Scalable Pattern Matching in Metadata Graphs via Constraint Checking

TAHSIN REZA, HASSAN HALAWA, and MATEI RIPEANU, University of British Columbia
GEOFFREY SANDERS and ROGER A. PEARCE, Lawrence Livermore National Laboratory

Pattern matching is a fundamental tool for answering complex graph queries. Unfortunately, existing solutions have limited capabilities: They do not scale to process large graphs and/or support only a restricted set of search templates or usage scenarios. Moreover, the algorithms at the core of the existing techniques are not suitable for today's graph processing infrastructures relying on horizontal scalability and shared-nothing clusters, as most of these algorithms are inherently sequential and difficult to parallelize.

We present an algorithmic pipeline that bases pattern matching on *constraint checking*. The key intuition is that each vertex and edge participating in a match has to meet a set of constraints implicitly specified by the search template. These constraints can be verified independently and typically are less expensive to compute than searching the full template. The pipeline we propose generates these constraints and iterates over them to eliminate all the vertices and edges that *do not* participate in any match, thus reducing the background graph to a subgraph that is the union of all template matches—the *complete set* of all vertices and edges that participate in at least one match. Additional analysis can be performed on this annotated, reduced graph, such as full match enumeration, match counting, or computing vertex/edge centrality. Furthermore, a *vertex-centric* formulation for constraint checking algorithms exists, and this makes it possible to harness existing high-performance, vertex-centric graph processing frameworks.

This technique (i) enables highly scalable pattern matching in metadata (labeled) graphs; (ii) supports arbitrary patterns with 100% precision; (iii) enables tradeoffs between precision and time-to-solution, while always selects all vertices and edges that participate in matches, thus offering 100% recall; and (iv) supports a set of popular data analytics scenarios. We implement our approach on top of HavoqGT, an open-source asynchronous graph processing framework, and demonstrate its advantages through strong and weak scaling experiments on massive scale real-world (up to 257 billion edges) and synthetic (up to 4.4 trillion edges) labeled graphs, respectively, and at scales (1,024 nodes / 36,864 cores), orders of magnitude larger than used in the past for similar problems.

This article serves two purposes: First, it synthesises the knowledge accumulated during a long-term project [Reza et al. 2017, 2018; Tripoul et al. 2018]. Second, it presents new system features, usage scenarios, optimizations, and comparisons with related work that strengthen the confidence that pattern matching based on iterative pruning via constraint checking is an effective and scalable approach in practice. The new contributions include the following: (i) We demonstrate the ability of the constraint checking approach

This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 (LLNL-JRNL-817625). Experiments were performed at the Livermore Computing facility.

Authors' addresses: T. Reza, H. Halawa, and M. Ripeanu, University of British Columbia, Electrical and Computer Engineering, 2332 Main Mall, Vancouver, BC Canada V6T 1Z4; emails: {treza, hhalawa, matei}@ece.ubc.ca; G. Sanders and R. A. Pearce, Lawrence Livermore National Laboratory, L-495, 7000 East Avenue, Livermore, CA USA 94550; emails: {sanders29, rpearce}@llnl.gov.

2329-4949/2020/01-ART2 $15.00
https://doi.org/10.1145/3434391

to efficiently support two additional search scenarios that often emerge in practice, *interactive incremental search* and *exploratory search*. (ii) We empirically compare our solution with two additional state-of-the-art systems, Arabsque [Teixeira et al. 2015] and TriAD [Gurajada et al. 2014]. (iii) We show the ability of our solution to accommodate a more diverse range of datasets with varying properties, e.g., scale, skewness, label distribution, and match frequency. (iv) We introduce or extend a number of system features (e.g., work aggregation, load balancing, and the ability to cap the generated traffic) and design optimizations and demonstrate their advantages with respect to improving performance and scalability. (v) We present bottleneck analysis and insights into artifacts that influence performance. (vi) We present a theoretical complexity argument that motivates the performance gains we observe.

CCS Concepts: • **Computer systems organization** → **Distributed architectures**; • **Information systems** → **Data mining**; • **Mathematics of computing** → **Graph algorithms**;

Additional Key Words and Phrases: Pattern matching, subgraph isomorphism, graph processing, distributed computing

ACM Reference format:
Tahsin Reza, Hassan Halawa, Matei Ripeanu, Geoffrey Sanders, and Roger A. Pearce. 2020. Scalable Pattern Matching in Metadata Graphs via Constraint Checking. *ACM Trans. Parallel Comput.* 8, 1, Article 2 (January 2021), 45 pages.
https://doi.org/10.1145/3434391

1 INTRODUCTION

Pattern matching in labeled graphs, that is, finding subgraphs that *match* a small *template graph* within a large *background graph*, is fundamental to graph analysis and has applications in social network analysis [Fan et al. 2010, 2013; Gupta et al. 2014; Tong et al. 2007], bioinformatics [Alon et al. 2008], anomaly and fraud detection [Iyer et al. 2018], program analysis [Lo et al. 2009], as well as in various machine learning contexts [Grover and Leskovec 2016; Henderson et al. 2012]. A *match* can be broadly categorized as either *exact* (i.e., there is a *bijective* mapping between the vertices/edges in the template and those in the matching subgraph) or *approximate* (the template and the match are just similar by some defined similarity metric) [Bunke and Allermann 1983; Conte et al. 2004; Zhang et al. 2010].

Unfortunately, existing pattern matching solutions (related work in Section 2) have limited capabilities: (i) They do not scale to the massive graphs with hundreds of billions of edges commonly mined nowadays, (ii) they often support only a restricted set of search templates or usage scenarios, and (iii) they rely on algorithms that are not suitable for implementation on top of today's graph processing infrastructures that aim at horizontal scalability and shared-nothing clusters, as most of these algorithms are inherently sequential and difficult to parallelize [Mckay and Piperno 2014; P. Cordella et al. 2004; Ullmann 1976].

We propose a new algorithmic pipeline based on *constraint checking*. This approach is motivated by viewing the search template as specifying a set of constraints the vertices and edges that participate in a match must meet. The pipeline iterates over these constraints to eliminate *all* and *only* the vertices and edges that *do not* participate in any match. The intuition for the effectiveness of this technique stems from four key observations:

(i) First, the traditionally used *graph exploration* techniques [Han et al. 2014; P. Cordella et al. 2004; Shang et al. 2008; Ullmann 1976] generally attempt to *enumerate* all matches through explicit search. When an exploration branch fails, it has to be marked invalid and ignored in the subsequent steps. In the same vein as past works that use graph pruning [Lulli et al. 2017; Zhou et al. 2012] or, more generally, input reduction [Kusum et al. 2016], we observe that it is much cheaper to focus on eliminating the vertices and edges that do not meet the label and topological constraints introduced by the search template. *One key contribution of this work is a pruning-based solution*

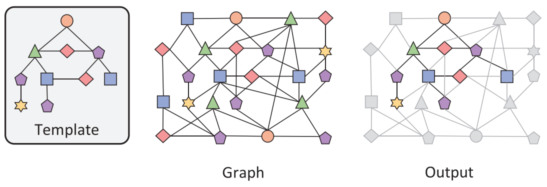


Fig. 1. An example of a background graph $\mathcal{G}$ (center), template $\mathcal{G}_0$ (left), and the output—the solution subgraph $\mathcal{G}^*$ after vertex and edge elimination (right). The output is a refined set of vertices and edges that participate in at least one subgraph $\mathcal{H}$ that matches $\mathcal{G}_0$. Here (and in the rest of the article), vertex metadata are presented as colored shapes. The eliminated vertices and edges are colored solid grey. (Reused from Reza et al. [2018].)

that eliminates all and only the vertices and edges that do not participate in any match, limits the exponential growth of the algorithm state, scales to massive graphs and distributed memory machines with a large number of processing elements, and supports arbitrary search templates. The result of pruning is the *complete set* of all vertices and edges that participate in at least one match, with *no false positives* or *false negatives*. Figure 1 illustrates the general idea using an example graph and a search template.

(ii) Second, we observe that *full match enumeration* is not the most efficient avenue to support many high-level graph analysis scenarios. Depending on the final goal of the user, pattern matching problems fall into a number of categories that include the following: (a) determining if a match exists in the background graph (yes/no answer), (b) selecting all the vertices and edges that participate in matches, (c) ranking these vertices or edges based on their *centrality with respect to the search template*, e.g., the frequency of their participation in matches, (d) counting/estimating the total number of matches, or (e) enumerating all distinct matches in the background graph. The traditional approach [P. Cordella et al. 2004; Ullmann 1976] is to first enumerate the matches (category (e) above) and to use the result to answer (a)–(d). However, this approach is limited to small background graphs or is dependent on a low number of near and exact matches within the background graph (due to exponential growth of the algorithm state). Our experiments suggest that that a pruning-based approach is not only a practical solution to scenarios (a)–(d) (and to other pattern matching related analytics) but also an efficient path toward full match enumeration in large graphs. There are three reasons for the effectiveness of this approach: First, the pruned graph can be multiple orders of magnitude smaller than the background graph, and existing high-complexity enumeration routines thus become applicable. Second, our pruning techniques collect additional key information to accelerate match enumeration—for each vertex in the pruned graph, our algorithms build a list of its possible match(es) in the template. Last, the intermediate algorithm state is much smaller.

(iii) Third, such pruning approach lends itself well to developing a *vertex-centric* algorithmic solution, and this makes it possible to harness existing high-performance, vertex-centric frameworks (e.g., GraphLab [Gonzalez et al. 2012], Giraph [Giraph 2016], or HavoqGT [Pearce et al. 2014]). In our vertex-centric formulation for pruning, a vertex must satisfy two types of constraints, namely, *local* and *non-local*, to possibly be part of a match. *Local constraints* involve only the vertex and its neighborhood: A vertex in an exact match needs to (a) match the label of a corresponding vertex in the template and (b) have edges to vertices labeled as prescribed in the adjacency structure of this corresponding vertex in the search template. *Non-local constraints* are topological requirements beyond the immediate neighborhood of a vertex (e.g., that the vertex must be part of a cycle). We describe how these constraints are generated, and our algorithmic solution to verify them in Section 4.

(iv) Finally, decomposing the search template in a set of constraints enables, in addition to *exact matching*, efficiently supporting a number of additional usage scenarios. These include the following: (a) Tradeoffs between precision and time-to-solution as search can be stopped early after checking only a subset of the constraints, leading to lower precision in the solution set (Reza et al. [2018], Section 5E); (b) *incremental searches*, an interactive search scenario where the user incrementally updates the search template (while the system takes advantage of the existence of common constraint(s) in the past and the updated search template, to offer fast response time) (Section 7.7); (c) *exploratory search*, a search scenario where the user presents an over-constrained search template that may not have any match, and the system finds the "nearest" matches (e.g., the ones that satisfy most of the constraints of the original search template) (Section 7.7); and, finally, (d) *approximate searches* based on *edit-distance* [Bunke and Allermann 1983; Reza et al. 2020a; Zhang et al. 2010].

Contributions. This article serves two goals: First, it is a synthesis of an ongoing long-term project [Reza et al. 2017, 2020a, 2020b, 2018; Tripoul et al. 2018], and, second, it presents new system features, usage scenarios, empirical experiments, and comparisons with related projects that strengthen the confidence that pattern matching based on iterative pruning via constraint checking is an effective and scalable approach.

Summary of Previously Published Work. We have introduced the *constraint checking* approach and discussed the opportunities it presents to pattern matching in large-scale graphs in a preliminary study [Reza et al. 2017]. This study focused on the exact matching scenario only and a restricted set of search templates: *acyclic* or *edge-monocyclic* with unique vertex labels (see Section 3 for definitions). A key contribution of this preliminary study proving that, for some templates, only inexpensive local constraint checking is sufficient for a precise solution. Following this initial investigation, we introduced PruneJuice (PJ) [Reza et al. 2018], a distributed system for exact pattern matching that is *generic* (no restrictions on the set of patterns supported), *precise* (no false positives), *offers 100% recall* (retrieves all matches), is *efficient* (small algorithm state ensuring low generated network traffic), and is *scalable* (able to process graphs with up to trillions of edges on tens of thousands of cores). Here, also the focus was the exact matching scenario only. Strong and weak scaling experiments using massive background graphs and scaling to up to 1,024 nodes (36,864 cores), confirmed the scalability of this approach. We demonstrated that, depending on the input, pruning leads to a *solution subgraph* that can be orders of magnitude smaller than the original background graph, which enables match enumeration and counting in massive graphs. While this study used simple heuristics to select and order constraints, we have demonstrated the effectiveness of advanced heuristics for constraint selection and ordering in the context of a shared memory implementation in Tripoul et al. [2018]. At the level of system architecture and implementation, the following key design ingredients make our system successful: asynchronicity and aggressive vertex and edge elimination while harnessing massive parallelism, intelligent work aggregation to ensure low message overhead, and lightweight per-vertex state. While in this article we focus on exact matching and closely associated scenarios, we have shown the finer granularity a constraint-based approach facilitates, can enable a version of approximate matching based on edit-distance [Reza et al. 2020a].

Summary of New Contributions. While one goal of this manuscript is to synthesize and organize the experience we have acquired during this project, it also includes an entirely new evaluation section, and original material as follows.

(i) *Demonstrating Support for a Diverse Set of Graph Analytics Scenarios* (Section 7.7). We show the ability of our approach to efficiently support an *exploratory search* scenario where the user starts form an over-constrained search template the system progressively relaxes the search until matches are found. We also present an overview of an *interactive incremental search* that supports the following usage scenario: The user may not know exactly what (s)he is looking for and, based on returned results, will incrementally revise the search template, possibly multiple times. Two high-level features presented by the exact matching pipeline are essential to support these additional scenarios: First, decomposing the search template into a set of constrains enables partial result reuse; second, a prunning-based approach makes it natural to focus on the part of the data of interest thus improving locality and reducing generated network traffic. We discuss the unique design optimizations enabled by the constraint checking approach to support these scenarios and demonstrate their effectiveness through experiments on massive graphs.

(ii) *Comparison with State-of-the-Art Systems* (Section 7.8). We extensively compare our work with two additional state-of-the art distributed solutions, Arabesque [Teixeira et al. 2015] (Section 7.8.3) and TriAD [Gurajada et al. 2014] (Section 7.8.2), in multiple scenarios (match enumeration and counting) and using labeled and unlabeled templates and multiple real-world graphs. These experiments demonstrate the significant advantages our system offers when handling large graphs and complex labeled or unlabeled patterns.

(iii) *Additional System Optimizations and Experiments Using New Datasets.* We have further added a number of optimizations aimed at enhancing performance, scalability, robustness, and efficiency. These include the following: *work aggregation*, a light-weight yet highly effective technique to prevent relaying duplicate messages (Section 5); load balancing (Section 7.6); and the ability to control the processing rate to lower memory pressure. We demonstrate that these techniques offer multitude of performance gains as well as robustness when processing at a massive scale (Section 7.5, Section 7.6, and Reza et al. [2018]). The extended evaluation includes new real-world graphs and three synthetic graphs with varying topology, skewness, and degree distribution and several new search templates, both labeled and unlabeled (Section 7.4).

(iv) *Bottleneck Analysis and Insights into Artifacts That Influence Performance.* We present experiments that uncover artifacts that influence performance along multiple axes. We explore the artifacts that cause load imbalance (Section 7.6); and we investigate the influence of search template and background graph properties (e.g., label distribution and topology) on runtime performance (Section 7.9 and Section 7.10).

(v) *Complexity Analysis* (Section 6). We present runtime, message count, and storage complexity of the core constraint checking algorithms. We also present a theoretical complexity argument that motivates the performance gains we observe.

2 RELATED WORK

The volume of related work on graph processing in general [Giraph 2016; Gonzalez et al. 2012, 2014; Hong et al. 2015; Malewicz et al. 2010; Sundaram et al. 2015], and on pattern matching algorithms in particular [Berry et al. 2007; Fan et al. 2013; Mckay and Piperno 2014; P. Cordella et al. 2004; Ullmann 1976; Zhu et al. 2011], is humbling. We summarize closely related work in Table 1.

2.1 General Algorithmic Approaches for Exact Pattern Matching

Early work on graph pattern matching mainly focused on solving the graph isomorphism problem [Ullmann 1976]. The well-known Ullmann's algorithm [Ullmann 1976] and its improvements (in terms of join order, pruning strategies, and space complexity), e.g., VF2 [P. Cordella et al. 2004] and QuickSI [Shang et al. 2008], belong to the family of *tree-search*-based algorithms. Ullman proposed a backtracking algorithm that finds exact matches by incrementing partial solutions and

Table 1. Comparison of Past Work on Distributed Pattern Matching

Contribution	Model	Framework/ Language	Match Type	Max. Query Size	Metadata	#Compute Nodes	Max. Real Graph	Max. Synthetic Graph
Arabesque [Teixeira et al. 2015]	Tree-search	Spark	Exact	10 edges	N/A	20	887M edges	N/A
QFrag [Serafini et al. 2017]	Tree-search	Spark	Exact	7 edges	Real	10	117M edges	N/A
Fractal [Dias et al. 2019]	Tree-search	Spark	Exact	10 edges	Real	10	44M edges	N/A
PGX.D/Async [Roth et al. 2017]	Async. DFS	Java/C++	Exact	4 edges	Synthetic	32	N/A	2B edges (Unif. rand.)
G-Miner [Chen et al. 2018]	Tree-search	C++	Exact	4 edges	N/A	15	1.8B edges	N/A
Sun et al. [Sun et al. 2012]	Subgraph Indexing	C#.Net4	Exact	30 edges	Synthetic	12	16.5M edges	4B vertices
Plantenga [Plantenga 2013]	Tree-search	Hadoop	Approximate	4-Clique	Real	64	107B edges	R-MAT Scale 20
SAHAD [Zhao et al. 2012]	Color-coding	Hadoop	Approximate	12 vertices	Synthetic	40	N/A	269M edges
FASCIA [Slota and Madduri 2014]	Color-coding	MPI	Approximate	12 vertices	N/A	15	117M edges	1M edges (Erdős-Rényi)
Chak. et al. [Chakaravarthy et al. 2016]	Color-coding	MPI	Approximate	10 vertices	N/A	512 (BG/Q)	2.7M edges	R-MAT
Gao et al. [Gao et al. 2014]	Subgraph Indexing	Giraph	Approximate	50 vertices	Synthetic	28	3.7B edges	N/A
Ma et al. [Ma et al. 2012]	Graph Simulation	Python	Approximate	15 vertices	Type only	16	5.1M edges	100M vertices
Fard et al. [Fard et al. 2013]	Graph Simulation	GPS	Approximate	N/A	N/A	8	300M edges	N/A
ASAP [Iyer et al. 2018]	Neighborhood Sampling	Spark	Probabilistic	6 edges	N/A	16	3.7B edges	N/A
Yuan et al. [Yuan et al. 2015]	Tree-search/Join	Java	Exact	17 edges	N/A	17	1.4B edges	64M vertices

The table highlights the characteristics of the solution presented (exact vs. approximate matching), its implementation infrastructure, and summarizes the details of the largest-scale experiment performed. We highlight the fact that *our solution is unique in terms of demonstrated scale, ability to perform exact matching, and ability to retrieve all matches.*

uses heuristics to prune unprofitable paths. VF2 improves the time and space complexity over Ullman's algorithm. The algorithm uses a heuristic that is based on the analysis of the vertices adjacent to vertices that have been included in a partial solution. The VF2 algorithm is known to be robust and performs well in practice and consecutively has been included in the popular Boost Graph Library (BGL) [BGL 2017]. A recent effort, Turbo$_{\text{ISO}}$ [Han et al. 2013], is considered to be the most optimized among the tree-search-based sequential subgraph isomorphism techniques. (Note that the pattern search can be performed in a depth-first or a breadth-first manner. The naïve pattern matching technique recursively searches the full template from each vertex in the background graph in a depth-first manner. The tree-search algorithms are merely optimizations of the depth-first search technique.)

