Fast and Robust Distributed Subgraph Enumeration

Xuguang Ren
Griffith University, Australia
x.ren@griffith.edu.au

Junhu Wang
Griffith University, Australia
j.wang@griffith.edu.au

Wook-Shin Han
POSTECH, Public of Korea
wshan@dblab.postech.ac.kr

Jeffrey Xu Yu
The Chinese University of Hong Kong
yu@se.cuhk.edu.hk

ABSTRACT

We study the subgraph enumeration problem under distributed settings. Existing solutions either suffer from severe memory crisis or rely on large indexes, which makes them impractical for very large graphs. Most of them follow a synchronous model where the performance is often bottlenecked by the machine with the worst performance. Motivated by this, in this paper, we propose RADS, a **R**obust **A**synchronous **D**istributed **S**ubgraph enumeration system. RADS first identifies results that can be found using single-machine algorithms. This strategy not only improves the overall performance but also reduces network communication and memory cost. Moreover, RADS employs a novel ***r****egion-grouped* ***m****ulti-round* ***e****xpand* ***v****erify* & ***f****ilter* framework which does not need to shuffle and exchange the intermediate results, nor does it need to replicate a large part of the data graph in each machine. This feature not only reduces network communication cost and memory usage, but also allows us to adopt simple strategies for memory control and load balancing, making it more robust. Several optimization strategies are also used in RADS to further improve the performance. Our experiments verified the superiority of RADS to state-of-the-art subgraph enumeration approaches.

PVLDB Reference Format:
Xuguang Ren, Junhu Wang, Wook-Shin Han, Jeffrey Xu Yu. Fast and Robust Distributed Subgraph Enumeration. *PVLDB*, 12(11): 1344-1356, 2019.
DOI: https://doi.org/10.14778/3342263.3342272

Keywords

Distributed System, Asynchronous, Subgraph Enumeration

1. INTRODUCTION

Subgraph enumeration is the problem of finding all occurrences of a query graph in a data graph. Its solution is the basis for many other algorithms and it finds numerous applications[11]. This problem has been well studied under single machine settings [9][17]. However in the real world, the data graphs are often fragmented and distributed across different sites. This phenomenon highlights the importance of distributed systems of subgraph enumeration. Also, the increasing size of modern graph makes it hard to load the whole graph into memory, which further strengthens the requirement of distributed subgraph enumeration.

In recent years, many approaches have been proposed [1, 20, 12, 13, 5, 4, 16]. However all existing approaches are facing one or more of the following problems: (1) *Memory crisis.* Huge numbers of intermediate results are generated in [20, 12, 13]. Lacking of effective pruning and compression techniques results in a memory crisis to those approaches. Approaches like [1, 5, 4] replicate large parts of the data graph on each machine, which may be impractical for busy or low-end computer clusters. (2) *Heavy network communication.* The large intermediate results of [20, 12, 13] need to be shuffled through the network. A big network latency may be generated when the data graph is distributed across different sites. (3) *Heavy index.* The index files of [16] can be many times larger than the data graph as shown in Table 2 of Section 8, and computing/maintaining such big indexes can be very expensive. (4) *Synchronization delay.* Most of the current systems are synchronous, hence they suffer from synchronization delay, making the overall performance equivalent to that of the slowest machine.

Different approaches may be facing different problems but may complement each other. One may argue that we can design a better system by simply stacking their successful ideas together. However this is very challenging across different system designs. For example, the compression strategy proposed in [16] cannot be used in [13] since [13] requires the intermediate results to be grouped by joining keys. The asynchronous method used in [4] is not suitable for join-based approaches [12, 13].

In this paper, we present RADS, a **R**obust **A**synchronous **D**istributed **S**ubgraph enumeration system. RADS is asynchronous, index free, with light communication cost, and also significantly reduces and compresses intermediate results. RADS is a solution which employs a multi-round framework and comprises several key ideas and optimization strategies. To be specific, we make the following contributions:

(1) We propose a method to identify embeddings that can be found on each local machine independent of other machines, and use single-machine algorithm to find

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.3342272

them. This strategy not only improves the overall performance, but also reduces network communication and memory cost.

(2) RADS employs a framework of ***region-grouped multi-round expand verify & filter*** where the key ideas are (i) to communicate undetermined edges and verification results instead of exchanging intermediate results, where the size of former is much smaller than that of the latter; (ii) to exchange parts of data graph in a region-grouped multi-round manner where the peak size of data graphs replicated in each machine is reduced.

(3) We propose effective memory control strategies to minimize the chance of memory crash, making our system more robust. Our strategy also facilitates workload balancing.

(4) We propose optimization strategies to further improve the performance. These include: (i) a set of rules to compute an efficient execution plan; and (ii) a dynamic data structure to compactly store intermediate results.

(5) We conduct extensive experiments which demonstrate that our system is not only faster than existing solutions (except for some queries using [16], which relies on heavy indexes), but also more robust.

Paper Organization We first introduce the preliminaries in Section 2. In Section 3, we present the architecture of our system RADS. The core framework **R-Meef** of RADS is given in Section 4. In Section 5, we present algorithms for computing the execution plan. In Section 6, we present the embedding trie data structure to compress our intermediate results. Our memory control strategy is given in Section 7. We present our experiments in Section 8, and discuss related works in Section 9. We conclude the paper in Section 10.

2. PRELIMINARIES

Data Graph & Query Graph Both the data graph and query graph (a.k.a query pattern) are assumed to be unlabeled, undirected, and connected graphs. We use $G = (V_G, E_G)$ and $P = (V_P, E_P)$ to denote the data graph and query graph respectively, where V_G and V_P are the vertex sets, and E_G and E_P are the edge sets. We will use *data* (resp. *query*) *vertex* to refer to vertices in the data (resp. query) graph. Generally, for any graph g, we use V_g and E_g to denote its vertex set and edge set respectively, and for any vertex v in g, we use $adj(v)$ to denote v's neighbour set and use $deg(v)$ to denote the degree of v.

Subgraph Isomorphism Given a data graph G and a query pattern P, P is subgraph isomorphic to G if there exists an injective function f: $V_P \rightarrow V_G$ such that for any edge $(u_1, u_2) \in E_P$, there exists an edge $(f(u_1), f(u_2)) \in E_G$. The injective function is also known as an *embedding* of P in G, and it can be represented as a set of vertex pairs (u, v) where $u \in V_P$ is mapped to $v \in V_G$. We will use $\mathbb{R}_G(P)$ to denote the set of all embeddings of P in G.

The problem of subgraph enumeration is to find the set $\mathbb{R}_G(P)$. In the literature, subgraph enumeration is also referred to as subgraph isomorphism search [14][9][17] and subgraph listing [11][20].

Partial Embedding A *partial embedding* of graph P in graph G is an embedding in G of a vertex-induced subgraph of P.

Symmetry Breaking A symmetry breaking method based on automorphism is conventionally used to reduce duplicate embeddings [7]. As a result the data vertices in the final embeddings should follow a preserved order of the query vertices. We apply this technique in this paper by default and we will specify the preserved order when necessary.

Graph Partition & Storage Given a data graph G and m machines $\{M_1, \dots, M_m\}$ in a distributed environment, a partition of G is denoted $\{G_1, G_2, \dots, G_m\}$ where G_t is the partition located in the t^{th} machine M_t. In this paper, we assume each partition is stored as an adjacency-list. For any data vertex v, we assume its adjacency-list is stored in a single machine M_t and we say v is owned by M_t (or v resides in M_t). We call v a foreign vertex of M_t if v is not owned by M_t.

For any v owned by M_t, we call v a *border* vertex if at least one of its neighbors resides in other machines than M_t. Otherwise we call it a *non-border* vertex. We use $V^b_{G_t}$ to denote the set of all border vertices in M_t.

3. RADS ARCHITECTURE

In this section, we present the architecture of RADS as shown in Figure 1.

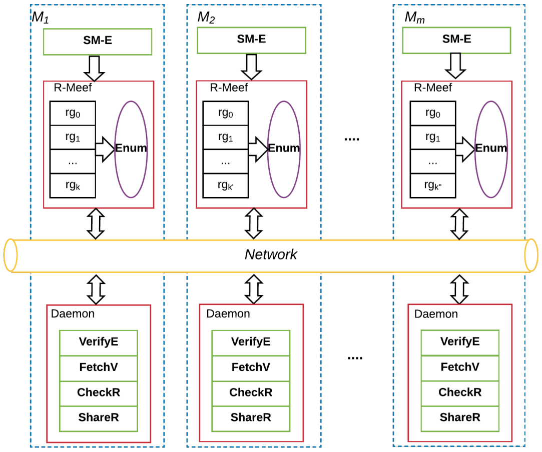


Figure 1: RADS Architecture

Given a query pattern P, within each machine, RADS first launches a process of single-machine enumeration (SM-E) and a **Daemon** thread, simultaneously. After SM-E finishes, RADS launches a **R-Meef** thread subsequently. Note that the **R-Meef** threads of different machines may start at different time.

- **Single-Machine Enumeration** The idea of SM-E is to try to find a set of local embeddings using a single-machine algorithm, such as TurboIso[9], which does not involve any distributed processing. The subsequent distributed process only has to find the remaining embeddings. This strategy can not only boost the overall enumeration efficiency but also significantly reduce the memory cost and communication cost of the subsequent distributed process. Moreover the local embeddings can be used to estimate the space cost of a *region group*, which will help to effectively control the memory usage (see Section 7).

We first define the concepts of *border distance* and *span*, which will be used to identify embeddings that can be found by SM-E.

