A Survey on Distributed Graph Pattern Matching in Massive Graphs

Sarra Bouhenni, Saïd Yahiaoui, Nadia Nouali-Taboudjemat, Hamamache Kheddouci

Institutions: University of Lyon

Published on: 09 Feb 2021 - ACM Computing Surveys (ACMPUB27New York, NY, USA)

Topics: Subgraph isomorphism problem, Time complexity and Context (language use)

A Survey on Distributed Graph Pattern Matching in Massive Graphs

Sarra Bouhenni, Said Yahiaoui, Nadia Nouali-Taboudjemat, Hamamache Kheddouci

A Survey on Distributed Graph Pattern Matching in Massive Graphs

SARRA BOUHENNI, Ecole nationale Supérieure d'Informatique, Algérie and Université de Lyon, Université Lyon 1, LIRIS CNRS, France
SAÏD YAHIAOUI and NADIA NOUALI-TABOUDJEMAT, CERIST, Centre de Recherche sur l'Information Scientifique et Technique, Algérie
HAMAMACHE KHEDDOUCI, Université de Lyon, Université Lyon 1, LIRIS CNRS, France

Besides its NP-completeness, the strict constraints of subgraph isomorphism are making it impractical for graph pattern matching (GPM) in the context of big data. As a result, relaxed GPM models have emerged as they yield interesting results in a polynomial time. However, massive graphs generated by mostly social networks require a distributed storing and processing of the data over multiple machines. Thus, requiring GPM to be revised by adopting new paradigms of big graphs processing, e.g. Think-Like-A-Vertex and its derivatives. This paper discusses and proposes a classification of distributed GPM approaches with a narrow focus on the relaxed models.

CCS Concepts: • **Theory of computation → Distributed algorithms**; **Graph algorithms analysis**; • **Computing methodologies →** *Distributed algorithms.*

Additional Key Words and Phrases: Graph pattern matching, distributed graphs, graph simulation, subgraph isomorphism.

ACM Reference Format:
Sarra Bouhenni, Saïd Yahiaoui, Nadia Nouali-Taboudjemat, and Hamamache Kheddouci. 2020. A Survey on Distributed Graph Pattern Matching in Massive Graphs. *ACM Comput. Surv.* 1, 1 (October 2020), 35 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

With the rapid growth of the Internet, huge amounts of data are being generated with every passing minute. The challenges brought about by big data have led to the research community revising traditional techniques and algorithms that were designed for collecting, storing, analyzing, and visualizing data. According to an IBM report published in 2017 [93], 90 percent of the data in the world today has been generated in the last two years alone. A large part of these data is modeled as graphs, a natural and flexible data structure for modeling complex relationships and interactions between objects. Graph pattern matching (GPM) is a classical graph challenge, considered as one of the most studied problems in the literature. It refers to the problem of finding similarities between a

Authors' addresses: Sarra Bouhenni, cs_bouhenni@esi.dz, Ecole nationale Supérieure d'Informatique, BP M68, Oued Smar, 16309, Algérie , Université de Lyon, Université Lyon 1, LIRIS CNRS, Villeurbanne, 69622, France; Saïd Yahiaoui, syahiaoui@cerist.dz; Nadia Nouali-Taboudjemat, nnouali@cerist.dz, CERIST, Centre de Recherche sur l'Information Scientifique et Technique, Ben Aknoun, 16030, Algérie; Hamamache Kheddouci, hamamache.kheddouci@univ-lyon1.fr, Université de Lyon, Université Lyon 1, LIRIS CNRS, Villeurbanne, 69622, France.

pattern graph and a data graph. GPM has a wide range of applications, such as in-database analytics, search engines and software plagiarism detection.

The most classical approaches used to evaluate GPM queries are through subgraph isomorphism. Ullmann's paper [85] is one of the pioneering works in this direction, in which the first simple to understand algorithm for solving GPM via subgraph isomorphism was proposed. Since then, several algorithms have been put forward, aiming to reduce time and space complexity and addressing the problems of specific graph classes, e.g. trees, bipartite graphs, and direct acyclic graphs. Nevertheless, subgraph isomorphism is known to have several limitations. It is an NP-Complete problem which makes it impractical for large graphs, not to mention the massive graphs containing up to billions of nodes and trillions of edges. Furthermore, the strict constraints imposed by this model are not interesting for the current real-world applications especially in social networks. As a result, graph simulation has been proposed. It is a recent and flexible model that relaxes the matching constraints by matching edges between the pattern graph and the data graph instead of matching the whole query, which makes it solvable in polynomial time. Since then, several models emerged, aiming to reduce the time and space complexity while keeping an acceptable accuracy depending on the application domain and its requirements in terms of precision and response time, for example bounded simulation [22] and strong simulation [53]. These newly introduced models consider not only the topological structure of the graphs, but also the semantic information carried by labels and attributes on nodes and edges.

Moreover, with the advance of technologies and the increasing amounts of data needing to be processed in real time, a great deal of work still needs to be carried out in this field. Due to their large size and the hardware constraints, big graphs need to be distributed over multiple machines. Which raises new challenges that require traditional graph algorithms to be reexamined. Furthermore, recent studies have shown that the general-purpose programming models like Hadoop Map/Reduce are inefficient in processing distributed big graphs. That is why, new paradigms specially designed for big graph processing appeared including vertex-centric, edge-centric and subgraph-centric programming models. They allow us to build linearly scalable algorithms intended to run on clusters composed of dozens to hundreds of machines. Our research study is focused on distributed GPM algorithms that are designed for scalable and highly dynamic graphs under these programming paradigms. Even though some works address GPM for special categories of queries like triangles and cliques, we review in this paper only the approaches addressing arbitrary patterns, i.e. all categories of patterns from simple to very complex graph queries.

Many previous works reviewed and proposed different taxonomies of sequential subgraph isomorphism algorithms [14, 28, 30, 88]. Other works like [8, 49, 79] conducted a benchmarking to compare some of the very well known subgraph isomorphism algorithms on the same environment. Furthermore, to find subgraph isomorphism matches, we can distinguish two categories of algorithms; exploration-based and join-based. Lai *et al.* surveyed and conducted an experimental comparison of a set of join-based GPM algorithms and classified them into three categories based on the join strategy applied in [48]. However, to the best of our knowledge, this is the first work to survey distributed GPM approaches including relaxed GPM. Unlike previous surveys, we provide a taxonomy of the distributed GPM approaches based on graph exploration and which adopt the emerging *Think-Like-A-Vertex* (TLAV) paradigm and its derivatives. We also discuss these programming models and how they improve solving GPM in massive graphs. This survey is a synthesis of the work conducted recently in GPM on distributed graph processing systems with a narrow focus on relaxed matching that has gained an increased interest during the past ten years.

Section 2 provides an overview of subgraph isomorphism algorithms with alternative models, i.e. graph simulation and its extensions, while Section 3 presents some of the programming models and systems used in distributed graph processing. Section 4 outlines the distributed solutions suggested for GPM on massive graphs. Section 5 proposes a taxonomy for classifying the different approaches discussed, whereas, Section 6 provides some conclusions and highlights some open issues.

2 Graph pattern matching

Before delving into the different solutions of subgraph matching proposed recently, we firstly define some basic concepts in this field of study.

GPM is the problem of finding similarities between an input graph called pattern graph and a data graph, it has several formulations or representations. We find structural graph matching that includes graph isomorphism and subgraph isomorphism. Other models based on simulation represent a flexible alternative, and are widely used in today's real-life applications. Under this category, referred to as relaxed GPM, we find graph simulation and its extensions such as bounded simulation, dual simulation and relaxation simulation. Structural matching considers the structural information contained in the graph, i.e. the information available is the edges between the graph vertices and it does not capture matches that are close to the query. Vertices and edges may contain semantic information in the form of labels making the matching semantic. On the other hand, there is the relaxed graph matching that considers less the structural information, and more the semantic one carried by labels on the data graph vertices or edges. These labels are very informative as they participate in the matching process, such that two nodes or edges with the same label are considered similar. A node/edge can have multiple labels, they are generally represented as a set of attributes in the form of *(key, value)* where *key* is the label type and *value* is the label itself. In this type of graph pattern matching, structure of the answers may vary from that of the input query because the structural matching constraints are relaxed seeking lower complexity and to capture more similarities.

Next, we give a formal definition for a data graph, pattern graph (a.k.a. query graph), a subgraph and other notions used in evaluating GPM queries, namely a vertex eccentricity, a graph diameter and its center.

Definition 2.1 (Data Graph). *A data graph is defined as a directed graph $G = (V, E, f)$ where V is the set of its vertices, E is the set of all edges between vertices and f is a function that maps each vertex $v \in V$ to a label value $f(v)$ in Σ, the set of all labels.*

Definition 2.2 (Pattern Graph). *In the same way, we define a pattern graph as a directed graph $Q = (V_q, E_q, f_q)$ where V_q is the set of its vertices, E_q is the set of all edges and f_q is a function that maps each vertex $v \in V_q$ to a label value $f_q(v)$ in Σ_q, the set of all labels.*

Definition 2.3 (Subgraph). *A directed graph $G' = (V', E', f')$ is considered subgraph of a graph $G = (V, E, f)$ if and only if (i) V' is a subset of V, (ii) E' is a subset of E and (iii) $\forall v \in V', f'(v) = f(v)$.*

Definition 2.4 (Graph distance). *The distance $d(u, v)$ between two vertices u and v in a connected graph $G = (V, E, f)$ is the minimum length of the paths connecting them. The length of a path $p = (u_1, u_2, ..., u_n)$, such that $\forall i \in \{1, n-1\} : (u_i, u_{i+1}) \in E$, is the number of its edges.*

Definition 2.5 (Eccentricity of a vertex). *Eccentricity $\epsilon(v)$ of a vertex $v \in V$ in graph $G = (V, E, f)$ is defined as the maximum graph distance between v and any other vertex in G. $\epsilon(v) = \max d(v, u), \forall u \in V - \{v\}$*

Definition 2.6 (Graph diameter). *Given a graph $G = (V, E, f)$, the diameter of G is the maximum eccentricity.*

Definition 2.7 (Graph Center and Radius). *Given a graph $G = (V, E, f)$, the centers of G are the vertices having minimum eccentricity. While its radius is the minimum eccentricity.*

2.1 Structural graph pattern matching

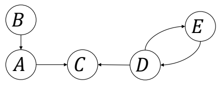
We define graph isomorphism and subgraph isomorphism, two of the most studied models in the literature. They are purely structural, i.e. emphasize the relations between vertices.

2.1.1 Graph isomorphism: Two graphs $G = (V, E, f)$ and $Q = (V_q, E_q, f_q)$ are isomorphic if and only if there is a bijective mapping $R(R : V_q \longrightarrow V)$ that satisfies the following condition: $\forall u, u' \in V_q$, if $R(u) = v \in V$ and $(u, u') \in E_q$ then $\exists v' \in V$ such that $R(u') = v'$ and $(v, v') \in E$.

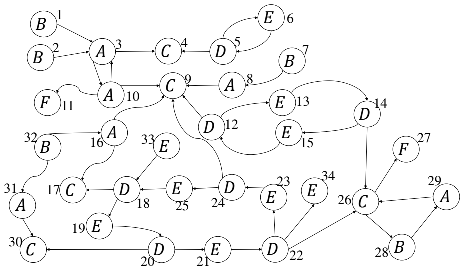
Graph isomorphism is used for comparing graphs of the same size. In this paper, we are interested in techniques and models that allow us to find answers to a query graph in a much larger graph.

2.1.2 Subgraph isomorphism: Subgraph isomorphism (ISO) is defined as an injective mapping between the vertices of the two graphs $Q = (V_q, E_q, f_q)$ and $G = (V, E, f)$. In other words, it is the problem of finding all the subgraphs of a data graph G that are isomorphic to a graph pattern Q. Formally, G matches Q via subgraph isomorphism if and only if: $\forall u, u' \in V_q$, if $R(u) = v \in V$ and $(u, u') \in E_q$ then $\exists v' \in V$ such that $R(u') = v'$ and $(v, v') \in E$, we note here that the mapping is not bijective.

Figure 1 shows an example of a pattern graph Q_1 and a data graph G_1. G_1 represents a social network, where vertices in G_1 refer to employees, while an edge between vertices v and v' means that v has endorsed v' on this social network (they worked together in the past). A label on a data node indicates the job of the employee represented by this vertex. In the process of hiring a new team, a pattern graph modeling the relations required between the members of such team is used to find eligible candidates. For example, an edge (A, B) in the pattern graph is translated to the following requirement: the person who takes job A should have a past experience working with the employee getting job B, A should also have recommended B on this social network. When applying subgraph isomorphism, we can extract two matched subgraphs from G_1. Their corresponding vertices are $\{1, 3, 4, 5, 6\}$ and $\{2, 3, 4, 5, 6\}$ as it is shown in Figure 2.


$$\begin{tikzcd} B \arrow[d] & & & E \arrow[dl, bend left] \\ A \arrow[r] & C & D \arrow[l] \arrow[ur, bend left] & \end{tikzcd}$$

(a) A pattern graph Q_1



(b) A data graph G_1

Fig. 1. An example of a data graph containing employees in a professional social network, where labels refer to their jobs and an edge between two vertices means that the source endorsed the destination.

2.1.3 Algorithms for evaluating subgraph isomorphism: Subgraph isomorphism is an NP-Complete problem [33] that has been the subject of several studies over the past decades. We distinguish two types of algorithms to evaluate it; exact and approximate algorithms. The exact algorithms find all subgraphs isomorphic to the input query which leads to an exponential time complexity, whereas the other algorithms provide approximate matches yielding to some improvements in the required running time and space but with a loss in the results quality. The comprehensive Surveys [14, 28, 30, 88] give an overview of the different techniques used to evaluate exact and inexact subgraph isomorphism since the seventies until 2015.

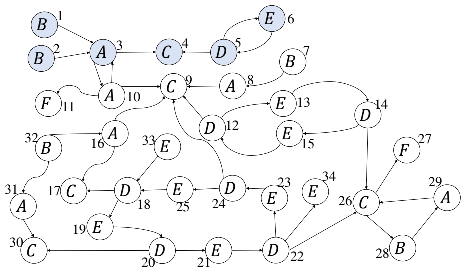


Fig. 2. Results for ISO are colored in blue.

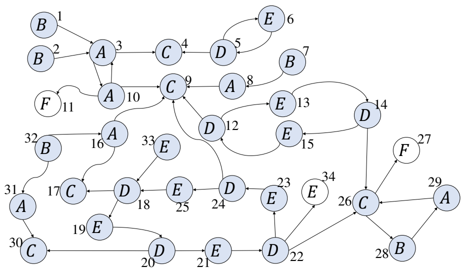


Fig. 3. Results for GSIM are colored in blue.

In its basic definition, subgraph isomorphism does not capture the semantic similarities that are generally represented by labels on both graph vertices and edges. However, recent approaches consider the semantic information in addition to the pattern structure.

