Join Processing for Graph Patterns: An Old Dog with New Tricks

Dung Nguyen[1] Molham Aref[1] Martin Bravenboer[1] George Kollias[1]
Hung Q. Ngo[2] Christopher Ré[3] Atri Rudra[2]

[1]LogicBlox, [2]SUNY Buffalo, and [3]Stanford

ABSTRACT

Join optimization has been dominated by Selinger-style, pairwise optimizers for decades. But, Selinger-style algorithms are asymptotically suboptimal for applications in graphic analytics. This suboptimality is one of the reasons that many have advocated supplementing relational engines with specialized graph processing engines. Recently, new join algorithms have been discovered that achieve optimal worst-case run times for any join or even so-called beyond worst-case (or instance optimal) run time guarantees for specialized classes of joins. These new algorithms match or improve on those used in specialized graph-processing systems. This paper asks *can these new join algorithms allow relational engines to close the performance gap with graph engines?*

We examine this question for graph-pattern queries or join queries. We find that classical relational databases like Postgres and MonetDB or newer graph databases/stores like Virtuoso and Neo4j may be orders of magnitude slower than these new approaches compared to a fully featured RDBMS, LogicBlox, using these new ideas. Our results demonstrate that an RDBMS with such new algorithms can perform as well as specialized engines like GraphLab – while retaining a high-level interface. We hope our work adds to the ongoing debate of the role of graph accelerators, new graph systems, and relational systems in modern workloads.

1. INTRODUCTION

For the last four decades, Selinger-style pairwise-join enumeration has been the dominant join optimization paradigm [12]. Selinger-style optimizers are designed for joins that do not filter tuples, e.g., primary-key-foreign-key joins that are common in OLAP. Indeed, the result of a join in an OLAP query plan is often no smaller than either input relation. In contrast, in graph applications, queries search for structural patterns, which filter the data. These regimes are quite different, and not surprisingly there are separate OLAP and graph-style systems. Increasingly, analytics workloads contain both traditional reporting queries and graph-based queries [11]. Thus, it would be desirable to have one engine that is able to perform well for join processing in both of these different analytics settings.

GRADES'15, May 31 - June 04 2015, Melbourne, VIC, Australia.
© 2015 ACM. ISBN 978-1-4503-3611-6/15/05 ...$15.00.
DOI: http://dx.doi.org/10.1145/2764947.2764948.

Unifying these two approaches is challenging both practically and theoretically. Practically, graph engines offer orders of magnitude speedups over traditional relational engines, which has led many to conclude that these different approaches are irreconcilable. Indeed, this difference is fundamental: recent theoretical results suggest that Selinger-style, pair-wise join optimizers are *asymptotically suboptimal* for this graph-pattern queries [3,9,14]. The suboptimality lies in the fact that Selinger-style algorithms only consider pairs of joins at a time, which leads to runtimes for cyclic queries that are asymptotically slower by factors in the size of the data, e.g., $\Omega(\sqrt{N})$-multiplicative factor worse on a database with N tuples. Nevertheless, there is hope of unifying these two approaches for graph pattern matching as, mathematically, *(hyper)graph pattern matching is equivalent to join processing*. Recently, algorithms have been discovered that have strong theoretical guarantees, such as optimal runtimes in a worst-case sense or even instance optimally. As database research is about three things: performance, performance, and performance, the natural question is:

> *To what extent do these new join algorithms speed up graph workloads in an RDBMS?*

To take a step toward answering this question, we embed these new join algorithms into the LogicBlox (LB) database engine, which is a fully featured RDBMS that supports rich queries, transactions, and analytics. We perform an experimental comparison focusing on a broad range of analytics workloads with a large number of state-of-the-art systems including row stores, column stores, and graph processing engines. Our technical contribution is the first empirical evaluation of these new join algorithms. At a high-level, our message is that these new join algorithms provide substantial speedups over both Selinger-style systems and graph processing systems for some data sets and queries. Thus, they may require further investigation for join and graph processing.

We begin with a brief, high-level description of these new algorithms, and then we describe our experimental results.

An Overview of New Join Algorithms. Our evaluation focuses on two of these new-style algorithms, LeapFrog TrieJoin (LFTJ) [14], a worst-case optimal algorithm, and Minesweeper (MS), a recently proposed "beyond-worst-case" algorithm [8].

LFTJ is a multiway join algorithm that transforms the join into a series of nested intersections. LFTJ has a running time that is worst-case optimal, which means for every query there is some family of instances so that any join algorithm takes as least much time as LFTJ does to answer that query. These guarantees are nontrivial; in particular, any Selinger-style optimizer is slower by a factor that depends on the size of the data, e.g., by a factor of $\Omega(\sqrt{N})$ on a database with N tuples. Such algorithms were discovered recently [9, 14], but LFTJ has been in LogicBlox for several years.

Minesweeper's main idea is to keep careful track of every comparison with the data to infer where to look next for an output. This allows Minesweeper to achieve a so-called "beyond worst-case guarantee" that is substantially stronger than a worst-case running time guarantees: for a class called comparison-based algorithms, containing all standard join algorithms including LFTJ, beyond worst-case guarantees that *any* join algorithm takes no more than a constant factor more steps on *any* instance. Due to indexing, the runtime of some queries can even be *sublinear* in the size of the data. However, these stronger guarantees only apply for a limited class of acyclic queries (called β-acyclic).

Benchmark Overview. The primary contribution of this paper is a benchmark of these new style algorithms against a range of competitor systems on graph-pattern matching workloads. To that end, we select a traditional row-store system (Postgres), a column-store system (MonetDB) and graph systems (virtuoso, neo4j, and graphlab). We find that for cyclic queries these new join systems are substantially faster than relational systems and competitive with graph systems. We find that LFTJ is performant in cyclic queries on dense data, while Minesweeper is superior for acyclic queries.

