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

Master thesis draft iteration 1.2

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 [25], 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 [23]. Our thesis provides the first distributed, open-source implementation of a WCOJ in a data processing system widely used in industry, namely Spark.

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, e.g. hash joins, work fine. 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, worst-case optimal join algorithms excel because matching a pattern consisting of multiple joins causes binary joins to generate a rapidly increasing set of intermediate results, e.g. navigating a social graph with an out-degree in the hundreds, of which many matches are useless and get eliminated by later joins, e.g. the join closing the cycle. These kinds of join

patterns are frequently found during graph analysis, e.g. for graph clustering on social network graphs for customer relationship management or recommendation systems and fraud detection in the financial sector [4, 17]. Worst-case optimal join algorithms 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 continue with a short example for a cyclic query and compare how this query is evaluated traditionally and with the new WCOJ's in place. The simplest example of a cyclical join query enumerates all triangles in a graph. This can be formulated as the following datalog query

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

where $R = S = T$ are aliases for the edge relationship. Traditionally, this would be processed by using multiple binary joins:

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

Independent of the chosen order, it can be proven that there exists cases where the intermediary result size is in $\mathcal{O}(n^3)$ with $n = |R| = |S| = |T|$. However, it is provable that maximal output of this query is in $\mathcal{O}(n^{3/2})$ [6, 28]. Hence, binary joins materialize huge intermediary results after processing parts of the query, which are much bigger than the final result after applying all joins. The described problem has been shown to be a fundamental issue with traditional join-at-a-time approaches [6, 28]. Fortunately, worst-case optimal join algorithms can materialize cyclic joins with memory usage linear to their output size by avoiding to produce large intermediary results [35, 29]. In practice, these algorithms have been shown to be highly beneficial for cyclic queries in analytical graph workloads in an optimized, single machine system [35, 30] and later in distributed shared-nothing settings [11, 3] - we describe these systems in more detail in section 3.

We identify two challenging, novel directions for our research. First, although, all of the systems cited above focus on queries widely used in graph pattern matching, e.g. clique finding or path queries, they use WCOJ's which are developed for general multi-way joins, however, graph pattern matching uses only self-joins on a relationship with two attributes, namely the edge relationship of the graph. This raises the question if WCOJ's can be optimized by specializing them for graph pattern matching - which is so far the only use-case that has been shown to benefit from WCOJ's in the literature. Second, while the communication costs for worst-case optimal joins in map-reduce like systems (an excellent definition of the term is given in [1]) is well-understood [1, 11], their scalability has not been studied in depth. Given the high complexity of worst-case optimal joins used and the fact that their only integration in a map-reduce like system [11] exhibits a speedup of 8 on 64 nodes (an efficiency of 0.125), leads us to the conclusion that designing a scalable, distributed WCOJ for a map-reduce like system is an unsolved challenge. We believe 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 would be also the first time a WCOJ is integrated with an industrial-strength cluster computing model. We detail our research questions and goals in **??** and explain how to address challenges in **??**.

2 Background

TODO introduction

2.1 Analysis of public real-world graph datasets

TODO will include histogram of graph sizes, average outdeegree, maybe clustering coefficient and further interesting metrics, with regards to the thesis, for (all?) graphs of the SNAP and Labaratory of Web Algorithms dataset collection.

2.2 Compressed sparse row representation

Compressed sparse row representation (short CSR) is a well known, low-memory representation for static graphs [10, 34]. 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 and a second of size `|V + 1|` which stores indices into the first array. 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.

The CSR format has two beneficial properties in the context of this thesis. First, it allows locating 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 2.1).

3 Related Work

We first provide a short overview of worst-case optimal join algorithms, then we introduce Spark and finally summarize existing distributed implementations of worst-case optimal join algorithms.

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 [6]. In 2012, Ngo, Porat, Re and Rudra published a join algorithm matching this bound [29]. 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. Both alorithms have been shown to be instances 'Generic Join' in 2013 Ngo et al. [28]. Leapfrog Triejoin is the only worst-case join algorithm that has been implemented and benchmarked numerous times in widely different settings, e.g. Oxford course work, published research and a commercial database system [35, 2, 30, 18, 3, 33].

Spark is the probably most widely used and industry accepted cluster computing model. It improves over former computing models, e.g. MapReduce [14], Hadoop or Haloop [9], by allowing to cache results in memory between multiple queries, using so-called resilient distributed datasets [36]. However, although Spark implements most operations in memory, shuffles are an exception which still writes and reads the whole intermediary dataset to and from disk. Consequently, shuffling remains to be a highly expensive operation which is the bottleneck of many workflows. Still, Spark provides the user with a general distributed data processing model with strong fault- and struggler tolerance on commodity shared-nothing clusters.

Due to its generality, widespread acceptance in the industry, the ability to use cloud hardware and its fault tolerance by design, it is an attractive target for big graph processing. For example, GraphFrames [13], GraphX [16] (a Pregel [24] implementation) or graph query languages as G-CORE [4] and openCypher with their 'Cyper for Apache Spark' [31] 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 [5]. 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.

In 2014, a Leapfrog Triejoin variant, dubbed "Tributary Join", was used as a distributed join algorithm on a shared-nothing architecture called "Myria" [11]. They use Tributary Join as a

local, serial worst-case optimal join algorithm, combined with the Hypercube shuffle algorithm to partition the data between their machines [1]. However, it is not obvious how well Hypercube shuffles scales because it replicates many of its input tuples [11]. The combination of Hypercube shuffles and Tributary Join in Myria does not scale well (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; we therefore, explain Hypercube shuffles in detail and show that they tend to replicate all data to all nodes for bigger graph patterns section 3.1. Their approach is directly applicable to Spark.

A second distributed version of worst-case optimal joins was published in 2018 based on a Timely Data Flow system [3, 27]. In Timely Data Flow, shuffling is a streaming, asynchronous operation which requires no disk access[1]. Therefore, the number of shuffle operations is less important than in Hadoop or Spark. The authors take advantage of this fact by using a uniform, non-replicating partitioning scheme for their relationships. Then they implement a worst-case optimal join using distributed data flow operators [27], e.g. min and intersection. Similar to us, the authors focus on scalability and efficiency in their work but due to the use of a streaming, in memory shuffles and a fine-grained batched processing scheme their approach is unlikely to be successful in Spark. Hence, their and our research share the same goals, however, we aim to achieve it in a more restrictive, but widely used and industrial accepted, system.

3.1 Hypercube shuffle

We first explain how the Hypercube shuffle algorithm, `HC`, partitions data. After, we provide an analysis of its scaling with regards to graph patterns with an increasing number of vertices.

`HC` partitions the input relationships for a multi-way join over w 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. `HC` realizes this partitioning by logical organizing all workers in a hypercube with one dimension per join variable; we call the number of variables A and use a_i to reference a single variable. Each dimension has a size p_i, the p_i's have to be chosen such that the constraint $w \geq \prod_i p_i$ is satisfied. In other words, each worker can be addressed by its coordinate in the hypercube of the form $\{1..p_1\} \times ... \times \{1..p_A\}$.

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 node. We choose a hash function h_i for each join variable which maps its values in the range of $\{1..p_i\}$. Then each worker determines where to send the tuple it holds by hashing its join variables. 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 not bound by the tuple. Then the tuple is sent to all workers with a matching coordinate. For example, assume a join with three variables a_1, a_2 and a_3 and tuple t that binds a_1 and a_2 then we get the coordinates $h_{a1}(t_{a1}) \times h_{a2}(t_{a2}) \times \{0..p_{a3}\}$ and send the tuple to all workers with a coordinate matching the first two attributes and arbitrary third component of the coordinate; the tuple is replicated across p_{a_3} machines.