In [85], Ullmann suggested a tree-based algorithm for solving graph isomorphism. It enumerates all the possible ways of matching query vertices with the data graph vertices starting from a mapping matrix. It also uses a refinement procedure that prunes the search space by eliminating unfruitful matches. It is based on the observation that a query vertex u and a data vertex v that are mapped together must also have their neighbors matched together. Thus, any match that does not satisfy this constraint is eliminated from the beginning.

All the works that followed are based on this seminal work. They address two common challenges; first, how to minimize the size of intermediate results by ordering the query vertices. Second, finding pruning strategies that eliminate useless computations.

The first algorithm is VF2 [15]. It is based on the state space representation for solving graph isomorphism. VF2 explores the search space starting from the neighbors of each matched query vertex. VF2Plus [11] improves the ordering of VF2 by picking the query vertices with the lowest chance of finding matches in the data graph and the highest number of neighbors among prior vertices in the ordering. VF3 [10] also improves VF2 by proposing heuristics for reducing the search space size and the time of processing each state. Moreover, VF3-Light [9]is another algorithm that eliminates some of VF3's heuristics which results in shorter running time.

Furthermore, we find algorithms that use exploration of the data graph to evaluate subgraph isomorphism such as QuickSI [72], RI [5] or VF2++ [41]. Others create tree/graph indices for storing the candidates' sets like GraphQL [38], TurboIso [37], BoostIso [63], CFL-Match [4], TurboFlux [45], DAF[36] and VC [80], while some construct indices based on the data graph and use them to assist the embeddings enumeration; GADDI[103] and SPath[105]. Finally, MQO-iso [64] constructs a DAG based on the relationships between multiple queries to guide subgraph isomorphism search.

As opposed to Ullmann that does not define an ordering of the query vertices, QuickSI [72] proposed to process vertices with infrequent vertex labels in the data graph as quickly as possible, which allows avoiding useless computations. QuickSI processes the edges/vertices having a lower number of possible embeddings first, hence, reduces the size of explored search space. It first selects a spanning tree that minimizes the number of possibly generated embeddings in the data graph. Then, it selects an order of processing by setting the first edge to explore as the one having minimum size of candidates for its source and destination. Finally, a DFS traversal of the spanning tree is conducted to enumerate all the possible embeddings. The two algorithms RI [5] and VF2++ [41] are similar to QuickSI as they all explore the data graph directly without generating any candidate sets. However, RI picks query vertices based on their degrees by first selecting the one with the highest degree, and then the ones with most neighbors in the current ordering. VF2++ picks query vertices with the least frequent labels in G and the highest degree in Q.

To improve the existing pruning rules, GraphQL [38] proposed to use a neighborhood signature of a query vertex u which is the set of its neighbors' labels. A data vertex v is pruned out if its neighborhood signature does not include the neighborhood signature of its matched query vertex. GraphQL picks the query vertex that is connected to an already matched vertex and that has a minimum number of candidates. Similarly, SPath [105] used an improved neighborhood signature to minimize the size of the candidates' set. It also matches a path-at-time.

TurboISO [37] is another algorithm that groups query vertices having the same label and the same neighborhood into one node to reduce the duplicated generation of unfruitful matches. It divides the data graph into separate Candidate Regions (CRs), where the root of every CR is a data node matched to the query tree root. For each CR, TurboISO enumerates the possible mappings based on a combine/permute strategy. So, given a sequence order and a candidate region CR, only one query vertex is processed and if it leads to an accepted answer, other combinations are generated from other vertices in its group. Otherwise, they are all eliminated.

Inspired by the query compression in TurboISO, BoostISO [63] compresses the data graph based on the relationships between data vertices. It groups the data vertices having the same matches to one hypernode in a new adapted graph that has a smaller size. BoostISO also groups similar query vertices which allows it to prune unfruitful matches as quickly as possible. A follow up revision of the filtering phase was made by the same authors in [90].

Furthermore, [64] proposes a multi-query optimization plan (MQO-iso) to reduce the processing time of multiple queries. The input query set is processed to detect the common subgraphs, then a DAG is built based on the isomorphic relationships between the queries such that there is an edge from q to q' if q is subgraph isomorphic to q'. This DAG guides the subgraph isomorphism search while exploiting the already generated intermediate results. The parents in the DAG are processed first and their results are cached then used to process their children.

CFL-Match [4] proposes to decompose the query graph into three substructures. It matches the cores first because they are less likely to be highly present in the data graph. Second, it processes the forest substructures that have high chances of being present in the graph, while leaves (one-degree vertices) are left to the end as they are most likely to generate most of the unnecessary embeddings that must be avoided. The core is a dense connected subgraph containing every single edge that is not present in a spanning tree of the query. Such substructure, when processed earlier, reduces the number of unpromising partial mappings by avoiding generating unnecessary Cartesian products. The forest is the complement of the core in the query graph without the leaves. To evaluate subgraph isomorphism, CFL-Match uses a Compact Path Index (CPI) having the same structure as a BFS tree q_t of the query graph. Its nodes represent a query vertex u with its label and candidate set. For every edge (u_1, u_2) in q_t, the CPI keeps the edges between the candidate sets of u_1 and u_2. This CPI is then traversed to enumerate ISO embeddings by processing the core, the forest and finally the leaves. For the core, it uses a path-based ordering that minimizes the number of generated embeddings while favoring the paths with non-tree edges.

On the other hand, DAF [36] addresses the limitations of TurboISO and CFL-Match by converting the query graph into a DAG instead of a spanning tree. The algorithm uses dynamic programming to find the candidate set for all the ISO embeddings then follows an adaptive, DAG-based ordering to build the possible embeddings.

Moreover, VC improves the path-based ordering with a cost function that considers edges also between the paths [80]. VC also uses a bigraph index for holding the candidate edges instead of a spanning tree as it is the case for CFL-Match.

Finally, to detect patterns in a dynamic graph, TurboFlux [45] uses a data-centric graph (DCG) representation of the intermediate results. A DCG is composed of the data graph such that for each data node, its candidate query vertices are represented by incoming edges. This DCG is then traversed to find subgraph isomorphism matches. TurboFlux defines transition rules on every edge in the DCG, then, whenever an edge is inserted or deleted, it first checks whether it matches an edge in the query graph or not and then updates the results according to the transition rules.

2.1.4 Parallel subgraph isomorphism approaches: In order to scale up the existing algorithms, two categories of approaches showed up, namely relational join-based and exploration-based.

2.1.4.1 Join-based approaches: The data graph edges are considered as relational database tables, such that each pair of labels forms a table, the attributes represent the two labels and the edge identifiers are the attribute values. These tables are filtered and a series of joins is applied to get the matching answers. However, joins are very costly and many works addressed this problem through different categories of joins: binary joins where the first two tables (edge candidates) are joined, then, the result is joined again with a third table of edge candidates until obtaining the end result [35, 61, 84, 101]. These works propose different join orders that aim at optimizing the size of intermediate results. Other works attempted to join substructures larger than edges which resulted into reducing further the size of intermediate results. In [81], the pattern graph was decomposed into stars (two level trees) that can be found through data graph exploration (they require only information about direct neighbors of each data vertex). After that, the matched substructures were joined to form a complete subgraph match. More recent works used different joining units such as the TwinTwig [46], Crystal [62] or Clique [47, 48].

On the other hand, Afrati *et al.* [1] used the multiway joins where all the edge candidate sets are joined at the same time. Furthermore, worst-case optimal join algorithms are also used for multiway joins. Two common algorithms are GenericJoin [57] and LeapFrog TrieJoin [87]. Parallel approaches for finding subgraph isomorphism based on these algorithms were proposed by [2] and [29] respectively. Mhedhbi *et al.* combined both binary joins and the GenericJoin algorithm to evaluate subgraph isomorphism in [56].

2.1.4.2 Exploration-based approaches: These are approaches based on data graph exploration, which naturally fit into the newly TLAV paradigms for distributed graph processing. We discuss this category more in detail in Section 4 after introducing the different concepts and programming models for distributed graph processing in Section 3. However, as an exception, we present here the four works QFrag [71], CECI [3], PSM [16] and BENU [92]. First, QFrag and CECI replicate the data graph on multiple machines where each processor runs ISO search on different parts of the data graph. Moreover, PSM evaluates ISO in parallel on a single machine, while BENU is presented here because it is also a parallel system that uses a similar approach to answer GPM queries.

QFrag [71] replicates the data graph on k workers and parallelizes TurboISO to evaluate subgraph isomorphism while using *task fragmentation* to achieve load balance. QFrag decomposes the first super-step of processing into two phases. In the first phase, each worker is assigned a list of roots from which it builds the corresponding CRs. It considers the embeddings enumeration of each CR as one subtask and estimates the processing time of each tree to detect any outliers that cannot be processed locally. Embedding enumeration of regular trees is conducted locally in the first super-step, while the outlier trees are split into equal subtasks and distributed over the workers to be processed in the second super-step.

CECI [3] is based on a spanning tree index constructed from a BFS traversal of the query graph. Each node in this tree maps a query node u to a data node v matched to it and to its neighbor for representing the tree edge candidates. It also keeps candidates of u's neighbors that are connected by a non-tree edge. This index is then used to enumerate ISO embeddings in parallel. Every candidate of the query tree root is used as a pivot of an Embedding Cluster (EC) that will be processed in parallel. To achieve load balancing, CECI estimates the cost of evaluating each EC, then decomposes the costly ones into multiple ECs. Each worker processes an equal number of ECs. Finally, CECI keeps the candidates of the non-tree edges in its index and uses an intersection between the two sets, tree-edge candidates and non-tree edges candidates, to enumerate the embeddings.

Furthermore, PSM [16] parallelizes backtracking based algorithms that adopt a DFS traversal of the candidates tree index on a single machine. PSM decomposes the tree index into search regions and each worker is assigned a region that it can expand independently in parallel. Load balancing is conducted dynamically where busy workers split their search regions into two equal subtasks.

Finally, BENU [92] also uses backtracking and decomposes the search tree into regions to be processed by different workers. The workers use adjacency lists of the data graph to enumerate possible embeddings through the application of set intersections. Load balancing is achieved by splitting a tree task into subtasks if the degree of the current tree root vertex is larger than a certain threshold.

2.1.5 Limitations of subgraph isomorphism: Although discussions regarding the approximate solutions have dominated the research on subgraph matching over the recent years, their performance is not yet able to manage the actual size of the GPM problem. We are facing today a growing need for more performing techniques and tools that can process big graphs; that are highly dynamic and distributed over hundreds of machines because of their enormous size. For example, Facebook is the largest social network with 2.4 billion monthly active users in June 2019 [20]. Furthermore, subgraph isomorphism requires the input graph to be structurally identical to the data graph. In its stringent formulation, it does not consider the intrinsic deformations of an object from its ideal model. Such drawbacks have led the research community to investigate other models of GPM that are less expensive in time and space and exploit better the semantic information held by labels. Graph simulation and its extensions are considered nowadays the best fit techniques for GPM in many of the emerging applications like analyzing associations in social networks [58].

2.2 Relaxed Graph Pattern Matching

In this section, we review graph simulation and the other models proposed as an alternative to structural GPM. Models in this category aim to capture more similarities by relaxing the matching constraints imposed by structural GPM, which may result into answers that do not have the same structure of the query. This way of defining similarity is highly needed in the current real-world applications such as plagiarism detection. For example, Chao *et al.* proposed an approach based on graph pattern matching to detect plagiarism in programs having thousands of lines of code [52]. They use the program dependence graph to find similar graphs in an available code base.

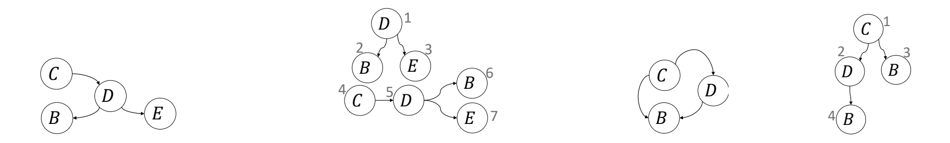
2.2.1 Graph simulation [39]: Graph simulation (GSIM) is a flexible model that allows one query vertex to be mapped to multiple data vertices. It maps separately every edge from the pattern graph to edges in the data graph resulting into one-to-many matching as an output. A vertex v in the data graph $G(V, E, f)$ is mapped to a vertex u in the query graph $Q(V_q, E_q, f_q)$ if they have the same labels and the vertices on the outgoing edges of u have matches in among the children of v. In a formal way: given a data graph G and a pattern graph Q, a binary relation $R \subseteq V_q \times V$ is said to be a match if:

(1) $\forall\,(u, v) \in R, f(u) = f_q\,(v)$,
(2) $\forall(u, u') \in E_q, \exists(v, v') \in E$ such that $(u', v') \in R$.

Graph G matches pattern Q via graph simulation, if there exists a total match relation R where $\forall u \in V_q, \exists v \in V$, such that $(u, v) \in R$. It was proved that a total match can be found in polynomial time (quadratic time) using a refined version of the first algorithm of graph simulation, called HHK [39]. Another algorithm for graph simulation was proposed in [21].

Given the data graph G_1 in Figure 1b and the pattern graph Q_1 in Figure 1a, the application of graph simulation returns a match set R containing all the vertices of G_1 except three vertices $\{11, 27, 34\}$, as it is shown in Figure 3. We notice that several vertices appearing in R should not have been conserved, notably $\{10, 33\}$.

Graph simulation may appear like inexact graph pattern matching, but this is not true. Error-tolerant approaches allow mismatches while graph simulation does not, i.e. the answers returned by graph simulation respect the constraints defined by the model when inexact matching tolerates incorrect matches to some degree. Besides its low complexity, graph simulation is a natural fit for performing graph matching on a fragmented data graph [21]. However, its weak part is failing to capture topological similarities between matched graphs, since a disconnected graph can be correctly set as a match for a connected one (a query example is given in Figure 4a and its answer in Figure 4b), and a cyclic pattern can be matched to tree subgraphs (a cyclic pattern example is shown in Figure 4c and its answer in Figure 4d).



(a) Connected query (b) Disconnected answer (c) Cyclic query (d) Tree answer

Fig. 4. An example showing the limits of graph simulation.

2.2.2 Dual simulation [53]: It considers parent relationships in addition to the child relationships imposed by GSIM. A vertex v from $G = (V, E, f)$ is mapped to a vertex u in $Q = (V_q, E_q, f_q)$, if they have the same labels and the vertices on their incoming and outgoing edges are also mapped together. Formally, a data graph G matches a pattern graph Q via dual simulation if there exists a binary match relation $R \subseteq V_q \times V$ such that:

(1) $\forall(u, v) \in R$, $f_q(u) = f(v)$, i.e. u and v have the same label,
(2) $\forall u \in V_q, \exists v \in V$ such that $(u, v) \in R$ and:
 (a) $\forall(u, u') \in E_q, \exists(v, v') \in E$ such that $(u', v') \in R$ (child relationship),
 (b) $\forall(u'', u) \in E_q, \exists(v'', v) \in E$ such that $(u'', v'') \in R$ (parent relationship).

