DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine

Hyeonji Kim ‡
hjkim@dblab.postech.ac.kr

Juneyoung Lee ‡
jylee@dblab.postech.ac.kr

Sourav S. Bhowmick †
assourav@ntu.edu.sg

Wook-Shin Han ‡*
wshan@dblab.postech.ac.kr

JeongHoon Lee ‡
jhlee@dblab.postech.ac.kr

Seongyun Ko ‡
syko@dblab.postech.ac.kr

Moath H.A. Jarrah ‡
moath@dblab.postech.ac.kr

‡Pohang University of Science and Technology (POSTECH), Korea
†Nanyang Technological University, Singapore

ABSTRACT

Subgraph enumeration is important for many applications such as subgraph frequencies, network motif discovery, graphlet kernel computation, and studying the evolution of social networks. Most earlier work on subgraph enumeration assumes that graphs are resident in memory, which results in serious scalability problems. Recently, efforts to enumerate all subgraphs in a large-scale graph have seemed to enjoy some success by partitioning the data graph and exploiting the distributed frameworks such as MapReduce and distributed graph engines. However, we notice that all existing distributed approaches have serious performance problems for subgraph enumeration due to the explosive number of partial results. In this paper, we design and implement a disk-based, single machine *parallel subgraph enumeration solution* called DUALSIM that can handle massive graphs without maintaining exponential numbers of partial results. Specifically, we propose a novel concept of the *dual approach* for subgraph enumeration. The dual approach swaps the roles of the data graph and the query graph. Specifically, instead of fixing the matching order in the query and then matching data vertices, it fixes the data vertices by fixing a set of disk pages and then finds all subgraph matchings in these pages. This enables us to significantly reduce the number of disk reads. We conduct extensive experiments with various real-world graphs to systematically demonstrate the superiority of DUALSIM over state-of-the-art distributed subgraph enumeration methods. DUALSIM outperforms the state-of-the-art methods by up to orders of magnitude, while they fail for many queries due to explosive intermediate results.

Keywords

Subgraph enumeration; dual approach; graph analytics

1. INTRODUCTION

Given an unlabeled query graph q and an unlabeled data graph g, the subgraph enumeration finds all occurrences of q in g. Subgraph enumeration is important for several applications such as subgraph frequencies [31], network motif discovery [2, 12, 18, 22, 34, 35], graphlet kernel computation [25], and studying the evolution of social networks [16]. Furthermore, triangle enumeration, a special case of the subgraph enumeration, facilitates applications such as clustering coefficient computation [33] and community detection [32].