Next, we analyse the scalability of `HC` on growing graph patterns - that is, joins over a single relationship, the edge relationship of the graph E and with two variables per atom. In this context, atoms of the join can be seen as the edges of the pattern and variables as vertices. In the following we consider the join query represented by the Datalog rule $Q(a_1, ..., a_A) = R_1(a_1, a_2), ..., R_k(a_{A-1}, a_A)$, we call $atoms(Q)$ the set of all atoms in Q, $p_1(R)$ and $p_2(R)$ the p_i corresponding to the variables of R. We use the method described in [11] to calculate optimal shares allocation $p_1...p_A$.

[1]The most commonly used cluster computing engines (Hadoop and Spark) implement shuffling as a synchronizing operation that requires to write and read all tuples from disk.

Pattern	Edges	workload [64]/[128]
Triangle	3	0.18 / 0.12
4-clique	6	0.59 / 0.44
5-clique	10	0.9 / 0.82
House	5	0.42 / 0.32
Diamond	8	0.76 / 0.67

Table 1: Workload, on 64 and 128 workers, in percentage of tuples of the edge table assigned to each worker, using Poison binominial distribution to estimate the workload and the method from [11] to determine the optimal shares configuration.

Each worker receives $\sum_{R \in atoms(Q)} |R|/(p_1(R) * p_2(R))$ tuples under the assumption of uniform data distribution and good hash functions. Our argument is that the tuples of each R are divided onto $p_1(R) * p_2(R)$ workers - the workers that form the hypercube planes 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 `HC` 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 $Pr(n, k, u_0, ..., u_n)$ allows us to calculate the likelihood that k out of n independent, binary and differently distributed trials succeed, under the condition that the i'th trial succeeds with a probability of u_i. We then use $n = |atoms(Q)|$, $k = 0$ and $u_i = 1/(p_1(R_i) * p_2(R_i))$ to calculate the probability that a tuple is not assigned to an arbitrary, fixed worker w. This allows us to predict the number of tuples assigned to each worker by $|E| * (1 - Pr(|atoms(Q)|, 0, u_0, ..., u_{|atoms(Q)|})$.

Table 1 shows the expected percentage of tuples from E assigned to each node for graph patterns of different sizes calculated using Poison binomial distribution and optimal shares assignments according to the method used in [11]. 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 and that doubling the number of workers is inefficient to counter this growth.

The second observation has two reasons. First, doubling the number of workers does not allow to double the dimensions of the hypercube - a hypercube always needs $p_1 \times ... \times p_A$ workers to be built. Second, the number of replicated tuples increases with a growing hypercube because each tuple from R_i is replicated to $\prod_{R_j \in atoms(Q)/R_i} p_1(R_j) * p_2(R_j)$; due to the fact that each tuple binds only two out of A variables the tuple is replicated over many dimensions, e.g. let the bounded varibales be a_p and a_s then we get $|\{0..p_0\} \times ...a_p... \times ...a_s... \times \{0..p_A\}|$ matching worker coordinates.

In light of the numbers presented in table 1 and in line [3], we conclude that the communication costs for `HC` converge towards a full broadcast for bigger graph patterns and scaling becomes increasingly inefficient. Anyhow, `HC` is proven to be communication cost optimal for general multi-way joins in map-reduce like systems in a single round of shuffling in multiple settings [7, 8, 20]. Therefore, we decide to follow a different direction for distributing WCOJ's which we explain in **??**.

4 Worst-case optimal join parallelization

Based on the fact that Shares is an optimal partitioning scheme for n-ary joins in MapReduce like systems [**shares**] and our analysis that Shares converges to a full broadcast of the graph edges

(see **??**), we decided to forego physical partitioning of the graph. We cache the graph 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 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 4.1.

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 4.2

4.1 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 over all 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 [11] and for Haddoop [**TODO**]. We describe the Shares and Myria in more detail in **??** 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 different 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 attribute does 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 **??**, 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 [**shares**, 11]. We implement the exhaustive search algorithm used in the Myria system [11].

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 **??**. 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 influece 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.

4.1.1 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 and a value matches a coordinate if it is in the correct range. In contrast to hash-based mappings, we can push a range based mapping into the *TrieIterators*. They can check after each call to *seek* and *next* if they are still in their range. Contrary, to hash-based mappings which are checked value by value until one matches, a range check is a single conditional after each function call. Furthermore, this conditional is predictable for the processor: for all but one call, the value is in range and returned.

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 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 for the first two attributes and assign the ranges of the later attributes randomly. 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.

4.2 Work-stealing

5 Implementation

5.1 General sequential version (*seq*)

We implement the Leapfrog Triejoin [35] as our general sequential version of a WCOJ. However, instead of using B-Trees as a backing data structure, we use sorted arrays and a binary search, which has been described in [11] and is called Tributary join in their paper. Our Leapfrog Triejoin is implemented in three components which we explain in order below: *LeapfrogJoin*, *ArrayTrieIterable* and *LeapfrogTriejoin*.

The Leapfrog join is a variant of the sort-merge join for unary relationships, originally described in [15, 19]. To join k unary relations $A_1(x)$, $A_2(x)$, ..., $A_k(x)$ it takes one iterator per input relations and offers an iterator interface that yields the intersection of all relations. It requires that it's input iterators offer a *key* method in $\mathcal{O}(1)$, a *next* method and a *leastUpperBound(key: Int)* both in $\mathcal{O}(\log n)$ (n defined as the size of the input relationship). *leastUpperBound* moves the iterator to the first position of the sought *key* or the first position of the next higher value. An idiotmatic implementation of a Leapfrog join is shown in listing 1, for the optimized implementation see `leapfrogTriejoin.LeapfrogJoin` in our repository.

```
class LeapfrogJoinIdiomatic(var iterators: Array[LinearIterator]) {
  var atEnd: Boolean = false
  var p = 0
  var key = 0L

  def init(): Unit = {
    atEnd = iterators.exists(li => li.atEnd)
    p = 0
    key = -1

    iterators.sortBy(_.key)
    leapfrogSearch()
  }


  def leapfrogNext(): Unit = {
    iterators(p).next()
    p = (p + 1) % iterators.length
    leapfrogSearch()
  }

  def leapfrogLeastUpperBound(key: Long): Unit = {
    iterators(p).leastUpperBound(key)
    p = (p + 1) % iterators.length
    leapfrogSearch()
  }

  private def leapfrogSearch(): Unit = {
    if (!iterators(p).atEnd) {
      var max = iterators((p - 1) % iterators.length).key
      var min = iterators(p).key

      while (min != max && !iterators(p).atEnd) {
        iterators(p).leastUpperBound(max)
        max = iterators(p).key
        p = (p + 1) % iterators.length
        min = iterators(p).key
      }
      key = min
    }
    atEnd = iterators(p).atEnd
  }
}
```

Listing 1: Leapfrog join.

To support j-arity relations, $A(a_1, a_2, \ldots, a_j)$ we add two methods to the iterator interface that represents the input relationships: *up* and *open*; both are required to work in $\mathcal{O}(\log n)$. We call

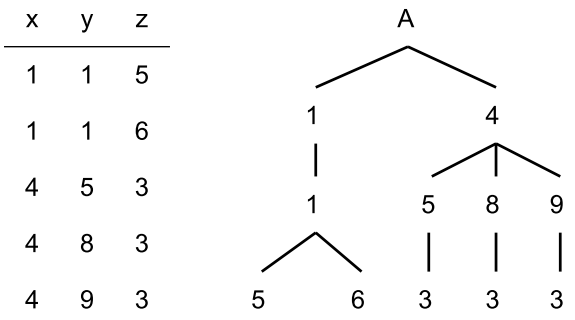


Figure 1: A 3-ary relationship as table (left) and trie (right), to position the iterator at the tuple (1, 1, 5) one calls *open* twice, *key* returns now 5, after a call to *next*, *key* returns 6 and *up* would lead to *key* returning 1.

this new iterator Trieiterator because it represents the relationship as a trie, see fig. 1.