The results of evaluating dual simulation (DSIM) on the example of Figure 1 are shown in Figure 5. It removes vertices $\{10, 33\}$ from the match set returned by graph simulation. The dual match graph is the induced subgraph containing only vertices of the data graph G_1 that are already in the dual match relation. For example, even though vertices 26 and 28 belong to the dual match set, the edge linking them is not in the dual match graph because there is no edge with labels (C, B) in Q_1.

Considering that a connected graph is considered an answer to a certain query if and only if $\forall u \in V_q, \exists v \in V$ that matches u. Dual simulation brings duality into graph simulation i.e., it eliminates cases when disconnected graphs are considered answers to connected graph queries (back to the example in Figure 4, the partial answer composed of vertices $\{1, 2, 3\}$ in Figure 4b is not valid as an answer on its own to the query in Figure 4a in the case of dual simulation due to the absence of label C).

However, dual simulation does not take into account the data locality. Data locality indicates the scope in which we should look for answers to our query. It could be the distances between vertices in the resulting graph or its size. It is obvious that dual simulation does not take into account data locality since cycles in the query graph can be matched to very long cycles in the data graph. The data locality problem becomes relevant when the returned answers are very large, thus difficult to analyze and also when the distances between data vertices are not kept, making the relationships between distant vertices lose their meaning. That is why this model was reinforced by introducing strong simulation (SGSIM).

2.2.3 Strong simulation [53]: This model improves dual simulation by adding data locality, i.e. matches are only searched in a ball b of center v and having as radius d_q, the diameter of the pattern graph Q. The subgraph of G contained in this ball is denoted $\hat{G}\left[v, d_q\right]$. Formally, G matches Q via strong simulation, if there exists a vertex $v \in V$ such that:

(1) Q matches $\hat{G}\left[v, d_q\right]$ with maximum dual match set $R^b_{d_q}$ in ball b of radius d_q,

(2) v is member of at least one of the pairs of $R^b_{d_q}$.

For every vertex v in the data graph, the connected part of the result match graph of each ball with respect to its $R^b_{d_q}$ containing v is called a maximum perfect subgraph (Max-PG) of G with respect to Q. By proposing a cubic time algorithm, the authors proved that strong simulation not only succeeds to capture more topological similarities, but it also keeps the same polynomial time complexity as graph simulation.

Applying strong simulation to the example given in Figure 1 eliminates matches that fall within the long cycle composed of $\{18, 19, 20, 21, 22, 23, 24, 25\}$, because the distance between vertices 18 and 25 is greater than the pattern's diameter, which is equal to 4. The maximum match set contains the vertices $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 16, 26, 28, 29, 32\}$ as shown in Figure 6. However, strong simulation is limited in terms of scalability since it incurs very long processing time and memory consumption for locality balls creation, especially when the data graph is very large.

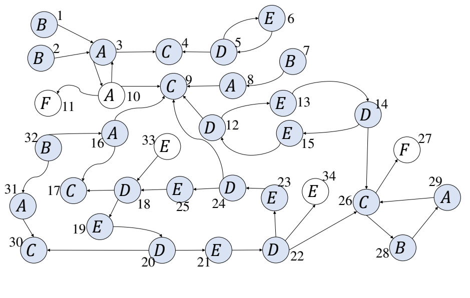


Fig. 5. Results for DSIM are colored in blue.

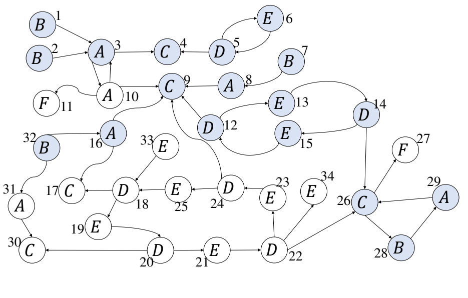


Fig. 6. Results for SGSIM are colored in blue.

2.2.4 Strict simulation [26]: It improves strong simulation with a revised definition of locality which enables achieving better quality of results with performance enhancements. Locality balls are created from the members of the dual match graph instead of generating them directly from dual match set in the initial data graph as it is the case in strong simulation. The difference is that in strong simulation the balls may contain data vertices that are not in dual match while in strict simulation the balls contain only members from the dual match graph. Formally, given a data graph $G = (V, E, f)$ and a query graph $Q = (V_q, E_q, f_q)$ such that $G_d = (V_d, E_d, f_d)$ is the dual match graph. G matches Q via strict simulation if there exists a vertex $v \in V_d$, such that:

(1) $\hat{G}_d[v, d_q]$ matches Q where $\hat{G}_d$ is a subgraph of G_d centered at v with radius d_q,
(2) v is member of the resulting Max-PG.

Strict simulation (STSIM) results into better efficiency by decreasing the size of the locality balls, while obtaining better matches. The authors proposed an algorithm for calculating the Max-PGs in cubic time. When applying this model on the example of Figure 1, it results into the same match graph as strong simulation. However, we notice that the size of the balls created is smaller than in the case of strong simulation because the latter contains also non candidate vertices, the results are shown in Figure 7. We notice that the final Max-PGs are the same as in strong simulation, but this is only valid for this particular example. Even though strict simulation improved strong simulation, it still incurs considerable processing time and may result into duplicate answers. Actually, two adjacent vertices will most likely give the same Max-PG, a limitation that motivated Fard *et al.* to propose tight simulation (TSIM) in [25].

2.2.5 Tight simulation [25]: It is based on strict simulation where only the most important vertices of the query graph are considered in locality balls creation. If strong simulation generates a ball centered at each data vertex, and strict simulation creates a ball for each member of the dual match graph, then tight simulation picks only data vertices candidate to the query vertex with the highest selectivity score. The selectivity score here is a measure computed for each query vertex based on its degree and label frequency. This measure allows us to choose the most important vertex among the graph vertices. Tight simulation reduces the number of locality balls, resulting in shorter processing time, because the size of intermediate results is reduced and duplicated partial answers are minimized. It is formally defined as follows: Given a pattern graph $Q = (V_q, E_q, f_q)$ and a data graph $G = (V, E, f)$. G matches Q, if there are vertices u in Q and u' in G such that:

(1) u is a center of Q with highest defined selectivity,
(2) $(u, u') \in R_D$, where R_D is dual match set between Q and G,
(3) Q matches $\hat{G}_D[u', d_q]$ where $\hat{G}_D$ is a ball extracted from G_D (result of DSIM) with radius d_q,
(4) u' is member of the result Max-PG.

Among the centers of the query graph, the authors pick the one with the highest ratio of its degree to its label frequency ($\max_{u \in C} \frac{deg(u)}{freq(f_q(u))}$, where C is the set of query centers). The locality balls are created with a radius equal to the eccentricity of the query graph. They proposed a distributed algorithm with cubic time complexity for finding all the Max-PGs with respect to TSIM [25].

We apply TSIM on the example of Figure 1, the list of eccentricities for each query vertex is $\{(A : 3), (B : 4), (C : 2), (D : 3), (E : 4)\}$. There is only one center in Q_1 having a minimum eccentricity equal to 2 which is C. The data vertices $\{4, 9, 17, 26, 30\}$ are part of the dual match graph between Q_1 and G_1. Hence, according to tight simulation, we only create balls centered at these vertices with radius equal to 2. Then, for each ball, we verify the dual match relation between the ball members and the query vertices. The ball centered at 4 gives a match set containing the following data vertices: $\{1, 2, 3, 4, 5, 6\}$. The ball centered at $\{9\}$ gives us the same previous match set. Then, the balls centered at 17, 26 and 30 result all in zero matches. We notice that tight simulation here gives the exact same result as subgraph isomorphism (results are shown in Figure 8), while it can be computed in cubic time. The limitation of tight simulation is that it is too strict for some applications of GPM where zero answers are not welcomed.

2.2.6 Bounded simulation (BSIM)[22]: This model is an extension of graph simulation that came as a revision for the traditional subgraph isomorphism model, that falls short of capturing similarities in the actual emerging applications, e.g. detecting communities in social networks. A bounded path p in G is a sequence of vertices $v_1, v_2, \ldots, v_n \in V$, such that $\forall i \in \{1, \ldots, n-1\}, (v_i, v_{i+1}) \in E$. Here, a data graph is defined as $G = (V, E, f_A)$, such that f_A is used instead of the labeling function f.

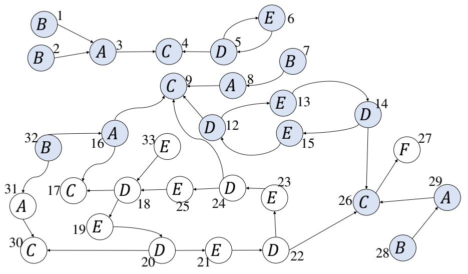


Fig. 7. Results for STSIM are colored in blue.

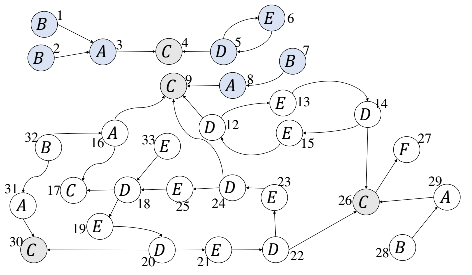


Fig. 8. Results for TSIM are colored in blue while the selected centers are gray.

For each node u in V, $f_A(u)$ is a tuple of attributes $(A_1 = a_1, \ldots, A_n = a_n)$. Furthermore, a pattern graph is given as $Q = (V_q, E_q, f_v, f_e)$ where two functions f_v and f_e are introduced. First, $f_v(u)$ is a conjunction of predicates on the attributes of u. f_e is another function defined on E_q to limit the length of paths matched to each edge of E_q ($f_e(u, u') = \{k, *\}$ where * means no requirement on the path length). A data graph G matches a pattern graph Q via bounded simulation if there exists a binary relation $R \subseteq V_q \times V$ such that:

(1) $\forall u \in V_q, \exists v \in V$ such that $(u, v) \in R$,
(2) $\forall (u, v) \in R$:
   (a) The attributes $f_A(v)$ of v satisfy the predicate $f_v(u)$ of u,
   (b) $\forall (u, u') \in E_q$, (u, u') is mapped to a non-empty, bounded path p of length k from v to v' in G such that $(u', v') \in R$ and $len(p) \leq k$ if $f_e(u, u')$ is a constant k.

Graph simulation is a special case for bounded simulation where only edge-to-edge mappings are allowed. In the same paper, the authors proposed an algorithm for computing bounded simulation that runs in cubic time. Incremental algorithms for bounded simulation were also proposed for the case of a continuously updated data graph.

We present in Figure 9 another example of, respectively, a pattern graph Q_2 and a data graph G_2. By applying bounded simulation, we get the edge (B, A) mapped to the edge $(2, 1)$, edge (A, C) mapped to the path $(1, 4, 6)$ of length 2, (C, D) mapped to paths $(6, 5)$ and $(6, 3, 5)$. Also, the edges (D, E) and (E, D) are mapped respectively to $(5, 7)$ and $(7, 6, 5)$. Note that (D, E) cannot be mapped to the path $(7, 6, 3, 5)$ because this latter is of length 3, which does not satisfy the constraint on the edge (D, E) in Q_2. The result of applying bounded simulation is shown in Figure 9c.

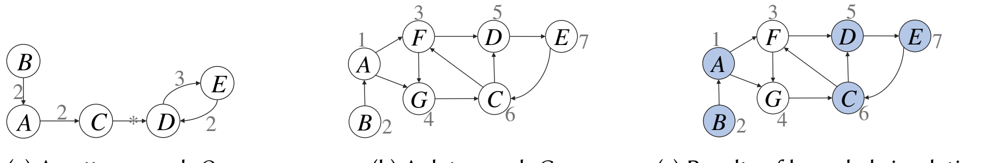

(a) A pattern graph Q_2. (b) A data graph G_2 (c) Results of bounded simulation

Fig. 9. An example of a labeled data graph, a query graph and the results of bounded dual simulation where matched vertices are colored in blue.

2.2.7 *Relaxation simulation [31]:*

An extension of graph simulation that allows partially absent vertices with the condition of substituting them with their children and/or parents. To avoid the problem of zero answers, the authors relax the constraints imposed by graph simulation and dual simulation with respectively two models: unidirectional relaxation simulation (URS) and dual relaxation simulation (DRS). Given a data graph $G = (V, E, f)$, a pattern graph $Q = (V_q, E_q, f_q)$ and a matching relation $R \subseteq V_q \times V$ with $(u, u') \in R$. If GSIM requires all children of $u \in V_q$ to be matched with some or all the children of $u' \in V$, then URS relaxes this constraint by accepting also that grand-children of u' replace their absent parents if they match the children of u. The same goes for DRS such that in addition to children constraints, if there are no parents to u' matching some parents of u', the grand-parents of u' that respect the matching constraints are considered in the final match relation.

2.2.8 Multi-constrained simulation (MCS)[76]: This model addresses the problem of multiple constraints on edges by providing a non-trivial extension to bounded simulation. As opposed to bounded simulation that supports only one constraint on the edge, i.e. the path length, multi-constrained simulation allows for multiple constraints on the edge attributes. A multi-threading heuristic algorithm was proposed for performing MCS, having the name M-HAMC.

2.2.9 Limited simulation (LSIM) [19]: A model that considers the attributes on both vertices and edges. It uses a similarity measure between vertices called *k-similarity*. Given a data graph $G\,(V, E, f_V, f_E)$ and a pattern graph $Q\left(V_q, E_q, f_{Vq}, f_{Eq}, w\right)$ where (i) f_V and f_{Vq} are the respective vertex labeling functions associated to G and Q, (ii) f_E and f_{Eq} are the respective edge labeling functions associated to G and Q, (iii) w is a weight function that maps each $u \in Q$ to an element in $\mathbb{N} \cup \infty$. This weight is defined based on the application domain, it could be for example the diameter of a circle of authority of a manager in a social network. *k-limited similarity* is a relation between two nodes v in G and u in Q denoted by $v \geq_k u$. Formally, v is *k-limited similar* to u if: (i) $f_V(v) = f_{Vq}(u)$ when $k = 0$, (ii) $v \geq_0 u$; $\forall e_q = (u, u') \in Eq, \exists e = (v, v') \in E$ with $f_{Eq}\left(e_q\right) = f_E(e)$ and $v \geq_{k-1} u$ when $k > 0$. G matches Q via LSIM if there exist a node mapping relation $R_v \subseteq V_q \times V$ and an edge mapping relation $R_E \subseteq E_q \times E$ such that:

(1) $\forall\,(u, v) \in R_v, v \geq_{w(u)} u$,

(2) $\forall\,(e = (u, u'), e' = (v, v'))$ in R_E, we have $f_{Eq}\,(e) = f_E\,(e')$, $(u, v) \in R_V$ and $(u', v') \in R_V$.

