Scalable Distributed Subgraph Enumeration

Longbin Lai[1] Lu Qin[2] Xuemin Lin[1] Ying Zhang[2]
Lijun Chang[1]

[1] University of New South Wales, Australia
{llai,lxue,ljchang}@cse.unsw.edu.au
[2] Centre for Quantum Computation & Intelligent Systems,
University of Technology, Sydney, Australia
{lu.Qin,ying.zhang}@uts.edu.au

Technical Report
UNSW-CSE-TR-201606
March 2016

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Subgraph enumeration aims to find all the subgraphs of a large data graph that are isomorphic to a given pattern graph. As the subgraph isomorphism operation is computationally intensive, researchers have recently focused on solving this problem in distributed environments, such as MapReduce and Pregel. Among them, the state-of-the-art algorithm, TwinTwigJoin, is proven to be instance optimal based on a left-deep join framework. However, it is still not scalable to large graphs because of the constraints in the left-deep join framework and that each decomposed component (join unit) must be a star. In this paper, we propose SEED - a scalable subgraph enumeration approach in the distributed environment. Compared to TwinTwigJoin, SEED returns optimal solution in a generalized join framework without the constraints in TwinTwigJoin. We use both star and clique as the join units, and design an effective distributed graph storage mechanism to support such an extension. We develop a comprehensive cost model, that evaluates the number of matches of any given pattern graph by considering power-law degree distribution in the data graph. We then generalize the left-deep join framework and develop a dynamic-programming algorithm to compute an optimal bushy join plan. We also consider overlaps among the join units. Finally, we propose clique compression to further improve the algorithm by reducing the number of the intermediate results. Extensive performance studies are conducted on several real graphs, one containing billions of edges. The results demonstrate that our algorithm is more than one order of magnitude faster than all other state-of-the-art algorithms in all datasets.

1 Introduction

In this paper, we study subgraph enumeration, a fundamental problem in graph analysis. Given an undirected, unlabeled data graph G and a pattern graph P, subgraph enumeration aims to find all subgraph instances of G that are isomorphic to P. Subgraph enumeration is widely used in many applications. It is used in network motif computing [26, 2] to facilitate the design of large networks from biochemistry, neurobiology, ecology, and bioinformatics. It is used to compute the graphlet kernels for large graph comparison [31, 28], property generalization for biological networks [25], and is considered to be a key operation for the synthesis of target structures in chemistry [29]. It can also be adopted to illustrate the evolution of social networks [19] and to discover information trends in recommendation networks [22].

1.1 Motivation

Enumerating subgraphs in a large data graph, despite its varied applications, is extremely challenging for two reasons: First, its core operation, known as subgraph isomorphism, is computationally hard. Second, the lack of label information often causes a large number of intermediate results, that can be much larger than the size of the data graph itself. As a result, existing centralized algorithms [3, 11] are not scalable to large graphs, and researchers have recently explored efficient subgraph enumeration algorithms in distributed environments, such as MapReduce [7] and Pregel [24]. Typically, there are two ways of solving subgraph enumeration - the depth-first search and the join operation. While the former is hard to parallelize, people tend to use the join algorithm to solve subgraph enumeration in the distributed context.

In MapReduce, Multiway join [1] enumerates subgraph instances in single MapReduce round by duplicating each edge on multiple machines, while it can surrender to serious scalability problem as each machine may have to store the whole graph for complex queries. the authors in [20] studied the StarJoin algorithm, which first decomposes the pattern graph into a set of disjoint stars. Here, a star is a tree of depth one. Then StarJoin solves subgraph enumeration by joining the matches of the decomposed stars following a left-deep join framework. However, it is sometimes inefficient to process a star due to the generation of numerous intermediate results. For example, a *celebrity node* with $1{,}000{,}000$ neighbors in the social network would render $O(10^{18})$ matches of a star of three edges, making it impossible to compute and maintain for future join. Aware of the deficiency of StarJoin, the authors proposed the TwinTwigJoin algorithm [20], which inherits the left-deep join framework from StarJoin, but processes TwinTwig- a star of either one or two edges - instead of a general star. The authors further proved the instance optimality of TwinTwigJoin, that is, given a join that involves general stars (a StarJoin), we always have an alternative TwinTwigJoin that draws no more cost than the StarJoin.

In Pregel, Shao et al. [30] proposed PSgL that enumerates subgraphs via graph traversal opposed to join operations. The algorithm applies a breadth-first-search strategy - that is, each time it picks up an already-matched but not fully-expanded node v, and searches the matches of its neighbors in order to generate finer-grained results. Essentially, PSgL is considered to be a StarJoin algorithm [20] that processes the joins between the matches of the star rooted on v and the partial subgraph instances obtained from the previous step. As a result, PSgL does not outperform TwinTwigJoin as shown in [20].

As the state-of-the-art, TwinTwigJoin only guarantees optimality under two con-

straints: (1) each decomposed component (also called *join unit* in this paper) is a star, and (2) the join structure is left-deep. These constraints hamper its practicality in several respects. First, TwinTwigJoin only mitigates but not resolves the issues of StarJoin by using TwinTwig instead of star. For example, the celebrity node of degree 1,000,000 still produces $O(10^{12})$ matches of a two-edge TwinTwig. Second, it takes TwinTwigJoin at least $\frac{m}{2}$ (m is the number of pattern edges) rounds to solve subgraph enumeration, making it inefficient to handle complex pattern graph. Finally, the algorithm follows a left-deep join framework, which may result in a sub-optimal solution [18]. Last but not least, TwinTwigJoin bases the cost analysis on the Erdös-Rényi random (ER) graph model [8], which can be biased considering that most real-life graphs are power-law graphs.

1.2 Contributions

In this paper, we propose SEED, a **S**ubgraph **E**num**E**ration approach in **D**istributed environment, that handles the subgraph enumeration in a general join framework without the above constraints. SEED can be implemented in a general-purpose distributed dataflow engine, such as MapReduce [7], Spark [38], Dryad [17], and Myria [12]. For the ease of presentation, we describe the proposed algorithm in MapReduce in this paper. We make the following contributions in this paper.

First, we generalize the graph storage in TwinTwigJoin by introducing the star-clique-preserved (SCP) storage mechanism to support both clique (a complete graph) and star as the join units (Section 4). With clique as an alternative, we can make a better choice other than star, where possible, and reduce the number of execution rounds. Ultimately, this leads to a huge reduction of the intermediate results. Although there exist other join units besides star and clique, we show that it can hamper the scalability of the algorithm to support these alternatives (details are in Section 4).

Second, we propose a comprehensive cost model to measure the cost of SEED in the distributed context (Section 5). We base the cost analysis on the power-law random (PR) graph model [4] instead of the Erdös-Rényi random (ER) graph model [20]. Considering that many real graphs are power-law graphs, the PR model offers more realistic estimation than the ER model.

Third, we develop a dynamic-programming algorithm to compute an **optimal** bushy join plan (Section 6). In TwinTwigJoin, the authors compute the left-deep join plan with space and time complexities of $O(2^m)$ and $O(d_{max} \cdot m \cdot 2^m)$ respectively, where m is number of edges and d_{max} is the maximum degree in the pattern graph. With the same space complexity and a slightly larger time complexity $O(3^m)$, we arrive at optimality by solving the more challenging bushy join plan. We also show that it is beneficial to overlap edges among the join units. Given some practical relaxation, we compute an optimal join plan that overlaps the join units with the same complexities as the non-overlapped case.

Fourth, we devise the clique-compression technique (Section 7), which prevents us from computing and materializing partial join results in large cliques, and thus improves the performance of SEED.

Finally, we conduct extensive performance studies in six real graphs with different graph properties - the largest containing billions of edges. Experimental results demonstrate that our proposed algorithm achieves high scalability and efficiency and is more than one order of magnitude faster than the state-of-the-art algorithms in all datasets.

1.3 Outline

Section 2 presents the preliminaries and gives the formal problem definition. Section 3 shows the algorithm framework. Section 4 introduces the SCP storage mechanism to allow using clique as the join unit. Section 5 studies the cost of the algorithm based on the PR model. Section 6 explores the bushy-join-based execution plan and the overlapping of the join units. Section 7 discusses how to use the large clique inside the data graph to optimize the algorithm. Section 8 evaluates all introduced algorithms using extensive experiments. Section 9 reviews the related work, and Section 10 concludes the paper.

2 Preliminaries

Given a graph g, we use $V(g)$ and $E(g)$ to denote the set of nodes and edges of g. For a node $\mu \in V(g)$, denote $\mathcal{N}(\mu)$ as the set of neighbors, and $d(\mu) = |\mathcal{N}(\mu)|$ as the degree of μ. A *subgraph* g' of g, denoted $g' \subseteq g$, is a graph that satisfies $V(g') \subseteq V(g)$ and $E(g') \subseteq E(g)$.

A *data graph* G is an undirected and unlabeled graph. Let $|V(G)| = N$, $|E(G)| = M$ (assume $M > N$), and $V(G) = \{u_1, u_2, \ldots, u_N\}$ be the set of data nodes.

A *pattern graph* P is an undirected, unlabeled and connected graph. We let $|V(P)| = n$, $|E(P)| = m$, and $V(P) = \{v_1, v_2, \ldots, v_n\}$ be the set of pattern nodes. We use $P = P' \cup P''$ to denote the merge of two pattern graphs, where $V(P) = V(P') \cup V(P'')$ and $E(P) = E(P_1) \cup E(P_2)$.

Definition 1. ***(Match)*** *Given a pattern graph P and a data graph G, a match f of P in G is a mapping from $V(P)$ to $V(G)$, such that the following two conditions hold:*

- *(Conflict Freedom) For any pair of nodes $v_i \in V(P)$ and $v_j \in V(P)$ ($i \neq j$), $f(v_i) \neq f(v_j)$.*
- *(Structure Preservation) For any edge $(v_i, v_j) \in E(P)$, $(f(v_i), f(v_j)) \in E(G)$.*

We use $f = (u_{k_1}, u_{k_2}, \ldots, u_{k_n})$, to denote the match f, i.e., $f(v_i) = u_{k_i}$ for any $1 \leq i \leq n$.

We say two graph g_i and g_j are isomorphic if and only if there exists a match of g_i in g_j, and $|V(g_i)| = |V(g_j)|$, $|E(g_i)| = |E(g_j)|$. The task of *Subgraph enumeration* is to enumerate all $g \in G$ such that g is isomorphic to P.

Remark 1. *An automorphism of P is an isomorphism from P to itself. Suppose there are A automorphisms of the pattern graph. If the number of enumerated subgraphs is s, then the number of matches of P in G is $A \times s$. Therefore, if P has only one automorphism, the problem of subgraph enumeration is equivalent to enumerating all matches (Definiton 1). Otherwise, there will be duplicate enumeration. In this paper, for the ease of analysis, we will assume that the pattern graph P has only one automorphism, and focus on enumerating all matches of P in G. When P has more than one automorphism, we apply the same technique as [20] to avoid duplicates. We first define the following total order for the data nodes as:*

Definition 2. ***(Node Order)*** *For any two nodes u_i and u_j in $V(G)$, $u_i \prec u_j$ if and only if one of the two conditions holds:*

- *$d(u_i) < d(u_j)$,*
- *$d(u_i) = d(u_j)$ and $i < j$,*

Then we assign a partial order (denoted as $<$) among some pairs of nodes in the pattern graph P using the symmetry-breaking technique (see the appendix). Ultimately, we enforce an Order-Preservation constraint in the match (Definiton 1), that is,

(Order Preservation) For any pair of nodes $v_i, v_j \in V(P)$, if $v_i < v_j$, then $f(v_i) \prec f(v_j)$.

We use $R_G(P)$ to denote the matches of P in G, or simply $R(P)$ when the context is clear. Since a match is a one-to-one mapping from the pattern nodes to the data nodes, we regard $R(P)$ as a relation table with $V(P)$ as its attributes.

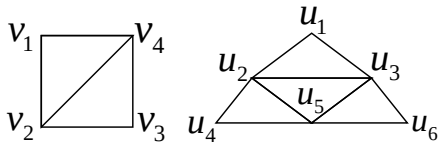


Figure 2.1: Pattern Graph P (Left) and Data Graph G (Right).

Example 1. *Figure 2.1 shows a pattern graph P, and a data graph G. Figure 2.1 shows a pattern graph P, and a data graph G. There are three matches of P in G, which are (u_1, u_2, u_5, u_3), (u_4, u_2, u_3, u_5), and (u_6, u_3, u_2, u_5). The partial orders on the pattern graph for symmetry breaking are $v_1 < v_3$ and $v_2 < v_4$. We can check that, for example, (u_1, u_2, u_5, u_3) satisfies the Order-Preservation constraint as $u_1 \prec u_5$ and $u_2 \prec u_3$ according to Definiton 2.*

Problem Statement. Given a data graph G stored in the distributed file system, and a pattern graph P, the purpose of this work is to enumerate all matches of P in G (based on Definiton 1) in the distributed environment.

Remark 2. *For simplicity, we discuss the algorithm in MapReduce. However, all techniques proposed in this paper are platform-independent, so it is seamless to implement the algorithm in any general-purpose distributed dataflow engine, such as Spark [38], Dryad [17] and Myria [12].*

Power-Law Random (PR) Graph Model. We model the data graph (N nodes and M edges) as a power-law random (PR) graph, denoted as $\mathcal{G}$, according to [4]. Corresponding to the set of data nodes, we consider a non-decreasing degree sequence $\{w_1, w_2, \ldots, w_N\}$ that satisfies power-law distribution, that is, the number of nodes with a certain degree x is proportional to $x^{-\beta}$, where β is the power-law exponent [1]. For any pair of nodes u_i and u_j in a PR graph, the edge between u_i and u_j is independently assigned with probability

$$\mathsf{Pr}_{i,j} = w_i w_j \rho,$$

where $\rho = 1/\Sigma_{i=1}^{N} w_i$. It is easy to verify that the $\mathbb{E}[d(u_i)] = w_i$ for any $1 \leq i \leq N$ ($\mathbb{E}[\cdot]$ computes the expected value). We define the average degree as $w = (\Sigma_{i=1}^{N} w_i)/N$, and the expected maximum degree as w_{max}. In case that $\mathsf{Pr}_{i,j} < 1$ holds, we require $w_{max} \leq \sqrt{wN}$ [36]. As shown in [20], in real-life graphs, although there are nodes with degree larger than $\sqrt{wN}$, the intermediate results from such nodes are not the dominant parts in subgraph enumeration. In this work, if not otherwise specified, we simply let $w_{max} = \sqrt{wN}$. Given β, w, N and w_{max}, a degree sequence can be generated using the method in [36].

In this paper, we compute the number of matches based on the PR model in order to evaluate the graph storage mechanism (Section 4) and the cost of the algorithm

[1] If not specially mentioned, β is set to $2 < \beta < 3$ in this paper, a typical setting of β for real-life graphs [5, 6].

(Section 5). In the computation, we relax the conflict-free condition of a match (Definiton 1) to allow duplicate nodes and self-loops for ease of analysis, hence the number of matches calculated is an *upper bound* of the actual value.

Summary of Notations. Table 2.1 summarizes the notations frequently used in this paper.

Notations	Description
$V(g), E(g)$	The set of nodes and edges of a graph g
$\mathcal{N}(\mu), d(\mu)$	The set of neighbor nodes and the degree of $\mu \in V(g)$
G	The data graph
N, M	The number of nodes and edges in the data graph
u, u_i	A data node, the i[-th] data node regarding the node id
P	The pattern graph
n, m	The number of nodes and edges in the pattern graph
v, v_i	A pattern node, the i[-th] patter node
P_i	The i[-th] partial pattern, $P_i \subseteq P$
P_i^l, P_i^r	The left and right patterns while joining to produce P_i
f	A match of P in G
$R_G(P), R(P)$	The relation of the matches of P in G
$\Phi(G)$	The storage mechanism of G
G_u	The local graph of $u \in V(G)$, where $G_u \in \Phi(G)$
$\mathcal{G}$	A power-law random (PR) graph
β	The power-law exponent of $\mathcal{G}$
w_i	The expected degree of u_i in $\mathcal{G}$

Table 2.1: Notations frequently used in this paper.

3 Algorithm Overview

In this section, we generalize the algorithm framework for subgraph enumeration, based on which we can describe the TwinTwigJoin algorithm [20] and SEED algorithm.

3.1 Algorithm Framework

We solve the subgraph enumeration in a decomposition-and-join manner. Specifically, we first decompose the pattern graph into a set of structures, called *join unit*, then we join the matches of these join units to get the results.

Graph Storage. To determine what structure can be the join unit, we first introduce the *graph storage mechanism*, which is defined as $\Phi(G) = \{G_u \mid u \in V(G)\}$, where $G_u \subseteq G$ is a connected subgraph of G with $u \in V(G_u)$, and we have $\bigcup_{u \in V(G)} E(G_u) = E(G)$. Each G_u is called the *local graph* of u. Specifically, the data graph G is maintained in the distributed file system in the form of key-value pairs $(u; G_u)$ for each $u \in V(G)$ according to $\Phi(G)$. We then define the *join unit* as:

Definition 3. ***(Join Unit)*** *Given a data graph G and the graph storage $\Phi(G) = \{G_u \mid u \in V(G)\}$, a connected structure p is a join unit w.r.t. $\Phi(G)$, if and only if*

$$R_G(p) = \bigcup_{G_u \in \Phi(G)} R_{G_u}(p).$$

In other words, a join unit is a structure whose matches can be enumerated independently in each local graph $G_u \in \Phi(G)$. We further define *pattern decomposition* as:

Definition 4. *(**Pattern Decomposition**) Given a pattern graph P, a pattern decomposition is denoted as $D = \{p_0, p_1, \ldots, p_t\}$, where $p_i \in P (0 \leq i \leq t)$ is a join unit and $P = p_0 \cup p_1 \cup \cdots \cup p_t$.*

Join Plan. Given the decomposition $D = \{p_0, p_1, \ldots, p_t\}$ of P, we solve the subgraph enumeration using the following join:

$$R(P) = R(p_0) \bowtie R(p_1) \bowtie \cdots \bowtie R(p_t). \tag{3.1}$$

A *join plan* determines an order to solve the above join, and it processes t rounds of two-way joins. We denote P_i as the i[-th] *partial pattern* whose results are produced in the i[-th] round of the join plan. Obviously, we have $P_t = P$. The join plan is usually presented in a tree structure, where the leaf nodes are (the matches of) the join units, the internal nodes are the partial results - the matches of the partial patterns.

A join tree uniquely specifies a join plan, and we use join tree and join plan interchangeably. If all internal nodes of the join tree has *at least* one join unit as its child, the tree is called a left-deep tree [1]. Otherwise it is called a bushy tree [16]. Note that a left-deep tree is also a bushy tree.

Example 2. *Consider the pattern graph P and its decomposition $D(P) = \{q_0, q_1, q_2, q_3\}$ in the left part of Figure 3.1. Here we use the triangle (3-clique) as the join unit. We present a left-deep tree $\mathcal{E}_1$ and a bushy tree $\mathcal{E}_2$ to solve $R(P) = R(p_0) \bowtie R(p_1) \bowtie R(p_2) \bowtie R(p_3)$. They both process three rounds. We denote P_i^{ld} and P_i^b as the i[-th] partial patterns in the left-deep tree and the bushy tree, respectively. For example, in the first round of the left-deep tree, we process $R(P_1^{ld}) = R(p_0) \bowtie R(p_1)$ to produce the matches of the partial pattern P_1^{ld}. Observe that in the left-deep tree, each internal tree node (R_1^{ld}, R_2^{ld} and $R(P)$) has a join unit as its child, while in the bushy tree, neither children of $R(P)$ are join units.*

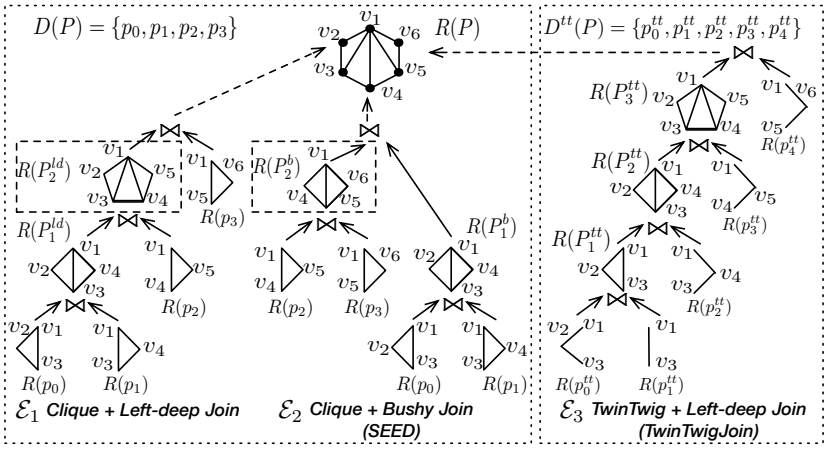


Figure 3.1: Different Join Trees.