The implementation of a Trieiterator backed by a columnwise representation of the relation using one array per column is straight forward, we outline the basic ideas here and refer the interested reader to `leapfrogTriejoin.ArrayTrieIterable.TrieIteratorImpl` in our repository for further details. It helps to think about the Trieiterator as consisting out of a linear component, containing the functions *key*, *next* and *leastUpperBound*, and horizontal component, made off the functions *up* and *open*, to move the linear component from one trie level to another.

First, we explain the horizontal component. They keep track of the current *level* of the Trieiterator and the *startPosition* and *endPosition* for in the column, e.g. in fig. 1 when the current *level* is 1 (or x), the key equals 4, the *startPosition* is 2 and the *endPosition* is 5 because the value 4 occurs 3 times. With these bookkeeping variables, updated by *up* and *open*, one can implement the linear part by a binary search over the current column (given by *level*) which is limited to *startPosition* and *endPosition*.

The Leapfrog Triejoin combines TrieIterators and Leapfrog joins to join k relationships of arbitrary arity. Its input is one Trieiterator per relationship, with these it builds one Leapfrog join per attribute which receives references to all Trieiterator of relationships containing the attribute, e.g. for the triangle query *R(a, b), S(b, c), T(a, c)* the Leapfrog Triejoin receives three Trieiterator, for R, S and T, and builds three Leapfrog joins, for a, b, c, which receive references to two Trieiterators each. To generate the join result the Leapfrog Triejoin operates the horizontal components of the Trieiterators directly and uses the Leapfrog joins to operate the linear component.

We show an idiomatic implementation of a Leapfrog Triejoin in listing 2 and 3, a performance oriented implementation can be found in our repository in `leapfrogTriejoin.LeapfrogTriejoin`. These listings contain two important functions: the initialization function from line 3 to line 14 and the *moveToNextTuple* function at line 16. We go through these functions in order.

The initializer gets two arguments: a mapping from variables to *TrieIterators* (each *TrieIterator* belongs to the list of attributes of its relationship) and the global variable ordering as a sequence of *Strings*. First, it creates one *LeapfrogJoin* per variable (line 5) which receives references to each *TrieIterator* operating on a relationship with this attribute. Then it builds a mapping from variables to all *TrieIterators* acting on a relationship with an attribute of the same name (line 8). Finally, it initializes *maxDepth*, *action*, *depth*, *bindings* and *atEnd* (line 10 to line 14). *depth* and *bindings* are an internal variable storing the index of the variable to bind currently and the current bindings for all variables up to *depth*. *atEnd* signals that the join has been completed to the client.

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: Summarizes which action follows from the current action under which condition. *lf* abbreviates the *LeapfrogJoin* of the current variable. *reachedMaxDepth* is true if we currently find bindings for the last variable in the global order. The columns *Yields* details if the main loop of the state machine yields before computing the next action, this is the case, when the *bindings* array contains a complete tuple.

moveToNextTuple implements a depth-first traversal of the *TrieIterators*. 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 bound (loop condition in line 35). 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*, *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.

NEXT moves the *LeapfrogJoin* at the current depth to the next possible binding for its variable (line 2 in listing 3). If the *LeapfrogJoin* reached its end, we continue with the *UP* action (line 4), 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 7).

DOWN moves to the next variable in the global variable ordering by opening all related *TrieIterators* (line 20) A *DOWN* can be followed by an *UP* if the *LeapfrogJoin* is *atEnd* (line 22), by a *NEXT* action if the trie join is at its lowest level (line 25), or by another *DOWN* to reach the last level.

UP 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_ACTION* line 28). Otherwise, all *TrieIterators* corresponding to the current variable are moved upwards by calling *triejoinUp* (line 31) 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 32).

```
class LeapfrogTriejoinIdiomatic(trieIterators: Map[Set[String], TrieIterator],
                                variableOrdering: Seq[String]) {
  // One LeapfrogJoin per variable with references to all TrieIterators
  // which relationships have an attribute of the same name
  private val leapfrogJoins: Array[LeapfrogJoin]

  // A mapping of each variable to all TrieIterators related to it
  private val variable2TrieIterators: Map[String, Seq[TrieIterator]]

  private val maxDepth = variableOrdering.length - 1
  private var action = DOWN_ACTION // The important line
  private var depth = -1
  private var bindings = Array.fill(variableOrdering.size)(-1L)
  var atEnd: Boolean = trieIterators.values.exists(i => i.atEnd)//

  def moveToNextTuple(): Array[Long] = {//
    if (action == NEXT_ACTION) action = nextAction()
    do {
      if (action == DOWN_ACTION) {
        triejoinOpen()
        if (leapfrogJoins(depth).atEnd) {
          action = UP_ACTION
        } else {
          bindings(depth) = leapfrogJoins(depth).key
          action = if (depth == maxDepth) NEXT_ACTION else DOWN_ACTION
        }
      } else if (action == UP_ACTION) {
        if (depth == 0) {
          atEnd = true
        } else {
          triejoinUp()
          action = if (leapfrogJoins(depth).atEnd) UP_ACTION else nextAction()
        }
      }
    } while (!((depth == maxDepth && bindings(maxDepth) != -1) || atEnd))
    bindings
  }
}
```

Listing 2: Shows the main methods of *LeapfrogTriejoin*, the initializer and *moveToNextTuple* functionality helper methods are detailed in listing 3.

```scala
private def nextAction(): Int = {
  leapfrogJoins(depth).leapfrogNext()
  if (leapfrogJoins(depth).atEnd) {
    UP_ACTION
  } else {
    bindings(depth) = leapfrogJoins(depth).key
    if (depth == maxDepth) NEXT_ACTION else DOWN_ACTION
  }
}

private def triejoinOpen(): Unit = {
  depth += 1
  val trieIterators = variable2TrieIterators(variableOrdering(depth)).foreach(_.open())
  leapfrogJoins(depth).init()
}

private def triejoinUp(): Unit = {
  variable2TrieIterators(variableOrdering(depth)).foreach(_.up())
  bindings(depth) = -1
  depth -= 1
}
```

Listing 3: *LeapfrogTriejoin* helpers.

5.1.1 Optimizations

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 spent roughly 2 weeks to optimize our first implementation. After, we are able to beat Spark's `BroadcastHashjoin` on nearly all queries and datasets. In this section, we discuss the implemented optimization and give a rough estimate of how important each of these is. In total, we improved the WCOJ running time from 248.2 seconds to 44.5 seconds on the unfiltered 5-clique query on the `Amazon-0602` dataset. We list all optimizations in table 3 and label them 'very important', 'important' and 'minor' based on the performance improvement directly after applying it.

It is not helpful to give more detailed information on the effect of single optimization because they are not independent of each other. Hence, they might have a hugely different effect when applied in a different order, e.g. we first applied an optimization to the binary search and then optimized the *LeapfrogJoin.next* method to avoid many searches Hence, giving detailed runtime measurements for the binary search optimization would overestimate its value. It is out of the scope of this work to study the dependency and order of the optimization to gain correct runtime measurements.

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 [11].

Category	Optimization	Impact
LFTJ	*LeapfrogJoin.init* avoid sorting iterators	very important
	ArrayTrieIterable.next in $\mathcal{O}(1)$ for deepest level	very important
Binary search	linear search for short search spaces	important
	avoiding unnecessary conditions	important
Spark	direct use of arrays instead of `ColumnVector`	important
Scala	use *while* instead of *map*, *foreach*, *exists*, etc	very important
	use *Array* instead of Scala's collections	very important
	use of *private[this]*	minor
	enable compiler optimization	minor
General	remove array lookups from the critical path *column(depth)(position)* → *currentColumn(position)*	very important
	use *Array* instead of *Map* if keys are integers and dense	important
	strength reduction *(i + 1) % 5* → *if (i == 4) 0 else i+ 1*	important

Table 3: Summary of all optimizations used for *seq* and an estimate of their impact.

We applied two Tributary join specific optimizations. The first in the class `leapfrogTriejoin.LeapfrogJoin` (see also listing 1) and the second in the `leapfrogTriejoin.ArrayTrieIterable.TrieIteratorImpl`.

