in section 2.3 and 8.1.

1.3 Graphs on Spark

Spark is an attractive target for big graph processing, due to its generality, widespread acceptance in the industry, the ability to use cloud hardware and its fault tolerance by design. For example, GraphFrames [21], GraphX [25] (a Pregel [37] implementation) or graph query languages as G-CORE [8] and openCypher with 'Cyper for Apache Spark' or CAPS [45] all aim to ease graph processing on Spark. The last two technologies translate their graph specific operations to the relational interface of Spark (SparkSQL) to profit from Spark's relational query optimizer Catalyst [11]. Moreover, they allow the user to formulate graph pattern matching queries naturally.

Hence, we believe that the WCOJ's, with their efficiency for analytical graph queries, are a valuable addition to Spark's built-in join algorithms in general and these graph-on-spark systems in particular. Ideally, they are integrated such that they can be naturally used in the ecosystem of Catalyst. This would allow easier use in SQL like graph languages as G-CORE or Cypher for graph pattern matching.

1.4 Research questions and contributions

We identify two challenging, novel directions for our research. First, all papers about WCOJ focus on queries widely used in graph pattern matching, e.g. clique finding or path queries. As explained above, graph pattern matching uses only self-joins on a single relationship with two attributes; namely the edge relationship of the graph. However, all systems use worst-case optimal joins developed for general n-ary joins. This raises the question if and how WCOJ's can be specialized for graph pattern matching.

Second, while the communication costs for worst-case optimal joins in MapReduce like systems[1] is well-understood [4, 3, 13, 30], their scalability has not been studied in depth. Given that the only integration in a MapReduce like system exhibits a speedup of 8 on 64 nodes over two workers (an efficiency of 0.125) [18], we find that designing a scalable, distributed WCOJ for a MapReduce like system is an unsolved challenge.

It is time to investigate how these algorithms scale in the probably most widely used, general-purpose big data processing engine: Spark. To the best of our knowledge, this is also the first time a worst-case optimal join is integrated with an industrial-strength cluster computing model. We detail our research questions below.

1. Can we gain performance in WCOJ's by specializing them to graph pattern matching?
   (a) How much performance can we gain by using compressed sparse row representations to back the iterators of WCOJ?
   (b) Can we exploit the fact that most real-world graphs have a low average out-degree in the algorithm to build intersections within the worst-case optimal join?
2. How well do WCOJ's scale in Spark when used for graph pattern matching?
   (a) How well does Shares scale as a logical partitioning scheme given that it replicates work?
   (b) How to integrate scalable work-stealing into Spark to counter tuple replication and skew?

Towards answering our research questions, we make the following contributions.

1. We integrate a sequential, general worst-case optimal join into Spark. This implementation serves as a baseline for our WCOJ optimized to graph pattern matching.

[1] An excellent definition of the term MapReduce like systems is given in [4]

2. We design and implement *GraphWCOJ* which is a worst-case optimal join specialized to graph pattern matching. It is backed by a compressed sparse row representation of the graph which reduces its memory footprint and speeds up execution by TODO over a normal LFTJ because it acts as an index. Furthermore, we exploit the typical low out-degree of most graphs to by specializing the LFTJ for small intersections, gaining further TODO x of execution time.
3. We analyse how many tuples Shares replicates for typical graph pattern matching queries. From this analysis and the fact that Shares is an optimal partitioning scheme, we follow that we should replicate the graph on all workers for parallelization.
4. Based on a replicated edge relationship, we design *logical* Shares. This is a Shares partitioning which is integrated directly into the WCOJ. We measure a speedup of TODO on 48 workers for some queries and beat the aforementioned speedup of Myria by TODO. The results show that Shares is good in dealing with skew but requires too much replicated work to scale well.
5. Therefore, we abandon static *logical* partitioning and apply work-stealing. We show that work-stealing can scale linear on some input queries and beats *logical* Shares for all levels of parallelism for 3-cliques and 5-cliques on three different datasets.
6. We run experiments on TODO datasets, TODO queries, for up to 96 workers in Spark's local mode using Spark's build-in hash join, a general Leapfrog Triejoin and our specialize GraphWCOJ.

1.5 Thesis overview

In this section, we outline the main ideas, motivation and decisions taken in this thesis. We summarize the whole thesis as a graph in fig. 5. We see the background that motivates our decision in the corners and our system in the centre. The edges show connections between different ideas and components. We give an overview of the thesis in the next paragraphs.

As motivated in section 1.3, Spark is a good platform for our work because many graph pattern matching systems use Spark. In particular, they build on top of Spark's structured query execution offered by Catalyst. Catalyst is designed to be easily extendable and allows to introduce new operators, as a worst-case optimal join, without modifying the core of Spark. So that these new operators can be used with a native, unchanged installation of Spark. We describe Catalyst query compilation process in section 2.1.3. Our integration is detailed in section 6.

Normally, Spark achieves parallelism and distributed algorithms by partitioning data overall workers. When data from different tables need to be joined, Spark repartition the data such that each worker can process parts of the join locally. However, shuffling an expensive operations, involving disk writes and reads, which should be avoided if possible. Moreover, we can show that a communication optimal partitioning scheme converges to a full broadcast for bigger graph pattern queries; we explain this in detail in section 2.3.1.

Given this finding, we design our system to build on a replicated and cached edge relationship on each worker. This has a few distinct advantages.

First, the broadcast can be done once at system startup. Then, we can reuse it for any graph pattern matching query; all of them need to join over the edge relationship many times. Therefore, we can answer many graph pattern matching queries without shuffling data. We explain the integration of replicated edge relationships into Spark in section 6.4 and introduce the necessary background in section 2.1.4.

Furthermore, such reuse helps to amortize the none trivial setup costs for worst-case optimal joins; they require their input data to be sorted.

Finally, this allows us to use dynamic work-sharing schemes as work-stealing. Normally, work-sharing is done completely statically in Spark. However, this is problematic given that many

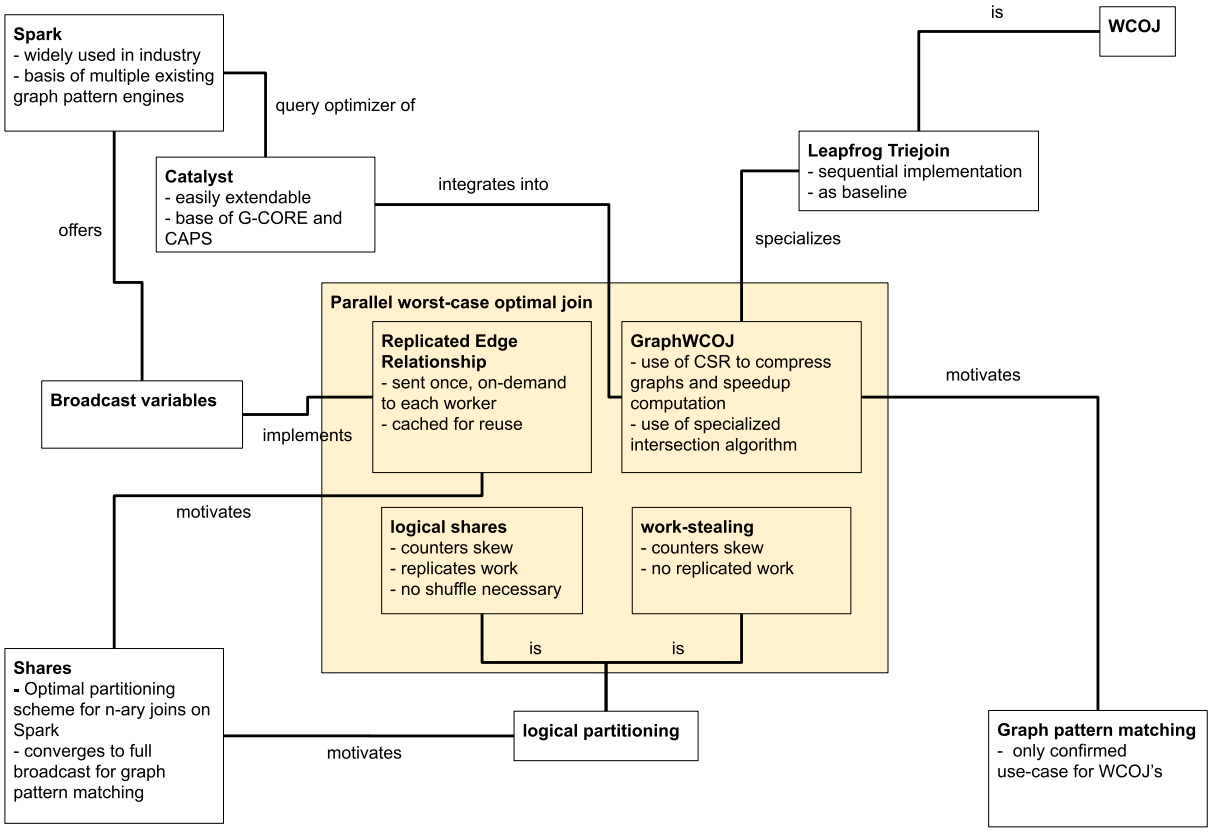


Figure 5: Main ideas and components of the thesis. Background and related work shown in the corners. The center shows the main component of our parallized worst-case optimal join

real-world graphs are highly skewed, e.g. power-law graphs such as many follower graphs (Facebook, Twitter) or web graphs. We explain how to integrate work-stealing with Spark in section 3.4.

Using a fully replicated edge relationship and potentially work-stealing does require us to build partitioning of the data into the worst-case optimal join operators. This is because Spark would normally have an operator work on all local data and archive parallelism via physically partitioning the data over multiple workers. We call partitioning build into our operators *logical* partitioning. The concept and its implementation are described in section 3.

We choose to use the Leapfrog Triejoin [51] as a basis for our system; this choice is motivated in section 2.2. This join requires its input relationships to be presented in a sorted data structure which searchable for upper bounds in $\mathcal{O}(\log_N)$. Furthermore, it mainly uses intersections to compute the join. The algorithm is explained in detail in section 2.2.1.

We specialize the Leapfrog Triejoin to graph pattern matching by introducing compressed sparse row representaton (see section 2.4) as backing data structure for the input relationships. CSR can compress the graph edge relationship by a compression factor of nearly 2. Additionally, we show that it speeds up the WCOJ execution to be backed by a CSR because this representation acts like an index.

Another graph specific optimization we apply to LFTJ is that we change the intersection building algorithm for one that is specialized in small intersections. This is motivated by the fact that real-world graphs have normally small average out degrees. Hence, the intersection of multiple adjacency lists is predictably small. We discuss both specializations to the Leapfrog Triejoin algorithm in section 4.

1.6 Results

TODO summarize the most important results here.

2 Background

TODO introduction

2.1 Spark

Spark is the probably most widely used and industry accepted cluster computing model. It improves over former computing models, e.g. MapReduce [22], Hadoop [9] or Haloop [16], by allowing to cache results in memory between multiple queries, using so-called resilient distributed datasets [53]; often abbreviated to RDD.

This section introduces Spark and is organized in four subsections. Section 2.1.1 describes the core data structure of Spark: the RDD's. In section 2.1.2, we explain the different components and processes in a Spark cluster. The query optimizer of Spark, Catalyst, is explained in section 2.1.3. It is the component we integrate our WCOJ with; therefore, it is the module of Spark that is most relevant to this thesis. Finally, in section 2.1.4 we highlight important details about *Broadcast variables* which are used to implement our parallel worst-case optimal join.

2.1.1 Resilient distributed datasets

RDD's form the core of Spark. However, for this thesis, it is not necessary to understand them in great detail. In the next paragraph, we give a short introduction to the relevant aspects of RDD's. For the interested reader, a more in-depth description is given in the original paper [53].

Resilient distributed datasets describe a distributed collection of data items of a single type. In contrast, to other distributed share memory solutions, RDD's do not use fine-grained operations to manipulate single data items but coarse-grained operations which are applied to all data items, e.g. *map* to apply a function to each data item. These operations are called *transformations*. An RDD is built starting from a persistent data source and multiple transformations to apply to this data source. One can store the transformations applied to the input data source as a directed acyclic graph, the so-called *lineage graph*. This graph fully describes the dataset without materializing it because the transformations are deterministic. Hence, the dataset can be computed and recomputed on demand, e.g. when the user asks for the count of all items in the set. Operations which require that the data in the RDD is computed are called *actions*.

RDD's are distributed by organizing their data items into partitions. The partitioning can be chosen by the user or the Spark query optimizer such that it allows to run transformations on all partitions in parallel. For example, one might choose a round-robin partitioning to generate splits of equal size when reading data items from disk or one groups items by hashing a specific key to support parallelizable aggregation on that key per partition. The process of repartitioning an RDD is called a *shuffle*. It is an expensive operation because it involves writing and reading the whole RDD to disk.

Describing datasets as RDD's comes with two main benefits. First, it is resilient because if the dataset or some partitions of it get lost, it is possible to recompute them from persistent storage using lineage graph information. Second, it allows Spark to compute RDD's in parallel.

Spark can parallelize the computation of RDD in two ways. First, by data-parallelism, since different partitions of an RDD can be computed independently from each other. Second, by task parallelism, because some parts of the DAG can be computed without dependence of the others. Indeed, it is possible to compute all parts of an RDD in parallel which are not related in a topological sort of the graph.

2.1.2 Spark architecture

Spark allows the user to run his program on a single machine or hundreds of machines organized in a cluster. In this section, we explain the architecture that allows this flexibility. Figure 6 shows the schematics of a Spark cluster setup.

In Spark, each physical machine is called a *worker*. On each worker, Spark starts one or multiple Spark processes in their own JVM instance; each of them is called *executor*. Nowadays, many Spark deployments use a single executor per worker[2]. Each executor runs multiple threads (often one per core on its worker) to execute multiple tasks in parallel. In total, a Spark cluster can run *# workers* $\times$ *# executors per worker* $\times$ *# threads per executor* tasks in parallel.

Spark uses two kinds of processes to execute an application: a *driver program* and multiple *executors*. When started, the driver program acquires resources from the *cluster manager* for its executor processes. These executors stay alive during the whole Spark application. Then, the driver program continues executing the Spark application. When it encounters parallelizable tasks, it schedules them on the available executors.

[2]This is the setup Databricks uses; Databricks is the leading maintainer of the Spark platform and offers professional deployment to many customers.

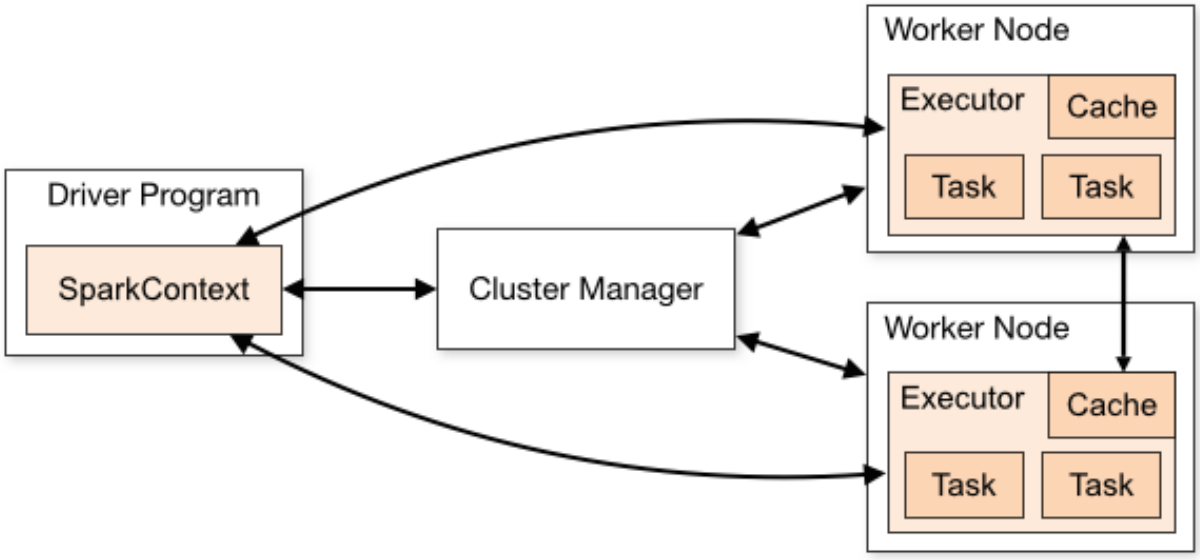


Figure 6: Schematics of a Spark cluster with two workers, each of them with one exectuor and two threads per executor. Source: Apache Spark Documentation, https://spark.apache.org/docs/latest/cluster-overview.html

All tasks scheduled on the same executor share a cache for in-memory data structures like *Broadcast variables* or persisted RDD partitions. This is important in the context of this thesis because it means that we cache the input graph once per executor; which in many Spark deployments is once per worker or physical machine. This would not be possible if different tasks in the same JVM would not share the same cache.

Spark allows the user to choose a cluster manager to manage resources in the cluster. It comes with good integration for Hadoop YARN [50], Apache Mesos [27] and Kubernetes [31], as well as, a standalone mode where Spark provides its own cluster manager functionality. Finally, one can run Spark on a single machine in *local mode*. In local mode, the driver program and a single executor share a single JVM. The executor uses the cores assigned to Spark to run multiple worker threads. For our experiments, we run Spark purely in local mode.

2.1.3 Catalyst

Catalyst [11] is Spark's query optimizer. It can process queries given as a SQL string or described using the DataFrame API. From a given query it constructs an executable *physical plan*. The query compilation process is organized in multiple stages. Its inputs and stages are shown in fig. 7. Below we explain these in order. We use the triangle given by the datalog rule $COUNT(triangle(A, B, C)) \leftarrow R(A, B), S(B, C), T(A, C), A < B < C$ as a running example.

The input of Catalyst is a query in the form of a DataFrame or SQL string. From this the optimizer builds a *unresolved logical plan*. This plan can include unresolved attributes, e.g. attribute names which are not matched to a specific data source yet or which have no known type. To resolve this attributes Catalyst uses a *Catalog* of possible bindings which describe the available data sources. This phase is referred to as *Analysis* and results in a *logical plan*. The logical plan represents *what* should be done for the query but not exactly *how*, e.g. it might contain a Join operator but not a Sort-merge join.

We show the logical plan for the triangle query in fig. 8a. As we see, the query is represented as

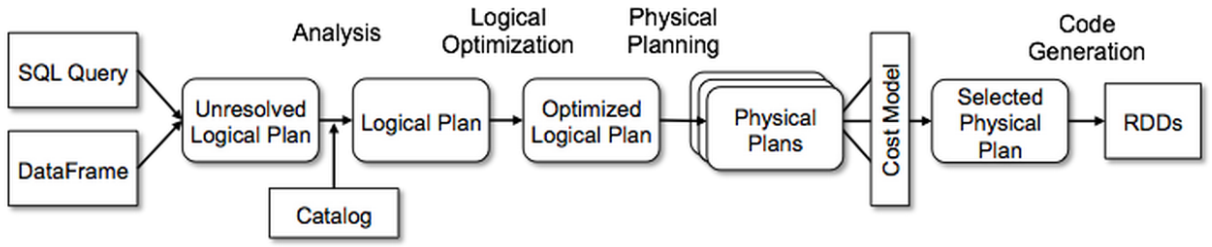
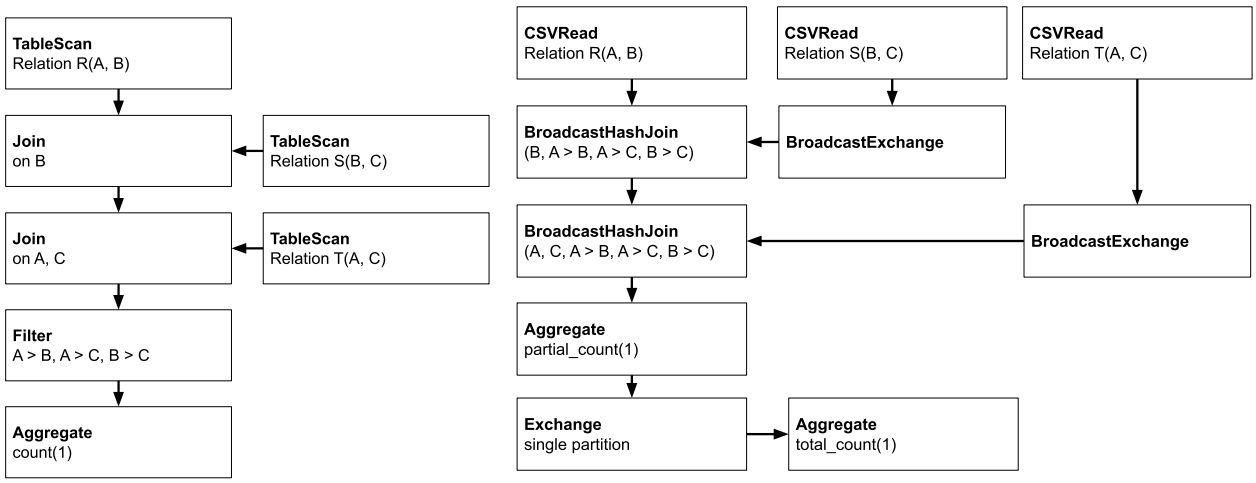


Figure 7: Input and stages of the Catalyst optimizer. Source: Databricks Blog, https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html



(a) Logical plan

(b) Physical Plan

Figure 8: Logical and physical plan for the triangle count query as generated by Catalyst.

tree where the vertices are operators and the edge indicate dataflow from one operator to another. The leaves of the tree are three aliases of the edge relationship. zTwo of these source relationships are the input the join between R and S via B. The result of this join and the leaf relationship T are input to the second join. The tuples produced by this join are filtered to fulfil $A < B < C$. Finally, at the root of the tree, there is an aggregation to count all results and report the sum.

The *logical optimization phase* applies batches of rewriting rules until a fixpoint is reached. A simple example of a logical optimization would be rewriting $2 + 2$ into 4. In the running example of the triangle query, this phase pushes the filters into the two joins. This optimization is called Filter Pushdown. It is efficient because it applies filters earlier within the pipeline reducing the number tuples to process by later operators.

From the *optimized logical plan* the optimizer generates one or multiple *physical plans* by applying so called *Strategies*. They translate a logical operator in one or multiple *physical operators*. *Strategies* are also allowed to return multiple physical plans for a single *logical plan*. In this case, the optimizer selects the best one according to a *cost model*.

The physical plan for the triangle query is shown in fig. 8b. We see multiple examples of translation of a logical operator, which describes what to do, to its physical pendant that also describes how to do it: the *TableScan* becomes a *CSVRead* and the *Joins* are implemented as *BroadcastHashJoins*.

Furthermore, we see the introduction of exchanges. *BroadcastExchanges* precede the *BroadcastHashJoins*. They build a hashtable from their input operators and make them available as a broadcast variable to all executors of the cluster; we explain broadcast variables in depth in section 2.1.4. When an executor is tasked to execute the hash join operator, it acquires the broadcasted hashtable and executes a local hash join of its assigned partitions.

Another exchange operator is introduced for the aggregation. It is broken up into a partial aggregation directly after the last join, an exchange reorganizing all partial counts into a single partition and a second aggregation over that partition to calculate the total count. The last is a good example of Catalyst introducing a shuffle.

To conclude, the translation to a physical plan translates logical operators into concrete implementations of these and adds exchanges to organize the data such that it can be processed independently in partitions.

After generating and choosing a physical plan, Catalyst enters the *code generation* phase in which it compiles Java byte code for some of the physical operators. This code executes often magnitudes faster than interpreted versions of the same operator [11] because it can be specialized towards this particular query, e.g. if a join operates only on integers, code generation can prune all code paths dealing with strings. Indeed, the code generation phase is part of another Spark project called *Tungsten* [52, 5]. In this thesis, we do not build any code generated physical operators. Hence, we do not treat this topic in depth. It is enough to know that all freshly generated Java code is wrapped into a single physical operator. Therefore, it integrates seamlessly with interpreted operators.

Finally, Catalyst arrives at an optimized physical plan which implements the query. The execution of this plan is called *structured query execution* [33]. It translates the plan into RDD operations implemented by Spark's core. Hence, the result of Catalysts query compilation is an RDD representing the query. One should note that structured query execution does not materialize the query: the result is an RDD which is a none materialized representation of the operations necessary to generate the result. In this thesis, we are not concerned with the internals of RDD's. We do not need to introduce any new RDD operations or even touch Spark's core functionality. Thanks to the extensibility of Catalyst, we can integrate worst-case optimal joins by adding one logical operator, multiple physical operators and a Strategy to translate between them.