Execution Plan. An *execution plan* of subgraph enumeration task, denoted as $\mathcal{E} = (D, J)$, contains two parts - a *pattern decomposition* D and a *join plan* J. Consider an execution space Σ and a cost function $\mathcal{C}$ defined over Σ. We formulate the problem of *optimal execution plan* for solving subgraph enumeration as follows:

[1] More accurately, it is the deep tree, which is further classified into the left-deep and right-deep tree. As it is insignificant to distinguish them in this paper, we simply refer to the deep tree as left-deep tree.

Definition 5. ***(Optimal Execution Plan)*** *An optimal execution plan for solving subgraph enumeration is an execution plan $\mathcal{E}_o = (D_o, J_o) \in \Sigma$ to enumerate P in G using Equation 3.1, such that,*

$$\mathcal{C}(\mathcal{E}_o) \text{ is minimized.}$$

3.2 TwinTwigJoin

We briefly introduce the TwinTwigJoin algorithm by showing its storage mechanism and the left-deep join framework.

Graph Storage. We denote the storage mechanism used in TwinTwigJoin as $\Phi^0(G) = \{G^0_u \mid u \in V(G)\}$, where $V(G^0_u) = \{u\} \cup \mathcal{N}(u)$ and $E(G^0_u) = \{(u, u')|u' \in \mathcal{N}(u)\}$ [20]. A star is a qualified join unit w.r.t. $\Phi^0(G)$, as the matched stars rooted at u can be independently generated by enumerating the node combinations in $\mathcal{N}(u)$. Aware that a general star may introduce enormous cost, TwinTwigJoin utilizes TwinTwig, a star with either one or two edges, as the join unit.

Left-deep Join. After decomposing the pattern graph into a set of TwinTwigs, TwinTwigJoin solves Equation 3.1 using a left-deep join structure, which processes t rounds of joins, and the following join is executed in the i[-th] round:

$$R(P_i) = R(P_{i-1}) \bowtie R(p_i),$$

where $P_0 = p_0$. In order to approach optimality, TwinTwigJoin exhaustively traverses all possible left-deep join plans, evaluates the cost of each plan based on the ER model, and selects the one with the minimal cost as the optimal plan.

In Figure 3.1, we show the optimal execution plan $\mathcal{E}_3$ of TwinTwigJoin for the given P, which includes the TwinTwig decomposition $D^{tt}(P)$ and the optimal left-deep join plan.

Drawbacks. There are three major drawbacks of the TwinTwigJoin algorithm. First, the simple graph storage mechanism only supports using star as the join unit, which can result in severe scalability issues. Although TwinTwigJoin uses TwinTwig as a substitution, the issues are only mitigated but not evaded, especially when handling nodes with very large degree. Second, TwinTwigJoin should process at least $\frac{m}{2}$ rounds, which limits its utilization for complex pattern graph. Ultimately, the left-deep join framework may render sub-optimal solution as it only searches for "optimality" in the left-deep space [18].

3.3 SEED

SEED tackles the issues of TwinTwigJoin by introducing the SCP *graph storage mechanism* and *the optimal bushy join structure*, which greatly improve the performance.

SCP **Graph Storage.** According to Definiton 3, the storage mechanism $\Phi(G)$ determines the join unit. We say $\Phi(G)$ is *p-preserved* if p can be a join unit w.r.t. $\Phi(G)$. In particular, we define the **S**tar-**C**lique-**P**reserved (SCP) storage mechanism as:

Definition 6. ***(SCP storage mechanism)*** $\Phi(G) = \{G_u \mid u \in V(G)\}$ *is an* SCP *storage mechanism, if both star and clique can be the join units w.r.t.* $\Phi(G)$.

The storage mechanism $\Phi^0(G)$ used in TwinTwigJoin is not an SCP storage mechanism, as clique can not be used as the join unit. With clique as an alternative, we can

Algorithm 1: SEED(data graph G, pattern graph P)

Input : G : The data graph, stored as $\Phi(G) = \{G_u \mid u \in V(G)\}$,
P : The pattern graph.
Output : $R(P)$: All Matches of P in G.

1 $\mathcal{E}_o \leftarrow$ computeExecutionPlan(G, P); (Algorithm 2)
2 **for** $i = 1$ **to** t **do**
3 $\quad R(P_i) \leftarrow R(P_i^l) \bowtie R(P_i^r)$ according to $\mathcal{E}_o$ (using map^i and reduce^i);
4 **return** $R(P_t)$;

5 **function** map^i(**key**: $\emptyset$; **value**: Either a match $f \in R(P_i^l)$, $h \in R(P_i^r)$ or $G_u \in \Phi(G)$)
6 $V_k = \{v_{k_1}, v_{k_2}, \ldots, v_{k_s}\} \leftarrow V(P_i^l) \cap V(P_i^r)$;
7 **if** P_i^l is a join unit **then** genJoinUnit(P_i^l, G_u, V_k);
8 **else** output $((f(v_{k_1}), f(v_{k_2}), \ldots, f(v_{k_s})); f)$;
9 **if** P_i^r is a join unit **then** genJoinUnit(P_i^r, G_u, V_k);
10 **else** output $((h(v_{k_1}), h(v_{k_2}), \ldots, h(v_{k_s})); h)$;

11 **function** genJoinUnit$(p, G_u, V_k = \{v_{k_1}, v_{k_2}, \ldots, v_{k_s}\})$
12 $R_{G_u}(p) \leftarrow$ all matches of p in G_u;
13 **forall the** match $f \in R_{G_u}(p)$ **do**
14 $\quad$ output $((f(v_{k_1}), f(v_{k_2}), \ldots, f(v_{k_s})); f)$;

15 **function** reduce^i(**key**:$r = (u_{k_1}, u_{k_2}, \ldots, u_{k_s})$; **value**: $F = \{f_1, f_2, \ldots\}$, $H = \{h_1, h_2, \ldots\}$)
16 **forall the** $(f, h) \in (F \times H)$ s.t. $(f - r) \cap (h - r) = \emptyset$ **do**
17 $\quad$ output $(\emptyset; f \cup h)$;

avoid processing star where possible, which not only saves cost in a single run, but reduces the rounds of execution as a whole. For example, the plans $\mathcal{E}_1$ and $\mathcal{E}_3$ shown in Figure 3.1 are both left-deep joins, but $\mathcal{E}_1$ uses triangles, while $\mathcal{E}_3$ uses TwinTwigs as the join units. Intuitively, we expect that $\mathcal{E}_1$ draws smaller cost as the triangle has much fewer matches than a two-edge TwinTwig does, and $\mathcal{E}_1$ is one round less than $\mathcal{E}_3$. We will detail the SCP storage mechanism in Section 4.

Bushy Join. SEED solves Equation 3.1 by exploiting the bushy join structure. Specifically, the following join is processed in the i[-th] round:

$$R(P_i) = R(P_i^l) \bowtie R(P_i^r), \tag{3.2}$$

where P_i^l and P_i^r are called the left and right join patterns, respectively. The left and right join patterns can be either a join unit, or a partial pattern processed in an earlier round.

The execution plan $\mathcal{E}_2$ in Figure 3.1 is a bushy tree, in which we have two join units $R(p_0)$ and $R(p_1)$ as the left and right join patterns in the first round, and two partial patterns $R(P_1^b)$ and $R(P_2^b)$ in the third round.

Compared to TwinTwigJoin, SEED searches the optimal solution among the bushy trees, which covers the whole searching space, and thus guarantees the optimality of the solution.

Algorithm. We show the algorithm of SEED in Algorithm 1. Given the pattern graph P, we first compute the optimal execution plan $\mathcal{E}_o$ in line 1 using Algorithm 2 (details in Section 6). According to the optimal execution plan $\mathcal{E}_o$, the i[-th] join in Equation 3.2 is processed using MapReduce via map^i and reduce^i (line 2). We apply the same reduce^i as in TwinTwigJoin, thus we focus on map^i here.

The function map^i is shown in lines 5-10. The inputs of map^i are either a match

$f \in R(P_i^l)$, a match $h \in R(P_i^r)$ or $(u; G_u)$ for all $G_u \in \Phi(G)$ if we are dealing with a join unit (line 5). We first calculate the join key $\{v_{k_1}, v_{k_2}, \ldots, v_{k_s}\}$ using $V(P_i^l) \cap V(P_i^r)$ (line 6). Then we compute the matches of P_i^l and P_i^r. Take P_i^l for example. We know whether P_i^l is a join unit in current round according to the execution plan. If P_i^l is a join unit, we invoke genJoinUnit (P_i^l, G_u, V_k) (line 7) to compute the matches of P_i^l in G_u for each $G_u \in \Phi(G)$ (lines 12-14). Note that $R(P_i^l)$ are complete by merging $R_{G_u}(P_i^l)$ for all $G_u \in \Phi(G)$ according to Definiton 3. If P_i^l is not a join unit, the matches of P_i^l must have been computed in previous round. Then we directly fetch the partial results and output them with the join key (line 8).

Challenges. To pursuit the optimality for SEED, we have to address multiple key challenges. Specifically,

- It is non-trivial to develop an effective SCP graph storage mechanism. In order to use clique as join unit, we have to introduce extra edges to the simple local graph used in TwinTwigJoin. However, the size of each local graph should not be too large for scalability consideration.
- A well-defined cost function is required to estimate the cost of each execution plan. In the subgraph enumeration problem, the tuples that participate in the joins are the matches of certain pattern graph, whose size is difficult to estimate, especially in a power-law graph.
- It is in general computationally intractable to compute an optimal join plan [18]. In TwinTwigJoin, an easier-solving left-deep join is applied, which may render sub-optimal solution. In this work, we target on computing the optimal bushy join plan - a much harder task given the larger searching space [16].

4 Beyond Stars: SCP Storage

In this section, we will propose an effective SCP storage mechanism, in which each local graph introduces a small number of extra edges to the local graph used in TwinTwigJoin. We leverage the PR model for analysis. Recall that w_i is the expected degree for the node u_i and w is the expected average degree, and β is the power-law exponent. Denote $\widetilde{w}$ as the second-order average degree, which can be computeed as [4]:

$$\widetilde{w} = (\sum_{i=1}^{N} w_i^2)/(\sum_{i=1}^{N} w_i) = \Psi w^{\beta-2} w_{max}^{3-\beta},$$

where $\Psi = \frac{(\beta-2)^{\beta-1}}{(3-\beta)(\beta-1)^{\beta-2}}$.

As we mentioned earlier, the storage mechanism - $\Phi^0(G)$ - used in TwinTwigJoin is not an SCP storage mechanism. In the following, we will explore two SCP storage mechanisms - $\Phi^1(G)$ and $\Phi^2(G)$, in which the local graphs are respectively denoted as G_u^1 and G_u^2 for each $u \in V(G)$. In order to use clique as the join unit, both mechanisms introduce extra edges to each local graph in $\Phi^0(G)$. We denote $\Delta_u^{(i)} = \mathbb{E}[|E(G_u^i)|] - \mathbb{E}[|E(G_u^0)|]$ as the expected number of extra edges introduced by $\Phi^i(G)$ for $u \in V(G)$, and let $\Delta_{max}^{(i)} = \max_{u \in V(G)}\{\Delta_u^{(i)}\}$ for $i \in \{1, 2\}$.

SCP **Graph Storage**. Let $\Phi^1(G) = \{G_u^1 \mid u \in V(G)\}$, where $V(G_u^1) = V(G_u^0)$ and $E(G_u^1) = E(G_u^0) \cup \{(u', u'') \,|\, u', u'' \in \mathcal{N}(u) \wedge (u', u'') \in E(G)\}$. We divide the edges of each G_u^1 into two parts, the *neighbor edges* $E(G_u^0)$, and the *triangle edges* that close triangles with the neighbor edges. Clearly the triangle edges are extra edges

introduced by $\Phi^1(G)$. The following lemma shows that $\Phi^1(G)$ is an SCP storage mechanism.

Lemma 1. *Given the storage mechanism $\Phi^1(G) = \{G^1_u \mid u \in V(G)\}$, p is a join unit w.r.t. $\Phi^1(G)$ if p is a star or a clique.*

Proof. Clearly, $\Phi^1(G)$ is star-preserved since each G^1_u contains G^0_u. Consider p is a k-clique. We assume that its matches exist in the data graph, as otherwise it's pointless. Let $V(p) = \{v_0, v_1, v_2 \ldots, v_{k-1}\}$. We prove that $\forall f \in R_G(p)$, $\exists u \in V(G)$ such that $f \in R_{G^1_u}(p)$. According to Definiton 3, this sufficiently concludes that p is a join unit. Consider $G^1_{u_0}$ where $u_0 = f(v_0)$ for a given f. Obviously, $(v_0, v_i) \in E(p)$ for any $1 \leq i \leq k-1$, which gives $(u_0, f(v_i)) \in E(G)$, and more specifically, $(u_0, f(v_i)) \in E(G^1_{u_0})$ as they are the neighbor edges of u_0. Furthermore, as $(v_i, v_j) \in E(p)$ for any $i \neq j$, we have $(f(v_i), f(v_j)) \in E(G)$. We know both $f(v_i)$ and $f(v_j)$ are the neighbors of u_0, thus we have $(f(v_i),\ f(v_j)) \in E(G^1_{u_0})$ as the triangle edge. It is immediate that $f \in R_{G^1_{u_0}}(p)$. □

Despite $\Phi^1(G)$ is an SCP storage mechanism, it can introduce numerous extra edges to a certain local graph in $\Phi^0(G)$, as shown in the following lemma.

Lemma 2. *Given a* PR *graph $\mathcal{G}$, and the node $u_i \in V(\mathcal{G})$, we have*

$$\Delta^{(1)}_{u_i} = \Psi^2 w^{\beta-2} N^{2-\beta} w_i^2, \text{ and}$$
$$\Delta^{(1)}_{max} = \Psi^2 w^{\beta-1} N^{3-\beta}.$$

Proof. We denote t_i as the expected number of triangles associated with u_i. It is easy to see $G^1_{u_i}$ contains w_i neighbor edges and t_i triangle edges by expectation. Thus:

$$\Delta^{(1)}_{u_i} = \mathbb{E}[|G^1_{u_i}|] - E[d(u_i)] = t_i.$$

Recall $\widetilde{w} = \frac{\sum_{i=1}^N w_i^2}{\sum_{i=1}^N w_i}$ is the second-order average degree. When $2 < \beta < 3$, we can compute $\widetilde{w}$ as [4]:

$$\widetilde{w} = \Psi w^{\beta-2} w_{max}^{3-\beta}, \tag{4.1}$$

where $\Psi = \frac{(\beta-2)^{\beta-1}}{(3-\beta)(\beta-1)^{\beta-2}}$.

For a given node u_i, we will locate u_j and u_k in the data graph to close a triangle. Following the PR model, we have:

$$\begin{aligned}
\Delta^{(1)}_{u_i} = t_i &= \sum_{j=1}^{N}\sum_{k=1}^{N} \rho w_i w_j \times \rho w_i w_k \times \rho w_j w_k \\
&= w_i^2 \rho \widetilde{w}^2 = w_i^2 \frac{\widetilde{w}^2}{N \times w} \quad (\text{by } \rho = \frac{1}{\sum_{i=1}^N w_i} = \frac{1}{N \times w}) \\
&= \Psi^2 w^{\beta-2} N^{2-\beta} w_i^2 \quad (\text{ by } w_{max} = \sqrt{wN})
\end{aligned}$$

We immediately have $\Delta^{(1)}_{max} = \Psi^2 w^{\beta-1} N^{3-\beta}$ by letting $w_i = w_{max}$. □

Lemma 2 shows that the number of extra edges introduced by $G^1_{u_i}$ is nearly proportional to w_i^2, which can cause severe workload skew and thus hamper the scalability of the algorithm.

Compact SCP Graph Storage. Targeting the deficiencies of $\Phi^1(G)$, we consider a more compact storage mechanism by leveraging the node order (Definiton 2). Specifically, we define $\Phi^2(G) = \{G^2_u \mid u \in V(G)\}$, where $V(G^2_u) = V(G^0_u)$ and

$E(G_u^2) = E(G_u^0) \cup \{(u', u'') \,|\, (u', u'') \in E(G) \wedge u \prec u' \wedge u \prec u''\}$. Compared to G_u^1, G_u^2 only includes the triangle edge when u is the minimum node in the triangle. It is clear that $G_u^0 \subseteq G_u^2 \subseteq G_u^1$. Next, we show that $\Phi^2(G)$ is also an SCP storage mechanism.

Corollary 1. *Consider a pattern graph P and its two nodes v_1, v_2, where v_1 are adjacent to all nodes in P. If an order is assigned between v_1 and v_2 using the symmetry-breaking algorithm (see the appendix), then v_2 must be adjacent to all nodes of P.*

Proof. As v_1 and v_2 are assigned an order using symmetry breaking, there must exist an automorphism (a match from P to itself) σ, such that $\sigma(v_1) = v_2$. By Structure-Preservation (Definiton 1), v_2 must be adjacent all nodes as v_1 in P. □

Lemma 3. *Given the storage mechanism $\Phi^2(G) = \{G_u^2 \mid u \in V(G)\}$, p is a join unit w.r.t. $\Phi^2(G)$ **if and only if** p is a star or a clique.*

Proof. (**If**.) Clearly, $\Phi^2(G)$ is star-preserved since each G_u^2 contains G_u^0. We next show a k-clique is a join unit w.r.t. $\Phi^2(G)$. Given a k-clique p where $V(p) = \{v_1, v_2, \ldots, v_k\}$, we apply a full order $v_1 < v_2 < \cdots < v_k$ for symmetry breaking [11]. Assume that the match of p exists. We prove that, $\forall f \in R_G(p)$, $\exists u \in V(G)$, such that $f \in R_{G_u^2}(p)$. Given a match f, we let $f(v_1) = u_1$. For any $2 \leq i < j \leq k$, it is clear that $v_1 < v_i < v_j$ and v_1, v_i, v_j close a triangle in p. This suggests $u_1 \prec f(v_i) \prec f(v_j)$ and $u_1, f(v_i), f(v_j)$ close a triangle in G. By the definition of G_u^2, we have:

$$(u_1, f(v_i)), (u_1, f(v_j)) \text{ and } (f(v_i), f(v_j)) \in E(G_{u_1}^2)$$

In other words, $G_{u_1}^2$ involves every edge of the matched instance of p. Hence, $f \in R_{G_{u_1}^2}(p)$.