The *LeapfrogJoin.init* method is origingally described in [35] to sort its *TrieIterators*. However, the method can be improved by avoiding to sort the *TrieIterators* (line 11). We can start moving the *TrieIterator* without sorting them and arrive at an ordered array in $\mathcal{O}(n)$ steps - n defined as the size of *iterators*. 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 *key* value in all iterators and store the index of this *TrieIterator* in p. Then we move the *TrieIterator* at $p+1$ to the least upper bound of this *max* (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 and we are done - the *LeapfrogJoin* is initialized - or we arrived at a state in which the *iterators* array is sorted according to *key* and can proceed as in the original *LeapfrogJoin.init* method. To apply this optimization one replaces the call to *sort* in line 11 with the procedure explained above[2].

The second Leapfrog Triejoin specific optimization is to change the `ArrayTrieIterable.TrieIteratorImpl.next` method. This method moves the iterator to the next value on the same level of the trie. Hence, it generally runs in $\mathcal{O}(\log n)$, n being the number of tuples in the relationship, because it needs to find the least upper bound of *key + 1*. However, under the assumption that all tuples are unique - which is fulfilled for the use-case of an edge relationship - the last level of the trie is unique. Hence, we can move to the next value by simply increasing the position by one, which is an operation in $\mathcal{O}(1)$.

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 - currently at 60 values. We experimented with values from 0 to 400 and found that 60 was optimal but even

[2]The implementation of Scala's array sort for objects is slow because it copies the array twice and casts the values to *Java.Object* such that it can use Java's sorting methods. Before we applied the sorting optimization above, we replaced Scala's sort method with an optimized insertion sort, which was faster than Scala's sorting method - the *iterator* array contains normally at most 20 items.

going as high as 120 values would not change the performance much.

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 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. The implementation is straightforward and can be found in our repository in `leapfrogTriejoin.ExposedArrayColumnVector`; we implemented it only for the `Long` datatype.

We found many standard optimizations and Scala specific optimizations to be really useful. Most likely these are the optimizations that brought the biggest performance improvements. However, they are well-known, so we mention them only in the table 3. For Scala specific optimizations one can find good explanations at [12].

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.moveToNextTuple*
- predicating the *action* variable in *LeapfrogTriejoin.moveToNextTuple*

Finally, we believe that code generation for specific queries that combines the functionality of *LeapfrogTriejoin*, *LeapfrogJoin* and *ArrayTrieIterator* 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.

5.2 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.find(
  """(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 4: Example usage of a WCOJ to find triangles in graph.

As one can see in line 16 of listing 4, we support a clean and precise DSL to match patterns in graphs. This DSL is inspired by GraphFrames [13]. 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, specify different relationships to be used as input for each edge of the pattern. 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.

5.3 Spark integration

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, this includes a *LogicalPlan*, a *Strategy* to convert this logical operator into a *SparkPlan*, a physical operator to materialize RDD's in sorted fashion into a columnar representation that supports a least upper bound operation in $\mathcal{O}(\log n)$ and finally a general implementation of Spark's *zipPartition* operation to support more than 5 children. We explain these components below; before we outline some limitations of our integration.

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 [26] 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.

To allow the user to call *findPattern* (as defined in section 5.2) on a *Dataset*, we added a new class to the package `org.apache.spark.sql` called `WCOJFunctions` and an implicit conversion from *Dataset* to `WCOJFunctions`. This function parses the pattern - the necessary code is borrowed from GraphFrame [13] - into a representation of a WCOJ join, `JoinSpecification`, and creates new `AttributeReferences` named accordingly to the join variables.

Due to our decision not to build optimizer rules to introduce WCOJ's, we have to implement a new *LogicalPlan* to represent WCOJ's. This plan has as many children as relationships joined by the WCOJ. Its *output* are the `AttributeReferences` produced by `findPattern`.

The physical operator for WCOJ's, `WCOJExec`, is an implementation of the trait `SparkPlan`. It expects as many children as the relationships in the join; these children have to support a Trieiterator interface as specified in section 5.1. The biggest challenge in the implementation of `WCOJExec` is that Spark does not support to zip the partitions of more than 5 RDD's. However, to operate synchronously on the partitions of all children, we need this operation for an arbitrary number of children. Fortunately, although, not publicly available, Spark does implement *zipPartitions* in a general manner in the abstract class `ZippedPartitionsBaseRDD`. Hence, supporting this functionality is a simple matter of subclassing this class publicly to expose its generality to our code (see `sparkIntegration.GeneralZippedPartitionsRDD`).

We need to materialize the input RDD's for `WCOJExec`, to support the Trieiterator interface. This is done by `ToTrieIterableRDD`, a `SparkPlan`, with a single child. It iterates over the child and stores all tuples in a columnar representation, a `ColumnarBatch`[3] [4]. Furthermore, our Trieiterator

[3]To all but highly experienced Spark developers, the construct used to allow us to create an RDD which controls how its partitions are stored might look unintuitive. Normally, the type parameter of the `RDD` class is used to refer to the type of its elements, e.g. Sparks Catalyst module uses mostly `RDD[InternalRow]`. However, it is possible - and practised in the Spark project itself - to use this type parameter to give the type of the partitions of an RDD. This is what we do in the class `TrieIterableRDD`, which at the same time is a subclass of `RDD[InternalRow]`, so it can be used in Catalyst.

[4]Later we noticed the RDD function `glom` which allows to operate on the partitions of RDD's an array. However, by then we had already implemented our own solution and decided to keep it for better control.

implementation demands that the tuples are sorted. Therefore, `ToTrieIterableRDD` requires its child to be ordered using the `requiredChildOrdering` functionality of Spark.

The last component of our Spark integration is a simple `Strategy` to translate a `WCOJ` logical operator into a `WCOJExec` physical operator. This is a one-to-one tranlation from `WCOJ` to `WCOJExec` as an implementation of Spark's abstract class `SparkStrategy`. Additionally, this `Strategy` also introduces a `ToTrieIterableRDD` operation before the WCOJ for every child.

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

5.4 Operator to create a CSR from a DataFrame

To be defined and written.

5.5 Graph pattern sequential version (*graph-pattern-seq*)

5.5.1 Combining LFTJ with CSR

For our graph pattern matching specialized *LeapfrogTriejoin* version we choose CSR (see section 2.2) 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 5.1[5]) 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 5.4[6].

We use the variables *depth*, *srcPosition* and *dstPosition* to store if we are operating on the source or destination level and the positions of the iterator on the respective level. We call the two arrays of the CSR datastructure *dst* for the array storing all edge destinations and *dstIndices* for the array storing indices into *dst* (refer to **??** for a complete explanation of these arrays). The *open* function does nothing if we open the first level; if we open the second level, it sets *dstPosition* to *dstIndices(srcPosition)*. Additionally, it updates *depth*. The *up* function only updates *depth*. The *next* method increases *srcPosition* or *dstPosition* by one depending on *depth*. The *seek(key)* method sets *srcPosition* to *key* if *depth* equals 0 or uses binary search to find *key* in *dst.slice(dstPosition, dstIndices[srcPosition + 1])* and then sets *dstPosition* accordingly. The *key* function returns *srcPosition* on the first level and *dst[dstPosition]* if *depth* equals 1. The *atEnd* function is: *if (depth == 0) srcPosition == dstIndices.length - 1 else dstPosition == dstIndices(srcPosition + 1)*.

<Alternative representation of the *TrieIterator* implementation as table. Let me know what you

[5]To be rewritten and integrated

[6]To be written.

prefer. I think the text is better after seeing both.>

Method	1st level	2nd level
open	update depth	dstPosition ← dstIndices[srcPosition]; update depth
up	update depth	update depth
next	srcPosition++	dstPosition++
seek(k)	srcPosition ← k	dstPosition ← lub(k, dst[dstPosition:dstIndices[srcPosition +1]]
key	srcPosition	dst[dstPosition]
atEnd	srcPosition = dstIndices.length - 1	dstPosition = dstIndices(srcPosition + 1)

Table 4: Tabular summary of *TrieIterator* implementation based on CSR. *lub* is *leastUpperBound*. *dst[<start>:<end>]* slices 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 *dstIndices(x) == dstIndices(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.

