Scalable Distributed Subgraph Enumeration

Longbin Lai[§], Lu Qin[‡§], Xuemin Lin[§], Ying Zhang[‡§], Lijun Chang[§] and Shiyu Yang[§]

[§] The University of New South Wales, Australia
[‡]Centre for QCIS, University of Technology, Sydney, Australia
[§]{llai,lxue,ljchang, yangs}@cse.unsw.edu.au; [‡]{lu.qin,ying.zhang}@uts.edu.au

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 estimates 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 outperforms all other state-of-the-art algorithms by more than one order of magnitude.

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 [24, 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 [29, 25], property generalization for biological networks [23], and is considered to be a key operation for the synthesis of target structures in chemistry [26]. It can also be adopted to illustrate the evolution of social networks [18] and to discover information trends in recommendation networks [21].

Proceedings of the VLDB Endowment, Vol. 10, No. 3

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, 10] are not scalable to large graphs, and researchers have recently explored efficient subgraph enumeration algorithms in distributed environments, such as MapReduce [7] and Pregel [22]. Typically, there are two ways of solving subgraph enumeration - the depth-first search and the join operation. Because the former is hard to parallelize, people tend to use the join algorithm to solve subgraph enumeration in the distributed context.

In MapReduce, the authors in [19] 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 incur $O(10^{18})$ matches of a star of three edges, which would exhaust both the computation and storage in any machine and become a huge bottleneck of the algorithm. Aware of the deficiency of StarJoin, the authors proposed the TwinTwigJoin algorithm [19], 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 can always find an alternative TwinTwigJoin that draws no more cost than the StarJoin.

In Pregel, Shao et al. [28] proposed PSgL that enumerates subgraphs via graph traversal opposed to join operation. 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 [19] 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 can not outperform TwinTwigJoin as shown in [19].

As the state-of-the-art, TwinTwigJoin only guarantees optimality under two constraints: (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 [17]. 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 [33], Dryad [16], and Myria [11]. 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 estimate 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 ER model [19]. 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 avoids computing and materializing partial results in large cliques, and thus further 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 SEED achieves high scalability, and outperforms the state-of-the-art algorithms by more than one order of magnitude in all datasets.

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. We define the following total order among the data nodes as:

Definition 1. (**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 $id(u_i) < id(u_j)$,

where $id(u)$ is the unique identity of node $u \in V(G)$.

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 2. **(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 2). 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 symmetry-breaking technique [10] to avoid duplicate. Specifically, we assign a partial order (denoted as $<$) among some pairs of nodes in the pattern graph P to break symmetry using the technique in [10], then we enforce an *Order-Preservation* constraint in the match (Definiton 2):

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

As this is not the main focus of this paper, we refer the reader to our technical report [20] for more details.

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.

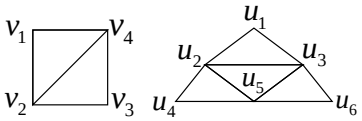


Figure 1: Pattern Graph P (Left) and Data Graph G (Right).

Example 1. Figure 1 shows a pattern graph P, and a data graph G. Figure 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 1.

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 2) 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 [33], Dryad [16] and Myria [11].

Power-Law Random (PR) Graph Model. We model the data graph (N nodes and M edges) as a power-law random (PR) graph according to [4], which is denoted as $\mathcal{G}$. 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 $\mathsf{Pr}_{i,j} \leq 1$ holds, we require $w_{max} \leq \sqrt{wN}$ [31]. As shown in [19], in real-life graphs, although there are nodes with degree larger than $\sqrt{wN}$, the intermediate results from these 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 [31].

In this paper, we estimate the result size based on the PR model in order to evaluate the graph storage mechanism (Section 4) and the cost of the algorithm (Section 5). In the computation, we relax the *Conflict-Freedom* condition of a match (Definiton 2) to allow duplicate nodes and self-loops for ease of analysis, hence the result size calculated is an *upper bound* of the actual value.

Summary of Notations. Table 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	An arbitrary data node and the data node with id i
P	The pattern graph
n, m	The number of nodes and edges in the pattern graph
v, v_i	An arbitrary pattern node and the patter node with id i
P_i	The i[-th] partial pattern, $P_i \subseteq P$
P_i^l, P_i^r	The left and right join patterns while processing 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 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 [19] 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 it must satisfy that $\bigcup_{u \in V(G)} E(G_u) = E(G)$. 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 graph storage $\Phi(G)$, 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* w.r.t. $\Phi(G)$ 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:

$$R(P) = R(p_0) \bowtie R(p_1) \bowtie \cdots \bowtie R(p_t). \quad (1)$$

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 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 have *at least* one join unit as its child, the tree is called a left-deep tree [2]. Otherwise it is called a bushy tree [15]. Note that a left-deep tree is also a bushy tree.