(**Only If**.) We prove this by contradiction. Let $V(p) = \{v_1, v_2, \ldots, v_n\}$, and one of its match be $f = (u_1, u_2, \ldots, u_n)$. Assume that p is neither a star nor a clique and the match f is preserved in some G_u^2. Given the structure of G_u^2, there must exist a node v_1 (w.l.g.) in p adjacent to all the other nodes of P, and the match must be preserved in $G_{u_1}^2$. There are at least two nodes that have no edge. Let them be v_2, v_3 (w.l.g.), and we assume that $(u_2, u_3) \notin E(G)$, where $u_2 = f(v_2)$ and $u_3 = f(v_3)$. As p is not a star, we must have at least two nodes v_i, v_j such that $(v_i, v_j) \in E(p)$ and $v_i, v_j \neq v_1$. There are two cases: (1) v_2 or v_3 is one of v_i, v_j; (2) neither v_2 nor v_3 is v_i or v_j. We show both cases are impossible. In case 1, let $v_i = v_2$ and $v_j = v_4$ (w.l.g.). Clearly, $(v_2, v_4) \in E(p)$ implies $(u_2, u_4) \in E(G_{u_1}^2)$. By the definition of G_u^2, we have $u_1 \prec u_2$ and $u_1 \prec u_4$, and by the order-preservation match, we must have $v_1 < v_2$ and $v_1 < v_4$. According to Corollary 1, v_2 must be adjacent to all nodes in p, this makes a contradiction as v_2 does not connect v_3. In case 2, let $v_i = v_4$ and $v_j = v_5$ (w.l.g.). Similar to case 1, this implies that v_4 connects v_2, which reduces to case 1 that has made a contradiction already. □

The next lemma shows that $\Phi^2(G)$ brings in much less extra edges than $\Phi^1(G)$ does to each local graph in $\Phi^0(G)$.

Lemma 4. *Given a* PR *graph $\mathcal{G}$ and a node $u_i \in V(\mathcal{G})$, we have*

$$\Delta_{u_i}^{(2)} \leq \Delta_{max}^{(2)} \leq \left[(3-\beta)(4-\beta)^{-\frac{4-\beta}{3-\beta}}\right]^2 \Psi^2 w^{\beta-1} N^{3-\beta}.$$

Proof. Let T'_i denote the set of triangles in T_i that have u_i as the minimum node, and $t'_i = |T'_i|$. We have:

$$\Delta^{(2)}_{u_i} = \mathbb{E}[|E(G^2_{u_i})|] - \mathbb{E}[d(u_i)] = t'_i.$$

Consider a triangle (u_i, u_j, u_k) rooted on u_i such that $u_i \prec u_j$ and $u_i \prec u_k$. Recall that we arrange the nodes in $\mathcal{G}$ by non-decreasing order of their degree. We hence have $j \geq i+1$, $k \geq i+1$. Therefore:

$$\begin{aligned}\Delta^{(2)}_{u_i} = t'_i &= \sum_{j,k=i+1}^{N} \rho w_i w_j \times \rho w_i w_k \times \rho w_j w_k \\ &= \rho w_i^2 \big(\rho \sum_{j=i+1}^{N} w_j^2\big)^2 = \rho w_i^2 \big(\rho \sum_{j=1}^{N} w_j^2 - \rho \sum_{j=1}^{i} w_j^2\big)^2.\end{aligned}$$

Note that $w_1, w_2, \ldots, w_i$ is a degree sequence that has w_i as the maximum degree. We then construct another sequence $w'_1, w'_2, \ldots, w'_i$ with $w'_i = w_i$ that has the same power-law distribution as the original sequence. In other words, they have the same β value. Since $i < N$, it is immediate that the frequency of w'_j must be smaller than that of w_j for any $1 \leq j \leq i$. It is easy to find:

$$\rho \sum_{j=1}^{i} w_j^2 \geq \rho \sum_{j=1}^{i} w_j'^2 = \Psi w^{\beta-2} w_i^{3-\beta}.$$

Therefore, we have:

$$\Delta^{(2)}_{u_i} \leq \varphi(w_i) = \Psi^2 w^{2(\beta-2)} \times \rho w_i^2 \big(w_{max}^{3-\beta} - w_i^{3-\beta}\big)^2.$$

Let $\frac{\partial \varphi(w_i)}{\partial w_i} = 0$, we have $w_i = \frac{w_{max}}{(4-\beta)^{1/(3-\beta)}}$, and:

$$\begin{aligned}\Delta^{(2)}_{max} &\leq \Psi^2 w^{2(\beta-2)} \rho \big[\frac{w_{max}}{(4-\beta)^{1/(3-\beta)}}\big]^2 (\frac{3-\beta}{4-\beta})^2 (w_{max}^2)^{3-\beta} \\ &= \big[(3-\beta)(4-\beta)^{-\frac{4-\beta}{3-\beta}}\big]^2 \Psi^2 w^{\beta-1} N^{3-\beta}\end{aligned}$$

□

In Lemma 4, we give an upper bound of $\Delta^{(2)}_{u_i}$, while its value is often much smaller. Specifically, $\Delta^{(2)}_{u_i} = 0$ when $d(u_i) = 1$ and $d(u_i) = \max_{u \in V(G)} d(u)$. In general, we show that $\Delta^{(2)}_{max}$ is much smaller than $\Delta^{(1)}_{max}$. In the PR graph, we set $w = 50$, $N = 1{,}000{,}000$ and vary $\beta = 2.1, 2.3, 2.5, 2.7, 2.9$, and then compare $\Delta^{(1)}_{max}$ and $\Delta^{(2)}_{max}$ in Table 4.1. It is clear that $\Delta^{(2)}_{max} \ll \Delta^{(1)}_{max}$ in all cases. When β increases, observe that $\Delta^{(2)}_{max}$ decreases significantly while $\Delta^{(1)}_{max}$ remains in the same order. In the experiments (Exp-1 in Section 8), we further compared $\Delta^{(1)}_{u}$ with $\Delta^{(2)}_{u}$ for each data node u using synthetic and real datasets, and the experimental results demonstrate that $\Delta^{(2)}_{u} \ll \Delta^{(1)}_{u}$ for all data nodes except those with very small degree in all datasets.

Δ_{max}	$\beta = 2.1$	$\beta = 2.3$	$\beta = 2.5$	$\beta = 2.7$	$\beta = 2.9$
$\Delta^{(1)}_{max}$	141,939	195,260	117,851	76,685	141,797
$\Delta^{(2)}_{max}$	7,652	7,652	2,586	710	174

Table 4.1: The number of extra edges introduced by G^1_u and G^2_u.

Discussion. Clearly, the more join units to support, the more edges we should involve in each local graph of the storage mechanism, which can hamper the scalability of the algorithm. We carefully consider such a tradeoff in this paper.

Given a node v in P, we say v' is its k-hop neighbor if there is a shortest path of length k between v and v'. A graph is called a *one-hop* graph if there is a node connecting all other nodes, and a *multi-hop* graph otherwise. Clearly, star and clique are both one-hop graphs. There are three cases regarding the usable join units.

- *All one-hop graphs*. We can actually adopt $\Phi^1(G)$ to support all one-hop graphs as the join units, which is yet unaffordable according to Lemma 2.
- *Multi-hop graphs*. In order to support multi-hop graphs, we need to involve all two-or-more-hop neighbors of u and relevant edges into G_u, which will render an even larger local graph than G_u^1.
- *Part of one-hop graphs*. According to Lemma 3, $\Phi^2(G)$ only supports star and clique as the join units. In order to support the other one-hop graphs, we have to involve more edges to each $G_u^2 \in \Phi^2(G)$, which makes it hard to bound the size of each local graph. We considers this as a future work to find other storage mechanisms that support more one-hop graphs as the join units, yet still have size-bounded local graphs.

Consequently, we adopt $\Phi^2(G)$ as the storage mechanism in this paper, which only supports star and clique as the join units. In the following, we will refer to G_u^2 simply as G_u if not specifically mentioned.

Implementation Details. Given a data graph G, we implement $\Phi^2(G)$ by constructing G_u^2 for each $u \in V(G)$. Specifically, we first aggregate the neighbor edges of each u to G_u^2. Then we apply existing triangle enumeration approaches such as [20, 15, 1]. For each triangle (u_1, u_2, u_3) generated with $u_1 \prec v_2 \prec v_3$, we add (u_2, u_3) to $G_{u_1}^2$ as the triangle edge. With G_u^2 for each $u \in V(G)$, we can compute the matches of any star or clique using an in-memory algorithm. The overheads of constructing the new graph storage is dominated by triangle enumeration, which are relatively small, as shown in the experiment, comparing to the performance gains of using clique as the join unit.

5 Cost Analysis

We follow the cost model in TwinTwigJoin by summarizing the map data $\mathcal{M}$ (the input and output data of the mapper), shuffle data $\mathcal{S}$ (the data transferred from mapper to reducer) and reduce data $\mathcal{R}$ (the input and output data of reducer) in each round of Algorithm 1. These data include the communication I/O among machines and the disk I/O of reading and writing the partial results, which dominate the cost in MapReduce [20]. Considering that most real-life graphs are power-law graphs, we further contribute to estimate the number of matches of a pattern graph based on the PR model instead of the ER model [20], and we show that the PR model delivers more realistic estimations.

To show how we compute the cost, we first consider an arbitrary join $R(P_\beta) = R(p) \bowtie R(P_\alpha)$, where p is a join unit and P_α, P_β are two partial patterns. Let $\mathcal{M}(P)$, $\mathcal{S}(P)$ and $\mathcal{R}(P)$ denote the map data, shuffle data and reduce data regarding a certain graph P. According to Algorithm 1, we have:

- The mapper handles the partial pattern P_α and the join unit p in different ways. For P_α, the mapper takes the matches $R(P_\alpha)$ as inputs and directly outputs them with the join key. Therefore, $\mathcal{M}(P_\alpha) = 2R(P_\alpha)$. As for the join unit p, the mapper first reads G_u for each data node u to compute $R(p)$, then outputs the results. Denote $\Delta(G)$ as the set of triangles in G, and it is clear that $\sum_{u \in V(G)} E(G_u) = \Delta(G)$. Therefore, $\mathcal{M}(p) = \Delta(G) + R(p)$.

- The shuffle transfers the mapper's outputs to the corresponding reducer. Therefore, the shuffle data is also the mapper's output data, and we have $\mathcal{S}(P_\alpha) = R(P_\alpha)$ and $\mathcal{S}(p) = R(p)$.
- The reducer takes $R(P_\alpha)$ and $R(p)$ as inputs to compute $R(P_\beta)$. Apparently, the input data are $\mathcal{R}(P_\alpha) = R(P_\alpha)$ and $\mathcal{R}(p) = R(p)$, and the output data are $\mathcal{R}(P_\beta) = R(P_\beta)$.

Summarizing the above, the cost for processing the join unit p in a certain join is:

$$\mathcal{T}(p) = |\mathcal{M}(p)| + |\mathcal{S}(p)| + |\mathcal{R}(p)| = |\Delta(G)| + 3|R(p)| \tag{5.1}$$

and the cost for processing the partial pattern P_α is:

$$\mathcal{T}(P_\alpha) = |\mathcal{M}(P_\alpha)| + |\mathcal{S}(P_\alpha)| + |\mathcal{R}(P_\alpha)| = 5|R(P_\alpha)| \tag{5.2}$$

Note that $R(P_\alpha)$ must have been generated in earlier round, while the cost to output $R(P_\alpha)$ is involved in $\mathcal{T}(P_\alpha)$ for consistency, and $R(P_\beta)$ will be accordingly computed in $\mathcal{T}(P_\beta)$.

Given an execution plan $\mathcal{E} = (D, J)$, where $D = \{p_0, p_1, \ldots, p_t\}$, it is processed using t rounds of joins, and in the i[-th] round the partial results $R(P_i)$ are generated. We compute the cost by putting all costs of processing p_i and P_i together as:

$$\mathcal{C}(\mathcal{E}) = \sum_{i=0}^{t} \mathcal{T}(p_i) + \sum_{i=1}^{t-1} \mathcal{T}(P_i), \tag{5.3}$$

where $\mathcal{T}(p_i)$ and $\mathcal{T}(P_i)$ are computed via Equation 5.1 and Equation 5.2, respectively.

Remark 3. *We present the cost model using MapReduce for easy understanding. Nevertheless, the cost model can be applied to other platforms with minor modifications. Take Spark as an example. Spark can maintain the intermediate results in the main memory between two successive iterations. Therefore, we do not need to consider the cost of accessing these data on the disk, which corresponds to the map data and reduce data in MapReduce.*

5.1 Result-Size Estimation

In order to compute the cost, we need to estimate $|R(P)|$ for a certain P in Equation 5.1 and Equation 5.2. It is obvious that all partial patterns in Algorithm 1 are connected. Given a connected pattern graph P, we next show how to estimate $|R_\mathcal{G}(P)|$ in the PR graph $\mathcal{G}$.

Suppose P is constructed from an edge by extending one edge step by step, and $P^{(1)}$ and $P^{(2)}$ are two consecutive patterns obtained along the process. More specifically, given $v \in V(P^{(1)})$ and $v' \in V(P^{(2)})$ where $(v, v') \notin E(P^{(1)})$, $P^{(2)}$ is obtained by adding the edge (v, v') to $P^{(1)}$. We let δ and δ' be the degrees of v and v' in $P^{(1)}$, respectively. Here, if $v' \notin V(P^{(1)})$, $\delta' = 0$. Given a match f of $P^{(1)}$, we let $f(v) = u$. We then extend f to generate the match f' of $P^{(2)}$ by locating another node $u' \in V(G)$ where $(u, u') \in E(G)$ and $f'(v') = u'$. Suppose there are by expectation γ matches of $P^{(2)}$ that can be extended from one certain match of $P^{(1)}$, we have:

$$|R_\mathcal{G}(P^{(2)})| = \gamma |R_\mathcal{G}(P^{(1)})|$$

The value of γ depends on how the edge is extended from $P^{(1)}$ to form $P^{(2)}$. There are two cases, namely, $v' \notin V(P^{(1)})$ and $v' \in V(P^{(1)})$, which are respectively discussed in the following.

(Case 1) $v' \notin V(P^{(1)})$. In this case, a new node v' is introduced to extend the edge (v, v'). The potential match of v', namely u', can be any data node in G. Therefore, we have:

$$\begin{aligned}\gamma &= \mathbb{E}\big[\sum_{u' \in V(G)} d(u)d(u')\rho\big] = \mathbb{E}[d(u)] \times \rho \sum_{u' \in V(G)} \mathbb{E}[d(u')] \\ &= \mathbb{E}[d(u)] \times \rho \sum_{i=1}^{N} w_i = \mathbb{E}[d(u)] = \sum_{i=1}^{N} \phi_i w_i.\end{aligned}$$

where ϕ_i is the probability that u appears as u_i in the matches of $P^{(1)}$.

For ease of analysis, we focus on the relationships between u and its neighbors. Denote k-star(u) as the star with k leaves rooted on u. Clearly, u and its neighbors in the match form a δ-star(u) (Note that the degree of u in the match is equal to the degree of v in $P^{(1)}$, that is δ). Thus, we have:

$$\begin{aligned}\phi_i &= \mathsf{Pr}(u = u_i \mid u \text{ and its neighbors form a } \delta\text{-star}(u)) \\ &= \frac{\mathsf{Pr}(u \text{ and its neighbors form a } \delta\text{-star}(u) \mid u = u_i)\mathsf{Pr}(u = u_i)}{\mathsf{Pr}(u \text{ and its neighbors form a } \delta\text{-star}(u))} \\ &= \frac{\mathsf{Pr}(u \text{ and its neighbors form a } \delta\text{-star}(u) \mid u = u_i)}{\sum_{j=1}^{N} \mathsf{Pr}(u \text{ and its neighbors form a } \delta\text{-star}(u) \mid u = u_j)} \\ &= \frac{\mathsf{Pr}(\mathsf{E}_i)}{\sum_{j=1}^{N} \mathsf{Pr}(\mathsf{E}_j)}.\end{aligned}$$

where E_i denotes the event that u and its neighbors form a δ-star(u) given $u = u_i$. Given any node set $\{u_{t_1}, u_{t_2}, \ldots, u_{t_\delta}\} \subset V(G)$, E_i can be witnessed as there is an edge connecting u_i and u_{t_s} for any $1 \leq s \leq \delta$. According to the PR model, the probability that any u_i and u_j are connected is $\mathsf{Pr}_{i,j} = w_i w_j \rho$, hence we have:

$$\begin{aligned}\mathsf{Pr}(\mathsf{E}_i) &= \sum_{\{u_{t_1}, u_{t_2}, \ldots, u_{t_\delta}\} \subset V(G)} \mathsf{Pr}_{i,t_1} \times \cdots \times \mathsf{Pr}_{i,t_\delta} \\ &= \sum_{\{u_{t_1}, u_{t_2}, \ldots, u_{t_\delta}\} \subset V(G)} w_i w_{t_1} \rho \times \cdots \times w_i w_{t_\delta} \rho \\ &= (w_i)^\delta \times C,\end{aligned}$$

where $C = \rho^\delta \sum_{\{u_{t_1}, u_{t_2}, \ldots, u_{t_\delta}\} \in V(G)} \prod_{s=1}^{\delta} w_{t_s}$.

Note that C is a constant for any i. Therefore:

$$\phi_i = \frac{\mathsf{Pr}(\mathsf{E}_i)}{\sum_{j=1}^{N} \mathsf{Pr}(\mathsf{E}_j)} = \frac{w_i^\delta}{\sum_{j=1}^{N} w_j^\delta}, \tag{5.4}$$

and

$$\gamma = \sum_{i=1}^{N} \frac{w_i^\delta}{\Sigma_{j=1}^{N} w_j^\delta} w_i = \frac{\Sigma_{i=1}^{N} w_i^{\delta+1}}{\Sigma_{i=1}^{N} w_i^\delta}. \tag{5.5}$$

(Case 2) $v' \in V(P^{(1)})$. In this case, a new edge is added between two existing nodes in $P^{(1)}$. In this case, a new edge is added between two existing nodes in $P^{(1)}$. Consider the two nodes v and v' in $P^{(1)}$, and u and u' are their matches in an arbitrary match of $P^{(1)}$. We compute γ as:

$$\gamma = \mathbb{E}[d(u)d(u')\rho] = \rho\mathbb{E}[d(u)d(u')].$$

We still consider the neighbors of u and u' in the match. Suppose u and u' has δ and δ' neighbors respectively. Denote $\phi_{i,j}$ as the probability that u and u' appear as u_i and u_j in the match. Then:

$$\mathbb{E}[d(u)d(u')] = \sum_{i,j=1}^{N} \phi_{i,j}\, w_i\, w_j.$$

There are two cases. If u and u' have no common neighbor in the match, it is obvious that:

$$\phi_{i,j} = \phi_i \phi_j = \frac{w_i^{\delta}}{\sum_{s=1}^{N} w_s^{\delta}} \frac{w_j^{\delta'}}{\sum_{s=1}^{N} w_s^{\delta'}},$$

where ϕ_i and ϕ_j are computed according to Equation 5.4.