Contributions. This paper makes two contributions:

1. We describe the first practical implementation of a beyond worst-case join algorithm.
2. We perform the first experimental validation that describes scenarios for which these new algorithms are competitive with conventional optimizers and graph systems.

2. BACKGROUND

We recall background on graph patterns, join processing, and hypergraph representation of queries along with the two new join algorithms that we consider in this paper.

2.1 Join query and hypergraph representation

A (natural) *join query* Q is specified by a finite set atoms(Q) of relational symbols, denoted by $Q = \Join_{R \in \text{atoms}(Q)} R$. Let vars$(R)$ denote the set of attributes in relation R, and

$$\text{vars}(Q) = \bigcup_{R \in \text{atoms}(Q)} \text{vars}(R).$$

Throughout this paper, let $n = |\text{vars}(Q)|$ and $m = |\text{atoms}(Q)|$. For example, in the following so-called *triangle query*

$$Q_\triangle = R(A,B) \Join S(B,C) \Join T(A,C).$$

we have vars$(Q_\triangle) = \{A,B,C\}$, vars$(R) = \{A,B\}$, vars$(S) = \{B,C\}$, and vars$(T) = \{A,C\}$. The structure of a join query Q can be represented by a hypergraph $\mathcal{H}(Q) = (\mathcal{V},\mathcal{E})$, or simply $\mathcal{H} = (\mathcal{V},\mathcal{E})$. The vertex set is $\mathcal{V} = \text{vars}(Q)$, and the edge set is defined by

$$\mathcal{E} = \{\text{vars}(R) \mid R \in \text{atoms}(Q)\}.$$

Notice that if the query hypergraph is exactly finding a pattern in a graph. For so-called *α-acyclicity* [2, 4], the celebrated Yannakakis algorithm [16] runs in linear-time (in data complexity). On graph databases with binary relations, both $\alpha-$ and $\beta-$ acyclic can simply be thought of as the standard notion of acyclic.

Worst-case Optimal Algorithm. Given the input relation sizes, Atserias, Grohe, and Marx [3] derived a linear program that could be used to upper bound the worst-case (largest) output size in number of tuples of a join query. For a join query, Q we denote this bound $\mathsf{AGM}(Q)$. For completeness, we describe this bound in Appendix A. Moreover, this bound is tight in the sense that there exists a family of input instances for any Q whose output size is $\Omega(\mathsf{AGM}(Q))$. Then, Ngo, Porat, Ré, and Rudra (NPRR) [9] presented an algorithm whose runtime matches the bound. This algorithm is thus *worst-case optimal.* Soon after, Veldhuizen [14] used a similar analysis to show that the LFTJ algorithm – a simpler algorithm already implemented in LogicBlox Database engine – is also worst-case optimal. A simpler exposition of these algorithms was described [10] and formed the basis of a recent system [1].

Beyond Worst-case Results. Although NPRR may be optimal for worst-case instances, there are instances on which one can improve its runtime. To that end, the tightest theoretical guarantee is so-called *instance optimality* which says that the algorithm is up to constant factors no slower than *any algorithm*, typically, with respect to some class of algorithms. These strong guarantees had only been known for restricted problems [5]. Recently, it was shown that if the query is *β-acyclic*, then a new algorithm called Minesweeper (described below) is log-instance optimal[1] with respect to the class of comparison-based joins, a class which includes essentially all known join and graph processing algorithms. Theoretically, this result is much stronger–but this algorithm's performance has not been previously reported in the literature.

2.2 The Leapfrog Triejoin Algorithm

We describe the Leapfrog Triejoin algorithm. The main idea is to "leapfrog" over large swaths of tuples that cannot produce output. To describe it, we need some notation. For any relation R, an attribute $A \in \text{vars}(R)$, and a value $a \in \pi_A(R)$, define

$$R[a] = \pi_{\text{vars}(R)-\{A\}}(\sigma_{A=a}(R)).$$

That is $R[a]$ is the set of all tuples from R whose A-value is a.

At a high-level, the Leapfrog Triejoin algorithm can be presented recursively as shown in Algorithm 1. In the actual implementation, we implement the algorithm using a simple iterator interface, which iterates through tuples.

Algorithm 1 High-level view: Leapfrog Triejoin ($\mathsf{LFTJ}(Q)$)

```
Require: All relations $R$ are in atoms$(Q)$
 1: Let vars$(Q) = (A_1,\dots,A_n)$
 2: $L \leftarrow \bigcap_{R:A_1 \in \text{vars}(R)} \pi_{A_1}(R)$
 3: if $n = 1$ then
 4:     return $L$
 5: for each $a_1 \in L$ do
 6:                 $\rhd$ The following forms a new query on $(A_2,\dots,A_n)$
 7:     $Q_{a_1} \leftarrow \Big((\Join_{R:A_1 \in \text{vars}(R)} R[a_1]) \Join (\Join_{R:A_1 \notin \text{vars}(R)} R)\Big)$
 8:
 9:     $S_{a_1} = \mathsf{LFTJ}(Q_{a_1})$                    $\rhd$ Recursive call
10: return $\bigcup_{a_1 \in L}\{a_1\} \times S_{a_1}$
```

One can show that the Leapfrog Triejoin (LFTJ) algorithm runs in time $\tilde{O}(N + \mathsf{AGM}(Q))$ for any query Q [14]. The simplified version shown above appeared later [10].