For large graphs, a tree search may fail midway and needs to backtrack, and, hence, this technique can be expensive. Efficient distributed implementation of this approach is difficult for a number of reasons: Existing algorithms are inherently sequential and difficult to parallelize. Furthermore, a key limitation of this technique is that the number of possible join operations (the process of adding a graph edge to an intermediate match) is combinatorially large; which makes its application to generic patterns and massive graphs, with billions or trillions of edges, impractical. Also, the above algorithms use heuristics for join order selection [Han et al. 2013], as a result, often the performance is sensitive to the graph topology, label frequency, and relies on expensive preprocessing for join order optimization, such as sorting the neighbor vertices by degree.

Perhaps the best known exact matching algorithm that does not belong to the family of tree-search algorithms is Nauty due to McKay [Mckay and Piperno 2014], which is based on *canonical labeling* of the graph. This approach, however, has high preprocessing overhead. Nauty can perform verification for isomorphism in $O(n^2)$ time (where n is the number of vertices in the background graph), however, transforming arbitrary input graphs to the canonical form requires exponential time [Miyazaki 1997].

In the same spirit as database indexing, *subgraph indexing* (i.e., indexing of frequent subgraph structures) is an approach attempted to reduce the number of join operations (between subgraph structures) and to lower query response time, e.g., SpiderMine [Zhu et al. 2011], R-Join [Cheng et al. 2008], C-Tree [He and Singh 2006], SAPPER [Zhang et al. 2010], TriAD [Gurajada et al. 2014], and the contributions by Sun et al. [2012] and Gao et al. [2014]. Unfortunately, for a billion edge graph, this approach is infeasible to generalize: First, searching frequent subgraphs in a large graph is notoriously expensive. Second, depending on the topology of the search template(s) and the background graph, the size of the index is often superlinear relative to the size of the graph [Sun et al. 2012].

2.2 Distributed Pattern Matching Solutions

We review a number of projects that offer pattern matching on a shared-nothing architecture aiming either to reduce time-to-solution or to scale to search in large background graphs. Table 1 summarizes the key differentiating aspects and the scale achieved. Below we group the contributions into exact and approximate matching categories.

Solutions Offering Exact Matching. Arabesque [Teixeira et al. 2015] is a distributed framework offering precision and recall guarantees, implemented on top of Apache Spark [Spark 2017] and Giraph [Giraph 2016]. Arabesque provides an API based on the Think Like an Embedding (TLE) paradigm, to express graph mining algorithms and a Bulk Synchronous Parallel (BSP) implementation of the embedding (pattern) search engine (which follows the tree-search approach for match enumeration and counting). 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 (the implementation also exploits HDFS storage to maintain partially computed embeddings). Through evaluation using several real-world graphs, Teixeira et al. [2015] showed Arabesque's superiority over other two key systems: G-Tries [Ribeiro and Silva 2014] and GRAMI [Elseidy et al. 2014]. In Section 7.8.3, we directly compare our work with Arabesque.

QFrag [Serafini et al. 2017] is a general purpose exact pattern matching system, built on top of Arabesque. Similar to Arabesque, QFrag assumes that the entire graph fits in the memory of each compute node and uses data replication to enable search parallelism. QFrag employs a sophisticated load balancing strategy to reduce time-to-solution. In QFrag, each replica runs an instance of the tree-search-based pattern enumeration algorithm, $\text{Turbo}_{\text{ISO}}$ [Han et al. 2013] (an improvement of Ullmann's algorithm [Ullmann 1976]). Through evaluation, the authors demonstrated QFrag's performance advantages over two other distributed pattern matching systems: TriAD [Gurajada et al. 2014], an MPI-based distributed Resource Description Framework (RDF) [RDF 2017] engine based on an asynchronous distributed join algorithm, and GraphFrames [Dave et al. 2016; GraphFrames 2017], a graph processing library for Apache Spark, also based on distributed join operations. Although Arabesque and QFrag outperform most of their competitors in terms of time-to-solution, they replicate the entire graph in the memory of each compute node, which limits their applicability to relatively small graphs. In Section 7.8.1 and Section 7.8.2, we present direct comparison of our work with QFrag and TriAD, respectively.

Similarly to Arabesque, G-Miner [Chen et al. 2018] offers a high-level API for implementing graph mining algorithms; however, its applicability seems to be restricted to a limited scenarios as evaluation results were presented only for counting triangles and small cliques. A new framework Fractal [Dias et al. 2019], also based on the TLE abstraction, addresses several limitations of Arabesque to offer improved performance and memory efficiency.

PGX.D/Async [Roth et al. 2017] is a distributed system by Oracle Labs offering exact matching. It relies on asynchronous depth-first traversal for match enumeration. PGX.D/Async offers an MPI-based implementation and incorporates a flow control mechanism with a deterministic guarantee of search completion under a finite amount of memory; however, compared to our work, was demonstrated at a much smaller scale, in terms of graph sizes and number of compute nodes.

Sun et al. [2012] present an exact subgraph matching solution that follows the tree-search and join approach and demonstrate it on large synthetic graphs, using larger search templates than in Plantenga [2013], yet not on real-world graphs. Also, the authors mentioned that they terminate the search after the match-count have reached a predefined threshold that was set to 1,024 in their experiments (i.e., does not offer recall guarantees).

The Graph database Engine for Multithreaded Systems (GEMS), a framework for implementing RDF databases on distributed platforms [Castellana et al. 2015] supports SPARQL and GraQL queries, as well as user queries written in C++. GEMS exploits the Partitioned Global Address Space (PGAS) programming model: PGAS exposes the distributed memory of cluster nodes with a shared memory abstraction [Morari et al. 2015]. The GEMS runtime engine manages parallelism supporting thousands of lightweight parallel tasks per node. The authors have shown GEMS scalability using an RDF dataset with up to 10 billion triples on up to 128 compute nodes on an High Performance Computing (HPC) platform [Morari et al. 2015]. GraQL is a query language for GEMS that address a number of limitations of typical relational and native graph abstractions for supporting the Property Graph model [Chavarría-Miranda et al. 2016]. Choudhury et al. [2015] also explored the problem of pattern matching in streaming graphs within GEMS and evaluated their solution using acyclic search templates.

A number of projects on high-performance pattern matching, primarily target genome sequencing/assembly problems. In Low et al. [2007], the authors present a solution for multiple sequence alignment for handling a large number of protein sequences on a distributed platform with the mesh topology. The technique divides the computational load among compute nodes as opposed to dividing a protein sequence among compute nodes, with the goal of maximizing throughput. In Liu et al. [2013], a parallel sequence assembly solution for multi-core shared memory platforms is presented: the solution incorporates a suffix array-based data structure and can efficiently handle a large number of *reads*. In Yin et al. [2016], an improved technique for median computation in gene matching (or difference computation) is proposed: the solution models the problem in a manner that enables harnessing subgraph matching and search space reduction, toward offering efficiency. Makkar et al. [2017] presents a GPU-based solution for triangle counting in dynamic graphs. The authors presented an inclusion-exclusion formulation for the problem that correctly counts the number of triangles; the solution also improves complexity bounds over prior approaches. Green et al. [2018] presents Logarithmic Radix Binning for triangle counting; the work presents a multi-threaded and vectorized solution that also eliminates branch misprediction.

Solutions Targeting Approximate Matching. The best demonstrated scale is offered by Plantenga [2013]: a MapReduce implementation of the *walk*-based algorithm for identifying *type-isomorphic* (approximate) matches, originally proposed in Berry et al. [2007]. Plantenga introduced the idea of *walk-level constraints* to type-isomorphism—the added constraints are expected to reduce the search space of candidate walks

SAHAD [Zhao et al. 2012] is a MapReduce implementation of the *color-coding* algorithm [Alon et al. 2008] originally developed for approximating the count of treelike patterns (a.k.a. *treelet*) in protein-protein interaction networks. SAHAD follows a hierarchical sub-template explore-join approach. Its application was presented only on small graphs with up to ~300 million edges. FASCIA [Slota and Madduri 2014] is also a color-coding-based solution for approximate treelet counting, whose MPI-based implementation offers superior performance to SAHAD. Chakaravarthy et al. [2016] extended the color-coding algorithm to count patterns with cycles (although does not support arbitrary patterns) and presented an MPI-based distributed implementation. However, the authors demonstrated performance on graphs with only a few million edges.

ASAP [Iyer et al. 2018] is a distributed solution enabling approximate match counting within a given error bound. ASAP is based on Apache Spark and GraphX [Gonzalez et al. 2014]. Like Arabesque, ASAP provides a high-level API for implementing graph mining algorithms. ASAP implements a neighborhood sampling technique that estimates the template match count by sampling

Table 2. Symbolic Notation Used

Object(s)	Notation
template graph, vertices, edges	$\mathcal{G}_0(\mathcal{V}_0, \mathcal{E}_0)$
template graph sizes	$n_0 := \lvert\mathcal{V}_0\rvert$, $m_0 := \lvert\mathcal{E}_0\rvert$
vertices in the template graph	$\mathcal{V}_0 := \{q_0, q_1, \ldots, q_{n_0-1}\}$
edges in the template graph	$(q_i, q_j) \in \mathcal{E}_0$
set of vertices adjacent to q_i in $\mathcal{G}_0$	$adj(q_i)$
background graph, vertices, edges	$\mathcal{G}(\mathcal{V}, \mathcal{E})$
background graph sizes	$n := \lvert\mathcal{V}\rvert$, $m := \lvert\mathcal{E}\rvert$
vertices in the background graph	$\mathcal{V} := \{v_0, v_1, \ldots, v_{n-1}\}$
edges in the background graph	$(v_i, v_j) \in \mathcal{E}$
set of vertices adjacent to v_i in $\mathcal{G}$	$adj(v_i)$
maximum vertex degree in $\mathcal{G}$	d_{max}
average vertex degree in $\mathcal{G}$	d_{avg}
standard deviation of vertex degree in $\mathcal{G}$	d_{stdev}
label set	$\mathcal{L} = \{0, 1, \ldots, n_\ell - 1\}$
vertex label of q_i	$\ell(q_i) \in \mathcal{L}$
vertex match function	$\omega(v_i) \subset \mathcal{V}_0$
set of non-local constraints for $\mathcal{G}_0$	$\mathcal{K}_0$
matching subgraph, vertices, edges	$\mathcal{H}(\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H})$
solution subgraph, vertices, edges	$\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$

the edges in the background graph. Unlike our system, the output produced by ASAP is only probabilistic; ASAP does not offer precision and recall guarantees, although it allows tradeoff between the result accuracy and time-to-solution and provides a technique to bound the counting error.

Gao et al. [2014] introduces an approximate matching technique based on tree-search and join and evaluate it on large queries (up to 50 vertices). Here, a query template is converted in to a single-sink directed acyclic graph and message transition follows its topology. Distributed approximate matching solutions based on *graph simulation* [Henzinger et al. 1995] are proposed in Fard et al. [2013] and Ma et al. [2012], although both are evaluated only on relatively small real-world graphs.

3 PRELIMINARIES

We aim to identify all structures within a large *background graph*, $\mathcal{G}$, identical to a small connected *template graph*, $\mathcal{G}_0$. We describe general graph properties for $\mathcal{G}$, and use the same notation (summarized in Table 2) for other graph objects.

A graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a collection of n vertices $\mathcal{V} = \{0, 1, \ldots, n-1\}$ and m edges $(i, j) \in \mathcal{E}$, where $i, j \in \mathcal{V}$ (i is the edge's *source* and j is the *target*). Here, we only discuss simple (i.e., no self-edges), undirected, vertex-labeled graphs, although the techniques are applicable to directed, non-simple graphs, with labels on both edges and vertices. An *undirected* $\mathcal{G}$ satisfies $(i, j) \in \mathcal{E}$ if and only if $(j, i) \in \mathcal{E}$. Vertex i's *adjacency list*, $adj(i)$, is the set of all j such that $(i, j) \in \mathcal{E}$. A *vertex-labeled graph* also has a set of n_ℓ labels $\mathcal{L}$ of which each vertex $i \in \mathcal{V}$ has an assignment $\ell(i) \in \mathcal{L}$.

A *walk* in $\mathcal{G}$ is an ordered subsequence of $\mathcal{V}$ where each consecutive pair is an edge in $\mathcal{E}$. A walk with no repeated vertices is a *path*. A path with equal first and last vertex is a *cycle*. An *acyclic* graph has no cycles.

We further characterize graphs with with cycles. Two *disjoint* cycles have no edge in common. Two *distinct* cycles have at least one edge not in common. We define the *cycle degree* of edge $(i,j) \in \mathcal{E}$ as the number of distinct cycles (i,j) is in, written $\delta_{(i,j)}$. The maximum cycle degree is $\delta_{max} := \max_{\mathcal{E}} \delta_{(i,j)}$. A graph is *edge-monocyclic* if $\delta_{max} = 1$.

We discuss several graph objects simultaneously: the *template graph* $\mathcal{G}_0(\mathcal{V}_0, \mathcal{E}_0)$, the *background graph* $\mathcal{G}(\mathcal{V}, \mathcal{E})$, and the *current solution subgraph* $\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$, with $\mathcal{V}^* \subset \mathcal{V}$ and $\mathcal{E}^* \subset \mathcal{E}$. Our techniques iteratively refine $\mathcal{V}^*$ and $\mathcal{E}^*$ until they converge to the union of all subgraphs of $\mathcal{G}$ that *exactly match* the template, $\mathcal{G}_0$.

For clarity, when referring to vertices and edges from the template graph, $\mathcal{G}_0$, we will use the notation $q_i \in \mathcal{V}_0$ and $(q_i, q_j) \in \mathcal{E}_0$. Conversely, we will use $v_i \in \mathcal{V}$ and $(v_i, v_j) \in \mathcal{E}$ for vertices and edges from the background graph $\mathcal{G}$ or the solution subgraph $\mathcal{G}^*$. In the rest of the article, particularly in Section 5, Algorithms 3, 4, 5, 6, and 7, we use subscripts (i, j, and k) to differentiate between distinct vertices of the background graph $\mathcal{G}$ and that of the template graph $\mathcal{G}_0$. (For example, in Algorithm 3, a vertex v_j's state may be updated if it has received a message from another vertex v_i, where $(v_i, v_j) \in \mathcal{E}$. To avoid confusion, we use a different subscript to represent a template vertex, e.g., q_k. When it is clear from context, we adapt notation to avoid double subscripts, using q_0 or v_5 in place of q_{i_0} or v_{i_5}.

We assume $\mathcal{G}_0$ is connected, because if $\mathcal{G}_0$ has multiple components, then the matching problem can be easily reduced to solving it for each component individually.

Definition 1. A subgraph $\mathcal{H}(\mathcal{V}_\mathcal{H}, \mathcal{E}_\mathcal{H}), \mathcal{V}_\mathcal{H} \subset \mathcal{V}, \mathcal{E}_\mathcal{H} \subset \mathcal{E}$ is an exact match of template graph $\mathcal{G}_0(\mathcal{V}_0, \mathcal{E}_0)$ (in notation, $\mathcal{H} \sim \mathcal{G}_0$) if there exists a bijective function $\phi : \mathcal{V}_0 \longleftrightarrow \mathcal{V}_\mathcal{H}$ with the properties (note that ϕ may not be unique for a given $\mathcal{H}$):

(i) $\ell(\phi(q)) = \ell(q)$, for all $q \in \mathcal{V}_0$ and
(ii) $\forall (q_1, q_2) \in \mathcal{E}_0$, we have $(\phi(q_1), \phi(q_2)) \in \mathcal{E}_\mathcal{H}$
(iii) $\forall (v_1, v_2) \in \mathcal{E}_\mathcal{H}$, we have $(\phi^{-1}(v_1), \phi^{-1}(v_2)) \in \mathcal{E}_0$.

Intuition for Our Solution. The algorithms we develop here, iteratively refine a *vertex match* function $\omega(v) \subset \mathcal{V}_0$ such that, for every $v \in \mathcal{V}$, $\omega(v)$ stores a superset of all template vertices v can possibly match. Set $\omega(v)$ converges to contain all possible values of $\phi^{-1}(v)$, where v is involved in one or more matching subgraphs. When a single constraint involving $q \in \mathcal{V}_0$ is violated/unmet, q is no longer a possibility for v in a match and q is removed: $\omega(v) \leftarrow \omega(v) \setminus \{q\}$.

Remark 1. Given an ordered sequence of all n_0 vertices $\{q_1, q_2, \ldots, q_{n_0}\} \subset \mathcal{V}_0$, a simple (although potentially expensive) search from $v_1 \in \mathcal{V}^*$ verifies if v_1 is in a match, with $\phi(q_1) = v_1$, or not. The search lists an ordered sequence $\{v_1, v_2, \ldots, v_{n_0}\} \subset \mathcal{V}^*$, with ϕ defined as $\phi(q_k) = v_k$. Search step k proposes a new v_k, checking Def. 1 (i) and (ii). If all checks are passed, then the search accepts v_k and moves on to step $(k+1)$, but terminates if no such v_k exists in $\mathcal{V}^*$. If the full list is generated with all label and edge checks passed, then there exists a $\mathcal{H} \sim \mathcal{G}$ with $\mathcal{V}_\mathcal{H} = \{v_1, v_2, \ldots, v_{n_0}\}$.

We call this Template-Driven Search (TDS), presented in the next section and develop an efficient distributed version in Section 5, to apply to the solution $\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$. If TDS has been applied successfully, then there are no false positives remaining independently of the structure of $\mathcal{G}_0$. We note that TDS is needed only for the general case, and multiple other specific cases simpler verification routines can be used.