Definition 1. *Given a graph partition G_t and data vertex v in G_t, the* border distance *of v w.r.t G_t, denoted $BD_{G_t}(v)$, is the minimum shortest distance between v and any border vertex of G_t, that is*

$$BD_{G_t}(v) = \min_{v' \in V^b_{G_t}} dist(v, v') \tag{1}$$

where $dist(v, v')$ is the shortest distance between v and v'.

Definition 2. *Given a query pattern P, the* span *of query vertex u, denoted $Span_P(u)$, is the maximum shortest distance between u and any other vertex of P, that is*

$$Span_P(u) = \max_{u' \in V_P} dist(u, u') \tag{2}$$

Proposition 1. *Given a data vertex v of G_t and a query vertex u of P, if $Span_P(u) \leq BD_{G_t}(v)$, then there will be no embedding f of P in G such that $f(u) = v$ and $f(u')$ is not owned by M_t, where $u' \in P$, $u' \neq u$.*

Proposition 1 states that if the border distance of v is not smaller than the span of query vertex u, there will be no cross-machine embeddings (i.e., embeddings where the query vertices are mapped to data vertices residing in different machines) which map u to v. The proof of Proposition 1 can be found in the extended version of this paper [18].

Let u_{start} be the starting query vertex (namely, the first query vertex to be mapped) and $C(u_{start})$ be the candidate vertex set of u_{start} in G_t. Let $C_1(u_{start}) \subseteq C(u_{start})$ be the subset of candidates whose border distance is no less than the span of u_{start}. According to Proposition 1, all embeddings that map u_{start} to a vertex in $C_1(u_{start})$ can be found using a single-machine subgraph enumeration algorithm over G_t, independent of other machines. In RADS, the candidates in $C_1(u_{start})$ will be processed by SM-E, and the other candidates will be processed by the subsequent distributed process. The SM-E process is simple, and we will next focus on the distributed process. For presentation simplicity, from now on when we say a candidate vertex of u_{start}, we mean a candidate vertex in $C(u_{start}) - C_1(u_{start})$, unless explicitly stated otherwise.

The distributed process consists of a daemon thread and a subgraph enumeration thread **R-Meef**:

- **Daemon Thread** listens to requests from other machines and supports four functionalities:
  (1) verifyE is to return the *edge verification* results for a given request consisting of vertex pairs. For example, given a request $\{(v_0, v_1), (v_2, v_3)\}$ posted to M_1, M_1 will return $\{true, false\}$ if (v_0, v_1) is an edge in G_1 while (v_2, v_3) is not.
  (2) fetchV is to return the adjacency-lists of the requested vertices of the data graph. The requested vertices sent to machine M_i must reside in M_i.
  (3) checkR is to return the number of unprocessed *region groups* (which is a group of candidate data vertices of the starting query vertex, see Section 4) of the local machine (i.e., the machine on which the thread is running).
  (4) shareR is to return an unprocessed region group of the local machine to the requester machine. *shareR* will also mark the region group sent out as processed.
- **R-Meef Thread** is the core subgraph enumeration thread. When necessary, the local **R-Meef** thread sends *verifyE* requests and *fetchV* requests to the Daemon threads of other machines, and the other machines respond to these requests accordingly. We present the details of **R-Meef** in next Section.

Once a local machine finishes processing its own region groups, it will broadcast a *checkR* request to the other machines. Upon receiving the numbers of unfinished region groups from other machines, it will send a *shareR* request to the machine with the maximum number of unprocessed region groups. Once it receives a region group, it will process it on the local machine. *checkR* and *shareR* are for load balancing purposes only, and they will not be discussed further in this paper.

4. THE R-Meef FRAMEWORK

Before presenting the details of the **R-Meef** framework, we need the following definitions.

Definition 3. *Given a partition G_t of data graph G located in machine M_t and a query pattern P, an injective function f_{G_t}: $V_P \to V_G$ is called an* embedding candidate (EC) *of P w.r.t G_t if for any edge $(u, u') \in E_P$, there exists an edge $(f_{G_t}(u), f_{G_t}(u')) \in E_{G_t}$ provided either $f_{G_t}(u) \in V_{G_t}$ or $f_{G_t}(u') \in V_{G_t}$.*

We use $\widetilde{\mathbb{R}}_{G_t}(P)$ to denote the set of ECs of P *w.r.t* G_t. Note that for an EC f_{G_t} and a query vertex u, $f_{G_t}(u)$ is not necessarily owned by G_t. That is, the adjacency-list of $f_{G_t}(u)$ may be stored in other machines. For any query edge (u, u'), an EC only requires that the corresponding data edge $(f_{G_t}(u), f_{G_t}(u'))$ exists if at least one of $f_{G_t}(u)$ and $f_{G_t}(u')$ resides in G_t. Therefore, an EC may not be an embedding. Intuitively, the existence of the edge $(f_{G_t}(u), f_{G_t}(u'))$ can only be verified in G_t if one of its end vertices resides in G_t. Otherwise the existence of the edge cannot be verified in M_t, and we call such edges *undetermined* edges.

Definition 4. *Given an EC f_{G_t} of query pattern P, for any edge $(u, u') \in E_P$, we say $(f_{G_t}(u), f_{G_t}(u'))$ is an* undetermined edge *of f_{G_t} if neither $f_{G_t}(u)$ nor $f_{G_t}(u')$ is in G_t.*

Example 1. *Consider a partition G_t of a data graph G and a triangle query pattern P where $V_P = \{u_0, u_1, u_2\}$. The mapping $f_{G_t} = \{(u_0, v_0), (u_0, v_1), (u_0, v_2)\}$ is an EC of P in G w.r.t G_t if $v_0 \in V_{G_t}$, $v_1 \in adj(v_0)$ and $v_2 \in adj(v_0)$ and neither v_1 nor v_2 resides in G_t. (v_1, v_2) is an undetermined edge of f_{G_t}.*

Obviously if we want to determine whether f_{G_t} is actually an embedding of the query pattern, we have to verify its undetermined edges in other machines. For any undetermined edge e, if its two end vertices reside in two different machines, we can use either of them to verify whether $e \in E_G$ or not. To do that, we need to send a *verifyE* request to one of the machines.

Note that it is possible that an undetermined edge is shared by multiple ECs. To reduce network traffic, we do not send *verifyE* requests once for each individual EC, instead, we build an *edge verification index* (EVI) and use it to identify ECs that share undetermined edges. We assume each EC is assigned an ID (to be discussed in Section 6).

Definition 5. *Given a set $\widetilde{\mathbb{R}}_{G_t}(P)$ of ECs, the* edge verification index (EVI) *of $\widetilde{\mathbb{R}}_{G_t}(P)$ is a key-value map I where*

(1) for any tuple $(e, IDs) \in I$, the key e is a vertex pair (v, v'); and the value IDs is the set of IDs of the ECs in $\widetilde{\mathbb{R}}_{G_t}(P)$ of which e is an undetermined edge.

(2) for any undetermined edge e of $f_{G_t} \in \widetilde{\mathbb{R}}_{G_t}(P)$, there exists a unique tuple in I with e as the key and the ID of f_{G_t} in the value.

Intuitively, the EVI groups the ECs that share each undetermined edge together. It is straightforward to see:

Proposition 2. *Given data graph G, query pattern P and an edge verification index I, for any $(e, IDs) \in I$, if $e \notin E_G$, then none of the ECs corresponding to IDs can be an embedding of P in G.*

Like SEED [13] and TwinTwig [12], we decompose the pattern graph into small decomposition units.

Definition 6. *A* decomposition *of query pattern P is a sequence of decomposition units $\mathcal{DE} = (dp_0, \ldots, dp_l)$ where every $dp_i \in \mathcal{DE}$ is a subgraph of P such that*

(1) The vertex set of dp_i consists of a pivot *vertex piv and a non-empty set LF of* leaf[1] vertices*, all of which are vertices in V_P; and for every $u' \in LF$, $(piv, u') \in E_P$.*

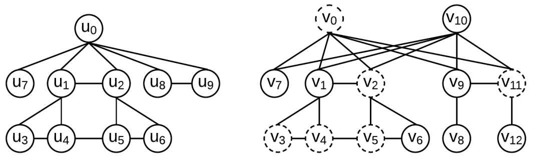
(2) The edge set of dp_i consists of two parts, $E^{star}_{dp_i}$ and $E^{sib}_{dp_i}$, where $E^{star}_{dp_i} = \bigcup_{u' \in LF} \{(dp_i.piv,\ u')\}$ is the set of edges between the pivot vertex and the leaf vertices, and $E^{sib}_{dp_i} = \bigcup_{u,u' \in dp_i.LF} \{(u, u') \in E_P\}$ is the set of edges between the leaf vertices.

(3) $\bigcup_{dp_i \in \mathcal{DE}}(V_{dp_i}) = V_P$, and for $i < j$, $V_{dp_i} \cap dp_j.LF = \emptyset$.

Note condition (3) in the above definition says the leaf vertices of each decomposition unit do not appear in the previous units. Unlike the decompositions in SEED [13] and TwinTwig [12], our decomposition unit is not restricted to stars and cliques, and $\bigcup_{dp_i \in \mathcal{DE}}(E_{dp_i})$ may be a *proper* subset of E_P.

