Distributed Subgraph Counting: A General Approach

Hao Zhang, Jeffrey Xu Yu, Yikai Zhang, Kangfei Zhao, Hong Cheng
The Chinese University of Hong Kong, Hong Kong, China
{hzhang,yu,ykzhang,kfzhao,hcheng}@se.cuhk.edu.hk

ABSTRACT

In this paper, we study local subgraph counting, which is to count the occurrences of a user-given pattern graph p around every node v in a data graph G, when v matches to a given orbit o in p, where the orbit serves as a center to count p. In general, the orbit can be a node, an edge, or a set of nodes in p. Local subgraph counting has played an important role in characterizing high-order local structures that exhibit in a large graph, and has been widely used in denser and relevant communities mining, graphlet degree distribution, discriminative features selection for link prediction, relational classification and recommendation. In the literature, almost all the existing works support a k-node pattern graph, for $k \leq 5$, with either 1 node orbit or 1 edge orbit. Their approaches are difficult to support larger k due to the fact that subgraph counting is to count by subgraph isomorphism. In this work, we develop a new general approach to count any k pattern graphs with any orbits selected. The key idea behind is that we do local subgraph counting by homomorphism counting, which can be solved by relational algebra using joins, group-by and aggregation. By homomorphism counting, we do local subgraph counting by eliminating counts for those that are not subgraph isomorphism matchings from the total count for any possible matchings. We have developed a distributed system named DISC on *Spark*. Our extensive experiments validate the efficiency of our approach by testing 114 local subgraph counting queries used in the existing work over real graphs, where no existing work can support all.

PVLDB Reference Format:
Hao Zhang, Jeffrey Xu Yu, Yikai Zhang, Kangfei Zhao, Hong Cheng. Distributed Subgraph Counting: A General Approach. *PVLDB*, 13(11): 2493-2507, 2020.
DOI: https://doi.org/10.14778/3407790.3407840

1. INTRODUCTION

Graph has been widely used in modelling complex interconnectivity of objects in real applications such as commerce, medicine, and social network [40, 23, 52, 31, 82, 72, 61]. Among many graph algorithms/systems being studied, local subgraph counting is to count how many times a user-given pattern graph p exists around every node(s) in a large data graph G. Such local subgraph counting has played an increasingly important role in characterizing high-order local structures that exhibit in a large graph. In [75, 17, 76] local subgraph counting is used to mine denser and relevant communities. In [86, 85] local subgraph counting is used to compute metrics that quantify the clustering of nodes such as higher-order local clustering coefficient and local closure coefficient, and in [60] local subgraph counting provides insights (e.g., graphlet degree distribution) into summarizing the structure of an entire network. Furthermore, a wide range of studies leverage the counts of local subgraphs as powerful discriminative features for pattern matching and recognition. Such studies include network comparison and alignment in biological networks [36], anomaly detection in computer networks [55, 11], detecting strong ties in social networks [69], and the statistical relational learning tasks for link prediction, relational classification and recommendation [66, 47], where local subgraph counting improves the node or edge representations [8, 68, 7, 65]. As indicated in [66], the subgraph patterns to count are graphlets from the simplest reciprocity (e.g., 2-star, triangle) to complex patterns (e.g, clique). In order to optimize a constructed model and its training process in mining/learning where feature selection plays an important role, it is highly demanded to efficiently process local subgraph counting for any pattern graphs in a distributed manner.

Local subgraph counting has been studied on large graphs [9, 29, 30, 38, 39, 58, 49, 48, 50, 38, 10, 26, 58]. Let Q be a local subgraph counting query as $Q = (p, o, t)$, where t specifies the type of counting by subgraph isomorphism (e.g., local subgraph counting or local induced subgraph counting), p is a connected k-node pattern graph, and o is a given orbit of p which serves as a center to count p. Both local subgraph counting and local induced subgraph counting are to count how many p exist around every node v in a data graph G if the orbit is a node orbit and o maps to v. For example, consider Fig. 1. Assuming u_2 of p is the orbit, the local induced subgraph count of v_5 is 1 because there is exactly one match of p in G that maps u_2 to v_5, namely $\{u_1 \rightarrow v_6, u_2 \rightarrow v_5, u_3 \rightarrow v_4, u_4 \rightarrow v_1\}$. We emphasize that the orbit can also be an edge, or a set of nodes in p.

As given by the survey on local subgraph counting [63], there are enumeration, matrix-based, and decomposition-based approaches. An enumeration approach [57, 81] is to

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407840

Table 1: The number of combinations for k-node patterns

Orbits	1	2	3	4	5	6	7
none	1	1	2	6	21	112	853
1 node	1	1	3	11	58	407	4,306
1 edge	0	1	2	10	57	486	5,985
1 triangle	0	0	1	3	21	197	2,752
2 nodes	0	0	1	8	67	701	10,047

count by enumerating all matches of p in G. A matrix-based approach [38, 39, 48, 50, 26] is to count by solving linear algebra equations based on the enumeration of some k'-node pattern graphs that are not the same as the k-node pattern graph p, for $k' \leq k$. And a decomposition-based approach [30, 58] is to count p based on counting smaller graphs that are obtained by decomposing p. Local subgraph counting is challenging, since there are a large number of combinations for pattern graphs p with the orbits o selected. We show the number of patterns in Table **??** that need to be counted for a k-node pattern graph for $k \leq 7$. Take 1 node orbit (center) as an example, when $k = 5$, there are 58 combinations, and when $k = 6$ there are 407 combinations. The existing work can only support a certain k-node graph (e.g., $k \leq 5$) and can only support some of k-node graphs (e.g., 32 out of 58 5-node pattern graphs) with certain types of orbit (e.g., 1 node orbit or 1 edge orbit). The approaches that support 1 node orbit cannot be used to support 1 edge orbit and vice versa. The difficulty of local subgraph counting is due to the fact that the nature of subgraph counting is by subgraph isomorphism. The only approach that can handle any k-node pattern graphs with node orbits selected is the single machine matrix-based approach JESSE [49, 48, 50]. But JESSE cannot support counting efficiently when $k > 4$. In this paper, we give a general approach that can handle large k with different orbits (e.g., 1 node orbit, 1 edge orbit, etc.) efficiently.

We explore how to support subgraph counting by homomorphism [37, 28, 34, 25, 27, 18, 12, 24], which has not yet been well studied in database community. In general, the problem of deciding whether a homomorphism from a graph p to another graph G exists is hard. Given G to be fixed, the decision problem is NP-complete if G is not bipartite and does not contain a loop [37], and the counting counterpart (i.e., counting the number of homomorphisms from p to G) is #P-complete if G has a connected component that is not a complete graph with all loops present or a complete bipartite graph with no loops present [28]. On the other hand, when p is restricted to a recursively enumerable class of graphs $\mathcal{P}$, the decision problem (resp. the counting problem) is in P if and only if $\mathcal{P}$ has bounded treewidth modulo homomorphic equivalence, under the assumption of FPT $\neq$ W[1] (resp. FPT $\neq$ #W[1]) [34, 25]. As shown in [27], if a tree decomposition of p with width w is provided, then the number of homomorphisms from p to G can be computed in $O(n_p \cdot n^{w+1} \min\{n, w\})$ time with n_p and n being the numbers of nodes in p and G, respectively. All works focus on global subgraph counting [28, 25, 27, 18, 12, 24], which is to count how many subgraphs that match the pattern graph p in an entire graph G. Our focus is on local subgraph counting.

Contributions. We propose a new distributed decomposition based approach by adapting the homomorphism counting in [18] for local subgraph counting for different orbits. It is important to note that the approach of subgraph counting by homomorphism counting is mainly discussed in terms of complexity, and there are no systems to do so [63]. The main reason behind is that by homomorphism counting it is to do subgraph counting by eliminating counts for those that are not subgraph matchings from the total count for any possible matchings, which can be costly since it needs to explore all possible matchings in practice. First, we give details on how to reduce local subgraph counting from isomorphism to homomorphism in the presence of orbits where orbits can be 1 node orbit, 1 edge orbit, or any other orbits. Second, we take a relational-based approach to support homomorphism counting by relational algebra (e.g., natural joins with group-by and aggregation). The reason that we do so instead of taking a native graph approach is as follows. To reduce the cost, we need (a) to decompose a complex pattern graph p into smaller subgraphs, and (b) to use counting to prune unnecessary enumeration. However, it is difficult to come up with a way that supports both (a) and (b) for any possible p with different orbits by a native graph approach. By a relational-based approach, we decompose a query into smaller queries, and push down group-by and aggregations over joins. Third, we have implemented a distributed system DISC on *Spark* based on the state-of-the-art approach for join queries HCubeJoin [22]. We discuss how to further reduce the communication cost and computing cost with the additional group-by and aggregation over joins. Fourth, we conducted extensive experimental studies to compare DISC with the representative existing approaches over 9 real datasets. Our testing is based on 114 local induced subgraph counting queries used in the existing systems [57, 30, 38, 39, 58, 26, 81, 49, 48, 50]. It is important to mention that there is no single system that can run all the 114 queries efficiently.

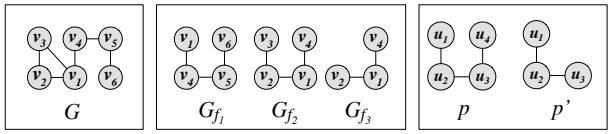
Organizations. We give preliminaries and the problem statement in Section 2, and discuss the existing approaches in Section 3. We propose a new decomposition-based approach by homomorphism counting in Section 4, discuss homomorphism counting by relational algebra in Section 5 and distributed processing in Section 6. We discuss related works in Section 7, and report the efficiency of our approach in Section 8. We conclude our work in Section 9.

2. PRELIMINARIES

We model a simple undirected graph as $G = (V, E)$, where V and E denote the sets of nodes and edges in G, respectively. Let $n = |V|$ and $m = |E|$ denote the number of nodes and the number of edges, respectively. A graph $G' = (V', E')$ is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$, and is further an induced subgraph of G if E' contains all the edges in G that have both endpoints belonging to V'. We call a graph a k-node graph (or k-graph) if it has k nodes.

Homomorphism & Subgraph Isomorphism. Let $G = (V, E)$ be a data graph and $p = (V_p, E_p)$ be a pattern graph. A function $f : V_p \mapsto V$ is called a *homomorphism* of p if for each edge $(u, u') \in E_p$, we have $(f(u), f(u')) \in E$. A homomorphism f of p is further called a *subgraph isomorphism* (or *isomorphism*[1] for short) of p if f is *injective*, i.e., f never maps distinct nodes in V_p to the same node in V. Each homomorphism f of p naturally induces a subgraph $G_f = (V_f, E_f)$ of G, where $V_f = \{f(u) \mid u \in V_p\}$ and $E_f = \{(f(u), f(u')) \mid (u, u') \in E_p\}$. Such a G_f is called an

[1]In graph theory, subgraph isomorphism and isomorphism are with different meanings. We use the term "isomorphism" solely as a shorthand for "subgraph isomorphism".



(a) Data Graph (b) Matches of p (c) Pattern Graphs

Figure 1: An Example

iso-match (resp. a *hom-match*) of p provided that f is an isomorphism (resp. a homomorphism). In addition, we classify isomorphisms f into two types in terms of G_f. Specifically, if G_f is an induced subgraph of G, f is an induced isomorphism; otherwise, f is a non-induced isomorphism. In the literature, homomorphism can be supported by relational joins [51, 13, 41].

Example 2.1: Fig. 1 shows a data graph G and a pattern graph p. Consider the following 3 mappings from V_p to V: $f_1 = \{u_1 \to v_1, u_2 \to v_4, u_3 \to v_5, u_4 \to v_6\}$, $f_2 = \{u_1 \to v_3, u_2 \to v_2, u_3 \to v_1, u_4 \to v_4\}$ and $f_3 = \{u_1 \to v_1, u_2 \to v_2, u_3 \to v_1, u_4 \to v_4\}$. Their corresponding G_{f_i}'s are shown in Fig. 1(b). It is not hard to verify that f_1, f_2 and f_3 are all homomorphisms. Indeed, for each edge (u, u') in p, $(f_i(u), f_i(u'))$ is present in G. However, only f_1 and f_2 are subgraph isomorphisms since f_3 is not injective (f_3 maps both u_1 and u_3 to v_1). As a result, G_{f_1} and G_{f_2} both are iso-matches, while G_{f_3} is just a hom-match. In addition, f_1 is an induced isomorphism because G_{f_1} is an induced iso-match, while f_2 is a non-induced isomorphism.