Example 2. Consider the pattern graph P and the decomposition $D(P) = \{q_0, q_1, q_2, q_3\}$ in the left part of Figure 2. 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.

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:

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 1, such that,

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

[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].

[2] 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.

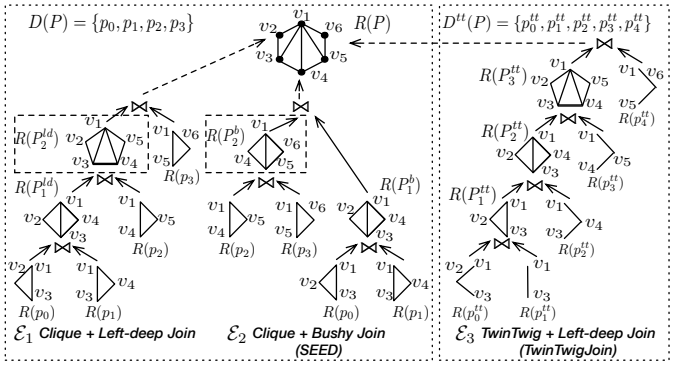


Figure 2: Different Join Trees.

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 $\Phi^0(G) = \{G_u^0 \mid u \in V(G)\}$, where $V(G_u^0) = \{u\} \cup \mathcal{N}(u)$ and $E(G_u^0) = \{(u, u') | u' \in \mathcal{N}(u)\}$ [19]. A star can be the join unit w.r.t. $\Phi^0(G)$, as the matched stars can be independently generated by enumerating the node combinations in $\mathcal{N}(u)$ for each $u \in V(G)$. Aware that a star with many edges can incur 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 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 returns the one with the minimum cost.

In Figure 2, 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 star as the join unit, which can result in severe performance bottlenecks. 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 must process at least $\frac{m}{2}$ rounds, which limits its utilization for complex pattern graph. Finally, the left-deep join framework may produce sub-optimal solution as it only searches for "optimality" among the left-deep joins [17].

3.3 SEED

SEED tackles the issues of TwinTwigJoin by introducing the SCP *graph storage mechanism* and *the optimal bushy join structure*.

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.

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 $(\emptyset; f \cup h)$;

With clique as an alternative, we can avoid processing star where possible, which not only saves the 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 2 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 result size of a triangle is much smaller than that of a two-edge TwinTwig. In addition, $\mathcal{E}_1$ is one round less than $\mathcal{E}_3$. We will detail the SCP storage mechanism in Section 4.

Bushy Join. SEED solves Equation 1 by exploiting the bushy join structure. Specifically, the following join is processed in the i[-th] round:

$$R(P_i) = R(P_i^l) \bowtie R(P_i^r), \tag{2}$$

where P_i^l and P_i^r are called the left and right join patterns regarding P_i, respectively. The left (or right) join pattern can be either a join unit, or a partial pattern processed in an earlier round. Take the execution plan $\mathcal{E}_2$ in Figure 2 as an example. In the first round, P_1 uses two join units p_0 and p_1 as the left and right join patterns, while in the third round, P uses two partial patterns P_1^b and P_2^b.

Compared to TwinTwigJoin, SEED searches the optimal solution among the bushy joins, 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 2 is processed using MapReduce via map^i and reduce^i (line 3). 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 we

fully compute $R(P_i^l)$ 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 [17]. In TwinTwigJoin, an easier-solving left-deep join is applied, which may render sub-optimal solution. In this work, we target on the optimal bushy join plan - a much harder task given the larger searching space [15].

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. 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 denoted as G_u^1 and G_u^2 for $u \in V(G)$, respectively. In order to use clique as the join unit, both mechanisms introduce extra edges to each local graph in $\Phi^0(G)$. Denote $\Delta_u^{(i)} = \mathbb{E}[|E(G_u^i)| - |E(G_u^0)|]$ as the expected number of extra edges introduced by $\Phi^i(G)$ to G_u^0 for $i \in \{1, 2\}$, and let $\Delta_{max}^{(i)} = \max_{u \in V(G)}\{\Delta_u^{(i)}\}$.

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 $E(G_u^1) = E(G_u^0) \cup \{(u', u'') \mid u', u'' \in \mathcal{N}(u) \wedge (u', u'') \in E(G)\}$. 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 $\Phi^1(G) = \{G_u^1 \mid u \in V(G)\}$, p is a join unit w.r.t. $\Phi^1(G)$ if p is a star or a clique.* [3]

Despite $\Phi^1(G)$ is an SCP storage mechanism, it can introduce a large number of 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*

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

[3]Omitted proofs and full proofs of proof sketches can be found in the technical report [20].