Example 2. *Consider the query pattern in Figure 2 (a), we may have a decomposition $(dp_0,\ dp_1,\ dp_2,\ dp_3)$ where $dp_0.piv = u_0$, $dp_0.LF = \{u_1, u_2, u_7\}$, $dp_1.piv = u_1$, $dp_1.LF = \{u_3, u_4\}$, $dp_2.piv = u_2$, $dp_2.LF = \{u_5, u_6\}$, and $dp_3.piv = u_0$, $dp_3.LF = \{u_8, u_9\}$. Note that the edge $(u_4,\ u_5)$ is not in any decomposition unit.*

In the above example, the edge (u_4, u_5) is between vertices that belong to different units. We call such edges *cross-unit* edges. More formally, let $PL = (dp_0, \ldots, dp_l)$ be a decomposition. For each $i \in [0, l]$, we define $E^{cro}_{dp_i} = \{(u_1, u_2) \in E_P | u_1 \in \bigcup_{j<i} V_{dp_j} - \{dp_i.piv\}, u_2 \in dp_i.LF\}$, and call the edges in $E^{star}_{dp_i}$, $E^{sib}_{dp_i}$ and $E^{cro}_{dp_i}$ the *expansion* edges, *sibling* edges, and *cross-unit* edges respectively. The sibling edges and cross-unit edges are both called *verification* edges. Note that the expansion edges of all the units form a spanning tree of P, and the verification edges are the edges not in the spanning tree. Consider dp_2 in Example 2, we have $E^{star}_{dp_2}$=$\{(u_2, u_5), (u_2, u_6)\}$, $E^{sib}_{dp_2}$=$\{(u_5, u_6)\}$, $E^{cro}_{dp_2}$=$\{(u_4, u_5)\}$.



(a) Query Pattern *P*

(b) Data Graph *G*

Figure 2: Running Example

Given a decomposition $\mathcal{DE} = (dp_0, \ldots, dp_l)$ of pattern P, we define a sequence of *sub-query* patterns $P_0, \ldots, P_l$, where $P_0 = dp_0$, and for $i > 0$, P_i consists of the union of P_{i-1} and dp_i together with the edges across the vertices of P_{i-1} and dp_i, that is, $V_{P_i} = \bigcup_{j \le i} V_{dp_j}$, $E_{P_i} = \bigcup_{j \le i} (E_{dp_j} \cup E^{cro}_{dp_j})$. Note that (a) none of the leaf vertices of dp_i can be in P_{i-1}; and (b) P_i is the subgraph of P induced by the vertex set V_{P_i}, and $P_l = P$. We say $\mathcal{DE}$ forms an *execution plan* if for every $i \in [1, l]$, the pivot vertex of dp_i is in P_{i-1}.

Definition 7. *A decomposition $\mathcal{DE} = (dp_0, \ldots, dp_l)$ of P is an* execution plan *(PL) if $dp_i.piv \in V_{P_{i-1}}$ for all $i \in [1, l]$.*

For example, the decomposition in Example 2 is an execution plan.

Now we are ready to present the details of **R-Meef** as shown in Algorithm 1.

```
Algorithm 1: R-Meef FRAMEWORK
  Input: Query pattern $P$, partition $G_t$ on machine $M_t$,
         execution plan $PL$
  Output: $\mathbb{R}_{G_t}(P)$
1  $RG = \{rg_0 \ldots rg_k\} \leftarrow regionGroups\big(C(dp_0.piv, M_t)\big)$
2  for each region group $rg \in RG$ do
3      init embedding trie $\mathcal{ET}$ with size $|V_P|$
4      init edge verification index $I$
5      for each data vertex $v \in rg$ do
6          $f \leftarrow (dp_0.piv, v)$
7          updated $(\mathcal{ET},\ I) \leftarrow$
             $expandEmbedTrie(f, M_t, dp_0, \mathcal{ET})$
8      $\mathcal{R} \leftarrow verifyForeignE(I)$
9      $filterFailedEmbed(\mathcal{R}, I, \mathcal{ET})$
10     for Round $i = 1$ to $|PL|$ do
11         clear $I$
12         $fetchForeignV(i)$
13         for each $f \in I$ do
14             $I \leftarrow expandEmbedTrie(f, M_t, dp_i, \mathcal{ET})$
15         $\mathcal{R} \leftarrow verifyForeignE(I)$
16         $filterFailedEmbed(\mathcal{R}, I, \mathcal{ET})$
17     $\mathbb{R}_G(P) \leftarrow \mathbb{R}_G(P) \cup \mathcal{ET}$
18     clear $\mathcal{ET}$
```

Within each machine, we group the candidate data vertices of $dp_0.piv$ within M_t into region groups (Line 1). For each region group rg, a multi-round mapping process is conducted (Line 2 to 18). Within each round, we use a data

[1] In an abuse of the word "leaf".

structure $\mathcal{ET}$ (embedding trie) to save the generated intermediate results, i.e., embeddings and embedding candidates (Line 3). The EVI I is initialized in Line 4, which will be reset for each round of processing (line 11).

(1) *First Round (round 0).* Starting from each candidate v of rg, we match v to $dp_0.piv$ in the execution plan. After the pivot vertex is matched, we find all the ECs of dp_0 with respect to M_t and compress them into $\mathcal{ET}$. We use a function $expandEmbedTrie$ to represent this process (Line 7). For each EC compressed in $\mathcal{ET}$, its undetermined edges need to be verified in order to determine whether this EC is an embedding of dp_0. We record this information in the edge verification index I, which is constructed in the $expandEmbedTrie$ function. After we have the EVI I in M_t, we send a $verifyE$ request to verify those undetermined edges within I in the machine which has the ability to verify it (function $verifyForeignE$ in Line 8). After the edges in I are all verified, we remove the failed ECs from $\mathcal{ET}$ (Line 9).

(2) *Other Rounds.* For each of the remaining rounds of the execution plan, we first clear the EVI I from the previous round (Line 11). In the i^{th} round, we want to find all the ECs of P_i based on the embeddings in $\mathbb{R}_{G_t}(P_{i-1})$ (where $dp_i.piv$ has been matched). The process is to expand every embedding f of $\mathbb{R}_{G_t}(P_{i-1})$ with each embedding candidate of dp_i within the neighbourhood of $f(dp_i.piv)$. If not all the data vertices matched to $dp_i.piv$ by the ECs in $\mathbb{R}_{G_t}(P_{i-1})$ reside in M_t, we will have to fetch the adjacency-lists of those foreign vertices from other machines in order to expand from them. A sub-procedure $fetchForeignV$ is used to represent this process (Line 12). After fetching, for each embedding f of $\mathbb{R}_{G_t}(P_{i-1})$, we find all the ECs of P_i by expanding from $f(dp_i.piv)$ (Line 14). The found ECs are compressed into $\mathcal{ET}$. Then $verifyForeignE$ and $filterFailedEmbed$ are called to make sure that the failed ECs are filtered out from the embedding trie, which will only contain the actual embeddings of P_i, i.e., $\mathbb{R}_{G_t}(P_i)$ (Line 15, 16).

After all the rounds of this region group have finished, we have a set of embeddings of P compressed into $\mathcal{ET}$. The results obtained from all the region groups are put together to obtain the embeddings found by M_t.

One important thing to note is that if a foreign vertex is already cached in the local machine, for the undetermined edges attached to this vertex, we can verify them locally without sending requests to other machines. Also we do not re-fetch any foreign vertex if it is already cached previously.

Example 3. *Consider the data graph G in Figure 2, where the vertices marked with dashed border lines reside in M_1 and the other vertices reside in M_2. Consider the pattern P and execution plan PL given in Example 2. We assume the preserved orders due to symmetry breaking are: $u_1 < u_2$, $u_3 < u_6$, $u_4 < u_5$ and $u_8 < u_9$.*

There are two vertices $\{v_0,\ v_2\}$ in M_1 and two vertices $\{v_1,\ v_{10}\}$ in M_2 with a degree not smaller than that of $dp_0.piv$. Therefore in M_1, we have $C(dp_0.piv) = \{v_0,\ v_2\}$ and in M_2 we have $C(dp_0.piv) = \{v_1,\ v_{10}\}$. After grouping, assume we have $RG = \{rg_0,\ rg_1\}$ where $rg_0 = \{v_0\}$ and $rg_1 = \{v_2\}$ in M_1, and $RG = \{rg_0\}$ where $rg_0 = \{v_1,\ v_{10}\}$ in M_2.

Consider the region group rg_0 in M_1. In round 0, we first match v_0 to $dp_0.piv$. Expanding from v_0, we may have ECs including by not limit to (we lock u_7 to v_7 for easy demonstration):

$f_{G_1} = \{(u_0,\ v_0),\ (u_1,\ v_1),\ (u_2,\ v_2),\ (u_7,\ v_7)\}$
$f'_{G_1} = \{(u_0, v_0),\ (u_1,\ v_1),\ (u_2,\ v_9),\ (u_7,\ v_7)\}$
$f''_{G_1} = \{(u_0, v_0),\ (u_1,\ v_9),\ (u_2,\ v_{11}),\ (u_7,\ v_7)\}$

These ECs are compressed into $\mathcal{ET}$. Note that a mapping such as $\{(u_0, v_0),\ (u_1,\ v_1),\ (u_2,\ v_{11}),\ (u_7,\ v_7)\}$ is not an EC of dp_0 w.r.t M_1 since $(v_1,\ v_{11})$ can be locally verified to be non-existent. Since the undetermined edge $(v_1,\ v_9)$ of f'_{G_1} cannot be determined in M_1, we put $\{(v_1, v_9), < f'_{G_1} >\}$ into the EVI I. We then ask M_2 to verify the existence of the edge. M_2 returns false, therefore f'_{G_1} will be removed from $\mathcal{ET}$.