Automorphism Orbit. We first define the notion of *automorphism*. For a given graph $G = (V, E)$, an *automorphism* is a *bijective* function $g : V \mapsto V$ such that $(v, v') \in E$ *iff* $(g(v), g(v')) \in E$. Intuitively, an automorphism g maps G to itself in a structure-preserving manner. The notion of *automorphism orbit* is defined over the subgraphs of G. For example, the *node orbits* of G are defined as the equivalence classes of V under automorphisms. That is, for any two nodes $v, v' \in V$, they belong to the same node orbit *iff* an automorphism g satisfying $g(v) = v'$ exists. Intuitively, a node plays the same topological role as the other nodes in the same orbit. The *edge orbits* of G are defined to be the equivalence classes of E under automorphisms. Specifically, for any two edges $(u, v), (u', v') \in E$, they are in the same edge orbit *iff* there is an automorphism g such that $g(u) = u' \wedge g(v) = v'$ or $g(u) = v' \wedge g(v) = u'$. Other orbits can be defined similarly. We would like to stress two points here. First, for ease of presentation, we focus on node orbits hereafter, but we emphasize that our approach also works with other orbits. Second, we assume that a representative node is selected for each node orbit. Each time we refer to a node orbit, we actually mean its representative. After all, the nodes in the same orbit are topologically equivalent.

Example 2.2: For the pattern graph p in Fig. 1, the functions $g_1 = \{u_i \to u_i, i \in [1, 4]\}$ and $g_2 = \{u_1 \to u_4, u_2 \to u_3, u_3 \to u_2, u_4 \to u_1\}$ are the only two automorphisms. There are two node orbits in p, i.e., $\{u_1, u_4\}$ and $\{u_2, u_3\}$.

Local Subgraph Counting. In this work, we study local subgraph counting queries. Specifically, a subgraph counting query Q is defined as $Q = (p, o, t)$, where $p = (V_p, E_p)$ is a *connected* pattern graph, o is a node orbit of p, and t specifies the type of the query. We provide two public query types, namely SubG and InSubG, for *local subgraph counting* and *local induced subgraph counting*, respectively. Specifically, given a data graph $G = (V, E)$, a node $v \in V$ and a query $Q = (p, o, \mathsf{SubG})$, let $\mathsf{SubG}_{p,o}(v) = \{G_f \mid f$ is an isomorphism of p with $f(o) = v\}$. In other words, $\mathsf{SubG}_{p,o}(v)$ contains all iso-matches of p that match v to o. With $\mathsf{SubG}_{p,o}(\cdot)$, the problem of local subgraph counting is to compute $|\mathsf{SubG}_{p,o}(v)|$ for every node $v \in V$. $\mathsf{InSubG}_{p,o}(\cdot)$ and the problem of local induced subgraph counting are defined similarly but in terms of induced isomorphisms.

Example 2.3: Consider Fig. 1 and a SubG query $Q_1 = (p', u_2, \mathsf{SubG})$. For v_5, there are two isomorphisms that map u_2 to v_5, i.e., $f_4 = \{u_1 \to v_4, u_2 \to v_5, u_3 \to v_6\}$ and $f_5 = \{u_1 \to v_6, u_2 \to v_5, u_3 \to v_4\}$. Moreover, G_{f_4} and G_{f_5} both refer to the same subgraph induced by $\{v_4, v_5, v_6\}$. Hence, $\mathsf{SubG}_{p',u_2}(v_5) = \{G_{f_4}\}$. Next, consider another query $Q_2 = (p', u_2, \mathsf{InSubG})$. We have $\mathsf{InSubG}_{p',u_2}(v_5) = \mathsf{SubG}_{p',u_2}(v_5)$. After all, both f_4 and f_5 are induced isomorphisms.

In order to handle SubG and InSubG queries, we further define 3 query types, namely ISO, InISO and HOM, for *local isomorphism counting*, *local induced isomorphism counting* and *local homomorphism counting*, respectively. Accordingly, for a query Q of these types, what $\mathsf{ISO}_{p,o}(\cdot)$, $\mathsf{InISO}_{p,o}(\cdot)$ and $\mathsf{HOM}_{p,o}(\cdot)$ keep are no longer subgraphs but functions. Specifically, $\mathsf{ISO}_{p,o}(v)/\mathsf{InISO}_{p,o}(v)/\mathsf{HOM}_{p,o}(v) = \{f \mid f$ is an isomorphism/induced isomorphism/homomorphism of p with $f(o) = v\}$. For convenience, for a query $Q = (p, o, t)$, when p and o are clear, we also use $t(\cdot)$ in place of $t_{p,o}(\cdot)$.

Example 2.4: Continuing with Example 2.3, for p' and orbit u_2, we have $\mathsf{ISO}(v_5) = \mathsf{InISO}(v_5) = \{f_4, f_5\}$. As for HOM, we have $\mathsf{HOM}(v_5) = \{f_4, f_5, f_6, f_7\}$, where $f_6 = \{u_1, u_3 \to v_4, u_2 \to v_5\}$ and $f_7 = \{u_1, u_3 \to v_6, u_2 \to v_5\}$.

Next, we show $|\mathsf{SubG}(v)| = |\mathsf{ISO}(v)|/c$ and $|\mathsf{InSubG}(v)| = |\mathsf{InISO}(v)|/c$ where c is a constant solely dependent on p. As a result, it suffices to evaluate $|\mathsf{ISO}(\cdot)|/|\mathsf{InISO}(\cdot)|$ to answer $\mathsf{SubG}/\mathsf{InSubG}$ queries. It is easy to verify that $\mathsf{SubG}(v) = \{G_f \mid f \in \mathsf{ISO}(v)\}$ and $\mathsf{InSubG}(v) = \{G_f \mid f \in \mathsf{InISO}(v)\}$ for $\forall v \in V$. Consider a subgraph $G_f \in \mathsf{SubG}(v)$. Let $F = \{f_1, f_2, \cdots, f_{|F|}\}$ be the isomorphisms in $\mathsf{ISO}(v)$ that induce G_f and let A be the automorphisms g of p that satisfy $g(o) = o$. We have $|F| = |A|$. Indeed, (i) for $\forall g \in A$, the function composition $f_1 \circ g$ of f_1 and g is an isomorphism that induces G_f and maps o to v; hence, for $\forall g \in A$, $f_1 \circ g$ belongs to F, implying $|F| \geq |A|$; and (ii) for $\forall f_i \in F$, with f_1^{-1} being the left inverse of f_1, $f_1^{-1} \circ f_i$ is an automorphism mapping o to o and thus is in A, implying $|F| \leq |A|$. In light of these, we have $|\mathsf{SubG}(v)| = |\mathsf{ISO}(v)|/|A|$ and can show $|\mathsf{InSubG}(v)| = |\mathsf{InISO}(v)|/|A|$ in a similar manner. We thus focus on queries of types ISO and InISO instead hereafter.

Counting subgraphs by $Q = (p, o, t)$ is challenging since there are a large number of combinations for pattern graphs p with the orbits o selected. We show the number of patterns in Table **??** that need to be counted for a k-node pattern graph for $k \leq 7$. There are many different pattern graphs, p, that need to be counted, as shown in the first row with none orbits selected. In the following rows, we show the number of combinations with some orbits selected. The second row shows the number of combinations with 1 node orbit. For 3-node pattern graph (e.g., $\{u, v, w\}$), there are 2 possible connected graphs: a tree with 2 edges and a triangle with 3 edges. In the tree there are 2 orbits, and in the triangle there is 1 orbit. In total, the number of combinations with 1 node orbit selected is 3. The 11 combinations for 4-node pattern graphs with 1 node orbit are shown in Fig. 2, where the black node is the orbit. The third row shows the number

Table 2: Subgraph Counting Approaches

Name	Approach	k-graph	Orbit	Flexible?	Sharing?	Platform	Lang
PTE [57]	Enumeration	3 (some)	any	✓	✗	Distributed	Java
DIST [30]	Decomposition	3, 4	1 (Node)	✗	✓	Distributed	C++
ORCA [38, 39]	Matrix	≤ 5	1 (Node)	✗	✓	Serial	C++
EVOKE [58]	Decomposition	≤ 5	1 (Node)	✗	✓	Serial	C++
ECLOG [26]	Matrix	≤ 5 (some)	2 (Edge)	✗	✓	Multi-core	C++
BENU [81]	Enumeration	5, 6 (some)	any	✓	✗	Distributed	Java
JESSE [49, 48, 50]	Matrix	any	1 (Node)	✗	✓	Serial	Java
DISC (ours)	Decomposition	any	any	✓	✓	Distributed	Scala

Table 3: The Total # of Matches for the 11 4-Node Graphs

	The Total # of Matches			The # of Matches Enumerated			
Dataset	# Ind-Matches	# Ind-Iso	# Homo	Enum	JESSE	DIST	DISC
WB	3.9×10^{14}	2.30×10^{15}	2.30×10^{15}	3.9×10^{14}	2.0×10^{11}	9.1×10^{11}	1.4×10^{11}
AS	9.3×10^{13}	5.83×10^{14}	5.84×10^{14}	9.3×10^{13}	1.1×10^{11}	2.5×10^{11}	6.9×10^{10}
LJ	8.9×10^{12}	4.58×10^{13}	4.59×10^{13}	8.9×10^{12}	5.2×10^{10}	3.1×10^{11}	2.8×10^{11}
OK	1.2×10^{14}	6.46×10^{14}	6.47×10^{14}	1.2×10^{14}	3.2×10^{11}	1.7×10^{12}	2.8×10^{11}
UK	2.0×10^{15}	1.20×10^{16}	1.20×10^{16}	2.0×10^{15}	1.4×10^{12}	1.5×10^{13}	4.2×10^{12}

of combinations with 2 node orbits selected, assuming that the 2 nodes are connected by an edge in p. The fourth row shows the number of combinations with 3 node orbits selected assuming the 3 node orbits form a triangle in p. And the last row shows the number of patterns with any 2 nodes selected as orbits. It is important to note that the number of orbits can be larger than 2, and the value of k for a k-node pattern graph can be larger than 7.

3. THE EXISTING APPROACHES

In a recent survey on subgraph counting [63], Ribeiro et al. categorized subgraph counting approaches into *enumeration-based*, *matrix-based*, and *decomposition-based* approaches. To be specific, (i) an enumeration-based approach is by name to count by enumerating all matches of a pattern graph p in a data graph G; (ii) a matrix-based approach is to count a collection $\mathcal{P}$ of pattern graphs by enumerating a subset of $\mathcal{P}$ and obtaining the remaining counts by solving linear equations; and (iii) a decomposition-based approach is to count by decomposing p into smaller graphs.

We list some representative approaches in Table 2, all of which are to handle (induced) subgraph counting. We shall discuss them in the following 7 aspects: (1) *Approach*: This column shows the category an approach belongs to. (2) *k-graph*: This column shows the connected k-node pattern graphs (p) that an approach is able to handle. (3) *Orbit*: This column is about the orbit (o) an approach is able to deal with. An approach that handles edge orbits (e.g., ECLOG) is different from an approach for node orbits (e.g., ORCA, EVOKE, ECLOG, BENU). They count different things. Note that PTE can only support 1 out of 2 3-node pattern graphs; ECLOG supports 32 out of 57 5-node pattern graphs with edge orbits; and BENU supports only 4 out of 21 5-node pattern graphs and 5 out of 112 6-node pattern graphs. (4) *Flexible?*: This column indicates whether all k-node pattern graphs allowed need to be computed altogether or not. DIST, ORCA, EVOKE, ECLOG, and JESSE must compute all possible k-node pattern graphs at the same time. That is, it cannot compute some of them selectively, even if some patterns are not useful for applications. (5) *Sharing?*: This column indicates if an approach allows sharing cost when counting patterns. (6) *Platform*: This column tells the platform (e.g., serial, multi-core, distributed) the approach is developed for. (7) *Lang*: This aspect is about the programming language used to implement the approach.

Next, we discuss the efficiency of the three approaches, namely enumeration, matrix, and decomposition. Suppose that we need to compute all the 11 4-node pattern graphs with node orbit (Fig. 2) based on enumeration, JESSE (matrix), DIST (decomposition), and our DISC (decomposition).

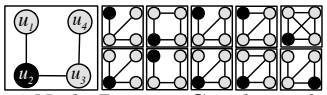


Figure 2: 11 4-Node Pattern Graphs with Node Orbit

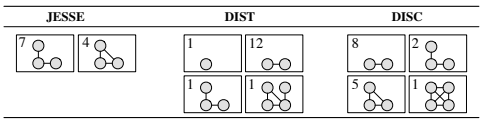


Figure 3: Patterns Enumerated by JESSE/DIST/DISC

To count induced subgraphs by enumeration, it needs to enumerate all the 11 4-node pattern graphs, which is costly. Different from the enumeration approach, the patterns that JESSE, DIST, and DISC need to enumerate are shown in Fig. 3. As shown in Fig. 3, a pattern to be enumerated is represented in a box with a number associated on the upper-left which indicates how many times the pattern needs to be enumerated for counting the 11 4-node pattern graphs. The patterns that need to enumerate are small in size, and the number of such patterns is much less than 11.