2.3 The Minesweeper Algorithm

Consider the set of tuples that could be returned by a join (i.e., the cross products of all domains). Often many fewer tuples are part of the output than are not. Minesweeper's exploits this idea to

[1]Instance optimal up to a logarithmic factor in the database size [8]. This factor is unavoidable.

focus on quickly ruling out where tuples are not located rather than where they are. Minesweeper starts off by obtaining an arbitrary tuple $\mathbf{t} \in \mathbb{N}^{\text{vars}(Q)}$ from the output space, called a *free tuple* (also called a *probe point* in [8]). By probing into the indices storing the input relations, we either confirm that **t** is an output tuple or we get back $\tilde{O}(1)$ "gaps" or multi-dimensional rectangles inside which we know for sure *no* output tuple can reside. We call these rectangles *gap boxes*. The gap boxes are then inserted into a data structure called the *constraint data structure* (CDS). If **t** is an output tuple, then a corresponding (unit) gap box is also inserted in to the CDS. The CDS helps compute the next free tuple, which is a point in the output space not belonging to any stored gap boxes. The CDS is a specialized cache that ensures that we maximally use the information we gather about which tuples must be ruled out. The algorithm proceeds until the entire output space is covered by gap boxes. Algorithm 2 gives a high-level overview of how Minesweeper works.

Algorithm 2 High-level view: Minesweeper algorithm

```
1: CDS ← $\emptyset$                                  ▷ No gap discovered yet
2: while CDS can find t not in any stored gap do
3:     if $\pi_{\text{vars}(R)}(\mathbf{t}) \in R$ for every $R \in \text{atoms}(Q)$ then
4:         Report t and insert t as a gap into CDS
5:     else
6:         Query all $R \in \text{atoms}(Q)$ for gaps around t
7:         Insert those gaps into CDS
```

A key idea from [8] was the proof that the total number of gap boxes that Minesweeper discovers using the above outline is $\tilde{O}(|C|)$, where C is the minimum set of comparisons that *any* comparison-based algorithm must perform in order to work correctly on this join. Essentially all existing join algorithms such as Block-Nested loop join, Hash-Join, Grace, Sort-merge, index-nested, PRISM, double pipelined, are comparison-based (up to a log-factor for hash-join). For β-acyclic queries, Minesweeper is instance optimal up to an (unavoidable) $\log N$ factor.

3. EXPERIMENTS

3.1 Data sets, Queries, and Setup

The data sets we use for our experiments come from the SNAP network data sets collection [7]. We use the following data sets:

name	nodes	edges	triangle count
wiki-Vote	7,115	103,689	608,389
p2p-Gnutella31	62,586	147,892	2,024
p2p-Gnutella04	10,876	39,994	934
loc-Brightkite	58,228	428,156	494,728
ego-Facebook	4,039	88,234	1,612,010
email-Enron	36,692	367,662	727,044
ca-GrQc	5,242	28,980	48,260
ca-CondMat	23,133	186,936	173,361
ego-Twitter	81,306	2,420,766	13,082,506
soc-Slashdot0902	82,168	948,464	602,592
soc-Slashdot0811	77,360	905,468	551,724
soc-Epinions1	75,879	508,837	1,624,481
soc-Pokec	1,632,803	30,622,564	32,557,458
soc-LiveJournal1	4,847,571	68,993,773	285,730,264
com-Orkut	3,072,441	117,185,083	627,584,181

Some queries also require a subset of nodes to be used as part of the queries. We execute these queries with different random samples of nodes, with varying size. A random sample of nodes is created by selecting nodes with probably $1/s$, where s is referred to as selectivity in our results. For example, for selectivity 10 and 100 we select respectively approximately 10% and 1% of the nodes.

Queries. We execute experiments with the following queries. We include the Datalog formulation. Variants for other systems (e.g. SQL, SPARQL) are available online.

- $\{3,4\}$-clique: find subgraphs with $\{3,4\}$ nodes such that every two nodes are connected by an edge. The 3-clique query is also known as the triangle problem. Similar to other work, we treat graphs as undirected for this query.

  `edge(a,b), edge(b,c), edge(a,c), a<b<c.`
- 4-cycle: find cycles of length 4.

  `edge(a,b), edge(b,c), edge(c,d), edge(a,d), a<b<c<d`
- $\{3,4\}$-path: find paths of length $\{3,4\}$ for all combinations of nodes `a` and `b` from two random samples `v1` and `v2`.

  `v1(a), v2(d), edge(a, b), edge(b, c), edge(c, d).`
- $\{1,2\}$-tree: find complete binary trees with $2n$ leaf nodes s.t. each leaf node is drawn from a different random sample.

  `v1(b), v2(c), edge(a, b), edge(a, c).`
- 2-comb: find left-deep binary trees with 2 leaf nodes s.t. each leaf node is drawn from a different random sample.

  `v1(c), v2(d), edge(a, b), edge(a, c), edge(b, d).`
- $\{2,3\}$-lollipop: finds $\{2,3\}$-path subgraphs followed by $\{3,4\}$-cliques. The start nodes '`a`' are a random sample '`v1`'.

  `v1(a), edge(a, b), edge(b, c), edge(c, d), edge(d, e), edge(c, e).`

The queries can be divided in acyclic and cyclic queries. This distinction is important because Minesweeper is instance-optimal for the acyclic queries [8]. From our queries, n-clique and n-cycle are β-cyclic. All others are β-acyclic. We add predicates v_1 and v_2. As we vary the size of these predicates, we also change the amount of redundant work. Minesweeper is able to exploit this redundancy, as we show below. All queries are executed as a count, which returns the number of results to the client. We verified the result for all implementations.