In round 1, we have two embeddings $\mathbb{R}_{G_1}(P_0) = \{f_{G_1},\ f''_{G_1}\}$ to start with. To expand f_{G_1} and f''_{G_1}, we need to fetch the adjacency-lists of v_1 and v_9 respectively. We send a single $fetchV$ request to fetch the adjacency-lists of v_1 and v_9 from M_2. After expansion from v_1, we get a single embedding $\{(u_0,\ v_0),\ (u_1,\ v_1),\ (u_2,\ v_2),\ (u_3,\ v_3),\ (u_4,\ v_4),\ (u_7,\ v_7)\ \}$ in $R_{G_t}(P_1)$. There is no embedding of P_1 expanded from v_9. Hence f''_{G_1} will be removed from the embedding trie.

In round 2, we expand from v_2 to get the ECs of P_2. $dp_2.piv$ was already mapped to v_2 as seen above, and v_2 has neighbors v_5, v_6 and v_{10} that are not matched to any query vertices. Since there are sibling edge (u_5, u_6) and cross-unit edge (u_4, u_5) in P_2, we need to verify the existence of (v_4, v_5) and (v_5, v_6) if we want to map u_5 to v_5 and map u_6 to v_6. The existence of both $(v_4,\ v_5)$ and (v_5, v_6) can be verified locally. Similarly if we want to map u_5 to v_5, u_6 to v_{10}, we will have to verify the existence of (v_5, v_{10}), and so on. It can be locally verified that (v_5, v_{10}) does not exist, and remotely verified that (v_6, v_{10}) does not exist. Therefore, at the end of this round, we will get a single embedding for P_2 which extends the embedding for P_1 by mapping u_5, u_6 to v_5, v_6 respectively. We expand the embedding trie accordingly.

Following the above process, after we process the last round, we have an embedding of P starting from region group rg_0 in machine M_1, which will be saved in $\mathcal{ET}$:

$f_{G_1} = \{(u_0, v_0),\ (u_1,\ v_1),\ (u_2,\ v_2),\ (u_3,\ v_3),\ (u_4,\ v_4),\ (u_5,\ v_5),\ (u_6,\ v_6),\ (u_7,\ v_7),\ (u_8,\ v_9),\ (u_9,\ v_{11})\}$

In order to achieve the best performance, each component of Algorithm 1 should be carefully designed. In the following sections, we address the issues one by one.

5. COMPUTING EXECUTION PLAN

It is obvious that we may have multiple valid execution plans for a query pattern and different execution plans may have different performance. The challenge is how to find the most efficient one among them? In this section, we present some heuristics to find a good execution plan.

5.1 Minimizing Number of Rounds

Given query pattern P and an execution plan PL, we have $|PL|+1$ rounds for each region group, and once all the rounds are processed we will get the set of final embeddings. Also, within each round, the workload can be shared. To be specific, a single undetermined edge e may be shared by multiple ECs. If these embedding candidates are generated in the same round, the verification of e can be shared by all of them. The same applies to the foreign vertices where the cost of fetching and memory space can be shared among multiple embedding candidates if they happen to be in the same round. Therefore, our first heuristic is to minimize the number of rounds (namely, the number of decomposition units) so as to maximize the workload sharing.

Here we present a technique to compute a query execution plan, which guarantees a minimum number of rounds. Our technique is based on the concepts of *maximum leaf spanning tree* [6] and *minimum connected dominating set*.

Definition 8. *A* maximum leaf spanning tree (MLST) *of pattern P is a spanning tree of P with the maximum number of leafs (a leaf is a vertex with degree 1). The number of leafs in a MLST of P is called the* maximum leaf number *of P, denoted l_P.*

Definition 9. *A* connected dominating set (CDS) *of P is a subset D of V_P such that (1) D is a dominating set of P, that is, any vertex of P is either in D or adjacent to a vertex in D, and (2) the subgraph of P induced by D is connected. A* minimum connected dominating set (MCDS) *is a CDS with the smallest cardinality among all CDSs. The number of vertices in a MCDS is called the* connected domination number*, denoted c_P.*

It is shown in [3] that $|V_P| = c_P + l_P$.

Theorem 1. *Given a pattern P, any execution plan of P has at least c_P decomposition units, and there exists an execution plan with exactly c_P decomposition units.*

Proof. Suppose $\{dp_0, \ldots, dp_k\}$ is an execution plan with $k+1$ decomposition units. The pivot vertices of the decomposition units form a connected dominating set of P. Therefore, $k+1 \geq c_P$. This proves any execution plan has at least c_P decomposition units.

Suppose T is a MLST of P. From $|V_P| = c_P + l_P$ we know the number of non-leaf vertices in T is c_P. We can construct an execution plan by choosing one of the non-leaf vertices v_0 as $dp_0.piv$, and all neighbors of v_0 in T as the vertices in $dp_0.LF$. Regarding v_0 as the root of the spanning tree T, we then choose each of the non-leaf children v_i of v_0 in T as the pivot vertex of the next decomposition unit $dp_i.piv$, and all children of v_i as the vertices in $dp_i.LF$. Repeat this process until every non-leaf vertex of T becomes the pivot vertex of a decomposition unit. This decomposition has exactly c_P units, and it forms an execution plan. This shows that there exists an execution plan with c_P decomposition units. □

Theorem 1 indicates that c_P is the minimum number of rounds of any execution plan. The above proof provides a method to construct an execution plan with c_P rounds from a MLST.

Example 4. *Consider the pattern P in Figure 2, it can be easily verified that the tree obtained by erasing the edges (u_1, u_2), (u_3, u_4), (u_4, u_5), (u_5, u_6) and (u_8, u_9) is a MLST of P. Choosing u_0 as the root, we will get a minimum round execution plan PL_1=$\{dp_0,\ dp_1,\ dp_2\}$ where $dp_0.piv = u_0$, $dp_0.LF = \{u_1,\ u_2,\ u_7,\ u_8,\ u_9\}$, $dp_1.piv = u_1$, $dp_1.LF = \{u_3, u_4\}$ and $dp_2.piv = u_2$, $dp_2.LF = \{u_5, u_6\}$. If we choose u_1 as the root, we will get a different minimum-round execution plan PL_2=$\{dp_0, dp_1, dp_2\}$, where $dp_0.piv = u_1$, $dp_0.LF = \{u_0, u_3, u_4\}$, $dp_1.piv = u_0$, $dp_1.LF = \{u_2,\ u_7\ u_8, u_9\}$, $dp_2.piv = u_2$, $dp_2.LF = \{u_5, u_6\}$.*

5.2 Minimizing the span of $dp_0.piv$

Given a pattern P, multiple execution plans may exist with the minimum number of rounds, while their $dp_0.piv$ can be different. Our second heuristic is to choose the plan(s) whose $dp_0.piv$ have the smallest span. This strategy will maximize the number of embeddings that can be found using SM-E. Recall the RADS architecture where $dp_0.piv$ is the starting query vertex u_{start}, based on Proposition 1, we know that the more candidate vertices of $dp_0.piv$ can be processed in SM-E, the more workload can be separated from the distributed processing, and therefore the more communication cost and memory usage can be reduced.

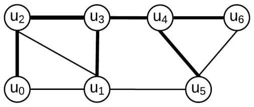


Figure 3: A Query Pattern

Consider the pattern in Figure 3, the bold edges demonstrate a MLST based on which both u_3 and u_4 can be chosen as $dp_0.piv$. The execution plans from them have the same number of rounds. However, $Span_P(u_3) = 2$ while $Span_P(u_4) = 3$. Therefore we choose the plan with u_3 as the $dp_0.piv$.

5.3 Maximizing Filtering Power

Given a pattern P, multiple execution plans may exist with the minimum number of rounds and their $dp_0.piv$ have the same smallest span. Here we use the third heuristic which is to choose plans with more verification edges in the earlier rounds. The intuition is to maximize the filtering power of the verification edges as early as possible. To this end, we propose the following score function $\mathcal{SC}(PL)$ for an execution plan $PL = \{dp_0,\ \ldots,\ dp_l\}$:

$$\mathcal{SC}(PL) = \sum_{dp_i \in PL} \frac{1}{(i+1)^{\rho}} \times (|E^{sib}_{dp_i}| + |E^{cro}_{dp_i}|) \qquad (3)$$

$|E^{sib}_{dp_i}| + |E^{cro}_{dp_i}|$ is the number of verification edges in round i, and ρ is a positive parameter used to tune the score function. In our experiments we use $\rho = 1$. The function $\mathcal{SC}(PL)$ calculates a score by assigning larger weights to the verification edges in earlier rounds (since $\frac{1}{(i+1)^{\rho}} > \frac{1}{(j+1)^{\rho}}$ if $i < j$).

Example 5. *Consider the query plans PL_1 and PL_2 in Example 4. The total number of verification edges in these plans are the same. In PL_1, the number of verification edges for the first, second and third round is 2, 1, 2 respectively. In PL_2, the number of verification edges for the three rounds is 1, 2, and 2 respectively. Therefore, we prefer PL_1. Using $\rho = 1$, we can calculate the scores of the two plans as follows:*
$\mathcal{SC}(PL_1) = 2/1 + 1/2 + 2/3 \approx 3.2$
$\mathcal{SC}(PL_2) = 1/1 + 2/2 + 2/3 \approx 2.7$.

When several minimum-round execution plans have the same score, we use another heuristic rule to choose the best one from them: the larger the degree of the pivot vertex, the earlier we process the unit. The pivot vertex with a larger degree has a stronger power to filter unpromising candidates.

6. EMBEDDING TRIE

In this section we present the data structure used for compressing the intermediate results as well the algorithms to maintain it. We first define a matching order, following which the query vertices are matched. It is also the order the nodes in the embedding trie are organized.

