Table 5. Result of full match enumeration for some of the search queries in Fig. 10 in the Twitter and UK-Web graphs (Table 3). For each template, the table lists number of vertices and edges in the final solution graph $\mathcal{G}^*$, match count, and time-to-solution. In the UK-Web graph, the Q8 is the most abundant, 45 billion matches: however, the solution graph is the smallest among the three queries. The numbers highlights the challenges associated with listing matches that are highly concentrated.

	Twitter			UK Web		
$\lvert\mathcal{L}\rvert$ in $\mathcal{G}$	50	100	150	25	50	100
Template	Q4	Q6	Q8	Q4	Q6	Q8
$\lvert\mathcal{V}^*\rvert$	944K	91K	25K	8M	1.4M	230K
$2\lvert\mathcal{E}^*\rvert$	6M	950K	338K	67M	13M	2.5M
Count	10B	78M	615M	3.8B	2.1B	45B
Time	1.2hr	5hr	1hr	12.6min	49.4min	8.1hr

5.5 Pattern Matching in Graphs with Diverse Topology

We extend the set of graphs we analyze to a more diverse set of topologies (node-degree distributions, density, diameter). We use there real-world graphs: Twitter, UK-Web and Road USA (the graph properties are listed in Table 3); and three R-MAT graphs (generated to have the same size yet different vertex degree distribution). The vertex degree distribution plot for these graphs are available in Appendix A. Unless otherwise specified, we run each experiment on 64 compute nodes.

Large Real-World Power-Law Graphs. Twitter and UK Web are billion-edge, real-world graphs that have previously used for a wide range of graph analysis problems; yet, rarely in the context of exact pattern matching. Although both are power-law, they have significantly diffrent topologies: Twitter has a more skewed degree distribution, but the larger UK-Web graph is denser - it has a higher average vertex degree.

Since Twitter and UK-Web graphs are unlabeled[2], we use a labeling technique often used in this area [Serafini et al. 2017]: we randomly assign vertex labels: for the Twitter graph up to 150 unique labels uniformly distributed among ~41M vertices. For the relatively less skewed UK Web graph, we use up to 100 labels. For our experiments, we consider some of the patterns in Fig. 10 (previously used by Serafini et al. [Serafini et al. 2017]). Table 5 lists, for each search template, the full match enumeration time (includes time spent pruning), match count, and number of vertices and edges in the solution graphs. The results suggest the abundance of acyclic substructures are higher in the Twitter graph (10B matches for Q4, compared to 3.8B in the UK Web graph). The denser UK Web graph has a higher concentration of the cyclic patterns, Q6 and Q8. Q8 is the most abundant - over 45B matches in the background graph; however, the long search duration suggests matches are potentially concentrated with in a limited number of graph partitions which limits task parallelism. Similar reasoning applies to long search duration of Q6 in the Twitter graph. (We discussed limitations stemming from load imbalance due to such artifacts in §5.3.)

Large Diameter Real-World Network. The Road-USA [Rossi and Ahmed 2015] is a real-world road network graph which has a very large diameter (at least 6,000, largest of all the graph datasets used in this work; however, a low average vertex degree, lower than three). The graph is not labelled. Table 6 lists time for counting matches for four unlabeled patterns. The results show, on the one side, abundance of small acyclic patterns compared to cyclic structures in the road network graph,

[2]The UK-Web, which is a webgraph, has a vertex label set similar to the original WDC graph, i.e., webpage URLs. We did not use it to ensure a fair comparison with the unlabeled Twitter graph.

Table 6. Runtime for match counting in the large diameter Road-USA graph (unlabeled). The reported times include time spent in pruning as well as counting. Template UQ4 has the same topology as Q4 in Fig. 10, however, is unlabeled. The 5-Star pattern is an acyclic graph - a central vertex with four one hop neighbors (mimicking a four way stop or an intersection). Given the background graph is relatively small, we run these experiments on eight compute nodes.