2.1.4 Broadcast variables

Broadcast variables readonly variables which are accessible by all tasks. They are initialized once by the driver program and should not be changed after initialization. The process of broadcasting them is handled by Spark. It is guaranteed that each broadcast variable is sent only once to each executor and allows it to be spilled to disk if it is not possible to keep the whole value in memory. Furthermore, 'Spark attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication costs' [19]; currently Spark uses a BitTorrent-like communication protocol[3]. Once sent, they are cached once per executor (see also section 2.1.2) and shared by all tasks on this executor. They are cached in deserialized form in memory but can be spilled to disk if they are too big. In this thesis, we use broadcast variables to cache the edge relationship of the graph on all workers.

2.2 Worst-case optimal join algorithm

The development of worst-case optimal joins started in 2008 with the discovery that the output size of a relational query is bound by the fractional edge number of its underlying hypergraph [12].

[3]See Spark sources: `org.apache.spark.broadcast.TorrentBroadcast`

In short, this bound proves that traditional, binary join plans perform asymtopically worse than theoretical possible for the worst-case database instances, e.g. heavely skewed instances. For example, the worst-case runtime of binary joins on the triangle query is in $\mathcal{O}(N^2)$, while the AGM bound shows the possibility to solve it in $\mathcal{O}(N^{3/2})$. The AGM bound has been treated widely in literature [43, 6, 12]. A particular good explanation is given by Hung Ngo et al in [43]. We refer the reader to these papers for further information. In the next paragraph, we discus different algorithms matching the AGM bound which are called worst-case optimal joins.

In 2012, Ngo, Porat, Re and Rudra published the first join algorithm matching the AGM bound, called *NPRR* join [42]. In the same year, Veldhuizen proved that the algorithm Leapfrog Triejoin used in LogicBlox, a database system developed by his company, is also worst-case optimal with regards to the fractional edge number bound. We often abbreviate Leapfrog Triejoin to LFJT. Both algorithms have been shown to be instances of a single algorithm, the *Generic Join*, in 2013 by Ngo et al. [43].

Three worst-case optimal join algorithms are known in literature. We choose Leapfrog Triejoin as the basis for our work. The argumentation for this decision is given below. First, we identify the main criteria for this choice. Then, we use them to compare the different algorithms.

The most important argument for our decision is the degree to which the algorithm has been shown to be of practical use. In particular, the number of systems it is used in and openly available data on its performance. If an algorithm is used in academia as well as in industry, we deem this as a advantage. This criteria carries a lot of weight because the first literature on worst-case optimal joins has been rather theoretical but in our work we take a more praxis and system oriented perspective.

The practical character of our work also motivates the second dimension which we compare the algorithms in, namely ease of implemenation. If two of the three algoritms both have well proven performance, we would like to choose the algorithm that takes less time to implement and is easier to adapt and experiment with. That is, to be able to spent more time on evaluation and optimizations for the graph use-case, instead of, time spent on replicating existing work.

The Leapfrog Triejoin is used in two commercial database solutions: LogicBlox [10] and RelationalAI[4]. Its performance has been reported on in two publications [18, 44]. In particular, it beats various general and graph specific databases for graph pattern matching, i.e. PostgresSQL, MonetDB, NEO4J, graphLab and Virtuoso [44]. The broadest study of its performance uses 15 different datasets and 7 queries [44]. We conclude that the performance of LFTJ is well established by peer reviewed publications as well as industrial usage.

The *NPRR* algorithm has been well analyzed from the theoretical point of view. However, we are not able to find any openenly available sources with performance measurements. This disqualifies NPRR as basis for our thesis.

The *Generic Join* is used in at least three academic graph processing engines, namely GraphFlow [28], EmptyHeaded [1] and a unnamed implementation in Timely Dataflow [7]. All three show good performance. However, we are not aware of any commercial systems using GJ.

The comparision of Leapfrog Triejoin, NPRR and *Generic Join* by proven performance rules out NPRR and puts LFTJ and GJ on a similar level. Next, we compare these two algorithm in ease of implementation.

The description of the Leapfrog Triejoin implementation in its original paper [51] is excellent. Furthermore, multiple open source implementation exists [48, 18]. In particular, the implementation of Christian Schroeder for a course at Oxford is helpful because it is standalone and does not require us to understand a whole system[5].

[4]https://www.relational.ai/

[5]https://github.com/schroederdewitt/leapfrog-triejoin

Generic Join is described as a generalization of NPRR and Leapfrog Triejoin in its original paper [43]. Although, well written and algorithmically clear, this explanation is much less practical than the one given for LFTJ which is backed by an executable implementation.

To conclude, we choose Leapfrog Triejoin as basis for our work based on its openely available records of performance, use in academia as well as industrial systems and good description for direct implementation. Furthermore, Peter Boncz (supervisor of this thesis) has direct contact to the inventors of LFTJ giving us access to valuable expertise if necessary.

2.2.1 Leapfrog Triejoin

In this section, we described the Leapfrog Triejoin algorithm. In the first paragraph, we give the high-level idea behind the algorithm and some of its requirements. Then we discuss the kind of queries that can be answered with it. The main part of the section, discusses the conceptual algorithm itself. We finish with a short discussion of two implementation problems, namely the data structure to represent the input relationships and the problem of choosing a good variable ordering.

The Leapfrog Triejoin is a variable-oriented join. Given an input queries, it requires a variable ordering. For example in the triangle query $triangles(a, b, c) \leftarrow R(a, b), S(b, c), T(a, c)$, the variable ordering could be a, b, c. Furthermore, the Leapfrog Triejoin requires its input relationships to be sorted by lexicographic, ascending order over the given variable ordering, e.g. R needs to be sorted by primarily by a and secondary by b given the variable ordering a, b, c. The algorithm is variable-oriented because it fixes one possible binding for a, one for b given a and finally one for c given a and b. This allows it to enumerate the result of the join query without intermediary results. The process can be thought of as a backtracking, depth-first search for possible bindings.

The algorithms impelemted in this thesis can process joins of the full conjunctive fragment of first order logic or conjunctive equi-joins in relational algebra terms. Possible extentsions to disjunctions, ranges (none-equi joins), negation, projection, functions and scalar operations on join variables are explained in the original Leapfrog Triejoin paper [51]. However, they are not relevant to the core of this work because many interesting graph patterns can be answered using the full conjunctive fragment, e.g. cliques or cycles.

The Leapfrog Triejoin algorithm uses three components which are composed in a layered fashion. The concrete composition used for the triangle query is shown in fig. 9. In this figure, we see three layers each of them made of one or more instances of a component. The components are the *TrieIterator*, *LeapfrogJoin* and *LeapfrogTriejoin*. In the next paragraphs, we explain each layer in order, starting with the lowest layer.

The lowest layer is made of one *TrieIterator* per input relationship. In our example, we have three instances one for R, S and T each. The *TrieIterator* interface represents the input relationship as a trie with all values for the first attribute on the first level, the values for the second attribute on the second level and so on; an example for this is shown in fig. 10.

The trie contains one level per attribute of the relationship; in the case of the triangle query, there are two levels: one for a and one for b. Each level is made of all possible values for its attribute. All tuples of the relationship can be enumerated by a depth-first traversal of the trie.

The *TrieIterator* component offers six methods shown in table 1. The *open* and *up* methods control the level the iterator is positioned at; *open* moves it one level down and *up* moves it one level up. Additionally, *open* places the iterator at the first value for the next level and the up method returns to the value of the upper level that was current when the deeper level was opened. We call these two methods the vertical component of the *TrieIterator* interface.

The other four methods are called linear component. All of them operate on the current level of

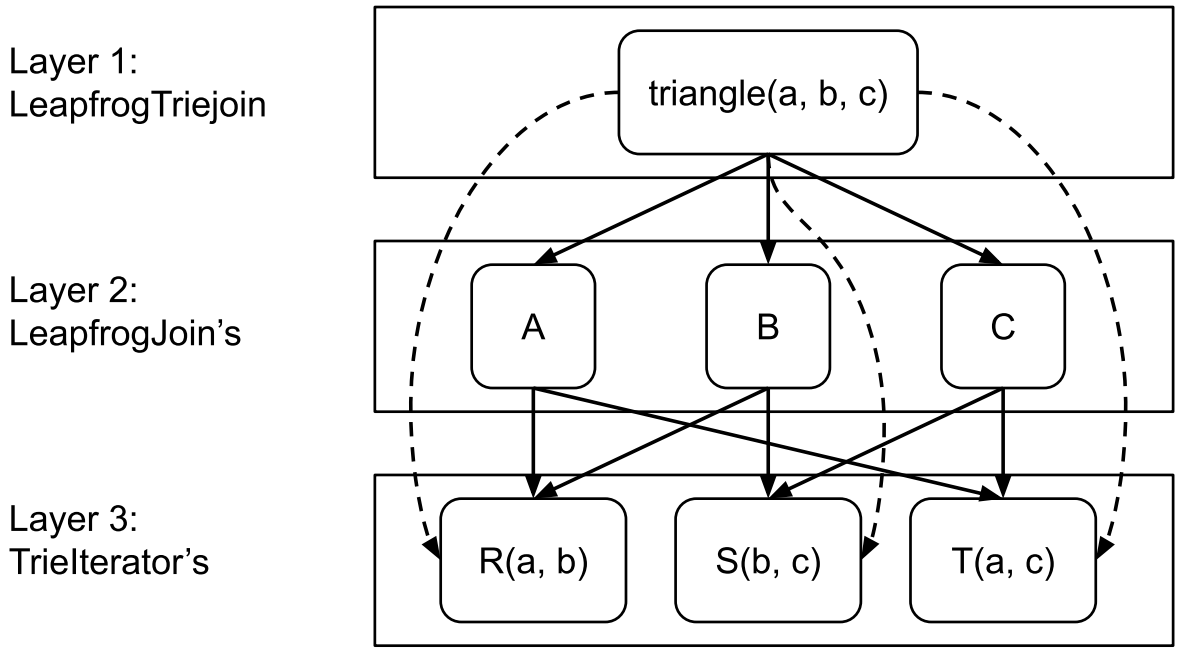


Figure 9: The three layers of the Leapfrog Triejoin algorithm. The configuration for a triangle query is shown: three *TrieIterators* one per input relationship, three *LeapfrogJoins* one per variable and one *LeapfrogTriejoin* component are neccessary. The arrows indicate that a component uses another. The *LeapfrogTriejoin* uses all other components but only the vertical part of the *TrieIterators* (dashed arrows). The *LeapfrogJoins* uses the linear part of two *TrieIterators* each.

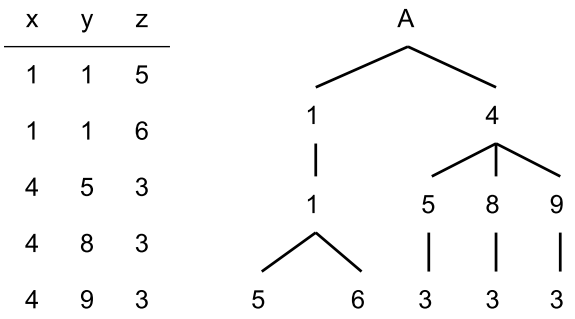


Figure 10: A 3-ary relationship as table (left) and trie (right), to position the iterator at the tuple (1, 1, 5) one calls *open* twice, *key* returns now 5, after a call to *next*, *key* returns 6 and *up* would lead to *key* returning 1.

Method	required complexity
TrieIterator	
int key()	$\mathcal{O}(1)$
bool atEnd()	$\mathcal{O}(1)$
void up()	$\mathcal{O}(\log_N)$
void open()	$\mathcal{O}(\log_N)$
void next()	$\mathcal{O}(\log_N)$
void seek(key)	$\mathcal{O}(\log_N)$
LeapfrogJoin	
int key()	$\mathcal{O}(1)$
bool atEnd()	$\mathcal{O}(1)$
void init()	$\mathcal{O}(\log_N)$
void next()	$\mathcal{O}(\log_N)$
void seek(key)	$\mathcal{O}(\log_N)$

Table 1: The interfaces of *TrieIterator* and *LeapfrogJoin* with required complexity. N is the size of relationship represented by the iterator.

the *TrieIterator*. The *key* function returns the current key (a single integer). The *next* method moves the iterator to the next key on the same level. The *seek(key)* operation finds the least upper bound for its parameter *key*. Finally, the *atEnd* method returns *true* when the iterator is placed behind the last value of the current level.

The middle layer of the Leapfrog Triejoin is made of one *LeapfrogJoin* per variable in the join. This join generates possible bindings for its variable by intersecting the possible values for all input relationship containing the variable. Therefore, it operates on the linear component of all *TrieIterators* of relationships with this variable. Figure 9 for the triangle query shows three *LeapfrogJoin* instances (for a, b and c); each of them uses two *TrieIterators*.

The *LeapfrogJoin* interface has five methods shown in table 1, with their required asymptopical performance. In the following paragraphs, we explain each of them. In short, the join offers an iterator interface over the intersection of its input iterators. This intersection is found by repeatingly seeking the value of the largest input iterator in the smallest input iterator. This process resemples a frog taking a leap which gives the join its name. When all iterators point to the same value leapfrogging stops and the value is emitted as part of the intersection.

The *init* operation sorts the input iterator by their current key and finds the first value of the intersection. To find the first value it uses the private method *leapfrogSearch* which is the work-horse of the whole join. The algorithm of this method is shown in algorithm 1. This method loops the process of calling the *seek* method of the its smallest input iterator with the key of the largest input iterator until the smallest and the largest (and therefore all iterators) point to the same value.

The *leapfrogNext* method moves the join to its next value. Internally, it uses the *next* function of its smallest iterator and then *leapfrogSearch*.

The operation *leapFrogSeek(key)* first uses the *seek* method of the smallest input iterator to forward it to *key*; then it uses *leapfrogSearch* to either verify that this key is available in all iterators (hence in the intersection) or to find the upper bound of this key.

Finally, the functions *key* and *atEnd* return the current key or if the intersection is complete respectively.

Algorithm 1: leapfrogSearch()

Data:
iters *sorted array of TrieIterators*
p *index of the smallest iterator*
Result: *Either* atEnd *is true or* key *is set to next key of intersection*

```
1  maxKey ← iters[p % iters.length].key()
2  while iters[p].key() ≠ maxKey do
3  |  iters[p].seek(maxKey)
4  |  if iters[p].atEnd() then
5  |  |  atEnd ← true
6  |  |  return
7  |  else
8  |  |  maxKey ← iters[p].key()
9  |  |  p ← (p + 1) % iters.length
10 key ← iters[p].key()
```

The last layer of the whole algorithm is a single *LeapfrogTriejoin* instance. It interacts with both lower layers to enumerate all possible bindings for the join. For this it aquires one binding for the first variable from the corresponding *LeapfrogJoin*. Then it moves the *TrieIterators* containing this variable to the next level and finds a binding for the second variable using the next *LeapfrogJoin*. This process continues until all variables are bound and a tuple representing this binding is emitted by the join operator. Then it finds the next possible binding by backtracking.

Algorithm 2 shows the backtracking depth-first traversal. This traversal needs to stop each time when a complete tuple has been found to support the iterator interface of the join. Therefore, it is implemented as a state-machine which stops each time the deepest level is reached and all variables are bound (loop condition in line 33). The next action of the state machine is determined by the outcome of the current action. Hence, we can characterize the state machine by describing each possible action and its possible outcomes. There are three possible actions: *next*, *down* and *up*. We summarize the possible actions, conditions for the next action and if the main loop of the state machine yields the next tuple in table 2 and describe each action below.

The *next* action moves the *LeapfrogJoin* at the current depth to the next possible binding for its variable (line 4). If the *LeapfrogJoin* reached its end, we continue with the *up* action (line 6), otherwise we set the binding and continue by another *next* action, if we are at the deepest level or by moving to the next deeper level by the *down* action (line 8ff).

The *down* action moves to the next variable in the global variable ordering by opening all related *TrieIterators* and initializing the corresponding *LeapfrogJoin* (line 14 call to *trieJoinOpen*). A *down* can be followed by an *up* if the *LeapfrogJoin* is *atEnd* (line 16), by a *next* action if the trie join is at its lowest level (line 20), or by another *down* action to reach the deepest level.

The *up* action can signal the completion of the join if all bindings for the first variable in the global ordering have been explored, or in other words, the first *LeapfrogJoin* is *atEnd* (condition $depth == 0 \wedge action == UP$ line 24). Otherwise, all *TrieIterators* corresponding to the current variable are moved upwards by calling *triejoinUp* (line 28) which also updates *depth* and *bindings*. Then, this action is followed by another *up* or a *next* depending on *atEnd* of the current *LeapfrogJoin* (lines 29).

TrieIterator implementation, backing data structure While we can implement the *LeapfrogJoin* and *LeapfrogTriejoin* component of the Leapfrog Triejoin from the algorithmic

Algorithm 2: LeapfrogTrieJoin state machine. *trieJoinUp* and *trieJoinOpen* move all *TrieIterators* that involve the current variable a level up respectively down.

```
   Data: depth the index of the variable to find a binding for, ranges from -1 to #variables - 1
   bindings array holding the current variable bindings or −1 for no binding
   MAX_DEPTH the number of variables - 1
   action state of the state machine
 1 repeat
 2     switch action do
 3         case NEXT do
 4             leapfrogJoins[depth].leapfrogNext()
 5             if leapfrogJoins[depth].atEnd() then
 6                 action ← UP
 7             else
 8                 bindings(depth) ← leapfrogJoins[depth].key()
 9                 if depth == MAX_DEPTH then
10                     action ← NEXT
11                 action ← DOWN
12         case DOWN do
13             depth ← depth + 1
14             trieJoinOpen()
15             if leapfrogJoins[depth].atEnd() then
16                 action ← UP
17             else
18                 bindings(depth) ← leapfrogJoins[depth].key()
19                 if depth = MAX_DEPTH then
20                     action ← NEXT
21                 else
22                     action ← DOWN
23         case UP do
24             if depth = 0 then
25                 atEnd ← true
26             else
27                 depth ← depth - 1
28                 trieJoinUp()
29                 if leapfrogJoins[depth].atEnd() then
30                     action ← UP
31                 else
32                     action ← NEXT
33 until ¬ (depth = MAX_DEPTH  bindings[MAX_DEPTH] ≠ -1) ∨ atEnd
```

Action	Condition	Next action	Yields
	lf.atEnd	UP	no
NEXT	$\neg$*lf.atEnd* $\wedge$ *reachedMaxDepth*	NEXT	yes
	$\neg$*lf.atEnd* $\wedge$ $\neg$*reachedMaxDepth*	DOWN	no
	lf.atEnd	UP	no
DOWN	$\neg$*lf.atEnd* $\wedge$ *reachedMaxDepth*	NEXT	yes
	$\neg$*lf.atEnd* $\wedge$ $\neg$*reachedMaxDepth*	DOWN	no
	depth = 0, means highest *lf.atEnd* is true	– (done)	yes
UP	*lf.atEnd*	UP	no
	$\neg$*lf.atEnd*	NEXT	no

Table 2: Summary of actions, conditions for the following action and if a complete tuple has been found. *reachedMaxDepth* is true if we currently find bindings for the last variable in the global order. *lf* abbreviates the *LeapfrogJoin* of the current variable. The columns *Yields* details if the main loop of the state machine yields before computing the next action, this is the case, when all variables have been bound.

description given above, we are missing some details for a concrete implementation of the *TrieIterator* interface. Mainly, we need to decide for a datastructure to back the *TrieIterator*.

We choose to use sorted arrays as described in [18]. One array is used per column of the input relationship and binary search on these arrays allows us to implement the *TrieIterator* interface with the required asymptotic complexities (see table 1).

Variable ordering Finding a good variable ordering for the LFTJ is an interesting research problem in itself. We are aware of two existing approaches.

The first is to create and maintain representative samples for each input relationship and determine the best order based on runs over these samples. This has been implemented in LogicBlox, the first system to use Leapfrog Triejoins [10]. To the best of our knowledge, the exact method of creating the representative samples has not been published.

The second approach is described in great detail by Mhedhbi and Salihoglu in [39]. Its has been implemented in their research graph database Graphflow [28].

They define a novel cost-metric for WCOJs which estimates the costs incured by constructing the intersections of adjacency lists. The metric takes three factors into account. First, the size adjacency lists.

Second, the number of intermediate matches. The concept of intermedieate matches is best understood by a simple example; we see the tailed-triangle query in fig. 11. Two very different vertice ordering categories exist for this query. The ones that start on v_4 and find all 2-paths of the graph; and vertice orderings that start with v_1, v_2, v_3 in any order which close the triangle first. Clearly, there are more 2-paths in any graph than triangles. Hence, the second category produces far less intermediate matches.

Finally, they implement an intersection cache in their system which takes advantage of the fact that some queries can reuse already constructed intersections. So, the last factor taken into account by their cost metrics is the usage of this intersection cache.

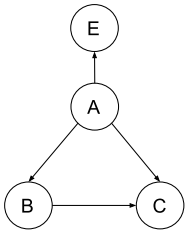


Figure 11: The tailed triangle; an example for the cost of intermediate matches.

They use the described cost metric, a dynamic programming approach to enumerate possible plans and a catalogue of sampled subgraph instances containing the sizes of adjacency lists to intersect and produced intermediate results to estimate the costs for all variable orderings.

Moreover, they implement the ability to change the query ordering adaptively during query execution based on the real adjacency list sizes and intermediate results. They show that adaptive planning can improve the performance of many plans. Furthermore, it makes the query optimizer more robust against choosing bad orderings.

To conclude, the work of Mhedhbi et al. is the most comprehensive study on query vertex orderings for WCOJs currently available; they introduce a cost metric, a query optimizer to use this metric and prove that it is possible and beneficial to compute parts of the results using a different variable order.

In our work, we do not implement an automatic process to choose the best variable order. The order we choose is based on experiments with different orders and intuition of the author.

Integrating the approach of LogixBlox would be possible but require the implementer to find a good sampling strategy because no details are openenly available.

The approach of Mhedhbi and Salihoglu is much better documented but also more complex. It consists out of four contribution which build up on each other but could be useful on its own. The cost metric described in their paper applies to our system as well and could be used.

They use this metric for cost estimation in connection with a none-trivial subgraph catalogue. The main challenge in integrating this way of cost estimation with our system is to elegantly integrate catalogue creation in Spark.

Their solution for adaptive variable orderings is helpful because it proves that this technique is beneficial; they also publish performance measurements, so the impact can be evaluated. However, there system employs a *Generic Join* while we use a Leapfrog Triejoin. The integration of adaptive variable orderings into Leapfrog Triejoin is not trivial and it is likely that their implementation is not directly applicable.

Finally, they introduce an intersection cache to make use of repeatedly used intersections. This can be directly applied to our system, e.g. using the decorator pattern around *LeapfrogJoins*. We note that they only cache the last, full n-way-interesection of multiple adjacency lists. It would be interesting to research if the system would benefit from caching partial n-way intersections as well because we noticed that for some queries, e.g. 5-clique, the intersection between the first two lists can be reused more often than the full intersection. This opens the interesting question in which order we should intersect the lists.

We conclude that two concepts to choose a good variable ordering exists which are both (partially) applicable to our system. The LogixBlox approach is simpler and directly integratable but not well documented. The solution used in GraphFlow is far more complex and developed for another WCOJ. Anyhow, the paper describes it in great detail and parts of it could be integrated directly,

while others need some engineering effort or need to be redesigned completely.

2.3 Distributed worst-case optimal join in Myria

In 2014, a Leapfrog Triejoin variant, dubbed Tributary Join, was used as a distributed join algorithm on a shared-nothing architecture called Myria [18]. They use Tributary Join as a local, serial worst-case optimal join algorithm, combined with the Hypercube shuffle algorithm, also called *Shares*, to partition the data between their machines [4]. The combination of a shuffle algorithm with a WCOJ allows them to distribute an unchanged serial worst-case optimal join version by running it only on a subset of the data on each worker.

This approach is directly applicable to Spark. We could implement a hypercube shuffle for Spark and then choose any WCOJ to run on each partition. However, it is not obvious how well this approach scales because Shares replicates many of its input tuples [18]. The experiments on Myria indicate that the combination of Hypercube shuffles and Tributary Join does not scale well. They report a speedup of 8 on 64 workers compared to the time it takes on 2 nodes, which, although unlikely to be optimal, is not investigated in great detail.

Therefore, we decide to analyse the expected scaling behaviour of Shares for graph pattern matching. Our main concern is that the number of duplicated tuples in the system increases with the query size (number of vertices) and with the number of workers added to the system. We provide a theoretical analysis of the number of duplicated tuples for different query sizes and available workers in a later section of this thesis (2.3.1). As result of this investigation, we decide to not physically partition our data but to duplicate it to all workers and seek for alternative strategy to parallelize a worst-case optimal join.

To conclude, the implementation in Myria and in particular the Shares algorithm is the starting point of this thesis. We see it as our baseline for a distributed WCOJ implementation. In the coming section, we explain Shares in detail.