Definition 10. *Given a query execution plan $PL = \{dp_0, \ldots, dp_l\}$ of pattern P, the matching order w. r. t PL is a relation $\prec$ defined over the vertices of P that satisfies the following conditions:*

(1) $dp_i.piv \prec dp_j.piv$ *if* $i < j$*;*

(2) For any two vertices $u_1 \in dp_i.LF$ *and* $u_2 \in dp_j.LF$, $u_1 \prec u_2$ *if* $i < j$*.*

(3) For $i \in [0, l]$*:*

- *(i)* $dp_i.piv \prec u$ *for all* $u \in dp_i.LF$*;*
- *(ii) for any vertices* u_1, $u_2 \in dp_i.LF$ *that are not the pivot vertices of other units,* $u_1 \prec u_2$ *if* $deg(u_1) > deg(u_2)$*, or* $deg(u_1) = deg(u_2)$ *and the vertex ID of* u_1 *is less than that of* u_2*;*
- *(iii) if* $u_1 \in dp_i.LF$ *is a pivot vertex of another unit, and* $u_2 \in dp_i.LF$ *is not a pivot vertex of another unit, then* $u_1 \prec u_2$*.*

It is easy to verify $\prec$ is a strict total order over V_P. Following the matching order, the vertices of P can be arranged into an ordered list. Consider the execution plan PL_1 in Example 4. The vertices in the query can be arranged as $(u_0, u_1, u_2, u_7, u_8, u_9, u_3, u_4, u_5, u_6)$ according to the matching order.

Let $PL = \{dp_0, \ldots, dp_l\}$ be an execution plan, P_i be the subgraph of P induced from the vertices in $dp_0 \cup \cdots \cup dp_i$ (as defined in Section 4), and $\widetilde{\mathbb{R}}$ be a set of *results* (i.e., embeddings or embedding candidates) of P_i. For easy presentation, we assume the vertices in P_i have been arranged into the list $u^0, u^1, \ldots, u^n$ by the matching order, that is, the query vertex at position j is u^j. Then each result of P_i can be represented as a list of corresponding data vertices.

Next we formally define embedding trie and present the algorithms for the maintenance of the embedding trie.

6.1 Structure of the Embedding Trie

Definition 11. *Given a set* $\widetilde{\mathbb{R}}$ *of results of* P_i*, the* embedding trie *of* $\widetilde{\mathbb{R}}$ *is a collection of trees used to store the results in* $\widetilde{\mathbb{R}}$ *such that:*

(1) Each tree represents a set of results that map u^0 *to the same data vertex.*

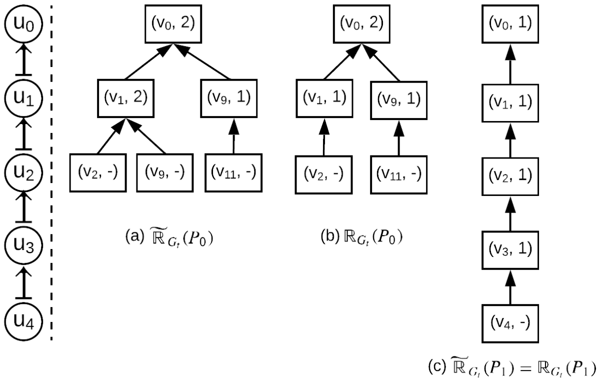
(2) Each tree node $\mathcal{N}$ *has*

- ***v****: a data vertex*
- ***parentN****: a pointer pointing to its parent node (the pointer of the root node is null).*
- ***childCount****: the number of child nodes of* $\mathcal{N}$*.*

(3) If two nodes have the same parent, then they store different data vertices.

(4) Every leaf-to-root path represents a result in $\widetilde{\mathbb{R}}$*, and every result in* $\widetilde{\mathbb{R}}$ *is represented as a unique leaf-to-root path.*

(5) If we divide the tree nodes into different levels such that the root nodes are at level 0, the children of the root nodes are at level 1 and so on, then the tree nodes at level j *(*$j \in [0, 1]$*) store the set of values* $\{f(u^j) | f \in \widetilde{\mathbb{R}}\}$*.*



(a) $\widetilde{\mathbb{R}}_{G_t}(P_0)$

(b) $\mathbb{R}_{G_t}(P_0)$

(c) $\widetilde{\mathbb{R}}_{G_t}(P_1) = \mathbb{R}_{G_t}(P_1)$

Figure 4: Example of Embedding Trie

Example 6. *Consider* P_0 *in Example 3, where the vertices are ordered as* u_0, u_1, u_2 *according to the matching order. There are three ECs of* P_0*:* (v_0, v_1, v_2)*,* (v_0, v_1, v_9) *and* (v_0, v_9, v_{11})*. These results can be stored in a tree shown in Figure 4(a). When the second EC is filtered out, we have* $\mathbb{R}_{G_t}(P_0)$ *compressed in a tree as shown in Figure 4(b). The first EC can be expanded to an EC of* P_1 *(where the list of vertices of* P_1 *are* $(u_0, u_1, u_2, u_3, u_4)$*, which is as shown in Figure 4(c).*

Although the structure of embedding trie is simple, it has some nice properties: (1) *Compression* Storing the results in the embedding trie saves space than storing them as a collection of lists. (2) *Unique ID* For each result in the embedding trie, the address of its leaf node in memory can be used as the unique ID. (3) *Retrieval* Given a particular ID represented by a leaf node, we can easily follow its pointer *parentN* step-by-step to retrieve the corresponding result. (4) *Removal* To remove a result with a particular ID, we can remove its corresponding leaf node and decrease the *childCount* of its parent node by 1. If *ChildCount* of this parent node reaches 0, we remove this parent node. This process recursively affects the ancestors of the leaf node.

6.2 Maintaining the embedding trie

Recall that in Algorithm 1, given an embedding f of P_{i-1}, the function *expandEmbedTrie* is used to search for the ECs of dp_i within the neighbourhood of the mapped data vertex v_{piv}, where $v_{piv} = f(dp_i.piv)$. Moreover, the *expandEmbedTrie* function handles the task of expanding the embedding trie $\mathcal{ET}$ by concatenating f with each newly found EC of dp_i. If an EC is filtered out or if an embedding cannot be expanded to a final result, the function must remove it from $\mathcal{ET}$.

The details of the *expandEmbedTrie* function are given in Algorithm 2. Lines 3 to 7 set the candidate set for each $u \in dp_i.LF$ as the intersection of the neighbour set of $v_{piv} = f(dp_i.piv)$ and the neighbour set of each $f(u')$, where (u', u) is a cross-unit edge and $f(u')$ is in M_t. If any of the candidate sets is empty, it removes the corresponding trie node $\mathcal{N}$ from $\mathcal{ET}$. Otherwise it picks the first leaf vertex u and calls a recursive subroutine *adjEnum* (Line 11, 12). By expanding any f of P_{i-1}, *adjEnum* finds the ECs of P_i within the neighbourhood of v_{piv}. If f cannot be expanded into an EC of P_i, we will remove $\mathcal{N}$ from $\mathcal{ET}$ (Line 13).

The details of the *adjEnum* function are presented in Algorithm 3. In each round of *adjEnum*, we try to match

```
Algorithm 2: expandEmbedTrie
  Input: an embedding $f$ of $P_{i-1}$, local machine $M_t$, unit
         $dp_i$, embedding trie $\mathcal{ET}$
  Output: expanded $\mathcal{ET}$ and an edge verification index I
1  $v_{piv} \leftarrow f(dp_i.piv)$
2  get $\mathcal{N}$ corresponding to $f$
3  for each $u \in dp_i.LF$ do
4      $C(u) \leftarrow adj(v_{piv})$
5      for each $(u, u') \in E^{cro}_{dp_i}$ do
6          if $f(u')$ resides in $M_t$ then
7              $C(u) \leftarrow adj(f(u')) \cap C(u)$
8      if $C(u) = \emptyset$ then
9          remove $\mathcal{N}$ from $\mathcal{ET}$
10         return
11 $u \leftarrow$ first leaf vertex in $dp_i.LF$
12 if adjEnum($\mathcal{N}$, $u$) is false then
13     remove $\mathcal{N}$ from $\mathcal{ET}$
```

u' to a candidate vertex v, where u' is a query vertex in $dp_i.LF$. If v can pass the local verification (Line 4,5), we add (u, v) to f and create a trie node $\mathcal{N}'$ for v (Line 6 to 9). If all leaf vertices have been verified, we add all undetermined edges of f to $I[e]$ (Line 11, 12). Otherwise, we call a deeper *adjEnum*. If the current f can be expanded to an EC, we chain $\mathcal{N}'$ to the corresponding node $\mathcal{N}$ of $\mathcal{ET}$ (Line 17, 19).

```
Algorithm 3: adjEnum
  Input: Trie node $\mathcal{N}$ representing embedding $f$ of $P_{i-1}$ ,
         leaf vertex $u$ of $dp_i$
  Output: $true$ or $false$
1  $\mathcal{F}_{current} \leftarrow false$
2  for each $v \in C(u)$ do
3      $\mathcal{E} \leftarrow true$
4      if $v$ resides in $M_t$ then
5          $\mathcal{E} \leftarrow isJoinable(u, v)$
6      if $\mathcal{E}$ is true then
7          add $(u, v)$ to $f$
8          create a trie node $\mathcal{N}'$
9          $\mathcal{N}'.v \leftarrow v$, $\mathcal{N}'.parentN \leftarrow \mathcal{N}$
10         if $|f| = |V_{P_i}|$ then
11             for each undetermined edge $e$ of $f$ do
12                 add $\mathcal{N}'$ to $I[e]$
13             $\mathcal{F}_{current} \leftarrow true$
14         else
15             $u' \leftarrow$ next vertex in $dp_i.LF$
16             $\mathcal{F}_{deeper} \leftarrow adjEnum(\mathcal{N}', u')$
17         if $F_{deeper}$ is true then
18             $\mathcal{N}.childCount++$
19             add $\mathcal{N}'$ as a child node of $\mathcal{N}$ in $\mathcal{ET}$
20         remove $(u, v)$ from $f$
21 return $\mathcal{F}_{current}$
```

7. MEMORY CONTROL STRATEGIES

This section focuses on the robustness of **R-Meef**. Since **R-Meef** still caches fetched foreign vertices and intermediate results in memory, memory consumption is still a critical issue when the data graph is large. We propose a grouping strategy to keep the peak memory usage under the memory capacity of the local machine.