This model is better suited for applications in social media since it considers also the types of vertices and edges. The authors proposed also a polynomial algorithm to evaluate LSIM.

Furthermore, we find other models that are derived from GSIM or BSIM. First, we have surjective simulation [75] (SJS) that extends BSIM with constraints on nodes with multiple labels, such that nodes from the query and the data graph are mapped if they share a minimum number of common labels. Second, time-constraint simulation [104] (TCS) also revises BSIM for temporal graphs. Lastly, taxonomy simulation [50] (TXS) relaxes the label constraints in GSIM by allowing the mapping of different labels l in Q and l' in G if and only if l' is a descendent of l in the taxonomy hierarchy tree T that defines existing relationships among the graph labels.

Summary

To sum up what has been presented in this section, graph simulation was the first model that came to relax the matching constraints imposed by subgraph isomorphism. Three extensions to this model were proposed, bringing a new definition of similarity; dual simulation, that captures more topological structures (connected components and undirected cycles), bounded simulation that matches edges with paths of bounded length, and limited simulation that considers a new similarity information called *k-similarity*. Multi-constrained simulation and surjective simulation are extensions to bounded simulation that support multiple labels. Time-constraint simulation is another extension designed for temporal graphs. On the other hand, relaxation simulation relaxes further the constraints of graph simulation and dual simulation and hence allows for larger results. Similarly, taxonomy simulation allows mapping even nodes of different labels but that are related in terms of taxonomy. Strong simulation is an extension to dual simulation that imposes locality to reduce the answer size. Finally, strict and tight simulation shrink the size of the resulting match graphs to keep only answers closer to subgraph isomorphism in terms of quality.

In Figure 10, we illustrate how these extensions, based on graph simulation, make the constraints more or less relaxed or more stringent. The schema does not consider limited simulation as an extension since it redefines the notion of similarity rather than adding or omitting some constraints from the original model.

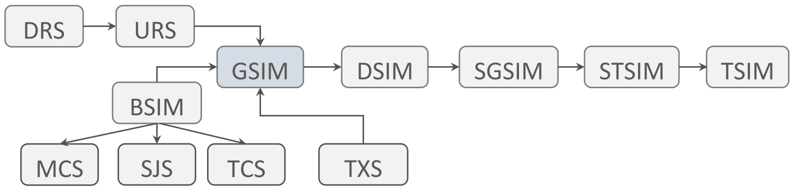


Fig. 10. Extensions of GSIM where the direction of the arrows indicates whether this extension came to restrict further the constraints imposed by GSIM or to relax them when the arrow is reversed (e.g. URS).

The following propositions were given to describe the relations between some GPM models:

(1) If Q matches G via subgraph isomorphism, then Q matches G via tight simulation.
(2) If Q matches G via tight simulation, then Q matches G via strict simulation.
(3) If Q matches G via strict simulation, then Q matches G via strong simulation.
(4) If Q matches G via strong simulation, then Q matches G via dual simulation.
(5) If Q matches G via dual simulation, then Q matches G via graph simulation.

The propositions above were proven in [25, 26, 53]. In Figure 11, we depict this inclusion between the different models. Based on the application needs, we should decide to opt for graph simulation when interested in flexibility and short response time. Moving from graph simulation toward tight simulation and subgraph isomorphism will result in longer response time, but will allow us to get better quality results compared to dual simulation and graph simulation.

However, even with the progress made in reducing subgraph matching complexity from exponential to a polynomial time, massive graphs containing billions of nodes and trillions of edges cannot be stored or processed in a single machine. The following section discusses this issue and presents the different mechanisms used to handle graphs distributed over multiple machines.

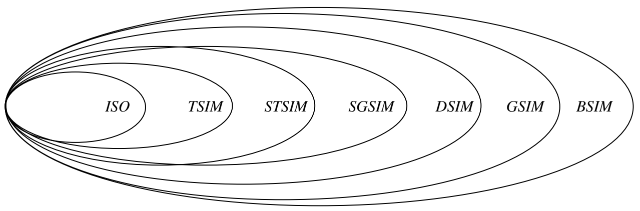


Fig. 11. The inclusion relation between results obtained by the different models discussed.

3 Distributed big graph processing

Huge amounts of raw data that need to be analyzed are collected every minute. It is common to partition and distribute them over multiple machines, so that queries can be efficiently evaluated in a distributed fashion. Therefore, sequential algorithms used in finding subgraph isomorphism or graph simulation need to be adapted for this context. Moreover, efforts are made to scale up the simulation-based algorithms to run on distributed architectures and benefit from the computing power resulting from the use of computing clusters. In this section, we provide an overview of the paradigms used to execute treatments in a distributed environment.

3.1 Distributed architectures

The most common distributed architecture used is the master-slave architecture. The master is a machine of the cluster executing tasks of coordination between different slave machines (a.k.a. workers) that are responsible for parallel computations. The master machine also performs some monitoring tasks to maintain a green state of the cluster so that all workers keep running smoothly. According to Yan *et al.* in [95], a cluster is composed of mainly four components starting from local storage, to distributed storage, communication layer and finally the computing module. (this architecture is sketched in Figure 12).

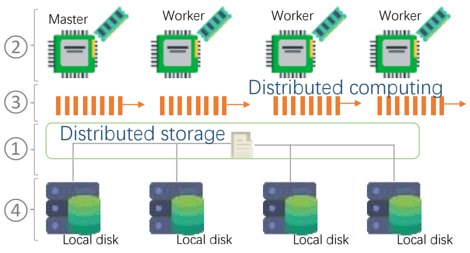


Fig. 12. Components of a distributed architecture.

Component (1): Data graphs reside physically on distributed storage. HDFS [6] and Google's file system (GFS) [34] are two of the most popular distributed file systems used currently by several high-scale graph processing systems. These two file systems share the same aspects but do have some differences including the block size, node division and the deletion strategy adopted.

Component (2): This part of the distributed architecture is responsible of loading graphs from the distributed file system to memory. In a parallel manner, the computation module performs the computations on each machine.

Component (3): The communication layer defines the technique used to exchange messages between machines in the cluster, e.g. by using the paradigms MPI or RPC.

Component (4): Each machine is responsible of its own local storage, which can be used in out-of-memory systems that need to offload data graphs from memory to local disks during computations.

3.2 Programming models for distributed graph processing

A programming model specifies an abstraction of how a parallel algorithm is designed and executed. MapReduce [17] is one of the leading paradigms employed for processing large data sets. However, algorithms that deal with graphs are different since they incur poor data locality and little processing on every vertex of the graph, which results into MapReduce job spending most of the time in I/O operations. This limitation of MapReduce makes it unsuitable for handling distributed graphs. As a result, Pregel framework was proposed in [55], along with other paradigms that have emerged later challenging the programmer to think like a vertex. In Figure 13, we give a classification of the commonly employed programming models for each layer in a big graph processing system. These models are classified based on three criteria namely execution model, communication and timing. Each one offering two or more ways of defining how certain tasks are performed by the system.

3.2.1 Timing: The timing defines the synchronization type used to schedule tasks in parallel. Thus, a parallel program can be either synchronous or asynchronous. In synchronous programming models like Bulk Synchronous Parallel (BSP) [86], programs are composed of super steps separated with synchronization points. After finishing a super step, a process is blocked until all the other processes, executed in parallel, terminate their tasks before moving to the next super step. Asynchronous programming models do not set any constraints on the order in which processes are executed, every process can send any message or execute a task whenever CPU and bandwidth are available.

Considered as the most common model used for parallel processing, the bulk synchronous parallel was proposed by Valiant in 1990 [86]. A BSP computer is composed of three parts: (1) Components that can perform computations locally, (2) A router for exchanging messages between each pairwise of such components, (3) A module allowing for synchronization of these components. Every component carries out a computation in a sequence of global super steps: (a) concurrent computation, (b) communication and (c) barrier synchronization. Figure 14 illustrates the functioning of BSP.

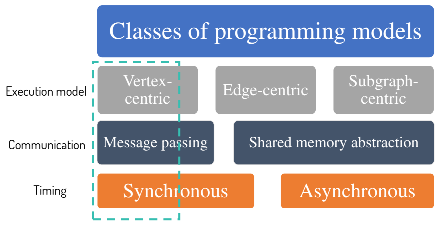


Fig. 13. Programming models for distributed graph processing.

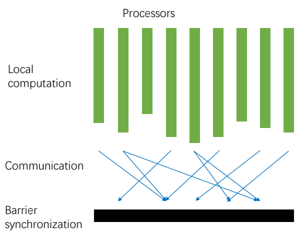


Fig. 14. The BSP model.

3.2.2 Communication: This criterion defines the way messages are exchanged between distant machines in the cluster. We use either a message passing approach or shared memory abstraction. In the first communication type, a message with its ID, containing information about the source, destination and the needed information is sent from one machine to another while in shared memory abstraction, vertices (programs) are considered as shared variables that are accessed directly from any machine in the cluster.

3.2.3 Execution model: The execution model determines the way a data graph is seen and iterated over. It could be vertex-centric, edge-centric or subgraph centric.

3.2.3.1 Vertex-centric: The vertex-centric concept was introduced by Google in their paper on Pregel system [55]. A graph algorithm is executed from the perspective of a vertex that can perform a list of operations as getting its ID, getting/setting its value or getting/counting its edges. A vertex is analogous to a process in the BSP model. We also define the notion of neighboring processes, i.e. a process can send or receive messages only from its neighbors (An illustration of the vertex-centric model is given in Figure 15).

Pregel defines the way a graph processing algorithm is executed in parallel. It is based on the BSP model, and adopts a message passing mechanism to communicate between vertices. The data graph vertices are partitioned among several workers where each vertex executes a *compute()* function, which is the same for all the vertices. A vertex can have an active or inactive state. A Pregel program consists of a number of synchronized iterations conducted in parallel by several vertices, and that can terminate only if all the processes vote to halt and there are no pending messages waiting to be achieved.

Since its apparition, this model has revolutionized the world of graph processing. Ever after, several open source frameworks were developed, adopting the same model while providing improvements on the communication mechanism, load balancing or fault tolerance. For example, Giraph [13] is the most popular framework with a large community working on it, including engineers from Yahoo, Facebook, LinkedIn and Twitter. This system enhanced the initial model of Pregel with three improvements by first, introducing parallelism at a fine grain using multi-threading on each worker. Also, to reduce the communication cost, Giraph serializes Java objects before sending them from one machine to another. Finally, to reduce the out-of-memory problems that occur generally with longer super-steps with too many messages to send, Giraph adopts splits large super-steps into smaller ones. We can cite other systems like GPS [69], MocGraph [106], Mizan [44], GraphX [94], Pregel+[97] and GraphD [100] that propose an improvement of Pregel's model in a way or another.

3.2.3.2 Edge-centric: In this class of programming models, the parallel program iterates over the graph edges. Therefore, the computing unit is an edge that can fulfill some basic operations, e.g. get/set its value or ID, or get the value of its target vertex. An edge can also communicate with its neighboring edges by sending and receiving messages (see the illustration provided in Figure 16). This paradigm is well-suited for data flows, and X-Stream [68] is one of the frameworks adopting it.

3.2.3.3 Subgraph-centric: Unlike vertex-centric programming where vertices cannot know whether a vertex is located at the same machine or not, subgraph-centric programming considers the fragment of vertices located at the same machine as one computing unit. Here, communications between vertices of the same subgraph are not delayed to the next super-step but executed immediately. Such improvements result to better performance in computing time, number of messages and total super-steps (see illustration of this model in Figure 17). In [95], Yan *et al.* recommended this programming model in solving graph pattern matching. Several frameworks support this paradigm including Giraph++ [83], GoFFish [77] and Blogel [96], but they are basically vertex-centric.

On the other hand, there are subgraph mining systems that are dedicated for specific graph applications like subgraph matching, triangles finding or frequent subgraphs listing, e.g. Arabesque [82], Quegel [98], G-miner [12], Rstream [91], Fractal [18] and G-thinker [99].

The most recent one is G-thinker which differs from previous frameworks that are known to be IO-bound, i.e. their processing time is mostly dominated by I/O operations. G-thinker is CPU-bound thanks to always keeping CPUs busy by switching to tasks that have their data ready instead of waiting for the current task to receive its requested data through the network. It handles two data tables: one for storing the graph available locally and another cache table for remote vertices. The latter is shared by the different subgraphs without the need to duplicate data within each subgraph. A subgraph is handled by one thread independently and hence no synchronization is needed between the different threads. Each thread, called *comper*, will construct its subgraph and work on it to evaluate subgraph isomorphism or to count the number of local triangles, etc.

GRAPE [24] is a system that is different from the above mentioned ones. It allows parallelization of sequential graph computations without adopting a TLAV paradigm. Given a data graph G that is composed of k fragments $\{F_1, F_2, \ldots, F_k\}$, GRAPE assigns each fragment to a processor. It uses partial evaluation to solve the graph problem on one fragment of the data graph, i.e. a processor P_i evaluates the problem based on the known data available in fragment F_i and exchanges messages based on the BSP model to get the missing information updated. Then, each processor updates its partial answer based on the received updates. When no processor emits new messages, an assemble procedure is triggered at a coordinator machine to combine the partial answers.

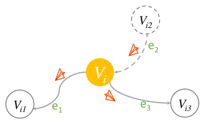


Fig. 15. Vertex-centric.

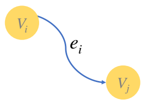


Fig. 16. Edge-centric.

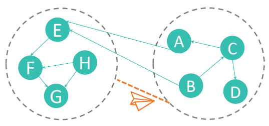


Fig. 17. Subgraph-centric.

Summary

Each one of the programming models presented in this section is well adapted for designing algorithms that respond to specific problems but not to others. For example, the vertex-centric programming model is adapted for algorithms based on communications where the vertex-program does not need to access neighborhood information, e.g. SSSP, Page Rank. On the other hand, the subgraph-centric programming model is better suited for algorithms where the vertex-program requires its neighborhood information in addition to its local information to update its local context, which is the case for GPM. Finally, the edge-centric model is used to design flow-centric algorithms where the main processing is executed on the graph edges and not on its vertices.

Other programming models also exist, but they are not as commonly used as the ones discussed above. For example, the partition-centric model is a special case of subgraph-centric where all nodes and edges that are located in the same machine are considered as one computing unit. Moreover, neighborhood-centric is another model that considers that the scope of a single vertex is composed of its local state and the states of its multi-hop neighborhood. We also find other programming models similar to vertex-centric such as Scatter-Gather and Gather-Sum-Apply-Scatter (GAS). These two models are slightly different from vertex-centric. For example, Scatter-Gather defines for each vertex two functions Scatter and Gather. The Scatter function defines and sends messages to its neighbors, while the Gather function allows a vertex to read the received messages and edit its local information. Both functions are executed in the same iteration respecting the BSP model. A comprehensive survey of the programming abstractions used for distributed graph processing can be found in [42].