Systems. We evaluate the performance of LogicBlox using Minesweeper and LFTJ by comparing the performance of a wide range of database systems and graph engines.

name	description
lb/lftj	LogicBlox 4.1.4 using LFTJ
lb/ms	LogicBlox 4.1.4 using Minesweeper
psql	PostgreSQL 9.3.4
monetdb	MonetDB 1.7 (Jan2014-SP3)
virtuoso	Virtuoso 7
neo4j	Neo4j 2.1.5
graphlab	GraphLab v2.2

We select such a broad range of systems because the performance of join algorithms is not primarily related to the storage architecture of a database (e.g. row vs column vs graph stores). Also, we want to evaluate whether general-purpose relational databases utilizing optimal join algorithms can replace specialized systems, like graph databases, and perhaps even graph engines.

Due to the complexity of implementing and tuning the queries across all these systems (e.g. tuning the query or selecting the right indices), we first select two queries that we execute across the full range of systems. After establishing that we can select representative systems without compromising the validity of our results, we run the remaining experiments across the two variants of LogicBlox, PostgreSQL, and MonetDB. The results will show that the graph databases have their performance dominated by our selected set. We evaluate GraphLab only for 3-clique and 4-clique queries. The 3-clique implementation is included in the GraphLab distribution and used as-is. We developed the 4-clique implementation with advice from the GraphLab community, but developing new algorithms on GraphLab can be a heavy undertaking, requiring writing C++ and full understanding of its imperative *gather-apply-scatter* programming model. Therefore, we cannot confidently extend coverage on GraphLab beyond these queries.

Hardware. For all systems, we use AWS EC2 m3.2xlarge instances. This instance type has an Intel Xeon E5-2670 v2 Ivy Bridge or Intel Xeon E5-2670 Sandy Bridge processor with 8 hyperthreads and 30GB of memory. Database files are placed on the 80GB SSD drive provided with the instance. We use Ubuntu 14.04 with PostgreSQL from Ubuntu's default repository and the other systems installed manually.

Protocol. We execute each experiment three times and average the last two executions. We impose a timeout of 30 minutes (1800 seconds) per execution. For queries that require random samples of nodes, we execute them with multiple selectivities. For small data sets we use selectivity 8 (12.5%) and 80 (1.25%). For the other data sets we use selectivities of 10 (10%), 100 (1%), and 1000 (0.1%). We ensure each system sees the same random datasets. Although across runs for the same system, we use different random draws.

3.2 Results

We validate that worst-case optimal algorithms like LFTJ outperform many systems on cyclic queries, while Minesweeper is fastest on acyclic queries.

3.2.1 Standard Benchmark Queries

Clique finding is a popular benchmark task that is hand optimized by many systems. Table 1 shows that both LFTJ and Minesweeper are faster than all systems except the graph engine GraphLab on 3-clique. On our C++ implementation of 4-clique, GraphLab runs out of memory for big data-sets. After the systems that implement the optimal join algorithms, Virtuoso is fastest. Relational systems that do conventional joins perform very poorly on 3-clique and 4-clique due to extremely large intermediate results of the self-join, whether materialized or not. The simultaneous search for cliques as performed by Minesweeper and LFTJ prevent this. This difference is particularly striking on 4-clique.

LFTJ and Minesweeper perform well on datasets that have few cliques. This is visible in the difference between Twitter vs Slashdot and Epinions data sets in which the performance is much closer to GraphLab.

Acyclic Queries: {3,4}-path. Table 2 shows the results for {3,4}-path and other acyclic queries. Minesweeper is faster here, outperforming LFTJ on virtually every data set for 3-path.

Minesweeper does very well for non-trivial acyclic queries such as {3,4}-path queries because it has a caching mechanism that enables it to prune branches using the CDS. Interestingly, PostgreSQL is now the next fastest system: it is even more efficient than the worst-case optimal join system for a few data sets on 3-path. The PostgreSQL query optimizer is smart enough to determine that it is best to start separately from the two node samples, and materialize the intermediate result of one of the edge subsets ($\texttt{v1}(a) \bowtie \texttt{edge}(a,b)$ or $\texttt{v2}(d) \bowtie \texttt{edge}(c,d)$). This strategy starts failing though on 4-path, due to two edge joins between these two results, as opposed to just one for 3-path. MonetDB starts from either of the random node samples, and immediately does a self-join between two edges, which is a slow execution plan.

LFTJ does relatively worse on 4-path, and times out on bigger datasets. LFTJ with variable ordering $[a,b,d,c]$ is fairly similar to a nested loop join where for every $\texttt{v1}(a) \bowtie \texttt{edge}(a,b)$ the join $\texttt{v2}(d) \bowtie \texttt{edge}(c,d)$ is computed, except that the last join includes a filter on $\texttt{edge}(b,c)$ for the current b. This is still workable for 3-path, but does not scale to 4-path for bigger data sets. The comparison with {3,4}-clique and 4-cycle is interesting here, because the join is very similar. These queries allow LFTJ to evaluate the self-joins from both directions, where one direction narrows down the search of the other. This is not applicable to {3,4}-path. This example shows that LFTJ does not eliminate the need for the query optimizer to make smart materialization decisions for some joins. If part of the 3-path join is manually materialized, then performance improves.

For 3-path, Minesweeper and LFTJ have an interesting difference in performance when executing with different sizes of random node samples. LFTJ is consistently the fastest of the two algorithms for very high selectivity, but Minesweeper is best with lower selectivity, where Minesweeper starts benefiting substantially from the caching mechanism. With lower selectivity, the amount of redundant work is increased due to repeatedly searching for sub-paths. To deal with this type of queries, we need to have a mechanism to not only be able to do simultaneous search, which both LFTJ and Minesweeper have and perform well on clique-type queries, but also to avoid any redundant work generated when computing the sub-graphs. The latter is easily integrated into Minesweeper and that integration is very natural. Also in Appendix B, we will show some experiments to illustrate the effect of this technique when changing the selectivity.

3.2.2 Other Query Patterns

We examine some other popular patterns against other systems that support a high-level language. We see that LFTJ is fastest on cyclic queries, while Minesweeper is the fastest on acyclic queries. We then consider queries that contain cyclic and acyclic components.