Our idea is to divide the candidate vertices of the first query vertex $dp_0.piv$ into disjoint groups and process each group independently. In this way, the overall cached data on each machine will be divided into several parts, where each part is no larger than the available memory Φ.

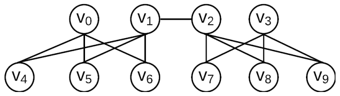


Figure 5: Grouping Example

A naive way of grouping the candidate vertices is to divide them randomly. However, random grouping of the vertices may put vertices that are "dissimilar" to each other into the same group, potentially resulting in more network communication cost. Consider the data graph in Figure 5. Suppose the candidate vertex set is $\{v_0,v_1,v_2,v_3\}$. If we divide it into two groups $\{v_0, v_1\}$ and $\{v_2, v_3\}$, then because v_0 and v_1 share most neighbours, there is a good chance for the ECs of dp_0 generated from v_0 and v_1 to share common verification edges, and share common foreign vertices that need to be fetched (e.g., if $dp_1.piv$ is mapped to v_5 by ECs originated from v_0 and v_1, and v_5 is not on the local machine). However, if we partition the candidate set into $\{v_0, v_2\}$ and $\{v_1, v_3\}$, then there is little chance for such sharing.

Our goal is to find a way to partition the candidate vertices into groups so that the chance of edge verification sharing and foreign vertices sharing by the results in each group is maximized.

Let $C \equiv C(dp_0.piv)$ be the candidate set of $dp_0.piv$, and Φ be the available memory. Our method is to generate the groups one by one as follows. First we pick a random vertex $v \in C$ and let $rg = \{v\}$ be the initial group. If the estimated memory requirement of the results originated from rg, denoted $\phi(rg)$, is less than Φ, we choose another candidate vertex in $C - rg$ that has the greatest proximity to rg and add it to rg; if $\phi(rg) > \Phi$ we remove the last added vertex from rg. This generates the first group. For the remaining candidate vertices we repeat the process, until all candidate vertices are divided into groups. Here an important concept is the proximity of a vertex v to a group of vertices, and we define it as the percentage of v's neighbors that are also neighbors of some vertex in rg, that is,

$$proximity(v, rg) = \frac{|adj(v) \cap \bigcup_{v' \in rg} adj(v')|}{|adj(v)|} \tag{4}$$

Intuitively the vertices put into the same group are within a region - each time we will choose a new vertex that has a distance of at most 2 from one of the vertices already in the group (unless there are no such vertices). Therefore we call the group a *region* group.

Estimating memory usage In our system, the main memory consumption comes from the intermediate results and the fetched foreign vertices. The space cost of other data structures is trivial.

Consider the set of intermediate results originated from the group $rg \subseteq C$. Recall that all results originated from the same candidate vertex of $dp_0.piv$ are stored in the same tree, while any results originated from different candidate vertices are stored in different trees. Therefore, if we know the space cost of the results originated from every candidate vertex, we can add them together to obtain the space cost of all results originated from rg.

To estimate the space cost of the results originated from a single vertex, we use the average space cost of local embeddings of a candidate vertex $v \in C_1(u_{start})$ in embedding trie format, which can be obtained when we conduct SM-E.

Recall that for each v of $C_1(u_{start})$ in SM-E, we find the local embeddings originated from v following a backtracking approach. In each recursive step of the backtracking approach, we may record the number of candidate vertices that are matched to the corresponding query vertex. The sum of all steps will be the number of trie nodes if we group the those local embeddings into embedding trie. Based on the sum, we know the space cost of local embeddings originating from v in the format of embedding trie.

Next, we consider the space cost of the fetched foreign vertices in each round. Recall that when expanding the embeddings of P_{i-1} to ECs of P_i, we only need to fetch vertex v if there exists $f \in \mathbb{R}_{G_t}(P_{i-1})$ such that $f(dp_i.piv) = v$. In the worst case, for every candidate vertex v of $dp_i.piv$, there exists some $f \in \mathbb{R}_{G_t}(P_{i-1})$ which maps $dp_i.piv$ to v, and none of these candidate vertices of $dp_i.piv$ resides locally. Therefore the number of data vertices that need to be fetched equals to $|C(dp_i.piv)|$ in the worst case.

8. EXPERIMENTS

In this section, we present our experimental results. We focus on performance comparsion with four state-of-the-art distributed subgraph enumeration approaches:

- PSgL [20], the algorithm using graph exploration originally based on Pregel.
- TwinTwig [12], the algorithm using joining approach originally based on MapReduce.
- SEED [13], an upgraded version of TwinTwig while supporting clique decomposition unit.
- Crystal [16], the algorithm relying on clique-index and compression and originally using MapReduce.

We will also compare the communication cost. In addition, we will test the impact of our execution plan and our compression strategy. For scalability, we will follow the approach of [13] and test the effect on query time by varying the number of machines in the distributed cluster, the number of data vertices, and the average degree of the data vertices.

Environment We conducted our experiments in a cluster platform where each machine is equipped with Intel CPU with 16 Cores and 16G memory. In our performance comparison we used 10 machines in the cluster. For scalability test on varying data graph size and vertex degree we used a larger cluster of 20 machines since the time taken on larger and denser graphs is much longer. The operating system of the clusters is Red Hat Enterprise Linux 6.5.

We implemented our approach in C++ with the help of Mpich2 [8] and Boost library [19]. We used Boost.Asio to achieve the asynchronous message listening and passing. We used TurboIso [9] as our SM-E processing algorithm. We implemented PSgL, TwinTwig and SEED using C++ with MPI library as well. For Crystal, we used the code provided by the authors.

Dataset & Queries We used five real datasets: DBLP, RoadNet, LiveJournal, UK2002 and Friendster, whose profiles are given in Table 1. DBLP is a small data graph which is to test whether the approaches can fully utilize the memory when there is enough space available. RoadNet is a sparse data graph, which is used to illustrate whether a subgraph enumeration solution has good filtering power to filter out false embeddings early. Two denser data graphs, LiveJournal and UK2002, are used to test the algorithms' ability to handle denser graphs with larger numbers of embeddings. Friendster is a super large data graph (32.4GB), where many queries will fail to complete in a reasonable time for all algorithms. We only used it in the scalability test against the graph size and average degree.

Table 1: Profiles of datasets, M– milion, B– billion

Dataset(G)	$\|V\|$	$\|E\|$	Avg. degree	Diameter
RoadNet	1.9M	2.7M	2.81	849
DBLP	0.3M	1.0M	6.62	21
LiveJournal	4.8M	42.9M	18	17
UK2002	18.5M	298.1M	32	22
Friendster	65M	1.8B	55	32

On disk, our data graphs are stored in plain text format where each line represents an adjacency-list of a vertex. We used Metis [10] to partition the data graphs and each machine randomly picks up one partition and no duplication is allowed. The approach of Crystal relies on the clique-index of the data graph which should be pre-constructed and stored on disk. In Table 2, we present the disk space cost of the index files generated by the program of Crystal. The index file for Frienster is not included since it takes too long to compute. As can be seen, the index files are more than 10 times larger than the original data graph. The queries we used are shown in Figure 6. Experiments using additional queries can be found in the long version of this paper [18].

Table 2: Illustration of the Size of Index Files of Crystal

Dataset(G)	Data Graph File Size	Index File Size
RoadNet	43.1MB	569MB
DBLP	13MB	210MB
LiveJournal	501MB	6.5GB
UK2002	4.1GB	60GB

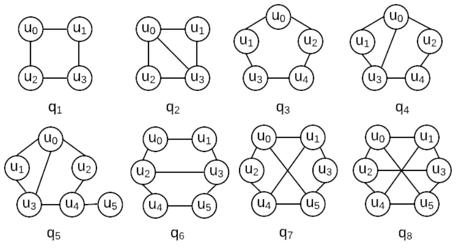
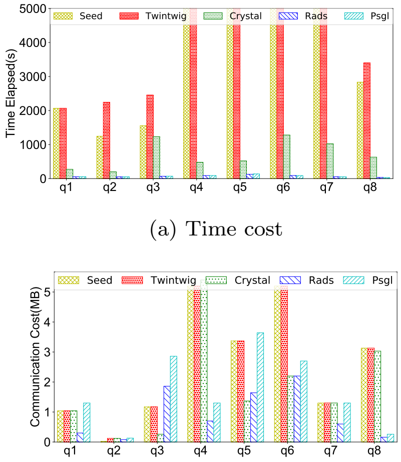


Figure 6: Query Set

8.1 Performance Comparison

We compared the performance of the five subgraph enumeration approaches by measuring the time elapsed (in seconds) and the volume of exchanged data during the processing of each query. The results of RoadNet, DBLP, LiveJournal and UK2002 are given in Figures 7, 8, 9 and 10, respectively. We mark the result as empty when the test fails due to memory crash. When any bar reaches the upper bound, it means the corresponding values is beyond the upper bound value shown in the chart.