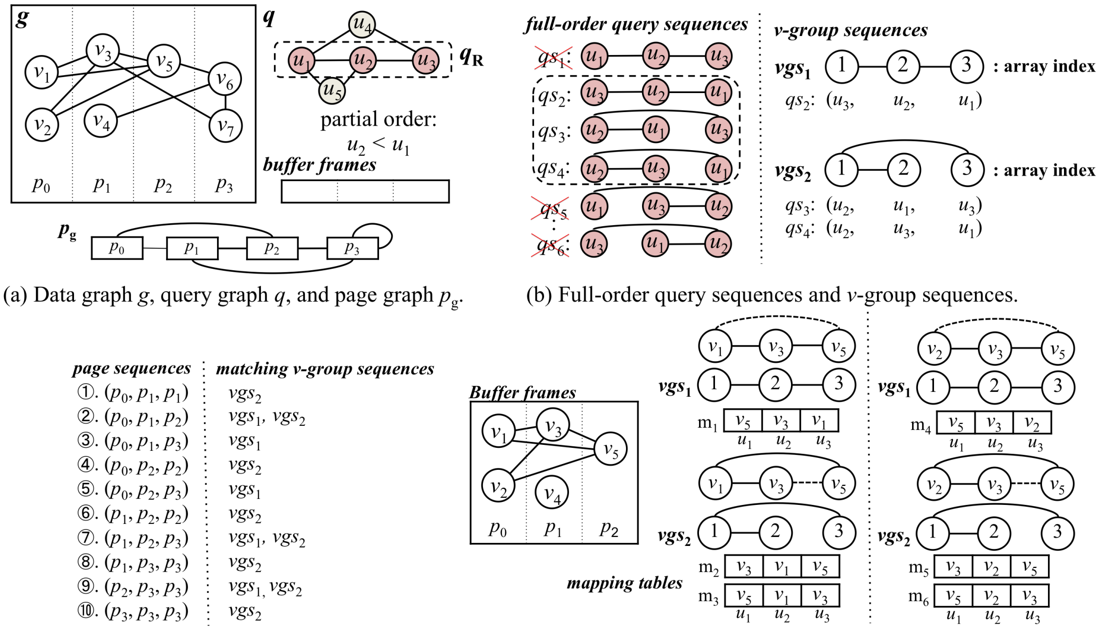
1.1 Motivation

Most of the earlier work on subgraph enumeration assumes that graphs are resident in memory [7, 12], which encounters serious scalability problems. Recently, efforts to enumerate all subgraphs in a large-scale graph have seemed to enjoy some success by partitioning the data graph and exploiting distributed frameworks such as MapReduce [1, 20] and distributed graph engines [24]. Afrati et al. [1] propose a seminal work for subgraph enumberation using single map-reduce round method. In order to do this, the method duplicates edges several times in multiple machines at the map phase so that each machine can perform a multi-way join independent of other machines. Although this multi-way join based method requires one map-reduce round, it is very likely to keep almost the entire graph in memory of each machine if the query graph becomes complex. In order to avoid expensive join operations, Shao et al. [24] proposed an enhanced method called PSGL which generates partial subgraph results level-by-level using the parallel breadth-first search (BFS) without performing join. Those partial results (a.k.a, *partial solutions*) are stored in the main memory of each machine. However, we show that the size of partial solutions grows exponentially as the number of query vertices increases. Thus, as we shall see later, this method fails to handle even moderate-size graphs for most of the queries we tested. More recently, Lai et al. [20] present Hadoop-based TWINTWIGJOIN, which is an enhanced map-reduce method that executes a left-deep join tree where each vertex in the tree corresponds to either a single edge or two incident edges of a vertex. However, this join-based approach can also be viewed as a variant of parallel BFS search. Thus, it is bound to generate an explosive number of partial solutions. As we shall show later, all existing distributed approaches have serious performance problems for subgraph enumeration.

Furthermore, although distributed computing resources are readily available through the Cloud nowadays, large-scale graph com-

[*] corresponding author

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
© 2016 ACM. ISBN 978-1-4503-3531-7/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915209



(a) Data graph g, query graph q, and page graph p_g.

(b) Full-order query sequences and v-group sequences.

page sequences	matching v-group sequences
①. (p_0, p_1, p_1)	vgs_2
②. (p_0, p_1, p_2)	vgs_1, vgs_2
③. (p_0, p_1, p_3)	vgs_1
④. (p_0, p_2, p_2)	vgs_2
⑤. (p_0, p_2, p_3)	vgs_1
⑥. (p_1, p_2, p_2)	vgs_2
⑦. (p_1, p_2, p_3)	vgs_1, vgs_2
⑧. (p_1, p_3, p_3)	vgs_2
⑨. (p_2, p_3, p_3)	vgs_1, vgs_2
⑩. (p_3, p_3, p_3)	vgs_2

(c) A page mapping example.

(d) A vertex mapping example for the page sequence (p_0, p_1, p_2).

Figure 1: A running example.

putation often demands judicious partitioning of the graph across cluster machines that minimizes communication between the machines, which is known to be a hard problem [21]. From the end users' perspective, debugging and optimizing distributed algorithms is hard [19]. Additionally, it is financially expensive for ordinary users or a small research group to purchase and maintain such cluster nodes. For example, it is highly unlikely that a biologist would invest in a distributed framework to discover motifs in a protein-protein interaction (PPI) network.

A natural question, therefore, is whether it is possible to devise a scalable and efficient *disk-based solution to the subgraph enumeration problem in a single machine?* In this paper, we provide an affirmative answer to this question. Specifically, we design and implement a disk-based, single machine *parallel subgraph enumeration* solution called **DUAL**ity-based **S**ubgraph Enumerat**I**on on a Single **M**achine (DUALSIM) that can handle massive graphs without maintaining exponential numbers of partial results.