4 PATTERN MATCHING VIA CONSTRAINT CHECKING: SOLUTION OVERVIEW

Our goal is to realize a technique that systematically eliminates all the vertices and edges that do not participate in any exact match $\mathcal{H} \sim \mathcal{G}_0$. This approach is motivated by viewing the template

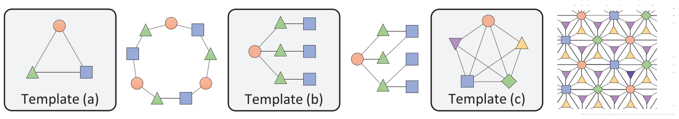


Fig. 2. Three examples of search templates and background graphs that justify the full set (local and non-local) of pruning constraints. Template (a) is a 3-Cycle; cycles of length $3k$ with repeated labels in the background graph meet neighborhood constraints, surviving local constraint checking. Template (b) contains several vertices with non-unique labels; to its right there is a background graph that meets individual point-to-point path constraints, also surviving (non-local) path checking. Template (c) is characterized by two 4-Cliques that overlap at a 3-Cycle; the background graph structure to the right is doubly periodic (a 4×3 torus) and meets all edge and vertex cycle constraints, surviving (non-local) cycle checking. Templates (b) and (c) require template-driven search to guarantee no false positives; template (a) only needs cycle checking in addition to checking the local constraints. (Reused from Reza et al. [2018].)

$\mathcal{G}_0$ as specifying a set of constraints the vertices and edges that participate in a match must meet. As a trivial example, any vertex v whose label $\ell(v)$ is not present in $\mathcal{G}_0$, cannot be present in an exact match. A vertex in an exact match also needs to have non-eliminated edges to non-eliminated vertices labeled as prescribed in the adjacency structure of the corresponding template vertex.

Local constraints that involve a vertex and its immediate neighborhood can be checked by having vertices communicate their "provisional" template match(es) with their one-hop neighbors in the solution subgraph $\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$ (i.e., the currently pruned background graph). We call this process Local Constraint Checking (LCC) and note that, since communication is limited to one-hop vertex neighbors, this is a relatively low cost step. For a restricted set of search templates, acyclic or edge-monocyclic with unique vertex labels, LLC is sufficient for a precise solution [Reza et al. 2017]. For more complex search templates, in our experimental setup, LLC often removes the bulk of non-matching vertices and edges; although in many cases, most of the search effort is allocated to verifying the non-local constrains we describe below [Reza et al. 2018].

Templates with topological requirements beyond the immediate neighborhood of a vertex (i.e., templates with cycles and/or repeated vertex labels) require additional routines to check non-local properties to guarantee that all non-matching vertices are eliminated. (Figure 2 illustrates the need for these additional checks with examples). To support arbitrary templates, we have developed a process that we dub Non-local Constraint Checking (NLCC): First, based on the search template $\mathcal{G}_0$, we generate the set of constraints $\mathcal{K}_0$ that are to be verified and then prune the graph using each of them.

4.1 Overview of the Constraint Checking Technique

Algorithm 1 presents an overview of our solution. This section provides high-level descriptions of the local and non-local constraint checking routines while Section 5 provides the detailed distributed asynchronous algorithms for a vertex-centric abstraction. As an overview, Figure 3 illustrates the complete workflow for the graph and pattern in Figure 1, for which constraint generation is detailed in Table 3.

Local Constraint Checking involves a vertex and its immediate neighborhood. The algorithm performs the following two operations: (i) *Vertex elimination*—the algorithm excludes the vertices that do not have a corresponding label in the template then, iteratively, excludes the vertices that do not have neighbors as labeled in the template. For templates that have vertices with multiple neighbors with the same label, the algorithm verifies if a matching vertex in the background graph has a minimum number of distinct active neighbors with the same label as prescribed in the template. (ii) *Edge elimination*—this excludes edges to eliminated neighbors and edges to neighbors

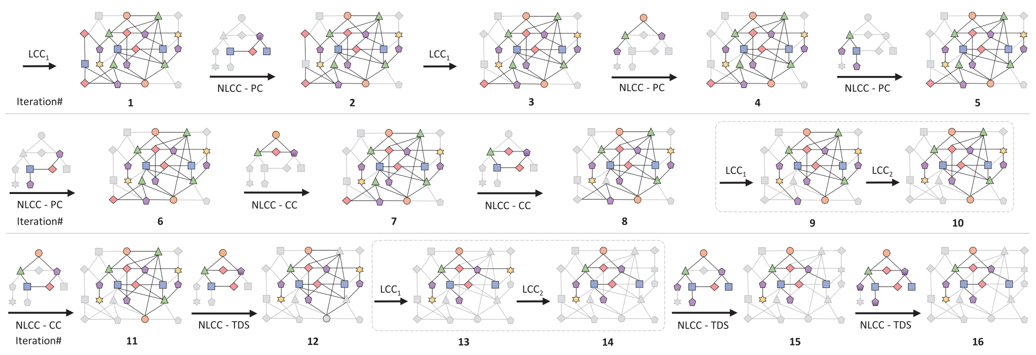


Fig. 3. Algorithm walk through for the example background graph and template in Figure 1, depicting which vertices and edges in $\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$ are eliminated (in solid grey) during each iteration. The non-local constraints for $\mathcal{G}_0$ are listed in Table 3. The example does not show application of some of the constraints as that do not eliminate vertices or edges. (Reused from Reza et al. [2018].)

ALGORITHM 1: Main Constraint Checking Loop

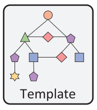
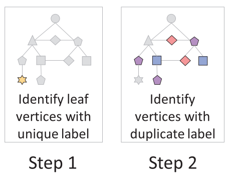
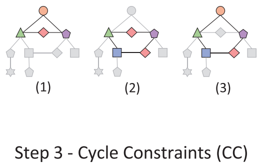
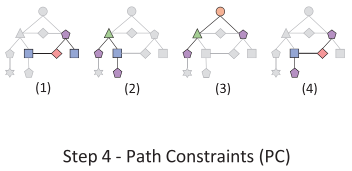
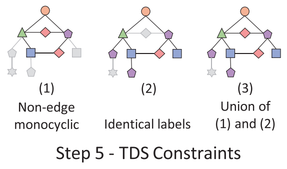
```
1:  **Input:** background graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, template $\mathcal{G}_0(\mathcal{V}_0, \mathcal{E}_0)$
2:  **Output:** solution subgraph $\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$
3:  $\mathcal{K}_0 \leftarrow$ NON_LOCAL_CONSTRAINTS ($\mathcal{G}_0$)
4:  INIT_VERTEX_STATE ($\mathcal{G}$, $\mathcal{G}_0$)
5:  $\mathcal{G}^* \leftarrow$ LOCAL_CONSTRAINT_CHECKING ($\mathcal{G}$, $\mathcal{G}_0$)
6:  while $\mathcal{K}_0$ is not empty do
7:      pick and remove the next constraint $C_0$ from $\mathcal{K}_0$
8:      $\mathcal{G}^* \leftarrow$ NON_LOCAL_CONSTRAINT_CHECKING ($\mathcal{G}^*$, $\mathcal{G}_0$, $C_0$)
9:      if any vertex has been eliminated or has one of its provisional matches removed then
10:         $\mathcal{G}^* \leftarrow$ LOCAL_CONSTRAINT_CHECKING ($\mathcal{G}^*$, $\mathcal{G}_0$)
11: return $\mathcal{G}^*$
```

whose labels do not match the labels prescribed in the adjacency structure of its corresponding template vertex (e.g., Figure 3, Iteration #1). Edge elimination is crucial for scalability, since, in a distributed setting, no messages are sent over eliminated edges thus significantly improving the overall efficiency of the system (evaluated in Section 7).

Non-local Constraint Checking aims to exclude vertices that fail to meet topological and label constraints beyond the one-hop neighborhood, that LCC is not guaranteed to eliminate (an example is presented Figure 2). We have identified three types of non-local constraints that can be verified independently: (i) cycle constraints (CC), (ii) path constraints (PC), and (iii) constraints that require TDS (see Remark 1). For arbitrary templates, TDS constraints based on aggregating multiple paths/cycles enable further pruning, and insure that pruning yields no false positives. Compared to CC and PC, checking TDS constraints, however, can be more expensive. To reduce the overall cost, we first generate single cycle- and path-based constraints, which are usually less costly to verify, and prune the graph using them before deploying TDS.

High-level Algorithmic Approach. Regardless of the constraint type, NLCC leverages a *token passing* approach: tokens are issued by background graph vertices whose corresponding template vertices are identified to have non-local constraints. After a fixed number of steps, we check if a token has arrived where expected (e.g., back to the originating vertex for checking the existence of a cycle). If not, then the token issuing vertex does not satisfy the required constraint and is eliminated. Along the token path, the algorithm verifies that all expected labels are encountered

Table 3. Step-by-step Illustration of Non-local Constraint Generation for the Template in Figure 1 (High-level Pseudocode, Accompanied by Pictorial Depiction)

	1: **Input:** template $\mathcal{G}_0(\mathcal{V}_0, \mathcal{E}_0)$ $\triangleright$ $\mathcal{G}_0$ is undirected and at least weakly connected 2: **Output:** non-local constraint set $\mathcal{K}_0$ of $\mathcal{G}_0$ 3: **procedure** NON_LOCAL_CONSTRAINTS($\mathcal{G}_0$) 4: $\mathcal{K}_0 \leftarrow \emptyset$; $CC \leftarrow \emptyset$; $PC \leftarrow \emptyset$; $TDS_{CC} \leftarrow \emptyset$; $TDS_{PC} \leftarrow \emptyset$; $TDS \leftarrow \emptyset$ $\triangleright$ $\mathcal{K}_0$ is an ordered set; the rest are lists for each constraint type	
Vertex Classification	5: $\mathcal{V}_u \leftarrow$ UNIQUE_LABEL_VERTICES($\mathcal{V}_0 \in \mathcal{G}_0$) 6: $\mathcal{V}_{ul} \leftarrow$ LEAF_VERTICES($\mathcal{V}_u \in \mathcal{G}_0$) $\triangleright$ Step 1 7: $\mathcal{V}'_0 \leftarrow \mathcal{V}_0 \setminus \mathcal{V}_{ul}$ 8: $\mathcal{E}'_0 \leftarrow \forall(q_i, q_j) \in \mathcal{E}_0$ **where** $q_i \in \mathcal{V}'_0$ **and** $q_j \in \mathcal{V}'_0$ 9: $\mathcal{G}'_0 \leftarrow (\mathcal{V}'_0, \mathcal{E}'_0)$ 10: $\mathcal{V}_d \leftarrow \mathcal{V}_0 \setminus \mathcal{V}_u$ $\triangleright$ Step 2	 Identify leaf vertices with unique label Identify vertices with duplicate label Step 1 Step 2
Cycle Constraints	11: $CC \leftarrow$ FIND_CIRCLES($\mathcal{G}'_0$)	
Path Constraints	12: **for all** vertex pairs $\{q_i, q_j\} \subset \mathcal{V}_d$ **where** $\ell(q_i) = \ell(q_j)$ **do** 13: $path \leftarrow$ FIND_SHORTEST_PATH($\mathcal{G}_0, q_i, q_j, 3$) $\triangleright$ a unique shortest path from q_i to q_j of length ≥ 3 14: **if** $path \neq \emptyset$ **then** $PC.add(path)$	
TDS Constraints	15: $TDS_{CC} \leftarrow$ TDS_CONSTRAINTS(CC) $\triangleright$ TDS cycle constraints, Step 5(1) 16: $TDS_{PC} \leftarrow$ TDS_CONSTRAINTS(PC) $\triangleright$ TDS path constraints, Step 5(2) 17: $TDS \leftarrow TDS_{CC} \cup TDS_{PC}$ 18: $TDS \leftarrow$ TDS_CONSTRAINTS(TDS) $\triangleright$ TDS constraints, Step 5(3)	 (1) Non-edge monocyclic (2) Identical labels (3) Union of (1) and (2) Step 5 - TDS Constraints
	19: **return** $\mathcal{K}_0 \leftarrow PC \cup CC \cup TDS_{CC} \cup TDS_{PC} \cup TDS$	

The figures show the steps to generate the required CC, PC, and higher-order constraints requiring TDS. Algorithm 2 is the pseudocode for the procedure TDS_CONSTRAINTS(). Definitions of the helper functions UNIQUE_LABEL_VERTICES(), LEAF_VERTICES(), and FIND_CIRCLES() are rather trivial and, hence, not included. (Figures adapted from Reza et al. [2018].)

and, where necessary, uses the path information accumulated with the token to verify that different/repeated vertex identity constraint expectations are met. Next, we discuss how each type of non-local constraint is verified.

Cycle Constraints. Higher-order structures within $\mathcal{G}$ that survive LCC, but do not contain $\mathcal{G}_0$, are possible if $\mathcal{G}_0$ contains a cycle (this happens if $\mathcal{G}$ contains one or more *unrolled* cycles as in Figure 2, Template (a)). To address this, we directly check for cycles of the correct length.

Path Constraints. If the template $\mathcal{G}_0$ has two or more vertices with the same label that are three or more hops away from each other, then structures in $\mathcal{G}$ that survive LCC, yet contain no match, are possible (Figure 2, Template (b)). Thus, for every vertex pair with the same label in $\mathcal{G}_0$, we directly check the existence of a path of the correct length and label sequence for prospective matching vertices in $\mathcal{G}^*$. Opposite to cycle checking, after a fixed number of steps, a token must be received by a vertex *different* from the token initiating vertex but with an identical label.

TDS Constraints. These are partial (for further pruning and performance optimization similar to path and cycle constraints) or complete (i.e., including all edges of the template) walks on the template (required to ensure correctness). The token *walks* the constraint in the background graph and verifies that each vertex visited meets its neighborhood constraints (Remark 1). In a distributed memory setting, this is done by maintaining a history of the walk and checking that previously visited vertices are revisited as expected. TDS constraints are crucial to guarantee zero false positives for templates that are *non-edge-monocyclic* or have repeated labels (Figure 2, Template (b) and (c)). Next, we describe how these three types of non-local-constraints are generated.

ALGORITHM 2: TDS Constraint Generation

```
1:  procedure TDS_CONSTRAINTS($\mathcal{K}_0'$)
2:      $i \leftarrow 0$;   $j \leftarrow 1$
3:      for all $\mathcal{K}_0'[i] \in \mathcal{K}_0'$;   $i \leftarrow i + 1$ do
4:          for all $\mathcal{K}_0'[j] \in \mathcal{K}_0'$;   $j \leftarrow j + 1$ do
5:              if $\mathcal{K}_0'[i]$ and $\mathcal{K}_0'[j]$ have at least one common vertex then
6:                  if $\mathcal{K}_0'[i]$ and $\mathcal{K}_0'[j]$ have at least one common edge or $type(\mathcal{K}_0') = PC$ then
7:                      $\mathcal{K}_0'[i] \leftarrow \mathcal{K}_0'[i] \cup \mathcal{K}_0'[j]$      ▷ union of two constraints (i.e., substructures of $\mathcal{G}_0$)
8:                      $\mathcal{K}_0' \leftarrow \mathcal{K}_0' \setminus \mathcal{K}_0'[j]$;   $j \leftarrow j - 1$
9:      if $type(\mathcal{K}_0') = PC$ then
10:         for all $C_0 \in \mathcal{K}_0'$ do
11:             $C_0 \leftarrow$ SPANNING_TREE($C_0$)
12:     return $\mathcal{K}_0'$
```

4.2 Non-local Constraint Generation

We generate non-local constraints following the heuristic presented in Table 3. The three types of non-local constraints, namely, cycle constraints, path constraints, and TDS constraints, are generated incrementally: For an example template, we provide a step-by-step illustration of the non-local constraint generation. Figure 3 shows a complete example of how pruning progresses using the generated constraints.

Step 1. Identify all the leaf vertices (i.e., a vertex with only one neighbor) with unique labels. They are not considered for non-local constraint checking as LCC guarantees pruning if there is no match.

Step 2. Identify all the vertices with duplicate labels. Path constraints are generated only for these vertices.

Step 3. If the template has cycles, then individual cycles are identified and a cycle constraint is generated for each cycle.

Step 4. For all possible combinations of vertex pairs with identical label, we identify all existing paths greater than or equal to three-hop length. (LCC precisely checks identical label pairs that are one or two hops from each other). One such path, for each vertex pair, is generated as a path constraint. Here, two optimization's are applied to minimize the number of path constraints to be verified: (i) If there are multiple paths connecting two terminal vertices, then the shortest path is generated as a path constraint. (ii) If all the edges comprising a path also belong to a cycle constraint, then that particular path is excluded from the set of path constraints. Verification of the cycle constraint will implicitly check for existence of a successful walk of appropriate length connecting the terminal vertices (of the path of interest).

Step 5. We generate TDS constraints in three steps. First, for templates with multiple cycles sharing more than one vertex (i.e., the template is non-edge-monocyclic), a TDS cyclic constraint is generated through the union of previously identified cycle constraints. This results in a higher-order cyclic structure with a maximal set of edges that cover all the cycles sharing at least one edge (e.g., Step 5(1)).

Second, for templates with repeated labels, a new TDS constraint is generated through the union of all previously identified path constraints. This procedure generates higher-order structure that covers all the template vertices with repeated labels (e.g., Step 5(2)).

The final step generates a TDS constraint as the union of the previously identified two constraints (e.g., Step 5(3)). Note that the above is a heuristic, more TDS constraints can be generated by creating various possible combinations of cycles and paths. Only this third step is mandatory to eliminate all false positives.

Constraint Optimization. Non-local constraint verification checks for existence of at least one successful walk of the appropriate length. There are alternatives to how tokens could be passed around to complete a walk. The non-local constraint generation also focuses on optimizing the walks for token passing.

Whenever possible, we orchestrate each walk so the vertices are visited in the increasing order of label frequency in the background graph. (This procedure has negligible overhead as label frequency is computed only once per label set and we only sort the vertex list of a template that, typically, has 10^1–10^2 elements). Here, the goal is to curb combinatorial growth of the algorithm state (or more specifically, in the distributed memory setting, the number of messages). This optimization has the potential of eliminating a large part of the graph without explorations deep into an excessive number of branches in the backgorund graph.

Non-local constraint generation also focuses of reducing the number of constraints and length of a walk (as the size of a constraint directly influences complexity). In addition to selecting the shortest path, if all the edges in a path constraint are also present in a cycle constraint, the path is ignored. When generating TDS constraints through union of the original path constrains, we are able to remove some (redundant) edges by obtaining a spanning tree for each TDS constraint (Algorithm 2, line #11).

Constraint Ordering Heuristics. We use a second set of heuristics to optimize the order in which constraints are scheduled for verification. First, we check for path and cycle constraints, since they tend to be less expensive than TDS constraints. Second, we order the non-local constraints with respect to increasing length of the walk as longer walks are more susceptible to combinatorial explosion. Tripoul et al. [2018] presents an avenue to design advanced heuristics.

Token Generation. For cyclic constraints, a token must be initiated from each vertex that may participate in the substructure, whereas for path constraints, tokens are only initiated from terminal vertices. Tokens are started from vertices (that belong to the same cyclic substructure) in the increasing order of their label frequency in the background graph. For duplicate/distinct label verification, there is also TDS path constraint checking. The substructure in question may contain a cycle or a tree. Similarly to path constraints, here tokens are initiated from vertices with duplicate labels.