- **4-cycle** Table 1 shows the 4-cycle results. PostgreSQL and MonetDB perform are slower by orders of magnitude, similar to the results for {3,4}-clique. LFTJ is significantly faster than Minesweeper on this cyclic query.
- **{1,2}-tree** LFTJ is the fastest for the 1-tree query, but has trouble with the 10% Orkut experiment. Minesweeper handles all datasets without issues for 1-tree, and is faster than LFTJ, which times out on many experiments. PostgreSQL and MonetDB both timeout on almost all of the 2-tree experiments. With a few exceptions, PostgreSQL does perform well on the 1-tree experiments. Minesweeper benefits from instance optimality on this acyclic query.

We consider the i-lollipop query combines the i-path and $(i+1)$-clique query for $i \in \{2,3\}$, so predictably PostgreSQL and MonetDB do very poorly. LFTJ does better than Minesweeper, which suffers from the clique part of the query, but LFTJ on the other hand suffers from the path aspect and times out for most bigger data sets. To illustrate that a combination of Minesweeper and LFTJ

		wiki-Vote	p2p-Gnutella31	p2p-Gnutella04	loc-Brightkite	ego-Facebook	email-Enron	ca-GrQc	ca-CondMat	ego-Twitter	soc-Slashdot0902	soc-Slashdot0811	soc-Epinions1	soc-Pokec	soc-LiveJournal1	com-Orkut
3-clique	lb/lftj	0	0	0	0	0	0	0	0	5	1	1	1	75	165	742
	lb/ms	1	1	0	2	1	3	0	1	23	7	5	6	282	-	-
	psql	1446	6	2	-	575	-	10	348	-	-	-	-	-	-	-
	monetdb	-	3	3	945	947	-	22	98	-	-	-	-	-	-	-
	virtuoso	18	2	1	17	23	46	1	4	296	75	68	158	-	-	-
	neo4j	348	19	6	212	250	418	4	32	-	1441	1308	1745	-	-	-
	graphlab	0	0	0	0	0	0	0	0	0	0	0	0	3	7	27
4-clique	lb/lftj	3	0	0	11	9	4	0	1	427	4	4	13	644	-	-
	lb/ms	11	1	0	10	31	25	1	2	288	39	32	96	-	-	-
	psql	-	52	10	-	-	-	1021	-	-	-	-	-	-	-	-
	monetdb		17	15				1219	-	-	-	-	-	-	-	-
	virtuoso	447	2	0	364	1240	968	2	38	-	1427	1273	-	-	-	-
	neo4j	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	graphlab	0	0	0	0	1	0	0	0	6	1	1	1	-	-	-
4-cycle	lb/lftj	11	1	0	4	8	7	0	1	171	31	29	34	1416	-	-
	lb/ms	24	3	1	17	23	59	0	3	587	183	156	268	-	-	-
	psql	309	4	1	1394	539	-	47	112	-	-	-	-	-	-	-
	monetdb	502	1	1	657	347	-	19	60	-	-	-	-	-	-	-

Table 1: Duration of cyclic queries {3,4}-clique and 4-cycle in seconds. "-" denotes a timeout.

ideas might be ideal, we crafted a specialized algorithm that runs Minesweeper on the path part of the query and LFTJ on the clique part. Table 2 shows this hybrid algorithm outperforms both. This may be an interesting research direction.

Summary. LogicBlox using the LFTJ or Minesweeper algorithms consistently out-performs other systems that support high-level languages. Among those systems, LFTJ performs fastest on cyclic queries and is competitive on acyclic queries (1-tree) or queries with very high selectivity. Minesweeper works best for all other acyclic queries and performs particularly well for queries with low selectivity due to its caching.

4. CONCLUSION

Our results suggest that this new class of join algorithms allows a fully featured, SQL relational database to compete with (and often outperform) graph database engines for graph-pattern matching. One direction for future work is to extend this benchmark to recursive queries and more graph-style processing (e.g., BFS, shortest path, page rank). A second is to use this benchmark to refine this still nascent new join algorithms. In the full version of this paper, we propose and experiment with a novel hybrid algorithm between LFTJ and Minesweeper. We suspect that there are many optimizations possible for these new breed of algorithms.

5. REFERENCES

[1] C. Aberger, A. Nötzli, K. Olukotun, and C. Ré. EmptyHeaded: Boolean Algebra Based Graph Processing. *ArXiv e-prints*, Mar. 2015.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.

[4] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514–550, 1983.

[5] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[6] M. Grohe and D. Marx. Constraint solving via fractional edge covers. In *SODA*, pages 289–298. ACM Press, 2006.

[7] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[8] H. Q. Ngo, D. T. Nguyen, C. Re, and A. Rudra. Beyond worst-case analysis for joins with Minesweeper. In *PODS*, pages 234–245, 2014.

[9] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms: [extended abstract]. In *PODS*, pages 37–48, 2012.

[10] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: New developments in the theory of join algorithms. In *SIGMOD RECORD*, pages 5–16, 2013.

[11] M. Rudolf, M. Paradies, C. Bornhövd, and W. Lehner. The graph story of the sap hana database. In *BTW*, 2013.

[12] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, New York, NY, USA, 1979. ACM.

[13] T. L. Veldhuizen. Incremental maintenance for leapfrog triejoin. *CoRR*, abs/1303.5313, 2013.

[14] T. L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014.

[15] T. L. Veldhuizen. Transaction repair: Full serializability without locks. *CoRR*, abs/1403.5645, 2014.