1.2 Overview & Contributions

At first glance, it may seem that the depth-first search (DFS) strategy used in existing in-memory subgraph matching algorithms [7, 12] could be adopted to address the aforementioned excessive partial solution problem. Specifically, DFS-based subgraph matching algorithms fix the matching order of query vertices and recursively match one query vertex with data vertices one at a time, requiring minimal space to store the current partial solution. However, this strategy may incur considerable redundant disk reads since those data vertices are scattered across pages in disk. To illustrate this problem, consider the data graph g in Figure 1(a) composed of 7 vertices and 9 edges and the query graph q. Observe that g is stored in four disk pages (p_0 to p_3) but we have only three memory frames available in the buffer. Each vertex in the *page graph* p_g represents a page, and each edge (p_i, p_j) in p_g represents the fact that there is at least one edge across two pages. We assume that the subgraph q_R in q having only three vertices u_1, u_2, and u_3 is used for graph traversal. Consequently, we need to evict loaded disk pages whenever necessary by following a buffer replacement policy. As a result, this strategy will incur many disk page accesses to enumerate all occurrences of q_R!

In order to solve this challenging problem, we propose the novel concept of the *dual approach* for subgraph enumeration. The key feature of dual approach is that it swaps the roles of the data graph and the query graph. Specifically, instead of fixing the matching order in the query and then matching data vertices, it fixes the data vertices by fixing a set of disk pages and then finds all subgraph matchings in these pages (*i.e.,* matching query vertices). This enables us to significantly reduce the number of disk reads.

In the dual approach, before enumerating data subgraphs, we first need to enumerate all possible query sequences, each of which matches an ordered data sequence, thus making the data sequence fixed. These query sequences are called *full-order query sequences*. The full-order query sequence $(u_{i_1}, u_{i_2}, \cdots, u_{i_k})$ matches only with ordered data sequence $(v_{j_1}, v_{j_2}, \cdots, v_{j_k})$ such that $j_1 < j_2 < \cdots < j_k$. For example, in Figure 1(b), there are six full-order query sequences (qs_1 to qs_6). Suppose that we have a partial order constraint, $u_2 < u_1$. Then, qs_1, qs_5, and qs_6 are pruned. When a full-order query sequence matches a page sequence $(p_{j_1}, p_{j_2}, \cdots, p_{j_k})$, the condition $j_1 \leq j_2 \leq \cdots < j_k$ must be satisfied. This is because a page can contain more than one data vertex.

The dual approach groups full-order query sequences by their topology in order to minimize CPU time. This group sequence is called a *v-group sequence*. Figure 1(b) shows examples of two v-group sequences vgs_1 and vgs_2.

Next, the dual approach traverses the data graph stored in pages based on v-group sequences. Figure 1(c) shows a series of page sequences for the two v-group sequences. First, the page sequence (p_0, p_1, p_1) matches vgs_2, and p_0 and p_1 are now in the buffer. Next, the page sequence (p_0, p_1, p_2) matches both vgs_1 and vgs_2. In this way, we match the page sequence with a set of v-group sequences.

After matching a page sequence with v-group sequences, we find valid data vertex mappings for each v-group sequence. For each valid mapping, we enumerate all full-order query sequences in the

v-group sequence and generate data vertex mappings for them. For example, in Figure 1(d), a page sequence (p_0, p_1, p_2) matches vgs_1 and vgs_2. Thus, in the vertex level mapping, we can find a vertex-level matching sequence (v_1, v_3, v_5). Since there is only one full-order query sequence in vgs_1, a full solution $\{(u_3, v_1), (u_2, v_3), (u_1, v_5)\}$ is obtained. Similarly, since there are two full-order query sequences in vgs_2, two full solutions $\{(u_2, v_1), (u_1, v_3), (u_3, v_5)\}$, $\{(u_2, v_1), (u_3, v_3), (u_1, v_5)\}$ are obtained. Similarly, for the vertex-level matching sequence (v_2, v_3, v_5), we also generate three solutions.