Lemma 2 shows that the number of extra edges introduced by $G_{u_i}^1$ 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 issues of $\Phi^1(G)$, we consider a more compact storage mechanism by leveraging the node order. Specifically, we define $\Phi^2(G) = \{G_u^2 \mid u \in V(G)\}$, where $V(G_u^2) = V(G_u^0)$ and $E(G_u^2) = E(G_u^0) \cup \{(u', u'') \mid (u', u'') \in E(G) \wedge u \prec u' \wedge u \prec u''\}$. Compared to G_u^1, G_u^2 only involves 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.

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.*

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

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

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

In Lemma 4, we give an upper bound of $\Delta_{u_i}^{(2)}$, while its value is often much smaller. Apparently, $\Delta_{u_i}^{(2)} = 0$ when $d(u_i) = 1$ and $d(u_i) = \max_{u \in V(G)} d(u)$. In general, we show that $\Delta_{max}^{(2)}$ is much smaller than $\Delta_{max}^{(1)}$. 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 $\Delta_{max}^{(1)}$ and $\Delta_{max}^{(2)}$ in Table 2. It is clear that $\Delta_{max}^{(2)} \ll \Delta_{max}^{(1)}$ in all cases. In the technical report, we further compared $\Delta_u^{(1)}$ with $\Delta_u^{(2)}$ for each data node u using synthetic and real datasets, and the experimental results demonstrate that $\Delta_u^{(2)} \ll \Delta_u^{(1)}$ for all data nodes except those with very small degree in all datasets.

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

Table 2: The number of extra edges introduced by G_u^1 and G_u^2.

Discussion. It is possible to find a different storage mechanism to support more join units. However, in order to do so, we need to involve more edges in each local graph according to Lemma 3, which makes it hard to bound the size of each local graph, and can hence hamper the scalability of the algorithm. Considering such a tradeoff, we adopt $\Phi^2(G)$ as the storage mechanism in this paper, which only supports clique and star as the join units. We will make detailed discussions in the technical report [20]. In the following, we will refer to $\Phi^2(G)$ (resp. G_u^2) simply as $\Phi(G)$ (resp. G_u) if not otherwise specified.

Implementation Details. Given a data graph G, we implement $\Phi(G)$ by constructing G_u for each $u \in V(G)$. Specifically, we first aggregate the neighbor edges of each u to G_u. Then we apply existing methods such as [19, 14, 1] to compute triangles. For each triangle (u_1, u_2, u_3) with $u_1 \prec v_2 \prec v_3$, we add (u_2, u_3) to G_{u_1} as the triangle edge. Given G_u 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 $\Phi(G)$, as shown in the experiment, are relatively small considering the big 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 [19]. 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 [19], and we show that the PR model delivers more realistic estimation.

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:

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

and the cost for processing the partial pattern P_α is:

$$\mathcal{T}(P_\alpha) = |\mathcal{M}(P_\alpha)| + |\mathcal{S}(P_\alpha)| + |\mathcal{R}(P_\alpha)| = 5|R(P_\alpha)| \quad (4)$$

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:

$$\mathcal{C}(\mathcal{E}) = \sum_{i=0}^{t} \mathcal{T}(p_i) + \sum_{i=1}^{t-1} \mathcal{T}(P_i), \quad (5)$$

where $\mathcal{T}(p_i)$ and $\mathcal{T}(P_i)$ are computed via Equation 3 and Equation 4, respectively.

Remark 3. We present the cost model using MapReduce for easy understanding. Nevertheless, the cost model can be applied to other platforms such as Spark, Myria and Dryad with potential modifications. For 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. Myria and Dryad are essentially distributed join systems, in which the data flows of join processing are similar to those of MapReduce, and thus we do not need to modify the cost model for both systems.

5.1 Result-Size Estimation

In order to compute the cost, we need to estimate $|R(P)|$ for a certain P in Equation 3 and Equation 4. 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'). We have:

$$\gamma = \frac{\Sigma_{i=1}^{N} w_i^{\delta+1}}{\Sigma_{i=1}^{N} w_i^{\delta}} \quad (6)$$

(Case 2). $v' \in V(P^{(1)})$. In this case, a new edge is added between two existing nodes in $P^{(1)}$. We have:

$$\gamma = \rho \times \frac{\Sigma_{i=1}^{N} w_i^{\delta+1}}{\Sigma_{i=1}^{N} w_i^{\delta}} \times \frac{\Sigma_{j=1}^{N} w_j^{\delta'+1}}{\Sigma_{j=1}^{N} w_j^{\delta'}} \quad (7)$$

The derivations of Equation 6 and Equation 7 are lengthy, hence have been included in the technical report [20].