If their neighbors coincide, u and u' are not independent. Let $V_c = \{u_{c_1}, u_{c_2}, \ldots, u_{c_t}\}$ denote the common neighbors of u and u', $V_d = \{u_{d_1}, \ldots, u_{d_{\delta-t}}\}$ denote only u's neighbors and $V_d' = \{u_{d_1'}, \ldots, u_{d_{\delta'-t}'}\}$ denote only u''s neighbors. The structure formed by u, u' and their neighbors is called a *twin star*, and u and u' are the roots of the twin star. We have:

$$\begin{aligned}
\phi_{i,j} &= \mathsf{Pr}(u = u_i, u' = u_j \mid u, u' \text{ root a twin star}) \\
&= \frac{\mathsf{Pr}(u, u' \text{ root a twin star} \mid u = u_i, u' = u_j)\mathsf{Pr}(u = u_i, u' = u_j)}{\mathsf{Pr}(u, u' \text{ root a twin star})} \\
&= \frac{\mathsf{Pr}(\mathsf{E}_{i,j})}{\sum_{s,t=1}^{N} \mathsf{Pr}(\mathsf{E}_{s,t})},
\end{aligned}$$

where $\mathsf{E}_{i,j}$ denotes the event that u and u' root a twin star given $u = u_i$ and $u' = u_j$. Following the same idea in deriving Equation 5.5, we have:

$$\begin{aligned}
\mathsf{Pr}(\mathsf{E}_{i,j}) = &\sum_{V_c \subset V(G)} \sum_{V_d \subset V(G)} \sum_{V_d' \subset V(G)} \mathsf{Pr}_{i,c_1} \times \mathsf{Pr}_{i,c_2} \times \cdots \mathsf{Pr}_{i,c_t} \\
&\times \mathsf{Pr}_{i,d_1} \times \mathsf{Pr}_{i,d_2} \times \cdots \mathsf{Pr}_{i,d_{\delta-t}} \times \mathsf{Pr}_{j,c_1} \times \mathsf{Pr}_{j,c_2} \times \cdots \mathsf{Pr}_{j,c_t} \\
&\times \mathsf{Pr}_{j,d_1'} \times \mathsf{Pr}_{j,d_2'} \times \cdots \mathsf{Pr}_{j,d_{\delta'-t}'} \\
&= w_i^{\delta} w_j^{\delta'} \times \sum_{V_c \subset V(G)} \prod_{s=1}^{t} \rho^2 w_{c_s}^2 \times \sum_{V_d \subset V(G)} \prod_{s=1}^{\delta-t} \rho w_{d_s} \\
&\times \sum_{V_d' \subset V(G)} \prod_{s=1}^{\delta'-t} \rho w_{d_s'} = w_i^{\delta} w_j^{\delta'} \times C,
\end{aligned} \tag{5.6}$$

where

$$\begin{aligned}
C = &\sum_{V_c \subset V(G)} \prod_{s=1}^{t} \rho^2 w_{c_s}^2 \times \sum_{V_d \subset V(G)} \prod_{s=1}^{\delta-t} \rho w_{d_s} \\
&\times \sum_{V_d' \subset V(G)} \prod_{s=1}^{\delta'-t} \rho w_{d_s'}.
\end{aligned}$$

Note that C is a constant for any i, j. Therefore:

$$\begin{aligned}
\phi_{i,j} &= \frac{\mathsf{Pr}(\mathsf{E}_{i,j})}{\sum_{s,t=1}^{N} \mathsf{Pr}(\mathsf{E}_{s,t})} = \frac{w_i^{\delta} w_j^{\delta'}}{\sum_{s,t=1}^{N} w_s^{\delta} w_t^{\delta'}} \\
&= \frac{w_i^{\delta}}{\sum_{s=1}^{N} w_s^{\delta}} \frac{w_j^{\delta'}}{\sum_{s=1}^{N} w_s^{\delta'}}.
\end{aligned}$$

As a result of both cases, we have:

$$\begin{aligned}\gamma &= \rho \sum_{i,j=1}^{N} \phi_{i,j} w_i w_j \\ &= \rho \times \frac{\Sigma_{i=1}^{N} w_i^{\delta+1}}{\Sigma_{i=1}^{N} w_i^{\delta}} \times \frac{\Sigma_{j=1}^{N} w_j^{\delta'+1}}{\Sigma_{j=1}^{N} w_j^{\delta'}}.\end{aligned} \tag{5.7}$$

Given Equation 5.5 and Equation 5.7, we compute $|R_{\mathcal{G}}(P)|$ for any connected pattern graph P as follows. First, we run Depth-First-Search (DFS) over P to obtain the DFS-tree. Then, starting from an edge e with $|R_{\mathcal{G}}(e)| = 2M$, we apply Equation 5.5 iteratively to compute the size of the tree. Finally, we apply Equation 5.7 iteratively as we extend the non-tree edges. Note that, given a graph G, the γ calculated by Equation 5.5 or Equation 5.7 only depends on δ and δ', thus can be precomputed.

$\lvert R_{\mathcal{G}}(P)\rvert$	$\beta = 2.1$	$\beta = 2.3$	$\beta = 2.5$	$\beta = 2.7$	$\beta = 2.9$
$\lvert R_{\mathcal{G}}(P_2^{ld})\rvert$	67618.5	14632.5	1993.0	610.2	83.4
$\lvert R_{\mathcal{G}}(P_2^{b})\rvert$	230.2	69.5	16.3	6.2	1.5

Table 5.1: The number of the matches of P_2^{ld} and P_2^b in the PR graph (in billions).

Remark 4. *The plans $\mathcal{E}_1$ and $\mathcal{E}_2$ shown in Figure 3.1 are actually the optimal execution plans computed using the* ER *model and the* PR *model, respectively. Observe that $\mathcal{E}_1$ differs from the $\mathcal{E}_2$ in the second round where P_2^{ld} is processed instead of P_2^b. Generally, we have $|R(P_2^{ld})| < |R(P_2^b)|$ in the* ER *model [20], but $|R(P_2^{ld})| \gg |R(P_2^b)|$ in the* PR *model. As a result, $\mathcal{E}_1$ and $\mathcal{E}_2$ are returned as the optimal plan ragarding the* ER *model and* PR *model, respectively. Next we consider an* ER *graph $\Re$ and a* PR *graph $\mathcal{G}$ with $N = 1{,}000{,}000$ and $M = 25{,}000{,}000$, and compute $|R(P_2^{ld})|$ and $|R(P_2^b)|$ in both graphs for a comparison. According to [20], we have $|R_{\Re}(P_2^{ld})| = 0.78$, and $|R_{\Re}(P_2^b)| = 312$. Then we compute $|R_{\mathcal{G}}(P_2^{ld})|$ and $|R_{\mathcal{G}}(P_2^b)|$ using the proposed method, and show the results with various power-law exponents in Table 5.1. It is clear to see that $|R_{\mathcal{G}}(P_2^{ld})| \gg |R_{\mathcal{G}}(P_2^b)|$ in all cases. In Section 8, we further experimented using real-life graphs, which confirms that the* PR *model offers more realistic estimation and consequently renders better execution plan.*

6 Execution Plan

In this section, we introduce the algorithm that computes the optimal execution plan. Rather than following the left-deep join framework [20], we propose a dynamic-programming algorithm to compute the optimal bushy join plan. We further consider overlaps among the join units. To show the basic idea, we first introduce the non-overlapped case.

6.1 Non-overlapped Case

Definition 7. ***(Partial Execution)*** *A partial execution, denoted $\mathcal{E}_{P_\alpha}$, is an execution plan that computes the partial pattern $P_\alpha \subseteq P$.*

Given a partial pattern $P_\alpha \subseteq P$, the optimal partial execution plan of P_α satisfies:

$$\mathcal{C}(\mathcal{E}_{P_\alpha}) = \begin{cases} 0, \ P_\alpha \text{ is a join unit,} \\ \min\limits_{P_\alpha^l \subset P_\alpha} \{\mathcal{C}(\mathcal{E}_{P_\alpha^l}) + \mathcal{T}(P_\alpha^l) + \mathcal{C}(\mathcal{E}_{P_\alpha^r}) + \mathcal{T}(P_\alpha^r)\}, \text{ otherwise.} \end{cases} \tag{6.1}$$

where $P_\alpha^r = P_\alpha \setminus P_\alpha^l$, $\mathcal{T}(P_\alpha^l)$ and $\mathcal{T}(P_\alpha^r)$ are computed via Equation 5.1 or Equation 5.2 depending on whether they are join units or partial patterns. The optimal partial execution $\mathcal{E}_{P_\alpha}$ is obtained while minimizing the sum of the cost of the optimal $\mathcal{E}_{P_l'}$ and $\mathcal{E}_{P_r'}$, and the cost of processing the join $R(P_\alpha) = R(P_\alpha^l) \bowtie R(P_\alpha^r)$, that is $\mathcal{T}(P_\alpha^l) + \mathcal{T}(P_\alpha^r)$. Note that $\mathcal{C}(\mathcal{E}_{P_\alpha}) = 0$ if P_α is a join unit, as no join is needed to compute $R(P_\alpha)$.

We use a hash map $\mathcal{H}$ to maintain the so far best partial execution for each $P_\alpha \subseteq P$. The entry of the hash map for P_α has the form $(P_\alpha, \mathcal{T}, \mathcal{C}, P_\alpha^l, P_\alpha^r)$, where $\mathcal{T}$ is an auxiliary cost computed via either Equation 5.1 or Equation 5.2, $\mathcal{C}$ is the so far best cost $\mathcal{C}(E_{P_\alpha})$ while evaluating P_α, P_α^l is the left-join pattern when the current best cost is obtained, and P_α^r is the corresponding right-join pattern, where $P_\alpha^r = P_\alpha \setminus P_\alpha^l$, as no overlap is considered. The hash map is indexed by P_α and we can access one specific item I for a certain P_α via $\mathcal{H}_{P_\alpha}(I)$, where $I \in \{\mathcal{T}, \mathcal{C}, P_\alpha^l, P_\alpha^r\}$.

Algorithm 2: computeExecutionPlan(data graph G, pattern graph P)

Input : The data graph G and the pattern graph P
Output : The optimal execution plan w.r.t. P.

```
1  forall the P_α ⊆ P, s.t. P_α is connected do
2      H ← H ∪ (P_α, T(P_α), ∞, ∅, ∅);
3  for s = 1...m, where m = |E(P)| do
4      forall the P_α ⊂ P s.t. P_α is connected and |E(P_α)| = s do
5          if P_α is a join unit then
6              H_{P_α}(C) = 0;
7          else
8              forall the P_α^l ⊂ P_α s.t. P_α^l and P_α^r = P_α \ P_α^l are connected do
9                  C ← H_{P_α^l}(C) + H_{P_α^l}(T) + H_{P_α^r}(C) + H_{P_α^r}(T); if C < H_{P_α}(C)
                   then
10                     H_{P_α}(C) ← C;
11                     H_{P_α}(P_α^l) ← P_α^l; H_{P_α}(P_α^r) ← P_α^r;
12 E_o ← ComputeOptPlan(H, P);
13 return E_o;
```

The algorithm to compute the optimal execution plan is shown in Algorithm 2. In line 2, We initialize an entry in the hash map for each connected $P_\alpha \subseteq P$ that is potentially a partial pattern (line 2). Note that we precompute $\mathcal{T}(P_\alpha)$ for each P_α. To find the optimal execution plan for P, we need to accordingly find the optimal partial execution plans for all P's subgraphs, in non-decreasing order of their sizes. The algorithm performs three nested loops. The first loop in line 3 confines the size of the partial patterns to s, and the second loop enumerates all possible partial patterns with size s (line 4). If the current partial pattern P_α is a join unit, we simply set the corresponding cost to 0 (line 6). Otherwise, the third loop is triggered to update the optimal execution plan for P_α (line 8). We enumerate all P_α^l (and $P_\alpha^r = P_\alpha \setminus P_\alpha^l$), and for each P_α^l where P_α^l and P_α^r are both connected, we compute $\mathcal{C}(\mathcal{E}_{P_\alpha})$ via Equation 6.1 (line 9). In this way, we finally find the P_α^l to minimize $\mathcal{C}(\mathcal{E}_{P_\alpha})$, and update the entry of P_α by setting $\mathcal{H}_{P_\alpha}(\mathcal{C})$, $\mathcal{H}_{P_\alpha}(P_\alpha^l)$ and $\mathcal{H}_{P_\alpha}(P_\alpha^r)$ correspondingly (line 10-11). After all the entries in the hash map are computed, we first look up the entry for P to locate the P_α^l and P_α^r and repeat the procedure recursively on P_α^l and P_α^r until $P_\alpha^l = \emptyset$. In

this way, we compute the optimal execution plan (line 12).

Lemma 5. *The space complexity and time complexity of Algorithm 2 are $O(2^m)$ and $O(3^m)$, respectively.*

Proof. We first show the space complexity. Each entry in $\mathcal{H}$ is uniquely identified by the partial pattern P_α, and there are at most 2^m partial patterns, which consumes $O(2^m)$ space.

Next we prove the time complexity. There are at most $\binom{m}{s}$ partial patterns of P sized s, which trigger $\binom{m}{s}$ calls over the second loop. For each partial pattern P_α of size s, we enumerate all possible connected subgraphs of it as P_α^l, which incurs at most 2^s calls. By pre-computing any connected partial patterns of P (which use $O(2^m)$ space and time), we can verify the connectedness of P_α^l and P_α^r in $O(1)$ time. Moreover, updating the entry has the cost $O(1)$. Note that $\sum_{s=1}^{m}\binom{m}{s}\cdot 2^s = (1+2)^m = 3^m$. Therefore, the time complexity of Algorithm 2 is $O(3^m)$. □

Discussion. In practice, the processing time for Algorithm 2 is much smaller than $O(3^m)$ since we require that all partial patterns are connected.

6.2 Overlapped Case

The following lemma inspires us to consider overlaps among the join units.

Lemma 6. *Given a pattern graph P, and another pattern graph P^+, where $P^+ = P \cup \{(v, v')\}, v, v' \in V(P)$ and $(v, v') \notin E(P)$, we have:*

$$|R(P^+)| \leq |R(P)|.$$

Proof. Apparently, $\forall f \in R(P^+), f \in R(P)$. Therefore, $|R(P^+)| \leq |R(P)|$. □

Example 3. *We have actually shown overlaps among the join units in Figure 3.1. For example, we have $E(p_0) \cap E(p_1) = \{(v_1, v_3)\}$ in the bushy tree $\mathcal{E}_2$. Let $p_1^- = p_1 \setminus (v_1, v_3)$. In the non-overlapped case, we will execute $R(P_1') = R(p_0) \bowtie R(p_1^-)$ instead. Clearly, $|R(p_1)| \leq |R(p_1^-)|$ according to Lemma 6, and hence the plan with overlaps is better.* □

A naive solution to allow the join units to overlap and still guarantee the optimality in Algorithm 2 is: when we evaluate $P_\alpha^r = P_\alpha \setminus P_\alpha^l$ in line 8, we further enumerate all possible P_α^{r*}, where P_α^{r*} are all *connected* structures formed by adding any subset of $E(P_\alpha^l)$ to P_α^r. As a result, the time complexity is of the order:

$$\sum_{s=1}^{m}\binom{m}{s}\cdot\sum_{t=1}^{s}\binom{s}{t}2^t = 4^m.$$

All or Nothing. The time complexity of computing the optimal execution in the overlapped case can be reduced to $O(3^m)$ with some practical relaxation. Given a partial pattern P_α, and its left-join (resp. right-join) pattern P_α^l (resp. P_α^r) (P_l' and P_α^r may overlap), we define the redundant node as:

Definition 8. ***(Redundant Node)*** *A node $v_r \in V(P_\alpha^l) \cap V(P_\alpha^r)$ is a redundant node w.r.t. $P_\alpha = P_\alpha^l \cup P_\alpha^r$, if $P_\alpha = (P_\alpha^l \setminus v_r) \cup P_\alpha^r$ or $P_\alpha = P_\alpha^l \cup (P_\alpha^r \setminus v_r)$.*

In other words, the removal of a redundant node from either P_α^l or P_α^r does not affect the join results. Denote V_r as a set of redundant nodes w.r.t. $P_\alpha = P_\alpha^l \cup P_\alpha^r$. We further define the cut nodes V_c and the cut edges E_c as follows:

$$V_c(P_\alpha^l, P_\alpha^r) = \big(V(P_\alpha^l) \cap V(P_\alpha^r)\big) \setminus V_r$$
$$E_c(P_\alpha^l, P_\alpha^r) = \{(v, v') \mid (v, v') \in E(P_\alpha) \wedge v, v' \in V_c(P_\alpha^l, P_\alpha^r)\}.$$

Example 4. *In Figure 6.1, we show a partial pattern P_α and its left-join (resp. right-join) pattern P_α^l (resp. P_α^r). Clearly, v_4 is a redundant node since $P_\alpha = P_\alpha^l \cup (P_\alpha^r \setminus v_4)$, and we have $V_c(P_\alpha^l, P_\alpha^r) = \{v_2, v_3\}$ and $E_c(P_\alpha^l, P_\alpha^r) = \{(v_2, v_3)\}$.* □

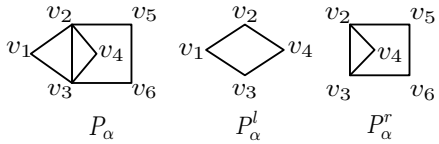


Figure 6.1: The redundant node, cut nodes and cut edges.

Based on the cut edges, we introduce an *all-or-nothing* strategy, which reduces the time complexity to $O(3^m)$. Specifically, when we evaluate $P_\alpha^r = P_\alpha \setminus P_\alpha^l$ in Algorithm 2 (line 8), instead of enumerating P_α^{r*} by considering all subsets of $E(P_\alpha^l)$ in the naive solution, we only consider adding all the cut edges w.r.t. $P_\alpha = P_\alpha^l \cup P_\alpha^r$, or none of them. We show that the all-or-nothing strategy returns an execution plan that is almost as good as the naive solution.

Denote $\mathcal{E}_o = \{D_o, J_o\}$ as the optimal execution plan obtained by the naive solution and $\mathcal{E}_o'$ as the best execution plan obtained by the all-or-nothing strategy. Suppose in the i[-th] round of the execution plan $\mathcal{E}_o$, the following join is processed:

$$R(P_i) = R(P_i^l) \bowtie R(P_i^r), \forall 1 \le i \le |D_o| - 1.$$

We then construct an intermediate execution plan $\widetilde{\mathcal{E}}$ by replacing each of the above join as $R(P_i) = R(P_i^l) \bowtie R(\widetilde{P_i^r})$, where $\widetilde{P_i^r} = P_i^r \cup \{e_1, e_2, \ldots, e_s\}$, and each $e_i \in E_c(P_i^l, P_i^r) \wedge e_i \notin E(P_i^r)$. In other words, the alternative right-join pattern $\widetilde{P_i^r}$ is obtained by adding all the cut edges to P_i^r. It is trivial when $\widetilde{P_i^r} = P_i^r$. Otherwise, we first generate $R(\widetilde{P_i^r})$ by performing the joins $R(P_i^r) \bowtie R(e_1) \bowtie \cdots \bowtie R(e_r)$ sequentially, and each join handles a cut edge. Then we execute the join $R(P_i) = R(P_i^l) \bowtie R(\widetilde{P_i^r})$.

Leveraging the intermediate execution plan $\widetilde{\mathcal{E}}$, we prove that $\mathcal{E}_o'$ (the best execution plan computed by "all-or-nothing" strategy) has the cost of the same order as $\mathcal{E}_o$ (the optimal solution). We first prove $\mathcal{C}(\widetilde{\mathcal{E}}) = \Theta(\mathcal{C}(\mathcal{E}_o))$.

Lemma 7. *If $\mathcal{C}(\mathcal{E}_o) \ge \Theta(M)$, then $\mathcal{C}(\widetilde{\mathcal{E}}) = \Theta(\mathcal{C}(\mathcal{E}_o))$.*

Proof. We divide $\widetilde{\mathcal{E}}$ into two parts. For $1 \le i \le t$, the first part, denoted as $\widetilde{\mathcal{E}_1}$, performs the join $R(P_i) = R(P_i^l) \bowtie R(\widetilde{P_i^r})$; the second part, denoted as $\widetilde{\mathcal{E}_2}$, handles the generation of each $R(\widetilde{P_i^r})$ by sequentially joining the matches of each cut edge to $R(P_i^r)$. According to Lemma 6, we have:

$$|R(\widetilde{P_i^r})| \le |R(P_i^r)|. \tag{6.2}$$

Clearly, $\mathcal{C}(\widetilde{\mathcal{E}}) = \mathcal{C}(\widetilde{\mathcal{E}_1}) + \mathcal{C}(\widetilde{\mathcal{E}_2})$. We accordingly divide the proof into two parts.