We show the cost of 5 datasets being tested (Table 4) in Table 3. In Table 3, on the left, we show the total # of matches over the 11 4-node graphs in Fig. 2. Here, "# Ind-Matches" is the total number of induced subgraphs matched. For example, for WB, which is a rather small graph with 1.3×10^7 edges, there exist $3.9 \cdot 10^{14}$ matches in total. On the right in Table 3, we show the # of matches enumerated by enumeration (Enum), JESSE, DIST, and our DISC. For Enum, the numbers are exactly those in the column "# of Induced Matches". For JESSE, DIST, and DISC, the numbers shown are the actual numbers of matches they need to enumerate. For all 4-node pattern graphs, JESSE is efficient, because JESSE uses an index to keep all possible 2-hop paths. Such an index is not effective in counting a k-node pattern for $k > 4$. As shown in our experimental studies, JESSE does not perform well when $k > 4$.

Some remarks can be made for matrix/decomposition-based approaches, for $k > 4$. All the existing approaches have limitations to support large k. And almost all of them are designed for a specific type of orbit (e.g., either node orbit or edge orbit). The main reason is due to the constraints imposed on induced subgraph counting by subgraph isomorphism. We use DIST as an example to explain such constraints for the leftmost 4-node pattern graph, p, in Fig. 2. At the node $f(u_2)$ in G, which u_2 maps to, there must have three edges $(f(u_2), f(u_3))$, $(f(u_2), f(u_1))$, $(f(u_3), f(u_4))$, and there must not have edges $(f(u_1), f(u_3))$, $(f(u_1), f(u_4))$, $(f(u_2), f(u_4))$. In addition, it needs to ensure that there are no other matchings by homomorphism such as $f(u_2) \neq f(u_4)$, $f(u_1) \neq f(u_4)$, and $f(u_1) \neq f(u_3)$. It comes with high overhead for a matrix/decomposition-based approach to ensure such constraints. For matrix-based approaches, they utilize the relationship among pattern graphs to reduce the enumeration cost. In other words, they are designed to compute *all* k-node pattern graphs together. For example, ECLOG [26] enumerates 14 out of 32 5-node pattern graphs and computes the remaining 18 based on some equations. ECLOG handles queries that take an edge as orbit (o) in a pattern graph p, and therefore it cannot be directly applied to count 5-node pattern graphs with node orbit. It needs effort to design a matrix-based approach for any $k > 5$ and any orbit, and the effectiveness is less observed if it is used to compute some but not all k-node pattern graphs. For decomposition-based approaches, they

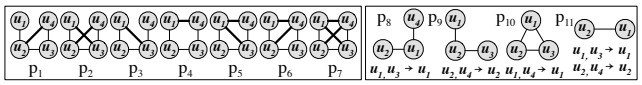


Figure 4: Sup(p) and Sub(p)

are flexible, since they are designed to decompose a pattern graph p into smaller graphs, and count p by counting the small graphs obtained. For 4-node pattern graphs, DIST [30] enumerates the patterns listed in Fig. 3 to compute all 4-node patterns. But it does not support $k > 4$. EVOKE supports $k = 5$, but it is difficult to extend it to $k > 5$, as it deal with every pattern specifically.

4. A NEW DECOMPOSITION APPROACH

HOM queries are usually much easier to handle than ISO and InISO queries. Indeed, unlike isomorphisms, homomorphisms are less restrictive in that they are not required to be injective. For a pattern graph p and a node orbit o of p, as will be detailed later, an InISO/ISO query Q is always systematically decomposable into a set of HOM queries $(p' = (V_{p'}, E_{p'}), o', \mathsf{HOM})$ with $|V_{p'}| \leq |V_p|$. Such a decomposition involves two conversions: one from InISO to ISO and the other from ISO to HOM. We elaborate them below.

From InISO to ISO. We first show how to reduce an InISO query to ISO queries. To this end, consider a pattern graph p and a node orbit o of p. Recall that an isomorphism of p can be either induced or non-induced. Therefore, we have

$$|\mathsf{ISO}(v)| = |\mathsf{InISO}(v)| + |\not\mathsf{InISO}(v)| \tag{1}$$

for $\forall v \in V$, where $\not\mathsf{InISO}(v)$ is the non-induced counterpart of $\mathsf{InISO}(v)$; that is, $\not\mathsf{InISO}(v) = \{f \mid f$ is a *non-induced isomorphism* of p with $f(o) = v\}$. In order to deal with the term $|\not\mathsf{InISO}(v)|$, we introduce the notion of *superP* below.

Definition 4.1: (SuperP) Consider a pattern graph $p = (V_p, E_p)$. Let $\overline{E_p} = (V_p \times V_p) \setminus E_p$; that is, $\overline{E_p}$ consists of the edges in the complement of p. A superP of p is defined as a graph that is obtainable by adding into p at least one edge from $\overline{E_p}$. Therefore, there are in total $2^{|\overline{E_p}|} - 1$ superPs of p. We use $\mathsf{Sup}(p)$ to denote the set of all superPs of p.

Example 4.1: For the pattern graph p in Fig. 1, $\overline{E_p}$ comprises 3 pairs, namely (u_1, u_4), (u_1, u_3) and (u_2, u_4). There are seven superPs for p, which are shown in Fig. 4.

Proposition 4.1: *f is a non-induced isomorphism of p iff f is an induced isomorphism of some superP of p.*

Proof: We first prove the $\Rightarrow$ part. Since f is non-induced, there exists at least one pair of nodes (u, u') in V_p such that the edge $(u, u') \notin E_p$ but $(f(u), f(u')) \in E$. Let $S \subseteq \overline{E_p}$ be the set of all such pairs and let $p' = (V_p, E_p \cup S)$. Then, f is an induced isomorphism of p'. Indeed, (i) f is an isomorphism of p' in that f is a homomorphism and is injective; and (ii) the iso-match of p' induced by f is an induced subgraph of G. We next prove the $\Leftarrow$ part. Let p' be the corresponding superP. Removal of any edge from p' makes f non-induced. Therefore, f is non-induced for p. □

Example 4.2: Consider G and p in Fig. 1. The non-induced isomorphism $\{u_1 \to v_3, u_2 \to v_2, u_3 \to v_1, u_4 \to v_4\}$ of p is an induced isomorphism of the superP p_3 (Fig. 4) of p.

According to Proposition 4.1, we have

$$|\not\mathsf{InISO}_{p,o}(v)| = \sum_{p' \in \mathsf{Sup}(p)} |\mathsf{InISO}_{p',o}(v)| \tag{2}$$

By combining Eq. (1) and (2), we have

$$|\mathsf{ISO}_{p,o}(v)| = \sum_{p' \in \{p\} \cup \mathsf{Sup}(p)} |\mathsf{InISO}_{p',o}(v)| \tag{3}$$

Since $\{p\} \cup \mathsf{Sup}(p)$ is a *poset* under edge inclusion, by Eq. (3) and the Möbius inversion formula, we have

$$|\mathsf{InISO}_{p,o}(v)| = \sum_{p' \in \{p\} \cup \mathsf{Sup}(p)} \mu_1(p, p') \cdot |\mathsf{ISO}_{p',o}(v)| \tag{4}$$

where $\mu_1(p, p') = (-1)^{|E_{p'}| - |E_p|}$ is the corresponding Möbius function. With Eq. (4), an InISO query on p can be decomposed into several ISO queries on $\{p\} \cup \mathsf{Sup}(p)$.

Example 4.3: Continuing with Example 4.1, we have

$$\begin{aligned} |\mathsf{InISO}_p(v)| = {} & |\mathsf{ISO}_p(v)| - |\mathsf{ISO}_{p_1}(v)| + |\mathsf{ISO}_{p_2}(v)| \\ & - |\mathsf{ISO}_{p_3}(v)| - |\mathsf{ISO}_{p_4}(v)| + |\mathsf{ISO}_{p_5}(v)| \\ & + |\mathsf{ISO}_{p_6}(v)| - |\mathsf{ISO}_{p_7}(v)| \end{aligned}$$

which works with any node orbit of p.

From ISO to HOM. We next show how to decompose an ISO query $Q = (p, o, \mathsf{ISO})$ into HOM queries. Recall that an isomorphism is defined to be an injective homomorphism. Hence, for a pattern graph p and a node orbit o, we have

$$|\mathsf{HOM}(v)| = |\mathsf{ISO}(v)| + |\not\mathsf{inj}\mathsf{HOM}(v)| \tag{5}$$

where $\not\mathsf{inj}\mathsf{HOM}(v) = \{f \mid f$ is a *non-injective* homomorphism of p with $f(o) = v\}$. By definition, for $\forall f \in \not\mathsf{inj}\mathsf{HOM}(v)$, there exist nodes of p mapped by f to the same nodes in G. Moreover, for the nodes in p that are mapped to the same node in G, they must be independent since otherwise, loops must be present in G to make f a homomorphism.

Definition 4.2: (SubP) Consider a pattern graph $p = (V_p, E_p)$. Let $\mathcal{I} = \{I_1, I_2, \ldots, I_k\}$ be a partition of V_p such that for $\forall i \in [1, k]$, I_i is an independent set of p; that is, there does not exist an edge in p between any two nodes of I_i. A subP graph $p' = (V_{p'}, E_{p'})$, induced by $\mathcal{I}$, of p is defined as follows: (i) $k < |V_p|$; (ii) p' has exactly k nodes; specifically, $V_{p'} = \{w_1, w_2, \ldots, w_k\}$ with w_i corresponding to I_i; and (iii) there is an edge between w_i and w_j if and only if there is an edge in p between some node of I_i and some node of I_j. For notational convenience, we also use w_i to denote the corresponding I_i of w_i. Moreover, we use $o(p')$ to denote the node $w \in V_{p'}$ that contains the orbit o. We denote the set of all subPs of p by $\mathsf{Sub}(p)$.

Example 4.4: Fig. 4 shows $\mathsf{Sub}(p)$ for the pattern graph p in Fig. 1. Specifically, the subPs p_8, p_9, p_{10} and p_{11} are induced by the partitions $\mathcal{I}_1 = \{\{u_1, u_3\}, \{u_2\}, \{u_4\}\}$, $\mathcal{I}_2 = \{\{u_2, u_4\}, \{u_1\}, \{u_3\}\}$, $\mathcal{I}_3 = \{\{u_1, u_4\}, \{u_2\}, \{u_3\}\}$, and $\mathcal{I}_4 = \{\{u_1, u_3\}, \{u_2, u_4\}\}$, respectively. In addition, if the node orbit o of p is u_2, then $o(p_8)$ and $o(p_9)$ refer to the node $\{u_2\}$ in p_8 and $\{u_2, u_4\}$ in p_9, respectively.

Proposition 4.2: *For pattern graph p and orbit o, we have*

$$|\not\mathsf{inj}\mathsf{HOM}_{p,o}(v)| = \sum_{p' \in \mathsf{Sub}(p)} |\mathsf{ISO}_{p', o(p')}(v)|$$

Proof: We first prove the $\leq$ part. Consider a homomorphism $f \in \not\mathsf{inj}\mathsf{HOM}_{p,o}(v)$. By definition, f is non-injective and maps o to v. Let $\mathcal{R}_f = \{v_1, v_2, \ldots, v_k\}$ be the range of f, i.e., $\mathcal{R}_f = \{f(u) \mid u \in V_p\}$. We have $|\mathcal{R}_f| = k < |V_p|$ because f is non-injective. For each $v_i \in \mathcal{R}_f$, let $I_i = \{u \in V_p \mid v_i = f(u)\}$ be the set of nodes in p that are mapped by

f to v_i. It is easy to see that I_i's are all independent sets and constitute a partition of V_p. Let $\mathcal{I}_f = \{I_1, I_2, \ldots, I_k\}$ and $p_f = (V_{p_f}, E_{p_f})$ be the subP of p induced by $\mathcal{I}_f$. We further define a function f' from V_{p_f} to $\mathcal{R}_f$ such that for any node w in p_f, $f'(w) = f(u)$, where u is any node contained in w. In this way, we make f' an isomorphism of p_f that maps $o(p_f)$ to v. Indeed, by construction, (i) f' is an injective homomorphism of p_f; and (ii) $f'(o(p_f)) = f(o) = v$. It is worth pointing out that our mapping described above from f to a pair (p_f, f') is injective. As a result, we have

$$|\mathsf{InjHOM}_{p,o}(v)| \leq \sum_{p' \in \mathsf{Sub}(p)} |\mathsf{ISO}_{p',o(p')}(v)|$$

We next prove the $\geq$ part. Let f' be an isomorphism of a subP p' of p that maps $o(p')$ to v. We construct f such that for $\forall u \in V_p$, $f(u) = f'(w)$ where w is the node in p' containing u. This f can be easily shown to be a non-injective homomorphism of p. Moreover, $f(o) = f'(o(p')) = v$. Because the mapping above from (p', f') to f is injective,

$$|\mathsf{InjHOM}_{p,o}(v)| \geq \sum_{p' \in \mathsf{Sub}(p)} |\mathsf{ISO}_{p',o(p')}(v)|$$

By the two inequalities above, the proposition is proved. □

By Eq. (5) and Proposition 4.2, we have

$$|\mathsf{HOM}_{p,o}(v)| = \sum_{p' \in \{p\} \cup \mathsf{Sub}(p)} |\mathsf{ISO}_{p',o(p')}(v)| \quad (6)$$

To solve this equation, we define a partial order $\leq$ over $\{p\} \cup \mathsf{Sub}(p)$ such that for any p' and p'' in $\{p\} \cup \mathsf{Sub}(p)$, $p' \leq p''$ if and only if either p'' is exactly p' or p'' is a subP of p'. As a result, for the poset $(\{p\} \cup \mathsf{Sub}(p), \leq)$, an interval $[p, p'] = \{p'' \mid p \leq p'' \leq p'\}$ is essentially a product of partition lattices. Therefore, its corresponding Möbius function is $\mu_2(p, p') = \prod_{I \in \mathcal{I}} (-1)^{|I|-1}(|I| - 1)!$, where $\mathcal{I}$ is the corresponding partition for p'. Particularly, $\mu_2(p, p) = 1$. By Eq. (6) and the Möbius inversion formula, we have

$$|\mathsf{ISO}_{p,o}(v)| = \sum_{p' \in \{p\} \cup \mathsf{Sub}(p)} \mu_2(p, p') \cdot |\mathsf{HOM}_{p',o(p')}(v)| \quad (7)$$

Example 4.5: Continuing with Example 4.4, we have

$$|\mathsf{ISO}_p(v)| = |\mathsf{HOM}_p(v)| - |\mathsf{HOM}_{p_8}(v)| - |\mathsf{HOM}_{p_9}(v)| - |\mathsf{HOM}_{p_{10}}(v)| + |\mathsf{HOM}_{p_{11}}(v)|$$

which works with any node orbit of p.