We note that a implementation of a distributed worst-case optimal join on Timely Dataflow exists. However, it is not applicable to Spark. Therefore, we treat it in the related work section (8.1) of this thesis.

2.3.1 Shares

Shares partitions the input relationships for a multi-way join over worker nodes, such that, all tuples, which could be joined, end up on the same worker in a single shuffle round. Hence, it allows running any multi-way join algorithm locally after one shuffle round. The output of the join is the union of all local results.

The idea is to organize all workers in a logical hypercube, such that each worker can be addressed by it hypercube coordinate. Then it is straightforward to find a mapping from the attribute values of a tuple to these coordinates, so that joinable tuples arrive on the same worker after one shuffle.

We first explain how to organize the workers in a hypercube and then how to map tuples to these workers. Next, we treat the problem of choosing a good hypercube configuration. Followed, by a summary about optimality of Shares. Finally, we provide analysis of the scaling of Shares for graph pattern matching.

A hypercube is characterized by the number of dimensions and the size of each dimension. Figure 12 shows a hypercube with three dimesions labelled a, b and c. They have the size of 3, 2 and 2 for a, b and c respectively. It is a possible configuration for the triangle query $triangle(a, b, c) \leftarrow R(a, b), S(b, c), T(a, c)$ with 12 workers.

a	b
1	2
2	3
2	4
3	1
(2, 0, *)	

b	c
1	2
2	3
2	4
3	1
(*, 0, 1)	

c	a
1	2
2	3
2	4
3	1
(2, *, 1)	

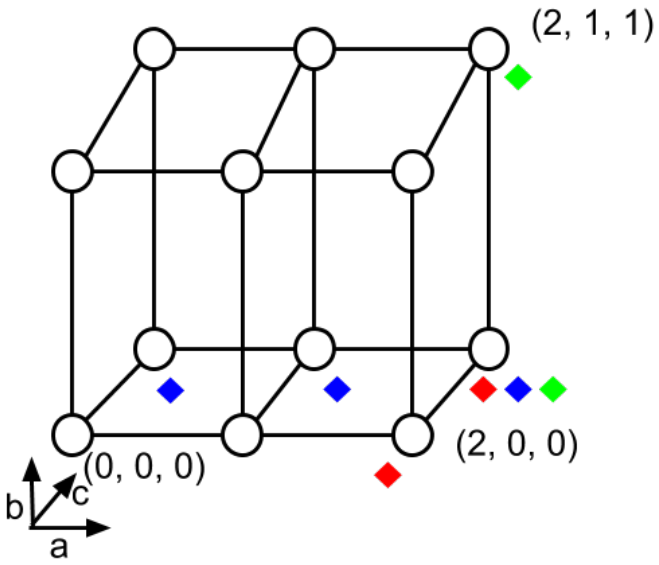


Figure 12: Left: Three aliases of an edge relationship with one triangle. The participating tuples are marked in red, green and blue. Their hypercube coordinates are shown below. Right: Example of a Shares hypercube configuration for the triangle query for 12 workers with three attributes/dimensions of the sizes 3, 2, 2. The tuples marked in red, green and blue end up on the workers with red, green and blue rhombs respectively.

Given an input query, Shares builds a hypercube with one dimension per variable in the input. It then chooses the size of each dimension, such that the product is smaller than number of workers. We call v the numbers of variables in the query and $p_0, \ldots, p_v$ the sizes of each dimension. This allows us to address each worker with a coordinate of the form $(0..p_0, \ldots, 0..p_v)$. If the product of all dimension sizes is smaller than the number of workers, additional workers are not used. The process of finding the best sizes for the dimensions depends on the input query and the input relationships. We discuss it in a later paragraph of this section.

With this topology in mind, it is straightforward to find a partitioning for all tuples from all relationships such that tuples that could join are sent to the same worker. We choose a hash function for each join variable a which maps its values in the range of $[0..p_a]$. Then each worker determines where to send the tuples it holds by hashing its values. This results in a coordinate in the hypercube which is fixed for all join variables, which occur in the tuple, and unbounded for join variables which do not occur in the tuple. Then the tuple is sent to all workers with a matching coordinate.

Figure 12 shows how tuples forming a triangle from three relationships are mapped to the workers. The blue, green and red tuple in the relationships form a triangle. The green and the red tuple are sent to 2 workers each and the blue tuple to three workers (marked with small rhombs). They are sent to all workers along the axis where the coordinate is not determined by a value in the tuple. We see that they all end up together on the worker with the coordinate (2, 0, 0). This is where the triangle occurs in the output of the join.

Finding the best hypercube configuration The problem of finding the best hypercube configuration is to choose the sizes of its dimensions such that (1) the product of all sizes is smaller than the number of available workers and (2) such that the number of tuples to single worker is

minimized. (2) is backed by the assumption that the number of tuples is a good indicator for the amount of work; this assumption is made in all papers discussing the problem [18, 13, 4]. Therefore, we want to minimize the number of tuples on each single worker because the slowest worker determines the run-time. Next, we discuss existing solutions and decide for one of them.

The original Shares paper proposes a none-convex solution which is hard to compute in praxis [4]. Later, Beame et al. define a linear optimization problem which is solvable but leads to fractional hypercube sizes [13]. Hence, it is not possible to use their solution directly. Rounding down would be an obvious solutions but as discussed in [18], it can lead to highly suboptimal solutions, in particular with low numbers of workers. Hence, the paper further considers to use a higher number of *virtual* workers and assign these to *physical* workers by a one-to-many mapping. Anyhow, a higher number of workers lead to more replicated tuples. Therefore, this solution does not scale well.

In the end, the paper that integrates Shares and the Tributary join in Myria suggests a practical solution. They enumerate all integral hypercube configurations smaller or equal to the number of available workers. For each configuration they estimate number of assigned tuples, then they choose the configuration with the lowest estimated workload.

They use the following equation to estimate the workload, where R are all relationships in the query, $var(r)$ gives the variables occuring in the relationship r, $size(v)$ gives the hypercube size of the dimension for variable v.

$$workload = \sum_{r \in R} |r| \times \frac{1}{\prod_{v \in var(r)} size(v)} \tag{3}$$

The term $\prod_{v \in var(r)} size(v)$ gives the numbers of workers that span the hyper-plain over which a relationship r is partitioned. For example, in fig. 12 the relationship (a, b) is partitioned over the plain spanned by the dimensions a and b with 6 workers. Each tuple in this relationship has a chance of $\frac{1}{6}$ to be assigned to any of these workers. Hence, the workload caused by (a, b) is $|(a, b)| \times \frac{1}{6}$.

The paper evaluates this strategy to assign hypercube configurations and finds that it is efficient and practical. We choose to use the same solution for our work.

Shares is worst-case communication-cost optimal Shares, as described above, is shown to be worst-case optimal in its communication costs in MapReduce like systems for n-ary joins using one shuffle round. First, Beame et al. prove that Shares scheme is optimal on databases without skew [13]. Later the same authors are able to give a proof that Shares is also an optimal algorithm for skewed databases if one knows the *heavy-hitter* tuples and splits the join into a skew free part and residual joins for the heavy hitters using different hypercube configurations for each residual join [30].

The implication of these proofs is that it is not possible to find a partitioning scheme for one shuffle round that replicates less data than Shares. This observation is central to our thesis because it is one argument to replicate the graph on all workers instead of using a shuffle algorithm to partition it.

In the rest of this thesis, Shares refers to the original algorithm [4] and not the skew resilient variant *SharesSkew* [3, 13]. This is mainly because even in the presence of skew the original Shares scheme offers good upper bounds, although it can not always match the lowest bound possible [3]. But also because the skew resilient variant requires to know which tuples are *heavy-hitters* (a definition of skew introducing tuples). Finally, while first experiments with SharesSkew exist [3], we are not aware of an extensive study verifying it is possible to integrate SharesSkew into a complete system. Hence, we deem it out of scope for this thesis to attempt a full integration.

Some readers might ask if there are better multi-round algorithms which replicate less data.

Indeed, the authors of a Shares related paper raise the same question as future work [30]. They are able to answer this question for specific join queries in [30, 3], e.g. chain-joins and cycle-joins. Later, they present an algorithm which is multi-round optimal for all acyclic queries [2] and one for all queries over binary relationships [29].

The papers about about multi-round optimal partitioning schemes are rather theoretical. To the best of our knowledge, only one paper provides practical experiments [3] but has no dedicated implementation section. Also, they have not been shown optimal for general conjunctive join queries but only for special cases. Two of the three papers [**c**]annot handle clique joins which are important class of joins in our thesis. Additionally, they add additional complexity to the query optimizer, e.g. they require the input query to be represented as generalized hypertree decomposition to calculate their intersection width [2] or to find many different hypercube configurations [29, 3, 30] which is not trivial in praxis and computation intensive as discussed in the last paragraph.

We leave it to future research to investigate the practical application of these algorithms to graph pattern matching. The most interesting paper in this direction is [29]. It develops a multi-round algorithm for n-ary joins on binary relationships like the edge relationship of a graph.

Analysis of Shares scalability Next, we analyse the scalability of Shares on growing graph patterns. That is, self-joins over a single relationship which has two variables. In this context, relationships of the join can be seen as the edges of the pattern and variables as vertices.

First, we fix the method to determine the best hypercube configuration $(p_1 \dots p_k)$, given a query. For this, we use the method described above and used in [18].

Given the hypercube configuration and a query, we can estimate the workload of each worker by the formula 3. Let R be the set of all atoms in the join[6], $size1(r)$ and $size2(r)$ be the size of the first respectively second hypercube dimension for the two variables in atom r. Then, each worker receives $\sum_{r \in R} \frac{|r|}{size1(r) * size2(r)}$ tuples under the assumption of uniform data distribution and good hash functions. Our argument is that the tuples of each atom r are divided onto $size1(r) * size2(r)$ workers; the workers that form the hypercube plain of its two variables.

In the special case of graph pattern matching where all atoms of the query are pointing to the same relationship, we can optimize the hypercube shuffle such that a tuple is only sent once to a worker, although it might be assigned to it via multiple atoms.

If we apply this optimization, we can predict the probability with which each tuple is assigned to a worker using the Poisson binomial distribution. The Poisson binomial distribution $\Pr(n, k, u_0, \dots, u_n)$ allows us to calculate the likelihood that k out of n independent, binary and differently distributed trials succeed, under the condition that the i the trial succeeds with a probability of u_i. We use $n = |R|$, $k = 0$ and $u_i = 1/(size1(r_i) * size2(r_i))$ to calculate the probability that a tuple is not assigned to an arbitrary, fixed worker. This allows us to predict the number of tuples assigned to each worker by $|E| * (1 - \Pr(|R|, 0, u_0, \dots, u_{|R|})$ with E being the edge relationship.

Table 3 shows the expected percentage of tuples from the edge relationship assigned to each worker for graph patterns of different sizes calculated using Poison binomial distribution and optimal shares assignments according to the method used in [18]. As we can see in this table, the number of tuples assigned to each worker grows over linear in the size of the graph pattern. Furthermore, doubling the number of workers is inefficient to counter this growth.

In particular, already small clique queries of four vertices replicate over half of the tuples on all

[6] An atom in a datalog join is the reference to a relationship, e.g. $triangle(a, b, c) \leftarrow R(a, b), S(b, c), T(a, c)$ has three atoms named R, S and T. In this section, we differentiate atoms and relationships because multiple atoms can point the same underlying relationship which becomes of particular importance.

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

Table 3: Workload on 64 and 128 workers in percentage of tuples of the edge table assigned to each worker estimated by using Poison binominial distribution to estimate the workload and the method from [18] to determine the optimal shares configuration.

64 workers. 5-clique queries require nearly a full broadcast with each worker holding 82% or 90% of all tuples with 128 respectively 64 workers. The diamond query used in practice by the Twitter recommendation engine has to replicate far more than half of the tuples to all workers.

The second observation has two reasons. First, doubling the number of workers does not allow us to double the dimensions of the hypercube because a hypercube always needs product of all dimension sizes to be built. Second, the number of replicated tuples increases with a growing hypercube because each tuple is replicated to more workers; namely $\prod_{r \in R/r} size1(r) * size2(r)$ workers. This is because each tuple binds only two out of all variables. Hence, it is replicated over many dimensions.

In light of the numbers presented in table 3 and in line [7], we conclude that the communication costs for Shares converge towards a full broadcast for bigger graph patterns and scaling becomes increasingly inefficient. By this observation and the fact that hypercube shuffling is an optimal scheme (see the last paragraph), we decide against using any partitioning scheme in our work but replicate the edge relationship on all workers.

2.4 Compressed sparse row representation

Compressed sparse row representation (short CSR) is a well known, low-memory representation for static graphs [17, 49]. To ease its explanation, we assume that the graph's vertices are identified by the numbers from 0 to $|V| - 1$. However, our implementation allows the use of arbitrary vertice identifiers in $\mathcal{N}$ by storing the translation in an additional array of size $/V/$.

CSR uses two arrays to represent the edge relationship of the graph: one of size $/E/$ which is a projection of the edge relationship onto the *dst* attribute (called *AdjacencyLists*) and a second of size `|V + 1|` which stores indices into the first array (called *Indices*). To find all destinations directly reachable from a source $src \in V$, one accesses the second array at *src* for the correct index into the first array for a list of destinations.

Figure 13 shows an example for a table and its CSR.

First, we note that vertices are not represented by their original ID but as numbers from 0 to |V| which gives the offset into the *Indices* array where to find the offset into the *AdjacencyLists* array for their neighbours, e.g. vertex 2 is represented by 1 which points to the offset 1 in the *Indices* array which in turn points to the adjacency list of 2 in *AdjacencyLists*.

Additionally, we point out that the 3rd entry in the *Indices* array is the same as the second. This is because vertex 3 has no outgoing edges. Hence, it has no adjacency list to point to. Therfore, it points to the same offset as the entry for 2.

The CSR format has two beneficial properties in the context of this thesis. First, it allows locating

a	b
1	2
2	3
2	4
2	5
2	6
4	2
5	2
5	6
6	5

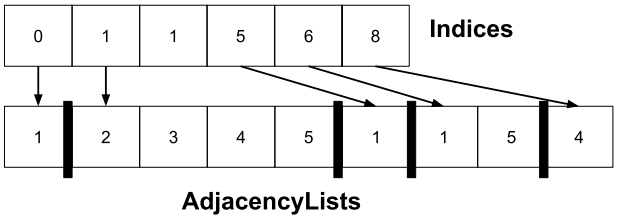


Figure 13: Example of a table and its compressed sparse row representation. The *Indices* array gives offsets into the *AdjacencyLists* array. The vertices are represented as the indexes into the indices array, e.g. vertex id 2 is represented as 1.

all destinations for a source vertice by one array lookup; hence, in constant time. Second, the representation is only, roughly, half as big than a simple columnar representation. A uncompressed columnar representation needs $2 \times |E|$ while CSR uses only $|V| + 1 + |E|$, note that for most real-world graph |V| « |E| holds (see section 2.5).

2.5 Sizes of public real-world graph datasets

In this section, we present a short analysis of the sizes of real worl graph datasets. For this, we collect data about all graphs from the SNAP and Laboratory of Web Algorithms dataset collection [35, 15]. The graphs in the Snap dataset are a bit older; they have been collected between 2000 and 2010. All Labaratory of Web Algorithms graph have been collected between 2007 and 2018. Both dataset collections are heavily used and cited in academia [7, 44, 18, 23, 32]. Two of these papers are from 2019.

For our size calculation we assume that the graph is stored in compressed sparse row representation (see section 2.4) using integers for the vertice ID's. Then, we determine the storage size in bits by the formulae $32 \times |V| + 32 \times |E|$ with 32 the size of an integer in bits, V the set of all vertices in the graph and E the set of all edges in the graph.

Figure 14 shows a histogram of sizes for all 157 graphs from the two datasets. 104 of these graphs are smaller than 1 GB and only 8 graphs are bigger than 100 GB. The biggest graph is the friendship graph of Facebook from the year 2017 with 552.2 GB.

We conclude that even the biggest graph can be fitted in main memory of many cluster machines today. The vast majority could be fitted in the main memory of a simple desktop machine or laptop. This supports our argument to replicate graph data over all machines.

3 Worst-case optimal join parallelization

Based on the fact that Shares is an optimal partitioning scheme for n-ary joins in MapReduce like systems [4] and our analysis that Shares converges to a full broadcast of the graph edges (see section 2.3.1), we decided to forego physical partitioning of the graph. We cache the graph

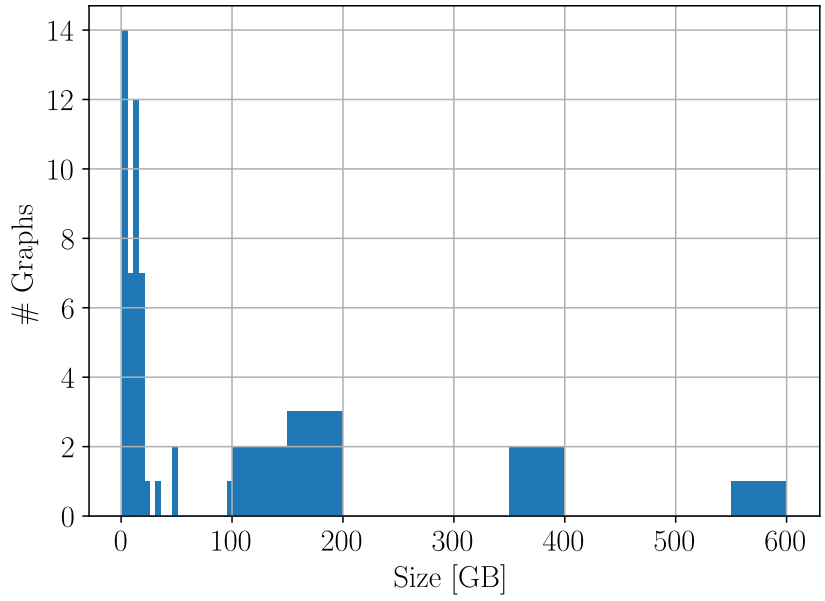


Figure 14: Sizes of all graphs from the SNAP and Labaratory of Web Algorithms dataset collection in giga bytes. The histogram shows graphs up to 100 GB in buckets of 5 GB and in buckets of 50 GB after. In total, we see data collected about 157 graphs.

in memory such that each Spark task can access the whole graph. Then, we experiment with multiple *logical* partitioning schemes which ensure that each task processes only some parts of the graph. This design has a big advantage over physical partitionings. Each worker holds the full edge relationship, therefore, it can answer any possible query without needing to shuffle data or materializing new data structures for the *LeapfrogTriejoin*, e.g. sorted arrays or CSR representations. Arranging the data into suitable data structures and shuffling data is a one-off action on system startup.

This design allows us to implement a new flavour of the Shares partitioning in which we filter the vertices of the graph on-the-fly while processing it with our *GraphWCOJ* algorithm. We describe this contribution in section 3.2.

We also consider partitioning the work based on a single variable. Here we use the values of the bindings for this variable to determine if a worker processes a specific part of the join.

Furthermore, we consider a work-stealing based partitioning which does not replicate any work and produces less skew than Shares. This comes at the price of implementing work-stealing on Spark. The design of work-stealing in Spark is described in section 3.4.

In section 3.3, we compare *logical* Shares, range based, *logical* Shares and single variable partitioning in terms of scalability and skew resilience.

3.1 Single variable partitioning

As first baseline, we implement partitioning along a single variable. We partition the values of this variable into as many ranges as the desired level of parallelism. Each variable can take V values where V is the number of vertices in the graph. So, if we have w workers and partition along the first variable, the first worker processes the first $\frac{V}{w}$ values for bindings of the first

variable and ignores the rest.

We implement this partitioning as a range based filter on the *TrieIterators* of the join. A range filtered *TrieIterator* interface is trivially implemented by changing the *open* method to seek for the upper bound of the first value in the range and rewrite the *atEnd* method to return true once the key value is higher than the upper bound of the range.

We chose a range filter because it is easily pushed into the *TrieIterator* interface and cheap to compute. As opposed to a hash-based filter which we found not suitable to be pushed into the *TrieIterator* interface (see section 3.2) and more expensive to compute.

The single variable partitioning is interesting because it allows us to trade duplicated work against skew resilience. A partitioning on the first variable to bind in the join is free of any duplicated work. Partitionings based on any other variable run the same work on each worker up to the depth of the variable to partition on but are duplication free after. These partitioning on later variables tend to be more skew resilient because there are fewer variables still to bind, which are often restricted by the bindings of earlier variables. In particular, partitioning on the second variable is interesting because bindings for the first variable are cheap to compute; they are the scan of the first level of a *TrieIterator*.

3.2 Logical Shares

Shares has been developed as an optimal shuffle for n-ary joins on MapReduce like systems. So, it is used to physical partition the tables participating in the join overall workers of the system. Then, each worker works only on the tuples it holds in its partition. This has been implemented in Myria to be used with a WCOJ [18] and for Hadoop [9]. We describe the Shares and Myria in more detail in section 2.3 and assume that the reader is familiar with this section.

The idea of Shares can also be used for a *logical* partitioning scheme. Instead, of partitioning the graph before computing the join, we determine if a tuple should be considered by the join on-the-fly. We do so by assigning a coordinate of a hypercube to each worker. Then each worker is responsible for the tuples which match its coordinate as in the original Shares. However, a huge difference to a physical Shares partitioning exists: while physical Shares keeps one prefiltered, materialized partition of the edge relationship per relationship of the join in memory, we keep only a single copy of the graph edge relationship in memory which can be used by all *TrieIterator* for all edge table aliases. Once, the edge relationship is broadcasted and cached, we do not need to materialize prefiltered partitions of it before every query.

Filtering tuples on-the-fly in the LFTJ comes with a challenge: in the *LeapfrogTriejoin* we do not consider whole tuples but only single attributes of a tuple at the time, e.g. a *LeapfrogJoin* only considers one attribute and cannot determine the whole tuple to which this attribute belongs. Fortunately, a tuple matches only if all attributes match the coordinate of the worker. Hence, we can filter out a tuple if any of its attributes do not match. For example, we can exclude a value in a *LeapfrogJoin* without knowing the whole tuple.

Integrating Shares and LFTJ comes with two important design decisions. First, the *LeapfrogTriejoin* operates on a complete copy of the edge relationship. Hence, we need to filter out the values that do not match the coordinate of the worker. Second, we need to compute the optimal Hypercube configuration. We describe our solutions below.

The first design decision is where to filter the values. The *LeapfrogTriejoin* consists out of multiple components which are composed as layers upon each other. On top we have the *LeapfrogTriejoin* which operates on one *LeapfrogJoin* per attribute. The *LeapfrogJoins* uses multiple *TrieIterators*. Our first instinct is to push the filter as deep as possible into these layers.

We built a *TrieIterator* that never returns a value which hash does not match the coordinate.

This is implemented by changing the *next* and *seek* methods such that they linearly consider further values until they find a matching value if the return value of the original function does not match. However, the resulting LFTJ was so slow that we abandoned this idea immediately. We hypothesize that this is the case because the original *next* and *seek* method is now followed by a linear search for a matching value. Furthermore, many of these values are later dropped in the intersection of the *LeapfrogJoin* which can also be seen as a filter over the values of the *TrieIterators*. As we know from section 4.2, the *LeapfrogJoin* is a rather selective filter. It does not make sense to push a less selective filter below a more selective filter.

With this idea in mind, we build a logical Shares implementation that filters the return values of the *leapfrogNext* method. This is implemented as a decorator pattern around the original *LeapfrogJoin*. The use of the decorator pattern allows us to easily integrate Shares with the LFTJ while keeping it decoupled enough to use other partitioning schemes.

The second design decision is how and where to compute the best hypercube configuration. The how has been discussed extensively in former literature [4, 18, 13, 3]. We implement the exhaustive search algorithm used in the Myria system [18].

In the interest of a simple solution, we compute the best configuration on the master before starting the Spark tasks for the join. We note that the exhaustive algorithm could be optimized easily and it would be worthwhile to introduce a cache for common configurations. Due to time constraints, we leave this to future work and keep our focus on the scaling behaviour of Shares.

To conclude, we succeeded to integrate Shares with *LeapfrogTriejoin* and report our results in section 7.5. We cannot improve on the main weakness of Shares that it duplicates a lot of work. Indeed, our design filters tuples only after the *LeapfrogJoin*. Therefore, all tuples are considered in the *TrieIterator* and their binary search of the first variable. This does not influence scaling much because only the correct logical partition of values for the first variable are used as bindings in the *LeapfrogTriejoin*. This means they are still filtered early enough before most of the work happens. We improve over a physical Shares by using the same CSR data structure for all *TrieIterator*. Therefore, we do not need to materialize a prefiltered data structure for each *TrieIterator* and query which saves time and memory if the partitions become large for bigger queries.