Template	UQ4	5-Star	3-Clique	4-Clique
Count	220M	66M	439K	90
Time	26.7s	17.3s	5.0s	1.6s

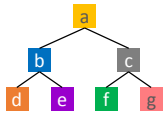
Table 7. Searches in the three R-MAT graphs (in Fig. 14 (a) – (c)) with varying degree distribution using the Q4 pattern in Fig. 10 and the RMAT-2 pattern in Fig. 9. For each query in each graph, the table lists the number of vertices and edge in the pruned solution graph, number of matches and time-to-solution (which includes time spent in pruning and match enumeration in the pruned graph).

Template		Graph500	Chakrabarti et al.	Uniform
Q4	$\|\mathcal{V}^*\|$	4.4K	641M	7.5M
	$2\|\mathcal{E}^*\|$	7K	3.5B	16.4M
	Count	946	4B	4.2M
	Time	4.2s	2.9min	54.1s
RMAT-2	$\|\mathcal{V}^*\|$	2.3K	303M	0
	$2\|\mathcal{E}^*\|$	4.1K	920M	0
	Count	551	1.4B	0
	Time	8.7s	83.4s	52.5s

and on the other side, the ability of our framework to support searches on large, unlabeled graphs with a completely different topology.

Synthetic Graphs with Varying Node-degree Distribution. We further study the influence of graph topology on search performance and match properties using R-MAT generated synthetic graphs [Chakrabarti et al. 2004]. The R-MAT model recursively sub-divides the graph (initially empty) into four equal-sized partitions and distributes edges within these partitions with unequal probabilities. Each edge chooses one of the four partitions with probabilities A, B, C, and D, respectively. These probabilities, determine the skewness of the generated graph. For our experiments, we create three R-MAT graphs with the following sets of probabilities: *(i)* The configuration used by the Graph 500 benchmark, (0.57, 0.19, 0.19, 0.05); *(ii)* Parameters suggested by Chakrabarti et al. in [Chakrabarti et al. 2004] to simulate real-world scale-free graphs, (0.45, 0.15, 0.15, 0.25); and *(iii)* Equal probability for for all four partitions to create graphs with uniform degree distribution (also known as the Erdős-Rényi graph), (0.25, 0.25, 0.25, 0.25). Fig. 14 (a) – (c) in the Appendix show the degree distribution of these R-MAT graphs. For all the graphs we use the same Scale (30) and (directed) edge factor (16). We follow the same approach as used for weak scaling experiments in [redacted] to generate vertex labels: we use vertex degree information to create vertex labels, computed using the formula, $\ell(v_i) = \lceil \log_2(d(v_i) + 1) \rceil$.

Table 7 shows the results for full match enumeration in the three R-MAT graphs for the following two queries: Q4 in Fig. 10 and RMAT-2 in Fig. 9. For both search templates, most matches are found for the configuration labeled Chakrabarti et al. This is because the graph in Fig. 14(b) has a higher frequency of high degree vertices compared to the graphs in Fig. 14(a) and (c); since we use degree based vertex labels, this has direct impact on match density. Although, for the smaller Q4 pattern, the R-MAT graph with uniform degree distribution has more matches than the Graph 500 configuration; in the uniform graph, no matches were found for the larger seven vertex RMAT-2



RMAT-2

Fig. 9. The template labels represent the the most frequent vertex labels in the respective background graphs (similar to the patterns in Fig. 10.

pattern with unique labels: low variance in degree distribution means 99.9% of the graph vertices have one of the top four most frequent labels.

5.6 Comparison with State-of-the-Art Systems

We empirically compare our work with three state-of-the-art pattern matching systems QFrag [Serafini et al. 2017], TriAD [Gurajada et al. 2014], and Arabesque [Teixeira et al. 2015]. QFrag is a generic pattern matching system; we use it for comparison using labeled patterns. Arabesque, although requires effort for writing pattern search algorithms, has demonstrated the ability to scale to much larger graphs; we use this system for comparison using unlabeled patterns. Since both QFrag and Arabesque are based on Apache Spark [Spark 2017] and inherit its limitations; we also compare with a MPI-based solution - TriAD. Similar to our system, all these three systems offer exact matching with 100% precison and recall. For all experiments, we report time for a single query. We do not report time spent in graph loading and partitioning, and preprocessing (such as index creation in TriAD) as they are done once for each graph dataset. We run these experiments on real-world graph datasets using a shared memory platform with 60 CPU-cores and 1.5TB physical memory.