5 DISTRIBUTED SYSTEM DESIGN AND IMPLEMENTATION

In this section, we present the constraint checking algorithms in the *vertex-centric* abstraction of HavoqGT [HavoqGT 2016], an MPI-based framework that supports *asynchronous* graph

ALGORITHM 3: Vertex State and Initialization

```
1:  vertex state: $\alpha(v_j) \leftarrow false$      ▷ indicates if a vertex $v_j$ is active ($true$) or eliminated ($false$)
2:  vertex state: $\omega(v_j) \leftarrow \emptyset$      ▷ list of possible vertex matches in template
3:  vertex state: $\varepsilon(v_j)$      ▷ map of active edges of a vertex $v_j$
4:  vertex state: $\tau(v_j) \leftarrow \emptyset$      ▷ set of already forwarded tokens by vertex $v_j$, used for work aggregation in NLCC (Algorithm 4)
5:  procedure INIT_VERTEX_STATE($\mathcal{G}$, $\mathcal{G}_0$)
6:      for all $v_j \in \mathcal{V}$ do
7:          for all $q_k \in \mathcal{V}_0$ do
8:              if $\ell(q_k) = \ell(v_j)$ then
9:                  if $\alpha(v_j) = false$ then
10:                     $\alpha(v_j) \leftarrow true$
11:                 $\omega(v_j).add(q_k)$
12:         if $\alpha(v_j) = true$ then
13:             for all $v_i \in adj(v_j)$ do
14:                 $\varepsilon(v_j).insert(v_i, \emptyset)$
15:         else
16:             $\varepsilon(v_j) \leftarrow \emptyset$      ▷ eliminate edges of an inactive vertex
```

algorithms in the distributed environment. Our choice for HavoqGT is driven by multiple considerations: First, unlike most graph processing frameworks that primarily support the BSP model, HavoqGT has been designed to support asynchronous algorithms, which is essential to achieve high-performance. Asynchronous algorithms can exploit the low latency (~1 μs) interconnect on leadership-class HPC platforms. Second, the framework has demonstrated excellent scaling properties for a number of graph traversal problems [Pearce et al. 2013, 2014]. Finally, it enables load balancing: HavoqGT's *delegate partitioned graph* distributes the edges of each high-degree vertex across multiple compute nodes, which is crucial for achieving scalability for scale-free graphs with skewed degree distribution.

In HavoqGT, graph algorithms are implemented as vertex-callbacks: the user-defined $visit()$ callback can only access and update the state of a vertex. The framework offers the ability to generate events (a.k.a. *visitors* in HavoqGT's vocabulary) that trigger this callback—either at the entire graph level using the $do\_traversal()$ method or for a neighboring vertex using the $push(visitor)$ call. When a vertex wants to pass data to a neighbor, invoking $push(visitor)$ enqueues the relevant visitor to the distributed message queue, which exploits MPI asynchronous communication primitives for exchanging messages. This enables asynchronous vertex-to-vertex communication. The asynchronous graph computation completes when all events have been processed, which is determined by a distributed quiescence detection algorithm [Wellman and Walsh 2000].

Algorithm 1 outlines the key steps of the graph pruning procedure. Below, we describe the distributed implementation of the local and non-local constraint checking, and match enumeration routines. Algorithm 3 lists the state maintained by each active vertex and its initialization.

5.1 Local Constraint Checking

Local Constraint Checking is implemented as an iterative process (Algorithm 4 and the corresponding callback, Algorithm 5). Each iteration initiates an asynchronous traversal by invoking the $do\_traversal()$ method and, as a result, each active vertex receives a visitor with $msg_{type} = init$. In the triggered $visit()$ callback, if the label of a vertex v_j in the graph is a match for the label of any vertex in the template and the vertex is still active, then it creates visitors for all its active neighbors in $\varepsilon(v_j)$ with $msg_{type} = alive$ (Algorithm 5, line #9). When a vertex v_j is visited with $msg_{type} = alive$, it verifies whether the sender vertex v_s satisfies one of its own (i.e., v_j's) local

ALGORITHM 4: Local Constraint Checking

```
1: $\eta(v_s, v_j)$ - verifies if $v_s$ satisfies a local constraint of $v_j$; returns $\omega(v_s)$ if constraints are met, $\emptyset$ otherwise
2: procedure LOCAL_CONSTRAINT_CHECKING($\mathcal{G}$, $\mathcal{G}_0$)
3:     do
4:         $do\_traversal(msg_{type} \leftarrow init)$
5:         barrier
6:         for all $v_j \in \mathcal{V}$ do
7:             $\omega' \leftarrow \emptyset$                              ▷ set of template matches for neighbors of $v_j$
8:             for all $v_i \in \varepsilon(v_j)$ do
9:                 if $\eta(v_i, v_j) = \emptyset$ then
10:                    $\varepsilon(v_j).remove(v_i)$                       ▷ edge eliminated
11:                    continue
12:                else
13:                    $\omega' \leftarrow \omega' \cup \eta(v_i, v_j)$     ▷ accumulate matched neighbor information
14:                    reset the value field of $v_i \in \varepsilon(v_j)$ for the next iteration
15:            for all $q_k \in \omega(v_j)$ do                         ▷ for each potential match
16:                if $adj(q_k) \nsubseteq \omega'$ then
17:                    ▷ $q_k$ does not meet neighbor requirements
18:                    $\omega(v_j).remove(q_k)$                         ▷ remove from the set of potential matches
19:                    continue
20:            if $\varepsilon(v_j) = \emptyset$ or $\omega(v_j) = \emptyset$ then
21:                $\alpha(v_j) \leftarrow false$                        ▷ vertex eliminated
22:    while vertices or edges are eliminated                           ▷ global detection
```

ALGORITHM 5: Local Constraint Checking Visitor

```
1: visitor state: $v_j$ - vertex that is visited
2: visitor state: $v_s$ - vertex that originated the visitor
3: visitor state: $\omega(v_s)$ - set of possible matches in template for vertex $v_s$
4: visitor state: $msg_{type}$ - $init$ or $alive$
5: procedure VISIT($\mathcal{G}$, $vq$)               ▷ $vq$ - visitor queue (the distributed message queue)
6:     if $\alpha(v_j) = false$ then return
7:     if $msg_{type} = init$ then
8:         for all $v_i \in \varepsilon(v_j)$ do
9:             $vis \leftarrow$ LCC_VISITOR($v_i$, $v_j$, $\omega(v_j)$, $alive$)
10:            $vq.push(vis)$
11:    else if $msg_{type} = alive$ then
12:        $\varepsilon(v_j).get(v_s) \leftarrow \omega(v_s)$
```

constraints by invoking the function $\eta(v_s, v_j)$. By the end of an iteration, if v_j satisfies all the template constraints, i.e., it has neighbors with the required labels (and, if needed, a minimum number of distinct neighbors with the same label as prescribed in the template), it stays active (i.e., $\alpha(v_j) = true$) for the next iteration. For templates that have multiple vertices with the same label, in any iteration, a vertex with that label in the background graph could match any of these vertices in the template, so each match must be verified independently. If v_j fails to satisfy the required local constraints for a template vertex $q_k \in \omega(v_j)$, then q_k is removed from $\omega(v_j)$. At any stage, if $\omega(v_j)$ becomes empty, then v_j is marked inactive ($\alpha(v_j) \leftarrow false$) and never communicate with its neighbors again. Edge elimination excludes two categories of edges: first, the edges to neighbors, $v_i \in \varepsilon(v_j)$ from which v_j did not receive a message of type *alive*, and, second, the edges to neighbors whose labels do not match the labels prescribed in the adjacency structure of the corresponding template vertex/vertices in $\omega(v_j)$. A vertex v_j is also marked inactive if its active edge list $\varepsilon(v_j)$ becomes empty. Iterations continue until no vertex or edge is marked inactive.

5.2 Non-local Constraint Checking

Non-local Constraint Checking iterates over $\mathcal{K}_0$, the set of non-local constraints to be checked, and validates each $C_0 \in \mathcal{K}_0$ one at a time. Algorithm 6 describes the solution to verify a single constraint: tokens are initiated through an asynchronous traversal by invoking the $do\_traversal()$ method and, as a result, each active vertex receives a visitor with $msg_{type} = init$. Each active vertex $v_j \in \mathcal{G}^*$ that is a potential match for the template vertex q_0 at the head of a walk (i.e., a non-local constraint) C_0, broadcasts a token to all its active neighbors in $\varepsilon(v_j)$ with $msg_{type} = forward$. A map γ is used to track these token issuers. A *token* is a tuple (t, r) where t is an ordered list of vertices that have forwarded the token and r is the hop counter; $t_0 \in t$ is the token-issuing vertex in $\mathcal{G}^*$. The ordered list t is essential for TDS, since it enables detection of distinct vertices with the same label in the token path. For simpler templates, such as templates with unique vertex labels and only edge-monocycles, t may only contain t_0 to keep the message size small.

When an active vertex v_j receives a token with $msg_{type} = forward$, it verifies that if $\omega(v_j)$ is a match for the next entry in C_0, if it has received the token from a valid neighbor (with respect to entries in C_0), and that the current hop count is less than $|C_0|$. If these requirements are satisfied (i.e., $\mu(v_j, C_0, token)$ returns $true$), then v_j sets itself as the forwarding vertex (v_j is added to t), increments the hop count, and broadcasts the token to all its active neighbors in $\varepsilon(v_j)$. If any of the constraints are not met, then v_j drops the token. If the hop count r is equal to $|C_0|$ and v_j is the same as the source vertex in the token, for a cyclic template, then a cycle has been found and v_j is marked $true$ in γ. For path constraints, an acknowledgement is sent to the token issuer to update its status in γ (Algorithm 7, lines #28–#31). Once verification of a constraint C_0 has been completed, the vertices that are not marked $true$ in γ, are invalidated/eliminated, i.e., $\alpha(v_j) \leftarrow false$ (Algorithm 6, line #9).

Our distributed implementation incorporates a number of design features aimed at improving performance, scalability, robustness and efficiency; we offer a light-weight yet highly effective technique, called *work aggregation*, to prevent relaying duplicate messages and the ability to load balance an intermediate pruned graph. In the remaining of the section, we first discuss these optimizations; we then provide details about how vertex metadata (labels) are managed, and various results our system can output.

ALGORITHM 6: Non-local Constraint Checking

```
1:  procedure NON_LOCAL_CONSTRAINT_CHECKING($\mathcal{G}$, $\mathcal{G}_0$, $C_0$)
2:      $\gamma \leftarrow$ map of token source vertices (in $\mathcal{G}$) for $C_0$; the value field (initialized to false) is set to true if the token source
          vertex meets the requirements of $C_0$
3:      $do\_traversal(msg_{type} \leftarrow init)$
4:      barrier
5:      for all $v_j \in \gamma$ do
6:          if $\gamma.get(v_j) \neq true$ then
7:              $\omega(v_j).remove(q_0)$ where $q_0$ is the first vertex in $C_0$      ▷ violates $C_0$, eliminate potential match
8:              if $\omega(v_j) = \emptyset$ then                                        ▷ no potential match left
9:                  $\alpha(v_j) \leftarrow false$                                       ▷ vertex eliminated
10:     $\forall v_j \in \mathcal{V}$, reset $\tau(v_j)$
```

5.3 Work Aggregation

All NLCC constraints attempt to identify if a walk exists from a vertex with a fixed label and through vertices with specific labels. Since the goal is to identify the existence of any such path and multiple intermediate/complete paths in the background graph often exist, to prevent combinatorial explosion, our duplicate work detection mechanism prevents an intermediary vertex (in the token path) from forwarding a duplicate token. NLCC uses an unordered set $\tau(v_j)$

ALGORITHM 7: Non-local Constraint Checking Visitor

```
1: visitor state: $v_j$ - vertex that is visited
2: visitor state: $token$ - the token is a tuple $(t, r)$ where $t$ is an ordered list of vertices that have forwarded the token
   and $r$ is the hop counter; $t_0 \in t$ is the vertex that originated the token
3: visitor state: $msg_{type}$ - $init$, $forward$ or $ack$
4: $\mu(v_j, C_0, token)$ - verifies if $v_j$ satisfies requirements of $C_0$ for the current state of $token$; returns $true$ if constraints
   are met, $false$ otherwise
5: procedure VISIT($\mathcal{G}$, $vq$)
6:     if $\alpha(v_j) = false$ thenreturn
7:     if $msg_{type} = init$ and $\exists q_k \in \omega(v_j)$ where $q_k = q_0 \in C_0$ then
8:         ▷ initiate a token; $v_j$ is the token source
9:         $t.add(v_j)$;   $r \leftarrow 1$;   $token \leftarrow (t, r)$;   $\gamma.insert(v_j, false)$
10:        for all $v_i \in \varepsilon(v_j)$ do
11:            $vis \leftarrow$ NLCC_VISITOR($v_i$, $token$, $forward$)
12:            $vq.push(vis)$
13:    else if $msg_{type} = forward$ then                                  ▷ $v_j$ received a token
14:        if $token \notin \tau(v_j)$ then                                  ▷ work aggregation optimization
15:            $\tau(v_j).insert(token)$
16:        elsereturn                                 ▷ ignore if $v_j$ already forwarded a copy of $token$
17:        if $\mu(v_j, C_0, token) = true$ and $token.r < |C_0|$ then
18:            ▷ the walk can be extended with $v_j$ and it has not reached the length $|C_0|$ yet
19:            $token.t.add(v_j)$;   $token.r \leftarrow token.r + 1$;
20:            for all $v_i \in \varepsilon(v_j)$ do                          ▷ forward the token
21:                $vis \leftarrow$NLCC_VISITOR($v_i$, $token$, $forward$)
22:                $vq.push(vis)$
23:        else if $\mu(v_j, C_0, token) = true$ and $token.r = |C_0|$ then
24:            ▷ the walk has reached the length $|C_0|$
25:            if $C_0$ is cyclic and $t_0 = v_j$ then
26:                $\gamma.get(v_j) \leftarrow true$ return                    ▷ $v_j$ meets requirements of $C_0$
27:            else if $C_0$ is acyclic and $t_0 \neq v_j$ then
28:                $vis \leftarrow$ NLCC_VISITOR($t_0$, $token$, $ack$)
29:                $vq.push(vis)$                       ▷ send $ack$ to the token originator, $t_0 \in t$
30:    else if $msg_{type} = ack$ then
31:        $\gamma.get(v_j) \leftarrow true$return                              ▷ $v_j$ meets requirements of $C_0$
```

(Algorithm 3, line #4) for work aggregation (see Algorithm 7, line #14): at each vertex, this is used to detect if another copy of a *token* has already visited the vertex v_j taking a different path. The performance impact of this optimization is evaluated in Section 7.5.

5.4 Load Balancing

Load imbalance issues are inherent to problems involving irregular data structures, such as graphs, especially when these need to be partitioned for processing over multiple nodes. For our pattern matching solution, load imbalance can be further caused by two artifacts: First, over the course of execution our solution causes the workload to mutate, i.e., we prune away vertices and edges. Second, the distribution of matches in the background graph may be nonuniform: the vertices and edges that participate in matches, may reside on a small, potentially concentrated, part of the graph. (In Section 7.6, we present a detailed characterization of these artifacts.)

The iterative nature of the constraint checking pipeline allows us to adopt a *pseudo-dynamic* load balancing approach: First, we checkpoint the current state of execution (at the end of an asynchronous constraint checking phase): the pruned graph, i.e., the set of active vertices and edges and the per-vertex state indicating template matches, $\omega(v_j)$ (Algorithm 3). Next, using HavoqGT's distributed graph partitioning module, we reshuffle the vertex-to-processor assignment to evenly distribute vertices (with $\omega(v_j)$ remained intact) and edges across processing cores. Processing is then resumed on the rebalanced workload. Furthermore, depending on the size the the pruned graph, it is possible to resume processing on a smaller deployment (primarily for efficiency reasons, such as conserving CPU Hours). Over the course of the execution, checkpointing and rebalancing can be repeated as needed. We evaluate the effectiveness of different load balancing strategies and present an analysis of their impact on performance in Section 7.6.

5.5 Termination and Output

If NLCC is not required, then the search terminates when no vertex is eliminated (or none of its provisional matches is removed) in an LCC iteration. Otherwise, the search terminates when all constraints in $\mathcal{K}_0$ have been verified. The output of constraint checking is (i) the set of vertices and edges that survived the iterative elimination process and, (ii) for each vertex in this set, the mapping in the template where a match has been identified.

A distributed match enumeration or counting routine can operate on the pruned solution subgraph: Algorithm 7 can be slightly modified to obtain the enumeration of the matches in the background graph; here, the constraint used is a walk on the full template, work aggregation is turned off, and each possible match is verified. For each of the vertices that remains in the solution set, the pruning procedure collects their exact match(es) to the search template. We use this information to accelerate match enumeration.

5.6 Metadata Store

The metadata are stored independent of the graph topology itself that uses the Compressed Sparse Row (CSR) format [Bell and Garland 2009]. At initialization, only the required attributes are read from the file(s) stored on a distributed file system. A light-weight distributed process builds the in-memory (or memory-mapped) metadata store. For example, on 256 compute nodes, for the 257 billion edge Web Data Commons graph [Robert Meusel 2016], the metadata store can be populated in under a minute. Although, in this work, we consider vertex metadata (i.e., labels) only, support for edge metadata is trivial within the presented infrastructure.

6 COMPLEXITY ANALYSIS

We attempt to estimate the space, time and generated message complexity for both LCC and NLCC routines presented in Section 5. Note that except for the first iteration of LCC, constraint checking routines are invoked on the current (pruned) solution subgraph $\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$, where $|\mathcal{G}^*| \leq |\mathcal{G}|$. (See Table 2 for the symbolic notation used in this section.)

6.1 Local Constraint Checking

We mainly focus on analyzing the complexity of one iteration of the LCC routine presented in Algorithm 4.

Space Complexity. In each iteration of LCC, each active vertex $v_i \in \mathcal{V}^*$ maintains a set of its template vertex matches/exclusions $\omega(v_i)$, where $|\omega(v_i)| = |\mathcal{V}_0|$. Therefore, space complexity of LCC is linear in the size of the template: $O(|\mathcal{V}^*| \times |\mathcal{V}_0|)$. In our implementation, we use a bit vector to store the template vertex matches to reduce memory overhead. For example, if the template has 64 vertices, then per-vertex (of $\mathcal{G}^*$) storage requirement is eight bytes. Additionally, in one iteration of LCC, an active vertex creates one visitor per active edge, therefore, the storage requirement for the visitor queue (the message queue in HavoqGT) is $O(|\mathcal{E}^*|)$.