4 Distributed Graph Pattern Matching approaches

After introducing the context of distributed graph processing and the different models and algorithms proposed to solve GPM, in this section, we present at first the distributed techniques proposed for structural GPM, then we will discuss the recent works addressing distributed relaxed GPM. Some definitions and performance evaluation criteria related to distributed algorithms and data graphs are necessary before discussing the different approaches.

Definition 4.1 (Distributed graph). *A distributed data graph is a graph G partitioned into k fragments $(F_1, F_2, \ldots, F_k)$, each fragment is located on a separate machine. $(V_1, V_2, \ldots, V_k)$ is a partition of $G(V, E, L)$ if and only if: (1) $\cup_{i=1}^{k} V_i = V$ and (2) for any $i \neq j \in [1, k] V_i \cap V_j = \emptyset$.*

Fragments of a data graph can depend on each other because of the crossing-edges between vertices in different partitions and such vertices are called boundary nodes.

Definition 4.2 (Fragment). *A fragment F_i is denoted as $(G[V_i], B_i)$ where $G[V_i]$ is the subgraph of G stored at machine i and B_i is the set of all boundary nodes in this subgraph. We denote B_V as the set of all boundary nodes and B_E as the set of all crossing-edges.*

The partitioning strategy we have just described is commonly used for distributing graphs in the context of GPM and it is called Edge-Cut partitioning. The example shown in Figure 18 has three fragments $\{F_1, F_2, F_3\}$, the global boundary set B_V is composed of the vertices $\{3, 5, 8, 9, 11\}$.

The alternative to Edge-Cut strategy is the Vertex-Cut partitioning where edges of the data graphs are split between the different machines. In this case, we have crossing vertices that are shared between multiple machines.

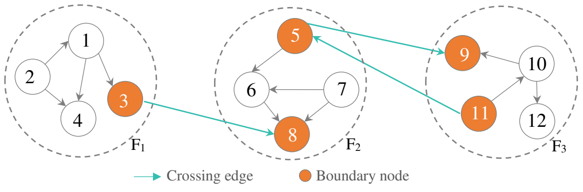


Fig. 18. Fragmentation of a data graph based on edge-cut partitioning strategy.

When evaluating a subgraph matching query, fragments of the data graph can be processed separately. Actually, the real challenge here is how we should evaluate GPM for the boundary nodes. Because, as we mentioned earlier in Section 2, to verify if a data vertex is a match to a certain query vertex, the matching information of its neighborhood is necessary in the evaluation of the model's constraints. Several answers to this question have been proposed in the literature, we present in this section some of the most relevant.

To find matches of a query graph in a data graph that is fragmented over multiple sites, a general algorithm follows these steps: A coordinator machine receives the pattern graph Q, then transfers it to the workers which, in turn, perform a local evaluation of the query resulting into a partial matching set. Such partial matches are then sent to the coordinator to combine them into one final result. Different mechanisms and techniques are involved in the evaluation of GPM for the set of boundary nodes. This can be achieved by either communicating intermediate results between workers or even shipping parts of the fragmented graph from one worker to another. We notice that some GPM propositions rely on the existing frameworks of distributed graph processing while others simply provide a *from-scratch* implementation of the distributed algorithms.

Different evaluation criteria are used to measure the performance of distributed graph algorithms. Firstly, we have *data shipment* as the amount of data exchanged between distinct machines during the execution of the algorithm. The second criterion is *visit times*, the number of messages that have been exchanged between workers in the cluster. Finally, there is the *make-span* (a.k.a. response time) that refers to the duration between receiving the query and finding the result.

4.1 Distributed Structural Graph Pattern Matching

In this section, we review the latest works proposed for finding subgraph isomorphism in a distributed environment. We classify them based on the timing and execution model applied. The studied works are divided into four classes: synchronous master-slave, asynchronous master-slave, synchronous vertex-centric and asynchronous vertex-centric. The works presented under this section are considered structural because they aim to find answers that have the exact same structure as that of the input query.

4.1.1 Synchronous Master-slave: This model synchronizes the computation through super-steps separated by communications between workers or between workers and a coordinator.

Peng *et al.* [60] adopt partial evaluation to find matches locally to each partition of the distributed data graph. For the cross matches that include data vertices between multiple fragments, each worker evaluates ISO based on its local information and leaves some matches unanswered. Then, the partial matches are assembled on a coordinator site to get cross matches. The assembly algorithm collects all the partial matches, joins them by starting with two partial matches from connected fragments and considers a new match at a time. If the match is complete, another assembly is started, otherwise, other partial matches are joined to form a final answer. The authors introduce a partitioning of the partial matches to avoid unfruitful joins. Partial matches that share an internal data vertex are collected in the same partition and the assembly is made between different partitions. Another distributed assembly that follows a BSP Master-slave was proposed: in the first super-step the partial-evaluation part is conducted locally, then partial matches are shared between workers to assemble the cross-matches in a distributed way such that an intermediate result is communicated each super-step. A one-direction communication is used to avoid duplicate answers.

To parallelize TurboISO on a fragmented data graph. Fan *et al.* [24] expand each fragment F_i with data vertices from other fragments such that subgraph isomorphism can be evaluated locally on each worker. Subgraph isomorphism is determinable locally for a fragment F_i if and only if, for every vertex $v \in F_i$, v can reach all its neighborhood within d hops such that d is the diameter of the query graph. This work is based on GRAPE that offers the three functions: partial evaluation (PEval), incremental evaluation (IncEval) and assemble. In PEval, each worker gathers the d-neighborhood of its boundary nodes. Then, it sends them to the coordinator that, in turn, sends each d-neighborhood to where its center resides. IncEval is simply a sequential implementation of TurboISO. Finally, the assemble function takes the union of the subgraph isomorphism matches returned by each worker. The limitation of this approach is that large parts of the data graph may end up duplicated among workers, especially for small diameter graphs and when dealing with complex queries of large diameters.

4.1.2 Asynchronous Master-slave: The main difference between an algorithm using a master-slave execution approach and the one adopting subgraph-centric is that the latter is fully centric while the first one involves a coordinator for exchanging data between different workers or to orchestrate the graph matching process. Both consider the subgraph as one computing unit on which all the operations are made at the same time.

COSI [7] relies on the RDF format[1] for describing both the data graph and query graph. The architecture used for evaluating subgraph isomorphism consists of a set of workers that communicate directly with each other and a master node behaving as an interface with the end user. COSI also uses a partitioning strategy that minimizes the edge cuts between different sites of the distributed architecture. It is based on the idea of keeping on the same machine every two vertices that are likely to be retrieved together by a random query. The query is received by the master that routes it to one or more slaves. Worker nodes process the query locally and send back a complete subgraph matching answer to the master node. The local algorithm uses a local index for retrieving neighbors of a data vertex. It defines an order for processing the query variables, finds data candidates of the current query variable, substitutes it with a data vertex v then searches for candidates of the next variable in the set of neighbors of v. It continues executing this operation until no candidates are found in the current partition. At the end of this step, the algorithm verifies if a valid match set is obtained or not to send back an answer to the master. So here, the algorithm explores the tree search in a depth first search fashion where each branch is traversed in parallel to reduce the processing time. This approach for finding subgraph isomorphism performs well in terms of time and data shipment. However, even though neighborhood indexes accelerate and facilitate the access to the data graph vertices, they have generally a super-linear creation time.

In [81], a GPM system developed on top of Trinity [73] was proposed that is based on query decomposition and graph exploration. The aim is to reduce the amount of data shipment involved in the processing and also limit the size of intermediate results by avoiding joins. It includes three

[1]The RDF format stands for Resource Description Framework, which is a graph data model for describing graphs using a set of triples of the form: *subject-predicate-object.*

major steps: *Query decomposition and STwig ordering*, *Exploration* and finally a *Joining phase*. The query graph is firstly decomposed into STwigs (trees of two levels). After that, a selection algorithm is executed to find the best order in which these STwigs will be processed minimizing further the size of intermediate results (the output is an ordered sequence of STwigs $S_1, S_2, \ldots, S_N$). The STwigs sequence is next delivered to each worker which will process it, explore the graph while joining the partial matches obtained for each STwig ($M_1, M_2, \ldots, M_N$), then send back the result R to the Query Proxy. The most important contribution here, is using exploration instead of joining whenever it is possible, which considerably reduces the size of intermediate results. The local evaluation of subgraph isomorphism requires accessing STwig root children located on different workers, and hence involves network communication. Compared to the propositions from previous approaches that rely on indexes, this work has shown performances in response time when handling a graph of one billion nodes. The only index required was intended to accelerate the access to each data vertex by mapping identifiers to their labels.

4.1.3 Synchronous Vertex-centric: Under this category we review synchronous vertex-centric approaches proposed for GPM.

Lighthouse [27] is a system that evaluates subgraph isomorphism and accepts as input a graph query language equivalent to Cypher (the one used by Neo4J [40]). It is an improvement for method proposed in [81]. Similarly, the query graph is decomposed into STwig data structures that will be matched individually using the vertex-centric approach, then, the matching results are joint to build the match result. This work discusses the possibility of simply processing the STwigs in parallel then joining them sequentially. Such optimization may require more memory to hold the intermediate results but will definitely reduce the number of super-steps and hence the query processing time. So here, the main contribution was taking the query decomposition approach of [81] and transforming it into a vertex-centric one and thus, enabling the system to achieve better performance and gain in scalability.

PSgL [74] evaluates subgraph isomorphism based on a parallel traversal of both the pattern graph Q and the data graph G to map vertices in Q with vertices in G. Each data vertex v is first mapped to a query vertex u (selected based on a cost function that minimizes the size of intermediate results). Then, in the next super-step, the mapping is expanded by mapping neighbors of u to the neighbors of v if the mapping satisfies the matching constraints. In this step, u is colored black, its neighbors are gray, and the rest of the pattern is white. In the next super-step, this mapping (tree) is forwarded to neighbors of v that can expand it further. A neighbor v' will expand a gray vertex (that becomes black), its white neighbors (not yet mapped) are now mapped to neighbors of v'. Every mapping is expanded in the same way until all the query vertices are mapped (it returns the subgraph) or there is no way to expand it, i.e. no remaining neighbors of v'. The authors propose heuristics that guide the tree forwarding among the data vertices. This approach that adopts a BSP-based vertex-centric paradigm was implemented on top of Giraph.

In [32], the authors use a Single Sink DAG extracted from an undirected pattern to guide the evaluation of ISO in a distributed data graph. The sink of the DAG is a query vertex that has no outgoing vertices, only incoming vertices. Every data vertex candidate to this sink will be responsible for collecting a subgraph match if it exists. Transition rules are attached to every edge in the DAG, then each data vertex considered a candidate to a non sink query vertex will generate and transfer the received messages that end at the sink data node. The data node accumulates the received messages to form a correct match or drop the messages if there are non respected ISO constraints. Furthermore, to support detection of the same query on an evolving graph, any update event triggers a message broadcast received by data vertices that will verify the transition rules and transfer these messages to their neighbors if the transition rules become unsatisfied. The receiving nodes will do the same recursively until reaching the sink nodes that will update their matching information accordingly.

4.1.4 Asynchronous Vertex-centric: Reza *et al.* [65] proposed recently a distributed algorithm for evaluating subgraph isomorphism on one trillion-edge graphs based on HavoqGT, a platform for asynchronous vertex-centric graph processing that was developed in 2013 [59]. The algorithm is composed of two phases. Firstly, a Local Constraint Checking procedure eliminates aggressively the vertices that do not satisfy the subgraph isomorphism constraints, several iterations can be executed until no vertex is eliminated. Then, a Cycle Checking procedure is executed only for cyclic queries. The first phase can successfully eliminate all the data vertices that do not participate in answering the input query. But when the query graph contains cycles, there can still be remaining vertices that do not verify the whole matching constraints. The idea proposed by the authors consists of checking the existence of cycles based on some cyclic constraints introduced by the user to keep only matches respecting them in the data graph. If the algorithm finds a cycle having a defined length d in the query graph then, each data vertex already matched to a member of this cycle will execute a token passing routine. This routine is used to verify if it is also part of a cycle with the same length in the data graph. The data vertex sends a token to its neighbors in a diameter of d such that if the data vertex does not get back its token at the end of d iterations, it will be eliminated because it is not part of any cycle. This approach showed good results for handling undirected graphs of more than one trillion edges.

PruneJuice [67] uses graph pruning to evaluate ISO. The data graph is pruned based on a set of constraints extracted from the query graph so that every data vertex respecting these constraints will be part of a match. The embedding enumeration can then be performed on the resulting pruned graph having the smallest size possible. These constraints are divided into local constraints such as the vertex label and the neighbor labels and that will be verified by exchanging direct messages between the data vertices. Non-local constraints require walks within the data graph such as path search, cycle checking when a query label is repeated in the pattern or a query edge is part of a cycle respectively. Both types of constraints require a token passing to verify the cycle membership and to verify that two data vertices having the same label have a specific labeled path . Another type of non-local constraints that ensure that the pruned graph contains exactly all the ISO matches is the Template-Driven Search (TDS). The TDS prunes further the data graph by walking through the pruned graph and checking for the union of cycle constraints and the union of repeated label constraints and eliminating every data vertex not satisfying them.

Furthermore, Reza *et al.* [66] use k edit-distance to find approximate matches. Here, a k edit-distance is the subgraphs of the data graph that become isomorphic to the input query upon the deletion of up to k edges from the query. The authors have already defined in [67] a set of local and non-local constraints that help pruning the data graph to keep only exact matches and thus enumerate subgraph isomorphism embeddings. In this contribution, they define a set of prototypes from the query graph (variations of the query graph within distance k) and then, they use the common local and non-local constraints to avoid reevaluating subgraph isomorphism for every single prototype separately. A data vertex that satisfies a non-local constraint for $k = \delta + 1$, will also satisfy this constraint for $k = \delta$. So, every time a data vertex meets specific non-local constraint x, this information is saved in the context of that vertex so that it can be reused in the constraints verification for prototypes of smaller k.

In [78], a vertex-centric approach was presented for performing topology pattern matching (subgraph isomorphism) in a distributed environment composed of a Wireless Sensor Network (WSN). Each sensor in this WSN is analogous to a data vertex having a partial view of its closest neighbors in the network topology. Different from the previous works, this proposition aims to reduce the disk storage and message overhead in a resource-limited context. The authors claim that it is the first proposition providing a fully distributed GPM without any central coordination. There is an initiator node v_0 that starts the matching process and builds a partial match set if possible. Then, it will ask its neighbors to further extend the partial match set with their local views of the graph. When a node receives a request from its neighbor to complete the match set, if it finds a complete match set, it will send back the result to v_0. Otherwise, it will propagate its partial answer to other nodes in the data graph. The authors confirmed that such approach can also be used in contexts other than a WSN. However, the solution was only tested on 400 nodes.