Given Equation 6 and Equation 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 6 iteratively to compute the size of the tree. Finally, we apply Equation 7 iteratively as we extend the non-tree edges. Note that, given a graph G, the γ calculated by Equation 6 or Equation 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 3: 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 2 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 [19], 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 plans regarding 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 [19], 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 3. 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 [19], 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:

$$\mathcal{C}(\mathcal{E}_{P_\alpha}) = \begin{cases} 0, \ P_\alpha \text{ is a join unit,} \\ \min\limits_{P^l_\alpha \subset P_\alpha} \{\mathcal{C}(\mathcal{E}_{P^l_\alpha}) + \mathcal{T}(P^l_\alpha) + \mathcal{C}(\mathcal{E}_{P^r_\alpha}) + \mathcal{T}(P^r_\alpha)\}, \text{ otherwise.} \end{cases} \tag{8}$$

where $P^r_\alpha = P_\alpha \setminus P^l_\alpha$, $\mathcal{T}(P^l_\alpha)$ and $\mathcal{T}(P^r_\alpha)$ are computed via Equation 3 or Equation 4 depending on whether they are join units or partial patterns. The optimal partial execution $\mathcal{E}_{P_\alpha}$ is computed by minimizing the sum of the cost of $\mathcal{E}_{P^l_\alpha}$ and $\mathcal{E}_{P^r_\alpha}$, and the cost of processing the join $R(P_\alpha) = R(P^l_\alpha) \bowtie R(P^r_\alpha)$ (that is $\mathcal{T}(P^l_\alpha) + \mathcal{T}(P^r_\alpha)$). Note that $\mathcal{C}(\mathcal{E}_{P_\alpha}) = 0$ if P_α is a join unit, as we do not need to compute $R(P_\alpha)$ using joins.

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 $(P_\alpha, \mathcal{T}, \mathcal{C}, P^l_\alpha, P^r_\alpha)$, where $\mathcal{T}$ is an auxiliary cost computed via either Equation 3 or Equation 4, $\mathcal{C}$ is the so far best cost $\mathcal{C}(E_{P_\alpha})$ while evaluating P_α, P^l_α is the left-join pattern when the current best cost is obtained, and $P^r_\alpha = P_\alpha \setminus P^l_\alpha$, as no overlap is considered. The hash map is indexed by P_α, so that we can access one specific item I for a certain P_α via $\mathcal{H}_{P_\alpha}(I)$, where $I \in \{\mathcal{T}, \mathcal{C}, P^l_\alpha, P^r_\alpha\}$.

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^l_α ⊂ P_α s.t. P^l_α and P^r_α = P_α \ P^l_α are
               connected do
9                  C ← H_{P^l_α}(C) + H_{P^l_α}(T) + H_{P^r_α}(C) + H_{P^r_α}(T);
                   if C < H_{P_α}(C) then
10                     H_{P_α}(C) ← C;
11                     H_{P_α}(P^l_α) ← P^l_α; H_{P_α}(P^r_α) ← P^r_α;
12 Compute the optimal execution plan E_o via H;
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^l_α (and $P^r_\alpha = P_\alpha \setminus P^l_\alpha$), and for each P^l_α where P^l_α and P^r_α are both connected, we compute $\mathcal{C}(\mathcal{E}_{P_\alpha})$ via Equation 8 (line 9). In this way, we finally find the P^l_α to minimize $\mathcal{C}(\mathcal{E}_{P_\alpha})$, and update the entry of P_α by setting $\mathcal{H}_{P_\alpha}(\mathcal{C})$, $\mathcal{H}_{P_\alpha}(P^l_\alpha)$ and $\mathcal{H}_{P_\alpha}(P^r_\alpha)$ 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^l_α and P^r_α and repeat the procedure recursively on P^l_α and P^r_α until $P^l_\alpha = \emptyset$. 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.*

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:*

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

Example 3. Observe that there are overlaps among the join units in Figure 2. 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 $P^r_\alpha = P_\alpha \setminus P^l_\alpha$ in line 8, we further enumerate all possible P^{r*}_α, where P^{r*}_α are all *connected* structures formed by adding any subset of $E(P^l_\alpha)$ to P^r_α. 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^l_α (resp. P^r_α) (P'_l and P^r_α may overlap), we define the redundant node as:

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

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

$$V_c(P^l_\alpha, P^r_\alpha) = \big(V(P^l_\alpha) \cap V(P^r_\alpha)\big) \setminus V_r$$

$$E_c(P^l_\alpha, P^r_\alpha) = \{(v, v') \mid (v, v') \in E(P_\alpha) \wedge v, v' \in V_c(P^l_\alpha, P^r_\alpha)\}.$$

Example 4. In Figure 3, we show a partial pattern P_α and its left-join (resp. right-join) pattern P^l_α (resp. P^r_α). Clearly, v_4 is a redundant node since $P_\alpha = P^l_\alpha \cup (P^r_\alpha \setminus v_4)$, and we have $V_c(P^l_\alpha, P^r_\alpha) = \{v_2, v_3\}$ and $E_c(P^l_\alpha, P^r_\alpha) = \{(v_2, v_3)\}$.