(**Part 1**) We prove $\mathcal{C}(\widetilde{\mathcal{E}}_1) \leq \Theta(\mathcal{C}(\mathcal{E}_o))$. Denote $\mathcal{E}_{P_i}$ and $\widetilde{\mathcal{E}}_{P_i}$ as the partial execution of generating $R(P_i)$ in $\mathcal{E}_o$ and $\widetilde{\mathcal{E}}_1$. According to Equation 6.1, we have:

$$\mathcal{C}(\mathcal{E}_{P_i}) = \mathcal{C}(\mathcal{E}_{P_i^l}) + \mathcal{T}(P_i^l) + \mathcal{C}(\mathcal{E}_{P_i^r}) + \mathcal{T}(P_i^r),$$
$$\mathcal{C}(\widetilde{\mathcal{E}}_{P_i}) = \mathcal{C}(\widetilde{\mathcal{E}}_{P_i^l}) + \mathcal{T}(P_i^l) + \mathcal{C}(\widetilde{\mathcal{E}}_{P_i^r}) + \mathcal{T}(\widetilde{P_i^r}),$$

where $\mathcal{T}(P_\alpha)$ are computed via Equation 5.1 or Equation 5.2 depending on whether P_α is a join unit or a partial pattern.

Let $t = |D_o| - 1$ denote the number of rounds of $\mathcal{E}_o$. We prove Part 1 by inducing on $i = 1, 2, \ldots, t$.

When $i = 1$, P_1^l and P_1^r must be the join units, thus $\mathcal{C}(\mathcal{E}_{P_i^l}) = \mathcal{C}(\widetilde{\mathcal{E}}_{P_i^l}) = \mathcal{C}(\mathcal{E}_{P_i^r}) = \mathcal{C}(\widetilde{\mathcal{E}}_{P_i^r}) = 0$. Further, we have $\mathcal{T}(\widetilde{P_1^r}) \leq \mathcal{T}(P_1^r)$ given that $|R(\widetilde{P_1^r})| \leq |R(P_1^r)|$ by Equation 6.2. Hence, $\mathcal{C}(\widetilde{\mathcal{E}}_{P_1}) \leq \Theta(\mathcal{C}(\mathcal{E}_{P_1}))$.

Assume that $\mathcal{C}(\widetilde{\mathcal{E}}_{P_i}) \leq \Theta(\mathcal{C}(\mathcal{E}_{P_i}))$ holds for all $1 < i \leq s-1, s < t$. Consider $i = s$. Note that P_s^l and P_s^r are some P_j with $j < i$, we hence have $\mathcal{C}(\widetilde{\mathcal{E}}_{P_s^l}) \leq \Theta(\mathcal{E}_{P_s^l})$ and $\mathcal{C}(\widetilde{\mathcal{E}}_{P_s^r}) \leq \Theta(\mathcal{E}_{P_s^r})$ by the assumption. Additionally, $\mathcal{T}(\widetilde{P_s^r}) \leq \mathcal{T}(P_s^r)$ by Equation 6.2. It is immediate that $\mathcal{C}(\widetilde{\mathcal{E}}_{P_s}) \leq \Theta(\mathcal{C}(\mathcal{E}_{P_s}))$. By induction, we have:

$$\mathcal{C}(\widetilde{\mathcal{E}}_1)(= \mathcal{C}(\widetilde{\mathcal{E}}_{P_t})) \leq \Theta(\mathcal{C}_{\mathcal{E}_o})(= \Theta(\mathcal{C}(\mathcal{E}_{P_t}))).$$

(**Part 2**) We prove $\mathcal{C}(\widetilde{\mathcal{E}}_2) \leq \Theta(\mathcal{C}(\mathcal{E}_0))$. In this part, we will generate each $R(\widetilde{P_i^r})$ by joining the matches of the cut edges $\{e_1, e_2, \ldots, e_s\}$ with $R(P_i^r)$. We suppose at least one cut edge is processed as otherwise it is trivial. Denote $\widetilde{P_i^r}[x]$ as $P_i^r \cup \{e_1, e_2, \ldots, e_x\}$. As each e_i is a cut edge for P_i^r, we have $R(\widetilde{P_i^r}[x]) \leq R(P_i^r)$ according to Equation 6.2.

Denote $\mathcal{C}(\widetilde{\mathcal{E}}_{\Delta\widetilde{P_i^r}})$ as the cost to generate $R(\widetilde{P_i^r})$ in the i[-th] round. We have:

$$C(\widetilde{\mathcal{E}}_2) = \sum_{i=1}^{t} \mathcal{C}(\widetilde{\mathcal{E}}_{\Delta\widetilde{P_i^r}}).$$

If $\widetilde{P_i^r} = P_i^r$, the case is trivial as $\mathcal{C}(\widetilde{\mathcal{E}}_{\Delta\widetilde{P_i^r}}) = 0$. Otherwise, denote $\widetilde{P_i^r}[x]$ as $P_i^r \cup \{e_1, e_2, \ldots, e_x\}$. Specifically, let $\widetilde{P_i^r}[0] = P_i^r$. According to Equation 5.3, we have:

$$\mathcal{C}(\widetilde{\mathcal{E}}_{\Delta\widetilde{P_i^r}}) = \sum_{j=1}^{s} \mathcal{T}(e_j) + \sum_{x=0}^{s} \mathcal{T}(\widetilde{P_i^r}[x]).$$

Note that the number of cut edges is often small, and we treat s as a constant. In this sense, $\sum_{j=1}^{s} \mathcal{T}(e_j) = \Theta(M)$ and $\sum_{x=0}^{s} \mathcal{T}(\widetilde{P_i^r}[x]) = \Theta(\mathcal{T}(P_i^r))$, as $\mathcal{T}(\widetilde{P_i^r}[x]) \leq \Theta(\mathcal{T}(P_i^r))$ for each x given that $|R(\widetilde{P_i^r}[x])| \leq |R(P_i^r)|$ by Equation 6.2. Therefore:

$$C(\widetilde{\mathcal{E}}_2) = \Theta(M) + \sum_{i=0}^{t} \Theta(\mathcal{T}(P_{i_r})) \leq \Theta(\mathcal{C}(\mathcal{E}_o)).$$

According to Part 1 and Part 2, $\mathcal{C}(\widetilde{\mathcal{E}}) \leq \Theta(\mathcal{C}(\mathcal{E}_o))$, and apparently, $\mathcal{C}(\widetilde{\mathcal{E}}) \geq \mathcal{C}(\mathcal{E}_o)$. Therefore, $\mathcal{C}(\widetilde{\mathcal{E}}) = \Theta(\mathcal{C}(\mathcal{E}_o))$.

□

We then show $\mathcal{C}(\mathcal{E}'_o) \leq \mathcal{C}(\widetilde{\mathcal{E}})$ under some practical relaxations.

Corollary 2. *Given a pattern graph P, and any P^l and P^r, such that $P = P^l \cup P^r$, we have $(P \setminus P^l) \subseteq P^r$.*

Proof. This is apparently true as $(P \setminus P^l)$ is the smallest P^r that satisfies $P = P^l \cup P^r$. □

Corollary 3. *Given a pattern graph P, and any left-join (resp. right-join) pattern P^l (resp. P^r), such that $P = P^l \cup P^r$, if there is no redundant node w.r.t. $P = P^l \cup P^r$, then we have $V(P^r) = V(P \setminus P^l)$.*

Proof. It is obvious that $V(P \setminus P^l) \subseteq V(P^r)$ by Corollary 2.

We then show $V(P^r) \subseteq V(P \setminus P^l)$. For $\forall v \in V(P^r)$, we claim that $v \in V(P \setminus P^l)$. There are two cases. If $v \notin V(P^l)$, it is immediate that the claim is true. Otherwise, assume that $v \notin V(P \setminus P^l)$. Then $(P \setminus P^l) \subseteq P^r \setminus \{v\}$. Therefore, we must have $P = P^l \cup (P^r \setminus \{v\})$. In other words, v is a redundant node w.r.t. $P = P^l \cup P^r$. This draws a contradiction.

Based on the above cases, the corollary holds. □

Lemma 8. *If there is no redundant node w.r.t. $P_i = P_i^l \cup P_i^r$ for all $1 \leq i \leq |D_o| - 1$ in $\mathcal{E}_o = (D_o, J_o)$, then $\mathcal{C}(\mathcal{E}'_o) \leq \mathcal{C}(\widetilde{\mathcal{E}})$.*

Proof. Given $P_i = P_i^l \cup P_i^r$ in $\mathcal{E}_o$, and $\widetilde{P_i^r} = P_i^r \cup E_c(P_i^l, P_i^r)$, we further denote $\widetilde{P_i^{r*}} = (P_i \setminus P_i^l) \cup E_c(P_i^l, (P_i \setminus P_i^l))$.

We claim that $\widetilde{\mathcal{E}}$ must be within the searching space of the all-or-nothing strategy. It suffices to show that the join $R(P_i) = R(P_i^l) \bowtie R(\widetilde{P_i^r})$ will be evaluated in the all-or-nothing strategy. Recall the process of the all-or-nothing strategy. When we have P_i^l, we will consider $\widetilde{P_i^{r*}}$ as the right-join pattern via the "all" strategy. In this sense, we simply prove the claim by showing that:

$$\widetilde{P_i^r} = \widetilde{P_i^{r*}}.$$

Since there is no redundant node w.r.t. $P_i = P_i^l \cup P_i^r$, according to Corollary 3. we have:

$$V(P_i^r) = V(P_i \setminus P_i^l).$$

Therefore, $V(\widetilde{P_i^{r*}}) = V(\widetilde{P_i^r})$.

We first show $\widetilde{P_i^r} \subseteq \widetilde{P_i^{r*}}$. $\forall e \in E(\widetilde{P_i^r})$, we have two cases: (1) if $e \in P_i^r$ and $e \notin E_c(P_i^l, P_i^r)$, then $e \notin P_i^l$. It is immediate that $e \in P_i \setminus P_i^l$. Hence, $e \in \widetilde{P_i^{r*}}$; (2) If $e \in E_c(P_i^l, P_i^r)$, let $e = (v_1, v_2), v_1, v_2 \in V(P')$. This means $v_1, v_2 \in V_c(P_i^l, P_i^r) = V(P_i^l) \cap V(P_i^r)$. On the other way, $V(P_i^l) \cap V(P_i^r) = V(P_i^l) \cap V(P' \setminus P_i^l) = V_c(P_i^l, P' \setminus P_i^l)$, which suggests $e = (v_1, v_2) \in E_c(P_i^l, P' \setminus P_i^l)$. Therefore, $e \in E(\widetilde{P_i^{r*}})$. With both cases, we have $\widetilde{P_i^r} \subseteq \widetilde{P_i^{r*}}$.

Then we show $\widetilde{P_i^{r*}} \subseteq \widetilde{P_i^r}$. $\forall e \in E(\widetilde{P_i^{r*}})$, it is obvious that $e \in P_i \setminus P_i^l$. By Corollary 2, we have $P_i \setminus P_i^l \subseteq P_i^r$, which suggests $e \in E(\widetilde{P_i^r})$. Thus, it holds that $\widetilde{P_i^{r*}} \subseteq \widetilde{P_i^r}$.

In conclusion, we have $\widetilde{P_i^r} = \widetilde{P_i^{r*}}$. This implies that $\widetilde{\mathcal{E}}$ must be within the searching space of the "all-or-nothing" strategy. While $\mathcal{E}'_o$ is the optimal solution in the space, it is immediate that $\mathcal{C}(\mathcal{E}'_o) \leq \mathcal{C}(\widetilde{\mathcal{E}})$. □

Theorem 1. *If $\mathcal{C}(\mathcal{E}_o) \geq \Theta(M)$ and there is no redundant node w.r.t. $P_i = P_i^l \cup P_i^r$ for all $1 \leq i \leq |D_o| - 1$ in $\mathcal{E}_o = (D_o, J_o)$, then $\mathcal{C}(\mathcal{E}'_o) = \Theta(\mathcal{C}(\mathcal{E}_o))$*

Proof. With Lemma 7 and Lemma 8, Theorem 1 holds. □

Discussion. We show that the two conditions in Theorem 1 are practically reasonable. First, $\mathcal{C}(\mathcal{E}_o) \geq \Theta(M)$. Actually, the cost of the execution is often far larger than the size of the data graph. Second, no redundant node is involved. In practice, the involvements of redundant nodes usually result in more iterations, while the gain of such redundancies is rather limited.

7 Clique Compression

To start this section, let us consider a motivating example.

Example 5. *We find a large clique with 943 nodes in the uk dataset used in our experiment in Table 8.2, which alone contributes to* $\binom{943}{5} \approx 6 \times 10^{12}$ *matches for a* 5*-clique, and causes huge burden on storage and communication. Alternatively, we can encode all these matches using the nodes of the large clique itself, and this costs linear space to the number of nodes in the clique.* □

This example motivates us to consider clique compression, aiming at reducing the cost of transferring and maintaining the intermediate results. In order to do so, we compute a set of non-overlapping cliques in the data graph G as a preprocessing step. In query processing, when p_k is considered as a join unit, instead of computing all the matches of p_k directly, we represent the matches in a compressed way, and we also try to maintain the compressed matches in further joins. In the following, we first show how to precompute the non-overlapping cliques, followed by discussing the way of compressing the matches of p_k. Finally, we introduce how to process joins with the compressed results.

Algorithm 3: Clique-Search(data graph G)

Input : G, the data graph.
Output : A set of non-overlapping cliques.

```
1 G' ← G;  S ← ∅;
2 while G' is not empty do
3     u ← The node with largest degree in G';
4     K ← A maximal clique containing u in G';
5     if |V(K)| > thresh then
6         S = S ∪ {K};
7     G' ← G' \ K;
8 return S;
```

Clique Precomputation. As a preprocessing step, we compute a set of non-overlapping (by nodes) cliques $S = \{K_1, K_2, \ldots, K_s\}$ in the data graph G. We show the greedy algorithm to compute S in Algorithm 3. Each time we select a node u with the largest degree from G, compute a maximal clique containing u in G, add the clique into S if its size is larger than a threshold (e.g., 50) and remove it from G. We repeat the process until all nodes are removed from G. After computing S, we index all the cliques on each machine in the cluster (e.g. using "Distributed Cache" in MapReduce [37]). Specifically, we maintain a map $\mathcal{M}$ in each machine, so that we can use $\mathcal{M}(u)$ to determine the clique that a node u ($u \in V(G)$) belongs to in constant time. Let $\mathcal{M}(u) = \emptyset$ if u does not belong to any clique in S. The space used to index the cliques

is small since we only need to index the nodes in each clique. We show in the experiment that the overhead of clique precomputation is relatively small, and it contributes to improving the performance of SEED, especially when the data graph contains some large cliques.

Online Clique Compression. During query processing, suppose a k-clique p_k is involved in the join, where $V(p_k) = \{v_1^\kappa, v_2^\kappa, \ldots, v_k^\kappa\}$ and $v_1^\kappa < v_2^\kappa < \cdots < v_k^\kappa$ (for symmetry breaking (Remark 1)). We compress the matches of p_k as follows. In each local graph G_u^2, we divide the nodes in $V(G_u^2)$ into two parts, namely, the clique nodes V_u^c and the non-clique nodes V_u^n. Here $V_u^c = \{u' | u' \in V(G_u^2) \backslash \{u\}, \mathcal{M}(u') = \mathcal{M}(u)\}$ is the set of nodes in G_u^2 that belong to the same clique as u in S, and $V_u^n = V(G_u^2) \backslash V_u^c$. Note that we have $u \in V_u^n$ for the ease of presentation. Specifically, when $\mathcal{M}(u) = \emptyset$, we have $V_u^c = \emptyset$ and $V_u^n = V(G_u)$. The nodes in both set are rearranged via the data node orders (Definiton 2). With the two different types of nodes, a compressed match, which represents multiple matches of the k-clique, is denoted as (f^c, f^n), where $f^c = (f^c.V, f^c.U) = (\{v_1^c, v_2^c, \ldots, v_s^c\}, \{u_1^c, u_2^c, \ldots, u_t^c\})$ is the compressed part of the match and $f^n = (f^n.V, f^n.U) = (\{v_1^n, v_2^n, \ldots, v_{k-s}^n\}, \{u_1^n, u_2^n, \ldots, u_{k-s}^n\})$ is the non-compressed part. We also regard f^n as a *partial match*, where $f^n(v) = u$ for each $v \in f^n.V$ and $u \in f^n.U$. Here, the following five constraints must be satisfied:

- C_1: $u \in f^n.U$.
- C_2: $f^n.U \subseteq V_u^n$ and the nodes in $f^n.U$ must form a clique in G_u^2.
- C_3: $f^c.U \subseteq V_u^c$ and every node in $f^c.U$ is adjacent to all nodes in $f^n.U$ in G_u^2.
- C_4: $|f^c.V| \leq |f^c.U|$.
- C_5: $f^c.V \cup f^n.V = V(p_k)$.

In this way, a compressed match (f^c, f^n) represents $\binom{t}{s}$ matches of a k-clique, that is, the $k - s$ nodes $f^n.U = \{u_1^n, u_2^n, \ldots, u_{k-s}^n\}$ and every combination of s nodes in $f^c.U = \{u_1^c, u_2^c, \ldots, u_t^c\}$ recover a match. Note that C_1 restricts that u must be in the match, which is applied to avoid duplicates. For example, consider a 5-clique $\{u_1, u_2, u_3, u_4, u_5\}$ as the data graph, and a 4-clique p_4 as the pattern graph. Without C_1, the match (u_2, u_3, u_4, u_5) will be computed twice in both G_{u_1} and G_{u_2}. However, this match will be removed from G_{u_1} by C_1 as u_1 does not appear in the match. Considering the Order-Preservation constraint (Remark 1), we actually match the smallest node in p_k to u (note that u is the smallest node in G_u). Thus, we accordingly rewrite C_1 as $f^n(v_1^\kappa) = u$.

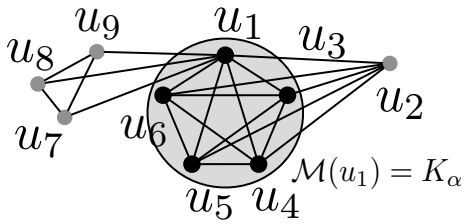


Figure 7.1: The local graph of u_1, and clique compression.

Example 6. *In Figure 7.1, we show the local graph G_{u_1}. Note that all nodes except u_1 have neighbors not presented in G_{u_1} and these nodes are already arranged by their orders (Definiton 2) in the data graph. The shadowed circle highlights a 5-clique K_α that u_1 belongs to. Observe that u_2 forms a larger clique with K_α but it does not*

belong to it. This can happen when we assign u_2 to the other larger clique. Thus, we have the clique nodes $V^c_{u_1} = \{u_3, u_4, u_5, u_6\}$, and the non-clique nodes $V^n_{u_1} = \{u_1, u_2, u_7, u_8, u_9\}$. Hereunder, we show the compressed matches of the 4-clique p_4 in G_{u_1}:

#	$f^c.V$	$f^c.U$	$f^n.V$	$f^n.U$	# matches
cm_1	$\{v_2^\kappa, v_3^\kappa, v_4^\kappa\}$	$\{u_3, u_4, u_5, u_6\}$	$\{v_1^\kappa\}$	$\{u_1\}$	4
cm_2	$\{v_3^\kappa, v_4^\kappa\}$	$\{u_3, u_4, u_5, u_6\}$	$\{v_1^\kappa, v_2^\kappa\}$	$\{u_1, u_2\}$	6
cm_3	$\emptyset$	$\emptyset$	$\{v_1^\kappa, v_2^\kappa, v_3^\kappa, v_4^\kappa\}$	$\{u_1, u_7, u_8, u_9\}$	1

Considering the Order-Preservation constraint (Remark 1), the data node sequence that matches p_4 must be arranged in the increasing order. In cm_1, u_1 together with each 3-combination of $f^c.U$ (increasing order) recover a match of the 4-clique, and thus cm_1 compresses $\binom{4}{3} = 4$ results. Similarly, cm_2 compresses $\binom{4}{2} = 6$ results. Additionally, cm_3 corresponds to a non-compressed match. As a whole, we use 3 compressed matches to represent 11 results. □

Algorithm 4: compressedClique(p_k, G_u^2)

Input : p_k, the k-clique, where $V(p_k) = \{v_1^\kappa, v_2^\kappa, \ldots, v_k^\kappa\}$,
G_u^2, the local graph of a certain $u \in V(G)$.
Output : A set of compressed matches of p_k.