Another interesting issue we need to consider is how to select a subset of query vertices for efficient graph traversal. Note that this has a similar flavor to the external triangulation problem where only two query vertices are used for matching data vertices in a database [15, 17]. Specifically, reducing the number of query vertices for graph traversal should reduce the number of disk accesses. This is because in order to match the remaining query vertices, we can utilize the adjacent lists which are already retrieved from disk. For example, reconsider Figure 1(a). Suppose that query vertices u_1 and u_2 are being matched with data vertices v_i and v_j, respectively. Then, we can identify data vertices for u_5 by intersecting the adjacent lists of v_1 and v_2.

In order to handle this issue, we propose the concept of the *red-black-ivory (*RBI*) graph* which is a vertex-colored graph where a color is assigned to a query vertex according to the operational semantics. If a query vertex u is colored *r*ed, we need to *r*etrieve the adjacency list of a data vertex for matching. If u is adjacent to m (>1) red query vertices, u is colored *i*vory. Consequently, in order to identify corresponding data vertices for u, we need to perform an (m-way) *i*ntersection of the adjacency lists of data vertices for the m red query vertices. If u is adjacent to only one red query vertex, u is colored *b*lack. In order to identify corresponding data vertices for u, we need to *b*rowse (or scan) the adjacency list of the data vertex corresponding to the red query vertex. For example, in Figure 1(a), we have 5 query vertices and 6 edges. In this case, only u_1, u_2, and u_3 are colored red (also called *red query graph*) and are used for graph traversal. The other two query vertices u_4 and u_5 are colored ivory. Thus, we need to perform 2-way intersections for identifying corresponding data vertices for u_4 and u_5. For example, once the two adjacency lists for u_1 and u_3 are obtained, we can obtain corresponding data vertices for u_4 by intersecting the two adjacency lists.

In summary, this paper makes the following key contributions.

- We propose an efficient and scalable disk-based subgraph enumeration method in a single machine for large-scale graphs. Unlike existing methods, DUALSIM does not maintain explosive partial solutions resulting in robust performance.
- We propose the novel concept of *dual approach* for subgraph enumeration. The dual approach uses novel concepts of *v-group sequence* and *v-group forest* in order to minimize processing time for subgraph enumeration. In addition, in order to realize the dual approach with a limited buffer size, we propose novel concepts of *candidate page sequence* and *current page window*. In order to overlap CPU processing with I/O processing, we propose novel concepts of *internal and external subgraphs*. All of these new concepts together lead to the concrete implementation of DUALSIM.
- We propose a technique that transforms a query graph to a RBI *query graph*, which enables us to lower the number of disk reads by matching part of the query graph only and utilizing their associated adjacency lists.
- We conduct exhaustive experiments using various real-world graphs to systematically demonstrate the superiority of DUALSIM over the-state-of-the-art subgraph enumeration methods. Specifically, DUALSIM using a single machine significantly outperforms TWINTWIGJOIN using a single machine by up to 866.61 times. Note that, TWINTWIGJOIN fails to process queries generating enormous partial solutions, while DUALSIM handles all queries without maintaining such explosive partial solutions. DUALSIM using a single machine also outperforms both PSGL and TWINTWIGJOIN using 51 machines by up to 35.04 and 903.47 times, respectively. Note that PSGL failed for 46.6% of experiments we tested due to enormous partial solutions.

1.3 Paper Organisation

The rest of the paper is organized as follows. In Section 2, we introduce fundamental graph concepts and formally define the subgraph enumeration problem addressed in this paper. We introduce the notions of the RBI query graph and the red query graph in Section 3 and explain how they are generated from a query graph. In Section 4, we introduce the dual approach using v-group sequence and v-group forest. In Section 5, we describe the algorithm of DUALSIM in detail. We present experimental results in Section 6 and review related work in Section 7. We conclude in Section 8.

2. BACKGROUND

A data graph $g(V, E)$ is modeled as an undirected and unlabeled graph where V is the set of vertices, and E ($\subseteq V \times V$) is the set of edges. A subgraph g' of $g(V, E)$ is a graph such that $V(g') \subseteq V(g)$ and $E(g') \subseteq E(g)$. Given a vertex v, we denote the degree of v as $d(v)$. $adj(v)$ denotes the adjacency list of v. $id(v)$ denotes the id of v. $P(v)$ denotes the page ID of the data vertex v. A query graph $q(V', E')$ is an undirected, unlabeled, and connected graph. We use u_i and v_j to represent the i-th query vertex and the j-th data vertex, respectively.