[16] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.

		wiki-Vote		p2p-Gnutella31		p2p-Gnutella04		loc-Brightkite		ego-Facebook		email-Enron		ca-GrQc		ca-CondMat	
		80	8	80	8	80	8	80	8	80	8	80	8	80	8	80	8
3-path	lb/lftj	0	2	**0**	**0**	**0**	**0**	1	20	0	0	1	40	0	0	0	2
	lb/ms	0	**1**	0	0	0	0	**1**	**4**	0	**0**	**1**	**3**	0	**0**	1	**1**
	psql	**0**	12	0	0	0	0	2	203	**0**	3	3	556	**0**	0	**0**	7
	monetdb	128	131	1	1	0	0	993	1036	45	56	-	-	6	5	57	68
	virtuoso	1	16	0	0	0	0	18	319	0	4	37	719	0	1	1	10
	neo4j	4	71	1	2	0	1	82	633	4	19	163	1584	1	4	6	42
4-path	lb/lftj	4	193	**0**	**0**	0	**0**	44	1155	1	9	75	-	1	5	6	59
	lb/ms	**1**	**1**	0	1	0	0	**4**	**9**	**0**	**1**	**4**	**7**	**0**	**0**	**2**	**4**
	psql	3	1099	0	1	**0**	0	299	-	0	102	914	-	0	39	4	437
	monetdb	-	-	3	4	1	2	-	-	-	-	-	-	230	321	-	-
	virtuoso	30	1363	0	1	0	0	1664	-	5	189	-	-	4	29	37	577
	neo4j	161	-	1	7	0	3	-	-	105	437	-	-	23	109	201	1309
1-tree	lb/lftj	0	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	0	0	**0**	**0**
	lb/ms	0	**0**	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	psql	**0**	1	0	0	0	0	0	1	0	0	0	2	**0**	**0**	0	0
	monetdb	4	5	1	1	0	0	-	-	-	-	-	-	0	0	1	1
2-tree	lb/lftj	8	-	1	**1**	0	1	531	-	3	-	-	-	**0**	250	6	-
	lb/ms	**1**	**1**	1	2	0	**1**	**6**	**9**	**0**	**1**	**4**	**8**	0	**1**	**3**	**4**
	psql	-	-	**0**	15	**0**	6	-	-	1622	-	-	-	61	-	1228	-
	monetdb	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2-comb	lb/lftj	0	6	**0**	**0**	0	**0**	1	20	**0**	3	1	50	0	0	0	2
	lb/ms	0	**0**	0	1	0	0	**1**	**3**	0	**0**	**1**	**2**	0	**0**	1	**1**
	psql	**0**	51	0	0	**0**	0	2	206	0	29	3	553	**0**	0	**0**	6
	monetdb	388	478	3	3	1	1	-	-	-	-	-	-	5	5	53	62
2-lollipop	lb/lftj	7	189	0	**0**	0	0	144	-	9	14	468	-	2	4	6	36
	lb/ms	16	169	0	1	0	0	407	-	25	38	-	-	5	12	18	73
	lb/hybrid	**1**	**1**	**0**	0	**0**	**0**	**7**	**8**	**0**	**1**	**10**	**13**	**0**	**0**	**1**	**2**
	psql	286	-	0	3	0	1	-	-	209	724	-	-	20	146	356	-
	monetdb	92	-	0	1	0	0	-	-	41	203	-	-	10	50	93	947
3-lollipop	lb/lftj	-	-	**0**	1	0	1	-	-	-	-	-	-	-	-	-	-
	lb/ms	-	-	1	3	0	1	-	-	-	-	-	-	-	-	-	-
	lb/hybrid	**19**	**20**	0	**1**	**0**	**0**	**193**	**195**	**21**	**26**	**313**	**312**	**6**	**8**	**25**	**27**
	psql	-	-	4	35	3	25	-	-	-	-	-	-	-	-	-	-
	monetdb	-	-	1	18	1	11	-	-	-	-	-	-	-	-	-	-