5.6.1 Comparison with QFrag. Similar to our solution, QFrag targets exact pattern matching on distributed platforms, yet there are two main differences: QFrag assumes that the entire graph fits in the memory of each compute node and uses data replication to enable parallelism. More importantly, QFrag employs a sophisticated load balancing strategy to achieve scalability. QFrag is implemented on top of Apache Spark and Giraph [Giraph 2016]. In QFrag, each replica runs an instance of a pattern enumeration algorithm called $\text{Turbo}_{\text{ISO}}$ [Han et al. 2013] (essentially an improvement of Ullmann's algorithm [Ullmann 1976]). Through evaluation, the authors demonstrated QFrag's performance advantages over two other distributed pattern matching systems: *(i)* TriAD [Gurajada et al. 2014] (which we confirm), and *(ii)* GraphFrames [Dave et al. 2016; GraphFrames 2017], a graph processing library for Apache Spark, also based on distributed join operations.

Given that we have demonstrated the scalability of our solution (Serafini et al. demonstrate equally good scalability properties for QFrag [Serafini et al. 2017], yet on much smaller graphs), we are interested to establish a comparison baseline at the single node scale. To this end, we run experiments on a modern shared memory machine with 60 CPU-cores, and use the two real-world graphs (*Patent* and *YouTube*) and four query patterns (Fig. 10) that were used for evaluation of QFrag [Serafini et al. 2017]. We run QFrag with 60 threads and HavoqGT with 60 MPI processes. The results are summarized in Table 8: QFrag runtimes for match enumeration (first pair of columns) are comparable with the results presented in [Serafini et al. 2017], so we have reasonable confidence that we replicate their experiments well. With respect to combined pruning and enumeration

[2]Although TriAD is a RDF query processing engine and follows a distributed join based design, it has been shown to perform well for scale-free graphs [Serafini et al. 2017]. Furthermore, TriAD is the only well performing MPI-based exact matching solution that is publicly available for evaluation.

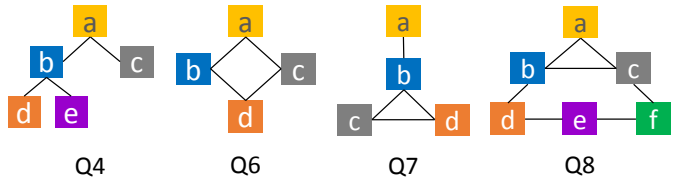


Fig. 10. The patterns (reproduced from [Serafini et al. 2017]) used for comparison with QFrag and TriAD (results in Table 8). The label of each vertex is mapped, in alphabetical order, to the most frequent label of the graph in decreasing order of frequency. Here, *a* represents the most frequent label, *b* is the second most frequent label, and so on.

Table 8. Performance comparison between QFrag, TriAD and our pattern matching system: The table shows the runtime in seconds for full enumeration for QFrag and TriAD; and separately for pruning and full match enumeration for our distributed system (labeled [redacted]-distributed), and for a single node implementation of our graph pruning-based approach tailored for a shared memory system (labeled [redacted]-shared). For [redacted] we split time-to-solution into pruning time (top row) and enumeration time (bottom row). We use the same graphs (*Patent* and *YouTube*) and the query templates as in Fig. 10 (Q4 – Q8) used for evaluation of QFrag and TriAD in [Serafini et al. 2017]. (The other queries used in [Serafini et al. 2017] are simpler and less interesting.) The best distributed runtime for a query, for each graph, is shown in bold font.