3.2.1 RangeShares

In the last section, we raised the point that our Shares implementation only filters out value after the *LeapfrogJoins*. This is because a hash-bashed filter needs to consider single values one-by-one. In this section, we explore the possibility to use range based filters which can be pushed into the *TrieIterators*. However, we warn the reader that this is a negative result. It leads to high skew which hinders good scaling of this idea.

We observe that the general idea behind Shares is to introduce a mapping per attribute from the value space into the space of possible hypercube coordinates, e.g. so far all Shares variants use a hash function per attribute to map the values onto the hypercube. We investigate the possibility to use ranges as mapping function, e.g. in a three-dimensional hypercube with three workers per dimension, we could divide the value space into three ranges; a value matches a coordinate if it is in the correct range. Contrary, to hash-based mappings which are checked value by value until one matches, a range check is a single conditional after each *seek* and *next* function call. Furthermore, this conditional is predictable for the processor because, for all but one call, the value is in range and returned. So, contrary to hash-based filter we can push a range based filter into the *TrieIterators*.

We implement this idea by dividing the vertice ids per attribute into as many ranges as the size of the corresponding hypercube dimension. For example, assume we have edge ids from 0 to 899, three attributes and the hypercube dimension have the size 3, 2 and 2. Then, we choose the ranges [0,300), [300,600) and [600,900) for the first attribute and the ranges [0,450) and [450,900)

for the other two attributes. The worker with the coordinate (0, 0, 0) is then assigned the ranges [0,300), [0,450) and [0,450). It configures its *TrieIterators* accordingly such that they are limited to these ranges.

We run the first experiments to evaluate this idea. We expect it to scale better than a hash-based Shares because it saves intersection work in the *LeapfrogJoins*. However, we find that high skew between the workers leads to much worse performance than a hash-based Shares. The explanation is that if a worker is assigned the same range multiple times and this range turns out to take long to compute, it takes much longer than all other workers.

To mitigate this problem, we break down the vertice ids into more ranges than there are workers in the hypercube dimension corresponding to the attributes. Then, we assign multiple ranges to each *TrieIterator* in such a way that the overlap on the first two attributes equals the overlap of a hash-based implementation and assign the ranges of the later attributes randomly such that all combinations are covered. However, experiments still show a high skew: some workers find many more instances of the searched pattern in their ranges than others. For the triangle query on *LiveJournal*, we find that the fastest worker outputs only 0.4 times the triangles than the slowest worker. We conclude that the pattern instances are unevenly distributed over the ranges of vertice ids which leads to high skew in a range based solution. We stopped our investigation in this direction.

3.3 Comparision of static partitioning schemes

In this section, we show how the static partitioning schemes implemented by us scale on the 3-clique query on the `LiveJournal` dataset. We partition the work according to logical Shares, on the first and second variable and according to both range-based, logical Shares schemes. The first range based logical Shares scheme uses a single range per *TrieIterator*, the second uses multiple ranges with improved overlapping; both have been described in the last section. We call them *SharesRange* and *SharesRangeMulti*.

The speedup with up to 48 workers is shown in fig. 15a. We measure the skew of a scheme as a relationship between the time it takes to compute the smallest and biggest partition. Figure 15b plots the skew for the different schemes, queries and levels of parallelism.

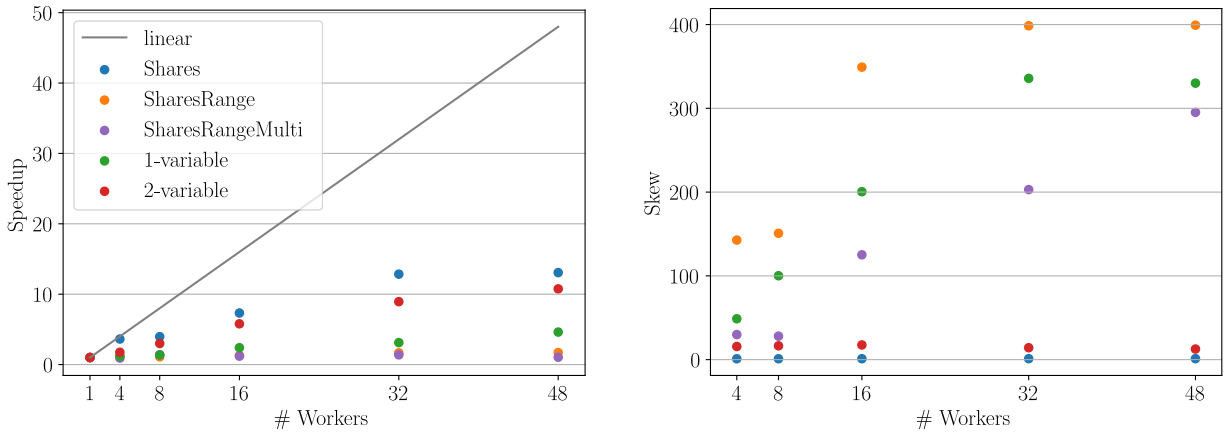
We see that logical Shares scales better than all other schemes. Partitioning on the second variable is slightly worse, while the other two schemes are a further behind.

Our explanation is that logical Shares and second variable partitioning inflect the lowest skew. There is a strong correlation between the skew shown in fig. 15b and speedup of fig. 15a.

Except for *SharesRangeMulti*, the amount of skew relates directly to the scaling behaviour. *SharesRangeMulti* is more skew resilient than first variable partitioning and *SharesRange* but does not scale better. That is because first variable partitioning does not replicate any work. Therefore, it's scaling is better even with higher skew between the workers. The difference between the two range based Shares partitioning in archived speedup is minimal. Most likely this are caused by the fact that *SharesRangeMulti* is implemented as a decorator around the *TrieIterator* interfaces while *SharesRange* is directly integrated into the *TrieIterator* interface.

Also, we see that *SharesRangeMulti* drastically reduces the skew compared to *SharesRange* which confirms that our optimization fulfils the intended goal. However, the reduced skew does not translate in better scalability for the range based Shares partitioning.

We conclude that various static partitioning schemes do not scale well. In the end, we find logical Shares is the best static partitioning scheme because it is best in managing skew which even beats work replication free schemes as partitionings on the first variable. We tried to improve hash-based logical Shares to push it deeper into the layers of the Leapfrog Triejoin



(a) Speedup

(b) Skew

Figure 15: 3-clique on LiveJournal for logical Shares, range based shares and partitioning on the first and second variable. Skew is measured as relationship between the time it takes to compute the biggest and the smallest partition.

algorithm but find that range-based Shares cannot handle skew well enough to be a competitor for hash-based Shares. Following from these results, we look into dynamic parallelization schemes as work-stealing.

3.4 Work-stealing

Normally, Spark uses static, physical partitioning of the data. As we learned in the last section that can lead to a trade-off between the ability to handle skew and duplicated work. A standard approach to handle skew and unbalanced workloads is work-stealing.

For this, the work is not statically partitioned before-hand but organized in many smaller tasks which can be solved by all workers. Workers are either assigned an equal split of tasks and steal tasks from other workers when they are out of work or all tasks are arranged in a queue accessible for all workers, so that workers can poll tasks from it whenever they are out of work. In either way, this results in a situation where no task is guaranteed to be solved by a single worker and each worker only finishes when no free tasks are left in the system. Hence, the maximum amount of skew roughly the size of the smallest task. There is no duplicated work because different tasks should not overlap and each task exists only once in the whole system.

We first describe a work-stealing version designed for the Spark's local mode where all tasks are computed on a single machine. Then we extend this design to the cluster mode of Spark.

Work-stealing requires two major design decisions from the developer. First, how to organize the workload of a *LeapfrogTriejoin* into tasks. Second, how do workers get their tasks? We address these questions in order by first describing our preferred solution and then their integration with Spark. We conclude the section with an evaluation of the limitations of this integration.

Before, we remember that we built our solution for the Spark *local-mode* (see section 2.1.2). Hence, the Spark master and all executors are threads within the same JVM process. This is important because it allows us to share data structures between multiple Spark tasks as normal JVM objects. We discuss how the design can be extended Spark's *cluster-mode* in future work (section 9.1.1) and related work (section 8.3).

The first design decision is the definition of a work-stealing task. It is not necessary to define the tasks such that they have all the same size. However, it is important to choose the task size small enough to avoid skew. Furthermore, the tasks should not overlap so that work is not duplicated. We choose to define a task as the work necessary to find all possible tuples for a single binding of the first join variable. This is none overlapping. The task size can vary widely and is query dependent. However, given the huge amount of tasks (as many as vertices in the graph), we believe this to be small enough. We plan to evaluate this during our experiments.

The second design choice in work-stealing is how to hand tasks to workers. For simplicity, we choose to use a shared, thread-safe queue that holds all tasks. The main drawback of this solution is that the access to the queue has to be synchronized between all workers. If there are too many workers contending for the critical section of polling a job from the queue, they can slow each other down. However, the critical section is short because it includes only the call to the poll method of the queue. Additionally, we decide to implement a batching scheme such that a single poll can assign multiple tasks to a worker. This allows us to fine-tune how often a worker needs to return to the queue for new tasks.

It turns out that the work-stealing scheme as described above is straightforward to integrate into Spark. We choose a Scala object[7] to hold a dictionary which associates an ID for each query with a thread-safe queue instance. This queue can be accessed by each Spark thread. Due to the association between query and queue, it is possible to run multiple queries in parallel without interference.

The queue for a query is filled by the first Spark task that accesses it. This can be implemented by a short synchronized code section at the beginning of all tasks. It checks if the queue is empty and if so pushes one task per possible binding (all graph vertices) or batch of possible bindings. The synchronized section is fast and only runs once when the tasks start. Hence, it comes at neglectable performance costs.

Once the queue is filled, we run our normal *LeapfrogTriejoin* with filtered *Leapfrog join* for the first attribute. This filter is implemented as a decorator around the original *Leapfrog join*. The *leapfrogNext* method of this decorator returns only values that are polled from the work-stealing queue before.

Our integration of work-stealing in Spark comes with some limitations. We see it more as a proof-of-concept that work-stealing is a good choice for the parallelization of worst-case optimal joins in Spark than as a solid implementation of work-stealing in Spark. The later is not possible within the time-frame of this thesis. In the following, we discuss the constraints of our integration.

Work-stealing leads to an unforeseeable partitioning of the results: it is not possible to foresee which bindings end up in a certain partition nor can we guarantee a specific partition size. If the user relies on any specific partitioning, he needs to repartition the results after. Moreover, we cannot guarantee to construct an equal partitioning over multiple runs of the same query. If the user depends on a stable partitioning per query, he should cache the query after the worst-case optimal join execution.

We do not integrate our work-stealing scheme into the Spark scheduler but we provide a best-effort implementation because we use all resources assigned to us as soon as they are assigned to us. We can handle all scheduler decisions. The first worker assigned fills the queue. The worker who takes the last element from the queue sets a boolean that this query has been completed. Hence, tasks that are started after the query has been computed, do not recompute the query.

We do not provide a fault-tolerant system. We see two possibilities to make our system fault-tolerant. First, one can stop all tasks if a single task fails and restart the computation with the last cached results before the worst-case optimal join. Second, one could extend the critical

[7]Methods and fields defined on a Scala object are the Scala equivalent to static methods and fields in Java. Most importantly they are shared between all threads of the same JVM.

section of polling a queue value by the *LeapfrogJoin* by two more operations: we peek at the value from the queue without removing it, log the value in a set of values per task and then poll it and remove it from the queue. With these operations, it is guaranteed that the master can reconstruct all values that a failed worker thread considered. So, after a task failure, the master can add these values to the work queue again such that other tasks will redo the computations.

3.4.1 Work-stealing in cluster mode

In this section, we describe a simple, yet promising design to integrate work-stealing with Spark's cluster mode. We assume clusters in which one worker runs one executor. This is the case in Databrick's clusters.

The main problem with distributing work-stealing is that Spark's executors cannot communicate with each other. Therefore, we choose a communication free approach in which the tasks share work only with other tasks on the same executor. The work is statically partitioned in between multiple executors.

We cannot control how Spark schedules tasks on its executors. The tasks for the join algorithm could be co-located on a single executor, balanced evenly between all of them or could be distributed over multiple executors in any fashion between these two extremes[8]. We need to use the slots assigned to our tasks as best as possible. To archive this goal, we would like to know how many tasks were scheduled on each worker. With this knowledge, we can split the workload between executors such that each of them deals with $\frac{w}{e} \times t_e$ where w is the workload, e the number of executors and t_e the number of tasks on executor e.

Spark's scheduler offers the so called *Barrier Execution Mode*. In this mode, it schedules all tasks of a stage together; either all of them are scheduled at the same time or none are scheduled. When tasks are scheduled in this mode they have access to the location of all other tasks for this stage. Hence, we can determine on how many tasks were scheduled on how many executors from within each task. Furthermore, we can tell on which executor the current task runs.

We distribute our local work-stealing approach by requesting barrier scheduling for the worst-case optimal join operator. Then, we can use the work-stealing design as described for the local mode on each executor by partitioning the queue that holds all bindings for the first variable.

We experimented with two different partitioning schemes for the queue: round-robin and range based partitioning. However, both schemes lead to similar run-times and behaviours. We choose the round-robin partitioning because it is generally more skew resilient.

To conclude, we distribute work-stealing in Spark by partitioning the work queue over all executors with respect to the number of tasks assigned on each of them.

This approach has two drawbacks.

First, we use the barrier mode which requires Spark to find as enough available resources to schedule all tasks at the same time. This is not a huge issue in our experiments where we run one join at the time. However, it could be difficult to find enough open spots in a busy production cluster. In particular, if it runs workloads with many small tasks.

Second, this scheme does not manage skew between executors. If it turns out that the part of the work-stealing queue assigned to any executor requires significantly more work than to the other executors, then this executor will dominate the overall run-time of the whole algorithm. We establish if this is a problem in practice in our experiments **??**.

[8]Spark's standalone scheduler default behaviour is to schedule the tasks as evenly spread over all executors as possible. Alternatively, the standalone mode offers the possibility to consolidate all tasks as much as possible on a single worker. This can be controlled by the setting `spark.deploy.spreadOut`.

4 GraphWCOJ

4.1 Combining LFTJ with CSR

For our graph pattern matching specialized Leapfrog Triejoin version we choose CSR (see section 2.4) as backing data structure. This data structure is typically used for static graphs and we show that it is a great match for LFTJ. In this section, we shortly describe the implementation of a CSR based *TrieIterator*, point out the differences between this new version and a column based *TrieIterator* (as described in section 2.2.1) and conclude with an experiment demonstrating the power of this optimization.

The implementation of a CSR based *TrieIterator* is straightforward except for one design change: instead of using the vertice identifier from the graph directly, we use their indices in the CSR representation. This change is rather minor because it can be contained at any level by using a hash map for translation, e.g. in the *TrieIterator* itself, in the *LeapfrogTriejoin* or at the end of the query by an additional mapping operation.

In the current system, the translation is performed by the LFTJ implementation to allow easy integration into other projects. However, it is possible to work on the indices throughout the whole system to safe the translation costs.

We now outline how to implement each of the *TrieIterator* methods, under the assumption that all vertices have outgoing edges. Then, we drop this assumption and explain the necessary changes. The creation of the CSR data structure itself is described in section 6.4.

We recall that a CSR uses two arrays to represent the graph edge relationship; the *AdjacencyLists* array and *Indices* array. The first one stores all ajacency lists in one direction (outgoing or incoming) as one concatenated array. The second stores indices into the first array, e.g. the value 5 at the position 1 means that the adjacency list for the second vertice starts at the 5th position in *AdjacencyLists*.

The vertical component of the *TrieIterator* consists out of the *open* and *up* methods. Both of them control on which of the two CSR arrays the iterator is operates. For the first level, it uses the *Indices* array and on the second level the *AdjacencyLists* array.

The *open* method does nothing when the first level is opened. When the second level is opened, it positions the iterator at the first element of the adjacency list. This position is given by the index in *Indices* where the first level is positioned.

Both methods use only a constant time. This differs from the *open* method of the array based *TrieIterator*. This method needs to find the number of second level elements when the second level is opened; it does so by counting the number of occurrences of the first level key in the first column. Additionally, both vertical component methods have more book keeping overhead for the array based implemenation.

The linear component of the *TrieIterator* is made of the functions: *key*, *atEnd*, *seek* and *next*.

The *key* and *atEnd* method are both only returning values computed by *next* or *seek*. They do not differ for the CSR based and array based *TrieIterator*.

The *next* method of the CSR based *TrieIterator* only changes at the first level. While the array based method needs to use the *seek* method to find the next higher value because it needs to jump over all entries of the same value as the current key, the CSR based value can simply increase the iterator position by one.

The *seek(key)* method exhibits the biggest possible performance improvement on the first level. For the array based version, we need to use a binary search to find the *key*. A CSR allows us to jump to the correct position in constant time because the *key* parameter is the correct index into

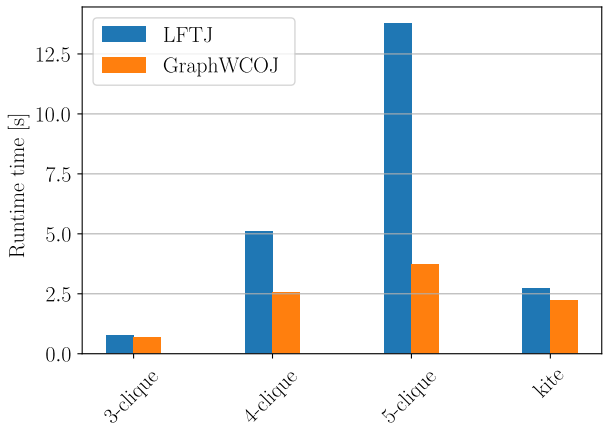


Figure 16: Run time of LFTJ and GraphWCOJ backed by CSR for multiple queries on `SNB-sf1`.

the array.

To resolve the assumption of no empty outgoing adjacency lists, we adapt *open*, *next* and *seek* to skip source positions without outgoing edges. This is easy to detect because then $Indices[x] = Indices[x+1]$. We can skip these cases by simple linear search until we find a valid position. This solution is sufficient because there are only a few vertices with no outgoing edges in real-world graphs. Therefore, this linear search does not majorly influence the run time.

The *TrieIterator* implementation based on CSR is much faster than the column based iterator; mainly due to the fact that the *seek* method on the first level can be implemented in $\mathcal{O}(1)$, instead of $\mathcal{O}(\log n)$. This optimization has huge potential because these searches are the most costly operations for a column based *TrieIterator* [18]. Note that searches on the second level are fast, due to the fact that most graphs have a low outdegree (see section 2.5).

Additionally to this advantage, CSR based *TrieIterator* do less bookkeeping because they support only 2 levels and spent nearly no time on processing *atEnd* for the second level, while a column based *TrieIterator* needs to calculate the number of outgoing edges for each source vertice in its *open* method, to allow a fast *atEnd* method.

We conclude that CSR based *TrieIterator*'s are a promising match for LFTJ and graph pattern matching. The improvements of this optimization can be seen in fig. 16. It demonstrates an up to 2.6 speedup over a column based LFTJ. We also see that the optimization has a stronger impact on queries with more edges and vertices, e.g. `5-clique`. For a more thorough evaluation refer to the experiment section 7.

4.2 Exploiting low average outdegrees

It is well-known that most real-world graphs have a low average outdeegree, mostly far below 200. This leads to the hypothesis that the intersection of multiple adjacency lists is small, e.g. below 10 in many cases.

We can exploit this fact by materializing the intersections in the *LeapfrogJoins* directly in one go; instead of, generating one value at-the-time in an iterator like fashion as described in the original paper [51]. We believe this to be beneficial because it allows us to employ a simpler intersection algorithm which builds the intersections touching each adjacency list only once, instead of, multiple times with yielding like the original *LeapfrogJoin* (see algorithm 1).

We structure the remainder of this section as follows. First, we shortly reiterate the most important facts about *LeapfrogJoins* for this chapter. Second, we analysis the intersection workload in terms of input sizes and result size to confirm our hypothesis and gain valuable insights to choose the best intersection algorithm. Third, we explain the algorithm we chose based on the analysis. Fourth, we point out differences to the original Leapfrog Triejoin. Finally, we present a short experiment showing the performance gains of this optimization.

LeapfrogJoins build the intersection between multiple adjacency lists. This is done in an iterator-like fashion in their *leapfrog_search* method by repeatedly finding the upper-bound for the largest value in the lowest iterator. This algorithm is asymptotically optimal for the problem of n-way intersections. However, we believe that it is (1) to complex for small intersections and (2) should generate all values at once instead of one-by-one to improve performance on real-world adjacency lists.

To determine the best algorithms to build the n-way intersection in the *LeapfrogJoins*, we run some experiments to characterize the workload. Towards this goal, we log the size of the full intersection, the size of the smallest iterator participating and the size of the largest intersection between the smallest iterator and any other iterator on 5-clique queries on `SNB-sf-1`. Figure 17 depicts these metrics as cumulative histograms. In the next paragraphs, we point out the most important observations in each of these graphs.

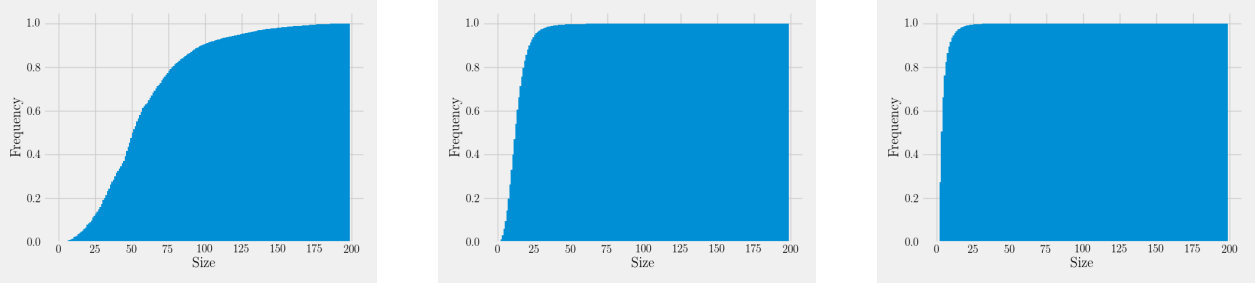
Figure 17a shows the size distribution of the smallest iterator, as to be expected for a social network graph, the outdegree is between 1 and 200. We do not see the long-tail distribution typical for power-law graphs because we choose the smallest iterator out of 5 and even though there are vertices with a much higher outdegree, the chance of encountering 5 of these in a single intersection is small. We note that in 80% of all cases the smallest iterator has a size lower than 80 and above that the distribution slowly increases to 100%

Figure 17b illustrates the size distribution of intersecting the smallest iterator with any other iterator, such that the intersection is maximal. We choose this specific metric to motivate one of our design choices later on. As for the smallest iterator, some of these intersections are as big as 200 but most of them are much smaller. However, unlike for the smallest iterator metric, 80% of the intersections contain less than 21 elements and the frequency increases to 100% in a steep curve.

This last observation is even stronger for the size of the total intersection (fig. 17c): the size is less than 5 in 80% of all intersections and increases similarly steep to 100%. The maximum is lower than 200.

These observations confirm our hypothesis that the size of the intersections is small (below 5) and do not show the same long-tail distribution as the whole social network graph. Hence, we can materialize them without running at risk of building big intermediary results.

Furthermore, the experiment shows that optimizing by taking iterator sizes into account is worthwhile but only for the smallest iterator because once we start with the smallest iterator the further intersections are small (below 21) in the vast majority of all instances.



(a) Smallest iterator

(b) Largest intersection including smallest iterator

(c) Total intersection

Figure 17: Cumulative histograms of total intersection sizes, largest intersection with the smallest iterator and any other, and size of the smallest iterator participating in `5-clique` on `SNB-sf-1`.

In the coming paragraphs, we detail how to build the n-way intersection of multiple adjacency list such that we gain performance by better use of data-locality than original the *LeapfrogJoin*. We choose to use pairwise intersections over multi-way intersection algorithms for their simpler memory access patterns; they touch only two lists at-a-time instead of all lists interchangly.

From our analysis, we conclude that the final intersection size is strongly dependent on the smallest iterator and that the intersection of the smallest iterator with any other iterator is close to the final size. These insights translate into two design decisions.

First, we start with the smallest iterator[9]. However, we do not take the sizes of any other iterators into account because the effort for sorting the iterators by size would not pay off.

Second, we use two different tactics to build the pairwise intersections. The first intersection between two iterators is built *in-tandem*, where we seek the upper bound of the higher value in the smaller iterator.

After this first intersection, the intermediary result is quite small. Therefore, we use the simpler scheme of linearly iterating the intermediary and probing the iterator by binary search with fall-back to linear search.