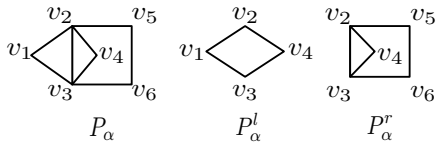


Figure 3: The redundant node, cut nodes and cut edges.

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 Sketch: 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 $R(\widetilde{P_i^r})$ by joining the cut edges to $R(P_i^r)$. Clearly, $\mathcal{C}(\widetilde{\mathcal{E}}) = \mathcal{C}(\widetilde{\mathcal{E}_1}) + \mathcal{C}(\widetilde{\mathcal{E}_2})$. According to Lemma 6, we have:

$$|R(\widetilde{P_i^r})| \le |R(P_i^r)|. \tag{9}$$

Using this fact, we can respectively prove that $\mathcal{C}(\widetilde{\mathcal{E}_1}) \le \Theta(\mathcal{C}(\mathcal{E}_o))$ and $\mathcal{C}(\widetilde{\mathcal{E}_2}) \le \Theta(\mathcal{C}(\mathcal{E}_0))$, and thus $\mathcal{C}(\widetilde{\mathcal{E}}) \le \Theta(\mathcal{C}(\mathcal{E}_o))$. On the other hand, $\mathcal{C}(\widetilde{\mathcal{E}}) > \mathcal{C}(\mathcal{E}_o)$. Therefore, this lemma holds. □

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

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

Proof Sketch: Given $P_i = P_i^l \cup P_i^r$ for some $1 \le i \le |D_o| - 1$ in $\mathcal{E}_o$, we denote $\widetilde{P_i^r} = P_i^r \cup E_c(P_i^l, P_i^r)$, and $\widetilde{P_i^{r*}} = (P_i \setminus P_i^l) \cup E_c(P_i^l, (P_i \setminus P_i^l))$. We know that $\widetilde{\mathcal{E}}$ will process the join $R(P_i) = R(P_i^l) \bowtie R(\widetilde{P_i^r})$. We then **claim** that $\widetilde{\mathcal{E}}$ must be within the searching space of the all-or-nothing strategy. Given P_i^l while computing P_i, the all-or-nothing strategy will consider $\widetilde{P_i^{r*}}$ as the right-join pattern via the "all" strategy. It suffices to prove the claim by showing that $\widetilde{P_i^r} = \widetilde{P_i^{r*}}$.

As $\widetilde{\mathcal{E}}$ must be within the searching space of the all-or-nothing strategy, and $\mathcal{E}_o'$ is the optimal solution in the space, it is immediate that $\mathcal{C}(\mathcal{E}_o') \le \mathcal{C}(\widetilde{\mathcal{E}})$. □

THEOREM 1. *If $\mathcal{C}(\mathcal{E}_o) \ge \Theta(M)$ and there is no redundant node w.r.t. $P_i = P_i^l \cup P_i^r$ for all $1 \le i \le |D_o| - 1$ 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, $\mathcal{C}(\mathcal{E}_o) \ge \Theta(M)$. 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 5, which alone contributes to $\binom{943}{5} \approx 6 \times 10^{12}$ matches of 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-overlapped (by nodes) cliques in the data graph G as a preprocessing step. In query processing, when a k-clique 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-overlapped cliques, followed by discussing the way of compressing the matches of p_k. Finally, we introduce how to process joins with the compressed results.

Clique Precomputation. As a preprocessing step, we compute a set of non-overlapped cliques $S = \{K_1, K_2, \ldots, K_s\}$ in the data graph G. We introduce the greedy algorithm to compute S in the following. 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 [32]). 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 or $S = \emptyset$. The space used to index the cliques is small since we only need to index the nodes in each clique. We will show in the experiment that the overheads of clique precomputation are relatively small, and it does improve the performance of SEED, especially when the data graph contains large cliques.

Online Clique Compression. During query processing, suppose a k-clique p_k is involved in the join, 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 $V_u^c = \{u' | u' \in V(G_u^2), \mathcal{M}(u') = \mathcal{M}(u)\}$ is the set of nodes in G_u^2 that belong to the same clique with u in S, and $V_u^n = V(G_u^2) \setminus V_u^c$. Specifically, let $V_u^c = \emptyset$ and $V_u^n = V(G_u)$ when $\mathcal{M}(u) = \emptyset$. 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 $f^n(v_i^n) = u_i^n$ for each $v_i^n \in f^n.V$ and $u_i^n \in f^n.U$. Here, the following