	QFrag		TriAD		[redacted]-distributed		[redacted]-shared	
	Patent	YouTube	Patent	YouTube	Patent	YouTube	Patent	YouTube
Q4	4.19	8.08	12.24	43.93	**0.238**	**0.704**	0.100	0.400
					0.223	**1.143**	0.010	0.010
Q6	5.99	10.26	0.89	16.49	**0.874**	**2.340**	0.070	1.730
					0.065	**0.301**	0.005	0.010
Q7	6.36	11.89	1.08	11.16	**0.596**	**1.613**	0.130	0.820
					0.039	**0.180**	0.005	0.010
Q8	10.05	14.48	**0.93**	29.09	0.959	**2.633**	0.100	1.370
					0.049	**0.738**	0.001	0.010

time, our system (second pair of columns, presenting pruning and enumeration time separately) is consistently faster than QFrag on all the graphs, for all the queries. We note that our solution does not take advantage of shared memory of the machine at the algorithmic or implementation level (we use different processes, one MPI process per core), and has the system overhead of MPI-communication between processes. (Additionally, unlike QFrag, our system is not handicapped by the memory limit of a single machine as it supports graph partitioning and can process graphs larger than the memory of a single node.)

To highlight the effectiveness of our technique and get some intuition on the magnitude of the MPI overheads in this context, we implemented our technique for shared memory and present runtimes (when using 60 threads) for the same set of experiments in Table 8 (the two rightmost columns). We notice up to an order of magnitude improvement in performance compared to the distributed implementation running on a single node.

In summary, our distributed solution works about 4–10× faster than QFrag, and, if excluding distributed system overheads and considering the pruning time for the shared memory solution and conservatively reusing enumeration runtime for the distributed solution, it is about 6–100× faster than QFrag.

5.6.2 Comparison with TriAD. TriAD [Gurajada et al. 2014] is distributed RDF [RDF 2017] engine, implemented in MPI, and based on an asynchronous distributed join algorithm which uses *partitioned locality* based indexing. The Resource Description Framework (RDF), is a metadata/typed graph model [RDF 2017], where information is stored as a linked *Subject-Predicate-Object* triple. The *Subject*, *Predicate* and *Object* are essentially designated types for graph vertices (forming a triple) and the links between them are edges in the graph. A SPARQL query disassembles a search template into a set of edges and the final results are constructed through *multi-way join* operations [Gurajada et al. 2014].

TriAD's design follows the classical master-slave architecture at indexing time, but allows for a direct, asynchronous communication among the slaves at query processing time. TriAD's index structure is optimized for processing hash joins. TriAD employs *hash-based sharding* for data partitioning and partitioning information in encoded into the triples; which enables locality awareness and allows potentially large number of concurrent join operations by multiple worker nodes without the need for remote communication. Furthermore, in TriAD, the master node maintains a global index statistics (collected at local index creation time on the slave nodes). This information is used by the query plan generator: query optimization is informed by a unified cost model for optimizing relational join operations.

We run the same experimnets for TriAD as we did earlier for comparison with QFrag on the same data, queries, and large shared-memory platform. The experiment results are summarzed in Table 8, next to the QFrag results. For the smaller, less skewed and dense Patent graph, except for search template Q4, TriAD's performance is on par with the distributed implementation of [redacted] (and better than QFrag). In all othre situations and, particularly for the more skewed YouTube graph, TriAD performs much worse. For Q4 and Q8, TriAD is ~5× and ~2× slower than QFrag, and ~20× and ~9× slowe than distributed [redacted].

Although at the implementation level TriAD shares some similarities with [redacted] (e.g., it leverages asynchronous MPI communication); in contrast to QFrag and [redacted], TriAD follows a different design philosophy - *distributed hash join operations*. Whereas the solution approach QFrag uses is can be categorized as *graph exploration* [Abdelaziz et al. 2017; Gurajada et al. 2014], an our solution ads graph pruning based on constraint checking to this. As expected high-level design decisions are key drivers for performance: although QFrag operates within a managed runtime environment (i.e., JRE - the Java Runtime Environment), slower than the native MPI/C++ runtime, and relies on TCP for remote communication, which again, is slower than MPI communication primitives (typically optimized to harness shared memory IPC); QFrag's design enables sophisticated load balancing which is crucial for achieving good performance in presence of often highly skewed real-world graphs. Our design, in addition to harnessing asynchronous communication and embracing horizontal scalability, offers aggressive search space pruning while maintaining small algorithm states to prevent combinatorial explosion; thus, able to scale to large graphs as well as has been demonstrated to be performant for relatively small datasets. TriAD, although implemented in MPI, the join-based design suffers in presence of larger graphs and patterns with larger diameter.