		ego-Twitter			soc-Slashdot0902			soc-Slashdot0811			soc-Epinions1			soc-Pokec			soc-LiveJournal1			com-Orkut		
		1K	100	10	1K	100	10	1K	100	10	1K	100	10	1K	100	10	1K	100	10	1K	100	10
3-path	lb/lftj	**1**	13	144	1	8	98	1	8	110	0	5	27	9	**120**	1521	**61**	1035	-	**60**	825	-
	lb/ms	1	**5**	**18**	1	**4**	**10**	1	**4**	**10**	0	**1**	**4**	25	129	**408**	68	**259**	-	111	**451**	-
	psql	2	215	-	**0**	5	938	**0**	6	890	**0**	2	243	**8**	166	-	142	1011	-	-	-	-
	monetdb	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	virtuoso	7	59	1435	8	52	1433	6	65	1268	2	15	403	75	784	-	-	-	-	-	-	-
	neo4j	57	323	-	28	370	-	41	405	-	15	88	877	-	-	-	-	-	-	-	-	-
4-path	lb/lftj	103	1286	-	**3**	203	-	62	240	-	4	68	-	710	-	-	-	-	-	-	-	-
	lb/ms	**8**	**22**	**46**	7	**13**	**24**	**7**	**14**	**23**	2	**6**	**10**	206	**556**	-	**470**	-	-	**697**	-	-
	psql	-	-	-	9	1211	-	10	1637	-	**1**	470	-	**94**	-	-	-	-	-	1378	-	-
	monetdb	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	virtuoso	710	-	-	1058	-	-	657	-	-	46	1785	-	-	-	-	-	-	-	-	-	-
	neo4j	-	-	-	-	-	-	-	-	-	1097	-	-	-	-	-	-	-	-	-	-	-
1-tree	lb/lftj	**0**	**0**	**2**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	1	**3**	**30**	**1**	**7**	**82**	**2**	**32**	443
	lb/ms	1	2	2	1	1	1	1	1	1	0	1	1	28	32	46	55	64	97	79	100	**152**
	psql	0	1	44	0	0	4	0	0	1	0	0	2	**1**	17	160	25	36	513	2	106	-
	monetdb	88	78	95	-	-	-	-	-	-	12	11	10	-	-	-	-	-	-	-	-	-
2-tree	lb/lftj	**2**	-	-	**1**	-	-	**2**	-	-	**0**	560	-	587	-	-	-	-	-	-	-	-
	lb/ms	21	**32**	**45**	10	**15**	**23**	9	**15**	**22**	4	**6**	**10**	**561**	**704**	-	**977**	**1249**	-	**1315**	-	-
	psql	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	monetdb	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2-comb	lb/lftj	**1**	15	180	1	8	117	1	11	101	0	5	41	**11**	**140**	1780	**66**	1161	-	395	-	-
	lb/ms	2	**7**	**12**	1	**4**	**6**	1	**4**	**6**	1	**1**	**3**	64	156	**272**	128	**282**	**507**	312	**575**	-
	psql	2	205	-	**0**	5	1014	**0**	6	936	**0**	3	288	14	196	-	153	1111	-	**162**	-	-
	monetdb	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2-lollipop	lb/lftj	185	829	-	88	664	-	130	671	-	77	235	-	396	-	-	-	-	-	-	-	-
	lb/ms	517	-	-	230	1498	-	233	-	-	167	439	-	-	-	-	-	-	-	-	-	-
	lb/hybrid	**18**	**37**	**58**	**17**	**26**	**30**	**26**	**47**	**51**	**8**	**13**	**15**	**203**	**625**	**878**	**1080**	-	-	**1663**	-	-
	psql	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	monetdb	-	-	-	-	-	-	-	-	-	1208	-	-	-	-	-	-	-	-	-	-	-
3-lollipop	lb/lftj	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	lb/ms	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	lb/hybrid	**1680**	-	-	**477**	**485**	**483**	**642**	**650**	**1263**	**255**	**275**	**281**	-	-	-	-	-	-	-	-	-
	psql	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	monetdb	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 2: Duration (seconds) of acyclic queries with different selectivities (80 and 8 for small datasets, 1K, 100 and 10 for others). "-" denotes a timeout.

APPENDIX

A. AGM BOUND

Given a join query Q whose hypergraph is $\mathcal{H}(Q) = (\mathcal{V}, \mathcal{E})$, we index the relations using edges from this hypergraph. Hence, instead of writing $R(\text{vars}(R))$, we can write R_F, for $F \in \mathcal{E}$.

A *fractional edge cover* of a hypergraph $\mathcal{H}$ is a point $\mathbf{x} = (x_F)_{F \in \mathcal{E}}$ in the following polyhedron:

$$\left\{ \mathbf{x} \mid \sum_{F : v \in F} x_F \geqslant 1, \forall v \in \mathcal{V}, \mathbf{x} \geqslant \mathbf{0} \right\}.$$

Atserias-Grohe-Marx [3] and Grohe-Marx [6] proved the following remarkable inequality, which shall be referred to as the *AGM's inequality*. For any fractional edge cover $\mathbf{x}$ of the query's hypergraph,

$$|Q| = |\Join_{F \in \mathcal{E}} R_F| \leqslant \prod_{F \in \mathcal{E}} |R_F|^{x_F}. \tag{1}$$

Here, $|Q|$ is the number of tuples in the (output) relation Q.

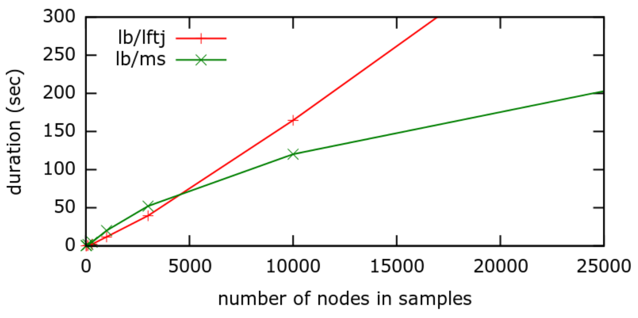
The optimal edge cover for the AGM bound depends on the relation sizes. To minimize the right hand side of (1), we can solve the following linear program:

$$\begin{aligned} \min \quad & \sum_{F \in \mathcal{E}} (\log_2 |R_F|) \cdot x_F \\ \text{s.t.} \quad & \sum_{F : v \in F} x_F \geqslant 1, v \in \mathcal{V} \\ & \mathbf{x} \geqslant \mathbf{0} \end{aligned}$$

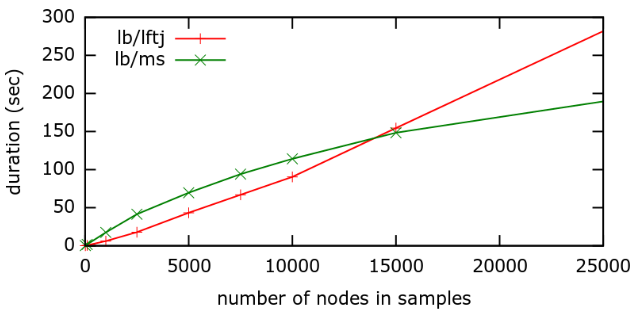
Implicitly, the objective function above depends on the database instance $\mathcal{D}$ on which the query is applied. We will use AGM(Q) to denote the best AGM-bound for the input instance associated with Q. AGM showed that the upper bound is essentially tight in the sense that there is a family of database instances for which the output size is asymptotically the same as the upper bound. Hence, any algorithm whose runtime matches the AGM bound is optimal in the worst-case.