Finally, we point out a few exceptional cases and pitfalls for implementors:

- If all iterators of the *LeapfrogJoin* are on their first level, the intersection is near |V|. In this case, we fall back to the original *LeapfrogJoin*.
- We use an array to materialize the intersections because Scala collections are slow. Instead of deleting elements, we replace them with a special value.
- Allocating a new array for every *LeapfrogJoin* initialization is costly. We estimate the size of the intersection by the size of the smallest iterator and reuse the array whenever possible. We use a sentry element to mark the end of the array.

Figure 18 shows that the new algorithm performance slightly better than the original on fast running queries on the `SNB-sf1` dataset. We believe that this has three possible reasons.

[9]We take advantage of the fact that CSR allows us to determine the size of iterators cheaply (see section 2.4)

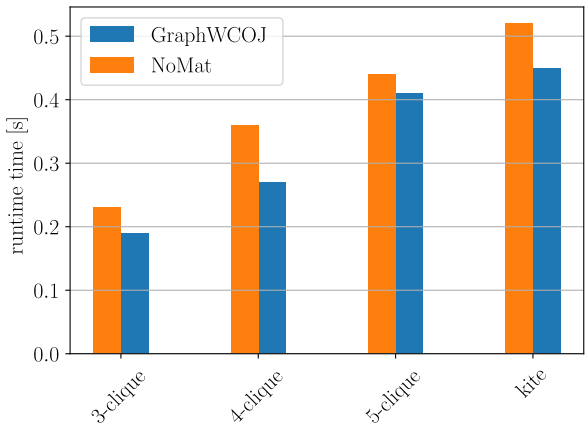


Figure 18: Runtimes of `GraphWCOJ` with and without *LeapfrogJoin* materialization enabled for different queries on `SNB-sf1`.

First, due to better cache usage. Most of the searches in the adjacency lists are linear searches; our binary search falls back to a linear search if the remaining list is small. Materializing the intersections means repeating linear searches for increasing elements on the same two arrays until they are fully processed. While the original algorithm repeats linear searches over all adjacency lists, interupted from filtering against first level iterators and does so only until one value is found; it then has to return to the same lists for the next elements.

Second, the *LeapfrogJoin* generates a single value, then yields control to other parts of the *LeapfrogTriejoin* algorithm and later touches the same adjacency lists again to generate the next value. Our approach touches the adjacency lists exactly once per *LeapfrogJoin* initialization and condenses the intersection into a much smaller array. This array is more likely to stay cached while the other parts of the *LeapfrogTriejoin* do their work.

Third, by starting with the smallest adjacency list of the intersection we get less *seek* calls to iterators over bigger adjacency lists.

We show the results of more experiments in section 7.4.

5 Optimizing a Leapfrog Triejoin in Scala

A simple, idiomatic Scala implementation of the Tributary join is not able to beat Spark's *BroadcastHashjoin* on any other query than the triangle query. Hence, we report on how to optimize the join. After, we are able to beat Spark's *BroadcastHashjoin* on nearly all queries and datasets. We report measured run-times for the unfiltered 5-clique on the Amazon0601 dataset for different optimizations in table 4. In total, we improved the WCOJ running time from 316.5 seconds to 54.1 seconds.

We discuss the optimization in categories: Leapfrog Triejoin specific, binary search specific, Spark related, Scala related and general. We conclude the section with some changes we tried that do not improve performance.

Binary search specific optimizations become a category on its own because the sorted search is the most expensive operation in the Tributary join. According to profiler sessions, the join spends more than 70% of its time in this method. This result is in line with the observation that 'in the Tributary join algorithm, the most expensive step is the binary search' from [18].

Category	Optimization	Runtime
NA	Baseline	316.5
Scala	Custom insertion sort instead of Scala's *sort* method	310.6
Scala	Maps instead of linear lookup of *LJ*'s and *TI*'s in *LFTJ*	171.6
General	Factor out computed values which are reused in *TI*	153.7
Binary Search	Linear search after galloping and binary search	138.0
Scala	Arrays instead of maps for *LJ*'s and *TI*'s in *LFTJ*	100.5
Scala	while loop instead of foreach loop in *LFTJ*	85.7
Scala	use of *private[this]*	84.3
Scala	use of *@inline* annotation	82.4
Spark	direct array access of input relationships	76.9
General	strength reduction of modulo operations	68.9
Binary search	linear search shortcut before galloping search and binary search	64.0
Binary search	less branches in binary search	58.9
Binary search	removing galloping search	56.4
LFTJ	no sorting in *LF init* method	54.1

Table 4: Optimizations to the LFTJ algorithm in Scala and their runtimes on the unfiltered 5-clique query on the Amazon0601 dataset. *LFTJ*, *LF* and *TI* refers to the *LeapfrogTriejoin*, *LeapfrogJoin* and *TrieIterator* component of the Leapfrog Triejoin algorithm.

We applied one LFTJ specific optimizations.

The *LeapfrogJoin.init* method is originally described to sort its *TrieIterators* so the method *leapfrogSearch* knows the position of the largest and smallest iterator (see section 2.2.1). However, the method can be improved by avoiding to sort the *TrieIterators*. We can start moving the *TrieIterator* without sorting them and arrive at an ordered array in $\mathcal{O}(n)$ steps with n defined as the size of array. This approach improves over the original algorithm in two ways: (1) it starts moving the *TrieIterators* to their next intersection immediately without sorting them first and (2) orders the array in fewer steps than traditional sorting algorithms.

To implement this we find the maximum value for all iterators and store the index in p. Then we move the *TrieIterator* at $p + 1$ to the least upper bound of the maximum value (by calling *seek*) and store the result as the new maximum. We proceed with this process, wrapping p around when it reaches *iterators.length*, until p equals the original maximum index. Now, we are either in a state in which all *TrieIterators* point to the same value or we arrived at a state in which the iterators are sorted by their key value. In the first state, the *LeapfrogJoin* is initialized; the array is sorted and the first value of the intersection found. In the other possible state, the array of *TrieIterators* is sorted and we can use the original *leapfrogSearch* (algorithm 1) to find the first key in the intersection. can proceed as in the original *LeapfrogJoin.init* method.

Table 4 mentions two optimizations for the sorting in the *LeapfrogJoin init* method. The one described above is the second one. For the first one, which is no completely replaced by the second, we used a self-written insertion sort which is faster than Scala's array sort. Scala's array sort is slow because it copies the array twice and casts the values to *Java.Object* such that it can use Java's sorting methods. An insertion sort is a asymtopical suboptimal algorithm but a good option given that a *LeapfrogJoin* normally operates on less than 20 *TrieIterators*.

The binary search is the most expensive operation of the Leapfrog Triejoin. Hence, special attention needs to be paid while implementing it. Our most important optimization is to change to a linear search once we narrowed the search space to a certain threshold. We experiment with different thresholds and show the results in section 7.2.

We directly perform a linear search if the search space is smaller than the threshold from the beginning (see 12th optimization in table 4).

Another important optimization is to avoid unnecessary if-statements in the loop of the binary search, e.g. the implementation on Wikipedia and many other example implementations use an if-statement with three branches for smaller, bigger and equal but two branches for greater than and less-or-equal suffice for a least upper bound search.

A similar optimization can be applied to a linear search on a sorted array: intuitively one would use the while-loop condition $array(i) > key \wedge i < end$ with *key* being the key to find the least upper bound for, i the loop invariant and *end* the exclusive end of the search space. Anyhow, it is faster to check for $key > array(end - 1)$ once before the loop and return if this is the case because the value cannot be found in the search space. This obviously circumvents the main loop of the linear search; additionally, it simplifies the loop condition to $array(i) > key$.

The impact of this optimization is shown in the 13th row of table 4.

The Spark infrastructure uses the interface *ColumnVector* to represent columns of relationships. The implementation *OnHeapColumnVector* is a simple wrapper around an array of the correct type with support for *null* values and *append* operations. First, we used this data structure to represent our columns but we could see a clear increase in performance by replacing it by an implementation that exposes the array to allow the binary search to run on the array directly. This is likely due to saving virtual function calls in the hottest part of our code. Table 4 shows the results of this change in row 11.

We found many standard optimizations and Scala specific optimizations to be really useful. These are the optimizations that brought the biggest performance improvements. However, they are well-known, so we mention them only in tabular form 4. For Scala specific optimizations one can find good explanations at [20].

Apart from the aforementioned very useful optimizations, we investigated multiple other avenues in hope for performance improvements which did not succeed, we list these approaches here to save others the work of investigating:

- reimplement in Java
- use of a Galloping search before the binary search
- unrolling the while-loop in *LeapfrogTriejoin* state machine (see algorithm 2)
- predicating the *action* variable in *LeapfrogTriejoin* state machine

Finally, we believe that code generation for specific queries that combines the functionality of *LeapfrogTriejoin*, *LeapfrogJoin* and *TrieIterator* into one query specific function would lead to noticeable performance improvements. The reason for this belief is that our implementation takes about 3.46 seconds for a triangle query on the Twitter social circle dataset while a triangle query specific Julia implementation, of a colleague of ours, needs only half a second. The main difference between our implementation and his are: the language used (Julia is a high-performance, compiled language) and the fact that his implementation has no query interpretation overhead but cannot handle any other query than the triangle query.

However, a code generated Leapfrog Triejoin is out of scope for this thesis, also, we are aware of efforts by RelationalAi to write a paper about this specific topic. We are looking forward to seeing their results.

6 Spark integration

6.1 User interface

```
val sparkSession = SparkSession.builder
  .master("local[1]")
  .appName("WCOJ-spark")
  .getOrCreate()

// read a dataframe
val df = sparkSession.read
  .option("inferSchema", "true")
  .csv("/path/to/edge/relationship")

// df needs columns called ‘edge_id‘, ‘src‘ and ‘dst‘
// Use WCOJ to find a triangle pattern
val triangles = df.findPattern(
  """(a) - [] -> (b);
    |(b) - [] -> (c);
    |(a) - [] -> (c)""".stripMargin,
  Seq("a", "b", "c")
)
triangles.limit(10).show()
// Shows a dataset with 3 columns: ‘a‘, ‘b‘, ‘c‘ being node ids
```

Listing 1: Example usage of a WCOJ to find triangles in graph.

As one can see in line 16 of listing 1, we support a clean and precise DSL to match patterns in graphs. This DSL is inspired by GraphFrames [21]. The user can define a pattern by its edges, each edge is written as *(a) - [] -> (b)* where a is the source vertice and b is the destination, multiple edges are separated by a semicolon. A connected pattern is expressed by defining multiple edges with the same source or destination. One should be aware, that a named source or destination is not guaranteed to be a distinct element in the graph, e.g. *(a) - [] -> (b); (b) - [] -> (c)* could be a linear path of size two or a circle between a and b; in the second case a and c are the same element. The reader might wonders, why we chose to stay with the GraphFrame syntax for edges of *- [] ->*, although, we could have went with something simpler, like *->*. However, sticking to the more verbose syntax allows us to include labels inside of the squared brackets in future extensions, e.g. for our stretch goal of integration with CAPS.

The second parameter to *findPattern* allows the user to specify the variable ordering used in the WCOJ algorithm. Furthermore, the user interface takes multiple optional arguments, e.g. to apply to common filters to the output of the result. The filters are *distinctFilter*, ensuring that each vertice can occur only as binding for one variable, and *smallerThanFilter* to allow only output bindings were the values decrease with regards to the specified variable ordering, e.g. the binding *[1, 2, 3]* but not *[2, 1, 3]* for the triangle query above. We experienced that these queries are typical for graph queries and that the performance greatly benefits from pushing them into the join. Implementing the possibility to push general filters into the join would be a valuable addition but we decided against it because it a pure engineering task.

6.2 Integration with Catalyst

We integrated our WCOJ implementation into Spark such that it can be used as function on *Datasets*. Therefore, we build all components necessary to execute a WCOJ in Spark's structured queries, provided by Catalyst (see section 2.1.3). First of all this is logical plan representing a worst-case optimal join. Then a strategy to convert this logical operator into multiple physical operators. One physical operator executes the join. In between, this operator and the graph edge relationships, we execute another physical operator that materializes the graph edge relationships into a data structure that can support a *TrieIterator* interface (see section 2.2.1). The integration itself is quite straightforward due to Catalysts extendability. We explain it in the next sections. First, we highlight some limitations of our integration into Catalyst.

It is not in the scope of this work to integrate WCOJ's into the SQL parser of Spark. Hence, WCOJ can only be used by Spark's Scala functional interface and not through Spark's SQL queries.

We do not integrate it into the query optimization components of Catalyst, e.g. we do not provide rules or cost-based strategies to decide when to use a WCOJ or a binary join. It is up to the user to decide when to use a WCOJ or a binary join. However, our integration allows the user to intermix these freely. The reason for this decision is, that at time of writing no published paper existed that systematically studies which queries benefit from WCOJ's in general, nor, does research exist that studies the combination of WCOJ's and binary joins. Only a month after we decided on our scope for Spark integration Salihoglu et al. published an arXiv paper [39] that tackled these problems for the first time. The lack of peer-reviewed papers and the high complexity of the arXiv paper clearly illustrate that deeper integration with Spark's optimizer is out of scope for this thesis.

6.3 A sequetial linear Leapfrog Triejoin

For this section we assume that the reader is familiar with the background section about Catalyst (see section 2.1.3) and LFTJ (see section 2.2.1) where we explain the components of Catalyst planning phase and the requirements of a Leapfrog Triejoin. In the current section, we outline how to satisfy the requirements of LFTJ within and help of Catalysts structured plans.

Our baseline implementation of the Leapfrog Triejoin is a sequential implemenation, i.e. it is not distributed. Therefore, all representations of the edge relationship have only a single partition which the join operates upon. In Spark this partitioning is called *AllTuples*. We enforce sequential execution of the complete Spark plan in our experiments by setting the number of executors to 1.

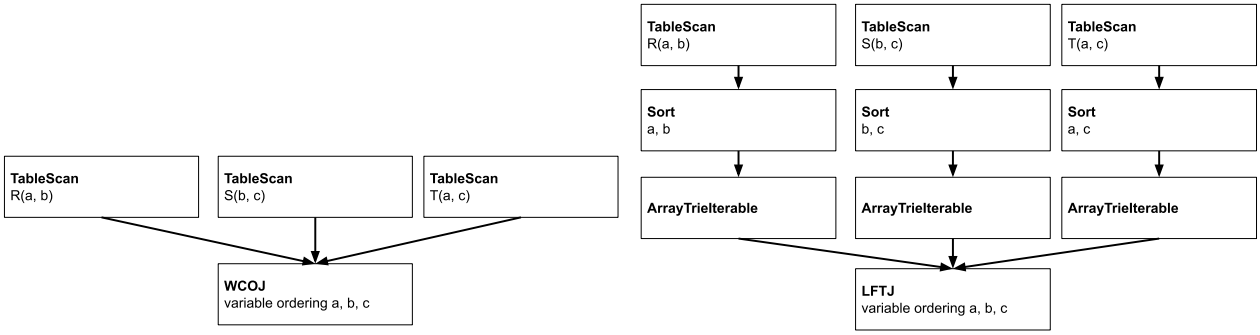
In the first phases of Catalyst query compilation process, the query plan is represented by logical operators. Integration into this phase only requires us to build a logical operator to represent the WCOJ join. The only thing that we need to describe for this logical operator are the number of children. A LFTJ can have 2 or more children, one for each input relationship.

The logical and physical plan for the triangle query is shown in fig. 19.

The strategy to translate the logical plan into a physical plan has two tasks. First, simply translating the n-ary logical plan into n-ary physical plan that executes the LFTJ with the children as input. Second, introducing a physical operator per child which materializes the RDD into a sorted, columnar array representation to support a *TrieIterator* interface.

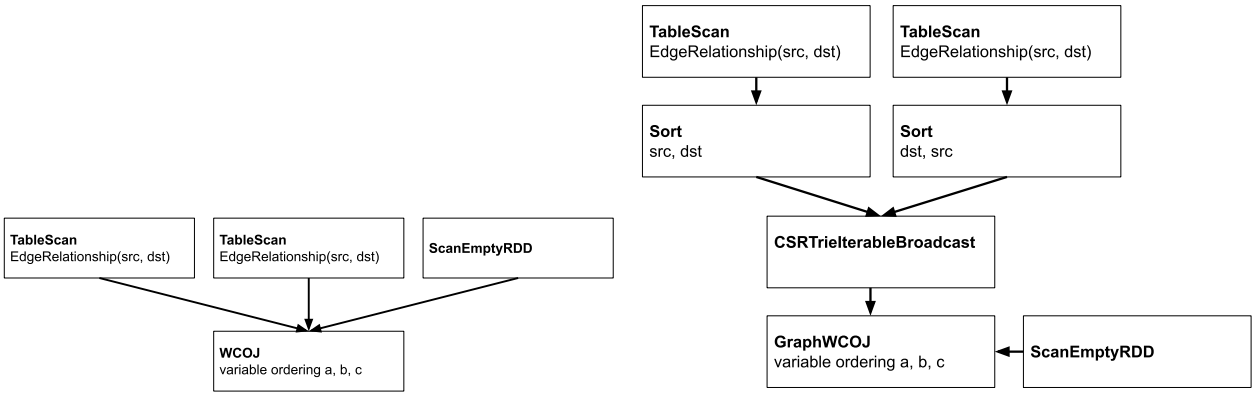
The first physical operator is straightforward to implement. It simply executes a *LFTJ* over the *TrieIterators* provided by the children. Given that each child has only one partition and there are no parallel operations, the algorithm can be implemented exactly as described in section 2.2.1.

The second physical operator translates the linear interator interface offered by Spark for RDD's



(a) Logical plan (b) Physical plan

Figure 19: Catalyst plans for the triangle query using a Leapfrog Triejoin.



(a) Logical plan (b) Physical plan

Figure 20: Catalyst plans for the triangle query using GraphWCOJ.

into a *TrieIterator* interface. In particular, it needs to offer a *seek* operation in $\mathcal{O}(\log N)$. To support this interface the operator requires its children to be sorted; this requirement can be fulfilled by Catalyst standard optimization rules. Then it takes the sorted linear iterator and materializes it into a column-wise array structure. Given this data structure, the *TrieIterator* can be implemented using binary search (as described in section 2.2.1).

6.4 GraphWCOJ

The integration of GraphWCOJ is quite similar to the one of LFTJ. Figure 20 shows the logical and physical plan constructred by our integration. There are three main difference: GraphWCOJ requires only two children for the input relationships, the children materialize the input relationships in a CSR data structure (see section 2.4) and we support parallel execution of the join (which requires a third child) and broadcasting of the CSR data structure. We address these differences in order.

Our GraphWCOJ operators need only two materialized version of the input relationship. This is because in graph pattern matching the joins are self-joins on the edge relationship. This relationship has two attributes. The LFTJ requires that its input relationships are sorted by an lexicographic sorting over the variable ordering. To support all possible variable orderings, we

need the edge relationship sorted by src, dst and dst, src. Hence, we need to separate, materialized versions of the edge relationship. However, we never need more materialized relationships because all *TrieIterators* can share the same underlying datastructures.

GraphWCOJ uses a CSR representation of the edge relationship (see section 4). Hence, we need to build two CSR representations. One with its *Indices* array build from the *src* attribute and the *AdjancencyLists* array build from the *dst* attribute. The other, with *Indices* from *dst* and *AdjacencyLists* from *src*. Next, we describe how to build these CSR's from two linear, sorted, row wise iterators as provided by Spark for the two child relationships.

First, we note that it is necessary to build both compressed sparse row data structures in tandem. This is because some vertices in the graph might have no outgoing or incoming edges. That means some vertice ID's do not occur in the *src* or *dst* attributes of any tuple. Therefore, the *Indices* arrays of the two CSR structures would differ if they are build from either *src* and *dst*, e.g. they could have different length. However, if this is the case, it is not possible to use both CSR's together in a single join.

To allow building the CSR's in-tandem, we introduce an *AlignedZippedIterator*. The *next* method of this iterator is shown in algorithm 3. It zips two iterators of two-tuple elements and aligns them on the first component, e.g. edges with *src* as first component and *dst* as second component. The zipped iterator emits triples where the first component is the aligned first element of both underlying iterators and the other two elements are the second components of both iterators. If the two iterators have different numbers of elements with the same first component, we advance only one iterator and fill the missing component in the emitted triple with a placeholder until the first component of both iterators aligns again.

Algorithm 3: *next* method of an *AlignedZippedIterator*.

```
1  if iter1.hasNext() ∧ iter2.hasNext() then
2  |  t1 ← iter1.peek()
3  |  t2 ← iter2.peek()
4  |  if t1[0] = t2[0] then
5  |  |  t1 ← iter1.next()
6  |  |  t2 ← iter2.next()
7  |  |  return (t1[0], t1[1], t2[1])
8  |  else if t1[0] < t2[0] then
9  |  |  t ← iter1.next()
10 |  |  return (t[0], t[1], -1)
11 |  else
12 |  |  t ← iter2.next()
13 |  |  return (t[0], -1, t[1])
14 else if iter1.hasNext() then
15 |  t ← iter1.next()
16 |  return (t[0], t[1], -1)
17 else
18 |  t ← iter2.next()
19 |  return (t[0], -1, t[1])
```

Given an *AlignedZippedIterator* over both input relationships, it is straightforward to build two CSR data structures. We consume the whole *AlignedZippedIterator*, for each element we append the 2nd and 3rd component to *AdjacencyLists* of the CSR's; while skipping placeholders. Whenever the first element of the three tuple changes, we append the current size of the *AdjaencyLists* buffers to the *Idices* arrays.

The final difference between the Spark integration for LFTJ and GraphWCOJ is that we build GraphWCOJ such that it can be run in parallel.

As argued in former chapters, we broadcast the edge relationship to all workers. The broadcast is supported by Spark's broadcast variables (see section 2.1.4) and Catalysts support to broadcast the execution of a physical operator.

Parallelism is introduced via the third child of our GraphWCOJ operators. It is an empty RDD with as many partitions as the desired level of parallelism. We schedule tasks by using the *mapPartitions* function of this empty RDD. Foreach partition, we run the WCOJ join backing its *TrieIterator* with the broadcasted CSR's and partition the data logical by one of the schemes described in section 3.

One of the main advantages of broadcasting the edge relationship to all workers is that we can reuse the same broadcast for all queries over the same graph. To support this in Catalyst, we introduce one additional physical operator which we call *ReusedCSRBroadcast* and a CSR broadcast variable cache maintained on the Spark master node.

The CSR broadcast variable cache is a simple dictionary with RDD ID's to broadcast variables of CSR structures. When, our system builds a broadcasted CSR structure, it registers the broadcast in the cache. Every time, we translate a logical WCOJ plan into a physical one, we check if the CSR for edge relationship has been broadcasted already if so, our strategy reuses this broadcast.

7 Experiments

TODO introduction

7.1 Setup

We run our experiments on machines of the type `diamond` of the Scilens cluster owned by the CWI Database Architecture research group. These machines feature 4 Intel Xeon E5-4657Lv2 processors with 12 cores each and hyperthreading of 2 (48 cores / 96 threads) Each core has 32 KB of 1st level cache, 32KB 2nd level cache. The 3rd level cache are 30 MB shared between 12 cores. The main memory consists of 1 TB of RAM DDR-3 memory.

The machines run a Fedora version 30 Linux system with the 5.0.17-300.fc30.x86_64 kernel. We use Spark 2.4.0 with Scala 2.11.12 on Java openJDK 1.8. In the majority of our experiments, we use Spark in its standard configuration with enabled code generation. We also tune the parameters for driver and executor memory usage (`spark.driver.memory` and `spark.executor.memory`) to fit all necessary data into main memory.

7.1.1 Algorithms

In our experiments we use 4 different join algorithms. Two of them are worst-case optimal joins. That is our Leapfrog Triejoin implementation and a graph-pattern matching specialized Leapfrog Triejoin developed in this thesis: GraphWCOJ. LFTJ is only run as sequential algorithm as a baseline against GraphWCOJ. We compare these two algorithms in section 7.4.

The other two algorithms are Spark's binary joins: *BroadcastHashJoin* and *SortmergeJoin*. We compare them against the sequential version of LFTJ and GraphWCOJ in section 7.3.

We adjust the `spark.sql.autoBroadcastJoinThreshold` parameter to control if Spark is using a *BroadcastHashJoin* or a *SortMergeJoin*.

Name	Variant	Vertices	Edges	Source
SNB	sf1		453.032	[34]
Amazon	0302	262,111	1,234,877	[35]
	0601	403,394	3,387,388	[35]
Twitter	sc-d	81,306	1,768,135	[35]
LiveJournal		4,847,571	68,993,773	[35]
Orkut		3,072,441	117,185,083	[35]

Table 5: A summary of all datasets mentioned in the thesis.