Time Complexity. In each iteration of LCC, all active vertices in $\mathcal{V}^*$ visit all their respective active neighbors (in $\mathcal{E}^*$). In iteration k, only the vertices and edges that survived iteration $k-1$, are considered. Therefore, the time complexity of the kth iteration is $O(|\mathcal{V}^*_{k-1}| + |\mathcal{E}^*_{k-1}|)$. Initially, i.e., when $k = 0$ and no vertices and edges have been eliminated, i.e., $\mathcal{V}^* = \mathcal{V}$ and $\mathcal{E}^* = \mathcal{E}$; we can write time complexity of the first iteration is $O(|\mathcal{V}^*| + |\mathcal{E}^*|)$, the most expensive of all LCC iterations. Assume LCC stops eliminating vertices and edges after k_{max} iterations; hence, total time complexity of LCC is $O(k_{max} \times (|\mathcal{V}^*| + |\mathcal{E}^*|))$. For an acyclic template with unique labels, $k_{max} = \text{diam}\,(\mathcal{G}_0) + 1$ (see Reza et al. [2017] for proof). An analysis for the worst case for an arbitrary template does not take us far—the upper bound of maximum number of iteration in LCC is $k_{max} \leq |\mathcal{E}|$. In practice, the worst case is when in each iteration only a few or no vertices and/or edges are eliminated and a large number of iterations is needed. However, for real-world, scale-free graphs, the first few steps of LCC reduce $|\mathcal{G}|$ by several orders of magnitude, yielding costs nowhere near the worst-case bounds (see the evaluation section (Section 7) for multiple examples).

Message Complexity. In each iteration, an active vertex creates one visitor per active edge, resulting in one message per edge. The analysis is similar to the one above: the message complexity of one iteration of LCC is $O(|\mathcal{E}^*|)$.

6.2 Non-local Constraint Checking

We study the complexity of the NLCC routine for checking a single constraint $C_0 \in \mathcal{K}_0$, presented in Algorithm 6. Note that for a cyclic constraint, a token must be initiated from each vertex in the background graph that may participate in the substructure representing C_0, i.e., in Algorithm 6, each vertex in $\mathcal{G}^*$, that match at least one vertex in C_0, initiates a token.

Space Complexity. The NLCC routine requires two additional algorithm states: (i) γ, the map of token source vertices (in $\mathcal{G}^*$) for C_0, requires at most $O(|\mathcal{V}^*|)$ storage, and (ii) $\tau(v_j)$, the set of already forwarded tokens by a vertex v_j used for work aggregation: If C_0 is edge-monocyclic and has unique vertex labels, then the per-vertex storage requirement for $\tau(v_j)$ is no more than $O(|\gamma|)$ or total $O(|\mathcal{V}^*| \times |\gamma|)$ for $\mathcal{G}^*$. For arbitrary templates, however, the cost is superpolynomial and proportional to the message complexity discussed later. Similarly, the worst-case storage requirement for the visitor queue is also superpolynomial (and directly related to the generated message traffic).

Time Complexity. In NLCC, each constraint $C_0 \in \mathcal{K}_0$ is verified by passing around tokens. Each active vertex in $\mathcal{V}^* \in \mathcal{G}^*$ that could be a template match for the first vertex in C_0, issues a token—identified by an entry in γ where $|\gamma| \leq |\mathcal{V}^*|$. In the distributed message passing setting, token passing happens in a breadth-first search manner (on shared memory, a more work-efficient depth-first search like implementation is possible). The effort related to token propagation is bounded by $|\gamma|$—the number of tokens, average degree connectivity, and the depth of the propagation (i.e., the size of the constraint $|C_0|$). For an arbitrary constraint C_0, the cost is exponential: Assume r indicates a step in the walk represented by C_0; at $r = 1$, in the worst case, a token is received by at most $(|\mathcal{V}^*| - 1)$ vertices, and at $r = 2$, each of these vertices forward the same token to at most $(|\mathcal{V}^*| - 2)$ vertices. To propagate $|\gamma|$ token, this results in visiting $|\gamma| \times (|\mathcal{V}^*| - 1) \times (|\mathcal{V}^*| - 2) \times \cdots \times (|\mathcal{V}^*| - r - 1)$ vertices, where $r = |C_0|$). Since $|\gamma| \leq |\mathcal{V}^*|$, we can write the sequential cost of verifying constraint C_0 is $O(|\mathcal{V}^*|^{|C_0|})$.

Message Complexity. As discussed above, in NLCC, each vertex visitation by (a copy of) a token results in one message. Therefore, the message complexity of checking a non-local constraint C_0 is $O(|\mathcal{V}^*|^{|C_0|})$. Heuristics like work aggregation, however, prevents a vertex from forwarding duplicate copies of a token, which reduces the time and message propagation effort in practice.

6.3 Motivating the Expected Gains from the Complexity Perspective

The previous section presents the time, space, and message complexity of the local and non-local constraint checking algorithms. Here, we attempt to give an intuition for the expected performance gains compared to the traditional direct enumeration approach [Ullmann 1976]. Direct enumeration has $O(|\mathcal{V}|^{|\mathcal{G}_0|})$ complexity in the general case [Ullmann 1976]. In our approach, the non-local constraint checking routines are the high-complexity routines: $O(|\mathcal{V}^*|^{|C_0|})$. These routines operate on the current solution subgraph graph $\mathcal{G}^*(\mathcal{V}^*, \mathcal{E}^*)$ after it has already been pruned by local constraint , and is generally expected to be significantly smaller than the original background graph, i.e., $|\mathcal{V}^*| \leq |\mathcal{V}|$ (we explore this in Reza et al. [2018]; note that we eliminate both vertices and edges). Also, $|C_0| \leq |\mathcal{G}_0|$ and, as we check constraints in the increasing order of their length, constraints (substructures of the search template) that require a longer walk, operate on the smaller pruned graph available in the later stages of processing. Finally, compared to direct enumeration, our constraint checking-based approach typically generates smaller algorithm state, thus limiting combinatorial explosion; and, at the same time, the work aggregation heuristic prevents a vertex from forwarding duplicate copies of a token, which reduces the generated network traffic (see Section 7.5). In the same vein, in our approach, match enumeration is performed on the pruned solution subgraph, hence, the complexity is $O(|\mathcal{V}^*|^{|\mathcal{G}_0|})$.

7 EVALUATION

This section is structured as follows: To demonstrate the ability of our system to process massive graphs on large deployments, we present *strong scaling* experiments on the largest real-world graph publicly available (Section 7.4). We evaluate the effectiveness of key design decisions, optimizations, and load balancing techniques our system incorporates (Sections 7.5 and 7.6). We demonstrate the versatility of our constraint checking approach and use it as a stepping stone to efficiently support additional usage scenarios, namely, *interactive incremental search* and *exploratory search* (Section 7.7). We compare our solution with three state-of-the-art exact pattern matching systems, Arabesque [Teixeira et al. 2015], QFrag [Serafini et al. 2017], and TriAD [Gurajada et al. 2014] (Section 7.8). Furthermore, we study how search template characteristics impact search performance and (Section 7.9) and demonstrate application to graphs with various vertex degree distributions (Section 7.10).

Our previous work [Reza et al. 2018], includes additional experimental results: *weak scaling* experiments on massive synthetic *R-MAT* graphs with up to ~4.4 trillion edges and using up to 1,024 compute nodes (36,864 cores) (Reza et al. [2018], Section 5A); demonstrates the ability to support *full match enumeration*, starting from the pruned solution subgraph (Reza et al. [2018], Section 5A) on these massive datasets; evaluation of various design decisions (Reza et al. [2018], Section 5F); shows support for realistic data analytics scenarios using two real-world graphs, *Reddit* and *IMDb* (Reza et al. [2018], Section 5D); and an exploration of time-to-solution vs. precision guarantees tradeoffs (Reza et al. [2018], Section 5E). Finally, Tripoul et al. [2018] explores advanced heuristics for constraint selection and ordering; and Reza et al. [2017] focuses on a restricted set of search templates, acyclic or edge-monocyclic without duplicate labels, that can be supported extremely efficiently.

7.1 Testbed

The testbed is the 2.6-petaflop Quartz cluster at the Lawrence Livermore National Laboratory, composed of 2,634 nodes and the Intel Omni-Path interconnect. Each node has two 18-core Intel Xeon E5-2695v4 @2.10 GHz processors and 128 GB of main memory [Quartz 2017]. We run one MPI process per core (i.e., 36 processes per node).

Table 4. Properties of the Datasets Used for Evaluation: Number of Vertices and Edges, Maximum, Average and Standard Deviation of Vertex Degree, and the Graph Dataset Size, in the Compact CSR-like Representation Used, which Includes the Vertex Metadata

	Type	$\lvert\mathcal{V}\rvert$	$2\lvert\mathcal{E}\rvert$	d_{max}	d_{avg}	d_{stdev}	Size
Web Data Commons [Robert Meusel 2016]	Real	3.5B	257B	95M	72.3	3.6K	2.7 TB
Reddit [Reddit 2017]	Real	3.9B	14B	19M	3.7	483.3	460 GB
Internet Movie Database [IMDb 2016]	Real	5M	29M	552K	5.8	342.6	581 MB
CiteSeer [Teixeira et al. 2015]	Real	3.3K	9.4K	99	3.6	3.4	741 KB
Mico [Teixeira et al. 2015]	Real	100K	2.2M	1.4K	22	37.1	36 MB
Patent [Serafini et al. 2017]	Real	2.7M	28M	789	10.2	10.8	480 MB
YouTube [Serafini et al. 2017]	Real	4.6M	88M	2.5K	19.2	21.7	1.4 GB
LiveJournal [Backstrom et al. 2006]	Real	4.8M	69M	20K	17	36	1.2 GB
Twitter [Kwak et al. 2010]	Real	41.7M	2.9B	3M	47.7	2.1K	47 GB
UK Web [Boldi et al. 2011; Boldi and Vigna 2004]	Real	105.9M	7.5B	975K	70.6	718	119 GB
Road USA [Rossi and Ahmed 2015]	Real	23.9M	58M	9	2.4	0.9	1.4 GB
R-MAT up to Scale 37 [Chakrabarti et al. 2004]	Synthetic	137B	4.4T	612M	32	4.9K	45 TB

7.2 Datasets

Table 4 summarizes the main characteristics of the datasets used in this work. We briefly explain below how the background graphs and their labels are created. Additional details can be found in Reza et al. [2018] and Tripoul et al. [2018]. For all graphs, we created undirected versions - two directed edges are used to represent each undirected edge.

The Web Data Commons (WDC) graph is a webgraph whose vertices are webpages and edges are hyperlinks. To create vertex labels, we extract the top-level domain names from the webpage URLs, e.g., *.org* or *.edu*. If the URL contains a common second-level domain name, then it is chosen over the top-level domain name. For example, from *ox.ac.uk*, we select *.ac* as the vertex label. A total of 2,903 unique labels are distributed among the 3.5B vertices in the background graph.

We curated the Reddit (RDT) social media graph from an open archive [Reddit 2017] of billions of public posts and comments from Reddit.com. Reddit allows its users to rate (upvote or downvote) others' posts and comments. The graph has four types of vertices: *Author, Post, Comment*, and *Subreddit* (a category for posts). For *Post*- and *Comment*-type vertices there are three possible labels: *Positive, Negative,* and *Neutral* (indicating the overall balance of positive and negative votes) or *No* rating. An edge is possible between an *Author* and a *Post*, an *Author* and a *Comment*, a *Subreddit* and a *Post*, a *Post* and a *Comment* (to that *Post*), and between two *Comments* that have a parent–child relation.

We use the smaller *Patent* and *YouTube* graphs for comparison with existing exact pattern matching systems, QFrag [Serafini et al. 2017] and TriAD [Gurajada et al. 2014]. The *Patent* graph has 37 unique vertex labels, while the *YouTube* graph has 108 unique vertex labels. We use *CiteSeer*, *Mico, Patent, YouTube*, and *LiveJournal* unlabeled, real-world graphs for performance comparison with Arabesque [Teixeira et al. 2015]. Additionally, we use two large (billions of edges) real-world, scale-free graphs, *Twitter* and *UK Web*, used in the past by many for studying various graph analysis problems; and a large diameter, real-world, road network graph, *Road USA*.

The synthetic Recursive MATrix (R-MAT) graphs exhibit approximate power-law degree distribution [Chakrabarti et al. 2004]. These graphs were created following the Graph 500 [Graph 500 2016] standards: 2^{Scale} vertices and a directed edge factor of 16. For example, a Scale 30 graph has $\lvert\mathcal{V}\rvert = 2^{30}$ and $\lvert\mathcal{E}\rvert \approx 32 \times 2^{30}$ (as we create an undirected version). Since we use the *R-MAT* graphs

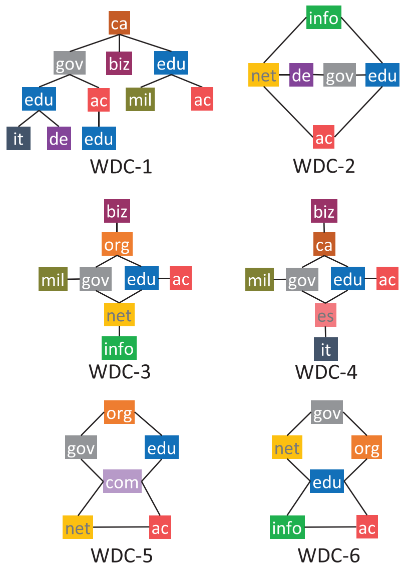


Fig. 4. *WDC* patterns using top/second-level domain names as labels. The labels selected are among the most frequent, covering ~81% of the vertices in the *WDC* graph: Unsurprisingly, *com* is the most frequent (covering over two billion vertices), *org* covers ~220M vertices; the second most frequent after *com* and *mil* is the least frequent among these labels, covering ~153K vertices.

for weak scaling experiments, we aim to generate labels such that the graph structure changes little as the graph scales. To this end, we leverage vertex degree information to create vertex labels, computed using the formula, $\ell(v_i) = \lceil \log_2(d(v_i) + 1) \rceil$. This, for instance for the Scale 37 graph, results in 30 unique vertex labels.

Notes on Data Storage and Loading. Our testbed is served by a distributed storage platform running the Lustre parallel file system [Lustre 2016]. To accelerate graph loading, HavoqGT can preprocess the adjacency lists to take advantage of the existing parallel file system: It splits each input dataset in the same number of parts/files as the MPI processes used in the respective experiment. HavoqGT's graph partitioning process also attempts to create balanced partitions by assigning an equal share of edges to each partition and, where necessary, splits the edge set of a high-degree vertex over multiple partitions. This however, can be a costly process for massive graphs: for example, for the *WDC* graph, graph partitioning for 128 nodes (4,608 partitions) takes about six hours. This distributed graph can then be loaded from the parallel file system in under two minutes. The vertex metadata, is stored (split in multiple parts/files) independently of the graph topology and can be loaded from the distributed file system relatively fast without preprocessing: for example, for the *WDC* graph, in about 30 s.

7.3 Search Templates and Experiment Design

To stress our system, we use templates based on patterns naturally occurring, and relatively frequent, in the background graphs. The *WDC* (Figure 4), *Twitter*, *UK Web*, *Patent*, *YouTube* and *R-MAT* patterns include vertex labels that are among the most frequent in the respective graphs. The *Reddit* and *IMDb* patterns include most of the vertex labels in these two graphs [Reza et al. 2018]. We chose templates to exercise different constraint checking scenarios: the search templates have

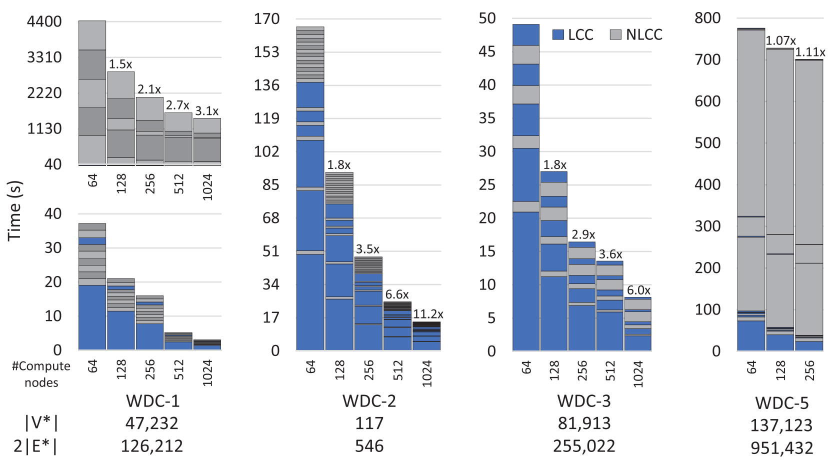


Fig. 5. Runtime for strong scaling experiments, broken down into individual (LCC and NLCC) phases, for four of the patterns in Figure 4. (For better visibility, for WDC-1, runtime for different iterations are split into two scales on the Y axis.) The last two rows are the number of vertices and edges in the pruned solution subgraph, respectively. Speedup over the 64 node configuration is also shown on top of each stacked bar plot. (Partially reused from Reza et al. [2018] with additional results.)

multiple vertices with the same label and non-edge-monocyclic properties (they require relatively expensive non-local constraint checking).

All runtime numbers provided are averages over 10 runs. Unless mentioned explicitly, the performance metric is the time to produce the solution subgraph for a single template.

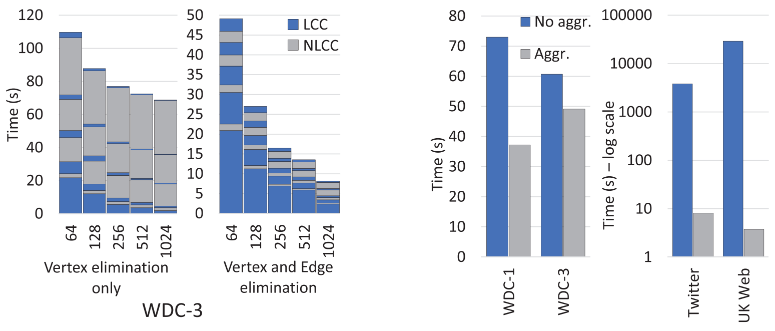
7.4 Strong Scaling Experiments

The strong scaling experiments evaluate the performance of pruning (i.e., we verify all the constraints required to guarantee zero false positives). The smallest experiment uses 64 nodes, as this is the lowest number of nodes that can load the graph topology and vertex metadata in memory. Figure 5 shows runtimes for strong scaling experiments when using the real-world *WDC* graph on up to 1,024 nodes (36,864 cores). Intuitively, pattern matching on the *WDC* graph is harder than on the *R-MAT* graph as the *WDC* graph is denser, has a highly skewed degree distribution, and the high-frequency labels used also belong to vertices with high neighbor degree.

We use the patterns presented in Figure 4. WDC-1 is acyclic, yet has multiple vertices with the same label and thus requires non-local constraint checking (PC and TDS). For better visibility, the plot splits checking initial LCC and NLCC-path constraints (bottom left) from NLCC-TDS constraints (top left). We notice near perfect scaling for the LCC phases, however, some of the NLCC phases do not show linear scaling (explained in Section 7.6).

WDC-2 is an example of a pattern with multiple cycles sharing edges, and relies on CC and TDS constraint checking. WDC-2 shows near-linear scaling with ~1/3 of the total time spent in the first LCC phase and little time spent in the NLCC phases. WDC-3 is a monocyclic template and, when edge elimination is used (bottom right), shows steady scaling for both LCC and NLCC phases.