Putting It All Together. By applying Eq. (7) to each term on the right-hand side of Eq. (4), we can obtain a formula of the following form to evaluate $|\mathsf{InISO}_{p,o}(v)|$:

$$|\mathsf{InISO}_{p,o}(v)| = \sum_{p' \in \mathsf{Enum}(p)} \alpha_{p'} \cdot |\mathsf{HOM}_{p',o(p')}(v)| \quad (8)$$

where $\alpha_{p'}$ is the corresponding coefficient for p' and $\mathsf{Enum}(p)$ contains all the graphs whose coefficients are non-zero.

5. HOMOMORPHISM COUNTING BY RELATIONAL ALGEBRA

As shown in Section 4, an InISO/ISO query is always decomposable into a collection of HOM queries in the form of linear combination. In light of this fact, in this section, for a pattern graph p, a node orbit o and a node $v \in V$, we show how to systematically evaluate $|\mathsf{HOM}_{p,o}(v)|$ by relational algebra, irrespective of $|V_p|$.

u_1	u_2	u_3	u_4
v_5	v_6	v_5	v_4
v_5	v_6	v_5	v_6
...	...	...	...

Figure 5: R_{HOM}

o	C
v_6	2
v_5	6
...	...

Figure 6: R_p

Suppose that the undirected data graph G is maintained in a relation $R_G(from, to)$ with two attributes $from$ and to such that for each edge (v, v') of G, there are two tuples (v, v') and (v', v) in R_G. For each edge $e_i = (u, u')$ of p, let relation $R_i(u, u')$ with attributes u and u' be a virtual copy of R_G. It is easy to verify that every tuple of R_i essentially corresponds to a homomorphism of e_i. Thus, for each tuple (v, v') in R_i, the mapping $f = (u \to v, u' \to v')$ is a homomorphism of $e_i = (u, u')$. To this end, all the homomorphisms of p can be obtained by conducting a series of natural joins on $R_1, R_2, \ldots, R_{|E_p|}$ [51, 13, 41]. Specifically, let

$$R_{\mathsf{HOM}} = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_{|E_p|} \quad (9)$$

Then, by construction, (i) R_{HOM} has $|V_p|$ attributes, one for each node in V_p; and (ii) R_{HOM} contains all and only the homomorphism of p. With R_{HOM} at hand, we can compute $|\mathsf{HOM}_{p,o}(v)|$ for all nodes v in G as follows:

$$R_p(o, C) = {}_o\Upsilon_{\mathsf{count}(*) \to C}(R_{\mathsf{HOM}}) \quad (10)$$

Here, ${}_o\Upsilon_{F \to A}(\cdot)$ is to apply the aggregate function F to each group partitioned according to the group-by attribute o and name the column corresponding to F as A. The resulting relation R_p contains two columns, o for the nodes v in G and C for their corresponding homomorphism counts $|\mathsf{HOM}_{p,o}(v)|$. If a node v is not present in R_p, then $|\mathsf{HOM}_{p,o}(v)| = 0$.

Example 5.1: There are 3 edges in the pattern graph p (Fig. 1), namely $e_1 = (u_1, u_2)$, $e_2 = (u_2, u_3)$ and $e_3 = (u_3, u_4)$. Accordingly, we have 3 relations with schemas $R_1(u_1, u_2)$, $R_2(u_2, u_3)$ and $R_3(u_3, u_4)$. We show $R_{\mathsf{HOM}} = R_1 \bowtie R_2 \bowtie R_3$, which contains the homomorphisms of p, in Fig. 5. With u_2 as the orbit, $R_p = {}_{u_2}\Upsilon_{\mathsf{count}(*)}(R_{\mathsf{HOM}})$ is shown in Fig. 6, recording for each v its $|\mathsf{HOM}_{p,u_2}(v)|$.

As discussed above, we can obtain $R_{p'}(o, C)$ for every pattern graph p' in $\mathsf{Enum}(p)$ by Eq. (9) and Eq. (10). To show how to compute Eq. (8), for easy discussion, we define two relational operators.

Definition 5.1: Let $R(o, C)$ and $R'(o, C)$ be two relations, both with attributes o and C. (i) The $\psi_\alpha(\cdot)$ operator: $\psi_\alpha(R)$ makes a copy of R and multiplies all the values of column C by α. (ii) The $\biguplus$ operator: $R \biguplus R'$ is defined as

$$R \biguplus R' = {}_o\Upsilon_{\mathsf{sum}(C) \to C}(R \cup R')$$

Here, we assume the union operator $\cup$ keeps duplicates.

With the two operators, namely ψ_α and $\biguplus$, we compute Eq. (8) for all nodes in G by Eq. (11).

$$\biguplus_{p' \in \mathsf{Enum}(p)} \psi_{\alpha_{p'}}(R_{p'}) \quad (11)$$

6. DISTRIBUTED PROCESSING

There are many approaches proposed to process join query in a distributed system over a cluster of servers. The state-of-the-art approach we adopt is HCubeJoin [22], which outperforms multi-round binary joins [41] due to free of shuffling

on intermediate results. HCubeJoin is a one-round multi-join approach built on two algorithms, namely HCube [6, 16] and Leapfrog [78]. Here, HCube is a one-round communication optimal shuffling method that shuffles data to every server in the cluster. The main idea is to divide the output of a join query Q into cubes, and assign a cube to one of the servers to process by hashing. The assignment minimizes the communication cost with respect to the constraints of input database D, the join query Q, and the memory budget M on one server to hold both input tuples and the partial result of Q in memory. HCube is proven in theory to be the optimal in worst-case sense for transmitting the tuples to servers such that each server can evaluate the query without further data exchange. Leapfrog is a fast in-memory sequential multi-join algorithm to process the join query at each server over the data shuffled to it, based on the attribute order among the attributes of Q using iterators. Leapfrog is proven in theory to be the optimal method in worst-case sense to evaluate a join query Q, following AGM bound [14].

The problem we study is local subgraph counting, where additional group-by and aggregation are needed with joins. A naive approach is to enhance HCubeJoin by computing additional group-by and aggregation together with joins. In other words, we process group-by and aggregation while processing the underneath join together. However, such an approach is not efficient. First, HCube is a one-round communication optimal shuffling method assuming large intermediate results for joins. It does not consider how to reduce the communication cost when dealing with group-by and aggregation, which may result in small intermediate results. Second, Leapfrog processes a multi-join based on the attribute order among the attributes of Q using iterators. However, the group-by and aggregation blocks the pipeline mechanism behind Leapfrog as it needs to compute the group-by and aggregation completely before the next join. Note that the group-by and aggregation results can also be large. It has significant impacts on Leapfrog. In this paper, to the first, as the intermediate results can be possibly small with group-by and aggregation, we study a multi-round communication method to significantly reduce the communication cost of the one-round HCube. To the second, we investigate a new attribute order to facilitate group-by and aggregation computing in Leapfrog, and give a cache-oblivious approach to reduce the group-by and aggregation computing in a pipeline fashion. Note that previously it will block Leapfrog if it needs to store the group-by and aggregation results.

Below, for a HOM query p, in Section 6.1, first we discuss how to push the group-by and aggregation over a join tree for p, by representing p as a join tree using tree decomposition. The tree decomposition also improves efficiency in general. Second, we discuss multi-round HCubeJoin for p in Section 6.2 aiming at minimization of communication cost. Third, we discuss how to support join with group-by and aggregation in Leapfrog by pipeline in Section 6.3.

6.1 Pushing Aggregation over Joins

As the join cost will be the dominating factor, to reduce the intermediate results, we push group-by and aggregation down as much as possible, instead of computing group-by and aggregation after joins (Eq. (9) and Eq. (10)). We adopt the work in [83], which pushes group-by and aggregations over joins for an acyclic join query. In doing so, we convert

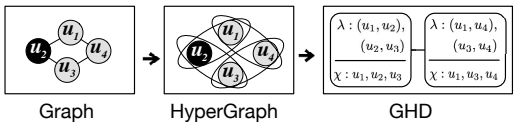


Figure 7: Hypergraph and GHD Decomposition

a cyclic join query to a join tree based on *generalized hypertree decomposition* (GHD) [32]. We introduce GHD and pushing aggregation over joins below in brief, and give the complexity.

Finding a Join Tree. GHD is the state-of-the-art approach for tree decomposition. As the name implies, it is done over a *hypertrophy* representation of a join query. Formally, for a join query $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k$ on k relations, its hypergraph $\mathcal{H}(V_\mathcal{H}, E_\mathcal{H})$, is defined as follows: (i) $V_\mathcal{H}$ contains the attributes of the k relations; (ii) for each R_i, $E_\mathcal{H}$ contains a hyperedge e_i connecting the attributes of R_i; and (iii) $E_\mathcal{H}$ contains no other hyperedges. For Eq. (9), its hypergraph is exactly p. GHD decomposes the hypergraph p into several smaller subqueries and obtains the final result by combining the results of those subqueries. In a nutshell, a GHD of p is a triple $(\mathcal{T}, \chi, \lambda)$ where $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$ is a tree. For each tree node τ of $\mathcal{T}$, it is required that $\chi(\tau) \subseteq V_p$ and $\lambda(\tau) \subseteq E_p$; that is, each tree node τ is associated with a set of relations (one for each edge in $\lambda(\tau)$) and a set of attributes $\mathsf{attrs}(\tau)$ (one for each node in $\chi(\tau)$). The labeling functions $\chi(\cdot)$ and $\lambda(\cdot)$ are not arbitrary. Specifically, they need to meet the following requirements for GHD and later aggregation push-down: (1) every node of p is contained in some tree node of $\mathcal{T}$; that is, $\bigcup_{\tau \in V_\mathcal{T}} \chi(\tau) = V_p$; (2) for each edge (u, u') of p, there exists a tree node τ such that $\{u, u'\} \subseteq \chi(\tau)$; (3) for each node u of p, the tree nodes τ with $u \in \chi(\tau)$ are connected in $\mathcal{T}$; (4) for each tree node τ, every node $u \in \chi(\tau)$ is contained in some edge $e \in \lambda(\tau)$; and (5) for aggregation push-down, there exists a tree node τ whose $\chi(\tau)$ fully contains the orbit o. We make $\mathcal{T}$ rooted by selecting a tree node τ whose $\chi(\tau)$ contains the orbit o as the root $\mathsf{root}(\mathcal{T})$.

Example 6.1: In Fig. 7, we show one GHD $(\mathcal{T}, \chi, \lambda)$ for the hypergraph p. The hypertree $\mathcal{T}$ contains only two nodes, namely τ_1 and τ_2. Inside each τ_i, the corresponding $\chi(\tau_i)$ and $\lambda(\tau_i)$ are shown. We can easily verify that both $\chi(\cdot)$ and $\lambda(\cdot)$ satisfy the requirements mentioned above. In particular, each hyperedge of p appears in at least one node. In other words, the relations are distributed among the nodes. τ_1 is selected as the root because it contains the orbit u_2.