Algorithm 3: map^i(**key**: $\emptyset$; **value**: either compressed matches $(f^c, f^n) \in R(P_i^l)$ and $(h^c, h^n) \in R(P_i^r)$ or $G_u \in \Phi(G)$)

```
1  V_join ← V(P_i^l) ∩ V(P_i^r);
2  if P_i^l is a star then genJoinUnit(P_i^l, G_u, V_join);
3  else if P_i^l is a clique then genCompressedClique(P_i^l, G_u, V_join);
4  else partialExpansion(f^c, f^n, V_join);
5  if P_i^r is a star then genJoinUnit(P_i^r, G_u, V_join);
6  else if P_i^r is a clique then genCompressedClique(P_i^r, G_u, V_join);
7  else partialExpansion(h^c, h^n, V_join);
8  function genCompressedClique(p_k, G_u, V_join)
9  F ← compressedClique(p_k, G_u);
10 foreach (f^c, f^n) ∈ F do
11     partialExpansion(f^c, f^n, V_join);
12 function partialExpansion(f^c, f^n, V_join)
13 V^n_join = {v^n_1, v^n_2, ..., v^n_p} ← f^n.V ∩ V_join;
14 U^n_join ← {f^n(v^n_1), f^n(v^n_2), ..., f^n(v^n_p)};
15 V^c_join ← f^c.V ∩ V_join;
16 foreach U^c_join ⊆ f^c.U s.t. |U^c_join| = |V^c_join| do
17     key ← U^c_join ∪ U^n_join;
18     f^c_out ← (f^c.V \ V^c_join, f^c.U \ U^c_join);
19     f^n_out ← (f^n.V ∪ V^c_join, f^n.U ∪ U^c_join);
20     output (key; (f^c_out, f^n_out));
```

four constraints must be satisfied[4]: (1) $|f^c.V| \leq |f^c.U|$, (2) $f^c.V \cup f^n.V = V(p_k)$, (3) $f^n.U \subseteq V_u^n$ and $f^n.U = \emptyset$ if $f^n.V = \emptyset$, otherwise the nodes in $f^n.U$ must form a clique in G_u^2, and (4) $f^c.U \subseteq V_u^c$ and every node in $f^c.U$ is adjacent to all nodes in $f^n.U$ in G_u^2 if $f^n.U \neq \emptyset$. 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. The detailed procedure to generate all compressed k-cliques from G_u^2 is shown in our technical report.

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). 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 the technical report). 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 3. 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 compressed 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 new map^i procedure is shown in Algorithm 1 to replace map^i in Algorithm 1. For reduce^i, we follow the same principle to process joins by keeping the compressed part as much as possible. The detailed algorithm for reduce^i is shown in the technique report.

[4]To remove duplicates, we should also ensure that $u \in f^n.U$. However, in this section, we remove the constraint for ease of discussion.

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 4. 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. We set the block size in HDFS to 128MB, the data replication factor of HDFS to 3, the I/O sort size to 512MB.

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

Table 4: Amazon virtual instance configurations

Datasets. We tested six real-world data graphs (see Table 5). 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). 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). 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

Table 5: 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+O: SEED with clique-compression (Section 7).
- TT: The TwinTwigJoin algorithm with all optimizations [19].
- PSgL: The Pregel-based subgraph enumeration algorithm with all optimizations proposed by Shao et al. [28].

All algorithms were compiled with Java 1.7. We implemented SEED and all its variants, and TT with Hadoop 2.6.0. All algorithms except PSgL ran on the Yarn framework. The authors of [28] kindly provided the codes for PSgL, implemented with Hadoop 1.2.0 based on the old MapReduce framework. The performance gap between Yarn and old MapReduce is very small, hence the comparisons between PSgL and the other algorithms are 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 denoted 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 4 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 symmetry breaking (Remark 1) under each query graph. Here, we have considered all queries ($q_1 - q_4$, q_7) except triangle from the state-of-the-art works [28, 19] for fair comparisons. Note

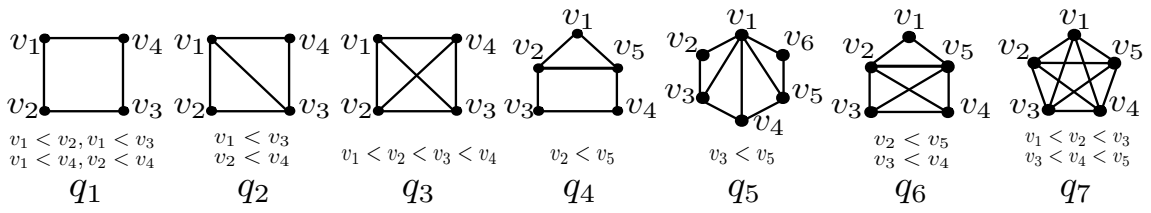


Figure 4: Queries

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	7658

Table 6: Cost comparisons while enumerating q_5 on *yt* with SEED and SEED-LD (in millions).

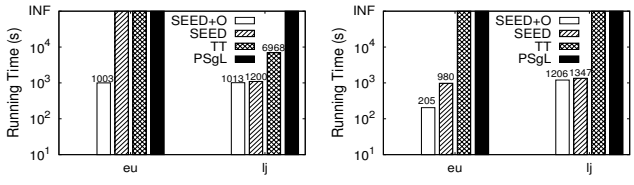
that triangle enumeration is used in this paper as a preprocessing step to construct the SCP storage. We add the query q_5 and q_6 to further demonstrate the advantages of our proposed techniques.