A graph $q(V', E')$ is *isomorphic* to a subgraph of a data graph $g(V, E)$ if there is an injection (or an embedding) $m : V \rightarrow V'$ such that $\forall (u_i, u_j) \in E$, $(m(u_i), m(u_j)) \in E'$.

When storing adjacency lists of data vertices in disk, we use the slotted page format, which is widely used in database systems. That is, each (v, $adj(v)$) for $v \in V$ is stored in pages. When the size of an adjacency list is larger than the page size, the adjacency list is broken into multiple sublists, and each sublist is stored in a page.

Following the convention in [15,17,20,24], we use the following *total order* $\prec$ for data vertices in g. For any two data vertices v_i and v_j, $v_i \prec v_j$ if and only if $d(v_i) < d(v_j)$ or $d(v_i) = d(v_j)$ and $id(v_i) < id(v_j)$. We rearrange the vertices in g according to $\prec$.

A *vertex cover* of an undirected graph $q(V_q, E_q)$ is a subset of vertices $V'_q \subset V_q$ such that for any edge $(u_i, u_j) \in E_q$, $u_i \in V'_q$, or $u_j \in V'_q$, or both [8]. For example, consider the query graph q in Figure 2. The vertex-covers of this graph are $\{u_1, u_4\}$, $\{u_2, u_3\}$, $\{u_1, u_2, u_3\}$, $\{u_1, u_2, u_4\}$, $\{u_2, u_3, u_4\}$, $\{u_1, u_3, u_4\}$, and $\{u_1, u_2, u_3, u_4\}$. A minimum vertex cover (MVC) is a vertex cover having the smallest number of vertices. A connected vertex cover is a vertex cover V'_q such that the induced subgraph using V'_q is connected [28]. If the size of a connected vertex cover is the minimum among all possible connected covers for q, the connected vertex cover is called a minimum connected vertex cover (MCVC) [28]. Reconsidering the above example, the vertex cover $\{u_1, u_2, u_3\}$ is an MCVC whereas $\{u_1, u_4\}$ is not.

In order to generate each subgraph exactly once, we need to eliminate the automorphism of a query graph q. Here, an automorphism of q is a graph isomorphism from q to itself [20]. Suppose that an embedding m is found. Then for any $u_i < u_j$ in *partial orders*, $m(u_i) \leq m(u_j)$ should be satisfied. Note that a set of partial or-

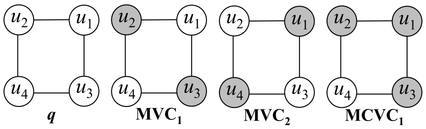


Figure 2: A query graph q, and two MVCs, and one MCVC.

ders can be obtained by breaking the automorphism of q [20, 24]. For example, if we have a triangle-shaped query having u_1, u_2, and u_3 as query vertices, partial orders $u_1 < u_2 < u_3$ can be obtained.

Definition 1. ***[Problem Statement]*** *Given a data graph g stored in disk, a query graph q, and a set PO of partial orders, the subgraph enumeration problem identifies all mappings of g that are isomorphic to q such that each mapping satisfies all partial orders in PO.*

3. RBI QUERY GRAPH

In order to minimize the number of disk reads, we need to use a minimum number of query vertices during graph traversal. Once adjacency lists for a subset of query vertices are retrieved from disk, we can match the remaining query vertices with only the adjacency lists retrieved. We realize this strategy by transforming the query graph q to an RBI *query graph* q_{RBI} by *coloring* its vertices.