Let $\mathsf{rel}(\tau)$ be the result of joining the relations in τ. The joins in Eq. (9) can be reduced to joins on $\mathsf{rel}(\tau)$'s.

$$R_1 \bowtie R_2 \bowtie \cdots \bowtie R_{|E_p|} = \bowtie_\tau \mathsf{rel}(\tau) \tag{12}$$

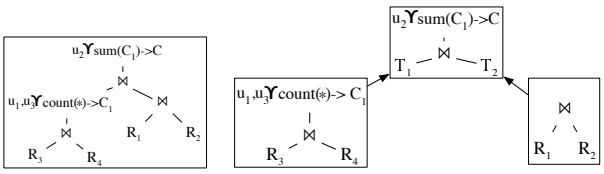
Here, the joins on $\mathsf{rel}(\tau)$'s are acyclic and can be efficiently computed in a bottom-up manner. Specifically, let $\mathcal{T}_\tau$ be the subtree of $\mathcal{T}$ rooted at τ and let $\mathsf{rel}(\mathcal{T}_\tau)$ be the result of joining the $\mathsf{rel}(\tau')$'s of the descendants τ' of τ. $\bowtie_\tau \mathsf{rel}(\tau) = \mathsf{rel}(\mathcal{T})$ can be evaluated recursively as follows. If τ is a leaf,

$$\mathsf{rel}(\mathcal{T}_\tau) = \mathsf{rel}(\tau) \tag{13}$$

Otherwise, given that $\tau_1, \ldots, \tau_k$ are the children of τ,

$$\mathsf{rel}(\mathcal{T}_\tau) = \mathsf{rel}(\tau) \bowtie \mathsf{rel}(\mathcal{T}_{\tau_1}) \bowtie \cdots \bowtie \mathsf{rel}(\mathcal{T}_{\tau_k}) \tag{14}$$

Example 6.2: Consider Fig. 7. Suppose $e_1 = (u_1, u_2)$, $e_2 = (u_2, u_3)$, $e_3 = (u_3, u_4)$ and $e_4 = (u_4, u_1)$. Then, we have $\mathsf{rel}(\tau_1) = R_1 \bowtie R_2$ and $\mathsf{rel}(\tau_2) = R_3 \bowtie R_4$.



(a) One-round (b) Multi-round

Figure 8: One-round vs. Multi-round HCube

Pushing Aggregation over Joins. With [83], we push down group-by and aggregation by Eq. (15) and Eq. (16) for Eq. (13) and Eq. (14), respectively.

$$\mathsf{rel}(\mathcal{T}_\tau) = {}_{A_\tau}\Upsilon_{count(*)\to C}(\mathsf{rel}(\tau)) \quad (15)$$

$$\mathsf{rel}(\mathcal{T}_\tau) = {}_{A_\tau}\Upsilon_{sum(C_1*\cdots*C_k)\to C}(\mathsf{rel}(\tau) \bowtie \rho_{C\to C_1}(\mathsf{rel}(\mathcal{T}_{\tau_1})) \bowtie \cdots \bowtie \rho_{C\to C_k}(\mathsf{rel}(\mathcal{T}_{\tau_k}))) \quad (16)$$

Here, the rename operator $\rho_{A\to B}$ renames A as B. The main idea behind is as follows. Since not all attributes of $\mathsf{rel}(\mathcal{T}_\tau)$ are needed in the bottom-up computation, let τ' be the parent of τ, it suffices to only keep the columns A_τ of $\mathsf{rel}(\mathcal{T}_\tau)$, where $A_\tau = \mathsf{attrs}(\tau) \cap \mathsf{attrs}(\tau')$ for non-root nodes and $= \{o\}$ for $\mathsf{root}(\mathcal{T})$. This is due to the requirement (3) of a GHD that an attribute in $\mathsf{attrs}(\tau) \setminus \mathsf{attrs}(\tau')$ does not appear in any tree node of $\mathcal{T}$ except those in the subtree $\mathcal{T}_\tau$.

Example 6.3: Let us continue with Example 6.2. Because $\mathsf{attrs}(\tau_1) = \{u_1, u_2, u_3\}$ and $\mathsf{attrs}(\tau_2) = \{u_1, u_3, u_4\}$, we have $A_{\tau_2} = \mathsf{attrs}(\tau_1) \cap \mathsf{attrs}(\tau_2) = \{u_1, u_3\}$. Moreover, since the orbit is u_2, $A_{\tau_1} = \{u_2\}$. The resulting query processing plan after pushing down aggregation is shown in Fig. 8(a).

Complexity Analysis. We give a complexity analysis of the recursive Eq. (15) and Eq. (16). Given a GHD $(\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T}), \chi, \lambda)$ for p, the total cost can be partitioned into two parts: (i) the cost to evaluate $\mathsf{rel}(\tau)$ for each tree node $\tau \in V_\mathcal{T}$ and (ii) the cost to evaluate the joins along the tree edges, i.e., the joins between a node and its children.

With *fractional hypertree width* [32], for our GHD $(\mathcal{T}, \chi, \lambda)$, its fractional hypertree width, denoted by fhw, is the minimum real number such that every tree node has a fractional edge cover of weight fhw. Recall that the relations R_i all have size $2m$. For each tree node τ, since its fractional edge cover is bounded by fhw, by [14] the size of $\mathsf{rel}(\tau)$ is bounded by $(2m)^{\mathsf{fhw}}$. Furthermore, it requires only $O((2m)^{\mathsf{fhw}})$ time to evaluate $\mathsf{rel}(\tau)$ using an algorithm (e.g., Leapfrog [78]). Therefore, the cost to evaluate all $\mathsf{rel}(\tau)$'s is $O\left(|V_\mathcal{T}| \cdot (2m)^{\mathsf{fhw}}\right)$. As for the cost to evaluate the joins along the tree edges, there are $|E_\mathcal{T}|$ such joins. And each such join is conducted on two relations of size at most $(2m)^{\mathsf{fhw}}$. Hence, each of the joins takes $O((2m)^{\mathsf{fhw}})$ time, resulting in $O\left(|E_\mathcal{T}| \cdot (2m)^{\mathsf{fhw}}\right)$ total time. Ours takes $O\left(|V_p| \cdot (2m)^{\mathsf{fhw}}\right)$ time, where we use the facts $|E_\mathcal{T}| = |V_\mathcal{T}| - 1$ and $|V_\mathcal{T}| \le |V_p|$.

By brute-force enumeration, we find that for any pattern graph p with $|V_p| \le 7$, there always exists a generalized hypertree decomposition with fhw as small as $|V_p|/2$. Therefore, for any small graph p with node orbit o, its local homomorphism counts are computable in $O(m^{|V_p|/2})$ time.

6.2 Multi-Round HCubeJoin

In this section, we focus on minimizing communication cost of the one-round HCubeJoin. The one-round HCubeJoin shuffles input relations over servers once (Fig. 8(a)), whereas a multi-round HCubeJoin we propose will shuffle data multiple times (3 times in Fig. 8(b)), if it can further reduce the communication cost.

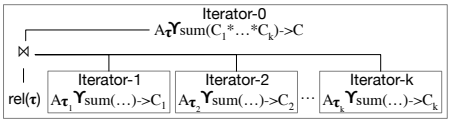


Figure 9: Iterators of Leapfrog with Aggregation

First, we explain the high communication cost of one-round HCubeJoin, or precisely HCube. Such a cost is related to how many duplications of every input relation to be distributed over servers in a one-round manner, regarding the given query p [6]. It is to minimize

$$cost(p) = \sum_{R_i \in p} |R_i| \times d(R_i, \phi) \quad (17)$$

Here, R_i is an input relation, and $d(R, \phi)$ is a duplication ratio to duplicate R_i as $d(R, \phi) = \prod_{A \in \mathsf{attrs}(p) \setminus \mathsf{attrs}(R_i)} \phi_A$, where ϕ_A is the number of partitions for attribute A to be duplicated on servers. In brief, a complex join query p with more attributes tends to result in a larger duplication ratio, which in return results in high communication cost.

With additional group-by and aggregation, multi-round HCube can significantly reduce the cost for two reasons: the input relation (e.g., the intermediate group-by and aggregation results) can be possibly small, and the intermediate input query can be small with less number of attributes involved as shown in Fig. 8(b).

The optimization problem here, based on Eq. (17), becomes how to cut the query (e.g., Fig. 8(a)) into smaller sub-queries so that the total communication cost for all subqueries divided is the smallest. However, this needs to explore all possible combination of cuts of the query with an estimation of the intermediate results, and it is difficult to get an accurate estimation without overhead. We adopt a heuristic to decide where to cut based on the size of an intermediate result referring to Eq. (15) and Eq. (16). Consider the group-by attributes (A_τ) of $\mathsf{rel}(\mathcal{T}_\tau)$ for a subtree $\mathcal{T}_\tau$ of $\mathcal{T}$. The size of $\mathsf{rel}(\mathcal{T}_\tau)$ is the size of the join over the relations in $\mathsf{rel}(\mathcal{T}_\tau)$ that contain at least one attribute in A_τ. The size of the latter is bounded by $O(m)$ either when $|A_\tau| = 1$ or when $|A_\tau| = 2$ where a relation in $\mathcal{T}_\tau$ contains all A_τ, following AGM bound [14]. Note that all relations in our setting are of 2 attributes and a relation is at most $2m$ for an undirected graph with m edges. Otherwise, the size of $\mathcal{T}_\tau$ will be at least $O(m^2)$ that cannot be a candidate to cut, since it will result in large communication cost in the next round. As an indicator, for 74 of HOM queries with 1 node orbit (e.g., 2nd row in Table **??** for $k \le 5$), almost all can be cut into smaller queries. We confirm the efficiency in our experimental studies.

6.3 Leapfrog: Join with Aggregation

Leapfrog is an in-memory algorithm to process joins in a pipeline fashion by expanding attributes one-by-one based on an attribute order using iterators, where all input relations are in memory and there is no I/O cost. We call the attribute order as the j-order, since it is to minimize the join cost. Consider the three joins $\cdots R_1(u_1, u_2) \bowtie R_2(u_2, u_3) \bowtie R_3(u_3, u_4) \bowtie R_4(u_1, u_4) \cdots$ in a join query. A possible j-order is $u_2 \prec u_1 \prec u_3 \prec u_4$. There are two implications by a j-order. First, all relations are sorted following the j-order. For example, $R_1(u_1, u_2)$ is sorted by u_2 followed by

u_1 since $u_2 \prec u_1$. Second, Leapfrog will extend an attribute following the order. In other words, to extend from an attribute u_i to the immediate next attribute u_j by the j-order is: both the intermediate result with u_i extended and the next relation have u_j to join. This condition is to avoid Cartesian product. With a j-order, Leapfrog uses iterators to join. Conceptually an iterator iter(u_i) is created for every attribute u_i in a pipeline fashion: iter(u_i) calls iter(u_j) with a specific u_i value, and iter(u_j) will return tuples to iter(u_i) that can join with the specific u_i value in pipeline.

The question is how to support additional group-by and aggregation with the fast Leapfrog algorithm in a pipeline fashion. It is important to note that group-by and aggregation block such pipeline. Consider the following query,

$$\cdots (R_1 \bowtie R_2) \ \bowtie \ {}_{(u_1,u_3)}\Upsilon_{sum(\cdot)}(R_3 \bowtie R_4) \cdots$$

which is a simplification referring to Eq. (16). By blocking, ${}_{(u_1,u_3)}\Upsilon_{sum(\cdot)}(R_3 \bowtie R_4)$ (or ${}_{(u_1,u_3)}\Upsilon_{sum(\cdot)}$) needs to be completed first before the next joins. We explore a way following the pipeline mechanism in Leapfrog without blocking, and we find a correct way to do so, which is to do J-iter before A-iter. We explain it based on Eq. (16). Here, the A-iter is the iterator for $\mathsf{rel}(\mathcal{T}_\tau)$ on the left side, as it is to group A_τ attributes followed by aggregation, and the J-iter is the iterator for $\mathsf{rel}(\tau)$. Note that $\mathsf{rel}(\mathcal{T}_{\tau_i})$ will appear at the left side of the equation when it is called in the same manner. Following Eq. (16), in pipeline, it gets a tuple $t \in \mathsf{rel}(\tau)$ by the J-iter to request the count for $\rho_{C\to C_1}(\mathsf{rel}(\mathcal{T}_{\tau_1}))$, ..., $\rho_{C\to C_k}(\mathsf{rel}(\mathcal{T}_{\tau_k}))$ that can join the tuple t. By this, we make J-iter before A-iter without blocking, as A-iters have a target to compute count for a given tuple t. Otherwise, it needs to compute all counts for all groups by A-iters, which cannot be done by pipeline without blocking. As there is a need to compute group by group using iterators, we extend attributes by a collection of attributes at once using iterators in the modified Leapfrog.

It is important to note that the attribute order to process group-by and aggregation by iterations efficiently may not be consistent with the j-order used in Leapfrog. As the example above, in the A-iter, the group-by attributes are (u_1, u_3). This requests that the attribute from the other iterators beforehand shall put (u_1, u_3) as the first attributes in the attribute order, which we call the g-order. The reason is that, if the two group-by attributes are sorted as the first 2 attributes in order, one group of (u_1, u_3) will be passed to the A-iter at most once. In our modified Leapfrog, we attempt to find an attribute order that satisfies both j-order and g-order. If we cannot find such an order, we put j-order at a higher priority. We will also use a cache to cache the count for a group (e.g., (u_1, u_3)), if the order is not g-order.