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* [11]. Note that searches on the second level are fast, due to the fact that most graphs have a low outdegree (see section 2.1). 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. 2. 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 6.

5.5.2 Exploiting low average outdegrees

In our study of real-world graphs (**??**), we show that most real-world graphs have a small, average outdegree. The outdegree over all graphs is TODO and the maximum average outdegree of a single graph is TODO at TODO. These results lead 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 *Leapfrog joins* directly in one go; instead of, generating one value at-the-time in an iterator like fashion as described in the original paper [35]. This is beneficial because it makes better use of data locality; we elaborate this statement in a later paragraph.

We structure the remainder of this section as follows. First, we shortly reiterate the most important facts about *Leapfrog joins* for this chapter (TODO just organize both sections behind each other?). 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

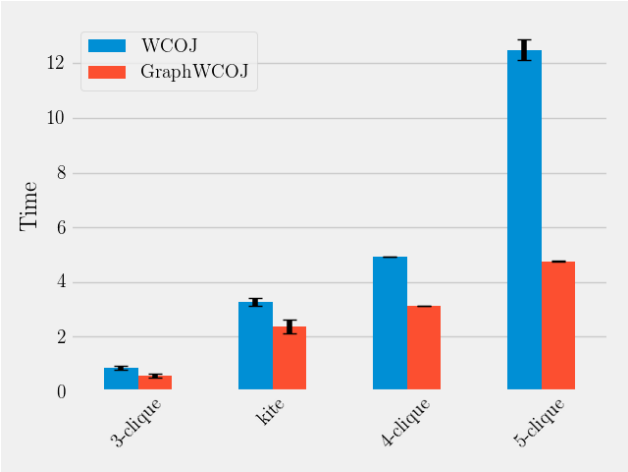


Figure 2: Barchart comparing join time of `WCOJ` and `GraphWCOJ` on multiple queries on `SNB-sf1`

to the original Leapfrog Triejoin. Finally, we present a short experiment showing the performance gains of this optimization.

Leapfrog joins build the intersection between multiple adjacency lists. This is done in an iterator-like fashion in their *leapfrog_next* 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 claim 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.

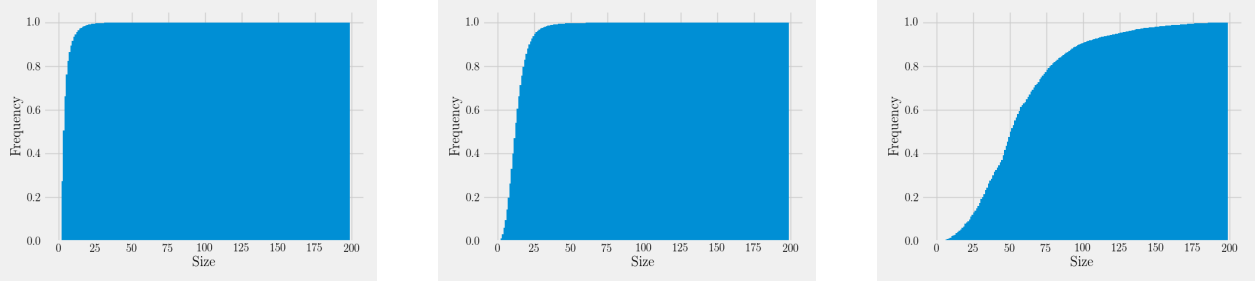
To determine the best algorithms to build the n-way intersection in the *Leapfrog joins*, we run some experiments to characterize the workload. Towards this goal, we log the size of the full intersection, the size of the smallest iterator participating and the size of the largest intersection between the smallest iterator and any other iterator on 5-clique queries on `SNB-sf-1`. Figure 3 depicts these metrics as cumulative histograms. In the next paragraphs, we point out the most important observations in each of these graphs.

Figure 3c 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 3b 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. 3a): the size is less than 5 in 80% of all intersections and increases similarly steep to 100%. The maximum is a little 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) Total intersection

(b) Largest intersection including smallest iterator

(c) Smallest iterator

Figure 3: 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 *Leapfrog join*. We choose to use pairwise intersections over multi-way intersection algorithms for their simpler, linear memory access patterns.

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[7]. 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. This algorithm guarantees to find the intersection between two iterators in the asymptotical fastest way. 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 *Leapfrog join* are on their first level, the intersection is near |V|. In this case, we fall back to the original *Leapfrog join*.
- 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 *Leapfrog join* 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.

The carefully chosen operations explained above are faster than the original *Leapfrog join* algorithm for two reasons. First, although, the original algorithm uses an asymptotical better n-way intersection, multiple binary intersections are preferable with the rather small adjacency lists of graph workloads. In particular, in our situation where the size of the first binary intersection is already very close to the size of the final intersection.

Second, the *Leapfrog join* 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

[7]We take advantage of the fact that CSR allows us to determine the size of iterators cheaply (see **??**)

value. Our approach touches the adjacency lists exactly once per *Leapfrog* 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.

Figure 4 shows that a materialized *Leapfrog join* out-performance better than the original algorithm. As expected, the optimization is more powerful for bigger queries because they work with more adjacency lists per *Leapfrog join*, e.g. for a triangle each *Leapfrog join* intersects only two iterators while for a `5-clique` each join handles 5 adjacency lists. Anyhow, even for the triangle query, we see a small but clear improvement and a lower error due to better cache use. We refer the reader to our experiment section (6) for further experiments on our graph specialized WCOJ and detailed descriptions of the datasets and queries used.

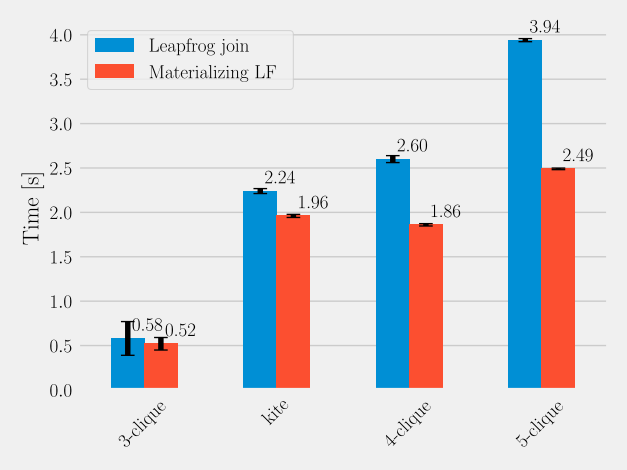


Figure 4: Barchart showing `GraphWCOJ` with and without *Leapfrog join* materialization enabled for different queries on `SNB-sf1`.

6 Experiments

TODO introduction

6.1 Setup

6.1.1 Hardware and Software

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 (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.

Name	Variant	Vertices	Edges	Source
SNB	sf1		453.032	[21]
Amazon	0302	262,111	1,234,877	[22]
	0601	403,394	3,387,388	[22]
Twitter	sc-d	81,306	1,768,135	[22]
	sc-u	TODO	TODO	[22]
LiveJournal		4,847,571	68,993,773	[22]
Orkut		3,072,441	117,185,083	[22]

Table 5: A summary of all datasets mentioned in the thesis. Explanation of them and for the variants is given in running text.

6.1.2 Algorithms

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

The other two algorithms are Spark's versions of *BroadcastHashJoin* and *SortmergeJoin*. We compare them against the sequential version of *LFTJ* and *GraphWCOJ* in **??** and their scaling with *GraphWCOJ* in **??**. We adjust the `"spark.sql.autoBroadcastJoinThreshold"` parameter to control if Spark is using a `BroadcastHashJoin` or a `SortMergeJoin`.