7.1.2 Datasets

We run the our experiments on multiple datasets from two different use-cases: social networks and product co-purchase. We motivate our choice in the next paragraph. Table 5 includes a list of all graph datasets mentioned throughout the thesis.

The SNB benchmark [34] generates data emulating the posts, messages and friendships in a social network. For our experiments, we only use the friendships relationship (`person_knows_person.csv`) which is an undirected relationship. Only edges of the kind $src < dst$ exist, we generate the opposing edges before loading the dataset, such that the edge table becomes truly undirected.

The benchmark comes with an extensively parameterizable graph generation engine which allows us to experiment with sizes as small as 1GB and up to 1TB for big experiments and different levels of selectivity. The different sizes are called scale-factor or `sf`, e.g. `SNB-sf1` refers to a Social network benchmark dataset generated with default parameters and scale-factor 1.

The Amazon co-purchasing network contains edges between products that have been purchased together and hence are closely related to each other [35]. This is a directed relationship from the product purchased first to the product purchased second, both directions of an edge can exist if the order in which products have been purchased varies.

The Snap dataset collection contains multiple Amazon co-purchase datasets, each of them containing a single day of purchases. We choose the smallest and biggest dataset from the 2nd of March and the 1st of June 2003 which we call them `Amazon-0302` and `Amazon-0601`.

We pick co-purchase datasets for evaluation because former work often concentrated on social networks and web crawl based graphs [18, 7] but [47] points out that the biggest graphs are actually graphs like the aforementioned Amazon graph containing purchase information.

To allow comparisons with former work, we run a subset of our experiments on the Twitter social circle network from [35]. This dataset includes the follower relationship of one thousand Twitter users; each of these follows 10 to 4.964 other users and relationships between these are included.

The `LiveJournal` graph represents the friendship relationship of a medium sized social network.

7.1.3 Graph patterns

In this section, we detail the graph patterns used throughout our experiments. Most of the queries are cyclic because that has been shown to be the primary use-case for WCOJ in former research [44, 18]. WCOJ's also have been successfully applied to selective path queries in [44, 32].

We apply filters to most of our queries to make them more realistic, e.g. a clique query does make more sense if it is combined with a smaller-than filter, which requires that the attributes are bound such that a smaller than b, smaller than c. Otherwise, one gets the same clique in all

Name	Parameters	Vertices	Edges
`triangle`	NA	3	3
`n-clique`	# vertices	n	$1/2 \times n \times (n-1)$
`n-cycle`	# vertices	n	n
`n-s-path`	# edges / selectivity	n	$n-1$
`kite`	NA	4	5
`house`	NA	5	9
`diamond`	NA	4	4
Filters			
`distinct`			
`less-than`			

Table 6: Summary of patterns and filters used.

possible orders, which not only takes much more time but is also most likely not the result a user would want.

We ensure that filters can be pushed down through or in the join by Spark as well as by the WCOJ to compare both algorithms on an equal basis. A complete list of all queries and filters used is shown in table 6. Figure 21 shows depiction of all graph patterns.

Patterns and filters are combined as follows. Cliques and the `kite` query use smaller than filters which require the bindings to increase in value according to the variable ordering. All other queries are run with a filter such that each of their binding must be distinct.

For a selective path query, we first select two sets of nodes with respect to the *selectivity* parameter. Then we search for all paths of a certain length according to the *edges* parameter, e.g. `4-0.1-path` finds all paths between two randomly selected, fixed sets of vertices of length 4. The sets of nodes contain roughly 10% of all input nodes and are not guaranteed to be intersection free.

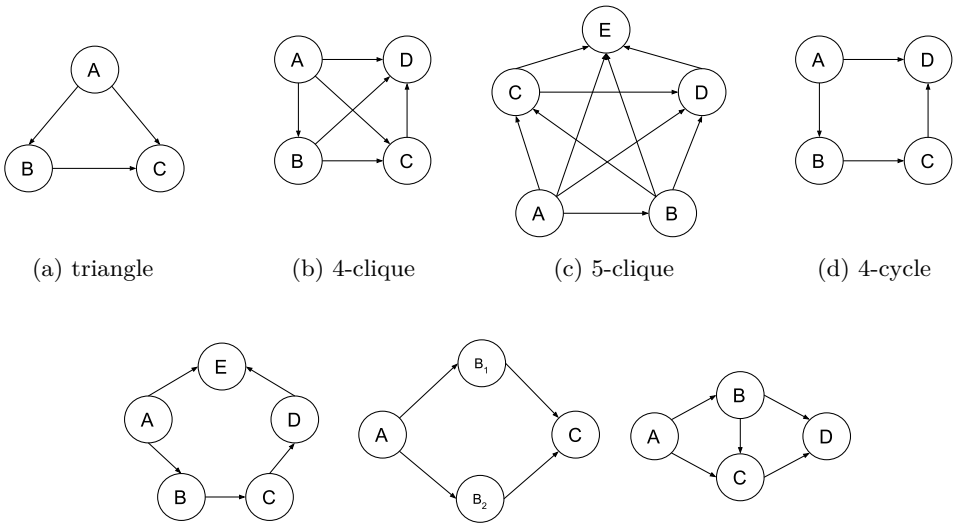
7.2 Linear search threshold

We run LFTJ and GraphWCOJ with different settings for the *linear search threshold*. As explained in section 2.2.1, we use a binary search to implement the *seek* method of the *TrieIterators*; it is also used for GraphWCOJ. It is well known, that a binary search can be optimized by ending it with a linear search on small search spaces because linear memory access patterns are cheaper than random accesses. The threshold gives the size of the search space from which to use a linear search instead of a binary search, e.g. a threshold of 40 means that the algorithm switches to a linear search once the search space is 40 numbers or less.

We note that LFTJ and GraphWCOJ could behave differently for the same threshold. This is because Leapfrog Triejoin uses a binary search for both levels of its *TrieIterators*, while GraphWCOJ only uses the binary search for the second levels; the first level is indexed in a CSR.

In this experiment, we vary the threshold between 1 and 1600 to determine the best value. These values are chosen such that 1 does not trigger any linear search and that 1600 does not improve the performance anymore (for LFTJ) and triggers no binary search for GraphWCOJ. We do so for the 5-clique query on the SNB-1 and on the Twitter dataset.

The results are shown in section 7.2. The optimum for LFTJ is around 200 while GraphWCOJ shows the best performance at 1600 and 800 for SNB-1 respectively Twitter. This means that GraphWCOJ performs best when there are nearly no binary searches. A threshold of 1600 triggers



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

(e) 5-cycle (f) diamond (g) kite

Figure 21: Queries used in our experiments.

no binary search in either of the datasets.

We also note that the effect on the performance of the Leapfrog Triejoin is bigger.

We explain the observations as follows: LFTJ does use searches on the first and second level of the *TrieIterators* while GraphWCOJ uses it only on the second level. Hence, the impact is bigger.

The optimal values are different because data of the levels is differently distributed. The first level lists nearly all vertices while the second level is made of adjacency lists which are more sparse. Hence, we assume that the linear searches on the first level are generally longer than the one on the second level; note that the threshold only gives an maximum length for linear searches but this is not necessarily a good indicator for the length of the performed search.

We tried to use different threshold values for the two levels in LFTJ. We choose the values 200 for the first level and 1600 for the second level because these are the optimal values according to our experiments. However, we note that no huge performance gain can be measured. This is most likely because the runtime is dominated by the searches on the first level. For simplicity, we do not use two different thresholds for LFTJ in any further experiments.

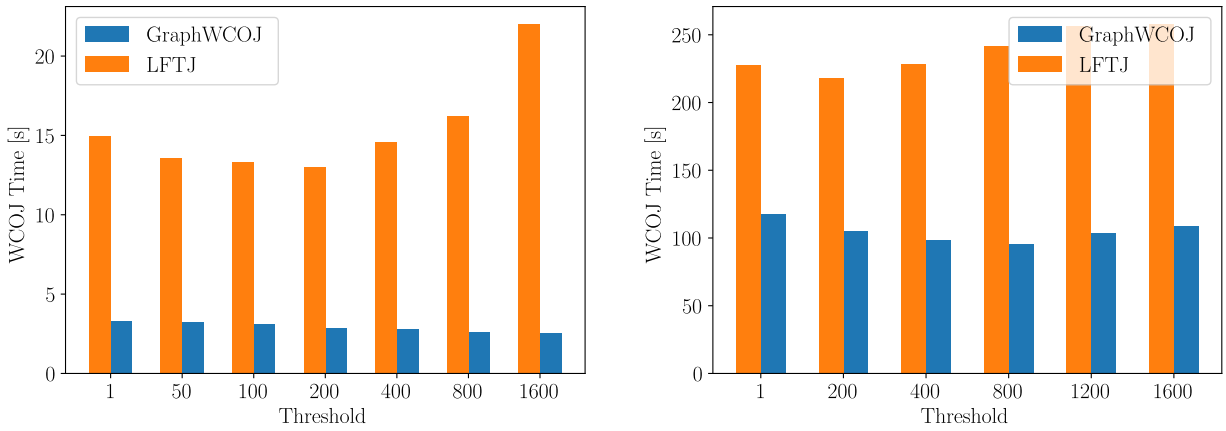
From this experiment, we conclude that the optimal threshold for LFTJ is 200 and 800 for GraphWCOJ. We choose 800 for GraphWCOJ because it is on the safe site: a binary search performance degrades less than the one of a linear search. We set these values accordingly in the all further experiments.

7.3 Baseline: `BroadcastHashJoin` vs `seq`

In this experiment, we compare the runtime of our sequential Leapfrog Triejoin implementation with the runtime of Spark's `BroadcastHashjoin`. Towards, this goal we ran all queries from table 6 on our three of our datasets: `Amazon-0302`, `Amazon-0601` and `SNB-sf1`.

We show our results in fig. 23. Section 7.3.2 analyzes the results.

Our experiment measures the time it takes to perform a `count` on the cached dataset using



(a) SNB-1

(b) Twitter

Figure 22: Runtime of WCOJs with different settings for the linear search thresholds.

`BroadcastHashjoin` and LFTJ. For `BroadcastHashjoin`, the time to run the whole query is reported. For `seq`, we report setup time and the time, it takes to run the join, separately. Setup time includes the sorting and materialization. This section is focused on comparing the runtimes excluding the setup time - rational given in section 7.3.1.

7.3.1 Experiment Setup

Question: Why do we compare against Spark's `BroadcastHashjoin` instead of `SortMergeJoin`?
Answer: Because even when all data is arranged in a single partition, for simple sequential processing, Spark schedules its `SortMergeJoin` to use a shuffle. A shuffle writes and reads data to and from disk. Hence, `SortMergeJoin` is much slower than a `BroadcastHashJoin`. We compared the algorithms on the `Amazon-0601` dataset for the `triangle` (8.1 seconds vs 58.9 seconds) and `5-clique` pattern (32.9 seconds vs 850.9 seconds). We assume that Spark is able to optimize its broadcasts when `local[1]` is used to start the Spark session because then Spark uses the driver as executor.

Question: Why do we exclude setup times from the WCOJ times?
Answer: Because system is meant to cache the readily sorted and formatted as CSR's and reuse it for multiple queries. We anticipate that this is necessary to benefit from WCOJ's in general.

Question: Is Spark's code generation a huge advantage for the `BroadcastHashjoin`?
Answer: Yes, we ran Spark without code generation for comparision on the `Amazon-0302` dataset for the `triangle` query and `5-clique`: with code generation Spark takes 3.1 and 4.2 seconds without 14 and 16 seconds.

7.3.2 Analysis

We are able to beat Spark's `BroadcastHashjoin` on all datasets and queries except `5-clique-lt` on `Amazon-0602`. Generally, we see that for `n-clique` patterns the speedup over Spark decreases for bigger n. This is due to the fact that many binary joins in a `n-clique` are actually semi-joins which to do not increase but decrease the size of intermediary results, e.g. for `5-clique` on `Amazon-0302` only 3 out of 9 joins lead to a bigger intermediary result.

The cycle query results are highly interesting because we see an increasing speedup for higher n on `Amazon-0602` but a decreasing speedup on `Amazon-0302`.

The `House` and `5-clique` pattern seem to be quite similiar - the `House` is a `5-clique` with two missing edges. However, as the count of their results indicates these two edges lead to dramatically different outcomes. Hence, their different timing and speedup behaviour.

The `Kite` pattern produces consistently the second highest speedup after the `3-clique`. Most likely due to the fact that a `Kite` is two triangles back-to-back.

The path query shows very different behaviour on the `Amazon` and the `SNB` datasets. This might be due to the different selectivity; it is extremely high on the co-purchase datasets and rather low on the social network benchmark. This different in selectivity is not surprising given that the `SNB` network fulfills the small world property, while the `Amazon` dataset relates products purchased together which naturally leads to multiple loosely connected, denser components.

Finally, we observe that all three datasets lead to quite different results which are most likely not comparable to each other without deeper research in the characteristics of the datasets themselves.

In particular, it becomes clear that co-purchase datasets and social network datasets must have very different characteristics. Although, `SNB-sf1` is much smaller than `Amazon-0601`, queries on it take a similar or even much more time, e.g. `5-clique` takes 14.21 seconds on the bigger dataset and 12.65 seconds smaller, even though, the result set is much smaller on `SNB-sf1`; `4-cycles` takes roughly 8 times longer on the small dataset and has a much bigger result set. In general, we see a higher speedup on `SNB-sf1`

7.4 LFTJ vs GraphWCOJ

In this experiment, we compare sequentials runs of the Leapfrog Triejoin and GraphWCOJ on the `Amazon`, `SNB-1` and `Twitter` datasets. We do not show the run-time of `Path` queries because GraphWCOJ is not able to run them[10].

We show the run-time of the queries on the different datasets in fig. 24. We present the performance of the Leapfrog Triejoin, GraphWCOJ and GraphWCOJ without the materialization optimization.

Comment for Peter and Bogdan: The materialization showed to be not as good of an optimization as I thought; in some cases it turns out to be slower. This has two reasons:

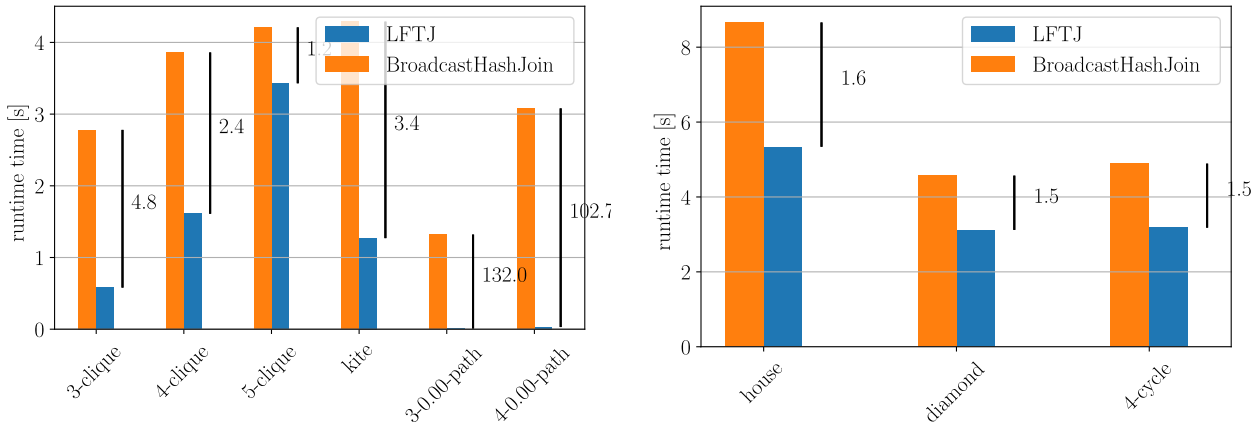
I found that it is faster to build the intersection my way because it causes less seek calls to the underlying iterators due to the fact that I start the intersection with the smallest list. So it does not improve computation because of better caching behaviour as I used to believe.

Then I experimented with the linear search thresholds and found that the algoritm gets faster when I use a higher threshold. Now, there are nearly no binary searches. Hence, the original algorithm got faster and the overhead that used to be avoided by my optimizations (binary searches) is gone. Therefore, the optimizations doesn't look great anymore.

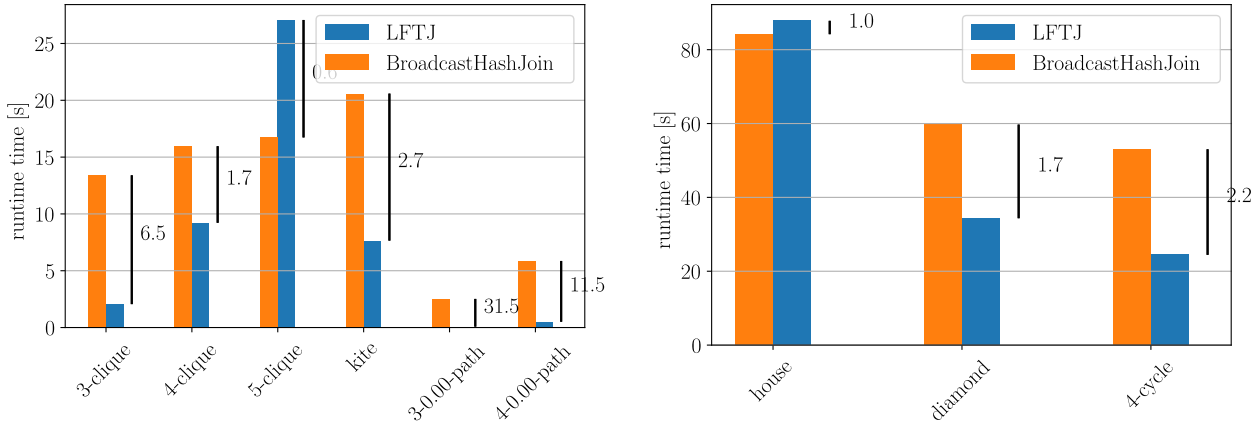
So far I'm not sure what to do about it and present the results as they are. In many cases, it's slightly faster, in some slower. In general, it is not a huge improvement anymore.

Observations We first compare the Leapfrog Join algorithm with the GraphWCOJ without the materialization optimization.

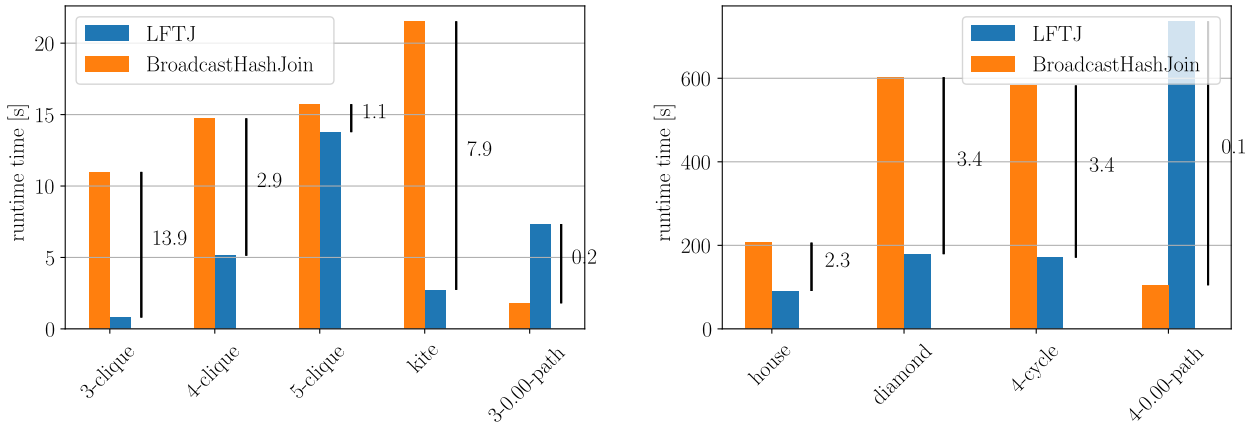
[10] `Path` queries require us to filter the input relationship before the join. This has not been implemented for GraphWCOJ due to time constraints.



(a) Amazon0302

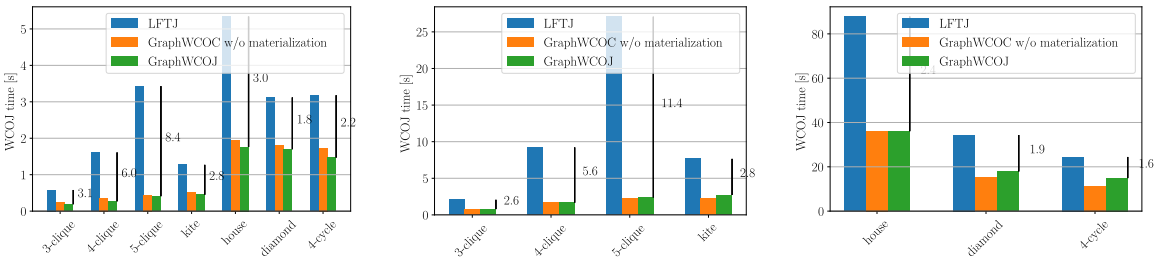


(b) Amazon0601



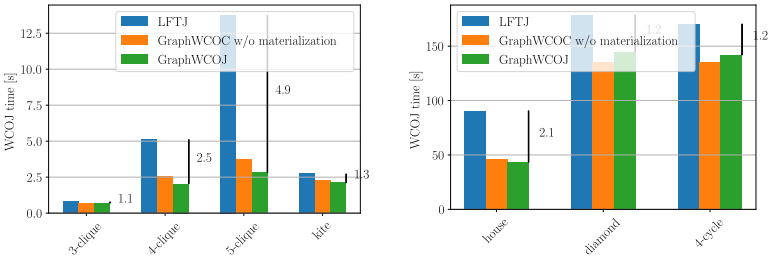
(c) SNB-sf-1

Figure 23: Runtime of a Leapfrog Triejoin and Spark's `BroadcastHashJoin`

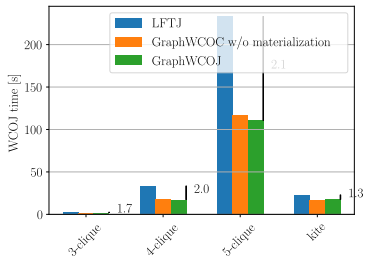


(a) Amazon-0302

(b) Amazon-0601



(c) SNB-1



(d) Twitter

Figure 24: WCOJ run time of LFTJ and GraphWCOJ on different datasets and queries. Diamond, house and cycle queries are not reported for Twitter because of their high run-time of over an hour.

Throughout all datasets, we see that GraphWCOJ is faster than a LFTJ for all queries. The biggest speedups are reached for 5-cliques and the lowest speedup for 4-cycles. The maximum speedup over all queries and datasets is 11.4 for a 5-clique query on `Amazon-0601`. The lowest speedup is 1.2 on a 4-cycle query on `SNB-sf-1`. Generally, the performance gain from using GraphWCOJ is higher for bigger and denser queries like clique queries.

The `Amazon` datasets show quite similar behaviour over all queries. `SNB-sf-1` and `Twitter` queries do not profit from using GraphWCOJ as much as the `Amazon` datasets.

The influence of the materialization optimization (described in section 4.2) is mixed; some queries can benefit, others cannot.

In general, queries that see the highest speedup from using GraphWCOJ, also benefit most strongly from materialization of the intersections, i.e. clique queries. 4-cycles, diamonds and kites are sometimes slower when we materialize the intersections.

The `Amazon-0302` dataset is the only dataset that shows increased performance for all queries.

`Amazon-0601` does not profit from materialization; the query run-time does not change for the clique queries and house query and increases for kite, diamond and 4-cycle.

On the other two datasets, we see improvements for the clique and house query but decreased performance on diamond and cycle.

7.5 Scaling of *Graph*WCOJ

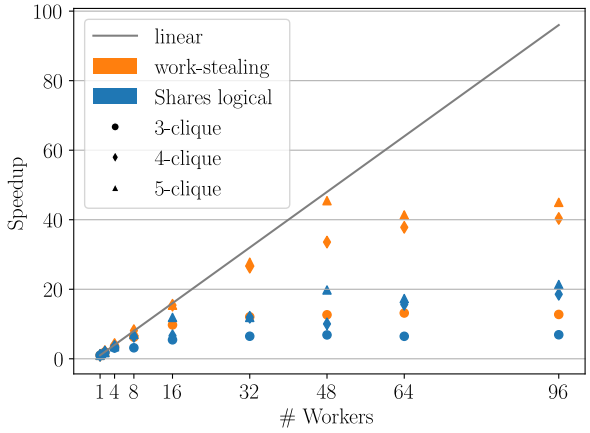
In this section, we aim to analyse and compare the scaling of GraphWCOJ using different partitioning schemes. Towards this goal, we run GraphWCOJ on datasets of different size namely `Twitter`, `LiveJournal` and `Orkut`. We compare two partitioning schemes: Shares and work-stealing. These are the two most promising schemed identified in section 3. The experiment is performed on 3-clique, 4-clique and 5-clique. 3-clique is the smallest of our queries. Therefore, it is most difficult to scale. 4-clique and 5-clique take much longer than 3-clique. Hence, it shows how query size influences the scaling. Also, it increases the job size for the *work-stealing* partitioning scheme.