4.2 Distributed Relaxed Graph Pattern Matching

Since the recent works on graph pattern matching are mostly oriented toward reaching higher performance and scaling up the existing algorithms into larger graphs, we will focus on distributed relaxed GPM. Graph simulation along with its induced models have given an evidence of superiority over its conventional counterpart, and this is due to relaxing the structural constraints imposed by subgraph isomorphism. The works discussed here are divided into four categories. First, we have the master-slave execution model with both synchronous and asynchronous timing. Then, we have BSP-based approaches adopting a vertex-centric or subgraph-centric abstraction.

4.2.1 Synchronous Master-slave: We review under this section a work that distributes processing following a master-slave model where the workers communicate only with the master.

4.2.1.1 Graph simulation: Similar to the work presented in [23], graph simulation is evaluated on top of GRAPE [24] by incrementally updating the match set of a query vertex u in each fragment F_i. Then, the matching information of a vertex v in F_i is needed by their original fragments to complete their matching. So, at the end of partial evaluation (PEval), each processor sends a variable $X(u, v)$, with true or false value to indicate whether $u \in Q$ is matched to $v \in G$ or not, to the coordinator. The coordinator will then propagate these variables into the processors that need to complete their matching information. Upon the reception of these variables (in IncEval), each processor will verify the matching constraints and update its local variables accordingly; if a variable $X(u, v)$ switches to false, it will propagate this update by sending a message to the coordinator, which in turn propagates the change to the appropriate processors until there is no update issued. Finally, the coordinator will execute the assemble function which is the union of partial matches generated by each processor.

4.2.2 Asynchronous Master-slave: We present under this section the recent works adopting an asynchronous programming model where the evaluation of the query is triggered on a master and then executed on the different worker machines.

4.2.2.1 Graph simulation: The first work to settle GPM in a distributed environment was published in 2012 [54]. The authors adopted a message-passing approach for solving graph simulation in the context of big graphs. Their solution is based on the following principle: They have proved that evaluating separately the connected components (CCs) of a data graph allows us to find a correct matching set by just computing the union of different partial matches found for each CC. First, each worker receives the query graph and finds candidate vertices for each query vertex in parallel with other workers. After this filtering step, the graph will contain a set of connected components, some of which reside on the same worker while others are distributed over multiple machines. Each worker executes locally the algorithm localHHK (an updated version of HHK algorithm [39]) on the CCs that reside fully on it. Then, it sends back its partial matching set to a coordinator machine, along with the information about vertices in a CC across several machines. The coordinator executes a scheduling algorithm to assign the remaining CCs to a set of workers while minimizing make-span and data shipment at the same time. The assignment is then sent back to these workers that will ship the CCs to their corresponding workers. Assigned workers will perform local graph matching on the received CCs and send back their results to the coordinator. At the end, a set of partial matches for each connected component will be available on the coordinator that builds a final matching set from their union. It is important to mention that the proposed solution lies much more on distributing the data while executing the same algorithms in parallel (data-parallel approach) rather than on the notion of distributed processing.

In [23], Fan *et al.* proposed a distributed algorithm for solving GPM via graph simulation with bounded response time and data shipment. Each worker maintains locally a dependency graph for tracking linked vertices that are situated on other workers. To perform local graph simulation correctly, the boundary nodes and crossing edges are duplicated on both the source and destination machine and treated separately. Here, instead of shipping parts of the data graph from one site to another, a Boolean variable $X_{(v,u)}$ is used to indicate whether the data vertex v is a match to the query vertex u. If the two vertices v and u share the same label, then there must be at least one matching relation between child nodes u_i of u and a child nodes v_i of v to add this tuple to the match set. This constraint is formulated as follows: $X_{(v,u)} = \vee(\wedge X_{(v_i,u_i)})$. To evaluate graph simulation, the algorithm simply propagates the values of these variables inside each worker such that each variable will be written in function of only the variables that relate to virtual nodes (nodes copied from other fragments) in this machine. Then, a message passing is conducted where each machine broadcasts the values of its in-node variables (nodes considered as virtual-nodes in other fragments) following the dependency graph. After receiving new values of variables related to its virtual-nodes, each machine propagates this information among its local variables. Next, it updates them and conduct a new message passing phase if one or more of its in-node variables were updated. The algorithm converges when no more messages are exchanged. The use of a subgraph-centric paradigm while allowing for asynchronous communication made it possible for this system to achieve better response time, compared to the solutions proposed in [26].

4.2.2.2 Strong simulation: In an earlier paper [53], the outlines of a distributed algorithm were given for computing strong simulation. It was the first algorithm proposed in this category but with no performance guarantees. The algorithm simply ships the contents of each data locality ball that is stored across different machines to the machine with the smallest index, then evaluate strong simulation locally. After this step, the workers send their partial results to a coordinator machine that merges them into a final match set.

In [89], the process of finding matches of a pattern graph is split into two phases: An off-line redistribution phase is conducted to make the data graph locally determinable, then an on-line evaluation phase is held for evaluating strong simulation. The main challenge brought by strong simulation in the distributed setting is the data shipment incurred. Indeed, we need to ship data vertices from different machines for each locality ball created. Thus, the larger the query graph, the larger its diameter d_q and the larger the locality balls. As a reflection, for reducing the size of these balls and hence the amount of data shipped, we can think of reducing the diameter of query graphs. The idea on which this paper is based is to set a maximal query diameter d_q, and use it to further optimize data shipment by redistributing the data graph offline. The redistribution is made such that all the balls centered at $v \in V$ will reside on the same machine. An online processing algorithm is conducted at the reception of every query graph Q. The online phase consists of partitioning the query graph into partition-trees (a tree rooted at each query vertex and having as diameter d_q) locally on each worker. Then, *p-match-graph* is computed for every partition-tree, knowing that p-mach here is a revision of graph simulation allowing to find partial results of a query graph locally when using its partition-trees as input. Then the local p-match-graph obtained is sent to a coordinator. This latter merges the local p-match-graphs received into one final p-match-graph, then conducts strong simulation locally to find a final answer to the query. Even though this algorithm achieved controllable data shipment in the online phase, it is not well adapted for highly dynamic graphs. (a redistribution phase has to be executed after each update event). Also, this proposal puts too much effort into reducing the data shipment at the expense of response time, another drawback is that no data shipment was involved between different workers but the algorithm still uses network traffic between workers and master (which may lead to a load-balance problem). Such proposition does not take advantage of the distributed systems and their optimizations.

4.2.3 Synchronous Vertex-centric: We find several works adopting this programming model to evaluate graph simulation, strong simulation, strict simulation or tight simulation. All the studied works adopt a BSP-based vertex-centric abstraction.

4.2.3.1 Graph simulation: We review here three different algorithms of graph simulation.

In [26], the first algorithm to adopt a BSP-based vertex-centric approach to evaluate graph simulation was proposed. Each vertex in the data graph uses two local variables: a match flag and a match set. Upon receipt of the query graph, it sets its matching flag to true if its label is the same as one of the query vertices', and match set will contain identifiers of the similar query vertices. Then, if match flag is true, it will also ask its children for their match sets. In the second super-step, vertices that receive a request while their match flags are true will respond with their match sets. In a third-round, each vertex receiving information from its children will update its match flag and match set according to the received match sets and the query graph, it will then inform its parents of any update made. In the fourth round and beyond, each vertex that receives an updated match set will update its match flag and match set accordingly and inform its parents. This algorithm stops when all the vertices stop sending messages, it takes at minimum 4 super-steps to evaluate graph simulation. The same goes for dual simulation when each vertex will ask for information about its children and parents. Similarly, In the following super-steps, when a match set is updated, the vertex will inform its children and parents about the updates made.

In [70], S2X was proposed for evaluating dual simulation on RDF data, on top of GraphX. Spark allows for data-parallelism thanks to its *Resilient Distributed Datasets* (RDDs) that apply an efficient partitioning strategy permitting to minimize the cut-edges between worker machines. Moreover, graph-parallel computation is made possible with its vertex-centric programming model called GraphX. The query graph consisting of a set of triplets (edges) is broadcast to every data vertex that will verify dual simulation constraints locally. Then, it will exchange its partial match sets with its neighbors to further validate the matching sets built in the first super-step. At the end and after receiving the partial results obtained by each data vertex, a joining phase is executed to construct the final answer. If we compare the contribution made by this paper with the algorithms in [26], here the authors addressed RDF graphs that are characterized by labeled edges and that may have multiple edges between two nodes.

Another recent work proposed two vertex-centric algorithms to evaluate graph simulation for acyclic queries and cyclic queries separately [51]. The first algorithm uses a Boolean variable local to each data vertex which holds the matching information that can be either True, False or Unknown. The algorithm is executed in three different stages: Initialization, Message Broadcast and Match trial. In the first step, matching information for data vertices verifying the match constraints are set to true (the vertices candidates to a query vertex that has no children in the query graph), the remaining vertices set their values to unknown. Then, a message broadcasting step is carried out to propagate these truth values to parents. The vertices having their local matching variables set to True send their labels to their parents and go inactive. On the other side, vertices with unknown values will wait for messages from their children to complete their matching information. Once a data vertex with unknown value receives the labels from all its children, it will compare this label set with the label set needed for satisfying the graph simulation constraints. If the two sets are equal, then it sets its local variable to true and inform its parents. Otherwise, it will set it to false. But, if it does not receive all the matching information from its children, then it will wait for the next super-step. In this super-step, messages from the previous super-step will be delivered and data vertices that did not yet compute their matching value will evaluate it depending on the set of labels received from children. The algorithm proposed is not general since it sets another constraint on the query graph. Indeed, it does not allow repeated labels while it is almost impossible to find practical queries in real-world applications with a no repeated labels.

4.2.3.2 Relaxation simulation: Gao *et al.* [31] propose two BSP-Based vertex-centric algorithms to evaluate their models URS and DRS on distributed graphs. In URS, the authors define the condition set of a query node u in Q as the set of labels that should be satisfied if two nodes are matched together. It is composed of the set of labels of the children of u that can be substituted by its grand children's labels. The initial set is composed of the children of u, then new combinations are generated by replacing a child in the initial set with its children set. To evaluate an input query in a distributed way, the query is first broadcast to all the data nodes in the graph. Then, each node sets its match flag true if its label is equal to that of a query node. It propagates its label to its children and parents during the next two iterations. Then, in the third super-step, if its match flag is true, it computes its condition set, and checks whether there is a satisfied condition set among the labels of its children. If yes, it votes for halt, otherwise it sets its match value as false and informs its parents to update their matching information accordingly. The same goes for DRS, such that both grand-parents and grand-children may substitute the parents and children of a query node respectively in the condition set. In this case, a data node will compute both parent and child condition sets and exchange its matching information with its parents and children. Data nodes that do not find candidates satisfying the two condition sets will set their match value to false and inform their parents and children so they update their matching information accordingly. It is the only distributed approach found in literature that treats a model as interesting as URS and DRS. Nevertheless, the proposed algorithms only consider patterns with unique labels which is not always the case in real-word applications.

4.2.3.3 Strong simulation: In [26], the authors proposed an algorithm that, firstly, conducts dual simulation in a distributed manner. Secondly, each vertex having a match flag set to true will execute a BFS traversal to collect the contents of a ball, of which it is the center, with a radius equal to d_q (diameter of the input query). It then evaluates dual simulation locally to each data vertex to get only matches falling inside these locality balls. This approach is not scalable as it showed poor performance guarantees when dealing with average data graphs. Its limit lies in the way data locality are generated, because if we try to duplicate the neighborhood of every single node that has a true match flag, up to d_q hops, this will certainly create a bottleneck as the query graph gets larger. That is why the authors proposed a distributed algorithm for strict simulation which scales better as it uses only the dual match graph. The authors used the same approach to evaluate tight simulation in [25], i.e. the locality balls are collected by a subset of the dual match vertices.

Furthermore, G-thinker [99] can be used for evaluating strong simulation as the locality balls creation will not require the duplication of neighborhood data inside every single subgraph generated leading to an intermediate data graph of exponential size. Since strong simulation does not require communication between the subgraphs whose diameter is well defined, using G-thinker will allow it to scale to very large graphs.

4.2.4 Synchronous Subgraph-centric: Generally, when we use the vertex-centric model, it becomes difficult for the system to achieve an optimal make-span since vertices do not know whether their neighbors reside on the same worker or not. Hence, even the use of information from vertices on the same machine needs to be delayed to the next super-step. This generally leads to a greater response time. Thus, approaches that focus on the multi-hop neighborhood of data vertices can be well adapted for distributed GPM.

4.2.4.1 Graph simulation: We discuss the use of subgraph-centric paradigm in graph simulation. In [43], the authors focused on the big data velocity issue, i.e. streaming graphs by adopting a BSP-based subgraph-centric approach on top of GPS. The data graph is partitioned into several subgraphs that are mapped to the GPS vertices. The proposed incremental algorithm evaluates graph simulation in a vertex-centric fashion (an improvement to the proposition in [26]). Each data vertex is analogous to a BSP process and uses 2 local variables; a match flag and a match Set. They take into account different kinds of events that come in to update the data graph e.g. vertex addition/removal, edge addition/removal and attribute addition/removal. So, to evaluate the same query while having a graph updated every couple of seconds for example, the algorithm maintains the previous matching information instead of resetting the variables and starting from nothing. The update event will trigger a new iterations of graph simulation. The vertices concerned will update their match flags and match sets accordingly, inform their children that will conduct the basic updating process, and so on. In streaming graphs, the data graph is updated quite often with the reception of new events every couple of seconds. Here, the main improvement brought was to avoid computing the maximum perfect match from zero for the same query graph whenever the data graph is modified. Also, the use of subgraph-centric programming model efficiently reduced the amounts of network traffic.

Several other incremental algorithms were already proposed, but they do not deal with the problem at high scale (no distributed algorithms were found except this work). Fan *et al.* proposed in [22] three algorithms for incrementally evaluating graph matching via graph simulation, bounded simulation and subgraph isomorphism. A recent work addressing the same problem was done by [102] where the authors proposed incremental algorithms for performing dual simulation when no more than half of the data graph edges are updated.

Summary

Table 1 summarizes the presented GPM approaches, where every work is cited with the programming paradigm adopted and the GPM model applied. We also give the size of the largest graph tested within each work. The different approaches were not tested in the same context. In addition to the size of graphs, there is also the nature of graphs that differs from one experimentation to the other; e.g. the density of the data graph, number of the cycles appearing in it, and the number of present labels. Moreover, these different works use very different system configurations as the number of CPU cores varied from tens to thousands of cores. We cannot compare the response times since the size and nature of patterns used in the experiments change from one to another work. We give such details only to show how far the authors went in their experimentation to validate their approaches. The largest graph that has been tested contains 137 billion nodes of synthetic data. It is common to reach such size in the current big graphs issued from social networks, the World Wide Web and many other applications.