The WDC-5 pattern includes the top three most frequent labels, namely, *com*, *org*, and *net*, and covers ~72% vertices in the *WDC* graph. Similar to WDC-1, a majority of the time is spent verifying the non-local constraints. The NLCC phases do not scale well with increasing node count for two interrelated reasons: first, vertices participating in matches have high neighbor degree, and second,



(a) (b)

Fig. 6. (a) Performance and scalability comparison between the vertex elimination only (left), and combined vertex and edge elimination (right), for the WDC-3 pattern. (b) Impact of work aggregation on runtime using three real-world graphs: WDC-1 and WDC-3 patterns (for the sake of readability, only a subset of non-local constraints are considered for WDC-1), and the Q8 pattern (Figure 12) using the *Twitter* and *UK Web* graphs. (Partially reused from Reza et al. [2018] with additional results.)

and more importantly, heavily skewed template match distribution among the graph partitions, (further explored in Section 7.6).

7.5 Impact of Major Design Decisions and Optimizations

Here, we present the impact of two major design decisions and optimizations: (i) edge elimination and (ii) work aggregation. (In Reza et al. [2018] and Tripoul et al. [2018], we have studied the impact of additional design features on search performance.)

Edge Elimination. Figure 6(a) highlights the important scalability and performance impact of edge elimination: Without it, the NLCC phases take almost one order of magnitude longer and the entire pruning takes 2–9× longer. Without edge elimination, the WDC-3 pattern results in 3,180,678 edges selected (it includes false positives). Edge elimination identifies the true-positive matches and reduces the number of active edges to 255,022. In other words, the solution subgraph is 12.5× sparser that in turn improves overall message efficiency of the system. We note that this one order of magnitude reduction enables match enumeration and advanced analytics on the solution subgraph.

Work Aggregation. Figure 6(b) shows the performance gains enabled by the work aggregation strategy employed by the distributed non-local constraint checking routine (presented in Section 5 and Algorithm 7). We study the impact of work aggregation for three large real-world graphs: *WDC*, *Twitter*, and *UK Web*. The magnitude of the gain is data dependent and more pronounced when the pattern is abundant, e.g., 50% improvement for WDC-1 that has 600M+ matches in the background graph. The experiments using the *Twitter* and *UK Web* graphs further highlight the advantage of work aggregation: We compare the runtime of a single non-local constraint (a TDS constraint involving all the vertices and edges in the template) for the search pattern Q8 (Figure 12). (The experiment details are available in Section 7.9.) For the *Twitter* and *UK Web* graphs, the gain in runtime are two and three orders of magnitude, respectively (Figure 6(b), right chart). Unlike full match enumeration, NLCC does not need to identify all possible walks for each token; the goal is to identify the existence of any such walk (a complete path) in the background graph - sufficient to save the vertex that initiated the token from elimination. The significant improvement in runntime is due to reduction in number of complete paths traversed by all the tokens created; the number of messages communicated in non-local constraint checking is proportional to the number of paths traversed. For the *UK Web* graph, for 24,000 unique tokens, without work aggregation, 45 billion unique paths are discovered. Our work aggregation technique reduces the number of complete paths traversed to 71 million, a four orders of magnitude reduction; hence, the three orders of magnitude gain in runtime (Figure 6(b), right chart). Similarly, for the Twitter graph, the reduction in the number of complete paths traversed is three orders of magnitude. (In Figure 6(b), all experiments were run on 64 compute nodes.)

7.6 Load Balancing

For our pattern matching solution, load imbalance can be caused by two artifacts: First, over the course of execution our solution causes the workload to mutate, as it prunes away vertices and edges. Second, the distribution of matches in the background graph may be nonuniform: Matches may reside on a small, potentially concentrated, portion of the graph. This section, first presents a detailed characterization of these artifacts, then it discusses and evaluates two load balancing strategies.

Does Load Imbalance Occur? Indeed, load imbalance does occur. For instance, for the relatively rare WDC-2 (Figure 4) pattern, when using 64 nodes, for example, the vertices and edges that participate in the final selection are distributed over as few as 111 partitions out of the 2,304 (64 nodes $\times$ 36 MPI processes per node). The distribution is concentrated—more than half of the matching edges reside on only 20 partitions. For the more frequent WDC-1 pattern, 50% of the matching edges are on less than 5% of the partitions on a 64 node deployment, and less than 3% of the partitions on a 128 node deployment.

We observe further nonuniformity in the match distribution at the vertex granularity; the number of matches a vertex participates in, can significantly vary across the matching vertex set $\mathcal{V}^*$. As an example, let us consider the WDC-2 pattern (whose matches are shown in Reza et al. [2018], Figure 10; they form six connected components). The largest connected component contains 2,262 matches (bottom row, center). In this connected component, there is a single *gov* vertex, which participates in 2,262 matches (of a total of 2,444 matches). This artifact is more pronounced in the case of the WDC-1 and WDC-2 patterns. For WDC-1, 99% of the matching vertices are part of a single connected component. There are multiple vertices that belong to over three million matches. The numbers are more striking for the frequent WDC-3 pattern—a single vertex participates in over 34 million matches.

This irregularity has crucial performance implications, in particular, it hinders the scalability of the routines that rely on *multi-hop graph walks*, such as non-local constraint checking and full match enumeration. When the matches are concentrated on a few compute nodes and only a few vertices participate in a large number of matches, the partitions these vertices reside on send/receive a larger portion of the message traffic. In this case, increasing the number of processors does not help as, in our current infrastructure, processing at the vertex granularity can not be "scaled out" efficiently. Furthermore, since each partition processes the local message queue sequentially, message traffic targeting popular vertices can overwhelm the respective partitions. Consequently, these bottlenecked partitions become the key performance limiter. This reasoning explains why some of the non-local constraint checking phases do not scale well (e.g., Figure 5).

Strategies to Address Load Imbalance Issues. We explore two strategies to address load-balancing issues: (i) reshuffling the load and (ii) load consolidation, i.e., reloading the shuffled load on fewer nodes to also optimize for locality and reduce generated network traffic.

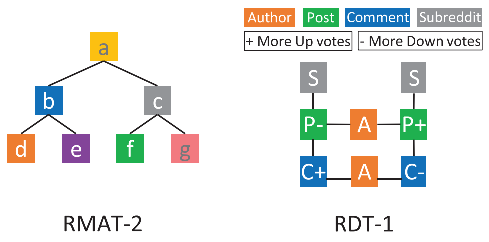


Fig. 7. RMAT-2 is the template used for *R-MAT* experiments (left); it includes the most frequent vertex labels in the background graph (as in Figure 12). The RDT-1 (*Reddit*) pattern is used for the load balancing experiments in Section 7.6 (right, reused from Reza et al. [2018]).

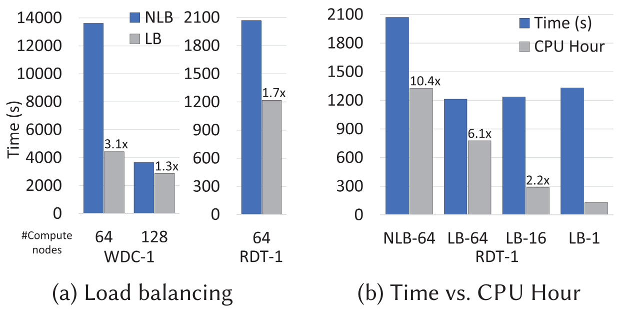


Fig. 8. (a) Impact of load balancing on runtime using differnt number of compute nodes for the the WDC-1 and RDT-1 patterns. Speedup achieved by LB (load balancing) over NLB (without load balancing) is also shown on the top of each bar. (b) Performance of RDT-1 for four scenarios: (left) without load balancing on 64 nodes (NLB-64), (center-left) with load balancing on the same number of nodes (LB-64), (center-right) relaunching on a 16 node deployment after load balancing (LB-16) and (right) relaunching on a single node (36 processes) after load balancing (LB-1). Time-to-solution and CPU Hours consumed (normalized to the LB-1 experiment) are numerically presented on the top of the respective bars.

Load Reshuffling. We employ a *pseudo-dynamic*, load balancing strategy. First, we checkpoint the current state of execution: the pruned graph, i.e., the set of active vertices and edges and the per-vertex state indicating template matches, $\omega(v_j)$ (Algorithm 3). Next, using HavoqGT's graph partitioning module, we reshuffle the vertex-to-processor assignment to evenly distribute vertices (with $\omega(v_j)$ remained intact) and edges across processing cores. Processing is then resumed on the rebalanced workload. Depending on the size the the pruned graph, it is possible to resume processing on a smaller deployment (discussed in the next section). Over the course of the execution, checkpointing and rebalancing can be repeated as needed (the identification of the optimal trigger point to perform load balancing, however, requires further investigation).

To examine the impact of this technique, we analyze the runs for WDC-1 (Figure 4) and RDT-1 (Figure 7) patterns, as real-world workloads are more likely to lead to imbalance. Figure 8(a) compares performance with and without load balancing. For these experiments, we perform rebalancing only once. For WDC-1, before verifying the TDS constraints, and for RDT-1, when the pruned graph is four orders of magnitude smaller. The extent of load imbalance is more severe for WDC-1 on the smaller 64 node deployment compared to using 128 nodes—workload rebalancing improves time-to-solution by 3.1× and 1.3×, on 64 and 128 nodes, respectively. In the case of RDT-1, load balancing improves time-to-solution by 1.7×. Given the pruned graphs are much smaller than the original graph; often the time spent in checkpointing, rebalancing, and relaunching the computation is negligible compared to the gain in time-to-solution.

Smaller Deployment. One may argue that when the current solution subgraph $\mathcal{G}^*$ is sufficiently small, it is more efficient to create load balanced partitions targeting a smaller deployment. Two different aspects of "efficiency" concerns support this approach: First, moving to a smaller deployment reduces power usage and may yield better normalized performance with respect to energy consumption. Second, for the scenario where the matches are highly concentrated on a limited number of nodes/partitions (which hinders the scalability of the non-local constraint checking phase), a smaller deployment offers locality (through reduced number of generated network traffic).

We setup a simple case study using the *Reddit* dataset and the RDT-1 pattern. We resume processing on the rebalanced workload on a smaller deployment - from the original 64 node deployment, we switch to a 4× smaller deployment comprised of 16 nodes. In a second use case, we resume processing on the rebalanced workload a on a single node (running 36 processes). Figure 8(b) compares four scenarios: (i) without load balancing (NLB-64), (ii) with load balancing (LB-64), (iii) with load balancing and relaunching on a smaller 16 node deployment (LB-16), and (iv) relaunching on a single node after load balancing (LB-1). In addition to time-to-solution, we also compare CPU Hours consumed by each of the four cases. (A platform's net energy consumption can be roughly approximated by the total CPU Hours expended.) Figure 8(b) shows that, with respect to time-to-solution, LB-64 has marginal advantage over LB-16 and LB-1. However, LB-1 holds significant advantage in terms CPU Hour consumption: It is 6.1× more efficient than LB-64. The overhead for NLB-64 is 10.4× compared to LB-1. These results support the argument that the load balanced partitions targeting a smaller deployment yields better normalized performance with respect to CPU Hour and energy consumption.

7.7 Advanced Usage Scenarios

This section highlights that our approach, based on constraint checking, can be extended to support a number of advanced pattern matching scenarios.

Interactive Incremental Search. In this scenario, the user starts with an under-constrained search template (possibly returning too many matches), and the system is setup for interactive use: the user can add/delete edges, observe the changes in the solution subgraph (or statistic over it), and continue to interact with the system. The only restriction we place on the user is that (s)he can remove only edges (not vertices) and has to maintain the search template is connected.

We take advantage of two observations: (i) For template revision through edge addition, adding an edge is similar to adding a constraint, and the search for the revised template can be limited within the vertex set of the current solution subgraph presented to the user. (ii) For template revision through edge deletion, we observe that, one can build a solution superset that is the union of all matches for all possible search templates that may be obtained from the initial template by removing just edges, using local constraints only, thus at a low cost. We use this restricted solution superset, which we refer to as the *candidate set*, to initially prune the backgtound graph, and to reinitiate the solution subgraph when an edge (from the current search template) is deleted. Furthermore, for the same vertex in the background graph, non-local constraints can be verified only once and this information can be reused in later searches (i.e., for revised templates); eliminating a large amount of potentially redundant work. We call this technique *work reuse.* (Design details are available in Reza et al. [2020b].)

For a preliminary evaluation of the effectiveness of these techniques we consider the search scenario presented in Figure 9 and explained in detail in Figure 10. We demonstrate the advantage of the optimized technique over a naive̎ approach that uses the exact matching solution to independently search the original query and each of its revisions. The experiment scenario, labelled PJI-X,

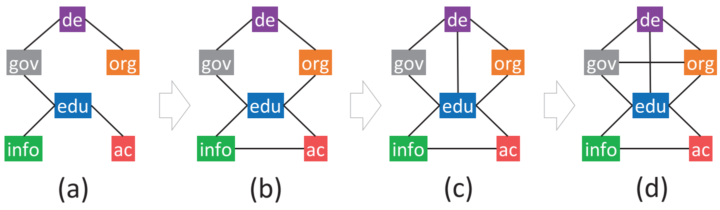


Fig. 9. An example showing the queries incrementally searched in the *WDC* graph. The user begins with the leftmost pattern (a) and incrementally revises the query by adding edges; gradually moves from left to right.

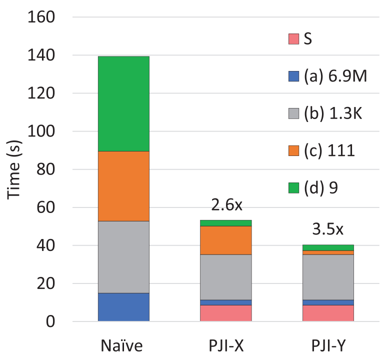
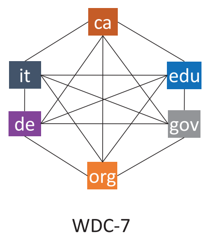


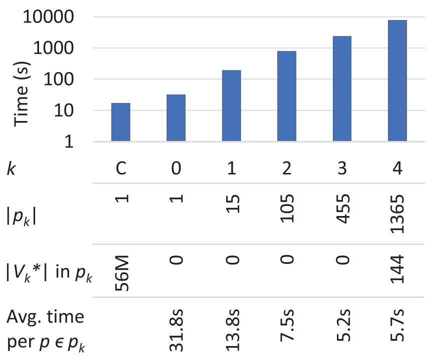
Fig. 10. Runtime comparison between the naïve and the optimized incremental search solution for the scenario in Figure 9. For each experiment, the stacked bar plot shows the time spent in each query. For the optimized incremental search solution (labeled PJI), we consider two setups: (i) PJI-X—we build the candidate set, directly search the initial query in it, then each of the remaining searches is limited within the vertex set of the solution subgraph of the previous search; (ii) PJI-Y—in addition to PJI-X, we also employ the work reuse technique. For PJI, speedup achieved over the naïve approach is shown on top of respective bar plots. The chart legend also shows the number of vertices that match respective queries. The fraction of the runtime labeled "S" is the overhead for building the candidate set. (Partially reused from Reza et al. [2020b].)

highlights the advantage building and restricting the searches to the candidate set: the solution first computes the candidate set, and then each template revisions is searched starting from this set; this version of the optimized pipeline offers 2.6× speedup. The experiment scenario PJI-Y, in addition to computing the candidate set at the start of the experiment, also employs the work reuse technique, to eliminate redundant non-local constraint verification. This yields a further 3.5× gain in time-to-solution over the naïve approach. We run these experiments on 128 compute nodes (4,608 cores).

Exploratory Search. We present an exploratory search scenario where the user starts from an over-constrained search template and the system progressively relaxes the template by removing edges until matches are found. The search progresses as follows: First, all variations of the initial search template with one edge removed are searched; then all variations with two edges removed are searches, and so on; until matches for at least one pattern are found. The system returns a subgraph that is the union of all matches at the first level where matches are found. As in the case of interactive incremental search described earlier, here, the key enabler is to identify the candidate set and use this reduced set in the later iterations of the search, as well as reuse the result of non-local constraint checking.

Figure 11(b) shows the runtime (when using 128 nodes), broken down to each level, for such a search, in the *WDC* graph, starting from an undirected 6-Clique (WDC-7 pattern in Figure 11(a)).





(a) (b)

Fig. 11. (a) The *WDC* pattern used for demonstration of exploratory search in Section 7.7. (b) Runtime for WDC-7, grouped by multiple levels depending on the number of edges removed from the initial search template. Note that no match is found until $k = 4$ edges are removed. X-axis labels: (First row) k is the number of edges removed. (Second row) $|p_k|$ is the number of distinct patterns that exist at level k; (third row) $|V_k^*|$ is the number of vertices that participate in any match of a pattern $p \in p_k$; (bottom row) average search time per pattern at each k. X-axis label "C" represents the initial candidate set generation. Note that the Y axis is in log scale.

For this search template, the first matches are found only after four edges are removed and involves sifting through over 1,900 variations of the original search template—only 144 vertices participate in these matches. Note that reducing the search space to the candidate set and the high efficiency of the exact matching pipeline (on average, it takes less than 6 seconds to explore each variant of the template), enables this type of exhaustive search.

7.8 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 an MPI-based solution, TriAD.[1] Similarly to our system, all these three systems offer exact matching with 100% precison and 100% 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, but we note that our system performs better or as well as the other systems.

We run these experiments using real-world graph datasets, on a large shared memory platform: the machine is equipped with four Intel Xeon E7-4870v2 @2.30 GHz CPU-sockets—a total of 60 CPU cores and 120 MB L3 memory, and 1.5 TB main memory. For QFrag and Arabesque, we deploy HDFS on the local SAS disk array (in RAID-5).

[1]Although TriAD is an 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.

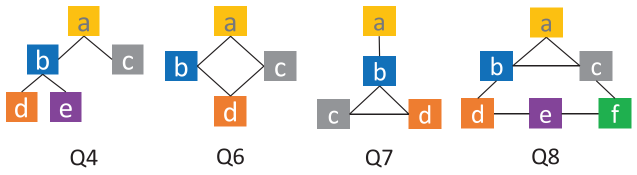


Fig. 12. The patterns (reproduced from Serafini et al. [2017]) used for comparison with QFrag and TriAD (results in Table 5). The label of each vertex is mapped, in alphabetical order, to the most frequent label of the graph in decreasing order of frequency. Here, *a* represents the most frequent label, *b* is the second most frequent label, and so on.

Table 5. Performance Comparison between QFrag, TriAD and PruneJuice Using the Patterns in Figure 12

	QFrag		TriAD		PruneJuice MPI		PruneJuice OpenMP	
	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

The table shows the runtime in seconds for full enumeration for QFrag and TriAD. For PruneJuice, we split time-to-solution into pruning time (top row) and enumeration time (bottom row). The best distributed runtime for a query, for each graph, is shown in bold.