Exp-1:RoadNet The results over the RoadNet dataset are given in Figure 7. As can be seen from the figure, RADS and PSgL are significantly faster than the other three methods (by more than 1 order of magnitude). RADS and PSgL are using graph exploration while the others are using join-based



(a) Time cost

(b) Communication Cost

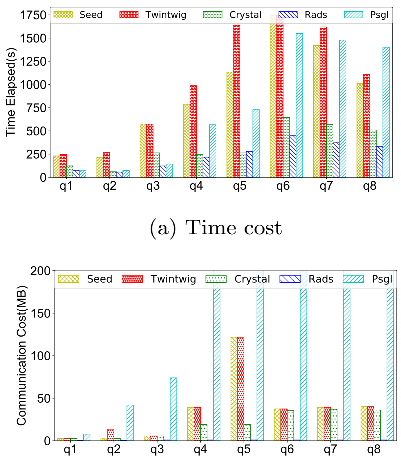
Figure 7: Performance over RoadNet

methods. Therefore, both RADS and PSgL demonstrated efficient filtering power for this sparse graph. The join-based methods need to group the intermediate results based on keys so as to join them together, their performance was significantly dragged down when dealing with sparse graphs compared with RADS and PSgL.

It is worth noting that PSgL was shown to be slower than SEED and TwinTwig in [12][13]. This may be because the datasets used in TwinTwig and SEED are much denser than RoadNet, hence a huge number of embeddings will be generated. The grouped intermediate results of TwinTwig and SEED significantly reduced the cost of network traffic. Another interesting observation is that although Crystal has heavy indexes, its performance is much worse than PSgL and RADS. The reason is that the number of cliques in RoadNet is relatively small considering the graph size. Moreover, there are no cliques with more than two vertices in queries q_1, q_3, q_6, q_7 and q_8. In such cases, the clique index cannot help to improve the performance.

As shown in Figure 7(b), the communication cost is not large for any of the approaches (less than 5M for most queries). In particular, for RADS, the communication cost is almost 0, which is mainly because most data vertices can be processed by SM-E, as such little network communication is required.

Exp-2:DBLP The result over DBLP is shown in Figure 8. As aforementioned, DBLP is smaller but much denser than RoadNet. The number of intermediate results generated in DBLP are much larger than that in RoadNet, as implied by the data communication cost shown in Figure 8 (b). Since PSgL does not consider any compression or grouping of intermediate results, the communication cost of PSgL is much higher than that of the other approaches (more than 200M for queries after q_4). Consequently, the time delay due to shuffling the intermediate results caused bad performance for PSgL. However, PSgL is still faster than SEED and TwinTwig. This may be because the time cost of grouping intermediate results of TwinTwig and SEED is high as well. It is worth noting that the communication cost of our RADS is quite small (less than 5M). The time efficiency of RADS is better than Crystal even for queries q_2,q_4 and q_5 where the triangle crystal can be directly loaded from index without any computation.

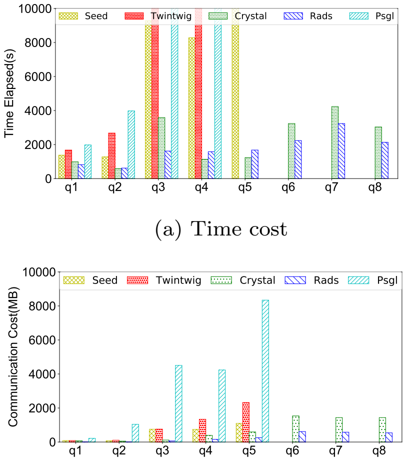


(a) Time cost

(b) Communication Cost

Figure 8: Performance over DBLP

Exp-3:LiveJournal As shown in Figure 9, for LiveJournal, SEED, TwinTwig and PSgL start becoming impractical for queries from q_3 to q_8. Due to the huge number of intermediate results generated, the communication cost increased significantly as well, especially for PSgL whose communication cost was beyond control when the query vertices reach 6. The method of Crystal achieved good performance for queries q_2, q_4 and q_5. This is mainly because Crystal simply retrieved the cached embeddings of the triangle to match the vertices (u_0, u_1, u_2) of those 3 queries. However, when dealing with the queries with no good crystals (q_6, q_7 and q_8), our method significantly outperformed Crystal. One important thing to note is that the other three methods (SEED, TwinTwig and PSgL) are sensitive to the end vertices, such as u_5 in q_5. Both time cost and communication cost increased significantly from q_4 to q_5. RADS processes those end vertices last by simply enumerating the combinations without caching any results related to them. The end vertices within Crystal will be bud vertices which only requires simple combinations. As indicated by query q_5 where their processing time increased slightly from that of q_4, RADS and Crystal are nicely tuned to handle end vertices.



(a) Time cost

(b) Communication Cost

Figure 9: Performance over LiveJournal

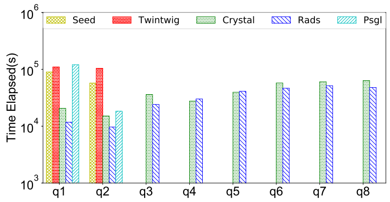


Figure 10: Time cost over UK2002

Exp-4:UK2002 As shown in Figure 10, `TwinTwig`, `SEED` and `PSgL` failed the tests of queries after q_2 due to memory failure caused by huge number of intermediate results. The communication cost of all other methods are significantly larger than `RADS` (more than 2 orders of magnitude), we omit the chart for communication cost here. Similar to that of LiveJournal, the processing time of `Crystal` is slightly better than that of `RADS` for q_4 and q_5 (which have triangles). This is because `Crystal` directly retrieves the embeddings of the triangles from the index. However, for q_2 and queries without good crystals, our approach demonstrates better performance. Note that the index file of `Crystal` is more than 10 times larger than the original data graph, as shown in Table 2.

Another advantage of `RADS` over `Crystal` is our memory control strategies ensures it is more robust: we tried to set a memory upper bound of 8G and test query q_6, `Crystal` starts crashing due to memory leaks, while `RADS` successfully finished the query.

8.2 Effectiveness of Compression

To show the effectiveness of our compression strategy, we conducted an experiment to compare the space cost of the simple embedding-list (EL) with that of our embedding trie (ET). We use the RoadNet and DBLP data sets for this test. The queries are as shown in Figure 6. We omit the test over the other two data sets because the uncompressed volume of the results are too large.

Table 3: Compression on RoadNet(Mb)

Query	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8
EL	264	13	65	81	136	183	-	-
ET	163	5	33	40	63	73	-	-

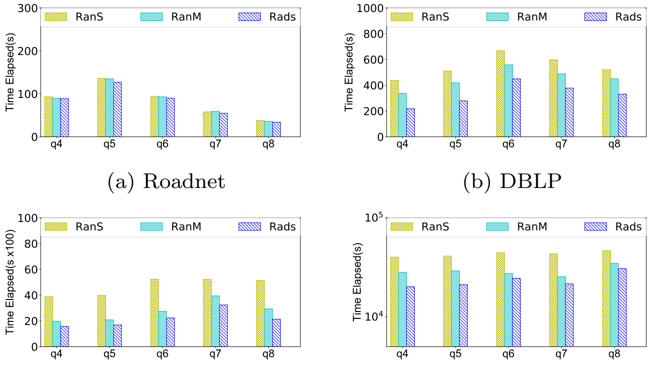
Table 4: Compression on DBLP (Gb)

Query	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8
EL	0.3	0.2	4.5	3.2	17.6	7.6	5.3	4
ET	0.08M	0.06	1.1	0.7	3.8	1.3	0.9	0.8

The results are as shown in Table 3 and Table 4, respectively. For RoadNet the intermediate results generated by Queries 7 and 8 are negligible, therefore they are not listed. The results for both datasets demonstrate a good compression ratio. It is worth noting that the compression ratios of all queries over RoadNet are smaller than that over DBLP. This is because the embeddings of RoadNet are very diverse and they do not share a lot of common vertices.

8.3 Effectiveness of Query Execution Plan

To validate the effectiveness of our strategy for choosing query execution plan, we compared the processing time of `RADS` with two other baseline plans which are generated by replacing the execution plan of `RADS` with the execution plans $RanS$ and $RanM$, respectively. $RanS$ represents a plan consisting of random star decomposition units (no limit on the size of the star) and $RanM$ represents plan with minimum number of rounds without considering the strategies in Sections 5.2 and 5.3. In order to cover more random query plans, we run each test 5 times and report the average. The queries are as shown in Figure 6. For queries q_1 to q_3, the query plans generated in the above three implementations are almost the same. Therefore, we omit the data for those three queries.



(a) Roadnet

(b) DBLP

(c) LiveJournal

(d) UK2002

Figure 11: Effectiveness of Execution Plan

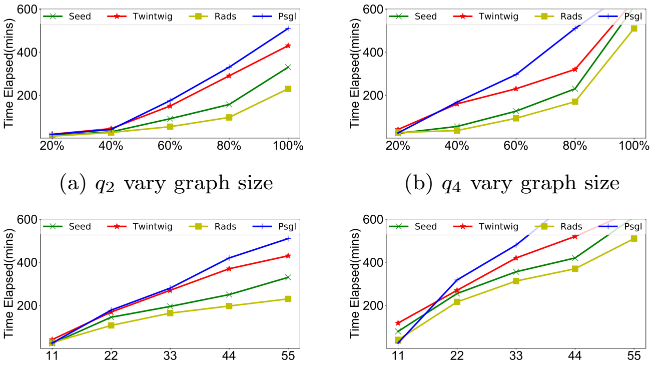
The results of Roadnet, DBLP, LiveJournal and UK2002 are shown in Figure 11. For RoadNet, it is not surprising to see that the processing time are almost the same for the 3 execution plans. This is because most vertices of each RoadNet partition can be processed by SM-E, and different distributed query execution plans have little effect on the total processing time. For all other three data sets, it is obvious that our fully optimized execution plan is playing an important role in improving the query processing time, especially when dealing with large graphs such as LiveJournal and UK2002 where large volumes of network communication are generated and can be shared.