7.5.1 Results

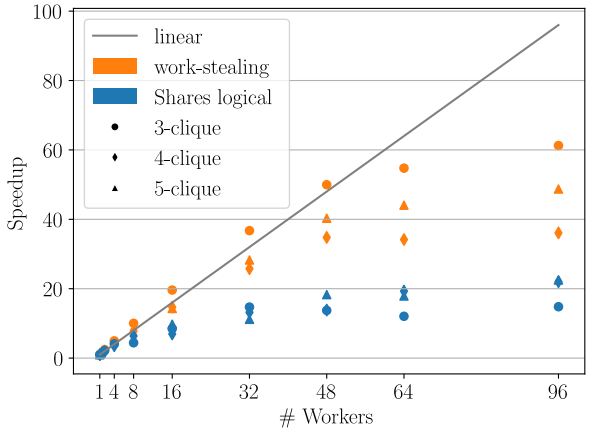
We first describe our expectations of the experiment outcome. We assume that scaling improves with the dataset size. Hence, we should see the highest speedups for Orkut, then LiveJournal and the lowest speedups for Twitter. Also, we expect the scaling to improve with the query size. Both hypothesis are grounded in the fact that more work to distribute often leads to stronger scaling. Additionally, we believe that *work-stealing* shows better scaling than Shares because it does not duplicate work. Finally, we have no clear cut expectations to the scaling behaviour of *work-stealing*. Theoretically, we could expect linear scaling for it because no work is duplicated, synchronization overhead is minimal and work balance should be given by the scheme. However, we measure on a quite complex hardware platform which complicates scaling behaviour.

First of all, we work on a machine with 4 sockets. This can influence scaling positively and negatively. Positively because adding more sockets means to add significantly more L3 cache (30 MB shared per socket). If we do not use all cores on a socket, each used core can use a bigger share of this cache. Negatively because each socket is in a different NUMA zone and the graph is not guarantued to be chached in all NUMA zone. Indeed, Spark shares the broadcasts for all tasks on a single executor. So there is only one copy in memory.

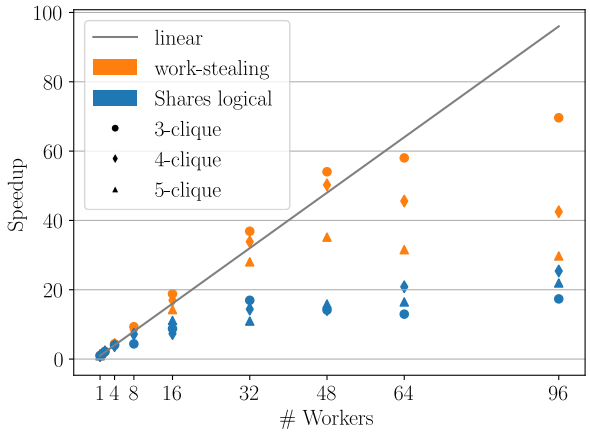
Additionally, we run on an Intel processor with hyperthreading. Hence, we can not expect linear



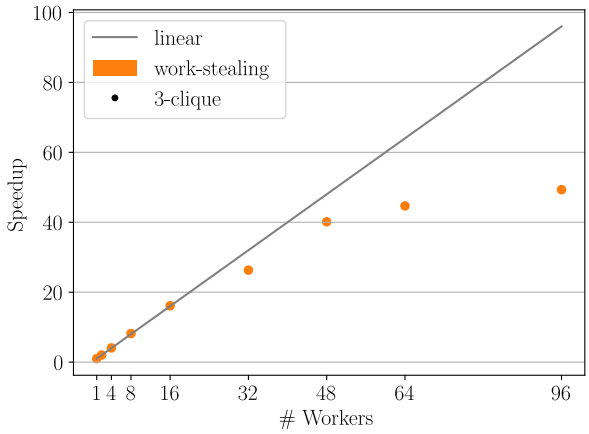
(a) Twitter dataset



(b) LiveJournal dataset



(c) Orkut dataset



(d) Twitter 3-clique join run-time only

Figure 25: Scaling behaviour of Shares and work-stealing on three different datasets and two different queries. The batch size parameter for *work-stealing* is chosen for balance between lock contention and worker skew: 50 for Twitter and 3-clique on LiveJournal, 1 for 5-clique on LiveJournal and 20 on the Orkut dataset.

speedup above 48 workers because after multiple threads will share resources and cannot be expected to reach the same performance as two cores.

To conclude, we expect sub-linear speedup for Shares and better but still sub-linear speedup for *work-stealing*. Anyhow, super-linear scaling in MapReduce like systems is not unheard of and could be possible on our machines.

We describe our observations per dataset; starting with Twitter. As expected, both partitioning schemes scale better when we increase the query size. For 5-clique, *work-stealing* exhibits near linear scaling up to 48 workers, while clique-3 reaches the maximum speedup of 6.22 for 8 workers. The highest speedup for 5-clique is 45 on 96 workers; clique-3 reaches its highest speedup with 13.2 on 64 workers. Shares lacks behind in scaling for both queries and all levels of parallelism. The best observed speedup is 21.3 for 5-clique and 96 workers.

We note that the 3-clique query does not scale well because there is not enough work. Therefore, the run-time is dominated by overheads, e.g. the time it takes until Spark starts the first job and the time it takes to finalize the query by Spark. The overhead calculated by $(queryEnd - queryStart) - (lastTaskCompleted - firstTaskScheduled)$ is roughly 0.13 seconds for all levels of parallelism and the whole query runs for 0.19 seconds on 48 cores. We depict the speedup archieved when we measure only the time spent with the join in fig. 25d.

The experiment on LiveJournal confirms our hypothesis that bigger datasets lead to better speedups; the highest observed speedup is 61.2 for *work-stealing* on 3-clique and 36.81 for Shares on 5-clique each with 96 workers. Also, we can confirm that Shares scales better on 5-clique than on 3-clique; with the exception of 32 workers. However, this is not the case for *work-stealing*. *work-stealing* shows better speedups on clique-3 than on clique-5. Nevertheless, *work-stealing* beats Shares on both queries and all levels of parallelism.

Additionally, we see three unexpected scaling behaviours for LiveJournal. First, super-linear scaling for 3-clique and *work-stealing*. We hypnotize that this is the fact because if the 32 processes are distributed over all 4 sockets they share in total 120 MB of L3 cache while a single process can use only 30 MB of L3 cache.

Second, the scaling of 4- and 5-clique is worse than for 3-clique. Our explanation is that 4- and 5-clique show skew even with *work-stealing*. This is because *work-stealing* partitions work along the first variable binding. Hence, the size of a single job in *work-stealing* is never smaller than the work of finding all bindings for a single first binding. That means a single long-running job towards the end of the *work-stealing* queue can result in a single task running longer than the others which delays the whole query.

We measure the time at which a task finishes. We define skew for *work-stealing* as the time between the average worker finishes and the time of the last worker to finish. In table 7 and table 8 we show the total skew and the percentage of skew in the whole query time for the LiveJournal respectively the Orkut dataset.

We note that the residual skew correlates with the scaling behaviour; the higher the percentage of skew in the whole query time the worse the query scales.

The total skew grows with the level of parallelism. We explain this as follows. Lets assume there is at least one job which takes significantly longer than most of the jobs in the work-stealing queue. This job is picked by a worker. When more executors are added which work on the remaining jobs, the likelyhood of this one big job adding significant skew raises because the other jobs are finished faster.

This experiment shows that the job size is not fine-grained enough for bigger queries and a high level of parallelism. We see that the skew can raise up to nearly half of the total run-time for the 5-clique query on Orkut. We address this issue in section 9.1.2. However, we want to point out that the skew raises significantly as soon as we use more cores than physical cores available.

Query	16 [s] / [%]	32 [s] / [%]	48 [s] / [%]	64 [s] / [%]	96 [s] / [%]
3-clique	0.1 / 1.04	0.1 / 2.31	0.1 / 2.86	0.1 / 4.74	0.1 / 5.74
4-clique	1.4 / 2.34	2.9 / 8.16	3.3 / 12.91	5.7 / 21.47	6.8 / 26.56
5-clique	0.0 / 0.00	0.0 / 0.00	3.8 / 0.48	18.4 / 2.55	23.8 / 3.66

Table 7: Total skew in seconds and percentage of skew in the total query time displayed for different queries and levels of parallelism on the LiveJournal dataset.

Query	16 [s] / [%]	32 [s] / [%]	48 [s] / [%]	64 [s] / [%]	96 [s] / [%]
3-clique	0.0 / 0.00	0.0 / 0.01	0.0 / 0.01	0.0 / 0.03	0.0 / 0.03
4-clique	0.2 / 0.04	0.3 / 0.13	0.4 / 0.26	30.1 / 16.28	62.2 / 32.37
5-clique	1.7 / 0.04	2.3 / 0.10	341.3 / 17.95	684.0 / 32.33	1025.0 / 45.44

Table 8: Total skew in seconds and percentage of skew in the total query time displayed for different queries and levels of parallelism on the Orkut dataset.

This could hint that part of the problem is causes by hyper-threading. We do not investigate this issue further.

Fourth, Shares exhibits lower speedup of 12.1 for 64 workers which is lower than for 32 workers (14.7) and 14.8 for 96 workers. This can be explained by the chosen Shares configuration. For 32 workers, the best configuration is given by the hypercube of the sizes 4, 4, 2. For 64 workers, we get the hypercube with 4 workers on each axis. Hence, although we are doubling the number of workers, we use the new workers only to partition work along the last axis, in the case of 3-clique along the C attribute axis. Partitioning work along the last axis leads to a high amount of duplicated work on the first two axis. Additionally, with 64 workers at least 12 of these workers are not exclusive cores but cores shared by two hyperthreads. In total, we get a lower speedup. This changes slightly for 96 workers because the optimal hypercube configuration here is 6, 4, 4 which adds more workers along the first axis. However, the scaling only increases marginally by 0.1 from 32 workers which is quite disappointing given that the number of threads increased by a threefold.

One could argue that we should use a different definition of *best* hypercube configuration. As we see, it is not necessarily efficient to distribute the computation along the last axis.

The Orkut and LiveJournal datasets lead to highly similar scaling results: super-linear scaling for *work-stealing* up to 48 workers, *work-stealing* scales significantly better than Shares. Shares exhibits less speedup for 64 workers than for 32 and 64 workers.

7.6 Distributed work-stealing

We run the distributed version of work-stealing as described in **??** on the LiveJournal and Orkut dataset for the 3-clique query.

For this experiment, we use four machines as described in **??**. Each of this machine has 48 physical cores with hyper-threading. In total, the Spark cluster has 192 physical cores and 384 virtual cores. We run the experiments on 16 to 192 of these cores. The tasks are evenly distributed over all four machines for all levels of parallelism by the standard behaviour of Spark's standalone mode scheduler.

8 Related Work

TODO introduction

8.1 WCOJ on Timely Data Flow

Mc Sherry et al. published a distributed worst-case optimal join based on Timely Data Flow in 2018 [7, 40]. In their paper they introduce three algorithms: *BigJoin*, *Delta-BigJoin* and *BigJoin-S*. They implement only the first two algorithms.

BigJoin-S is only described not implemented but comes with stronger theoretical guarantees. Namely, it is worst-case optimal in computation and communication with respect to the output size of the query given by the AGM bound. *BigJoin-S* can guarantee work balance. Moreover, it achieves optimality and work-balance while using low amounts of memory on all workers; the memory usage per worker is in $\mathcal{O}(\frac{IN}{w})$ with IN size of the input relationships and w the number of workers.

The other two join algorithm are only worst-case optimal in computation and communication costs but do not guarantue work-balance nor do they give the same low memory guarantues. Although, in praxis, they archieve both on many real-world datasets.

Delta-BigJoin is an incremental algorithm which computes the new instances of the subgraph given a batch of new edges. Hence, it operates in a different setting than our work. We assume static graphs while they operate on graphs with the ability to find new instances caused by insertions.

BigJoin is closest to our work. It has been implemented and is a worst-case optimal join for static graphs. In the following paragraphs, we describe *BigJoin*, analyse why it is not likely to be a good fit for Spark, discuss and compare the index structures used in their work to represent the input relationships and compare the guarantees given by them and us.

8.1.1 The *BigJoin* algorithm

BigJoin encodes a *Generic Join* (see section 2.2) into multiple timely dataflow operators.

In short, Timely Dataflow operators are distributed over multiple workers and each of them takes a stream of input data, operates on it and sends it to the next operator which can be processed on a different worker. Examples for operators are *map* functions, *filters count* or *min*. It is important to note that sending the output to an operator on a different worker is a fast, streaming operation, as opposed to, Spark's shuffles which are synchronous and slow because they involve disk writes and reads.

For the *BigJoin* the authors require each worker to hold an index for each input relationship which maps prefixes of the global variable order to the possible bindings for the next variable. In use-case of graph pattern matching, this means that each worker holds an index into the forward and backward adjacency lists.

Their algorithm runs in multiple rounds; one per variable in the join query. In each round, they bind one variable. So each round takes the prefixes as input and fixes one more binding.

A single round starts with all prefixes from the former round distributed among all workers arbitrarily. Then they find join relation that offers the smallest set of extensions for each prefix. This is done in steps with one step per relationship. In each step, the prefix is sent to a worker by the hash of the attributes bound in the relationship of that step. When the relationship offers less possible values for the new binding then the current minimum, i.e. the size of its matching

adjacency list is smaller, we remember it as the new minimum for the given prefix.

Next, they hash the values of the prefix which are defined in the relationship with the least extensions and use these hashes to distribute the tuples over all workers. Then, each worker produces all possible extensions for each prefix.

Finally, each round ends with filtering out all extensions that are not in the intersection of extensions offered by each relationship. This again takes one filtering step per relationship in the join.

This is a simple instance of the *Generic Join* implemented in data flow operators.

The algorithm described so far can build a high amount of possible extensions in each round. This keeps it from keeping worst-case optimal guarantees for memory usage. The authors fix this problem by batching the prefixes; they allow only a certain number of prefixes in the system at all times. They defer building new prefixes until the current batch of prefixes has been completed. This is natively supported by Timely Dataflow.

8.1.2 Applicability to Spark and comparison to GraphWCOJ

BigJoin is not suitable for Spark. This has multiple reasons.

Most importantly, it uses too many shuffle rounds. Each round and each step in a round requires communication and therefore a shuffle. In total, the algorithm uses $2R \times V$ rounds for a query with R relations and V variables. As pointed out before this is no big problem in Timely Dataflow because shuffles are fast and asynchronous. However, in Spark, this is not the case.

We would like to point out that binary join plans can solve the same queries in $R - 1$ shuffle rounds and that our solution does not require any communication rounds.

Second, Spark does not support batching queries naturally as Timely Dataflow. Building support for batching into Spark would be an engineering effort. Additionally, it would be hard to define a good user interface over a batched query in Spark.

GraphWCOJ does not require batching because it only processes at most as many prefixes as workers in the system in parallel. Therefore, we do not have the problem of memory pressure to remember prefixes. This is because the LFTJ algorithm is a none materialized representation of the join. When the Leapfrog Triejoin is executed, it changes its state such that the state always represents the none materialized part of the join; the state is encoded in the positions of the *TrieIterators*. In other words, the LFTJ performs a depth-first search of all possible bindings while the *BigJoin* performs a batched breadth-first search.

8.1.3 Indices used by *BigJoin* and GraphWCOJ

The index structures used in their and our solutions are the same; one forward and one backward index over the whole graph on each worker. It is possible to distribute the index of *BigJoin* such that each worker holds only a part of the index. This is because each worker needs to hold only the possible extensions for the prefixes that map to it for each relationship. We analyse this in the next paragraph.

The prefixes are mapped to workers by the hash of the attributes already bound. For graph pattern matching this is one or zero attributes; the edge relationship has two attributes and one is a new, yet unbound variable in the prefix.

We can reach a distribution of the indices such that each worker holds $\frac{I}{w}$ with I the size of the indices. For that, we choose the same hash function for each variable such that always the same

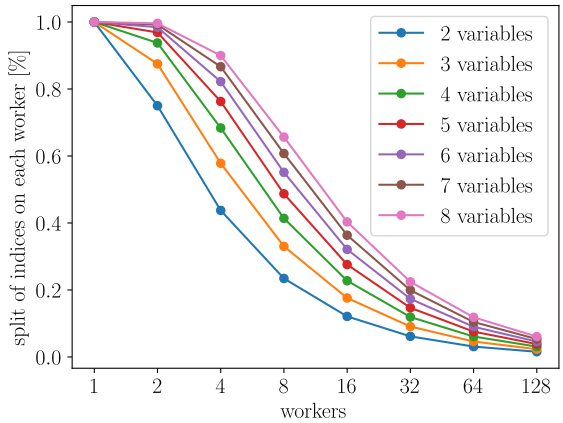


Figure 26: Expected split of the indexes hold on each worker for different numbers of workers used and variables in the query.

values match to the same worker. However, this solution is likely to lead to high skew and work imbalance because if a value is a heavy hitter the worker needs to process it for each binding over and over.

It is better to use different hash functions per variable. In this case, we can estimate the percentage of the whole index hold by each worker by the binomial distribution. This distribution models the probability that out of N independent trials k succeed with the likelihood of p for a single trial to succeed. We model the event of a key from the index being assigned to a worker as trial. The likelihood is $\frac{1}{w}$. We have as many trials as variables in the join: $N = V$. We are interested in the case that the tuple is not assigned by any of the variables, so $k = 0$. Then we have the likelihood that a tuple is not assigned given by $\mathcal{B}(V, 0, \frac{1}{w})$; so the fraction of the indices assigned to each worker is $1 - \mathcal{B}(V, 0, \frac{1}{w})$.

We plot this function for different numbers of workers and variables in fig. 26. The split of the indices hold by each worker decreases drastically with the numbers of workers in the system. Hence, this partitioning scheme scales relatively well.

8.1.4 Theoretical guarantees

BigJoin guarantees computational and communication worst-case optimality. However, the communication optimality does not take into account how the indices are generated on each worker. If they are sent to each worker, this would be not worst-case optimal. The extensions of *BigJoin-S* additionally give work-balance and low memory usage in $\mathcal{O}(\frac{IN}{w})$.

GraphWCOJ guarantees computational worst-case optimality which it inherits from LFTJ. Worst-case optimal communication is given by the fact that we do not communicate. This is if we do not take the distribution of the indices into account which is in line with the analysis of the discussed paper.

If we take the distribution of indices into account, our algorithm is not worst-case optimal. During our setup, we broadcast the indices used. This is not optimal for a single query; Shares would be optimal. However, it amortizes quickly over multiple queries, while Shares converges to broadcasting for big queries.

We can not guarantee work-balance. However, we get close to it by using work-stealing. With work-stealing, we are optimal within the size of a single task.

GraphWCOJ's memory footprint does not depend on the size of the input nor of the output of the join. Its memory usage is given by the size of the Java objects used which depend on the query. However, this size should be neglectful small for all but embedded use-cases.

8.1.5 Conclusion

We conclude that our approach is the better fit for Spark because it requires less shuffling and no batching. GraphWCOJ gives nearly the same theoretical guarantees as *BigJoin*. While they can distribute their indices we cannot; we rely on the fact that each worker holds the complete index.

Finally, we would like to point out that [7] does not publish any number on the amount of network traffic caused by their algorithm. Given that it sends many prefixes via the network this could be a bottleneck in many deployments, i.e. in cheaper instances int in the Amazon cloud. An analysis of the network traffic would be beneficial for a better understanding of the advantages and disadvantages of their approach.

8.2 Survey and experimental analysis of distributed subgraph matching

On the 28th July 2019, L. Lai et al. published a survey with experiments for multiple distributed graph pattern matching algoritms [32][11]. Here, we focus on four of the strategies they tested: *BigJoin* (see section 8.1), Shares, fully replicated graph and binary joins. All of their algorithms are implemented in Timely Dataflow; so far they are not open-source. They ran the all algorithms on 9 different queries over 8 datasets mostly on a cluster of 10 machines and 3 workers per machine. Below we first summarize the most important design decisions for each algorithm, then highlight their most interesting findings and finally compare their results with ours.

BigJoin is implemented as described above but uses a CSR data structure, triangle indexing and a specific form of compression as optimization.

The Shares algorithm is configured as described in section 2.3.1 and uses *DualSim* as local algorithm. *DualSim* is a specialized subgraph matching algorithm. The authors show that it beats the worst-case optimal join used in *EmptyHeaded* [1] which is a form of the *Generic Join* (see section 2.2).

The survey also covers our strategy of fully replicating the graph on all machines. They choose *DualSim* as a local algorithm and a round-robin partitioning on the second join variable.

Finally, they implement the binary joins with hash joins and use a sophisticated query optimizer to devise the best join order.

The most important finding of this work is that fully replicating the graph on all machines is the best option if the graph fits into memory even in Timely Dataflow with its deeply optimized and asynchronous communication routines. They establish that fully replicating the graph is nearly always the fastest strategy, has the lowest memory footprint[12], no further communication costs and scales better than all other strategies up to 60 workers.

In line with our argument against Shares, they find that this strategy is nearly always beaten by most other strategies. They establish that it takes longer than the other strategies on nearly all queries and datasets. Furthermore, it shows the weakest ability to scale.

They report that *BigJoin* or binary joins are the best option if fully replicating the graph is out

[11] The survey was published on arXiv 5 months after we started our thesis in February.

[12] Shares replicates too much data, *BigJoin* needs to hold many prefixes in memory and binary joins incur intermediary results.

of the question. Binary joins can be used for star and clique joins if it is possible to index all triangles in the graph and keep this index in memory. Otherwise, *BigJoin* is preferable in most cases.

Finally, they study the communication costs of the binary joins, Shares and *BigJoin*. They find that graph pattern matching is computation bound problem when 10 GB switches are used for networking but communication costs dominate in 1 GB switched networks. They draw there conclusions from experiments run with 10 GB network infrastructure.

Interestingly, Shares incurs fewer communication costs than *BigJoin*.

Their paper differs from our thesis in multiple ways. However, they come to the same conclusions, namely that fully replicating is the preferred strategy when the graph fits into main memory and that Shares is not a good strategy for graph pattern matching.

We implement our system in Spark which has wide-spread usage in industry and a surrounding eco-system of graph pattern matching systems (see section 1.3).

We give a comparison between a column store, binary search based Leapfrog Triejoin and our CSR based GraphWCOJ. They do not report on the benefits of CSR in the context of WCOJ.

Their implementation of the fully replicated strategy differs from ours in two important factors.

First, they use a different local algorithm

Second, they use a different partitioning scheme. Their scheme replicates work of finding bindings for the first and second variable in a query and does not actively counter skew. The skew-resilience of their scheme is based on the fact that it partitions the work on the second binding. Hence, it distributes skew of the first binding equally. However, as we see with our work-stealing approach this does not guarantee skew freeness for bigger queries (see section 7.5).

Their scheme could be applied to our system. It is simpler than work-stealing but less resilient to skew.

8.3 Fractal a graph pattern mining system on Spark

Fractal is a general-purpose graph pattern mining system built on top of Spark published at SIGMOD'19 [23]. We first describe the relevant aspects of their system. Then we compare it to our approach.

Graph pattern mining includes the problem of graph pattern matching (as defined in section 1.1). Additionally, it includes problems such as frequent subgraph mining or keyword-based subgraph search.

To support all these problems in a single system, the authors describe their own programming interface made off initialization operators, workflow operators and output operators. Each workflow is described as a sequence of these operators. All workflows are based around extending a subgraph starting from a single edge, vertex or a user-described pattern. This makes for three initialization operators one to start from a vertex, edge or pattern each.

The workflow operators process the subgraphs induced by the initialization operators. They can expand the subgraph, e.g. if the subgraph is vertex induced, one expand step adds all neighbouring vertices to the subgraph.

Another workflow operator is to filter the subgraph instances.

Then it is possible to aggregate subgraphs by computing a key, a value and possibly a reduction.

Finally, these workflow steps can be looped to be repeated multiple times.

To execute the workflow the user can use one out of two output operators: *subgraphs* and *aggregation* to list all matching subgraphs or aggregate all matching subgraphs respectively.

For example, the workflow $vertexInduced().filter(sg => fullyconnected).subgraphs()$ enumerates all cliques in a graph.