7.8.1 Comparison with QFrag. Similarly 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 the Turbo$_{\text{ISO}}$ [Han et al. 2013] pattern enumeration algorithm (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 (Figure 12) 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 5: 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 time, our system (second pair of columns, labeled PruneJuice MPI, 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 implementation level (we use different processes, one MPI process per core), and has the system overhead of MPI-based 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 main 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 (we use OpenMP for parallelization) and present runtimes (when using 60 threads) for the same set of experiments in Table 5 (the two rightmost columns, labeled PruneJuice OpenMP). 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 (PruneJuice MPI) solution works about 4–10× faster than QFrag, and, if excluding distributed system overheads and considering the pruning time for the shared memory solution (PruneJuice OpenMP) and conservatively reusing enumeration runtime for the distributed solution, it is about 6–100× faster than QFrag.

7.8.2 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 that uses *partitioned locality*-based indexing. The 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. An RDF SPARQL [SPARQL1.0 2017] 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-worker architecture at indexing time, but allows for a direct, asynchronous communication among the worker nodes 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 statistic (collected at local index creation time on worker 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 graph datasets, queries, and large shared memory platform. The experiment results are summarzed in Table 5, in the columns next to the QFrag results. For the smaller, less skewed, and sparser *Patent* graph, except for search template Q4, TriAD's performance is on par with the distributed implementation of PruneJuice (and better than QFrag). In all other cases, particularly for the more skewed and dense *YouTube* graph, TriAD performs much worse. For Q4 and Q8, TriAD is ~5× and ~2× slower than QFrag, and ~20× and ~9× slower than distributed PruneJuice.

Although at the implementation level TriAD shares some similarities with PruneJuice (e.g., it leverages asynchronous processing); in contrast to QFrag and PruneJuice, 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., the Java Runtime Environment (JRE)) that is slower than the native MPI/C++

Table 6. Performance Comparison between Arabesque and Our Pattern Matching System (PJ)

	3-Clique			4-Clique		
	Arabesque	PJ (1)	PJ (20)	Arabesque	PJ (1)	PJ (20)
CiteSeer	3.2 s	0.04 s	0.02 s	3.6 s	0.06 s	0.02 s
Mico	13.6 s	27 s	11 s	1 min	72 min	21 min
Patent	1.3 min	17.3 s	1.6 s	2.2 min	32.8s	8.3 s
Youtube	6.5 min	2.1 min	12.7 s	Crash	6.4 min	1.4 min
LiveJournal	8.9 min	2.4 min	11.2 s	2.5 hr+	1.8 hr	41.3 min

The table shows the runtime for counting 3-Clique and 4-Clique patterns. Here, PruneJuice runtimes for the single node, shared memory are under the column with header PJ (1) while runtimes for the 20 node, distributed deployment are under the column with header PJ (20).

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 that 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. [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.

7.8.3 Comparison with Arabesque. Arabesque is a exact matching framework offering precision and recall guarantees, implemented on top of Apache Spark and Giraph [Giraph 2016]. Arabesque provides an API based on the TLE paradigm, which enables a user 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 main memory of a single node. As 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 use case that is favourable to Arabesque as our system is not specifically optimized for match counting. Table 6 compares results of counting three- and four-vertex cliques, using Arabesque and our system (PJ), using the same real-world graphs used for the evaluation of Arabesque in [Teixeira et al. 2015]. These experiments use the same shared memory machine used earlier. Additionally, for PruneJuice, 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 128 GB main memory. Our multi-core

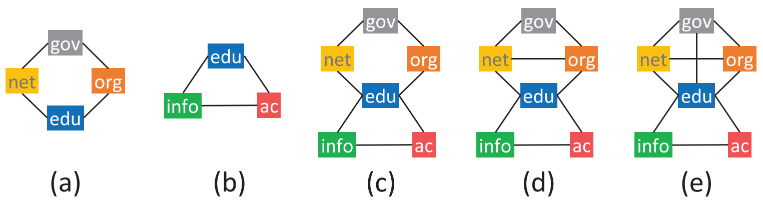


Fig. 13. *WDC* patterns used for template topology sensitivity analysis. Templates (a) and (b) are monocycles, each has a vertex with the label *edu*. Template (c) is created through union of (a) and (b). Templates (d) and (e) are constructed from (c) by incrementally adding one edge at a time.

shared memory testbed, however, has 1.5 TB 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).

PruneJuice 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 single node, shared memory machine. 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—PruneJuice is at most 3.7× faster. The 4-Clique pattern, highlights the advantage of our system: For the *Patent* graph, PruneJuice is 4× faster on the shared memory 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. PrinuJuice, however, completed clique counting for both graphs. For the smaller, yet highly skewed *Mico* graph Arabesque outperforms PruneJuice: For the 4-Clique pattern, Arabesque completes clique counting in about one minute, where as it takes PruneJuice 72 minutes on the same platform; this workload highlights the advantage of replicating the data graph for parallel processing that also presents the opportunity for harnessing load balancing techniques that are efficient and effective.

7.9 Analyzing Sensitivity to Search Template Properties

We investigate the influence of template properties, such as label selectivity and topology, on the runtime of the graph pruning procedure. For this study, we consider the *WDC* graph and the patterns in Figures 4 and 13.

Impact of Label Selectivity. We consider the WDC-3 and WDC-4 patterns (Figure 4): WDC-4, which has the same topology as WDC-3 yet has labels that are less frequent. The two patterns share five out of the eight vertex labels; the labels of WDC-3 and WDC-4, respectively, cover ~15% and ~4% of the vertices in the background graph. For WDC-4, the solution subgraph ($|\mathcal{V}^*| = 430$ and $2|\mathcal{E}^*| = 914$) is about two orders of magnitude smaller than that of WDC-3 (see Figure 5). The pruning time for WDC-4 is at most 2.6× faster on 512 nodes, averaging 1.8× faster across different scales.

Impact of Template Topology. The template topology dictates the type and the number of different constraints to be verified. For example, if the template has a single cycle (Figure 13(a)), then only a single cycle check is required; if the template is not edge-monocyclic (e.g., Figure 13(d)), then the relatively more expensive template-driven search is needed for precision guarantees. To

Table 7. Runtime for Pruning (with Precision Guarantees) and Size of the Pruned Solution Subgraph for Each *WDC* Pattern in Figure 13, Used for Template Topology Sensitivity Analysis

Template	(a)	(b)	(c)	(d)	(e)
$\|\mathcal{V}^*\|$	413,527	548	18,345	39	8
$2\|\mathcal{E}^*\|$	4,095,646	1,506	139,260	166	34
Time	41 min	39 s	2.6 min	2.1 min	1.8 min

The table lists the number of vertices ($|\mathcal{V}^*|$) and edges ($2|\mathcal{E}^*|$) in the solution subgraph.

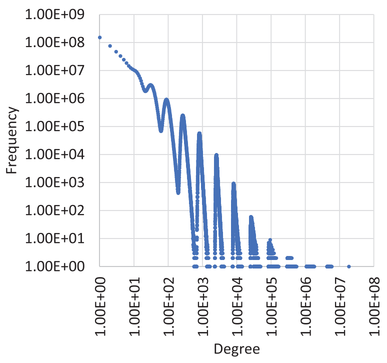
understand how the template topology influence performance, we study the *WDC* patterns in Figure 13: Templates (a) and (b) each has a single cycle. Template (c) is created through union of (a) and (b). Templates (d) and (e) are constructed from (c) by incrementally adding one edge at a time. Templates (a)–(c) are edge-monocyclic, thus only need checking cycle constraints. Non-edge-monocyclic templates (d) and (e) require the template-driven search; template (e) needs to verify the existence of a 4-Clique (consisting of vertices with labels *gov*, *org*, *edu*, and *net*). From the topology point of view, among all the constraints here in these examples, the clique is the the most complex substructure and its verification requires the longest walk (TDS constraint). We run these experiments on a 64 node deployment.

Table 7 lists the runtimes for pruning (with precision guarantees) for the *WDC* patterns in Figure 13. The table also shows the number of vertices ($|\mathcal{V}^*|$) and edges ($2|\mathcal{E}^*|$) in the solution subgraph for each pattern. While, at first sight, one would expect that the more constraints there are to verify, the slower the system will prune to a precise solution, our experience with the patterns in Figure 13 proves the contrary. Template (a) has only one four-cycle to check; however, it has the slowest time-to-solution as it leads to a large solution subgraph (due to the presence of 400M+ vertices in background graph with the labels *org* and *net*). Template (c) and the two templates (d) and (e) that require template-drive search, show, on average, ~20× faster time-to-solution compared to template (a). The complex templates (c), (d), and (e) introduce additional local and non-local constraints. There are at least an order of magnitude more vertices in the background graph, with labels *org* and *net*, that satisfy the constraints of template (a) than that of templates (c), (d) and (e). As a result, templates (c), (d), and (e) eliminate the majority of the non-matching vertices and edges early, leading to a faster time-to-solution; with the most complex template (e) being the rarest and the fastest to finish among the three.

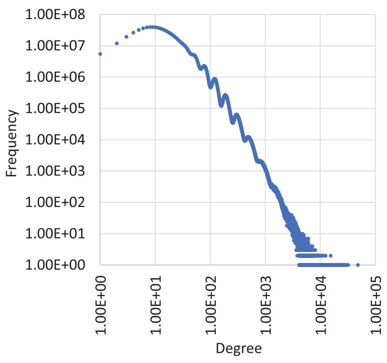
A key observation here is that it is the abundance of the constraints (in the background graph) that governs performance: template (c), which incorporates the four-cycle constraint that is not present in (b), has an order of magnitude more vertex and edge matches in the background graph, as well as has a slower runtime than that of (b). Similarly, there are only a handful of vertices in the background graph that satisfy the requirements of the complex substructure of (e), i.e., they belong to a clique. Rarity of this constrain leads to rapid pruning, resulting in (e) achieving a faster time-to-solution compared to (c) and (d).

7.10 Pattern Matching in Graphs with Diverse Topology

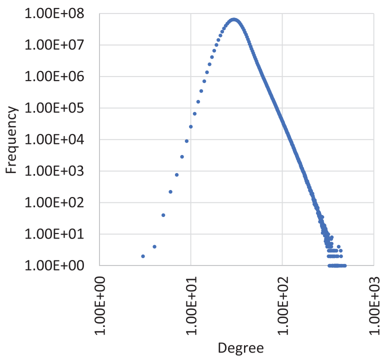
We demonstrate the ability of effectively processing both labeled and unlabeled graphs with different topological properties: vertex degree distributions, edge density and diameter. To this end, we use there real-world graphs: *Twitter*, *UK Web*, and *Road USA* (graph properties are listed in Table 4); and three R-MAT generated graphs (same size yet different vertex degree distribution, Figure 14).



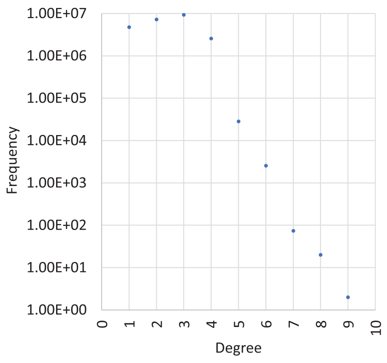
(a) Graph 500 - d_{max} 18.7M; d_{stdev} 1.6K; (0.57, 0.19, 0.19, 0.05)



(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. Figures (a)–(c) show vertex degree distribution of three R-MAT generated graphs with varying skewness. The figures show R-MAT edge probabilities (A, B, C, D) used. For (a)–(c), X and Y axes are in the log scale. Here, the graph Scale (30) and directed edge factor (16) are the same for all three graphs. The storage requirement for each graph is ~270 GB. Figure (d) shows vertex degree distribution of the large diameter Road USA graph. Here, only the Y axis is in log scale.

We present runtimes for full match enumeration. We run each experiment on 64 compute nodes; except for the smaller Road USA, for which we use eight compute nodes.

7.10.1 Large Real-World Power Law Graphs. Twitter and *UK Web* are billion edge, real-world graphs that have previously used to study a wide range of graph analysis problems; yet, rarely in the context of exact pattern matching. Although both are power-law graphs, they have significantly different 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 we use the same labeling technique used in the past [Plantenga 2013; 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 Figure 12 (previously used by Serafini et al. [2017]). Table 8 lists, for each search template, the full match enumeration time (includes time spent in pruning), match count, and the number of vertices

Table 8. Full Match Enumeration Time, for Some of the Search Queries in Figure 12, in the *Twitter* and *UK Web* Graphs (Table 4)

	Twitter			UK Web		
#Unique vertex labels	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
Match count	10B	78M	615M	3.8B	2.1B	45B
Time	1.2 hr	5 hr	1 hr	12.6 min	49.4 min	8.1 hr

For each template, the table lists number of vertices ($\lvert\mathcal{V}^*\rvert$) and edges ($2\lvert\mathcal{E}^*\rvert$) in the final solution subgraph, match count, and time-to-solution (includes time spent in pruning and match enumeration).

Table 9. Match Counting in the *Road USA* Graph: The Reported Runtimes Include Time Spent in Pruning as Well as Match Counting

Unlabeled template	UQ4	5-Star	3-Clique	4-Clique
Match count	220M	66M	1.3M	90
Time	26.7 s	17.3 s	5.0 s	1.6 s

UQ4 has the same topology as Q4 in Figure 12; however, it 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 that the Road USA graph is relatively small, we run these experiments on eight compute nodes.

and edges in the solution subgraphs. 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 within a limited number of graph partitions that 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 Section 7.6.)

7.10.2 Large Diameter Real-World Network. The *Road USA* graph has a very large diameter and is not labeled. Table 9 lists runtimes for counting matches for four unlabeled patterns. Results show, on the one side, abundance of small acyclic patterns compared to cyclic structures in the road network graph, and on the other side, the ability of our framework to support searches on large, unlabeled graphs with a completely different topology.

7.10.3 Experiments on Synthetic Graphs with Varying Vertex Degree Distribution. We further evaluate our system in presence of graphs with vastly different topologies using R-MAT generated synthetic graphs [Chakrabarti et al. 2004]. To generate power-law graphs, the R-MAT model recursively subdivides the graph (initially empty) into four equal-sized partitions and distributes edges within these partitions with predetermined probabilities. Each edge chooses one of the four partitions with probabilities A, B, $C = B$, and D, respectively. These probabilities, determine the skewness of the generated graph: in summary, the higher A is the more skewed the power-law distribution becomes. 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 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 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). Figure 14(a)–(c) shows

Table 10. Searches in the Three *R-MAT* Graphs, in Figure 14(a)–(c), with Varying Degree Distribution, Using the Q4 Pattern in Figure 12 and the RMAT-2 Pattern in Figure 7

		Graph500	Chakrabarti et al.	Uniform
#Unique vertex labels		26		17 8
Q4	$\|\mathcal{V}^*\|$	4.4K	641M	7.5M
	$2\|\mathcal{E}^*\|$	7K	3.5B	16.4M
	Match count	946	4B	4.2M
	Time	4.2 s	174.2 s	54.1 s
RMAT-2	$\|\mathcal{V}^*\|$	2.3K	313M	0
	$2\|\mathcal{E}^*\|$	4.1K	920M	0
	Match count	551	1.4B	0
	Time	8.7 s	83.4 s	52.5 s

For each query in each graph, the table lists the number of vertices ($|\mathcal{V}^*|$) and edges ($2|\mathcal{E}^*|$) in the pruned solution subgraph, number of matches and time-to-solution (includes time spent in pruning and match enumeration).

the degree distribution of these *R-MAT* graphs. For all the graphs, we use the same Scale (30) and (directed) edge factor (16), leading to (undirected) graphs with 34B edges. We follow the same approach as used for weak scaling experiments in Reza et al. [2018], Section 5A, 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 10 shows results for full match enumeration in the three *R-MAT* graphs for the following two queries: Q4 in Figure 12 and RMAT-2 in Figure 7. For both search templates, most matches are found for the configuration labeled Chakrabarti et al. This is because the graph in Figure 14(b) has a higher frequency of high-degree vertices compared to the graphs in Figure 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 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.

8 LIMITATIONS

We categorize the limitations of our proposed system based on their respective sources.

Limitations stemming from major design decisions. Our pipeline inherits the limitations of systems primarily designed for exact matching (compared to systems that trade accuracy for performance, e.g., based on sampling [Iyer et al. 2018] or graph simulation [Fan et al. 2010]). Similarly, our system inherits all limitations of its communication and middleware infrastructure, MPI and HavoqGT, respectively. One example is the lack of sophisticated flow/congestion control mechanism provided by these infrastructures that sometimes lead to message buildup and system collapse.

Limitations stemming from the targeted uses cases. In the same vein, we note that our system targets a graph analytics scenario (queries that need to cover the entire graph), rather than the traditional graph database queries that attempt to find a specific pattern around a vertex indicated by the user.

Limitations stemming from attempting to design a generic system. Systems optimized for specific patterns may perform better, e.g., systems optimized to count/enumerate triangles [Suri and Vassilvitskii 2011], treelets [Zhao et al. 2012], or systems relying on multi-join indices [Sun et al. 2012] to support patterns with limited diameter.

9 CONCLUSION

This article presents a new algorithmic pipeline to support pattern matching on large-scale metadata graphs on distributed memory systems. We capitalize on the idea of *graph pruning* via *constraint checking* and develop asynchronous algorithms that use both vertex and edge elimination to iteratively prune the original graph and reduce it to a subgraph that is the union of all matches. We have developed pruning techniques that *guarantee a solution with* 100% *precision* (i.e., no false positives in the pruned subgraph) and 100% *recall* (i.e., all vertices and edges participating in matches are included) for *arbitrary search patterns*. Our algorithms are *vertex-centric* and *asynchronous*, thus, they map well onto existing high-performance graph frameworks. Our evaluation using up to 257 billion edge real-world graphs and up to 4.4 trillion edge synthetic *R-MAT* graphs, on up to 1,024 nodes (36,864 cores), confirms the scalability of our solution. We demonstrate that, depending on the search template, our approach prunes the graph by orders of magnitude that enables match enumeration and counting on graphs with trillions of edges. Our success stems from a number of key design ingredients: asynchronicity, aggressive vertex and edge elimination while harnessing massive parallelism, intelligent work aggregation to ensure low message overhead, effective pruning constraints, and lightweight per-vertex state.

While we believe our system, as described, represents a significant advance in practical pattern matching in large, real-world graphs, further investigations in a number of areas can improve the efficiency and robustness of our solution. (i) The graph pruning pipeline introduces a number of decision problems. At present it uses ad-hoc heuristics, developed based on our intuition. We believe modelling approaches similar to the one explored in Tripoul et al. [2018] can be used to inform the following decisions: constraint selection and ordering, when to trigger load balancing, and when to switch from pruning to direct enumeration. (ii) The current prototype implementation can be extended to enable support for a richer set of subgraph matching scenarios, e.g., pattern matching in graphs and templates with edge metadata; querying dynamic/time-evolving graphs [Boldi et al. 2008; Han et al. 2014; Sallinen et al. 2016; Vora et al. 2017] and approximate pattern matching [Alon et al. 2008; Bunke and Allermann 1983; Conte et al. 2004; Iyer et al. 2018]. (iii) Further design/system optimizations, especially for non-local constraint checking and full match enumeration, to improve memory and message efficiency, load balance and task parallelism.