Some of the conclusions drawn out of this comparative study are as follows: First, Graph simulation is a promising model that aims to address the challenges of actual GPM applications like network motif discovery, finding communities in social networks, and many other applications. That is why it is important to consider this research area as one of the main current challenges of subgraph matching on big graphs. Second, strong simulation is a challenging model that should be considered in future works. The way its algorithm was distributed is not efficient and needs to be revised. Although, [89] provided a new distributed algorithm for strong simulation, it does not benefit from the aforementioned programming paradigms. Furthermore, redistributing the data graph after every single update of its contents may present a performance bottleneck when dealing with highly dynamic graphs. Finally, even though relaxed graph matching is highly common among the recent works addressing GPM for social networks, subgraph isomorphism approaches are also present and they showed high performance guarantees when tested on massive graphs.

Table 1. Distributed graph pattern matching.

Work	Year	Model	Timing	Execution model	size of graph
[7]	2010	Subgraph isomorphism	Async.	Master-slave	778M edges
[81]	2012	Subgraph isomorphism	Async.	Master-slave	1B nodes
[27]	2014	Subgraph isomorphism	BSP	Vertex-centric	100K nodes
[32]	2014	Inexact ISO	BSP	Veretx-centric	105M nodes
[74]	2014	Subgraph isomorphism	BSP	Vertex-centric	42M nodes
[60]	2016	Subgraph isomorphism	BSP	Master-slave	1.3B nodes
[65]	2017	Inexact ISO	Async.	Vertex-centric	68B nodes
[67]	2018	Subgraph isomorphism	Async.	Vertex-centric	137B nodes
[78]	2018	Subgraph isomorphism	Async.	Vertex-centric	400 nodes
[24]	2018	ISO and GSIM	BSP	Master-slave	65M nodes
[66]	2020	Inexact ISO	Async.	Vertex-centric	34B nodes
[53]	2011	Strong simulation	Async.	Master-slave	548K nodes
[54]	2012	Graph simulation	Async.	Master-slave	875K nodes
[26]	2013	GSIM, DSIM, STGSIM, STSIM	BSP	Vertex-centric	3M nodes
[23]	2014	Graph simulation	Async.	Master-slave	3M nodes
[25]	2014	Tight simulation	BSP	Vertex-centric	5M nodes
[31]	2014	Relaxation simulation	BSP	Vertex-centric	1M nodes
[70]	2015	Dual simulation	BSP	Vertex-centric	10M nodes
[43]	2017	Graph simulation	BSP	Subgraph-centric	9.5 M nodes
[89]	2018	Strong simulation	Async.	Master-slave	1M nodes
[51]	2018	Graph simulation	BSP	Vertex-centric	403K nodes

5 Classification of distributed GPM approaches

In Figure 19, we provide a taxonomy for the distributed algorithms we discussed earlier. Our classification is based on three selection criteria: the programming paradigm used, the pattern matching model evaluated and whether the algorithm proposed is synchronous or asynchronous.

Based on this comparative study, we notice that both synchronous and asynchronous approaches were employed with more works adopting the BSP model. The master-slave paradigm is the oldest one used in distributed graph processing. The vertex-centric approaches form the largest part of these works, and finally subgraph-centric is the rarest paradigm with only one work addressing graph simulation.Even though [99] proposed an asynchronous subgraph-centric algorithm to evaluate ISO on G-thinker, they did not give the algorithmic details of their approach. The edge-centric paradigm was not adopted in distributed GPM. GpSM [84] and some join-based algorithms process the data graph edge by edge. Its distributed version can be implemented in edge-centric.

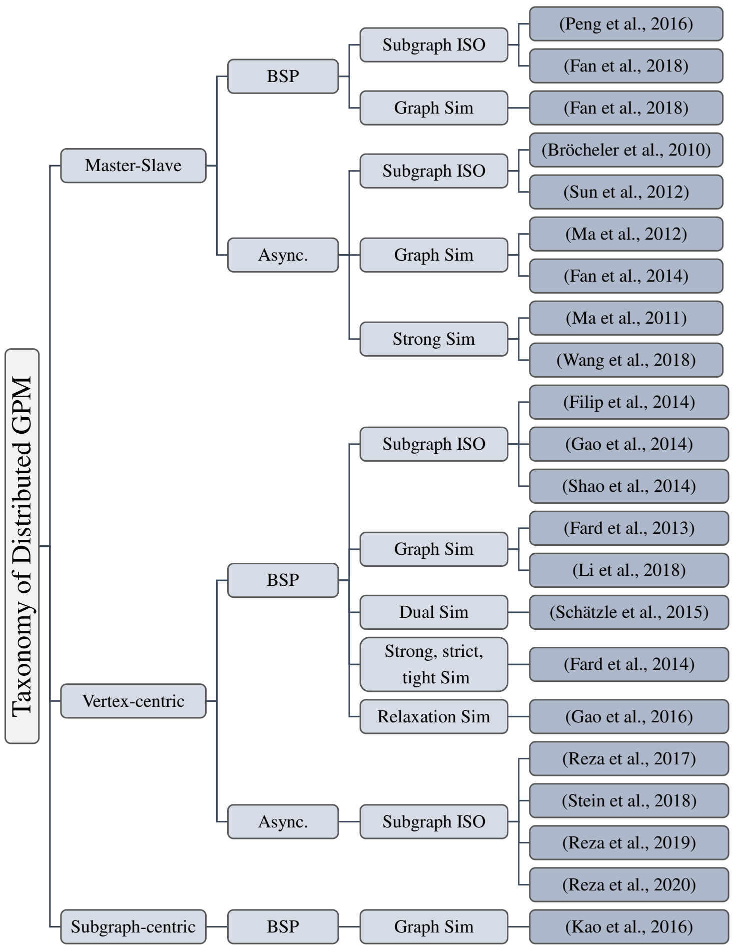


Fig. 19. Taxonomy of the discussed distributed GPM approaches.

Similarly, this paradigm can be applied to implement graph simulation that maps edges of the two graphs. Furthermore, this paradigm naturally supports dynamic graphs as events occur on the graph edges and need to be processed at an edge level, therefore, it is interesting to explore edge-centric GPM. The synchronous algorithms can be easily designed, also they are better understood compared to the asynchronous ones. A synchronous algorithm endures less communication costs. However, an asynchronous one allows a considerable improvement in the response time. Actually, the computing units (vertices, edges or even subgraphs) do not have to wait the termination of another process to send and receive requests or data to and from their neighbors.

The taxonomy provided categorizes only the existing distributed GPM approaches, some combinations were not employed and that is why we do not see them in this taxonomy. An open question would be: Why not all combinations appear in the literature of distributed GPM? One reason could be that the vertex-centric paradigm, which is the most common one, has been widely studied in the field of networks where machines behave naturally in a vertex-centric way. Moreover, as we stated earlier, the most intuitive way to evaluate subgraph matching is by a subgraph-centric algorithm since the neighborhood of a data node is needed to decide whether it is a match or not.

This could be a reason that makes the subgraph-centric paradigm more interesting compared to the vertex-centric one. However, subgraph-centric computing needs an additional level of abstraction to make the neighbourhood of multiple hops available locally to every single node in the graph. Such an abstraction layer is responsible for shipping data from one node to its neighbors whenever a local update is made, which may cause additional data shipment increasing the overall response times. Apart from the subgraph mining systems proposed such as Arabesque, Rstream or G-thinker, few works adopt this model even though adding an abstraction layer would certainly reduce the efforts for designing graph algorithms.

6 Discussion and conclusions

This survey presented recent advances in graph pattern matching on big graphs. We highlighted some of the most important extensions of graph simulation, along with detailed differences between them. Initially, we explained the reasoning behind every model and how it is possible to answer GPM queries in many ways according to the degree of flexibility needed and whether the response time is more or less important. This allowed us to draw an inclusion relationship between the answers returned by each model. As it was explained, subgraph isomorphism and graph simulation represent the two ends of the spectrum while the newly introduced models lie in between, e.g. Strong simulation and strict simulation. Nevertheless, bounded simulation and relaxation simulation are of great interest since they are even more flexible than graph simulation.

Several programming models, currently used for distributed graph processing, were also discussed. However, less work has been done to take advantage of such paradigms in evaluating graph pattern matching at a high scale. Subgraph-centric programming models are recommended for this kind of problem together with asynchronous timing. Noticeably, studies were recently focused on graph simulation due to its quadratic-time complexity. Furthermore, the recent research works are either addressing the velocity of data generation or the size of data graphs, seeking to reduce the time and space complexity of previously proposed algorithms. However, it is rare to find both issues handled in the same work.

Moreover, we identified categories of distributed GPM approaches and classified them based on multiple criteria. We gave more attention to the relaxed pattern matching since it is widely used in the current applications of social networks and massive graphs in general. An abundant literature exists discussing subgraph isomorphism queries, but the limitations of this model make it impractical for current applications of graph pattern matching. Hence, benefiting from subgraph isomorphism techniques to build an enhanced approach for strong simulation is one of the current challenges in this domain, especially in the distributed and parallel contexts. That is to say, we can combine the inherent flexibility of the model itself and exploit the strong background behind subgraph isomorphism to achieve better results.

We conclude this survey by suggesting that some of the challenges and research keys that may have a big impact on the future of the graph pattern matching domain are as follows:

- Handling highly dynamic data graphs.
- High-scale graph processing frameworks are a key tool for managing big graphs that are composed of billions of nodes and trillions of edges. Therefore, more interest should be paid to the proposition of distributed algorithms on top of these systems.
- Graph compression methods aim to reduce the size of the data graph allowing for the management of bigger data graphs. It will be interesting to propose general-purpose compression techniques that preserve the subgraph matching properties in a distributed environment.
- The use of indices helps accelerating access to the data graph vertices, but the index size increases with the size of the data graph reaching a point when it cannot be stored on the

main memory as it is the case for massive graphs. So, another challenge that should be considered is the use of distributed indices in answering GPM queries.

- GPM models like bounded simulation or multi-constrained simulation were not given enough attention in the recent works proposing algorithms adapted to the context of big graphs, so addressing these models is another important research direction.

Acknowledgments

This work is supported by the Franco-Algerian collaborative program PHC Tassili BiGreen N°18 MDU 111 and by the DGRSDT grant FNRSDT N°253.

References

[1] Foto N Afrati, Dimitris Fotakis, and Jeffrey D Ullman. 2013. Enumerating subgraph instances using map-reduce. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, Brisbane, QLD, 62–73.

[2] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *Proc. VLDB Endow.* 11, 6 (Feb. 2018), 691–704. https://doi.org/10.14778/3199517.3199520

[3] Bibek Bhattarai, Hang Liu, and H Howie Huang. 2019. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, Amsterdam, Netherlands, 1447–1462.

[4] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, San Francisco, California, USA, 1199–1214.

[5] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. 2013. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics* 14, S7 (2013), S13.

[6] Dhruba Borthakur. 2007. The hadoop distributed file system: Architecture and design. *Hadoop Project Website* 11, 2007 (2007), 21.

[7] M. Bröcheler, A. Pugliese, and V. S. Subrahmanian. 2010. COSI: Cloud Oriented Subgraph Identification in Massive Social Networks. In *2010 International Conference on Advances in Social Networks Analysis and Mining*. IEEE, IEEE, Odense, Denmark, 248–255. https://doi.org/10.1109/ASONAM.2010.80

[8] Vincenzo Carletti, Pasquale Foggia, Antonio Greco, Alessia Saggese, and Mario Vento. 2018. Comparing performance of graph matching algorithms on huge graphs. *Pattern Recognition Letters* 134 (2018), 58–67.

[9] Vincenzo Carletti, Pasquale Foggia, Antonio Greco, Mario Vento, and Vincenzo Vigilante. 2019. VF3-Light: A lightweight subgraph isomorphism algorithm and its experimental evaluation. *Pattern Recognition Letters* 125 (2019), 591–596.

[10] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. 2017. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with VF3. *IEEE transactions on pattern analysis and machine intelligence* 40, 4 (2017), 804–818.

[11] Vincenzo Carletti, Pasquale Foggia, and Mario Vento. 2015. VF2 Plus: An improved version of VF2 for biological graphs. In *International Workshop on Graph-Based Representations in Pattern Recognition*. Springer, Beijing, China, 168–177.

[12] 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 Thirteenth EuroSys Conference*. ACM, Porto, Portugal, 1–12.

[13] Avery Ching. 2013. Scaling apache giraph to a trillion edges. , 25 pages.

[14] D. Conte, P. Foggia, C. Sansone, and M. Vento. 2004. Thirty Years of Graph Matching in Pattern Recognition. *International Journal of Pattern Recognition and Artificial Intelligence* 18, 03 (2004), 265–298. https://doi.org/10.1142/S0218001404003228

[15] L P Cordella, P Foggia, C Sansone, and M Vento. 2001. An improved algorithm for matching large graphs. *Proceedings of the 3rd IAPR Workshop on Graph-Based Representations in Pattern Recognition* 219, 2 (2001), 149–159. https://doi.org/10.1.1.101.5342

[16] Shixuan Csun and Qiong Luo. 2018. Parallelizing Recursive Backtracking Based Subgraph Matching on a Single Machine. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, Singapore, Singapore, 1–9.

[17] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492

[18] Vinicius Dias, Carlos HC 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*. ACM, Amsterdam Netherlands, 1357–1374.

[19] Ruihuan Du, Jiannan Yang, Yongzhi Cao, and Hanpin Wang. 2018. Personalized graph pattern matching via limited simulation. *Knowledge-Based Systems* 141 (2018), 31–43. https://doi.org/10.1016/j.knosys.2017.11.008

[20] W. Stout Dustin. 2019. Social Media Statistics 2020: Top Networks By the Numbers. https://dustinstout.com/social-media-statistics/. Accessed: 2020-03-01.

[21] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, Yinghui Wu, and Yunpeng Wu. 2010. Graph Pattern Matching: From Intractable to Polynomial Time. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 264–275. https://doi.org/10.14778/1920841.1920878

[22] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Incremental Graph Pattern Matching. *ACM Trans. Database Syst.* 38, 3, Article 18 (Sept. 2013), 47 pages. https://doi.org/10.1145/2489791

[23] W. Fan, X. Wang, Y. Wu, and D. Deng. 2014. Distributed graph simulation: Impossibility and possibility. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1083–1094. https://doi.org/10.14778/2732977.2732983

[24] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing sequential graph computations. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 1–39.

[25] Arash Fard, M Usman Nisar, John A Miller, and Lakshmish Ramaswamy. 2014. Distributed and Scalable Graph Pattern Matching: Models and Algorithms. *International Journal of Big Data (ISSN 2326-442X)* 1, 1 (2014), 1–14. https://doi.org/10.29268/stbd.2014.1.1.1

[26] 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 *2013 IEEE International Conference on Big Data*. IEEE, Santa Clara, CA, USA, 403–411. https://doi.org/10.1109/BigData.2013.6691601

[27] Sınziana Maria Filip. 2014. *A scalable graph pattern matching engine on top of Apache Giraph.* Ph.D. Dissertation. VU University Amsterdam.