In Fig. 9, we sketch how to build up iterators in the modified Leapfrog according to Eq. (16). The Iterator-0 will generate groups with $\mathsf{rel}(\tau)$, and pass the groups to other iterators to get their aggregation values. With the aggregation values collected, the Iterator-0 will compute its aggregation. Consider the following example,

$$\cdots {}_{u_2}\Upsilon_{sum(\cdot)}((R_1 \bowtie R_2) \ \bowtie \ {}_{(u_1,u_3)}\Upsilon_{sum(\cdot)}((R_3 \bowtie R_4)) \cdots$$

$\mathsf{rel}(\tau)$ corresponds to $R_1 \bowtie R_2$, Iterator-1 corresponds to ${}_{(u_1,u_3)}\Upsilon_{sum(\cdot)}(R_3 \bowtie R_4)$, and Iterator-0 is to compute ${}_{u_2}\Upsilon_{sum(\cdot)}$.

We confirm the efficiency of the new pipelined Leapfrog with aggregation in the experimental studies.

6.4 The Distributed System

We have implemented a distributed system, called DISC, on top of *Spark*, to process a batch of induced subgraph counting queries, $\mathcal{Q}$, over a data graph G. Here, a data graph G is stored in an edge table $R_G(from, to)$, and a counting query in $\mathcal{Q}$ is in the form of $Q = (p, o, t)$, where p is a graph, o is an orbit of p, and t is a query type (either InSubG for induced subgraph counting or SubG for subgraph counting).

In DISC system, for each $Q = (p, o, \mathsf{InSubG})$ in $\mathcal{Q}$, we generate its query plan J in two parts (a) and (b) as follows. In (a), for each pattern graph p' in $\mathsf{Enum}(p)$, we obtain the pair of $(\alpha_{p'}, p')$ with its join plan to evaluate p' regarding Eq. (9) and Eq. (10). In (b), with the set of such pairs in (a), we further have a plan to evaluate Q by Eq. (11). After we have all the plans for all Q in $\mathcal{Q}$, we evaluate them in two main steps. In the first step, we evaluate the part (a) for every query plan J for $Q \in \mathcal{Q}$. In evaluating $(\alpha_{p'}, p')$ in (a) for J, we keep its result in a relation $R_{p'}$, together with $\alpha_{p'}$. In the second step, we evaluate the part (b) for every counting query Q with the results stored. In this way, we can share the computing cost as many p' are common in counting queries. Due to the space limit, we omit such discussion. We implement the part (a) by ourselves together with *Spark-SQL* on top of *Spark*, which includes the implementation of the multi-round HCubeJoin and the modified Leapfrog with aggregation. The part (b) can be supported by *Spark-SQL*.

7. RELATED WORK

Global Subgraph Counting. Global subgraph counting is to count the number of the subgraphs being matched in an entire graph. There are exact/approximate approaches. The exact approach ESCAPE [59] can process 5-node patterns, whereas the approximate approach MOTIVO [21] can handle up to 10-node patterns. Various estimation strategies have been explored for approximate counting such as path sampling [42, 80], color coding [21, 19, 20], and random walk [79, 70, 84]. It is worth noting that global subgraph counting can be solved by local subgraph counting, but not vice versa.

Local Subgraph Counting. Local subgraph counting has been studied for certain patterns with specific orbits on large networks. For local subgraph counting with 1 node orbit, Ahmed et al. [9] propose a multi-core algorithm for exact/approximate counting for 3/4-node patterns, while Elenberg et al. [29, 30] study exact/approximate counting for 3/4-node patterns in a distributed environment. Hocevar et al. [38, 39] propose a matrix-based method for 5-node patterns, while Pashanasangi et al. in [58] propose the decomposition-based method for 5-node patterns. Melckenbeeck et al. [49, 48, 50] explore the general approach to handle k-node patterns by generalizing the idea of [38]. For local subgraph counting with 1 edge orbit, Ahmed et al. [10, 26] explore the methods for selected 3/4/5-node patterns. Also, there are works that focus on local subgraph counting for directed graph [56] and heterogeneous graph [64, 67]. Almost all previous works focus on specific patterns with specific orbits. In contrast, we study a general decomposition-based approach for patterns with different orbits.

Subgraph Enumeration in Native Graph Systems. Subgraph enumeration is to enumerate all matches by subgraph isomorphism (homomorphism) of a pattern, p, in a

Table 4: 9 Datasets.

Datasets		|V|	|E|	Deg	Source
AC	(AC-caida)	2.6×10^4	1.1×10^5	4.0	[1]
TP	(Topology)	3.5×10^4	2.2×10^5	6.2	[1]
RC	(Reactcome)	6.3×10^3	2.9×10^5	46.2	[1]
FB	(Facebook)	6.4×10^4	1.6×10^6	25.6	[1]
WB	(web-BerkStan)	6.9×10^5	1.3×10^7	19.4	[3]
AS	(as-Skitter)	1.7×10^6	2.2×10^7	13.1	[3]
LJ	(soc-LiveJournal)	4.8×10^6	8.6×10^7	17.7	[3]
OK	(com-Orkut)	3.1×10^6	2.3×10^8	76.3	[3]
UK	(uk-2002)	1.9×10^7	5.2×10^8	28.3	[2]

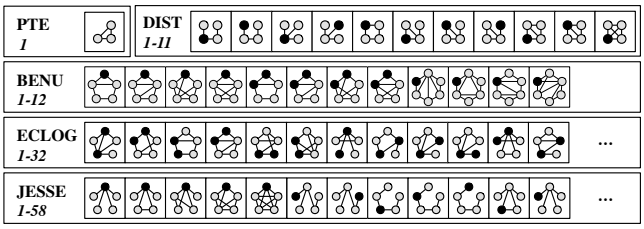


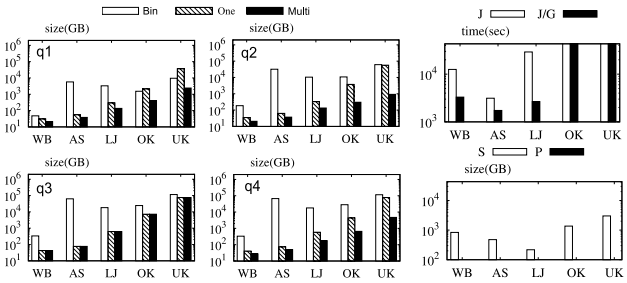
Figure 10: 114 Queries

graph G, and can be used to deal with subgraph (homomorphism) counting. There are single machine approaches [44, 73]. We focus on distributed approaches. The approaches in [74, 45, 46] decompose a pattern graph into specific sub-patterns. Such sub-patterns over a graph are pre-computed and then assembled one-by-one. Assembling produces overwhelming amount of intermediate results. The approaches in [62, 81] decompose a pattern graph into a core surrounded by leaves. The core and all leaves are assembled together that does not produce any intermediate results. These decomposition methods are optimized towards reducing the enumeration cost and intermediate sizes, which may not be compatible with the optimization for aggregation. DISC is optimized towards lowering the overall cost of join and aggregation.

Multi-join & Aggregation. Optimizing join queries has been studied for decades. Traditional, it is to compute a sequence of binary joins [71]. The *AGM* bound [14, 35] on the worst-case output size of a multi-join provides a standard to evaluate the computation efficiency of a join algorithm. In the worst-case, the approach of computing a sequence of binary joins becomes sub-optimal, and the worst-case optimal join algorithms such as NPRR [53], GenericJoin [54], Leapfrog [78] are optimal. In [4, 43, 77], the worst-case optimal join is combined with binary join based on the guidance of tree decomposition [33, 32] to achieve lower complexity. For distributed multi-join processing, it is known that the multi-round of binary joins [41] to process a join query for homomorphism enumeration result in overwhelming communication cost in [22]. For a join query that contains aggregate over join, [83] pushes down the aggregate partially. The similar idea is applied in factorized database [15], and is generalized to functional aggregate query [5]. Previous works focus on solving join query in distributed environment or join-aggregate query on single machine, while DISC is optimized towards evaluating join-aggregate query in distributed environment.

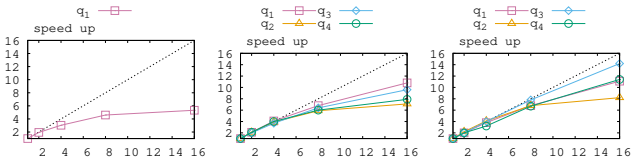
8. EXPERIMENTS

We compare our DISC with the state-of-the-art distributed and multi-core enumeration-based (PTE [57] and BENU [81]), matrix-based (ECLOG [26]) and decomposition-based (DIST [30]) approaches. In addition, we compare DISC with JESSE [48, 50], which is a single machine approach, since it is the



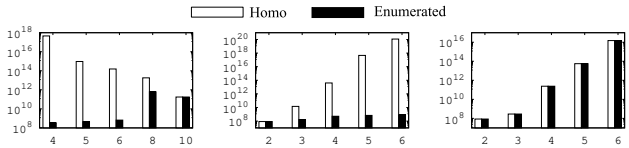
(a) Multi-Round HCube (b) Leapfrog-Agg

Figure 11: Distributed Optimizations



(a) 3-node query (b) 4-node query (c) 5-node query

Figure 12: DISC: Scalability



(a) k-edges (b) k-stars (c) k-cliques

Figure 13: Pattern Scalability

only general approach that can handle any k-node graphs with node orbits. The details can be found in Table 2. We used the code obtained from the authors and used their default parameters. It is worth noting that ECLOG, BENU and DIST were developed in C++, whereas DISC was implemented in Scala on top of the widely-used *Spark*. All experiments were conducted on a cluster consisting of a master server and 7 slave servers (2× Intel Xeon E5-2680 v4, 176 gigabytes of memory, interconnected via 10 gigabytes Ethernet). We used *Spark* 2.4.3, *Hadoop* 2.7.2, and *PowerGraph* 2.2. The experiments that involve only a single machine were conducted on one of the slave servers. We used wall clock time to evaluate the approaches, excluding the time of starting up the system and loading data. When presenting results, we use the notation "$> t$" if an approach failed after t seconds.

9 Datasets: We use 9 datasets used in the prior works from a variety of domains including internet topology networks, protein–protein interactions networks, social networks, and web graphs. We show the statistics of the 9 datasets in Table 4, where Deg is the average degree of the graph, Deg = |E|/|V|. We divide the datasets into two groups by the size. The group-1 consists of AC, TP, RC and FB used in single machine experiments. The group-2 consists of WB, AS, LJ, OK and UK used for distributed processing.

114 Queries: We test all 114 queries used in previous work. The queries are divided into 5 named query sets (PTE, DIST, BENU, ECLOG, JESSE) whose representative queries are listed in Fig. 10. For each query set, we number the queries as q_1, q_2, $\cdots$, from the left. Here, ECLOG and JESSE query set includes all 5-node queries with a local edge orbit (all nodes are connected to orbit edge) and a node orbit respectively. We compare the approaches using the query sets that are used in their own experiments.

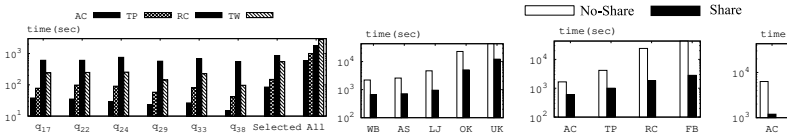
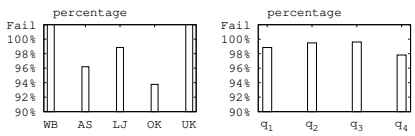


Figure 14: Selected vs. All

(a) DISC queries(b) ECLOG queries(c) JESSE queries

Figure 15: DISC: Sharing



(a) q_1 on 5 datasets (b) 4 queries on LJ

Figure 16: Checking Cost

Table 5: JESSE vs DISC.

	Enumerated Matches ($\times 10^9$)				Elapse Time (seconds)			
Dataset	AC	TP	RC	FB	AC	TP	RC	FB
JESSE	41.7	95.9	11.4	66.5	>43200	>43200	17497	>43200
DISC	1.9	19.1	449.0	28.7	1619	9153	17212	>43200

Table 6: PTE vs DISC.

	Enumerated Matches ($\times 10^7$)					Elapse Time (seconds)				
Dataset	WB	AS	LJ	OK	UK	WB	AS	LJ	OK	UK
PTE	1.3	6.5	2.9	28.6	62.8	86	76	81	92	472
DISC	7.8	38.8	17.3	171.4	376.6	14	15	35	77	161

Table 7: DIST vs DISC

	Enumerated Matches ($\times 10^{10}$)					Elapse Time (seconds)				
Dataset	WB	AS	LJ	OK	UK	WB	AS	LJ	OK	UK
DIST	91.1	25.8	31.9	178.8	1537.8	1714	505	97.6	fail	fail
DISC	14.0	6.9	27.7	28.1	431.4	659	713	953	5043	12179

Table 8: ECLOG vs DISC.