$\mathcal{F} \leftarrow \emptyset$;
$V_u^c \leftarrow \{u' | u' \in V(G_u) \setminus \{u\} \wedge \mathcal{M}(u') = \mathcal{M}(u)\}$;
$V_u^n \leftarrow V(G_u) \setminus V_u^c$;
if $|V_u| \geq k-1$ **then** $\mathcal{F} \leftarrow \mathcal{F} \cup (f^c = (V(p_k) \setminus \{v_1^\kappa\}, V_u^c), f^n = (\{v_1^\kappa\}, \{u\}))$;
compressedCliqueRec($p_k,\ V_u^c,\ V_u^n \setminus \{u\},\ \mathcal{F},\ 1,\ \{u\}$)
return $\mathcal{F}$;
function compressedCliqueRec($p_k,\ V_u^c,\ V_u^n,\ \mathcal{F},\ i,\ U'$)
foreach $j \in [i, |V_u^n|]$ **do**
  $u' \leftarrow V_u^n[j]$;
  if u' forms a clique with U' in G_u **then**
    $U' \leftarrow U' \cup \{u'\}$;
    Initialize the compressed match ($f^c = \{\emptyset, \emptyset\}, f^n = \{\emptyset, \emptyset\}$);
    if ($|U'| < k$) **then**
      $f^c.U \leftarrow CCN(U')$;
    if $|U'| + |f^c.U| \geq k$ **then**
      foreach $\{v_1^n = v_1^\kappa, v_2^n, \ldots, v_{k-s}^n\} \subseteq V(p_k)$ s.t. $v_1^n < v_2^n < \cdots v_{k-s}^n$, where $|U'| = k - s$ **do**
        $f^n.V \leftarrow \{v_1^n, v_2^n, \ldots, v_{k-s}^n\}$;
        $f^c.V \leftarrow V(p_k) \setminus f^n.V$;
        $f^n.U \leftarrow U'$;
        $\mathcal{F} \leftarrow \mathcal{F} \cup (f^c = (f^c.V, f^c.U), f^n = (f^n.V, f^n.U))$;
      if $|U'| < k \wedge j \neq |V_u^n|$ **then**
        compressedCliqueRec($f^c.U,\ V_u^n \setminus U',\ \mathcal{F}, j+1,\ U'$);

The algorithm to compute all compressed k-cliques in a certain G_u^2 is shown in Algorithm 4. It is worth noting that Algorithm 4 can handle the non-compressed (un-optimized) case by letting $\mathcal{M}(u) = \emptyset$ for each $u \in V(G)$. Before moving forward to the algorithm, we define the *clique neighbors* and *common clique neighbors* as follows:

Given $u' \in V_u^n$, the *clique neighbors* of u', denoted as $CN(u')$, are the nodes in V_u^c that are adjacent to u' in G_u, that is, $CN(u') = \{u''|u'' \in V_u^c \wedge (u', u'') \in E(G_u)\}$, and the *common clique neighbors* of a set of data nodes U', denoted as $CCN(U')$, are the common clique neighbors of all nodes in U', that is $CCN(U') = \bigcap_{u' \in U'} CN(u')$.

In Algorithm 4, we first assign the clique nodes V_u^c and the non-clique nodes V_u^n in line 2-3, and we use $V_u^n[i]$ to denote the i[-th] (start from 1) node in V_u^n. In line 4, we report $(f^c = (V(p_k) \setminus \{v_1^\kappa\}, V_u^c), f^n = (\{v_1^\kappa\}, \{u\}))$ as a fully compressed results. Clearly, u and each $k-1$ combination of V_u^c recover a match of k-clique. We then call the recursive function compressedCliqueRec to further generate the compressed matches (line 5). In the recursive function, we use U' to record a set of non-clique nodes that form a clique in G_u, which are exactly the nodes that will be in $f^n.U$. The algorithm then proceeds by checking the four constraints for a compressed match. For each non-clique nodes that have not been visited (line 8), we verify that if it forms a larger clique with U' (line 10) to satisfy C_2. The qualified node is then involved in U' (line 11). If $|U'| < k$, we let $f^c.U = CCN(U')$ (line 14) so that the C_3 is satisfied. Otherwise, U' must have included all nodes for a full-matched subgraph, and we simply leave $f^c.U = \emptyset$. The condition $|U'| + |f^c.U| \geq k$ in line 15 guarantees C_4 and C_5 are satisfied, and once satisfied, we follow the procedure from line 16-20 to generate the compressed matches. For each $k-s$ nodes from $V(p_k)$ that involves v_1 (line 16), we assign them to $f^n.V$ (line 17), and then we obtain a partial match $f^n = (f^n.V, f^n.U = U')$. Thus, we construct a compressed match $(f^c = (f^c.V, f^c.U), f^n = (f^n.V, f^n.U))$ (line 20). We recursively call compressedCliqueRec (line 22) as $|U'| < k$.

Example 7. *In Example 6, we show the compressed matches cm_1, cm_2 and cm_3 of p_4 in $G_{u_1}^2$. According to Algorithm 4, cm_1 is computed in line 4. Now that we have $V_u^n = \{u_2, u_7, u_8, u_9\}$ (u_1 is excluded). In the first recursive call, we have $U' = \{u_1, u_2\}$ after adding u_2 (line 11), and compute $CCN(U') = \{u_3, u_4, u_5, u_6\}$. We then select a node from $\{v_2^\kappa, v_3^\kappa, v_4^\kappa\}$ to match u_2, and we obtain cm_2 consequently (line 16). After processing u_2, we move to the next node in V_u^n , that is u_7, and the recursive function will return cm_3 as a non-compressed match. (line 13).* □

Online Join Processing. With the clique compression technique, we follow the framework in Algorithm 1 to process joins, but replace each match f in Algorithm 1 as a compressed match (f^c, f^n). We correspondingly revise the $mymap$ and $myreduce$ functions to handle the compressed match. Note that here we generalize the concept of "compressed match", which not only represents a compressed match of a k-clique, but also the compressed join results produced in each round (Details are in Algorithm 6). For a non-compressed match, we simply let $f^c.V = f^c.U = \emptyset$. The main challenge is that, when a compressed match (f^c, f^n) is involved in a join, we do not need to immediately recover all matches from (f^c, f^n). Instead we try to maintain its compressed part f^c as much as possible. We call this process partial expansion. Given a compressed match (f^c, f^n), suppose it is involved in a join with join attributes V_{join}, the process of partial expansion is shown in lines 12-20 in Algorithm 5. We first compute the non-clique join attributes V_{join}^n and its corresponding match U_{join}^n (lines 13-14). Then we compute V_{join}^c - the set of join attributes that need to be expanded in the clique part f^c (line 15). Line 16 enumerates all matches U_{join}^c of V_{join}^c in f^c. For each U_{join}^c, we output a key-value pair (line 20) where the key is computed as $U_{join}^c \cup U_{join}^n$ (line 17) and the value is a compressed match (f_{out}^c, f_{out}^n) by moving the original match of V_{join}^c from f^c to f^n (line 18-19). The revised map^i procedure is shown in Algorithm 5 to

Algorithm 5: map^i(**key**: $\emptyset$; **value**: either compressed matches $(f^c, f^n) \in R(P'_j)$ and $(h^c, h^n) \in R(P'_s)$ for some $j < i, s < i$ or $G_u \in \Phi(G)$)

$V_{join} \leftarrow V(P'_j) \cap V(P'_s)$;
if P'_j is a star **then** genJoinUnit(P'_j, G_u, V_{join});
else if P'_j is a clique **then** genCompressedClique(P'_j, G_u, V_{join});
else partialExpansion(f^c, f^n, V_{join});
if P'_s is a star **then** genJoinUnit(P'_s, G_u, V_{join});
else if P'_s is a clique **then** genCompressedClique(P'_s, G_u, V_{join});
else partialExpansion(h^c, h^n, V_{join});
function genCompressedClique(p_k, G_u, V_{join})
$\mathcal{F} \leftarrow$ compressedClique(p_k, G_u);
foreach $(f^c, f^n) \in \mathcal{F}$ **do**
    partialExpansion(f^c, f^n, V_{join});
function partialExpansion(f^c, f^n, V_{join})
$V^n_{join} = \{v^n_1, v^n_2, \ldots, v^n_p\} \leftarrow f^n.V \cap V_{join}$;
$U^n_{join} \leftarrow \{f^n(v^n_1), f^n(v^n_2), \ldots, f^n(v^n_p)\}$;
$V^c_{join} \leftarrow f^c.V \cap V_{join}$;
foreach $U^c_{join} \subseteq f^c.U$ s.t. $|U^c_{join}| = |V^c_{join}|$ **do**
    $key \leftarrow U^c_{join} \cup U^n_{join}$;
    $f^c_{out} \leftarrow (f^c.V \setminus V^c_{join}, f^c.U \setminus U^c_{join})$;
    $f^n_{out} \leftarrow (f^n.V \cup V^c_{join}, f^n.U \cup U^c_{join})$;
    output $(key; (f^c_{out}, f^n_{out}))$;

replace map^i in Algorithm 1.

Example 8. *Suppose $cm_1 = (f^n_1, f^c_1)$ in Example 6 is involved in the join with $V_{join} = \{v^\kappa_1, v^\kappa_3\}$. We have $f^n_1.V = \{v^\kappa_1\}$, $f^n_1.U = \{u_1\}$, $f^c_1.V = \{v^\kappa_2, v^\kappa_3, v^\kappa_4\}$ and $f^c_1.U = \{u_3, u_4, u_5, u_6\}$. We first compute $V^n_{join} = f^n_1.V \cup V_{join} = \{v^\kappa_1\}$ and $U^n_{join} = \{u_1\}$. Then we have $V^c_{join} = f^c.V \cap V_{join} = \{v^\kappa_3\}$. Consequently, we should partially expand f^c_1 by taking a node out of $f^c.U$ as U^c_{join}. We first take u_3, and the join key is $\{u_1, u_3\}$, then we compute $f^c_{out} = (\{v^\kappa_2, v^\kappa_4\}, \{u_4, u_5, u_6\})$ and $f^n_{out} = (\{v^\kappa_1, v^\kappa_3\}, \{u_1, u_3\})$. Ultimately, the key-value pair $(\{u_1, u_3\}; (f^c_{out}, f^n_{out}))$ is generated. We continue this process by iteratively taking u_4 , u_5 and u_6 from $f^c.U$.* □

Note that the clique to process can be a part of the pattern graph P, and the orders among $V(p_k)$ in P do not necessarily be the full order as in p_k. For example, the query q_6 in Figure 8.1 contains a 4-clique with nodes $\{v_2, v_3, v_4, v_5\}$, and the partial orders among these nodes in q_6 are $v_2 < v_5$ and $v_3 < v_4$. In this case, we should consider permutating the nodes of $V(p_k)$. Denote O_P as the set of partial orders in P. Given a $p_k \subseteq P$, we say a permutation σ of $V(p_k)$ satisfies O_P, if $\forall v_i < v_j \in O_P$, there exists $v^\kappa_s, v^\kappa_t \in V(p_k)$, such that $\sigma(v^\kappa_s) = v_i$ and $\sigma(v^\kappa_t) = v_j$. Consequently, when we output each compressed match in the function partialExpansion in Algorithm 5, we should apply all permutations that satisfies O_P in both $f^c.V$ and $f^n.V$.

Example 9. *Given the 4-clique $p_4 \subset q_6$ in Figure 8.1, we have $O_{q_6} = \{v_2 < v_5, v_3 < v_4\}$. We can verify that the permutation $\sigma = (v_3, v_2, v_5, v_4)$ satisfies O_{q_6}. Therefore, when we output, for example, cm_1 in Example 6 as $f^c_{out} = (\{v_3, v_5\}, \{u_4, u_5, u_6\})$ and $f^n_{out} = (\{v_2, v_4\}, \{u_1, u_3\})$ (Example 8), we should also output the result regarding σ as: $f^c_{out} = (\{v_2, v_4\}, \{u_4, u_5, u_6\})$, and $f^n_{out} = (\{v_3, v_5\}, \{u_1, u_3\})$.* □

Algorithm 6 presents the detailed algorithm of the revised reduce^i, which takes the compressed matches as inputs, and output the join results in the same compressed way,

Algorithm 6: reduce^i(**key**: U_{join}; **value**: Two sets of compressed matches H_1 and H_2)

```
   // We assume |h_1^c.V| ≥ |h_2^c.V| (w.l.g.).
1  foreach h_1 = (h_1^c, h_1^n) ∈ H_1, h_2 = (h_2^c, h_2^n) ∈ H_2 s.t.
   (h_1^n.U \ U_join) ∪ (h_2^n.U \ U_join) = ∅ do
2      if h_2^c.V = ∅ then
3          f^n.V = h_1^n.V ∪ h_2^n.V; f^n.U = h_1^n.U ∪ h_2^n.U;
4          f^c.V = h_1^c.V; f^c.U = h_1^c.U;
5          output (∅; (f^c, f^n));
6      else
           // We need to expand h_2 that has smaller compression.
7          f^c.V = h_1^c.V; f^c.U = h_1^c.U;
8          f^n.V = h_1^n.V ∪ h_2^n.V ∪ h_2^c.V;
9          foreach U' ⊆ h_2^c.U s.t. |U'| = |h_2^c.V| do
10             f^n.U = h_1^n.U ∪ h_2^n.U ∪ U' ;
11             output (∅; (f^c, f^n));
```

which can hence be treated as compressed matches in a future join. Given two compressed results $h_1 = (h_1^c, h_1^n)$ and $h_2 = (h_2^c, h_2^n)$, we say h_1 has larger compression power than h_2 if $|h_1^c.V| > |h_2^c.V|$. For the two compressed results from the current join patterns, the idea is to expand the one with smaller compression while keeping the other. In reduce^i, we process the join of h_1 and h_2 for each $h_1 \in H_1$ and $h_2 \in H_2$ (line 1). Without loss of generality, we assume that h_1 has larger compression power. If h_2 is not compressed (line 2), we simply union h_1^n and h_2^n to form the non-compressed part f^n (line 3), keep h_1^c in the compressed part f^c (line 4), and output the results. Otherwise, we have to expand h_2. We still keep h_1^c in f^c (line 8), while f^n now includes not only h_1^n and h_2^n, but a part from the expansion of h_2^c (line 8). We hence go through every $|h_2^c.V|$ nodes $U' \subseteq h_2^c.U$ (permutation may be needed, as Algorithm 4), construct each $f^n.U$ via the union of $h_1^n.U$, $h_2^n.U$ and U' (line 10), and output the compressed results (line 11). In this way, we compress the results in each join, and they can be treated just like the compressed matches of the k-cliques in the future join (Algorithm 5).

8 Performance Studies

In this section, we show our experimental results. We rented a cluster from Amazon of up to 15 computing nodes including one master node and 14 slave nodes and we used 10 slave nodes by default. The instance configurations of master and slave nodes are listed in Table 8.1. We allocated a JVM heap space of 1524MB for each mapper and 2848MB for each reducer, and we allowed at most 6 mappers and 6 reducers running concurrently in each machine. The block size in HDFS was set to be 128MB, the data replication factor of HDFS was set to be 3, the I/O sort size was set to be 512MB, and the I/O sort factor was set to be 10.

Node Type	Instance	vCPU	Memory	Storage
master	m3.xlarge	4	15GB	2×40GB SSD
slave	c3.4xlarge	16	30GB	2×160GB SSD

Table 8.1: Amazon virtual instance configurations

Datasets. We tested five real-world data graphs (see Table 8.2). Among them, *lj*, *orkut* and *fs* were downloaded from SNAP (http://snap.stanford.edu), *yt* was downloaded from KONECT (http://konect.uni-koblenz.de), and *eu* and *uk* was downloaded from WEB (http://law.di.unimi.it). *pg21* and *pg29* are two PR graphs generated via [36] with $\beta = 2.1$ and $\beta = 2.9$, respectively. For each dataset, we list the number of nodes and edges (in millions), and $T(G)$ - the time of constructing the SCP graph storage $\Phi^2(G)$ (Section 4) and $T(C)$ - the time of enumerating large cliques in the data graph for clique compression (Section 7). Note that the $T(G)$ and $T(C)$ for *pg21* and *pg29* are of no interest. The computations of SCP graph storage and the large cliques are query independent, and thus are considered as preprocessing steps.

dataset	name	N(mil)	M(mil)	**T(G)**(s)	**T(C)**(s)
youtube	*yt*	3.22	12.22	27	58
eu-2015	*eu*	0.86	16.14	41	129
live-journal	*lj*	4.85	42.85	54	170
com-orkut	*orkut*	3.07	117.19	185	345
uk-2002	*uk*	18.52	261.79	841	1270
friendster	*fs*	65.61	1806.07	2331	368
power-law($\beta = 2.1$)	*pg21*	10,000	50,000	-	-
power-law($\beta = 2.9$)	*pg29*	10,000	50,000	-	-

Table 8.2: Datasets used in Experiments

Algorithms. We compared six algorithms:

- SEED: The baseline SEED algorithm implemented in MapReduce with optimal bushy join plan (Section 6) and overlapping join units (Section 6.2).
- SEED-LD: SEED but with the (best) left-deep join plan.
- SEED-NO: SEED without overlapping join units.
- SEED+O : SEED with clique-compression (Section 7).
- SEEDS[(M), (DM), (D)]: The SEED+O algorithm implemented in Spark with MEMORY_ONLY cache, MEMORY_AND_DISK cache and DISK_ONLY cache, respectively.
- TT: The TwinTwigJoin algorithm implemented in MapReduce with all optimizations [20].
- TTS[(M), (DM), (D)]: The TwinTwigJoin algorithm implemented in Spark with the three caching mechanisms.
- PSgL: The Pregel-based subgraph enumeration algorithm with all optimizations proposed by Shao et al. [30].

All algorithms were compiled with Java 1.7. We implemented SEED and TT with Hadoop 2.6.0, and SEEDS[(M), (DM), (D)] and TTS[(M), (DM), (D)] with Spark 1.4. All algorithms except PSgL are running on the Yarn framework. The authors of [30] kindly provided the codes for PSgL, implemented with Hadoop 1.2.0 on an old MapReduce framework. The performance gap between Yarn and old MapReduce is

very small, hence the comparison between PSgL and the other algorithms is fair. We set the maximum running time to 3 hours. If a test did not stop within the time limit, or failed due to out-of-memory exceptions or other errors, we denote the running time as INF. The time to compute the join plan using Algorithm 2 is less than one second for all test cases, and thus has been omitted from the total processing time.

Queries. The seven queries denoted by q_1 to q_7 are illustrated in Figure 8.1 with the number of edges varying from 4 to 10 and the number of nodes varying from 4 to 6. We show the order of the nodes for automorphism resolution (Remark 1) under each query graph. Here, we have considered all queries except for triangle from the-state-of-the-art algorithms in the literature [30, 20]. Note that triangle enumeration is used in this paper as a preprocessing step to construct the SCP storage. We added the query q_5 and q_6 to further demonstrate the advantages of our proposed techniques.