6.1.3 Datasets

We run the majority of our experiments on two datasets from 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.

TODO add Vertices and edges numbers

The SNB benchmark [21] 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. After generation 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. We include the exact parameter used for generation in our repository under `experiments/snb/params.txt`.

The Amazon co-purchasing network contains edges between products that have been purchased together and hence are closely related to each other [22]. 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, 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 [11, 3] but [32] points out that the biggest graphs are actually graphs like the aforementioned Amazon graph

Name	Parameters	Vertices	Edges	Example pattern
`triangle`	NA	3	3	a → b; a → c; b → c
`n-clique`	# vertices	n	$1/2 \times n \times (n-1)$	see above
`n-cycle`	# vertices	n	n	a → b; b → c; c → z; z → a
`n-s-path`	# edges / selectivity	n	$n-1$	a → b; b → c; c → z
`house`	NA	5	9	a → b; a → c; a → d; b → c; b → d; c → d; c → e; d → e
Filters				
`distinct`				a ≠ b; a ≠ c; a ≠ d; b ≠ c; ...
`lt`				a <b; b <c; c <d; ...

Table 6: Summary of patterns and filters used.

containing purchase information.

To allow comparisons with former work, we run a subset of our experiments on the Twitter social circle network from [22]. 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 graph is originally directed but for some experiments, we add reversed edges to make the graph undirected - again for comparison with former work. We call this graph `Twitter-sc-d` and `Twitter-sc-u` for the directed respectively undirected variant.

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

6.1.4 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 [30, 11]. WCOJ's also have been successfully applied to selective path queries in [30]; however, this result have not been reproduced by any other paper.

To most of our queries, we apply a filter 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. Because otherwise, one gets the same clique in all possible orders in the output, 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. The less known queries are also detailed in text. Patterns and filters might be used in all possible combinations, we name the resulting query *<pattern>-<filter>*, e.g. *triangle-lt*.

For a selective path query, we first select two sets of nodes with respect to the *selectivity* parameter. Then we search for all path 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.

6.2 Baseline: `BroadcastHashJoin` vs `seq`

In this experiment, we compare the runtime of our sequential Leapfrog Triejoin implementation, `seq`, with the runtime of Spark's `BroadcastHashjoin`. Towards, this goal we ran all queries

from table 6 on our three main datasets: `ama-0302`, `ama-0601` and `snb-sf1`. The clique patterns are combined with the less-than filter, `n-clique-lt`, and the cycle pattern with the distinct filter, `n-cycle-distinct`. These seem to be the most realistic setups because cliques are fully symmetric and one wants to avoid redundant results. For cycles, the less-than filter is too restrictive because it excludes cycles for which $a < b > c$. We show our results in table 7 and in barcharts (fig. 5, 6 and 7). Section 6.2.2 analyzes the results.

Our experiment measures the time it takes to perform a `count` on the cached dataset using `BroadcastHashjoin` and `seq`. 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, materialization and copying the results of our join from a Scala `Array` into the `UnsafeInternalRow` format expected by Spark. TODO sorting not yet (data is presorted in files) This section is focused on comparing the runtimes excluding the setup time - rational given in section 6.2.1.

6.2.1 Experiment Rationale

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 our final implementation `dist` is meant to cache the readily sorted and formatted edge tables and reuse it for multiple queries. We anticipate that this is necessary to benefit from WCOJ's in general. Furthermore, we optimize the setup times in later implementation. Hence, the current setup code is much slower than the one we expect to use for later implementations.

Question: Why is the time to copy results into `UnsafeInternalRow` format for WCOJ counted as setup time?
Answer: It is time solely spent for integration with Spark and not Leapfrog Triejoin specific. Furthermore, it could be avoided by working directly on the `UnsafeInternalRow` format within our `seq` implementation. However, this would require us to work with unmanaged memory (the `UnsafeInternalRow` interface is slower than working on `Arrays`) and we deem this as an unnecessary engineering overhead.

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 `triangle` and `5-clique`: with code generation Spark takes 3.1 and 4.2 seconds without 14 and 16.

6.2.2 Analysis

For now, we settle to simply point out the most important observations and postpone deeper analysis, e.g. influence of dataset size and characteristics, to experiments run against our later implementations, i.e. `seq-graph-pattern` and `dist`.

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`. Unfortunately, we are (TODO currently) not able to provide `n-cycle` results for the `SNB-sf1` dataset, due to the fact that `BroadcastHashjoin`'s take more than 22 hours for the `6-cycle` which blocked our experiments. `5-cycle` runs at the moment.

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. We will run the path queries with a different selectivity on the two input vertice sets to confirm this hypothesis.

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-lt` 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-distinct` 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`