Fractal maps these workflows to Spark by splitting them into *fractal steps* on synchronization points, e.g. an aggregation which results is required in the next step. Each step maps to a Spark job which is scheduled by the Spark scheduler. Fractal schedules the *fractal steps* in the correct order and waits for them to complete before starting the next one.

A typical problem of graph pattern mining is the high amount of memory needed to keep partial matches; the state of the algorithm. Fractal counters this problem by enlisting subgraphs with a depth-first strategy. Furthermore, it starts computing all subgraphs from scratch for each steap, instead of keeping them in memory in between the steps. They only keep the results of the aggregations to be used by the next step.

Another problem in graph pattern mining is work-balance because some parts of the graph are more work-intensive than others. Fractal tackles this problem with work-stealing. They use a hierarchical work-stealing approach. First, each thread tries to steal work from another thread within the same Spark executor. Only if this is not possible, they request work from another machine.

The local work-stealing is implemented by sharing the same subgraph enumerators; an iterator-like data structure that saves the state of the subgraph matching algorithm in a prefix match. The subgraph enumerator offers a thread-safe method to generate the next prefix. Hence, a thread can steal work simply by using the enumerator of another thread.

The second hierarchy of work-stealing is between multiple Spark workers. The authors support that by using Akka to implement a simple message passing interface between all Spark workers.

Their experiments show near-linear scaling for the described work-stealing strategy up to 280 execution threads over 10 machines.

Fractal is similar to our system in some aspects. They also solve graph pattern matching on Spark, inspired our approach to work-stealing and choose a depth-first subgraph enumeration approach. We discuss these similarities below and outline the differences.

Like us, Fractal solves the problem of graph pattern matching. However, they offer the ability to solve multiple other common subgraph related problems as well. The biggest difference is that they directly support aggregation over subgraphs, which, for example, allows them to solve frequent subgraph mining.

They build an independent system on top of Spark's infrastructure which comes with their own imperative query language. We integrate a single algorithm deeply into Spark's query optimizer. Therefore, our contribution can be easily integrated into other graph systems building on Spark, e.g. G-Core [8] or CAPS [45]. These systems would offer a declarative interface to our worst-case optimal joins.

The work-stealing approach of Fractal inspired our solution. We also use a shared object to steal work within a single Spark executor. Anyhow, our approach is simpler and less fine-grained. They allow stealing work at every level of the depth-first enumeration of all subgraphs. We only share work on the first level. This makes their solution strictly more fine-grained and likely to perform better on big queries. We discuss this issue in our future work section **??**.

Fractal is built for Spark in cluster mode. Hence, they allow processes to steal work from different workers. GraphWCOJ is currently limited to a single worker. However, their message-passing based solution is directly applicable to our system if we extend to multiple workers. Again, we talk about this in more depth in future work (9.1.1). In short, we could use the same message-passing

implementation but instead of stealing from a subgraph enumerator, the work would be taken from the queue on each worker.

Both systems enlist the subgraphs in a depth-first like fashion. In both systems, this is highly beneficial to memory usage; the problem of breadth first algorithms has been discussed for *BigJoin* before (section 8.1).

To conclude, Fractal is a complete system with its own query interface. This makes it more powerful than our system but also less integrated into Spark. Hence, it forces the user to adapt to their imperative language and hinders integration with declarative graph query languages. We used a similar work-stealing algorithm in our work. Both systems recognize and demonstrate the advantages of a depth-first approach.

9 Conclusions

9.1 Future work

9.1.1 Cluster mode

9.1.2 Finer-grained work-stealing

In our experiment section 7.5, we noted that work-stealing that operates only on the first level of variable bindings can still lead to skew for bigger queries. Therefore, we describe a work-stealing LeapfrogTrieJoin algorithm that allows to steal work on all levels. We describe a possible design in two steps. First, we explain under which circumstances to steal work. Then, we describe how to steal work. In the following, we call the process that steals work thief and the other process victim.

We let each task start with its own bindings for the first variable, e.g. by assigning a range to each task based on its partition number. This can be implemented as a range filter in the *LeapfrogJoin* of the first variable.

Once all of these initial bindings are processed, we start stealing work from other *LeapfrogTriejoins*. It is beneficial to steal bindings of variables higher in the global order because this maximizes the amount of work stolen. A task is encoded as a prefix of variable bindings, e.g. if we steal work in a 5-clique query at the third level a prefix might be $[4, 1, 5]$. If the work-stealing request successfully returns a binding, we set the state of all components of the current join to the values of the prefix. Then we run the normal Leapfrog Triejoin algorithm to generate all complete bindings for the stolen prefix. When all bindings have been produced, we steal work and repeat the process.

If a work-stealing request cannot find any work, the task finishes.

We face four challenges for the question of how work is stolen.

First, the Leapfrog Triejoin encodes its state in the *TrieIterator* components. This state should not be changed when we steal work, except for the fact that a stolen prefix is not considered by the victim.

Second, when we use the *LeapfrogJoin* to steal work it is not guarantued that the underlying *TrieIterators* are set to the correct level for this *LeapfrogJoin*.

Third, we plan to use shared *LeapfrogJoin* instances to implement work-stealing. Therefore, this interface needs to become thread-safe.

Fourth, the *LeapfrogJoin* interfaces need to be accessible to the thieves. As in our current solution,

we implement communication via a shared data structure. This data structure allows access to all *LeapfrogJoin* instances on the same worker. If a task needs to steal work, it selects one of these *LeapfrogJoin* instances.

We describe the solution to this problems in order.

For the first problem of not changing the state of the *TrieIterators*, we suggest to add new *seek* and *next* methods to the *LeapfrogJoin* and *TrieIterator* interface which do not change the state but work exactly as the originals otherwise. Additionally, the stateless *next* method of the *LeapfrogJoin* interface stores the last value it returns. This value can be used from the stateful method to seek for the upper bound of it, such that values returned by the stateless version are skipped in the statefull version. Then thiefs can use the stateless versions and the owner of the interfaces uses the stateful versions.

The second issue of ensuring that the *LeapfrogJoin* uses the *TrieIterators* with the correct level is trivially solved by introducing one version per level of the stateless *next* and *seek* methods of the *TrieIterator* interface and store which to use in the *LeapfrogJoin*.

Introducing a thread-safe *LeapfrogJoin* interface requires one lock per instance which needs to be aquired before using any method and released after. It is not necessary to lock the underlying *TrieIterator* interfaces as we argue in the next paragraphs.

We start our argument that we only need to consider cases where one thief and one victim thread interfere with each other. This is because only one thief can be active at any *LeapfrogJoin* at any time due to the necessity to hold the lock for this *LeapfrogJoin*.

We observe that to use a specific *LeapfrogJoin* to steal work from, the victim needs to work on bindings which are later in the variable ordering. Otherwise, the variables above are not bound and it is not possible to steal a complete prefix. This is a precondition to be enforced on the thieves side when it chooses a *LeapfrogJoin* to steal work from.

Then, we require the victim to hold the lock of the respective *LeapfrogJoin* instance when its enters the *trieJoinUp* method. This guarantees that the victim *LeapfrogTriejoin* cannot break the assumption above during the process of work-stealing.

With these assumptions in place, we are ensured that the victim does not call any *TrieIterator* methods which interfere with the thief. There are two cases to consider. First, the *TrieIterator* is independent from the work-stealing process because it is not part of the intersecion of the *LeapfrogJoin* which is used by the thief. Second, the *TrieIterator* is used at a deeper level than the one which is used by the *LeapfrogJoin* to steal work of. In both cases, the use of the *TrieIterator* does not interfere with the thief.

The changes outlined above allow to share the *LeapfrogJoin* instances of all tasks of an executor to steal work at any level in the variable ordering. This should approach should lead to work-stealing jobs which are fine-grained enough to be nearly skew free for queries of all sizes. We end the section with a short discussion of lock contention.

The algorithm starts with totally uncontested locks until the first tasks finishes work in its range. Then the number of threads which could contend for locks grows linearly with the progress of the algorithm within finding all possible bindings because before a thread can start work-stealing it finishes its own range.

When threads start stealing work they can choose their victims such that they minimize lock contention. Hence, locks are contested mostly between the thief and the victim. For this case, we note that the thief chooses *LeapfrogJoins* as high in the variable ordering as possible which are less often used by the victim which spends most of its time with bindings for lower variables.

Finally, the locking time of the locks is short most of the times because it only needs to find one further binding. With materialized *LeapfrogJoins* the locked code section only reads one value

from an array, one or at most two *seek* calls on the first level of an *TrieIterator*[13] and stores the value returned.

[13]These calls need two array reads. Hence, they are actions in constant time.

References

[1] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. "Emptyheaded: A Relational Engine for Graph Processing." *Transactions on Database Systems (TODS)* 42.4 (2017), Page 20.

[2] F. N. Afrati, M. R. Joglekar, C. M. Re, S. Salihoglu, and J. D. Ullman. "GYM: A Multiround Distributed Join Algorithm." *20th International Conference on Database Theory (ICDT)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

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

[4] F. N. Afrati and J. D. Ullman. "Optimizing Multiway Joins in a Map-Reduce Environment." *Transactions on Knowledge and Data Engineering (TKDE)* 23.9 (2011), Pages 1282–1298.

[5] S. Agarwal, D. Liu, and R. Xin. *Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop, Deep dive into the new Tungsten execution engine*. 2016. URL: `https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html` (visited on 09/26/2019).

[6] A. Amler. "Evaluation of Worst-Case Optimal Join Algorithms." Master's thesis. Technische Universität München, 2017.

[7] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. "Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows." *VLDB Endowment*. Volume 11. 6. 2018, Pages 691–704.

[8] R. Angles, M. Arenas, P. Barcelo, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, et al. "G-CORE: A Core for Future Graph Query Languages." *International Conference on Management of Data (SIGMOD)*. ACM. 2018, Pages 1421–1432.

[9] *Apache hadoop*. URL: `http://hadoop.apache.org` (visited on 09/26/2019).

[10] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. "Design and Implementation of the LogicBlox System." *International Conference on Management of Data (SIGMOD)*. ACM. 2015, Pages 1371–1382.

[11] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. "Spark SQL: Relational data Processing in Spark." *International Conference on Management of Data (SIGMOD)*. ACM. 2015, Pages 1383–1394.

[12] A. Atserias, M. Grohe, and D. Marx. "Size Bounds and Query Plans for Relational Joins." *Foundations of Computer Science (FOCS)*. IEEE. 2008, Pages 739–748.

[13] P. Beame, P. Koutris, and D. Suciu. "Skew in Parallel Query Processing." *SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2014, Pages 212–223.

[14] A. Bodaghi and B. Teimourpour. "Automobile Insurance Fraud Detection Using Social Network Analysis." *Applications of Data Management and Analysis*. Springer, 2018, Pages 11–16.

[15] P. Boldi and S. Vigna. "The WebGraph Framework I: Compression Techniques." *World Wide Web Conference (WWW)*. Manhattan, USA: ACM Press, 2004, Pages 595–601.

[16] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. "HaLoop: Efficient Iterative Data Processing on Large Clusters." Volume 3. 1-2. VLDB Endowment, 2010, Pages 285–296.

[17] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication using Compressed Sparse Blocks." *Symposium on Parallelism in Algorithms and Architectures*. ACM. 2009, Pages 233–244.

[18] S. Chu, M. Balazinska, and D. Suciu. "From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System." *International Conference on Management of Data (SIGMOD)*. ACM. 2015, Pages 63–78.

[19] Databricks. *Apache Spark Documentation: RDD Programming Guide*. URL: `https://spark.apache.org/docs/2.2.3/rdd-programming-guide.html` (visited on 09/26/2019).

[20] Databricks. *Databricks Scala Guide*. 2018. URL: `https://github.com/databricks/scala-style-guide/blob/7eb5477781c11f9a75a2d8d6ef773ca6965f4ea0/README.md` (visited on 05/25/2019).

[21] A. Dave, J. Bradley, T. Hunter, and X. Meng. *GraphFrame*. 2016. URL: `https://databricks.com/blog/2016/03/03/introducing-graphframes.html` (visited on 02/18/2019).

[22] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *Symposium on Operating System Design and Implementation (ODSI)*. San Francisco, CA: Usenix, 2004, Pages 137–150.

[23] V. Dias, C. H. Teixeira, D. Guedes, W. Meira, and S. Parthasarathy. "Fractal: A General-Purpose Graph Pattern Mining System." *International Conference on Management of Data (SIGMOD)*. ACM. 2019, Pages 1357–1374.

[24] G. W. Flake, S. Lawrence, C. L. Giles, and F. M. Coetzee. "Self-Organization and Identification of Web Communities." *Computer* 3 (2002), Pages 66–71.

[25] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. "GraphX: Graph Processing in a Distributed Dataflow Framework." *Symposium on Operating System Design and Implementation (ODSI)*. Volume 14. Usenix, 2014, Pages 599–613.

[26] P. Gupta, V. Satuluri, A. Grewal, S. Gurumurthy, V. Zhabiuk, Q. Li, and J. Lin. "Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs." *VLDB Endowment*. Volume 7. 13. 2014, Pages 1379–1380.

[27] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." *NSDI*. Volume 11. 2011. 2011, Pages 22–22.

[28] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. "Graphflow: An Active Graph Database." *International Conference on Management of Data (SIGMOD)*. ACM. 2017, Pages 1695–1698.

[29] B. Ketsman and D. Suciu. "A worst-case optimal multi-round algorithm for parallel computation of conjunctive queries." *SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM. 2017, Pages 417–428.

[30] P. Koutris, P. Beame, and D. Suciu. "Worst-case Optimal Algorithms for Parallel Query Processing." *International Conference on Database Theory (ICDT)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.

[31] *kubernetes*. URL: `https://kubernetes.io` (visited on 09/26/2019).

[32] L. Lai, Z. Qing, Z. Yang, X. Jin, Z. Lai, R. Wang, K. Hao, X. Lin, L. Qin, W. Zhang, et al. "A Survey and Experimental Analysis of Distributed Subgraph Matching." *arXiv preprint arXiv:1906.11518* (2019).

[33] J. Laskowski. *The Internals of Spark SQL, QueryExecution - Structured Query Execution Pipeline*. URL: `https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-QueryExecution.html` (visited on 09/26/2019).

[34] LDBC. *LDBC SNB Documentation*. 2017. URL: `https://github.com/ldbc/ldbc_snb_docs` (visited on 04/10/2019).

[35] J. Leskovec and A. Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. June 2014.

[36] J. Leskovec and R. Sosič. "SNAP: A General-Purpose Network Analysis and Graph-Mining Library." *Transactions on Intelligent Systems and Technology (TIST)* 8.1 (2016), Page 1.

[37] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. "Pregel: A System for Large-Scale Graph Processing." *International Conference on Management of Data (SIGMOD)*. ACM. 2010, Pages 135–146.

[38] F. McSherry, M. Isard, and D. G. Murray. "Scalability! But at What COST?" *Hot Topics in Operating Systems (HotOS XV)*. 2015.

[39] A. Mhedhbi and S. Salihoglu. "Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins." *CoRR* abs/1903.02076 (2019). arXiv: `1903.02076`. URL: `http://arxiv.org/abs/1903.02076`.

[40] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. "Naiad: A Timely Dataflow System." *Symposium on Operating Systems Principles*. ACM. 2013, Pages 439–455.

[41] M. E. Newman. "Detecting Community Structure in Networks." *The European Physical Journal B* 38.2 (2004), Pages 321–330.

[42] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. "Worst-Case Optimal Join Algorithms." *SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM. 2012, Pages 37–48.

[43] H. Q. Ngo, C. Ré, and A. Rudra. "Skew Strikes Back: New Developments in the Theory of Join Algorithms." *arXiv preprint arXiv:1310.3314* (2013).

[44] D. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra. "Join Processing for Graph Patterns: An Old Dog with new Tricks." *GRADES*. ACM. 2015, Page 2.

[45] openCypher Project. *CAPS: Cypher for Apache Spark*. 2016. URL: `https://github.com/opencypher/cypher-for-apache-spark` (visited on 02/18/2019).

[46] G. Sadowski and P. Rathle. "Fraud Detection: Discovering Connections with Graph Databases." *White Paper-Neo Technology-Graphs are Everywhere* (2014).

[47] S. Salihoglu and M. T. Özsu. "Response to "Scale Up or Scale Out for Graph Processing"." *Internet Computing* 22.5 (2018), Pages 18–24.

[48] C. Schroeder dewitt. *Leapfrog Triejoin Implementation for 'Database Systems and Implementation' at Oxford University*. 2012. URL: `https://github.com/schroeder-dewitt/leapfrog-triejoin` (visited on 03/14/2019).

[49] W. F. Tinney and J. W. Walker. "Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization." *Proceedings of the IEEE*. Volume 55. 11. IEEE, 1967, Pages 1801–1809.

[50] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. "Apache Hadoop Yarn: Yet Another Resource Negotiator." *Symposium on Cloud Computing*. ACM. 2013, Page 5.

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

[52] R. Xin and J. Rosen. *Project Tungsten: Bringing Apache Spark Closer to Bare Metal*. 2015. URL: `https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html` (visited on 09/26/2019).

[53] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." *Conference on Networked Systems Design and Implementation*. USENIX. 2012, Pages 2–2.

A Experimental Results

A.1 *Graph*WCOJ scaling

Partitioning	Query	Parallelism	Time	Speedup
Shares	3-clique	1	0.0	1.0
Shares	3-clique	2	0.0	1.8
Shares	3-clique	4	0.0	3.4
Shares	3-clique	8	0.0	3.5
Shares	3-clique	16	0.0	6.8
Shares	3-clique	32	0.0	8.5
Shares	3-clique	48	0.0	10.1
Shares	3-clique	64	0.0	8.9
Shares	3-clique	96	0.0	9.9
Shares	4-clique	1	0.3	1.0
Shares	4-clique	2	0.2	1.9
Shares	4-clique	4	0.1	3.6
Shares	4-clique	8	0.0	6.7
Shares	4-clique	16	0.0	6.8
Shares	4-clique	32	0.0	12.7
Shares	4-clique	48	0.0	10.8
Shares	4-clique	64	0.0	17.3
Shares	4-clique	96	0.0	21.1
Shares	5-clique	1	2.6	1.0
Shares	5-clique	2	1.5	1.8
Shares	5-clique	4	0.8	3.5
Shares	5-clique	8	0.4	7.0
Shares	5-clique	16	0.2	12.0
Shares	5-clique	32	0.2	12.0
Shares	5-clique	48	0.1	20.1
Shares	5-clique	64	0.2	17.5
Shares	5-clique	96	0.1	21.7
work-stealing	3-clique	1	0.0	1.0
work-stealing	3-clique	2	0.0	2.0
work-stealing	3-clique	4	0.0	4.1
work-stealing	3-clique	8	0.0	8.2
work-stealing	3-clique	16	0.0	16.1
work-stealing	3-clique	32	0.0	26.3
work-stealing	3-clique	48	0.0	40.1
work-stealing	3-clique	64	0.0	44.7
work-stealing	3-clique	96	0.0	49.3
work-stealing	4-clique	1	0.3	1.0
work-stealing	4-clique	2	0.2	2.1
work-stealing	4-clique	4	0.1	4.2
work-stealing	4-clique	8	0.0	8.5
work-stealing	4-clique	16	0.0	16.9
work-stealing	4-clique	32	0.0	30.8
work-stealing	4-clique	48	0.0	42.7
work-stealing	4-clique	64	0.0	49.0
work-stealing	4-clique	96	0.0	54.6
work-stealing	5-clique	1	2.6	1.0
work-stealing	5-clique	2	1.4	1.9
work-stealing	5-clique	4	0.7	3.8
work-stealing	5-clique	8	0.3	7.8
work-stealing	5-clique	16	0.2	15.6
work-stealing	5-clique	32	0.1	28.3
work-stealing	5-clique	48	0.1	47.0

Partitioning	Query	Parallelism	Time	Speedup
Shares	3-clique	1	1.7	1.0
Shares	3-clique	2	0.8	2.2
Shares	3-clique	4	0.4	4.1
Shares	3-clique	8	0.4	4.4
Shares	3-clique	16	0.2	8.5
Shares	3-clique	32	0.1	14.7
Shares	3-clique	48	0.1	13.7
Shares	3-clique	64	0.1	12.1
Shares	3-clique	96	0.1	14.8
Shares	4-clique	1	15.1	1.0
Shares	4-clique	2	8.2	1.8
Shares	4-clique	4	4.2	3.6
Shares	4-clique	8	2.4	6.4
Shares	4-clique	16	2.2	7.0
Shares	4-clique	32	1.1	13.2
Shares	4-clique	48	1.1	13.8
Shares	4-clique	64	0.8	19.3
Shares	4-clique	96	0.7	22.0
Shares	5-clique	1	531.3	1.0
Shares	5-clique	16	54.3	9.8
Shares	5-clique	32	47.5	11.2
Shares	5-clique	48	29.0	18.3
Shares	5-clique	64	29.7	17.9
Shares	5-clique	96	23.6	22.5
work-stealing	3-clique	1	1.7	1.0
work-stealing	3-clique	2	0.7	2.4
work-stealing	3-clique	4	0.3	5.0
work-stealing	3-clique	8	0.2	10.0
work-stealing	3-clique	16	0.1	19.6
work-stealing	3-clique	32	0.0	36.7
work-stealing	3-clique	48	0.0	50.0
work-stealing	3-clique	64	0.0	54.7
work-stealing	3-clique	96	0.0	61.3
work-stealing	4-clique	1	15.1	1.0
work-stealing	4-clique	2	7.5	2.0
work-stealing	4-clique	4	3.8	4.0
work-stealing	4-clique	8	1.9	7.8
work-stealing	4-clique	16	1.0	14.7
work-stealing	4-clique	32	0.6	25.8
work-stealing	4-clique	48	0.4	34.8
work-stealing	4-clique	64	0.4	34.2
work-stealing	4-clique	96	0.4	36.1
work-stealing	5-clique	1	531.3	1.0
work-stealing	5-clique	16	37.1	14.3
work-stealing	5-clique	32	18.8	28.3
work-stealing	5-clique	48	13.2	40.3
work-stealing	5-clique	64	12.1	44.1
work-stealing	5-clique	96	10.9	48.7

Table 10: Table showing the speedup of *Graph*WCOJ for 3-clique and 5-clique on the LiveJournal dataset. Time is shown in minutes.

Partitioning	Query	Parallelism	Time	Speedup
Shares	3-clique	1	16.3	1.0
Shares	3-clique	2	7.7	2.1
Shares	3-clique	4	4.0	4.0
Shares	3-clique	8	3.7	4.4
Shares	3-clique	16	1.9	8.8
Shares	3-clique	32	1.0	17.0
Shares	3-clique	48	1.2	14.1
Shares	3-clique	64	1.3	13.0
Shares	3-clique	96	0.9	17.4
Shares	4-clique	1	136.5	1.0
Shares	4-clique	2	67.2	2.0
Shares	4-clique	4	35.4	3.9
Shares	4-clique	8	19.0	7.2
Shares	4-clique	16	18.6	7.3
Shares	4-clique	32	9.5	14.4
Shares	4-clique	48	9.6	14.3
Shares	4-clique	64	6.5	20.9
Shares	4-clique	96	5.4	25.4
Shares	5-clique	1	1112.1	1.0
Shares	5-clique	16	99.7	11.2
Shares	5-clique	32	101.4	11.0
Shares	5-clique	48	70.0	15.9
Shares	5-clique	64	67.5	16.5
Shares	5-clique	96	50.6	22.0
work-stealing	3-clique	1	16.3	1.0
work-stealing	3-clique	2	7.4	2.2
work-stealing	3-clique	4	3.7	4.4
work-stealing	3-clique	8	1.7	9.3
work-stealing	3-clique	16	0.9	18.8
work-stealing	3-clique	32	0.4	36.9
work-stealing	3-clique	48	0.3	54.0
work-stealing	3-clique	64	0.3	58.0
work-stealing	3-clique	96	0.2	69.6
work-stealing	4-clique	1	136.5	1.0
work-stealing	4-clique	2	65.2	2.1
work-stealing	4-clique	4	32.5	4.2
work-stealing	4-clique	8	16.2	8.4
work-stealing	4-clique	16	8.1	17.0
work-stealing	4-clique	32	4.0	33.8
work-stealing	4-clique	48	2.7	50.3
work-stealing	4-clique	64	3.0	45.6
work-stealing	4-clique	96	3.2	42.5
work-stealing	5-clique	1	1112.1	1.0
work-stealing	5-clique	16	77.8	14.3
work-stealing	5-clique	32	39.6	28.1
work-stealing	5-clique	48	31.6	35.2
work-stealing	5-clique	64	35.2	31.6
work-stealing	5-clique	96	37.4	29.8

Table 11: Table showing the speedup of *Graph*WCOJ for 3-clique and 5-clique on the Orkut dataset. Time is shown in minutes.