B. EXTENDED EXPERIMENTS

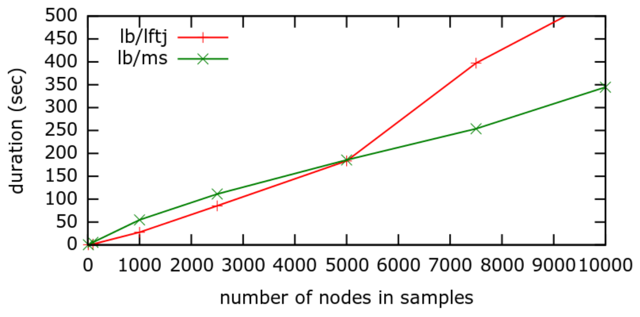
To illustrate that Minesweeper is better with lower selectivity by exploiting the caching idea, Figure 1 compare the performance of the algorithms when 3-path is executed with increasingly larger node samples.



(a) 3-path on LiveJournal with samples of N nodes



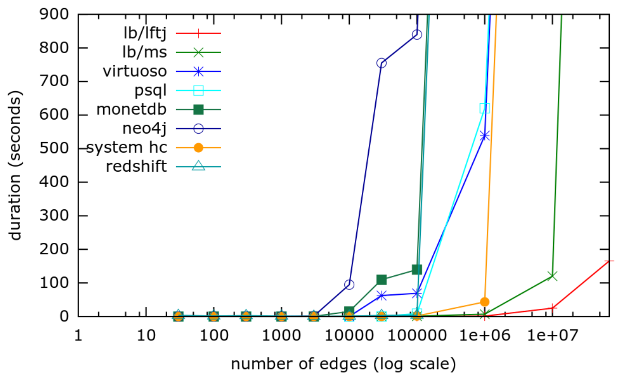
(b) 3-path on Pokec with samples of N nodes



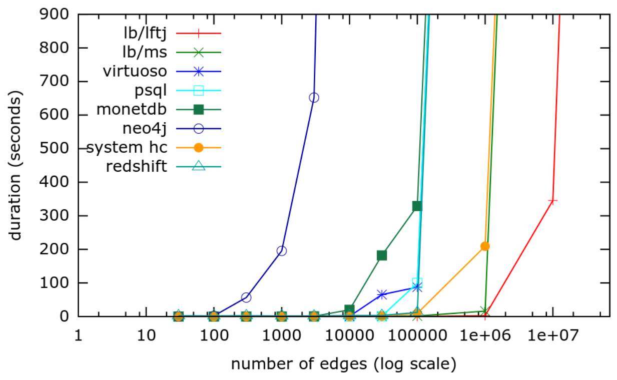
(c) 3-path on Orkut with samples of N nodes

Figure 1: 3-path on datasets with samples of N nodes

Scaling Behavior. To understand the scaling issues better, we eliminate the variability of the different datasets and execute a separate experiment where we gradually increase the number of edges selected from the LiveJournal dataset with a timeout of 15 minutes. This time we also include RedShift and System HC. The results are shown in Figures 2. This analysis shows that conventional relational databases (and Neo4J) do not handle this type of graph query even for very small data sets. Virtuoso is the best after the optimal joins. Optimal joins can handle subsets of two orders of magnitude bigger, and LFTJ supports an order of magnitude bigger graphs than Minesweeper.



(a) Duration of 3-clique on LiveJournal subset of N edges



(b) Duration of 4-clique on LiveJournal subset of N edges

Figure 2: Duration of clique query on LiveJournal subset of N edges

C. LOGICBLOX DATABASE SYSTEM

The LogicBlox database is a commercial database system that from the ground up is designed to serve as a general-purpose database system for enterprise applications. The LogicBlox database is currently primarily used by partners of LogicBlox to develop applications that have a complex workload that cannot easily be categorized as either analytical, transactional, graph-oriented, or document-oriented. Frequently, the applications also have a self-service aspect, where an end-user with some modeling expertise can modify or extend the schema dynamically to perform analyses that were originally not included in the application.

The goal of developing a general-purpose database system is a deviation from most current database system development, where the emphasis is on designing specialized systems that vastly outperform conventional database systems, or to extend one particular specialization (e.g. analytical) with reasonable support for a different specialized purpose (e.g. transactional).

The challenging goal of implementing a competitive general-purpose database, requires different approaches in several components of a database system. Join algorithms are a particularly important part, because applications that use LogicBlox have schemas that resemble graph as well as OLAP-style schemas, and at the same time have a challenging transactional load. To the best of our knowledge, no existing database system with conventional join algorithms can efficiently evaluate queries over such schemas, and that is why LogicBlox is using a join algorithm with strong optimality guarantees: LFTJ.

Concretely, the motivation for implementing new optimal join algorithms are:

- No previously existing join algorithm efficiently supports the graph queries required in applications. On the other hand, graph-oriented systems cannot handle OLAP aspects of applications.
- To make online schema changes easy and efficient, LogicBlox applications use unusually high normalization levels, typically 6NF. The normalized schemas prevent the need to do surgery on existing data when changing the schema, and also helps with efficiency of analytical workloads (compare to column stores). A drawback of this approach is that queries involve a much larger number of tables. Selection conditions in queries typically apply to multiple tables, and simultaneously considering all the conditions that narrow down the result becomes important.
- As opposed to the approach of highly tuned in-memory databases that fully evaluate all queries on-the-fly [11], LogicBlox encourages the use of materialized views that are incrementally maintained [13]. The incremental maintenance of the views under all update scenarios is a challenging task for conventional joins, in particular combined with a transactional load highly efficient maintenance is required [15].

The LogicBlox database system is designed to be highly modular as a software engineering discipline, but also to encourage experimentation. Various components can easily be replaced with different implementations. This enabled the implementation of Minesweeper, which we compare to the LFTJ implementation and other systems in this paper.