In a recent study, Abdelaziz et al. [Abdelaziz et al. 2017] pointed out a key scalability limitation of TriAD: following distributed join operations, to enable parallel processing, TriAD needs to re-shard intermediate results if the sharding column of the previous join is not the current join column. This cost can be significant for large intermediate results with multiple attributes. Also, their analysis in [Abdelaziz et al. 2017] shows the significant memory overhead of indexing in TriAD (often larger than the actual graph topology). Also, we noticed that the overhead of index creation increases with the graph size: index creation time for the Patent graph is about 2.5 minutes which goes up to about 7.7 minutes for the larger Youtube graph.

Table 9. Performance comparison between Arabesque and our pattern matching system (labeled [redacted] short for [redacted]. The table shows the runtime in seconds for counting 3-Clique and 4-Clique patterns. These search patterns as well as the following background graphs were used for evaluation of Arabesque in [Teixeira et al. 2015]. We run experiments on the same shared memory machine (with 1.5TB physical memory) we used for comparison with QFrag. Additionally, for [redacted] present runtimes on 20 compute nodes. Here, [redacted] runtimes for the single node, shared memory are under the column labeled [redacted](1) while runtimes for the 20 node, distributed deployment are under the column labeled [redacted](20).

	3-Clique			4-Clique		
	Arabesque	[redacted](1)	[redacted](20)	Arabesque	[redacted](1)	[redacted](20)
CiteSeer	3.2s	0.04s	0.02s	3.6s	0.06s	0.02s
Mico	13.6s	27s	11s	1min	72min	21min
Patent	80s	17.3s	1.6s	2.2min	32.8s	8.3s
Youtube	330s	126s	12.7s	Crash	6.4min	1.4min
LiveJournal	474s	144s	11.2s	2.5hr+	1.8hr	0.6hr

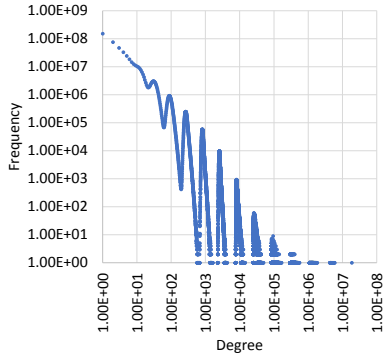
5.6.3 Comparison with Arabesque. Arabesque is a framework offering precision and recall guarantees implemented on top of Apache Spark and Giraph [Giraph 2016]. Arabesque provides an API based on the Think Like an Embedding (TLE) paradigm, which enables a user to to express graph mining algorithms tailored for each specific search pattern and a BSP implementation of the search engine. Arabesque replicates the input graph on all worker nodes, hence, the largest graph scale it can support is limited by the size of the memory of a single node. As Teixeira et al. [Teixeira et al. 2015] showed Arabesque's superiority over other systems: G-Tries [Ribeiro and Silva 2014] and GRAMI [Elseidy et al. 2014], we indirectly compare with these two systems as well.

For the comparison, we use the problem of *counting cliques* in an unlabeled graph (the implementation is available with the Arabesque release). This is a usecase that is favourable to Arabesque as our system is not specifically optimized for match counting. The following table compares results of counting three- and four-vertex cliques, using Arabesque and our system (labeled [redacted]), using the same real-world graphs used for the evaluation of Arabesque in [Teixeira et al. 2015]. In Experiments use the same large shared-memory machine. Additionally, for [redacted], we present runtimes on 20 compute nodes. (We attempted Arabesque experiments on 20 nodes too, however, Arabesque would crash with the out of memory (OOM) error for the larger Patent, Youtube and LiveJournal graphs. Each compute node in our distributed testbed has 128GB memory. Our multi-core shared memory testbed, however, has 1.5TB physical memory. Furthermore, for Arabesque, for the workloads that successfully completed on the 20 node deployment, we did not notice any speedup over the single node run.) Note that Arabesque users have to code a purpose-built algorithm for counting cliques, whereas ours and QFrag are generic pattern matching solutions, not optimized to count cliques only. Furthermore, in addition to replicating the data graph, Arabesque also exploits HDFS storage for maintaining the algorithm state (i.e., intermediate matches).