8.4 Scalability Test

Varying graph size and average degree Following the method of [13], we generated 5 subgraphs of Friendster by extracting 20, 40, 60, 80 and 100% of the vertices. By fixing 100% of the vertices, we randomly sample 20, 40, 60, 80 and 100% of the edges to get another 5 subgraphs whose average degrees range from 11 to 55. We compared the results of `PSgL`, `SEED`, `TwinTwig` and `RADS` by testing two queries q_2 and q_4, the results are as shown in Figure 12. The index of `Crystal` is too large on Friendster, therefore `Crystal` is not included in this experiment.

As we can see in Figure 12, the time cost of all four methods significantly increases when we increase the graph size and the average degree. However, `RADS` is consistently better than the other 3 methods. For q_4, the three previous methods all failed to finish the processing on the complete Friendster graph due to memory crash.

Varying the number of nodes in the cluster We compared the five approaches by varying the number of nodes in the cluster from 5 to 10 and 15. Instead of reporting the

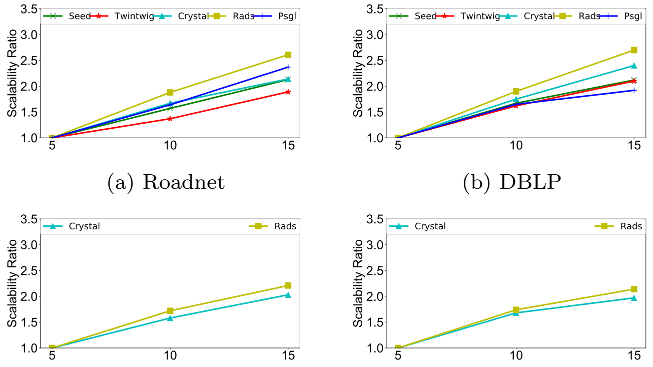


(a) q_2 vary graph size (b) q_4 vary graph size

(c) q_2 vary avg. degree (d) q_4 vary avg. degree

Figure 12: Results of varying graph size & avg.degree

processing time, here we report the ratio between the total processing time of all queries using 5 nodes and that of the other two cases, which we call *scalability ratio*. Because TwinTwig, SEED and PSgL failed some queries for LiveJournal and UK2002, we omit them in the two datasets. The result is shown in Figure 13. The most important thing to observe is that our approach demonstrates linear speedup when the number of nodes is increased for Roadnet and DBLP. For LiveJournal and UK2002 the difference between Crystal and RADS is not much while RADS is better for both.



(a) Roadnet (b) DBLP

(c) LiveJournal (d) UK2002

Figure 13: Vary Number of Nodes

9. RELATED WORK

The works most closely related to ours are TwinTwig [12], SEED [13] and PSgL [20]. Both [12] and [13] use multi-round two-way joins. [12] uses the same data partitioning as in our work, and it decomposes the query graph P into a set of small trees $dp_0, \ldots, dp_k$ such that the union of these trees is equal to P. The union of their embeddings on all machines is the set of all embeddings of dp_i over G. [13] is similar to [12], except that it allows decomposition units to be cliques as well as trees, and it uses bushy join rather than left-deep joins[2]. Both TwinTwig and SEED may generate huge intermediate results, and shuffling, re-distribution and synchronization cost a lost of time. Instead of using joins, we follow expand-verify-filter on each machine, as such we generate less intermediate results, and we do not need to re-distribute them to different machines.

PSgL [20] is based on Pregel [15]. It maps the query vertices one at a time following breath-first traversal. However, there are important differences between PSgL and our system (RADS). (1) In each step of expansion, PSgL needs to shuffle and send the partial matches (intermediate results) to other machines, while RADS does not need to do so. (2) PSgL stores each (partial) match as a node of a *static* result tree, while RADS stores the results in a dynamic and compact data structure. (3) There is no memory control in PSgL.

Also closely related to our work are [5] and [4], which introduce systems for parallelizing serial graph algorithms, including (but not limited to) subgraph isomorphism search. The work [1] treats the query pattern as a conjunctive query, where each predicate represents an edge, and computes the results as a multi-way join in a single round of *map* and *reduce*. The problem with [5][4][1] is that a large part of data graph have to be duplicated over all the machines which limits their practicality when the query pattern is complex.

Qiao *et al* [16] represent the set I_P of all embeddings of pattern P in a compressed form, $code(I_P)$, based on a minimum vertex cover of P. It decomposes the query graph P into a core $core(P)$ and a set of so-called *crystals* $\{p_1, \ldots, p_k\}$, such that $code(I_P)$ can be obtained by joining the compressed results of $core(P)$ and $\{p_1, \ldots, p_k\}$. The compressed results of $core(P)$ and the crystals can be obtained from the compressed results of components of P. It needs to build an index of all cliques of the data graph, as shown in Table 2. Although no shuffling of intermediate results is required, the indexes of [16] can be many times larger than the data graph.

BigJoin, one of the algorithms proposed in [2], treats a subgraph query as a join of $|E_P|$ binary relations where each relation represents an edge in P. Similar to RADS and PSgL, it generates results by expanding partial results a vertex at a time, assuming a fixed order of the query vertices. Different from our work, it still needs to shuffle and exchange intermediate results, and therefore synchronization before that.

10. CONCLUSION

We presented a novel approach for distributed subgraph enumeration. By processing the data vertices far away from the border using the single-machine algorithms, we isolated a large part of vertices which does not have to be involved in the distributed process. By passing edge verification requests/results and adjacency lists of foreign vertices, RADS significantly reduced the network communication cost. We also proposed a compact format to store the generated intermediate results. Our query execution plan and memory control strategies helped to improve the efficiency and robustness. Our experiments verified the superiority of our approach compared with several other approaches.

Acknowledgement Xuguang Ren and Junhu Wang were supported by the Australian Research Council (ARC) discovery grant DP130103051; Wook-Shin Han was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (No.NRF-2017R1A2B3007116) and Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1401-53. Jeffrey Xu Yu was supported by the Research Grants Council of Hong Kong, China, No. 14203618 and No. 14221716, and the Australian Research Council discovery grant DP130103051.

[2] There are different optimization strategies in each paper.

11. REFERENCES

[1] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *ICDE*, pages 62–73, 2013.

[2] K. Ammar, F. McSherry, S. Salihoglu, and M. Joglekar. Distributed evaluation of subgraph queries using worst-case optimal and low-memory dataflows. *PVLDB*, 11(6):691–704, 2018.

[3] R. J. Douglas. Np-completeness and degree restricted spanning trees. *Discrete Mathematics*, 105(1-3):41–47, 1992.

[4] W. Fan, P. Lu, X. Luo, J. Xu, Q. Yin, W. Yu, and R. Xu. Adaptive asynchronous parallelization of graph algorithms. In *SIGMOD*, pages 1141–1156, 2018.

[5] W. Fan, J. Xu, Y. Wu, W. Yu, J. Jiang, Z. Zheng, B. Zhang, Y. Cao, and C. Tian. Parallelizing sequential graph computations. In *SIGMOD*, pages 495–510, 2017.

[6] H. Fernau, J. Kneis, D. Kratsch, A. Langer, M. Liedloff, D. Raible, and P. Rossmanith. An exact algorithm for the maximum leaf spanning tree problem. *Theoretical Computer Science*, 412(45):6290–6302, 2011.

[7] J. A. Grochow and M. Kellis. Network motif discovery using subgraph enumeration and symmetry-breaking. In *RECOMB*, volume 4453, pages 92–106, 2007.

[8] W. Gropp. MPICH2: A new start for mpi implementations. In *PVM/MPI*, pages 7–7, 2002.

[9] W.-S. Han, J. Lee, and J.-H. Lee. TurboIso: towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, pages 337–348, 2013.

[10] G. Karypis and V. Kumar. Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

[11] H. Kim, J. Lee, S. S. Bhowmick, W. Han, J. Lee, S. Ko, and M. H. A. Jarrah. DUALSIM: parallel subgraph enumeration in a massive graph on a single machine. In *SIGMOD*, pages 1231–1245, 2016.

[12] L. Lai, L. Qin, X. Lin, and L. Chang. Scalable subgraph enumeration in mapreduce. *PVLDB*, 8(10):974–985, 2015.

[13] L. Lai, L. Qin, X. Lin, Y. Zhang, and L. Chang. Scalable distributed subgraph enumeration. *PVLDB*, 10(3):217–228, 2016.

[14] J. Lee, W. Han, R. Kasperovics, and J. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.

[15] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *PODS*, page 6, 2009.

[16] M. Qiao, H. Zhang, and H. Cheng. Subgraph matching: on compression and computation. *PVLDB*, 11(2):176–188, 2017.

[17] X. Ren and J. Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. *PVLDB*, 8(5):617–628, 2015.

[18] X. Ren, J. Wang, W.-S. Han, and J. X. Yu. Fast and robust distributed subgraph enumeration. *arXiv preprint arXiv:1901.07747*, 2019.

[19] B. Schling. *The Boost C++ Libraries*. XML Press, 2011.

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