6.3 Scaling of *Graph*WCOJ

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

compare schemes compare dataset sizes compare query scaling

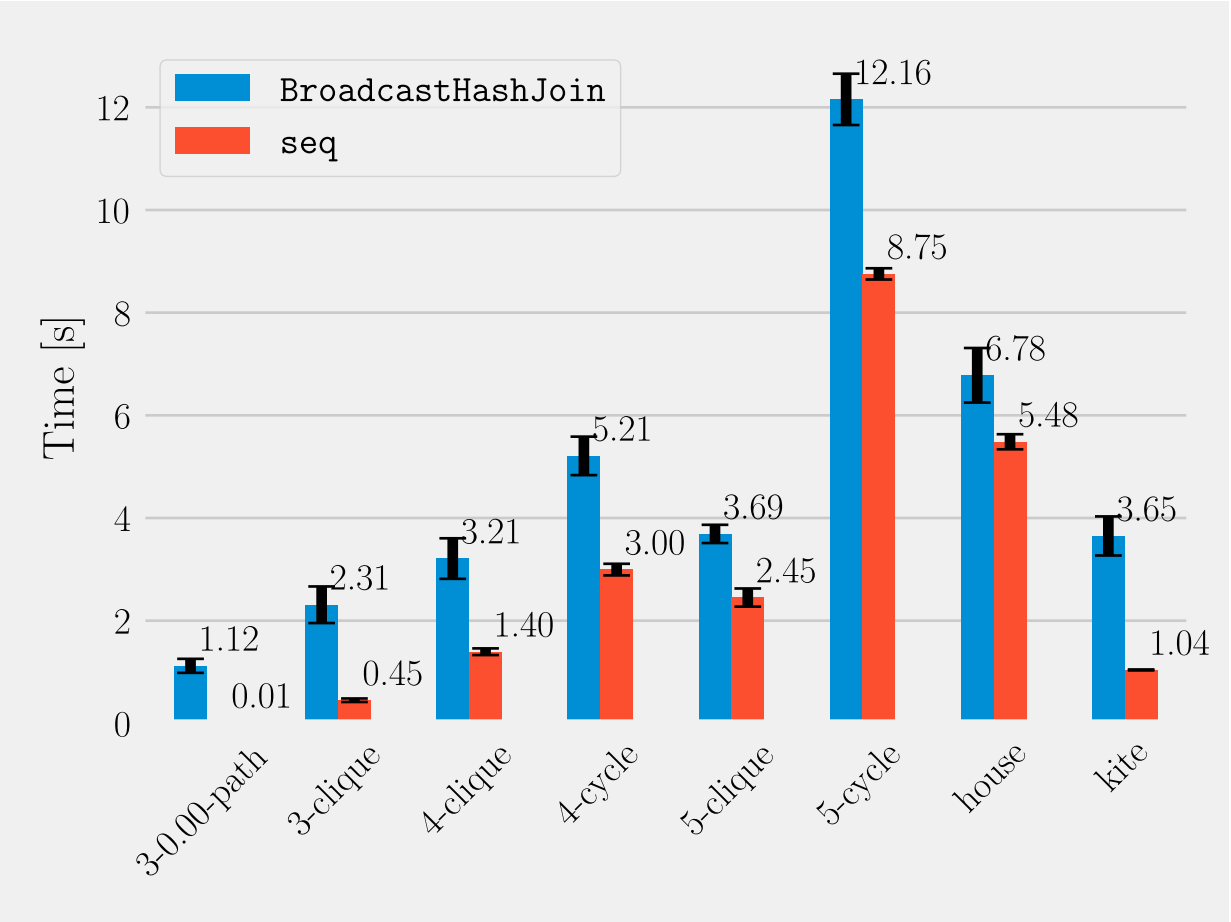


Figure 5: `seq` vs `BroadcastHashJoin` on `Amazon-0302`

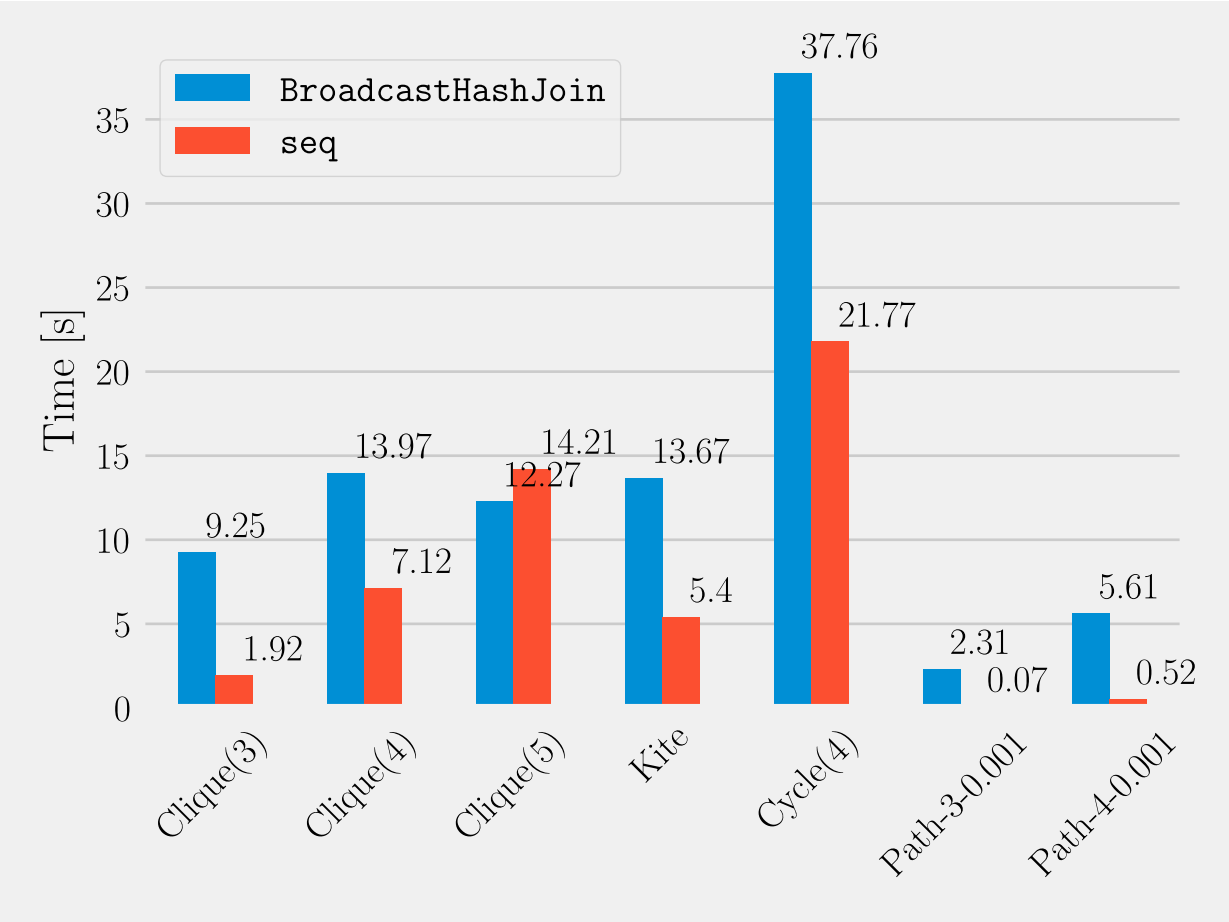


Figure 6: `seq` vs `BroadcastHashJoin` on `Amazon-0601`

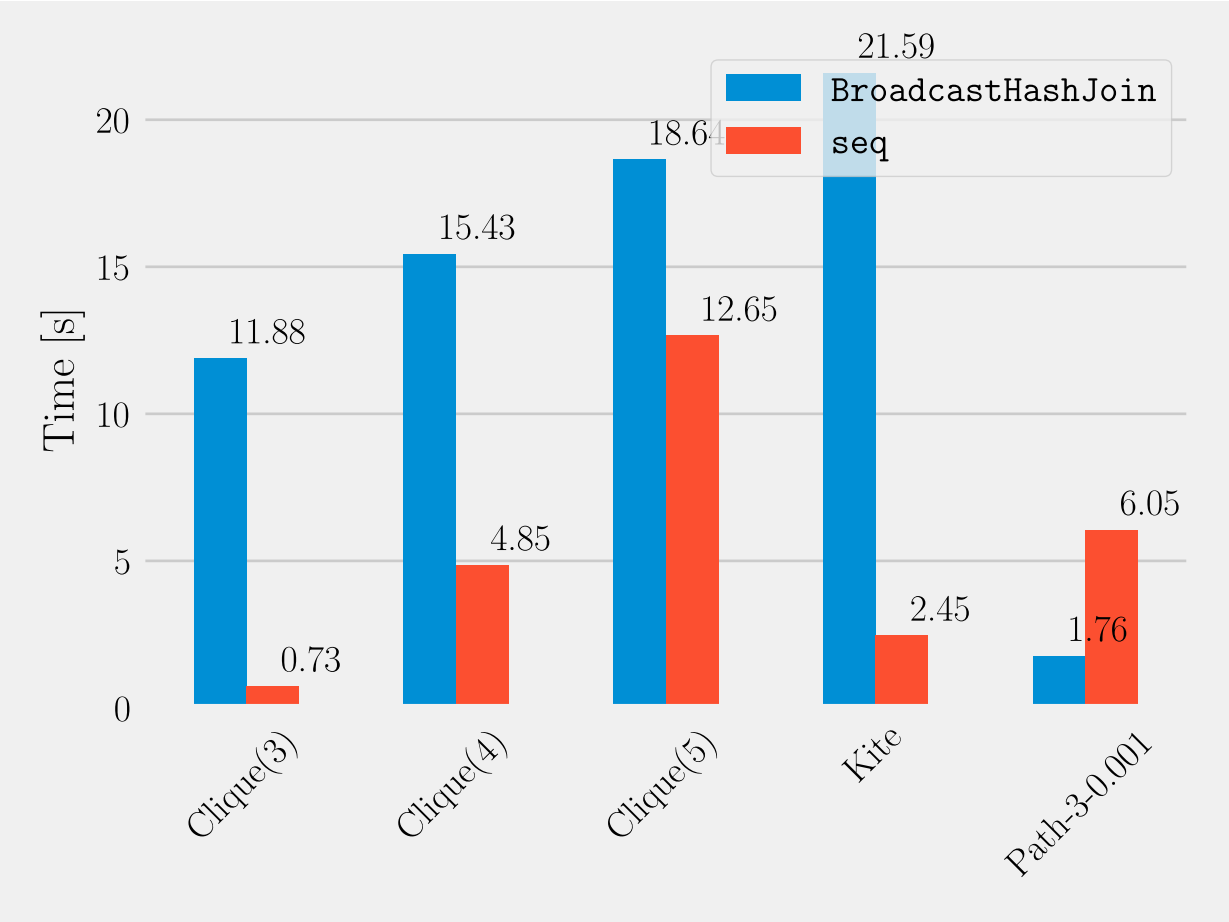


Figure 7: `seq` vs `BroadcastHashJoin` on `SNB-sf1`

Query	# Result	BroadcastHashJoin	seq	setup	Speedup
Clique(3)	188,705	3.08	0.49	2.51	6.3
Clique(4)	47,824	3.85	1.25	4.55	3.1
Clique(5)	7,064	4.22	2.32	7.60	1.8
House	2,941,664	7.87	5.66	6.18	1.4
Kite	220,637	3.82	1.04	6.73	3.7
Cycle(4)	3,076,324	4.90	2.75	3.40	1.8
Cycle(5)	6,389,425	13.15	8.28	4.87	1.6
Cycle(6)	11,682,732	38.00	27.62	6.79	1.4
Path-3-0.001	2,092	1.36	0.01	0.89	136.0
Path-4-0.001	8,829	3.11	0.03	1.58	103.7