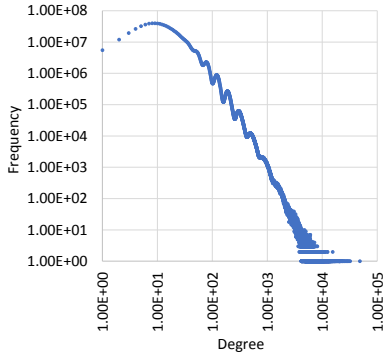
[redacted] was able to count all the clique patterns in all graphs; it took a maximum time of 1.8 hours to count 4-Cliques in the LiveJournal graph on the singel node, shared memory system. When using 20 nodes, for the same workload, the runtime came down to 41.3 minutes. Arabesque's performance degrades for larger graphs and search templates: Arabesque performs reasonably well for the 3-Clique pattern, for the larger graphs - [redacted] is at most 3.7× faster. The 4-Clique pattern, highlights the advantage of our system: for the Patent graph, [redacted] is 4× faster on the shared memeory platform. For the LiveJournal graph, Arabesque did not finish in 2.5 hours (we terminate processing). For the Youtube graph, Arabasque would crash after runing for 45 minutes.

[redacted] on the other hand, completed clique counting for both graphs. For the smaller, yet highly skewed Mico graph Arabesque outperforms [redacted]: for the 4-Clique pattern, Arabesque completes clique counting in about one minute, where as it takes [redacted] 72 minutes on the same platform; this workload highlights the advantage of replicating the data graph for parallel processing which presents the opportunity for harnessing load balancing techniques that are efficient and effective.

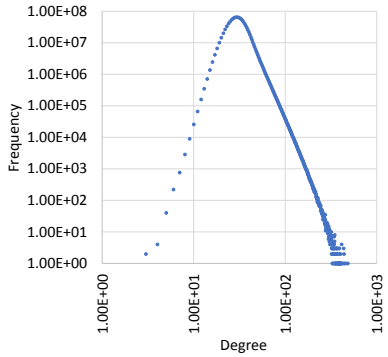
A GRAPH PROPERTIES



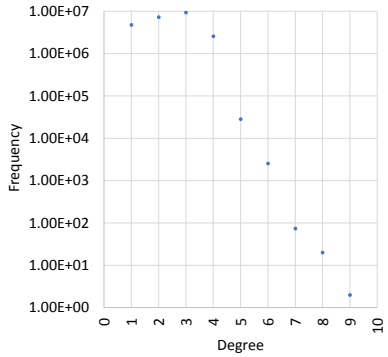
(a) Graph 500 - d_{max} 18.7M; d_{stdev} 1.6K; (0.57, 0.19, 0.19, 0.05)



(b) Chakrabarti et al. - d_{max} 48.3K; d_{stdev} 59.3; (0.45, 0.15, 0.15, 0.25)



(c) Uniform - d_{max} 482; d_{stdev} 8.6; (0.25, 0.25, 0.25, 0.25)



(d) Road USA - d_{max} 9; d_{avg} 2.4; d_{stdev} 0.9

Fig. 14. Vertex degree distribution of three R-MAX generated graphs with varying skewness labeled (a) Graph 500, (b) Chakrabarti et al. and (c) Uniform. For each graph, the figure shows maximum vertex degree and standard deviation of vertex degree in the background graph, and the R-MAT edge probabilities (A, B, C, D) used. For (a) – (c), X-xis and Y-axis are in log scale. Here the graph Scale (30) and directed edge factor (16) are fixed for all three graphs. The storage requirement for each graph is ~270GB. The last figure (d) shows vertex degree distribution of the large diameter Road USA graph. In this chart, only the Y-axis is in log scale.