Auxiliary Experiments. We have presented the auxiliary experiments in the technical report [20], including the local-graph statistics (Section 4), the effect of overlapping join units (Section 6.2), and the comparisons of MapReduce and Spark.

Exp-1: Bushy vs. Left-deep. We compare the performance of SEED and SEED-LD using query q_5 on *yt* to demonstrate the advantage of the bushy join plan. The plans $\mathcal{E}_1$ and $\mathcal{E}_2$ shown in Figure 2 illustrate the optimal execution plans for SEED-LD and SEED, respectively. Table 6 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. The output of reduce[3] is not shown, as it is the final result and excluded in the cost. Clearly, SEED, with smaller cost, outperforms SEED-LD by more than an order of magnitude. Here, we only show the results on *yt*, as SEED-LD runs out of time on all other datasets.

As we mentioned in Remark 4, the plans $\mathcal{E}_1$ and $\mathcal{E}_2$ in Figure 2 are also the optimal execution plans computed using the ER model and the PR model, respectively. In Table 6, 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 join plans computed using the ER model and PR model are different.



(a) Query: q_6 (b) Query: q_7

Figure 5: The effects of the proposed techniques.

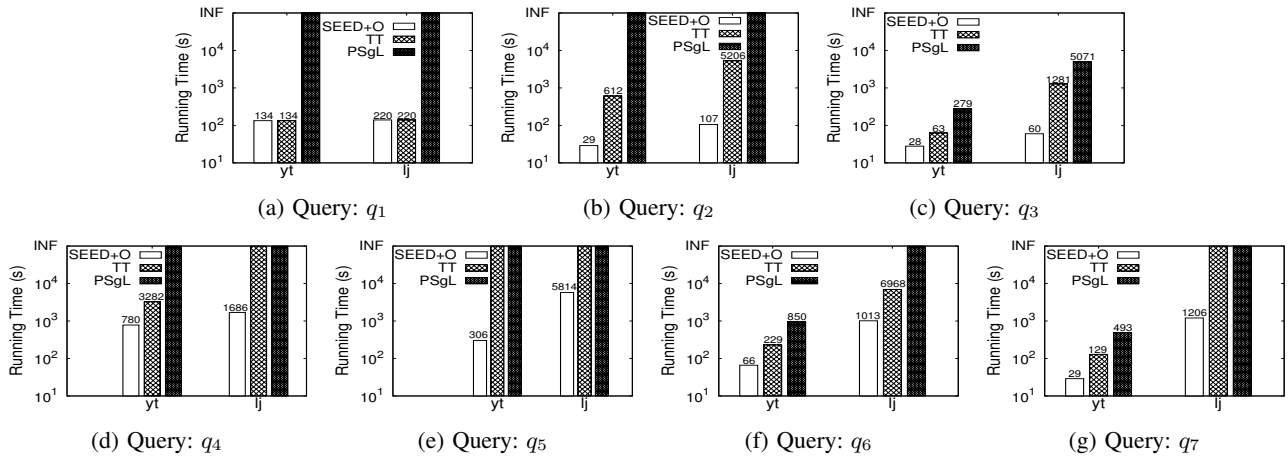
Exp-2: The effects of the proposed techniques. To show the effects of the proposed techniques, we evaluated the performance of SEED+O, SEED, TT and PSgL by querying q_4 and q_6 on *eu* and *lj*, and reported the results in Figure 5(a)-Figure 5(b). Observe that the baseline SEED already dominates the 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 5), 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 5) 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 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 for clear comparisons. Next we would compare the algorithms using all queries.

Exp-3: Test all queries. We compared SEED+O with TT and PSgL - by enumerating all queries on *yt* and *lj*, and reported the results in Figure 6(a)-Figure 6(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 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 [19]. 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 small datasets.

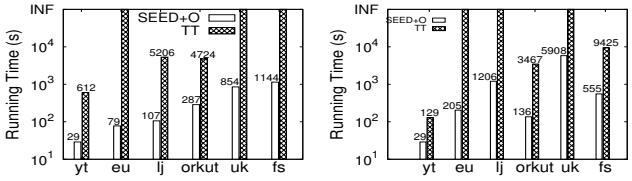
Exp-4: Vary Datasets. We compared SEED+O with TT by querying q_2 and q_7 on all datasets in order to show the advantages of SEED+O regarding different data properties. The results are shown in Figure 7(a)-Figure 7(b). In all tests, SEED+O significantly outperforms TT, with the performance gain varying from an order of magnitude to over $50\times$ (enumerating q_2 on *lj*). Specifically,



(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 6: Test all queries.