Query	# Result	BroadcastHashJoin	seq	setup	Speedup
Clique(3)	1,137,579	9.25	1.92	6.23	4.8
Clique(4)	911,548	13.97	7.12	11.74	2.0
Clique(5)	585,171	12.27	14.21	19.23	0.9
House	177,745,510	66.53	57.37	23.45	1.2
Kite	3,672,375	13.67	5.40	15.93	2.5
Cycle(4)	45,175,984	37.76	21.77	14.12	1.7
Cycle(5)	263,436,965	227.97	115.60	24.96	2.0
Cycle(6)	1,484,438,088	2020.04	872.62	83.12	2.3
Path-3-0.001	98,929	2.31	0.07	2.30	33.0
Path-4-0.001	922,888	5.61	0.52	4.25	10.8

Query	# Result	BroadcastHashJoin	seq	setup	Speedup
Clique(3)	540,225	11.88	0.73	1.09	16.3
Clique(4)	260,786	15.43	4.85	2.33	3.2
Clique(5)	40,137	18.64	12.65	3.08	1.5
House	100,206,468	300.13	85.18	6.58	3.5
Kite	1,780,235	21.59	2.45	2.20	8.8
Cycle(4)	232,636,376	636.77	162.89	10.68	3.9
Path-3-0.001	65,290,479	1.76	6.05	3.28	0.3
Path-4-0.001	7,235,522,926	111.73	655.52	293.27	0.2

Table 7: Runtimes for `BroadcastHashJoin` and `seq`. The speedup is calculated between join times and excludes setup. From top to bottom for dataset: `ama-0302`, `ama-0601` and `snb-sf1`. All times in seconds.

Partitioning	Query	Parallelism	Time	Scaling
FirstVariablePartitioningWithWorkstealing	3-clique	1	0.0	1.000000
FirstVariablePartitioningWithWorkstealing	3-clique	2	0.0	1.932485
FirstVariablePartitioningWithWorkstealing	3-clique	4	0.0	3.959073
FirstVariablePartitioningWithWorkstealing	3-clique	8	0.0	7.554412
FirstVariablePartitioningWithWorkstealing	3-clique	16	0.0	14.854730
FirstVariablePartitioningWithWorkstealing	3-clique	32	0.0	29.560705
FirstVariablePartitioningWithWorkstealing	3-clique	64	0.0	47.041242
FirstVariablePartitioningWithWorkstealing	3-clique	96	0.0	57.853794
Shares	3-clique	1	0.0	1.000000
Shares	3-clique	2	0.0	1.933259
Shares	3-clique	4	0.0	3.745935
Shares	3-clique	8	0.0	3.575031
Shares	3-clique	16	0.0	7.093603
Shares	3-clique	32	0.0	13.348219
Shares	3-clique	64	0.0	10.216631
Shares	3-clique	96	0.0	14.338119

Table 8: g

6.3.1 Results

6.3.2 Analysis

References

[1] Foto N Afrati and Jeffrey D Ullman. "Optimizing multiway joins in a map-reduce environment." In: *IEEE Transactions on Knowledge and Data Engineering* 23.9 (2011), pp. 1282–1298.

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

[3] Khaled Ammar et al. "Distributed evaluation of subgraph queries using worst-case optimal low-memory dataflows." In: *Proceedings of the VLDB Endowment* 11.6 (2018), pp. 691–704.

[4] Renzo Angles et al. "G-CORE: A core for future graph query languages." In: *Proceedings of the 2018 International Conference on Management of Data*. ACM. 2018, pp. 1421–1432.

[5] Michael Armbrust et al. "Spark sql: Relational data processing in spark." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1383–1394.

[6] Albert Atserias, Martin Grohe, and Dániel Marx. "Size bounds and query plans for relational joins." In: *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*. IEEE. 2008, pp. 739–748.

[7] Paul Beame, Paraschos Koutris, and Dan Suciu. "Communication steps for parallel query processing." In: *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*. ACM. 2013, pp. 273–284.

[8] Paul Beame, Paraschos Koutris, and Dan Suciu. "Skew in parallel query processing." In: *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM. 2014, pp. 212–223.

[9] Yingyi Bu et al. "HaLoop: efficient iterative data processing on large clusters." In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 285–296.

[10] Aydin Buluç et al. "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks." In: *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM. 2009, pp. 233–244.

[11] Shumo Chu, Magdalena Balazinska, and Dan Suciu. "From theory to practice: Efficient join query evaluation in a parallel database system." In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 63–78.

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

[13] Ankur Dave et al. *GraphFrame*. 2016. URL: `https://databricks.com/blog/2016/03/03/introducing-graphframes.html` (visited on 02/18/2019).

[14] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[15] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. "Adaptive set intersections, unions, and differences." In: *In Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA*. Citeseer. 2000.

[16] Joseph E Gonzalez et al. "GraphX: Graph Processing in a Distributed Dataflow Framework." In: *OSDI*. Vol. 14. 2014, pp. 599–613.

[17] Pankaj Gupta et al. "Real-time twitter recommendation: Online motif detection in large dynamic graphs." In: *Proceedings of the VLDB Endowment* 7.13 (2014), pp. 1379–1380.

[18] Daniel Halperin et al. "Demonstration of the Myria big data management service." In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM. 2014, pp. 881–884.

[19] Frank K. Hwang and Shen Lin. "A simple algorithm for merging two disjoint linearly ordered sets." In: *SIAM Journal on Computing* 1.1 (1972), pp. 31–39.

[20] Paraschos Koutris, Paul Beame, and Dan Suciu. "Worst-case optimal algorithms for parallel query processing." In: *19th International Conference on Database Theory (ICDT 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.

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

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

[23] Jure Leskovec and Rok Sosič. "Snap: A general-purpose network analysis and graph-mining library." In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 8.1 (2016), p. 1.

[24] Grzegorz Malewicz et al. "Pregel: a system for large-scale graph processing." In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 135–146.

[25] Frank McSherry, Michael Isard, and Derek G Murray. "Scalability! But at what {COST}?" In: *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*. 2015.

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

[27] Derek G Murray et al. "Naiad: a timely dataflow system." In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 439–455.

[28] Hung Q Ngo, Christopher Ré, and Atri Rudra. "Skew strikes back: New developments in the theory of join algorithms." In: *arXiv preprint arXiv:1310.3314* (2013).

[29] Hung Q Ngo et al. "Worst-case optimal join algorithms." In: *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*. ACM. 2012, pp. 37–48.

[30] Dung Nguyen et al. "Join processing for graph patterns: An old dog with new tricks." In: *Proceedings of the GRADES'15*. ACM. 2015, p. 2.

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

[32] Semih Salihoglu and M Tamer Özsu. "Response to "Scale Up or Scale Out for Graph Processing"." In: *IEEE Internet Computing* 22.5 (2018), pp. 18–24.

[33] Christian 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).

[34] William F Tinney and John W Walker. "Direct solutions of sparse network equations by optimally ordered triangular factorization." In: *Proceedings of the IEEE* 55.11 (1967), pp. 1801–1809.

[35] Todd L Veldhuizen. "Leapfrog triejoin: A simple, worst-case optimal join algorithm." In: *arXiv preprint arXiv:1210.0481* (2012).

[36] Matei Zaharia et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.