	Enumerated Matches ($\times 10^{10}$)				Elapse Time(seconds)			
Dataset	AC	TP	RC	FB	AC	TP	RC	FB
ECLOG	1559.6	3371.0	36.0	148.0	436	1367	335	597
DISC	0.04	0.8	40.6	1.46	593	1005	1802	2803

8.1 DISC Testing

One-round vs. Multi-round HCube: We compare our multi-round HCubeJoin (Multi) with the one-round HCube (One) and the multi-round binary joins (Bin) using the group-2 datasets. The counting queries tested are BENU q_1-q_4 (Fig. 10). The results are shown in Fig. 11(a). We show the communication cost for all the 4 queries. The reduction in communication cost leads to the reduction in running time. For q_1-q_4 on LJ, on average, it saves 28% running time. For communication cost, except q_3, Multi outperforms the others. For q_3, One and Multi perform the same and are better than Bin. The difference between Bin and One becomes smaller while the size of the graph grows, whereas Multi is stable. We explain it below. Complex queries and larger input data will result in more partitions needed in HCubeJoin, given the memory available. Thus, it results in higher communication cost. Comparing to One, each query of Multi is simpler, which makes it less prone to result in more partitions when the size of the graph grows.

Attribute Ordering. We test our attribute ordering used in the modified Leapfrog, using BENU q_3, where the attribute order satisfies both j-order and g-order. The result is shown in the upper Fig. 11(b), where we denote order satisfies both orders by J/G and the order by j-order by J. J/G can reduce the computing cost up to 10 times. Also, we compare our pipeline approach (P) with multiple-run Leapfrog which stores the group-by and aggregation results (S) without pipeline using q_3. The result is shown in the lower Fig. 11(b).

kwP outputs zero intermediate results, whereas S needs to store 3TB intermediate results.

Selected vs. All: We evaluate DISC in processing selected queries and all queries. We used 6 queries, q_1-q_6, taken from ECLOG (Fig. 10), since they are more important in link prediction tasks [26] than the others in the ECLOG query set. We conducted the testing using the group-1 datasets. The results are shown in Fig. 14, where we denote the 6 queries as a whole by selected and denote all the 32 queries used in ECLOG (Fig. 10) by all. We observe that DISC is able to efficiently evaluate a single query if needed. For example, on AC, q_{38} is 24 times faster than all. Such an ability to process selected counting queries efficiently is highly desirable.

Varying Pattern: We evaluate DISC with different patterns, and show the total homomorphism number (Homo) in the graph and matches we enumerate by DISC (Enumerated) using LJ. In Fig. 13(a), we use the JESSE 5-node queries (q_1-q_5) (Fig. 10), the x-axis is the number of edges in the 5-node pattern graph from a star to a clique. The more edges, the larger Enumerated. For the 5-clique (the rightmost), it needs to enumerate all Homo. In Fig. 13(b), we test using a k-star pattern graph for $1 \leq k \leq 6$ (x-axis) with the center of the star as the orbit node. The Enumerated is very small and is effective for a large k-star. In Fig. 13(c), we test using a k-clique pattern graph for $1 \leq k \leq 6$ (x-axis) with an arbitrary node as the orbit node. The Enumerated is the same as Homo, since a clique can neither be decomposed nor be pushed down group-by and aggregation.

Sharing: We test the effectiveness of cost sharing using the DIST, ECLOG, and JESSE query sets. The results are shown in Fig. 15. The bar No-Share denotes no sharing, whereas the bar Share denotes sharing. Share outperforms No-Share by up to 5, 16 and 13 times on the query set of DIST, ECLOG and JESSE, respectively. DISC failed to complete the query set of ECLOG on FB when sharing is not enabled.

Scalability: We test the scalability of DISC on LJ by varying the number of workers from 1 to 16 on *Spark*. We used 3-, 4-, and 5-node queries, which consist of PTE q_1, DIST q_1-q_4, and BENU q_1-q_4[2], respectively. The results are shown in Fig. 12. On average, by increasing the number of workers from 1 to 16, DISC achieves 5.3×, 8.9× and 11.2× speedup separately. The reason that DISC scaled worse on 3-node queries is that the query is relatively simple, and the computation cost is not the bottleneck.

8.2 Compare DISC with Others

We compared DISC with the state-of-the-art parallel/distributed approaches (i.e., PTE, DIST, ECLOG, and BENU) and a general approach (JESSE). First, in terms of the efficiency, we show that the main cost is to (1) check the condition for the matches enumerated to be subgraph matches and (2) increment the corresponding counter of the mapping of the orbit based on subgraph matches. We test using BENU (an enumeration-based approach) for the first 4 BENU queries (q_1-q_4) with the group-2 datasets. The results are shown in Fig. 16, as the percentage of the time to check conditions and increment counters over the total time. In Fig. 16(a), the results are for q_1 over all group-2 datasets. In Fig. 16(b), the results are for all 4 queries over

[2]We perform subgraph counting instead of induced subgraph counting for BENU q_1-q_4.

Table 9: BENU vs DISC

	Enumerated Matches ($\times 10^{10}$)								Elapse Time (seconds)							
	q_1		q_2		q_3		q_4		q_1		q_2		q_3		q_4	
Dataset	DISC	BENU	DISC	BENU	DISC	BENU	DISC	BENU	DISC	BENU	DISC	BENU	DISC	BENU	DISC	BENU
WB	5.9	2679.3	0.7	3612.8	42.2	750.0	3.0	551.4	1068	>12685	85	>15876	3298	10452	199	5826
AS	3.3	500.8	0.2	502.1	8.3	61.8	0.4	55.4	963	6971	114	4086	1735	1138	87	812
LJ	2.8	5355.3	5.2	895.2	31.5	3616.6	28.0	1884.2	632	20552	425	31346	2663	41803	1093	7295
OK	12.0	4969.2	3.9	3339.9	28.9	354.7	9.4	260.4	3895	22899	1278	11534	>43200	4425	1713	1195
UK	40.3	185463.4	8.16	305747.2	1106.8	117980.5	442.8	62088.8	19766	>43200	6295	>43200	>43200	>43200	11898	>43200

Table 10: BENU vs DISC

	Enumerated Matches ($\times 10^{10}$)				Elapse Time (seconds)			
	WB		AS		WB		AS	
Query	DISC	BENU	DISC	BENU	DISC	BENU	DISC	BENU
q_5	5.8	2679.3	3.3	500.8	1109	>43200	1014	9158
q_6	2.8	3612.8	0.5	502.1	117	>43200	132	6847
q_7	42.2	750.0	8.3	61.8	4904	>14468	1958	3033
q_8	5.2	551.4	0.7	55.4	173	>8586	99	1869
q_9	87.5	10765562.1	17.0	681815.5	4238	>43200	2011	>43200
q_{10}	48.1	4960415.6	9.2	408976.9	327	>43200	291	>43200
q_{11}	12.2	323920.0	1.9	38493.1	563	>43200	373	>27949
q_{12}	16.5	45396.7	2.3	2349.4	513	>43200	301	>7239

LJ. The major cost is to check conditions and increment counters, and the number of matches enumerated matters. In the following, in comparison with others, we show 2 sets of numbers in tables. One set named `Enumerated Matches` shows the numbers of matches to be enumerated, and one set named `Elapse Time` shows the wall clock time to execute the queries (seconds). It is worth noting that we record the `Enumerated Matches` and `Elapse Time` of executing all queries of a query set together when DISC is compared to DIST, ECLOG, JESSE. Also, note that the programming languages used are different. The code by C++ can be faster than the code by Scala.

Comparison with JESSE: We compare DISC with JESSE using the JESSE 58 5-node queries (Fig. 10) over the group-1 datasets. Since JESSE is a serial single-machine algorithm, we ran both DISC and JESSE on a single core of a machine. The results are shown in Table 5, from which we can observe that DISC significantly outperforms JESSE.

Comparison with PTE: We compare DISC with PTE on the triangle pattern graph, which is the only pattern supported by PTE, over the group-2 datasets by executing global subgraph counting. The results are shown in Table. 6. Observe that the number of matches enumerated by DISC was about 6 times of that by PTE, as expected: (i) DISC enumerates homomorphisms, while PTE enumerates matched subgraphs, and (ii) a triangle has 6 homomorphisms.

Comparison with DIST: We compare DISC with DIST on the DIST 11 4-node queries (Fig. 10) over the group-2 datasets. DISC enumerates a fewer number of matches than DIST, as shown in Table. 7. Note that DIST is implemented by C++ to deal with different cases, and DISC is a general approach developed on *Spark*. DIST outperforms DISC in AS and LJ, where the differences between DISC and DIST in terms of the number of matches enumerated are relatively small. DIST failed for these graphs large graphs such as OK and UK, as DIST needs to construct a two-hop index, to accelerate computation, which becomes overwhelmingly large for OK (557GB) and UK (1,180GB).

Comparison with ECLOG: We compare DISC with ECLOG on the DIST 32 5-node queries with 1 edge orbit (Fig. 10) over the group-1 datasets, since the number of queries is large. The results are shown in Table 8. DISC enumerates up to 10^3 times fewer matches than ECLOG. However, DISC enumerates more matches than ECLOG on RC. The reason is that for DISC, most (88.5%) of the matches enumerated on RC come from 5-clique which cannot be further decomposed. In terms of elapse time, since ECLOG is coded by C++ and DISC is by Scala, ECLOG outperforms DISC in AC, RC and FB.

Comparison with BENU: We compare DISC with BENU on the BENU 4 5-node queries with 1 node orbit, 4 5-node queries with 1 edge orbit, and 4 6-node queries with 1 node orbit (Fig. 10) over the group-2 datasets. In this experiment, we perform subgraph counting instead of induced subgraph counting for each test, as BENU enumerates over the iso-matches instead of induced iso-matches of the queries. The results for the 4 5-node queries with 1 node orbit (q_1, q_2, q_3, and q_4) are shown in Table 9 and the results for the remaining 8 queries are shown in Table 10. As shown in Table 9, (1) the number of matches enumerated by DISC is up to 10^4 times less than that of BENU, and (2) in terms of efficiency, DISC outperforms BENU in all cases except q_3 over both AS and OK, and q_4 over OK, due to the fact the number of matches enumerated are comparatively similar. It is worth noting that q_1, q_2, q_3, and q_4 are highly connected to which BENU is well designed. For the remaining 8 queries q_5-q_{12}, as the number of nodes increases, the difference between the numbers of matches enumerated by BENU and DISC increases, up to 10^5. DISC significantly outperforms BENU in all cases, and BENU could not complete most of the queries due to timeout or out-of-memory.

9. CONCLUSION

We propose an approach for local subgraph counting by local homomorphism counting, and process local homomorphism counting by natural joins with group-by and aggregation in a distributed system DISC on top of *Spark*. Our approach is general to handle any k-node pattern graph with any orbit o selected, since we only need to deal with the orbit using group-by. The key to achieving high efficiency by homomorphism is that there is no need to check subgraph isomorphism for any pattern graph p. Our approach can be used to efficiently process a counting query or a collection of counting queries by sharing their cost. We confirm the efficiency of homomorphism counting (e.g., eliminating counts for those that are not subgraph isomorphism matchings from the total count for any possible matchings) using 114 local subgraph counting queries over real large graphs. Extensive experiments demonstrate that our DISC consistently outperforms the state-of-the-art approaches, i.e., BENU, PTE, JESSE, significantly, and outperforms DIST on large graphs.

Acknowledgement

This work is supported by the Research Grants Council of Hong Kong, China under No. 14203618, No. 14202919, and No. 14205617.

10. REFERENCES

[1] http://konect.uni-koblenz.de.

[2] http://law.di.unimi.it/datasets.php.

[3] https://snap.stanford.edu/data/index.html.

[4] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. EmptyHeaded: A Relational Engine for Graph Processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.

[5] M. Abo Khamis, H. Q. Ngo, and A. Rudra. Faq: questions asked frequently. In *Proc. of PODS'16*, pages 13–28, 2016.

[6] F. N. Afrati and J. D. Ullman. Optimizing Multiway Joins in a Map-Reduce Environment. *TKDE*, 23(9):1282–1298, 2011.

[7] N. K. Ahmed, R. Rossi, J. B. Lee, T. L. Willke, R. Zhou, X. Kong, and H. Eldardiry. Learning role-based graph embeddings. *arXiv preprint arXiv:1802.02896*, 2018.

[8] N. K. Ahmed, R. A. Rossi, R. Zhou, J. B. Lee, X. Kong, T. L. Willke, and H. Eldardiry. A framework for generalizing graph-based representation learning methods. *arXiv preprint arXiv:1709.04596*, 2017.

[9] N. K. Ahmed, T. L. Willke, and R. A. Rossi. Estimation of local subgraph counts. In *Prof. of IEEE BigData'16*, pages 586–595, 2016.

[10] N. K. Ahmed, T. L. Willke, and R. A. Rossi. Exact and estimation of local edge-centric graphlet counts. In *Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pages 1–17, 2016.