REFERENCES

Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. 2017. A survey and experimental comparison of distributed SPARQL engines for very large RDF data. *Proc. VLDB Endow.* 10, 13 (September 2017), 2049–2060. DOI: https://doi.org/10.14778/3151106.3151109

Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S. Cenk Sahinalp. 2008. Biomolecular network motif counting and discovery by color coding. *Bioinformatics* 24, 13 (July 2008), 241–249. DOI: https://doi.org/10.1093/bioinformatics/btn163

Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06)*. ACM, New York, NY, 44–54. DOI: https://doi.org/10.1145/1150402.1150412

Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC'09)*. ACM, New York, NY, Article 18, 11 pages. DOI: https://doi.org/10.1145/1654059.1654078

J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. 2007. Software and algorithms for graph queries on multithreaded architectures. In *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium*. 1–14. DOI: https://doi.org/10.1109/IPDPS.2007.370685

Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web (WWW'11)*. ACM, New York, NY, 587–596. DOI: https://doi.org/10.1145/1963405.1963488

Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A large time-aware web graph. *SIGIR Forum* 42, 2 (Nov. 2008), 33–38. DOI: https://doi.org/10.1145/1480506.1480511

P. Boldi and S. Vigna. 2004. The webgraph framework I: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web (WWW'04)*. ACM, New York, NY, 595–602. DOI: https://doi.org/10.1145/988672.988752

H. Bunke and G. Allermann. 1983. Inexact graph matching for structural pattern recognition. *Pattern Recogn. Lett.* 1, 4 (May 1983), 245–253. DOI: https://doi.org/10.1016/0167-8655(83)90033-8

V. G. Castellana, A. Morari, J. Weaver, A. Tumeo, D. Haglin, O. Villa, and J. Feo. 2015. In-memory graph databases for web-scale data. *Computer* 48, 3 (2015), 24–35.

V. T. Chakaravarthy, M. Kapralov, P. Murali, F. Petrini, X. Que, Y. Sabharwal, and B. Schieber. 2016. Subgraph counting: Color coding beyond trees. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*. 2–11. DOI: https://doi.org/10.1109/IPDPS.2016.122

Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 4th SIAM International Conference on Data Mining*. Society for Industrial Mathematics, p. 442.

D. Chavarría-Miranda, V. G. Castellana, A. Morari, D. Haglin, and J. Feo. 2016. GraQL: A query language for high-performance attributed graph databases. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'16)*. 1453–1462.

Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-miner: An efficient task-oriented graph mining system. In *Proceedings of the 13th EuroSys Conference (EuroSys'18)*. ACM, New York, NY, Article 32, 12 pages. DOI: https://doi.org/10.1145/3190508.3190545

J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. 2008. Fast graph pattern matching. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. 913–922. DOI: https://doi.org/10.1109/ICDE.2008.4497500

Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A selectivity based approach to continuous pattern detection in streaming graphs. In *Proceedings of the 2015 18th International Conference on Extending Database Technology (EDBT'15)*.

D. Conte, P. Foggia, C. Sansone, and M. Vento. 2004. Thirty years of graph matching in pattern recognition. *Int. J. Pattern Recogn. Artif. Intell.* 18, 3 (2004), 265–298. DOI: https://doi.org/10.1142/S0218001404003228 arXiv:http://www.worldscientific.com/doi/pdf/10.1142/S0218001404003228

Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. 2016. GraphFrames: An integrated api for mixing graph and relational queries. In *Proceedings of the 4th International Workshop on Graph Data Management Experiences and Systems (GRADES'16)*. ACM, New York, NY, Article 2, 8 pages. DOI: https://doi.org/10.1145/2960414.2960416

Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A general-purpose graph pattern mining system. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'19)*. Association for Computing Machinery, New York, NY, 1357–1374. DOI: https://doi.org/10.1145/3299869.3319875

Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. GraMi: Frequent subgraph and pattern mining in a single large graph. *Proc. VLDB Endow.* 7, 7 (March 2014), 517–528. DOI: https://doi.org/10.14778/2732286.2732289

Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. 2010. Graph pattern matching: From intractable to polynomial time. *Proc. VLDB Endow.* 3, 1–2 (September 2010), 264–275. DOI: https://doi.org/10.14778/1920841.1920878

Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Diversified top-k graph pattern matching. *Proc. VLDB Endow.* 6, 13 (August 2013), 1510–1521. DOI: https://doi.org/10.14778/2536258.2536263

A. Fard, M. U. Nisar, L. Ramaswamy, J. A. Miller, and M. Saltz. 2013. A distributed vertex-centric approach for pattern matching in massive graphs. In *Proceedings of the 2013 IEEE International Conference on Big Data*. 403–411. DOI: https://doi.org/10.1109/BigData.2013.6691601

J. Gao, C. Zhou, J. Zhou, and J. X. Yu. 2014. Continuous pattern detection over billion-edge graph using distributed framework. In *Proceedings of the 2014 IEEE 30th International Conference on Data Engineering*. 556–567. DOI: https://doi.org/10.1109/ICDE.2014.6816681

Giraph. 2016. Giraph. Retrieved from http://giraph.apache.org.

Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, 17–30. http://dl.acm.org/citation.cfm?id=2387880.2387883

Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 599–613. http://dl.acm.org/citation.cfm?id=2685048.2685096

Graph 500. 2016. Graph 500 Benchmark. Retrieved from http://www. graph500.org.

GraphFrames. 2017. GraphFrames. Retrieved from http://graphframes.github.io.

O. Green, J. Fox, A. Watkins, A. Tripathy, K. Gabert, E. Kim, X. An, K. Aatish, and D. A. Bader. 2018. Logarithmic radix binning and vectorized triangle counting. In *Proceedings of the 2018 IEEE High Performance extreme Computing Conference (HPEC'18)*. 1–7.

Aditya Grover and Jure Leskovec. 2016. Node2Vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'16)*. ACM, New York, NY, 855–864. DOI:https://doi.org/10.1145/2939672.2939754

Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-time twitter recommendation: Online motif detection in large dynamic graphs. *Proc. VLDB Endow.* 7, 13 (August 2014), 1379–1380. DOI:https://doi.org/10.14778/2733004.2733010

Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. 2014. TriAD: A distributed shared-nothing rdf engine based on asynchronous message passing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. ACM, New York, NY, 289–300. DOI:https://doi.org/10.1145/2588555.2610511

Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*. ACM, New York, NY, Article 1, 14 pages. DOI:https://doi.org/10.1145/2592798.2592799

Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM, New York, NY, 337–348. DOI:https://doi.org/10.1145/2463676.2465300

HavoqGT. 2016. HavoqGT. Retrieved from http://software.llnl.gov/havoqgt.

Huahai He and A. K. Singh. 2006. Closure-Tree: An index structure for graph queries. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*. 38–38. DOI:https://doi.org/10.1109/ICDE.2006.37

Keith Henderson, Brian Gallagher, Tina Eliassi-Rad, Hanghang Tong, Sugato Basu, Leman Akoglu, Danai Koutra, Christos Faloutsos, and Lei Li. 2012. RolX: Structural role extraction and mining in large graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'12)*. ACM, New York, NY, 1231–1239. DOI:https://doi.org/10.1145/2339530.2339723

M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. 1995. Computing simulations on finite and infinite graphs. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS'95)*. IEEE Computer Society, Los Alamitos, CA, 453.

Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. 2015. PGX.D: A fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*. ACM, New York, NY, Article 58, 12 pages. DOI:https://doi.org/10.1145/2807591.2807620

IMDb. 2016. IMDb Public Data. Retrieved from http://www.imdb.com/interfaces.

Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. ASAP: Fast, approximate graph pattern mining at scale. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berekely, CA, 745–761.

Amlan Kusum, Keval Vora, Rajiv Gupta, and Iulian Neamtiu. 2016. Efficient processing of large graphs via input reduction. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC'16)*. ACM, New York, NY, 245–257. DOI:https://doi.org/10.1145/2907294.2907312

Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is twitter, a social network or a news media?. In *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*. ACM, New York, NY, 591–600. DOI:https://doi.org/10.1145/1772690.1772751

BGL. 2017. The Boost Graph Library (BGL). Retrieved from http://www.boost.org/doc/libs/master/libs/graph/doc/index.html.

X. Liu, P. R. Pande, H. Meyerhenke, and D. A. Bader. 2013. PASQUAL: Parallel techniques for next generation genome sequence assembly. *IEEE Trans. Parallel Distrib. Syst.* 24, 5 (2013), 977–986.

David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. 2009. Classification of software behaviors for failure detection: A discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'09)*. ACM, New York, NY, 557–566. DOI: https://doi.org/10.1145/1557019.1557083

Diana H. P. Low, Bharadwaj Veeravalli, and David A. Bader. 2007. On the design of high-performance algorithms for aligning multiple protein sequences on mesh-based multiprocessor architectures. *J. Parallel Distrib. Comput.* 67, 9 (2007), 1007–1017. DOI: https://doi.org/10.1016/j.jpdc.2007.03.007

A. Lulli, E. Carlini, P. Dazzi, C. Lucchese, and L. Ricci. 2017. Fast connected components computation in large graphs by vertex pruning. *IEEE Trans. Parallel Distrib. Syst.* 28, 3 (March 2017), 760–773. DOI: https://doi.org/10.1109/TPDS.2016.2591038

Lustre. 2016. The Lustre File System. Retrieved from http://lustre.org.

Shuai Ma, Yang Cao, Jinpeng Huai, and Tianyu Wo. 2012. Distributed graph pattern matching. In *Proceedings of the 21st International Conference on World Wide Web (WWW'12)*. ACM, New York, NY, 949–958. DOI: https://doi.org/10.1145/2187836.2187963

D. Makkar, D. A. Bader, and O. Green. 2017. Exact and parallel triangle counting in dynamic graphs. In *Proceedings of the 2017 IEEE 24th International Conference on High Performance Computing (HiPC'17)*. 2–12.

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. ACM, New York, NY, 135–146. DOI: https://doi.org/10.1145/1807167.1807184

Brendan D. Mckay and Adolfo Piperno. 2014. Practical graph isomorphism, II. *J. Symb. Comput.* 60 (January 2014), 94–112. DOI: https://doi.org/10.1016/j.jsc.2013.09.003

Takunari Miyazaki. 1997. The complexity of mckay's canonical labeling algorithm. *Groups and Computation II, DIMACS Series Discrete Mathematics Theoretical Computer Science* 28, 1 (1997), 239–256.

Alessandro Morari, Vito Castellana, O. Villa, J. Weaver, G. Williams, David Haglin, Antonino Tumeo, and John Feo. 2015. *Gems: Graph Database Engine for Multithreaded Systems*. 139–156.

A. Morari, J. Weaver, O. Villa, D. Haglin, A. Tumeo, V. G. Castellana, and J. Feo. 2015. High-performance, distributed dictionary encoding of RDF datasets. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing*. 250–253.

Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 26, 10 (October 2004), 1367–1372. DOI: https://doi.org/10.1109/TPAMI.2004.75

Roger Pearce, Maya Gokhale, and Nancy M. Amato. 2013. Scaling techniques for massive scale-free graphs in distributed (external) memory. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS'13)*. IEEE Computer Society, Washington, DC, 825–836. DOI: https://doi.org/10.1109/IPDPS.2013.72

Roger Pearce, Maya Gokhale, and Nancy M. Amato. 2014. Faster parallel traversal of scale free graphs at extreme scale with vertex delegates. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*. IEEE Press, Piscataway, NJ, 549–559. DOI: https://doi.org/10.1109/SC.2014.50

Todd Plantenga. 2013. Inexact subgraph isomorphism in mapreduce. *J. Parallel Distrib. Comput.* 73, 2 (February 2013), 164–175. DOI: https://doi.org/10.1016/j.jpdc.2012.10.005

Quartz. 2017. Quartz. Retrieved from https://hpc.llnl.gov/hardware/platforms/Quartz.

RDF. 2017. LargeTripleStores. Retrieved from https://www.w3.org/wiki/LargeTripleStores.

Reddit. 2017. Reddit Public Data. Retrieved from https://github.com/dewarim/reddit-data-tools.

T. Reza, C. Klymko, M. Ripeanu, G. Sanders, and R. Pearce. 2017. Towards practical and robust labeled pattern matching in trillion-edge graphs. In *Proceedings of the 2017 IEEE International Conference on Cluster Computing (CLUSTER'17)*. 1–12. DOI: https://doi.org/10.1109/CLUSTER.2017.85

Tashin Reza, Matei Ripeanu, Geoffrey Sanders, and Roger Pearce. 2020a. Approximate pattern matching in massive graphs with precision and recall guarantees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. Association for Computing Machinery, New York, NY, 1115–1131. DOI: https://doi.org/10.1145/3318464.3380566

Tahsin Reza, Matei Ripeanu, Geoffrey Sanders, and Roger Pearce. 2020b. Towards interactive pattern search in massive graphs. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences and Systems and Network Data Analytics (GRADES-NDA'20)*. Association for Computing Machinery, New York, NY, Article 9, 6 pages. DOI: https://doi.org/10.1145/3398682.3399166

Tahsin Reza, Matei Ripeanu, Nicolas Tripoul, Geoffrey Sanders, and Roger Pearce. 2018. PruneJuice: Pruning trillion-edge graphs to a precise pattern-matching solution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*. IEEE Press, Piscataway, NJ, Article 21, 17 pages.

Pedro Ribeiro and Fernando Silva. 2014. G-Tries: A data structure for storing and finding subgraphs. *Data Min. Knowl. Discov.* 28, 2 (March 2014), 337–377. DOI: https://doi.org/10.1007/s10618-013-0303-4

Oliver Lehmberg Robert Meusel, Christian Bizer. 2016. Web Data Commons—Hyperlink Graphs. Retrieved from http://webdatacommons.org/hyperlinkgraph/index.html.

Ryan A. Rossi and Nesreen K. Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI'15)*.

Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, Hassan Chafi, Anthony Potter, Boris Motik, and Ian Horrocks. 2017. PGX.D/Async: A scalable distributed graph pattern matching engine. In *Proceedings of the 5th International Workshop on Graph Data-management Experiences & Systems (GRADES'17)*. ACM, New York, NY, Article 7, 6 pages. DOI: https://doi.org/10.1145/3078447.3078454

Scott Sallinen, Keita Iwabuchi, Suraj Poudel, Maya Gokhale, Matei Ripeanu, and Roger Pearce. 2016. Graph colouring as a challenge problem for dynamic graph processing on distributed systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)*. IEEE Press, Piscataway, NJ, Article 30, 12 pages. http://dl.acm.org/citation.cfm?id=3014904.3014945

Marco Serafini, Gianmarco De Francisci Morales, and Georgos Siganos. 2017. QFrag: Distributed graph search via subgraph isomorphism. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*. ACM, New York, NY, 214–228. DOI: https://doi.org/10.1145/3127479.3131625

Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.* 1, 1 (August 2008), 364–375. DOI: https://doi.org/10.14778/1453856.1453899

G. M. Slota and K. Madduri. 2014. Complex network analysis using parallel approximate motif counting. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS'14)*. IEEE, 405–414. DOI: https://doi.org/10.1109/IPDPS.2014.50

Spark. 2017. Spark. Retrieved from https://spark.apache.org.

SPARQL1.0. 2017. SPARQL 1.0. Retrieved from https://www.w3.org/TR/rdf-sparql-query.

Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.* 5, 9 (May 2012), 788–799. DOI: https://doi.org/10.14778/2311906.2311907

Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High performance graph analytics made productive. *Proc. VLDB Endow.* 8, 11 (July 2015), 1214–1225. DOI: https://doi.org/10.14778/2809974.2809983

Siddharth Suri and Sergei Vassilvitskii. 2011. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web (WWW'11)*. ACM, New York, NY, 607–614. DOI: https://doi.org/10.1145/1963405.1963491

Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A system for distributed graph mining. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, NY, 425–440. DOI: https://doi.org/10.1145/2815400.2815410

Hanghang Tong, Christos Faloutsos, Brian Gallagher, and Tina Eliassi-Rad. 2007. Fast best-effort pattern matching in large attributed graphs. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'07)*. ACM, New York, NY, 737–746. DOI: https://doi.org/10.1145/1281192.1281271

N. Tripoul, H. Halawa, T. Reza, G. Sanders, R. Pearce, and M. Ripeanu. 2018. There are trillions of little forks in the road. choose wisely! Estimating the cost and likelihood of success of constrained walks to optimize a graph pruning pipeline. In *Proceedings of the 2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3'18)*. 20–27. DOI: https://doi.org/10.1109/IA3.2018.00010

J. R. Ullmann. 1976. An algorithm for subgraph isomorphism. *J. ACM* 23, 1 (January 1976), 31–42. DOI: https://doi.org/10.1145/321921.321925

Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 237–251. DOI: https://doi.org/10.1145/3093337.3037748

M. P. Wellman and W. E. Walsh. 2000. Distributed quiescence detection in multiagent negotiation. In *Proceedings of the 4th International Conference on MultiAgent Systems*. 317–324. DOI: https://doi.org/10.1109/ICMAS.2000.858469

Zhaoming Yin, Jijun Tang, Stephen W. Schaeffer, and David A. Bader. 2016. Exemplar or matching: Modeling DCJ problems with unequal content genome data. *J. Combin. Optimiz.* 32, 4 (1 November 2016), 1165–1181. DOI: https://doi.org/10.1007/s10878-015-9940-4

Ye Yuan, Guoren Wang, Jeffery Yu Xu, and Lei Chen. 2015. Efficient distributed subgraph similarity matching. *VLDB J.* 24, 3 (June 2015), 369–394. DOI: https://doi.org/10.1007/s00778-015-0381-6

Shijie Zhang, Jiong Yang, and Wei Jin. 2010. SAPPER: Subgraph indexing and approximate matching in large graphs. *Proc. VLDB Endow.* 3, 1-2 (September 2010), 1185–1194. DOI: https://doi.org/10.14778/1920841.1920988

Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. S. A. Kumar, and M. V. Marathe. 2012. SAHAD: Subgraph analysis in massive networks using hadoop. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 390–401. DOI: https://doi.org/10.1109/IPDPS.2012.44

Fang Zhou, Sébastien Mahler, and Hannu Toivonen. 2012. *Simplification of Networks by Edge Pruning*. Springer, Berlin, 179–198. DOI: https://doi.org/10.1007/978-3-642-31830-6_13

Feida Zhu, Qiang Qu, David Lo, Xifeng Yan, Jiawei Han, and Philip S. Yu. 2011. Mining top-K large structural patterns in a massive network. *Proc. VLDB Endow.* 4, 11 (8 2011), 807–818.

Received December 2018; revised October 2019; accepted August 2020