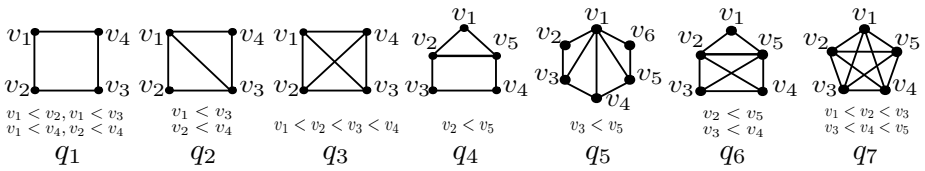
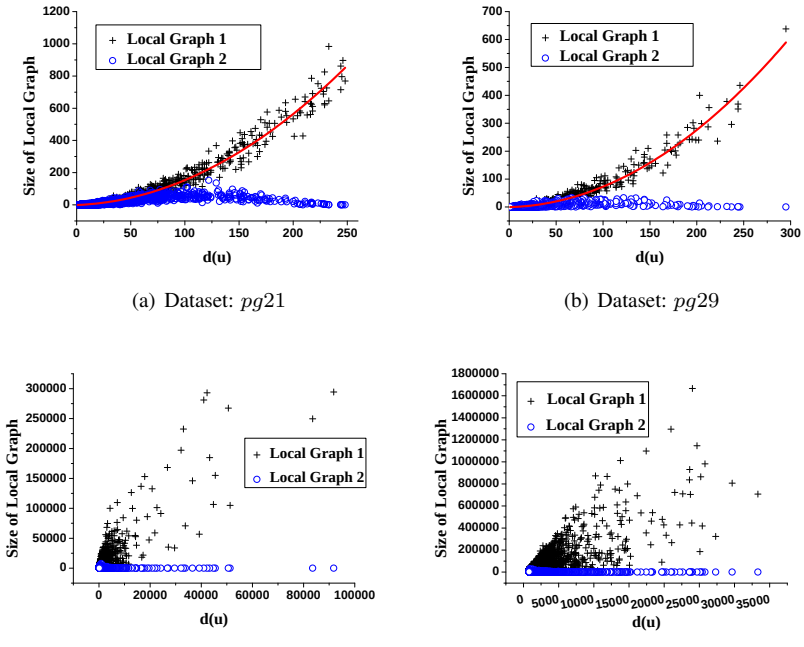


Figure 8.1: Queries



(a) Dataset: $pg21$

(b) Dataset: $pg29$

(c) Dataset: yt

(d) Dataset: $orkut$

Figure 8.2: Local Graph Statistics

Exp-1: Local-Graph Statistics. This experiment studied the relationships between $\Delta_u^{(1)}$ (Lemma 2) and $\Delta_u^{(2)}$ (Lemma 4) w.r.t. $d(u)$. Results on the datasets *pg21*, *pg29*, *yt* and *orkut* are shown in Figure 8.2, where the X-axis depicts $d(u)$ and the Y-axis depicts $\Delta_u^{(1)}$ and $\Delta_u^{(2)}$. For clarity, we only reported the results for top-10000 largest

nodes of *yt* and *orkut*. It is clear that $\Delta_u^{(1)}$ is much more skewed and larger than $\Delta_u^{(2)}$ in all tests. Particularly, we fit the curves for the $d(u)$-$\Delta_u^{(1)}$ relationship in the synthetic graphs. The results show that $\Delta_u^{(1)}$ is almost proportional to $d(u)^2$, which conforms with the theoretical result in Lemma 2. On the real graphs *yt* and *lj* (Figure 8.2(c), Figure 8.2(d)), besides skewness, we also observe that $\Delta_{max}^{(1)}$ on *lj* is nearly half the size of the data graph. As a result, $\Phi^1(G)$ can not scale for handling large data graphs. On the other side, the distributions of $\Delta_u^{(2)}$ are comparatively flat and small over $d(u)$ in all tests. The results validate that $\Phi^2(G)$ achieves good load balance and scalability. The local-graph statistics for all other datasets are similar to those shown in Figure 8.2 and hence have been omitted.

Exp-2: Bushy vs Left-deep. We compare the performance of SEED and SEED-LD using query q_5 on *yt* to test the advantage of using the bushy join plan. The plans $\mathcal{E}_1$ and $\mathcal{E}_2$ shown in Figure 3.1 illustrate the optimal execution plans for SEED-LD and SEED, respectively. Table 8.3 presents the experimental results, in which we observe a much better performance of SEED, compared to SEED-LD. We also show the output of mappers and reducers in each stage and compute the cost using Equation 5.3. The output of reduce^3 is not shown, as it is the final result and excluded in the cost. Observe that the algorithm with smaller cost always ends up with better performance, which supports our motivation to minimize the cost in Section 6. Clearly, SEED, with smaller cost, performs better than SEED-LD. The results are similar in the other datasets. We conclude that the optimal bushy join plan computed via Algorithm 2 outperforms the left-deep join plan.

M/R	map^1	red^1	map^2	red^2	map^3	Cost	Time(s)
SEED	12.3	3.2	12.3	3.2	6.4	471.9	306
SEED-LD	12.3	3.2	15.5	6110.9	6123.9	31365.2	INF

Table 8.3: Cost comparisons while enumerating q_5 on *yt* with SEED and SEED-LD (in millions).

As we mentioned in Remark 4, the plans $\mathcal{E}_1$ and $\mathcal{E}_2$ in Figure 3.1 are also the optimal execution plans computed via the ER model and the PR model, respectively. In Table 8.3, the outputs of reduce^2 of SEED and SEED-LD correspond to $|R(P_2^b)|$ and $|R(P_2^{ld})|$, and it is obvious that $|R(P_2^{ld})| \gg |R(P_2^b)|$. The results are consistent with our analysis in Remark 4 that the PR model offers more realistic cost estimation, which leads to better execution plan.

We chose q_5 in this experiment because of two reasons: (1) its optimal join plan is bushy; (2) the "optimal" join plans computed via ER model and PR model are different.

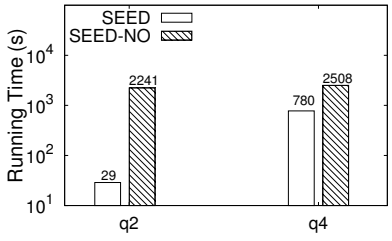
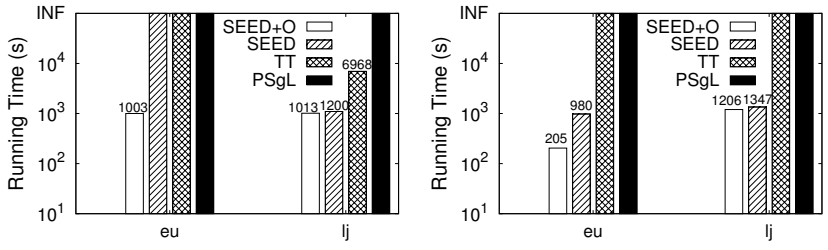


Figure 8.3: SEED vs SEED-NO.

Exp-3: Overlapping Join Units. This experiment studied the benefit of overlapping the join units (Section 6.2). We processed the queries q_2 and q_4 on the dataset *yt* using SEED and SEED-NO. In q_2, SEED joins two triangles $p_0 = ((v_1, v_2), (v_2, v_3), (v_1, v_3))$ and $p_1 = ((v_1, v_3), (v_1, v_4), (v_3, v_4))$, while SEED-NO, without overlapping the join units, can only join p_0 to a TwinTwig $p'_1 = ((v_1, v_4), (v_3, v_4))$. Similarly, while processing q_4, SEED handles the triangle $((v_1, v_2), (v_1, v_5), (v_2, v_5))$ on the top, while SEED-NO can only use the TwinTwig $((v_1, v_2), (v_1, v_5))$. In practice, the triangle often renders much fewer matches than the two-edge TwinTwig. Therefore, SEED outperforms SEED-NO, as shown in Figure 8.3. We obtained similar results on all queries other than q_1 (no overlapping exists), q_3, q_7 (clique is the join unit), and we only presented q_2 and q_4 as representatives.



(a) q_6 (b) q_7

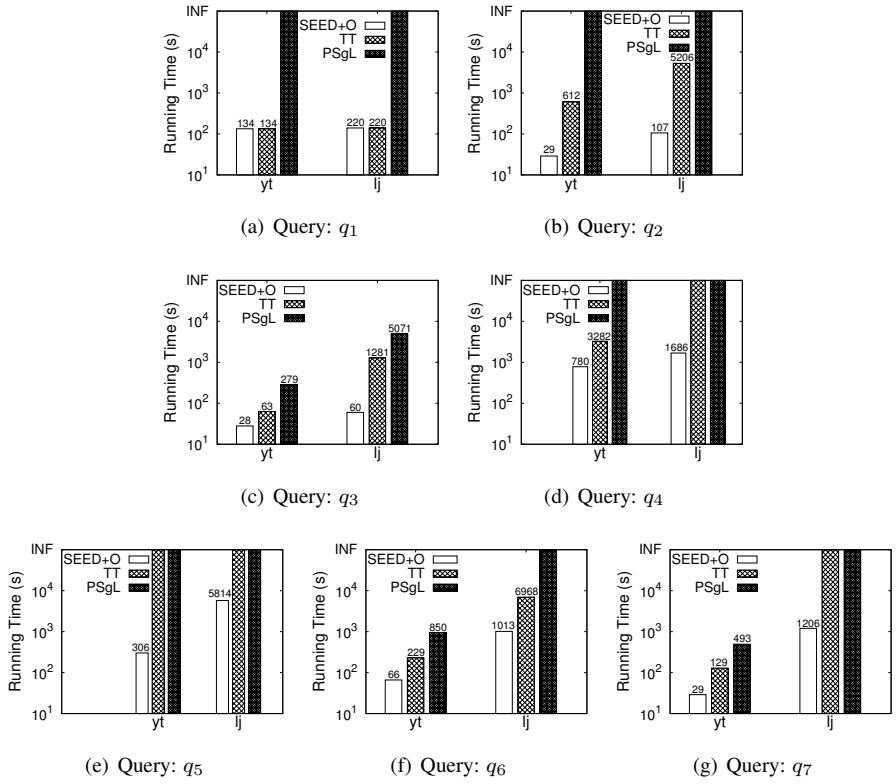
Figure 8.4: The effects of the proposed techniques.

Exp-4: The effects of the proposed techniques. To show the effects of the proposed techniques, we evaluated the performance of SEED+O , SEED, TT and PSgL by querying q_4 and q_6 on *eu* and *lj*, and reported the results in Figure 8.4(a)-Figure 8.4(b). Observe that the baseline SEED already dominates the state-of-the-art algorithms TT and PSgL. SEED processes q_7 on *eu* and *lj* in 980 seconds and 1347 seconds, respectively, while neither TT nor PSgL can terminate in the allowed time. SEED outperforms TT and PSgL, benefiting from the SCP graph storage ($\Phi^2(G)$) that supports clique as the join unit. Consequently, SEED processes q_6 by joining the upper triangle and the bottom 4-clique in just one single round, while TT and PSgL both process 3 rounds. Although there are extra overheads constructing the new graph storage (see $T(G)$ in Table 8.2), SEED still performs much better than TT and PSgL after considering these overheads. For example, SEED processes q_6 on *lj* in 54 + 1347 seconds, while TT runs 6968 seconds and PSgL cannot even terminate.

SEED+O further improves SEED via clique compression (Section 7). Observe that SEED+O runs faster than SEED in all tests, especially in the process of q_6 on *eu*, where SEED+O terminates in 1003 seconds but SEED runs out of time. Note that the effect of clique compression is more notable on *eu* than that on *lj*. The reason is that, to our best speculation, in a web graph like *eu*, web pages within a domain tend to link each other to form large cliques, while in a social network like *lj*, such a strong tie is rarely formed; Obviously, larger clique in the data graph contributes to better clique compression. Although we spend time enumerating and maintaining the large cliques (see $T(C)$ in Table 8.2) for clique compression, the technique does improve the performance of SEED, and it will play an important role when the data graph contains many large cliques (e.g. while processing q_6 on *eu*).As SEED+O beats SEED, we would only compare SEED+O , and exclude the baseline SEED in the rest

of the experiments.

We could also use the other queries in this test, but q_1, q_2, q_4 and q_5 do not contain cliques of more than three nodes, and the process of q_6 already includes enumerating q_3. Thus, we only use q_6 and q_7 here to fairly show the advantages of our proposed algorithms. Next we would compare the algorithms against all queries.



(a) Query: q_1 (b) Query: q_2

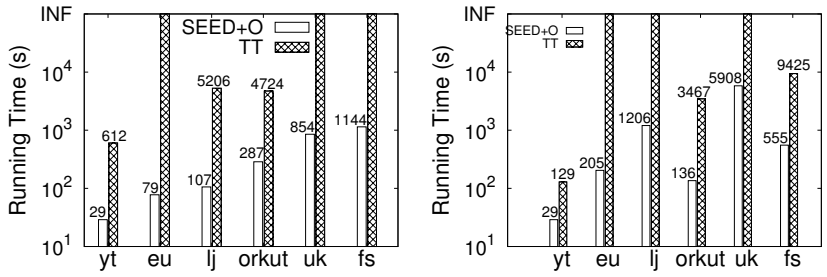
(c) Query: q_3 (d) Query: q_4

(e) Query: q_5 (f) Query: q_6 (g) Query: q_7

Figure 8.5: Test against all queries.

Exp-5: Test against all queries. We compared SEED+O with TT and PSgL - by enumerating all queries on *yt* and *lj*, and reported the results in Figure 8.5(a)-Figure 8.5(g). When enumerating q_1, SEED+O uses the same execution plan, and hence has the same performance as TT, and they outperform PSgL. In all the other queries, SEED+O significantly outperforms TT, due to the use of clique as the join unit. For example, SEED+O is over $30\times$ faster than TT while processing q_2 on both *yt* and *lj*, and over $20\times$ faster than TT while processing q_3 on *lj*. Moreover, SEED+O processes complex queries such as q_4, q_5, q_6 and q_7 efficiently on both *yt* and *lj*. On the contrary, TT often runs out of time when querying on *lj*. PSgL can only process q_3 on *yt* and *lj*, and q_7 on *yt*, and in these cases, PSgL performs worse than TT. The reasons are two aspects. First, PSgL can be seen as StarJoin, which is already proven to be worse than TwinTwigJoin [20]. Second, the Pregel-based PSgL maintains all intermediate results in the main memory, and the numerous intermediate results produced in subgraph enumeration can exhaust the memory. In conclusion, the proposed SEED+O algorithm significantly outperforms all existing algorithms, and TT also performs better than PSgL. Next we would exclude PSgL from the experiments, as it can only process

simple queries on relatively small datasets.



(a) Query: q_2 (b) Query: q_7

Figure 8.6: Vary Datasets.

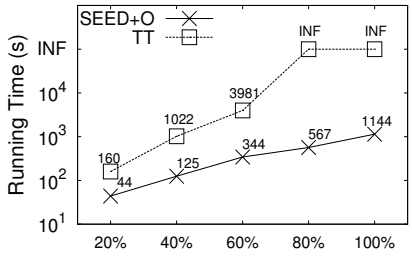
Exp-6: Vary Datasets. We compared SEED+O with TT by querying q_2 and q_7 on all datasets in order to show the advantages of SEED+O regarding different data properties. The results are shown in Figure 8.6(a)-Figure 8.6(b). In all tests, SEED+O significantly outperforms TT, with the performance gain varying from an order of magnitude to over $50\times$ (enumerating q_2 on *lj*). Specifically, SEED+O processes q_2 on the two largest datasets - *uk* and *lj*, in less than 20 minutes, while TT cannot terminate in the allowed time. This experiment demonstrates that SEED+O scales better for handling large data graphs due to the use of clique as the join unit, and the optimal bushy join plan with overlapping join units.

Exp-7: Vary Graph Size. We extracted subgraphs of 20%, 40%, 60%, 80%, and 100% nodes from the original graph of *fs*, and tested the algorithms using queries q_2 and q_7. The results are shown in Figure 8.7(a) and Figure 8.7(b) respectively. When the graph size increases, the running time of TT grows much more sharply than SEED+O . When the graph size is over 60%, only SEED+O finishes enumerating q_2 in the time limit. The test shows the high scalability of our SEED+O algorithm.

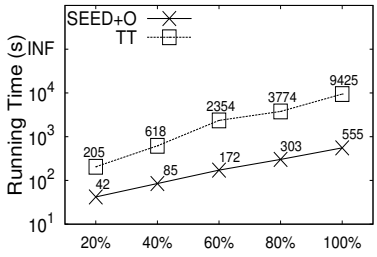
Exp-8: Vary Average Degree. We fixed the set of nodes and randomly sampled 20%, 40%, 60%, 80%, and 100% edges from the original graph *fs* to generate graphs with average degrees from 11 to 55, and tested the algorithms using queries q_2 and q_7. The results are shown in Figure 8.7(c) and Figure 8.7(d) respectively. In Figure 8.7(d), SEED+O is 10, 15, 19 and 17 times faster than TT when the average degree varies from 11 to 55, which shows the advantage of SEED+O for dense data graphs.

Exp-9: Vary Slave Nodes. In this experiment, we varied the number of slave nodes from 6 to 14, and evaluated our algorithms on the *lj* and *orkut* datasets using queries q_2 and q_7. The test results are shown in Figure 8.8(a)-Figure 8.8(d) respectively. When the number of slave nodes increases, the running time of all algorithms decreases, and it drops more sharply when the number of slave nodes is small. On the one hand, increasing the number of slave nodes improves performance by sharing the workload; on the other hand, it introduces extra communication costs from data transmissions among the slave nodes. As shown in Figure 8.8(b), even when 14 slave nodes are deployed, SEED+O is the only algorithm that can process q_7 on *lj*. We also tested other queries with various amount of slave nodes, and found curves similar to those in Figure 8.8(a)-Figure 8.8(d), thus the results have been omitted.