[28] Pasquale Foggia, Gennaro Percannella, and Mario Vento. 2014. Graph Matching And Learning in Pattern Recognition In The Last 10 Years. *International Journal of Pattern Recognition and Artificial Intelligence* 28, 01 (2014), 1450001. https://doi.org/10.1142/S0218001414500013 arXiv:https://doi.org/10.1142/S0218001414500013

[29] Per Fuchs, Peter Boncz, and Bogdan Ghit. 2020. EdgeFrame: Worst-Case Optimal Joins for Graph-Pattern Matching in Spark. In *Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. ACM, Portland OR USA, 1–11.

[30] Brian Gallagher. 2006. Matching Structure and Semantics : A Survey on Graph-Based Pattern Matching. *Technical Report FS-06-02* 6 (2006), 45–53.

[31] Jianliang Gao, Ping Liu, Xuedan Kang, Lixia Zhang, and Jianxin Wang. 2016. PRS: parallel relaxation simulation for massive graphs. *Comput. J.* 59, 6 (2016), 848–860.

[32] Jun Gao, Chang Zhou, Jiashuai Zhou, and Jeffrey Xu Yu. 2014. Continuous pattern detection over billion-edge graph using distributed framework. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, Chicago, IL, USA, 556–567.

[33] Michael R Garey and David S Johnson. 1979. Computers and intractability: a guide to NP-completeness.

[34] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. Association for Computing Machinery, Bolton Landing, NY, USA, 29–43. https://doi.org/10.1145/945445.945450

[35] 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*. ACM, Utah USA, 289–300.

[36] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, Amsterdam Netherlands, 1429–1446.

[37] 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)*. Association for Computing Machinery, New York, New York, USA, 337–348. https://doi.org/10.1145/2463676.2465300

[38] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-Time: Query Language and Access Methods for Graph Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. Association for Computing Machinery, Vancouver, Canada, 405–418. https://doi.org/10.1145/1376616.1376660

[39] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. 1995. Computing simulations on finite and infinite graphs. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE, USA, 453–462. https://doi.org/10.1109/SFCS.1995.492576

[40] Michael Hunger. 2014. *Neo4j 2.0: Eine Graphdatenbank für alle.* entwickler. Press, Eltville, Germany.

[41] Alpár Jüttner and Péter Madarasi. 2018. VF2++-An improved subgraph isomorphism algorithm. *Discrete Applied Mathematics* 242 (2018), 69–81.

[42] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. 2017. High-level programming abstractions for distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering* 30, 2 (2017), 305–324.

[43] Jyun-Sheng Kao and Jerry Chou. 2016. Distributed Incremental Pattern Matching on Streaming Graphs. In *Proceedings of the ACM Workshop on High Performance Graph Processing - HPGP '16*, Vol. 1828. ACM Press, New York, New York, USA, 43–50. https://doi.org/10.1145/2915516.2915519 arXiv:arXiv:1603.07016v1

[44] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: A System for Dynamic Load Balancing in Large-Scale Graph Processing. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13).* Association for Computing Machinery, Prague, Czech Republic, 169–182. https://doi.org/10.1145/2465351.2465369

[45] Kyoungmin Kim, In Seo, Wook-Shin Han, Jeong-Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. Turboflux: A fast continuous subgraph matching system for streaming graph data. In *Proceedings of the 2018 International Conference on Management of Data.* ACM, Houston TX USA, 411–426.

[46] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment* 8, 10 (2015), 974–985.

[47] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable distributed subgraph enumeration. *Proceedings of the VLDB Endowment* 10, 3 (2016), 217–228.

[48] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, et al. 2019. Distributed subgraph matching on timely dataflow. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1099–1112.

[49] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *Proceedings of the VLDB Endowment* 6, 2 (2012), 133–144.

[50] Jia Li, Yang Cao, and Shuai Ma. 2017. Relaxing graph pattern matching with explanations. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management.* ACM, Singapore Singapore, 1677–1686.

[51] Jingdong Li, Jin Li, and Xiaoling Wang. 2018. A Vertex-Centric Graph Simulation Algorithm for Large Graphs. In *Big Data*, Zongben Xu, Xinbo Gao, Qiguang Miao, Yunquan Zhang, and Jiajun Bu (Eds.). Springer, Singapore, 238–254.

[52] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. 2006. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Philadelphia, PA, USA) *(KDD '06).* Association for Computing Machinery, New York, NY, USA, 872–881. https://doi.org/10.1145/1150402.1150522

[53] Shuai Ma, Yang Cao, Wenfei Fan, Jinpeng Huai, and Tianyu Wo. 2011. Capturing Topology in Graph Pattern Matching. *Proceedings of the VLDB Endowment* 5, 4 (2011), 310–321.

[54] 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).* Association for Computing Machinery, Lyon, France, 949–958. https://doi.org/10.1145/2187836.2187963

[55] 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).* Association for Computing Machinery, Indianapolis, Indiana, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[56] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (July 2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

[57] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record* 42, 4 (2014), 5–16.

[58] Kirk Ogaard, Heather Roy, Sue Kase, Rakesh Nagi, Kedar Sambhoos, and Moises Sudit. 2013. Discovering Patterns in Social Networks with Graph Matching Algorithms. In *Social Computing, Behavioral-Cultural Modeling and Prediction*, Ariel M. Greenberg, William G. Kennedy, and Nathan D. Bos (Eds.). Springer, Berlin, Heidelberg, 341–349.

[59] R. Pearce. 2012. Highly Asynchronous VisitOr Queue Graph Toolkit. https://www.osti.gov//servlets/purl/1231683

[60] Peng Peng, Lei Zou, M Tamer Özsu, Lei Chen, and Dongyan Zhao. 2016. Processing SPARQL queries over distributed RDF graphs. *The VLDB Journal* 25, 2 (2016), 243–268.

[61] Todd Plantenga. 2013. Inexact subgraph isomorphism in MapReduce. *J. Parallel and Distrib. Comput.* 73, 2 (2013), 164–175.

[62] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph matching: on compression and computation. *Proceedings of the VLDB Endowment* 11, 2 (2017), 176–188.

[63] Xuguang Ren and Junhu Wang. 2015. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *Proceedings of the VLDB Endowment* 8, 5 (2015), 617–628.

[64] Xuguang Ren and Junhu Wang. 2016. Multi-query optimization for subgraph isomorphism search. *Proceedings of the VLDB Endowment* 10, 3 (2016), 121–132.

[65] T. Reza, C. Klymko, M. Ripeanu, G. Sanders, and R. Pearce. 2017. Towards Practical and Robust Labeled Pattern Matching in Trillion-Edge Graphs. In *2017 IEEE International Conference on Cluster Computing (CLUSTER).* IEEE, Honolulu, HI, 1–12. https://doi.org/10.1109/CLUSTER.2017.85

[66] Tashin Reza, Matei Ripeanu, Geoffrey Sanders, and Roger Pearce. 2020. Approximate Pattern Matching in Massive Graphs with Precision and Recall Guarantees. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* ACM, Portland OR USA, 1115–1131.

[67] T. Reza, M. Ripeanu, N. Tripoul, G. Sanders, and R. Pearce. 2018. PruneJuice: Pruning Trillion-edge Graphs to a Precise Pattern-Matching Solution. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, Dallas, Texas, USA, 265–281. https://doi.org/10.1109/SC.2018.00024

[68] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13).* Association for Computing Machinery, Farminton, Pennsylvania, 472–488. https://doi.org/10.1145/2517349.2522740

[69] Semih Salihoglu and Jennifer Widom. 2013. GPS: A Graph Processing System. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management (SSDBM).* Association for Computing Machinery, Baltimore, Maryland, USA, Article 22, 12 pages. https://doi.org/10.1145/2484838.2484843

[70] Alexander Schätzle, Martin Przyjaciel-Zablocki, Thorsten Berberich, and Georg Lausen. 2016. S2X: Graph-Parallel Querying of RDF with GraphX. In *Biomedical Data Management and Graph Online Querying*, Fusheng Wang, Gang Luo, Chunhua Weng, Arijit Khan, Prasenjit Mitra, and Cong Yu (Eds.). Springer International Publishing, Cham, 155–168.

[71] 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.* ACM, Santa Clara, CA, 214–228.

[72] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming verification hardness: an efficient algorithm for testing subgraph isomorphism. *Proceedings of the VLDB Endowment* 1, 1 (2008), 364–375.

[73] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13).* Association for Computing Machinery, New York, New York, USA, 505–516. https://doi.org/10.1145/2463676.2467799

[74] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel subgraph listing in a large-scale graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data.* ACM, Utah USA, 625–636.

[75] Ali Shemshadi, Quan Z Sheng, and Yongrui Qin. 2016. Efficient pattern matching for graphs with multi-labeled nodes. *Knowledge-Based Systems* 109 (2016), 256–265.

[76] Qun Shi, Guanfeng Liu, Kai Zheng, An Liu, Zhixu Li, Lei Zhao, and Xiaofang Zhou. 2017. Multi-Constrained Top-K Graph Pattern Matching in Contextual Social Graphs. *Proceedings - 2017 IEEE 24th International Conference on Web Services, ICWS 2017* 6, 1 (2017), 588–595. https://doi.org/10.1109/ICWS.2017.69

[77] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. 2014. GoFFish: A Sub-graph Centric Framework for Large-Scale Graph Analytics. In *Euro-Par 2014 Parallel Processing*, Fernando Silva, Inês Dutra, and Vítor Santos Costa (Eds.). Springer International Publishing, Cham, 451–462.

[78] Michael Stein, Alexander Frömmgen, Roland Kluge, Lin Wang, Augustin Wilberg, Boris Koldehofe, and Max Mühlhäuser. 2018. Scaling Topology Pattern Matching: A Distributed Approach. *self* 1 (2018), n2.

[79] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* ACM, Portland OR USA, 1083–1098.

[80] Shixuan Sun and Qiong Luo. 2020. Subgraph Matching with Effective Matching Order and Indexing. *IEEE Transactions on Knowledge and Data Engineering* (2020), 1–1.

[81] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *Proceedings of the VLDB Endowment* 5, 9 (2012), 788–799.

[82] Carlos HC 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.* ACM, Monterey California, 425–440.

[83] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "think like a vertex" to" think like a graph". *Proceedings of the VLDB Endowment* 7, 3 (2013), 193–204.

[84] Ha-Nguyen Tran, Jung-jae Kim, and Bingsheng He. 2015. Fast subgraph matching on large graphs using graphics processors. In *Proceedings of the 20th International Conference on Database Systems for Advanced Applications.* Springer, Cham, Hanoi, Vietnam, 299–315.

[85] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (1976), 31–42. https://doi.org/10.1145/321921.321925

[86] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111. https://doi.org/10.1145/79173.79181

[87] Todd L. Veldhuizen. 2012. Leapfrog Triejoin: a worst-case optimal join algorithm. arXiv:1210.0481 [cs.DB]

[88] Mario Vento. 2015. A long trip in the charming world of graphs for Pattern Recognition. *Pattern Recognition* 48, 2 (2015), 291–301. https://doi.org/10.1016/j.patcog.2014.01.002

[89] Hongzhi Wang, Ning Li, Jianzhong Li, and Hong Gao. 2018. Parallel algorithms for flexible pattern matching on big graphs. *Information Sciences* 436-437 (2018), 418–440. https://doi.org/10.1016/j.ins.2018.01.018

[90] Junhu Wang, Xuguang Ren, Shikha Anirban, and Xin-Wen Wu. 2019. Correct filtering for subgraph isomorphism search in compressed vertex-labeled graphs. *Information Sciences* 482 (2019), 363–373.

[91] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. USENIX association, Carlsbad, CA, USA, 763–782.

[92] Zhaokang Wang, Rong Gu, Weiwei Hu, Chunfeng Yuan, and Yihua Huang. 2019. BENU: Distributed Subgraph Enumeration with Backtracking-Based Framework. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, Macao, Macao, 136–147.

[93] Maria Winans, Dave Faupel, Amber Armstrong, Jay Henderson, Ellen Valentine, Loren McDonald, Dave Walters, Jeremy Waite, Michael Trapani, and Elizabeth Magill. 2016. 10 Key Marketing Trends for 2017 and Ideas for Exceeding Customer Expectations. , 18 pages.

[94] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *First international workshop on graph data management experiences and systems*. ACM, New York, USA, 1–6.

[95] Da Yan, Yingyi Bu, Yuanyuan Tian, and Amol Deshpande. 2017. Big Graph Analytics Platforms. *Foundations and Trends® in Databases* 7, 1-2 (2017), 1–195. https://doi.org/10.1561/1900000056 arXiv:1603.07016

[96] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.

[97] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2015. Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation. In *Proceedings of the 24th International Conference on World Wide Web* (Florence, Italy) *(WWW '15)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 1307–1317. https://doi.org/10.1145/2736277.2741096

[98] Da Yan, James Cheng, M Tamer Özsu, Fan Yang, Yi Lu, John CS Lui, Qizhen Zhang, and Wilfred Ng. 2016. A general-purpose query-centric framework for querying big graphs. *Proceedings of the VLDB Endowment* 9, 7 (2016), 564–575.

[99] Da Yan, Guimu Guo, Md Mashiur Rahman Chowdhury, M Tamer Özsu, Wei-Shinn Ku, and John CS Lui. 2020. G-thinker: A Distributed Framework for Mining Subgraphs in a Big Graph. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, Dallas, TX, USA, USA, 1369–1380.

[100] Da Yan, Yuzhen Huang, James Cheng, and Huanhuan Wu. 2016. Efficient Processing of Very Large Graphs in a Small Cluster. arXiv:1601.05590 [cs.DC]

[101] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A distributed graph engine for web scale RDF data. *Proceedings of the VLDB Endowment* 6, 4 (2013), 265–276.

[102] Lixia Zhang and Jianliang Gao. 2018. Incremental Graph Pattern Matching Algorithm for Big Graph Data. *Scientific Programming* 2018 (2018), 1–8. https://doi.org/10.1155/2018/6749561

[103] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: distance index based subgraph matching in biological networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. ACM, Saint Petersburg Russia, 192–203.

[104] Tianming Zhang, Yunjun Gao, Linshan Qiu, Lu Chen, Qingyuan Linghu, and Shiliang Pu. 2020. Distributed time-respecting flow graph pattern matching on temporal graphs. *World Wide Web* 23, 1 (2020), 609–630.

[105] Peixiang Zhao and Jiawei Han. 2010. On graph query optimization in large networks. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 340–351.

[106] Chang Zhou, Jun Gao, Binbin Sun, and Jeffrey Xu Yu. 2014. MOCgraph : Scalable Distributed Graph Processing Using Message Online Computing. *Vldb* 8, 4 (2014), 377–388. https://doi.org/10.14778/2735496.2735501