[11] L. Akoglu, H. Tong, and D. Koutra. Graph based anomaly detection and description: a survey. *Data Min. Knowl. Discov.*, 29(3):626–688, 2015.

[12] O. Amini, F. V. Fomin, and S. Saurabh. Counting subgraphs via homomorphisms. In *Proc. of ICALP'09*, pages 71–82, 2009.

[13] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed Evaluation of Subgraph Queries Using Worst-case Optimal Low-memory Dataflows. *PVLDB*, 11(6):691–704, 2018.

[14] A. Atserias, M. Grohe, and D. Marx. Size Bounds and Query Plans for Relational Joins. In *Proc. of FOCS'08*, pages 739–748, 2008.

[15] N. Bakibayev, T. Kociskỳ, D. Olteanu, and J. Závodnỳ. Aggregation and ordering in factorised databases. *PVLDB*, 6(14), 2013.

[16] P. Beame, P. Koutris, and D. Suciu. Communication steps for parallel query processing. In R. Hull and W. Fan, editors, *Proc. of PODS'13*, pages 273–284, 2013.

[17] A. R. Benson, D. F. Gleich, and J. Leskovec. Higher-order organization of complex networks. *Science*, 353(6295):163–166, 2016.

[18] C. Borgs, J. Chayes, L. Lovász, V. T. Sós, and K. Vesztergombi. Counting graph homomorphisms. In *Topics in Discrete Mathematics*, pages 315–371. 2006.

[19] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi. Counting Graphlets: Space vs Time. In *Proc. of WSDM'17*, pages 557–566, 2017.

[20] M. Bressan, F. Chierichetti, R. Kumar, S. Leucci, and A. Panconesi. Motif Counting Beyond Five Nodes. *TKDD*, 12(4):48, 2018.

[21] M. Bressan, S. Leucci, and A. Panconesi. Motivo: fast motif counting via succinct color coding and adaptive sampling. *arXiv preprint arXiv:1906.01599*, 2019.

[22] S. Chu, M. Balazinska, and D. Suciu. From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. In *Proc. of SIGMOD'15*, pages 63–78, 2015.

[23] J. S. Coleman. Social Capital in the Creation of Human Capital. *American Journal of Sociology*, 94:S95–S120, 1988.

[24] R. Curticapean, H. Dell, and D. Marx. Homomorphisms are a good basis for counting small subgraphs. In *Proc. of STOC'17*, pages 210–223, 2017.

[25] V. Dalmau and P. Jonsson. The complexity of counting homomorphisms seen from the other side. *Theoretical Computer Science*, 329(1-3):315–323, 2004.

[26] V. S. Dave, N. K. Ahmed, and M. A. Hasan. E-CLoG: Counting edge-centric local graphlets. In *Proc. of 2017 IEEE BigData*, pages 586–595, 2017.

[27] J. Díaz, M. Serna, and D. M. Thilikos. Counting h-colorings of partial k-trees. *Theoretical Computer Science*, 281(1-2):291–309, 2002.

[28] M. Dyer and C. Greenhill. The complexity of counting graph homomorphisms. In *Proc. of SODA'00*, pages 246–255, 2000.

[29] E. R. Elenberg, K. Shanmugam, M. Borokhovich, and A. G. Dimakis. Beyond Triangles: A Distributed Framework for Estimating 3-profiles of Large Graphs. In *Proc. of SIGKDD'15*, pages 229–238, 2015.

[30] E. R. Elenberg, K. Shanmugam, M. Borokhovich, and A. G. Dimakis. Distributed Estimation of Graph 4-Profiles. In *Proc. of WWW'16*, pages 483–493, 2016.

[31] G. Fagiolo. Clustering in complex directed networks. *Physical Review E*, 76(2):026107, 2007.

[32] G. Gottlob, G. Greco, N. Leone, and F. Scarcello. Hypertree Decompositions: Questions and Answers. In *Proc. of PODS'16*, pages 57–74, 2016.

[33] G. Gottlob, N. Leone, and F. Scarcello. Hypertree Decompositions and Tractable Queries. *Journal of Computer and System Sciences*, 64(3):579–627, 2002.

[34] M. Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *Journal of the ACM*, 54(1):1–24, 2007.

[35] M. Grohe and D. Marx. Constraint Solving via Fractional Edge Covers. *ACM Trans. Algorithms*, 11(1):4:1–4:20, 2014.

[36] W. Hayes, K. Sun, and N. Przulj. Graphlet-based measures are suitable for biological network comparison. *Bioinformatics*, 29(4):483–491, 2013.

[37] P. Hell and J. Nešetřil. On the complexity of h-coloring. *Journal of Combinatorial Theory, Series B*, 48(1):92–110, 1990.

[38] T. Hočevar and J. Demšar. A combinatorial approach to graphlet counting. *Bioinformatics*, 30(4):559–565, 2014.

[39] T. Hočevar and J. Demšar. Combinatorial algorithm for counting small induced graphs and orbits. *PloS one*, 12(2):e0171428, 2017.

[40] P. W. Holland and S. Leinhardt. A Method for

Detecting Structure in Sociometric Data. In *Social Networks*, pages 411–432. 1977.

[41] J. Huang, K. Venkatraman, and D. J. Abadi. Query optimization of distributed pattern matching. In *Proc. of ICDE'14*, pages 64–75, 2014.

[42] M. Jha, C. Seshadhri, and A. Pinar. Path sampling: A fast and provable method for estimating 4-vertex subgraph counts. In *Proc. of WWW'15*, pages 495–505, 2015.

[43] O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible caching in trie joins. *arXiv preprint arXiv:1602.08721*, 2016.

[44] H. Kim, J. Lee, S. S. Bhowmick, W.-S. Han, J. Lee, S. Ko, and M. H. Jarrah. Dualsim: Parallel subgraph enumeration in a massive graph on a single machine. In *Proc. of SIGMOD'16*, pages 1231–1245, 2016.

[45] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable Subgraph Enumeration in MapReduce. *PVLDB*, 8(10):974–985, 2015.

[46] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang. Scalable Distributed Subgraph Enumeration. *PVLDB*, 10(3):217–228, 2016.

[47] N. N. Liu, L. He, and M. Zhao. Social temporal collaborative ranking for context aware movie recommendation. *ACM TIST*, 4(1):15:1–15:26, 2013.

[48] I. Melckenbeeck, P. Audenaert, D. Colle, and M. Pickavet. Efficiently counting all orbits of graphlets of any order in a graph using autogenerated equations. *Bioinformatics*, 34(8):1372–1380, 2017.

[49] I. Melckenbeeck, P. Audenaert, T. Michoel, D. Colle, and M. Pickavet. An algorithm to automatically generate the combinatorial orbit counting equations. *PloS one*, 11(1), 2016.

[50] I. Melckenbeeck, P. Audenaert, T. Van Parys, Y. Van De Peer, D. Colle, and M. Pickavet. Optimising orbit counting of arbitrary order by equation selection. *BMC bioinformatics*, 20(1):27, 2019.

[51] A. Mhedhbi and S. Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *PVLDB*, 12(11):1692–1704, 2019.

[52] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network Motifs: Simple Building Blocks of Complex Networks. *Science*, 298(5594):824–827, 2002.

[53] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case Optimal Join Algorithms: [Extended Abstract]. In *Proc. of PODS'12*, pages 37–48, 2012.

[54] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *ACM SIGMOD Record*, 42(4):5–16, 2014.

[55] C. C. Noble and D. J. Cook. Graph-based anomaly detection. In *Proc. of SIGMOD'03*, pages 631–636, 2003.

[56] M. Ortmann and U. Brandes. Efficient orbit-aware triad and quad census in directed and undirected graphs. *Applied network science*, 2(1):13, 2017.

[57] H.-M. Park, S.-H. Myaeng, and U. Kang. Pte: Enumerating trillion triangles on distributed systems. In *Proc. of SIGKDD'16*, pages 1115–1124, 2016.

[58] N. Pashanasangi and C. Seshadhri. Efficiently counting vertex orbits of all 5-vertex subgraphs, by evoke. In *Proc. of WSDM'20*, pages 447–455, 2020.

[59] A. Pinar, C. Seshadhri, and V. Vishal. ESCAPE: Efficiently Counting All 5-Vertex Subgraphs. In *Proc. of WWW'17*, pages 1431–1440, 2017.

[60] N. Pržulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2):e177–e183, 2007.

[61] N. Pržulj, D. G. Corneil, and I. Jurisica. Modeling interactome: scale-free or geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

[62] M. Qiao, H. Zhang, and H. Cheng. Subgraph Matching: On Compression and Computation. *PVLDB*, 11(2):176–188, 2017.

[63] P. Ribeiro, P. Paredes, M. E. P. Silva, D. Aparicio, and F. Silva. A survey on subgraph counting: Concepts, algorithms and applications to network motifs and graphlets. *CoRR*, abs/1910.13011, 2019.

[64] R. A. Rossi, N. K. Ahmed, A. Carranza, D. Arbour, A. Rao, S. Kim, and E. Koh. Heterogeneous network motifs. *arXiv preprint arXiv:1901.10026*, 2019.

[65] R. A. Rossi, N. K. Ahmed, and E. Koh. Higher-order network representation learning. In *Proc. of WWW'18 (Companion)*, pages 3–4, 2018.

[66] R. A. Rossi, L. K. McDowell, D. W. Aha, and J. Neville. Transforming graph data for statistical relational learning. *J. Artif. Intell. Res.*, 45:363–441, 2012.

[67] R. A. Rossi, A. Rao, T. Mai, and N. K. Ahmed. Fast and accurate estimation of typed graphlets.

[68] R. A. Rossi, R. Zhou, and N. K. Ahmed. Deep feature learning for graphs. *arXiv preprint arXiv:1704.08829*, 2017.

[69] R. Rotabi, K. Kamath, J. Kleinberg, and A. Sharma. Detecting strong ties using network motifs. In *Proc. of WWW'17*, pages 983–992, 2017.

[70] T. K. Saha and M. Al Hasan. Finding network motifs using MCMC sampling. In *Complex Networks VI*, pages 13–24. 2015.

[71] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proc. of SIGMOD'79*, pages 23–34, 1979.

[72] S. Son, A. R. Kang, H.-c. Kim, T. Kwon, J. Park, and H. K. Kim. Analysis of Context Dependence in Social Interaction Networks of a Massively Multiplayer Online Role-Playing Game. *PLOS ONE*, 7(4):e33918, 2012.

[73] S. Sun, Y. Che, L. Wang, and Q. Luo. Efficient parallel subgraph enumeration on a single machine. In *Proc. of ICDE'19*, pages 232–243, 2019.

[74] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient Subgraph Matching on Billion Node Graphs. *PVLDB*, 5(9):788–799, 2012.

[75] C. Tsourakakis. The k-clique densest subgraph problem. In *Proc. of WWW'15*, pages 1122–1132, 2015.

[76] C. E. Tsourakakis, J. Pachocki, and M. Mitzenmacher. Scalable motif-aware graph clustering. In *Proc. of WWW'17*, pages 1451–1460, 2017.

[77] S. Tu and C. Ré. Duncecap: Query plans using

generalized hypertree decompositions. In *Proc. of SIGMOD'15*, pages 2077–2078, 2015.

[78] T. L. Veldhuizen. Triejoin: A Simple, Worst-Case Optimal Join Algorithm, 2014.

[79] P. Wang, J. Lui, B. Ribeiro, D. Towsley, J. Zhao, and X. Guan. Efficiently estimating motif statistics of large networks. *TKDD*, 9(2):8, 2014.

[80] P. Wang, J. Zhao, X. Zhang, Z. Li, J. Cheng, J. C. Lui, D. Towsley, J. Tao, and X. Guan. MOSS-5: A fast method of approximating counts of 5-node graphlets in large graphs. *TKDE*, 30(1):73–86, 2017.

[81] Z. Wang, R. Gu, W. Hu, C. Yuan, and Y. Huang. BENU: Distributed Subgraph Enumeration with Backtracking-Based Framework. In *Proc. of ICDE'19*, pages 136–147, 2019.

[82] B. F. Welles, A. V. Devender, and N. Contractor. Is a "friend" a friend? Investigating the structure of friendship networks in virtual worlds. In *Proc. of The 28th Annual CHI Conference on Human Factors in Computing Systems*, pages 4027–4032, 2010.

[83] W. P. Yan and P.-B. Larson. Eager aggregation and lazy aggregation. *Group*, 1:G2, 1995.

[84] C. Yang, M. Lyu, Y. Li, Q. Zhao, and Y. Xu. SSRW: A Scalable Algorithm for Estimating Graphlet Statistics Based on Random Walk. In *Proc. of DASFAA'18*, pages 272–288, 2018.

[85] H. Yin, A. R. Benson, and J. Leskovec. Higher-order clustering in networks. *Physical Review E*, 97(5):052306, 2018.

[86] H. Yin, A. R. Benson, and J. Leskovec. The local closure coefficient: a new perspective on network clustering. In *Proc. of WSDM'19*, pages 303–311, 2019.