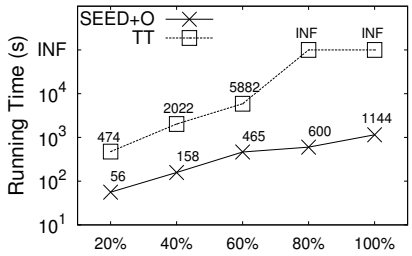
Exp-10: MapReduce VS. Spark. In this experiment, we compared the implementations of both SEED and TwinTwigJoin algorithms in MapReduce (Hadoop)



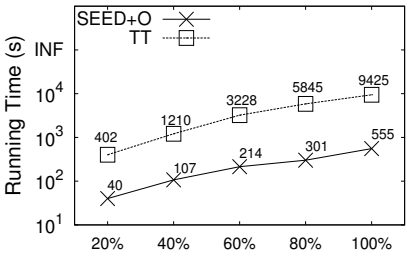
(a) Vary Graph Size: q_2



(b) Vary Graph Size: q_7

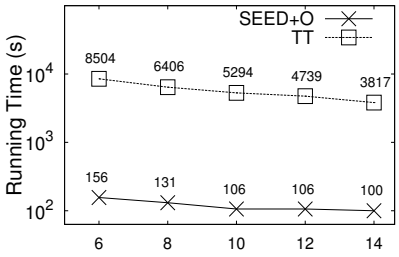


(c) Vary Avg. Degree: q_2

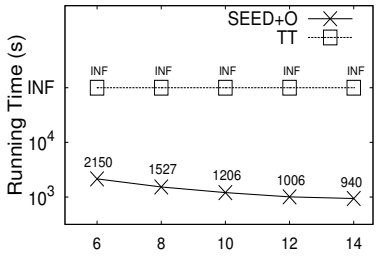


(d) Vary Avg. Degree: q_7

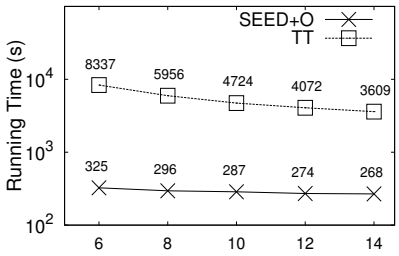
Figure 8.7: Vary Graph Properties



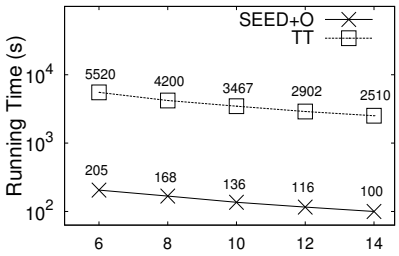
(a) On lj: q_2



(b) On lj: q_7

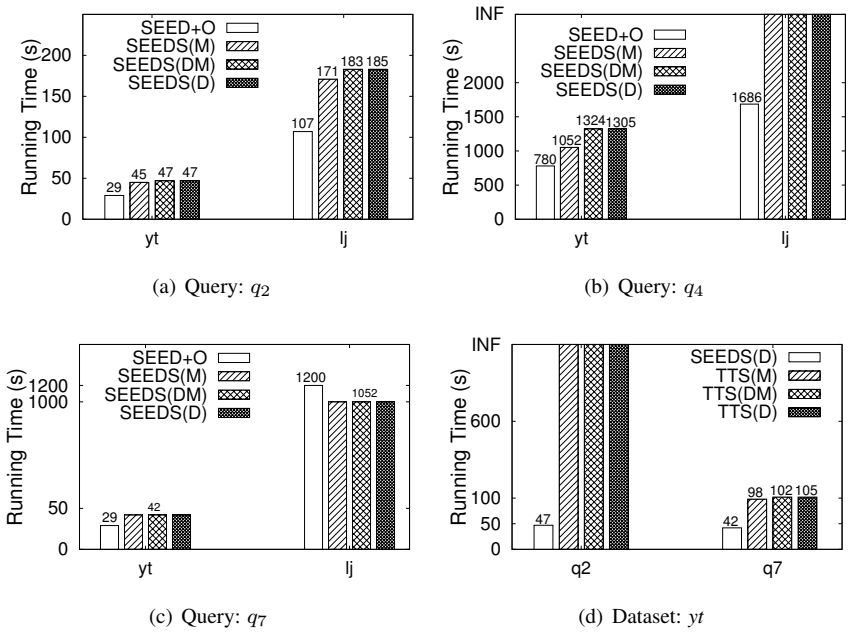


(c) On orkut: q_2



(d) On orkut: q_7

Figure 8.8: Vary Slave Nodes



(a) Query: q_2

(b) Query: q_4

(c) Query: q_7

(d) Dataset: *yt*

Figure 8.9: Spark VS. MapReduce

and Spark, the two most popular large data processing engines. We adopted three caching mechanisms in Spark, namely MEMORY_ONLY, MEMORY_AND_DISK and DISK_ONLY[1]. We first compared SEED+O with SEEDS[(M), (DM), (D)] by processing q_2, q_4 and q_7 on the datasets *yt* and *lj*, and show the results in Figure 8.9(a)-Figure 8.9(c). SEEDS(M) performs better than SEEDS(DM) and SEEDS(D), due to the memory-caching strategy. The benefits are not very obvious in most queries. While processing q_2, SEEDS(M) performs only slightly better (5%), and the reason may be that the communication cost dominates the cost of caching the (intermediate) results. As for q_7, SEEDS[(M), (DM), (D)] have the same performance, since the process terminates in a single round without result caching when we use clique as the join unit. When we compared the MapReduce version - SEED+O - to SEEDS(M), it is surprising to see that SEEDS(M) do not show advantages given its memory-caching mechanism. As we can see, SEED+O outperforms SEEDS(M) in the tests of q_2 and q_4. A possible reason is that MapReduce (Hadoop) overlaps the shuffle and map stages, which can effectively hide the huge communication overhead in subgraph enumeration. On the contrary, Spark's shuffle must wait for the completions of all mappers. The authors in [32] also reported similar observations. As for q_7, SEEDS(M) does perform better than SEED+O , and in this single-round process, Spark may possibly benefit from the adoption of the Kryo serialization, which is significantly faster and more compact than Java serialization adopted in Hadoop [2]. Finally, it is worth noting that all Spark implementations fails while processing q_4 on *lj*. We have found enormous matches of q_4 on *lj*, and SEEDS(M) stops with "OUT_OF_MEMORY" error when attempting to caching all (intermediate) results in memory. SEEDS(DM) and SEEDS(D) face an "RDD_OVERSIZE" error when some partitions of the results exceed the maximum size of an "RDD" (RDD is a basic storage unit in Spark). This has been reported as

[1] http://spark.apache.org/docs/latest/programming-guide.html
[2] http://spark.apache.org/docs/latest/tuning.html

one of the biggest yet unresolved issues of Spark at current version [3]. This concludes that Spark hardly competes with MapReduce in the task of subgraph enumeration. Subgraph enumeration often produces numerous intermediate results, which exceeds Spark's processing capacity, which partly explained that we performed most tests using MapReduce.

We further compared SEEDS(D) and TTS[(M), (DM), (D)] by processing the queries q_2 and q_7 on *yt*, and demonstrated the results in Figure 8.9(d). We did not show the results for *lj* as the TTS[(M), (DM), (D)] failed all cases. As we mentioned earlier, SEEDS(D) maintains the intermediate results on the disk and is relatively slower, but it beats TwinTwigJoin in all test cases. In particular, SEEDS(D) processes q_2 on *yt* in 47 seconds, while TTS[(M), (DM), (D)] cannot terminate in the allowed time. The result set of q_7 on *yt* is too small to demonstrate the advantage of SEED, but SEEDS(D) is still over two times faster than TTS[(M), (DM), (D)]. As a complement of the comparisons in MapReduce, the results in this test confirm that SEED outperforms TwinTwigJoin regardless of the platforms.

9 Related Work

Subgraph Matching. Most subgraph matching approaches work in labeled context, where nodes (and/or edges) are assigned labels in both data and query graphs. For example, node labels in the neighborhood are used to filter unexpected candidates in [14] and [39]. In [13], the authors observe that a good matching order can significantly improve the performance of subgraph query. Lee et al. [21] provide an in-depth comparison of subgraph isomorphism algorithms. Subgraph enumeration in a centralized environment is also studied in exact and approximate settings. The exact solutions including [3] and [11] are not scalable for handling large data graphs. The approximate solutions [2, 10, 40] only estimate the count, but do not locate all the subgraph instances.

Subgraph Matching in Cloud. Many recent works have focused on solving subgraph matching in the cloud. Zhao et al. [40] introduced a parallel color coding method for subgraph counting. Ma et al. [23] studied inexact graph pattern matching based on graph simulation in a distributed environment. Sun et al. [33] proposed a subgraph matching algorithm that uses node filtering to handle labeled graphs in the Trinity memory cloud. Recently, Shao et al. [30] developed PSgL to list subgraph instances in Pregel, which can be seen as a StarJoin-like algorithm and already proven to be worse than the TwinTwigJoin algorithm [20].

Subgraph Enumeration in MapReduce. Subgraph enumeration in MapReduce has attracted a lot of interests. Tsourakakis et al. [35] proposed an approximate triangle counting algorithm using MapReduce. Suri et al. [34] introduced a MapReduce algorithm to compute exact triangle counting. Afrati et al. [1] proposed multiway join in MapReduce to handle subgraph enumeration. Plantenga [27] introduced an edge join method in MapReduce which can be used for subgraph enumeration. In [9], small cliques are enumerated using MapReduce, however the method can only be used to enumerate small cliques rather than any general pattern graphs. The TwinTwigJoin algorithm was proposed in [20], which has proven to be instance optimal in the left-deep-join framework. However, using TwinTwig as the join unit is still inefficient

[3] https://issues.apache.org/jira/browse/SPARK-6190

when processing large-degree nodes and the left-deep join plan may result in non-optimal solutions.

10 Conclusions

In this paper, we studied the subgraph enumeration problem, considering that existing solutions did not scale well to large graphs. We proposed SEED, a scalable distributed subgraph enumeration algorithm. Compared to the-state-of-the-art TwinTwigJoin, SEED is featured with the following: (1) a novel SCP graph storage mechanism that allows using cliques, in addition to stars, as the join unit; (2) a comprehensive cost model based on the PR model; (3) a dynamic-programming algorithm to compute the optimal bushy join plan with overlapping join units; (4) the clique compression technique that further improves the performance. We have conducted extensive performance studies on real graphs with up to billions of edges, which shows that SEED outperforms the state-of-the-art works by over an order of magnitude.

Bibliography

[1] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *Proc. of ICDE'13*, 2013.

[2] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp. Biomolecular network motif counting and discovery by color coding. In *Proc. of ISMB'08*, 2008.

[3] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1), 1985.

[4] F. Chung, L. Lu, and V. Vu. Eigenvalues of random power law graphs. *Annals of Combinatorics*, 7(1):21–33, 2003.

[5] F. R. K. Chung, L. Lu, and V. H. Vu. The spectra of random graphs with given expected degrees. *Internet Mathematics*, 1(3), 2003.

[6] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Rev.*, Nov. 2009.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proc. of OSDI'04*, 2004.

[8] P. Erdos and A. Renyi. On the evolution of random graphs. In *Publ. Math. Inst. Hungary. Acad. Sci.*, 1960.

[9] I. Finocchi, M. Finocchi, and E. G. Fusco. Counting small cliques in mapreduce. *CoRR*, abs/1403.0734, 2014.

[10] M. Gonen, D. Ron, and Y. Shavitt. Counting stars and other small subgraphs in sublinear time. In *Proc. of SODA'10*, 2010.

[11] J. A. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *Proc. of RECOMB'07*, 2007.

[12] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, and et al. Demonstration of the myria big data management service. In *SIGMOD '14*.

[13] W.-S. Han, J. Lee, and J.-H. Lee. Turboiso: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proc. of SIGMOD'13*, 2013.

[14] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proc. of SIGMOD'08*, 2008.

[15] X. Hu, Y. Tao, and C.-W. Chung. Massive graph triangulation. In *SIGMOD '13*, pages 325–336.

[16] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *SIGMOD'91*, pages 168–177, 1991.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys '07*.

[18] M. Jarke and J. Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, June 1984.

[19] S. R. Kairam, D. J. Wang, and J. Leskovec. The life and death of online groups: Predicting group growth and longevity. In *Proc. of WSDM'12*, 2012.

[20] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *PVLDB*, 8(10), 2015.

[21] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2), 2012.

[22] J. Leskovec, A. Singh, and J. Kleinberg. Patterns of influence in a recommendation network. In *Proc. of PAKDD'06*, 2006.

[23] S. Ma, Y. Cao, J. Huai, and T. Wo. Distributed graph pattern matching. In *WWW*, 2012.

[24] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proc. of SIGMOD'10*, 2010.

[25] T. Milenkovic and N. Przulj. Uncovering biological network function via graphlet degree signatures. *Cancer Inform*, 6, 2008.

[26] 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), 2002.

[27] T. Plantenga. Inexact subgraph isomorphism in mapreduce. *J. Parallel Distrib. Comput.*, 73(2), 2013.

[28] N. Przulj. Biological network comparison using graphlet degree distribution. *Bioinformatics*, 23(2), 2007.

[29] G. Rücker and C. Rücker. Substructure, subgraph, and walk counts as measures of the complexity of graphs and molecules. *Journal of Chemical Information and Computer Sciences*, 41(6), 2001.

[30] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu. Parallel subgraph listing in a large-scale graph. In *SIGMOD'14*, pages 625–636. ACM, 2014.

[31] N. Shervashidze, S. V. N. Vishwanathan, T. Petri, K. Mehlhorn, and K. M. Borgwardt. Efficient graphlet kernels for large graph comparison. In *AISTATS*, 2009.

[32] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proc. VLDB Endow.*, 8(13):2110–2121, Sept. 2015.

[33] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. *PVLDB*, 5(9), 2012.

[34] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proc. of WWW'11*, 2011.

[35] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *Proc. of KDD'09*, 2009.

[36] F. Viger and M. Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. In *COCOON'05*.

[37] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[38] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *HotCloud'10*, pages 10–10.

[39] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1-2), 2010.

[40] Z. Zhao, M. Khan, V. S. A. Kumar, and M. V. Marathe. Subgraph enumeration in large social contact networks using parallel color coding and streaming. In *Proc. of ICPP'10*, 2010.

APPENDIX

In this appendix, we introduce the algorithm of symmetry breaking in [11] and formally prove its correctness. we first give some preliminary knowledge of automorphism. Then we show the intuition of eliminating duplicated enumeration by the order-preservation constraint, and present the symmetry-breaking algorithm via assigning the partial orders among nodes in the pattern graph introduced in [11]. Finally, we show the correctness of the algorithm.

Automorphism. We denote the automorphism group of a graph Γ as $\mathcal{A}_\Gamma$. We say v_1 and v_2 in a graph Γ is automorphism equivalent, denoted $v_1 \sim v_2$, iff there is an automorphism α of Γ s.t. $\alpha(v_1) = v_2$. The equivalence classes of the nodes of a graph under the action of the automorphisms are called *node orbits*. Here we simply call it orbit. An orbit is trivial if it contains only one node. We denote $\mathcal{O}_\mathcal{A}$ the set of orbits given by the automorphism group $\mathcal{A}$.

We often use a permutation π to express an automorphism, which is further represented in the *disjoint cycle form*.

Example 10. *Considering an automorphism that maps* $(v_1, v_2, v_3, v_4, v_5, v_6)$ *to* $(v_3, v_2, v_1, v_6, v_5, v_4)$*, correspondingly. It gives the permutation* $\pi = \begin{pmatrix} v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \\ v_3 & v_2 & v_1 & v_6 & v_5 & v_4 \end{pmatrix}$*, which has the disjoint cycle form as* $\pi = (v_1 v_3)(v_2)(v_4 v_6)(v_5)$.

For the sake of simplicity, we will ignore an element v in the cycle form of an automorphism α if $\alpha(v) = v$. For example, $(v_1 v_3)(v_2)(v_4)(v_5)(v_6)$ will be simplified as $(v_1 v_3)$. We use $\mathbb{I}$ to denote the identity.

Order-Preservation Constraint and Symmetry Breaking. Suppose a subgraph g of G is isomorphic to the pattern graph P. Let f be the match that maps $V(P)$ to $V(g)$. We consider an orbit $\{v_1, v_2, \cdots, v_k\}$ of P w.r.t $\mathcal{A}_P$, and a data node set $\{u_1, u_2, \cdots, u_k\}$. Without loss of generality, we assume u_1 has the smallest order, and $f(v_i) = u_i$ for all $1 \leq i \leq k$. Taking v_1 as an example, it is clear that there exists an automorphism $\alpha_i \in \mathcal{A}_P$ where $\alpha_i(v_1) = v_i$ for $2 \leq i \leq k$. Therefore, for all the α_i, the mapping $\alpha_i \circ f$ from $V(P)$ to $V(g)$ corresponds to the same subgraph instance g, leading to duplicated enumerations. We then consider an order $<$ among some nodes in the orbit and apply the order-preservation constraint on the mapping (order-preserved mapping). More specifically, if $v_i < v_j$, the mapping f is allowed iff $f(v_i) \prec f(v_j)$. In this case, we enforce the order $v_1 < v_i$ for all $2 \leq i \leq k$, which eliminates all $\alpha \circ f$ where $\alpha(v_1) \neq v_1$ (or similarly preserves only $\alpha \circ f$ where $\alpha(v_1) = v_1$) by order preservation. In this way, we avoid the duplicated enumeration caused by v_1. Note that by enforcing such an order, a mapping f from $V(P)$ to $V(g)$ is valid iff $f(v_1) = u_1$, which means v_1 can only be mapped to a fixed node in a given subgraph instance.

We call the node v in the pattern graph P the *fixed node* if given a subgraph g of G, all the valid matches (according to Definiton 1) that map v to a fixed node in g. We know that each node that belongs to a *trivial orbit* is a *fixed node*. As discussed previously, after assigning the orders to v_1, that is $v_1 < v_i$ for all $2 \leq i \leq k$, v_1 becomes a fixed node by the order-preserved matching.

Symmetry Breaking Algorithm. The algorithm in [11] first initializes the automorphism group $\mathcal{A}$ as $\mathcal{A} \leftarrow \mathcal{A}_P$. It then runs the following steps iteratively until $\mathcal{A} = \{\mathbb{I}\}$.

- Pick up the largest orbit $\{v_{i_1}, v_{i_2}, v_{i_3}, \cdots, v_{i_k}\}$ from $\mathcal{O}_{\mathcal{A}}$.
- Assign the order $v_{i_1} < v_{i_2}, v_{i_1} < v_{i_3}, \cdots, v_{i_1} < v_{i_k}$ to make v_{i_1} a *fixed node*.
- Refine $\mathcal{A} \leftarrow \{\alpha | \alpha \in \mathcal{A} \ \wedge \ \alpha(v_{i_1}) = v_{i_1}\}$.

The final step refines the automorphism group $\mathcal{A}$ which contains only the automorphisms that map v_1 to itself. It is easy to verify that after the refinement $\mathcal{A}$ is still a group. Note that after $\mathcal{A}$ is refined, $\mathcal{O}_{\mathcal{A}}$ is refined correspondingly. We show a running example using the pattern graph presented in Figure A.1.

Example 11. *The automorphism group* $\mathcal{A}$ *is initialized as* $\mathcal{A}_P = \{\mathbb{I}, (v_1 v_2 v_3)(v_4 v_5 v_6), (v_1 v_3 v_2)(v_4 v_6 v_5), (v_2 v_3)(v_5 v_6), \ (v_1 v_3)(v_4 v_6), (v_1 v_2)(v_4 v_5)\}$*. The automorphisms partition the nodes into two orbits, namely* $\mathcal{O}_{\mathcal{A}} = \{\{v_1, v_2, v_3\}, \{v_4, v_5, v_6\}\}$*. We first pick up* v_1 *from* $\{v_1, v_2, v_3\}$*, and assign the order* $v_1 < v_2, v_1 < v_3$*. We then refine* $\mathcal{A}$ *to contain the automorphisms that map node* v_1 *to itself, which gives* $\mathcal{A} = \{\mathbb{I}, (v_2 v_3)(v_5 v_6)\}$*, and the new orbits given by* $\mathcal{A}$ *on the remaining nodes are clearly* $\mathcal{O}_{\mathcal{A}} = \{\{v_2, v_3\}, \{v_5, v_6\}\}$*. We further pick up* v_2 *and assign the order* $v_2 < v_3$*. After this,* $\mathcal{A}$ *should be refined to contain the automorphisms that map* v_1 *to* v_1*, and* v_2 *to* v_2*, which gives* $\mathcal{A} = \{\mathbb{I}\}$*. The algorithm hence terminates by assigning the following order:* $v_1 < v_2, v_1 < v_3, v_2 < v_3$.

Next, we prove the correctness of the techniques proposed in [11] by showing that

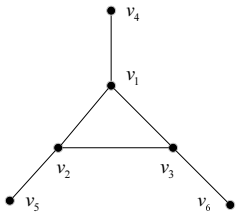


Figure A.1: A pattern graph for symmetry breaking

the results after applying the order restraints given by the algorithm 1) are complete; 2) contain no duplicated results.

Correctness of the Algorithm. 1) Completeness. We prove the completeness of the results by induction on the step of the algorithm. By completeness we mean that the algorithm only eliminates the matches introduced by automorphisms. To start, namely step 0, the results are clearly complete. In step k, we assume that the results are complete. In step $k+1$, according to the algorithm, we pick up an orbit, $\{v_{i_1}, v_{i_2}, \cdots, v_{i_k}\}$ from $\mathcal{O}_{\mathcal{A}}$, and assign the order $v_{i_1} < v_{i_2}, v_{i_1} < v_{i_3}, \cdots, v_{i_1} < v_{i_k}$. We consider $v_{i_1} < v_{i_2}$, and the automorphism $\alpha = (v_{i_1} v_{i_2})$. For any match f where $f(v_{i_1}) \prec f(v_{i_2})$, the order $v_{i_1} < v_{i_2}$ only eliminates $\alpha \circ f$ and nothing more is affected, which means that the order only eliminate duplicated results due to automorphism. We have identical results for the remaining orders. Therefore, the results after assigning the order (by order preservation) are still complete, which completes the proof.

2) No Duplicates. Assume that there are two isomorphisms f_1 and f_2 that map (by order preservation) the pattern graph to the same subgraph in the data graph. There must exist an non-trivial automorphism $\alpha \in \mathcal{A}$ s.t. $f_1 = \alpha \circ f_2$. This is impossible since this contradicts the termination condition of the algorithm in [11], where only the identity will be preserved in $\mathcal{A}$.

Given this, we conclude that by assigning the orders via [11], we can break the symmetry which ensures, 1) completeness; 2) no duplicates.