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

Figure 7: Vary Datasets.

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 large data graphs due to the use of clique as the join unit, and the optimal bushy join plan with overlapping join units.

Exp-5: 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(a) and Figure 8(b) respectively. The running time of both algorithms increases as the size of the graph. When the graph size is over 60%, TT fails to process q_2 in the allowed time. The test shows the high scalability of our SEED+O algorithm.

Exp-6: 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(c) and Figure 8(d), respectively. In Figure 8(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-7: 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 9(a)-Figure 9(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 9(b), even when 14 slave nodes are deployed, SEED+O is the only algorithm that can process q_7 on *lj*.

9. RELATED WORK

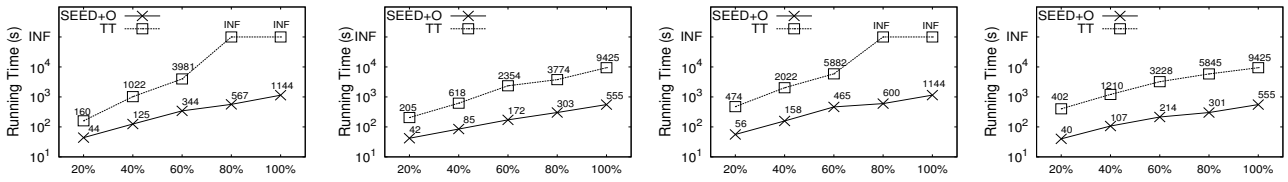
Subgraph Matching. Most subgraph matching approaches work in labeled context, where certain labels are assigned to nodes (and/or edges). For example, Shang et al. [27] propose an algorithm to search from nodes with infrequent labels in order to utilize the filtering power as early as possible. Node labels in the neighborhood are used to filter unexpected candidates in [13] and [34]. In [12], the authors observe that a good matching order can significantly improve the performance of subgraph query. Subgraph enumeration in a centralized environment is also studied in exact and approximate settings. The exact solutions including [3] and [10] are not scalable for handling large data graphs. The approximate solutions [2, 9, 35] 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. [35] introduced a parallel color coding method for subgraph counting. Sun et al. [30] proposed a subgraph matching algorithm that uses node filtering to handle labeled graphs in the Trinity memory cloud. Recently, Shao et al. [28] 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 [19].

Subgraph Enumeration in MapReduce. Subgraph enumeration in MapReduce has attracted a lot of interests. Afrati et al. [1] proposed multiway join in MapReduce to handle subgraph enumeration. The TwinTwigJoin algorithm was proposed in [19], which has proven to be instance optimal in the left-deep-join framework. However, using TwinTwig as the join unit is still inefficient when processing large-degree nodes and the left-deep join plan may result in non-optimal solutions.

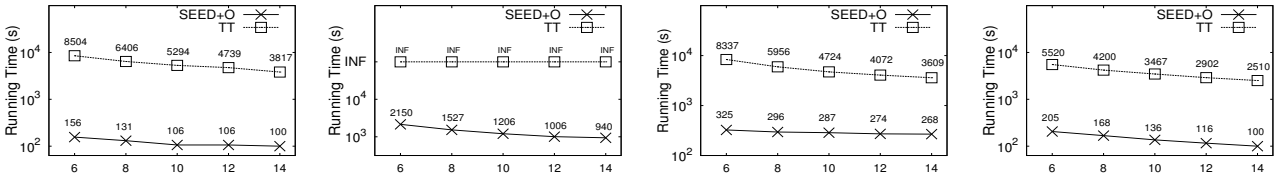
10. CONCLUSIONS

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 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.



(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: Vary Graph Properties



(a) On lj: q_2 (b) On lj: q_7 (c) On orkut: q_2 (d) On orkut: q_7

Figure 9: Vary Slave Nodes

Acknowledgements. We would like to thank all the reviewers for their dedicated work. Lu Qin is supported by ARC DE140100999 and ARC DP160101513. Xuemin Lin is supported by ARC DP140103578 and ARC DP150102728. Lijun Chang is supported by ARC DE150100563 and ARC DP160101513. Ying Zhang is supported by ARC DE140100679.

11. REFERENCES

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

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

[11] 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*.

[12] 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.

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

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

[15] 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.

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

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

[18] 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.

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

[20] L. Lai, L. Qin, Y. Zhang, X. Lin, and L. Chang. Scalable distributed subgraph enumeration. Technical report, 2016. "Available as `ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/201606.pdf`".

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

[22] 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.

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

[24] 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.

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

[26] 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.

[27] H. Shang, Y. Zhang, X. Lin, and J. X. Yu. Taming verification hardness: An efficient algorithm for testing subgraph isomorphism. *Proc. VLDB Endow.*, 1(1):364–375, 2008.

[28] 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.

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

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

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

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

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

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

[35] 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.