In order to handle this issue, we propose the *red-black-ivory (*RBI*) query graph* which is a vertex-colored graph where a color is assigned to a vertex according to the operational semantics. If u is colored red, we need to retrieve the adjacency list of a data vertex for matching. If u is colored ivory, it is adjacent to m (>1) red query vertices. Consequently, in order to identify corresponding data vertices for u, we need to perform an (m-way) intersection of the adjacency lists of data vertices for the m red query vertices. If u is colored black, it is adjacent to only one red query vertex. Then, in order to identify corresponding data vertices for u, we need to browse (or scan) the adjacency list of data vertices for the red query vertex. Figure 3 depicts two examples of query graphs and RBI query graphs. In Figure 3(a), u_1 and u_2 are colored red. The query vertices u_4 and u_5 are colored ivory because they are adjacent to two red query vertices u_1 and u_2. On the other hand, u_3 is colored black because it is adjacent to u_2 only. In Figure 3(b), u_1, u_2, and u_3 are colored red. There is no black colored vertex because u_4 and u_5 are both connected to two red query vertices.

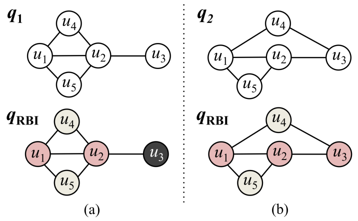


Figure 3: Two examples showing a query graph q and its RBI query graph q_{RBI}.

A *red query graph*(RQG) $q_R(V_R, E_R)$ is an induced subgraph of the red query vertices in $q_{RBI}(V_Q, E_Q)$ where $V_R \subset V_Q$ and $E_R \subset E_Q$. For example, in Figure 3(a), the RQG is an induced subgraph of query vertices $\{u_1, u_2\}$ and in Figure 3(b), it is an induced subgraph of query vertices $\{u_1, u_2, u_3\}$.

In order to select the smallest number of red query vertices, we can use the MVC. That is, all query vertices in a selected MVC are colored red. We can color the other query vertices based on the description above. Although the number of red query vertices increases slightly, we prefer to use the MCVC in this paper. This approach enables us to access data vertices reachable from the data vertices corresponding to the first red query vertex selected. Otherwise, we have to scan all vertices regardless of their reachability. This issue corresponds to the join versus cartesian product in relational databases. Although we use MCVC in this paper, the extension to support MVC is straightforward. Note that, while finding an MVC/MCVC is NP-hard, $|V_q|$ is generally very small, so its exponential complexity is not a problem in reality. We may use an approximate algorithm when $|V_q|$ is large [26].

A keen reader may observe that a query graph may contain more than one MCVC (or RQG). In this case, which one do we choose as a red query graph? In the following, we propose two rules which leverage partial orders and MCVCs, which can affect I/O time as well as CPU time.

Given a set of partial orders PO, a query graph $q(V_Q, E_Q)$ and an RQG (V_R, E_R), a partial order $(u_1 < u_2) \in PO$ is *an internal partial order* if $u_1 \in V_R$ and $u_2 \in V_R$.

Rule 1. *Given any two* RQG*s, we choose the* RQG *which has more internal partial orders.*

Rule 2. *If there is a tie in Rule 1, we choose a denser* RQG. *That is, we choose the* RQG *which has more edges.*

As shown in Figure 1(b), internal partial orders can prune full-order query sequences. Thus, Rule 1 can reduce candidate data vertices during graph exploration. Rule 2 can also reduce the CPU processing time during red vertex matching if one RQG has more edges than the other RQG.

4. DUAL APPROACH

Recall that in the dual approach, we swap the roles of the data graph and the query graph. Specifically, instead of fixing the matching order in the query and then matching data vertices, it fixes the data vertices by fixing a set of disk pages and then finds all subgraph matchings in these pages (*i.e.,* matching query vertices). This enables us to significantly reduce the number of disk reads.

Full-order query sequence. Before enumerating data subgraphs, the dual approach first must enumerate all possible query sequences, each of which matches an *ordered* data sequence, thus making the data sequence fixed. These query sequences are called *full-order query sequences*, which is formally defined below.

Definition 2. ***[Full-order query sequence]*** *Given an* RQG *$q_R(V_R, E_R)$ and a set of partial order PO, a full-order query sequence qs is a permuted sequence of vertices in V_R such that PO is a subset of the full order $(qs[1] < qs[2] < \cdots < qs[|V_R|])$ among query vertices in qs.*

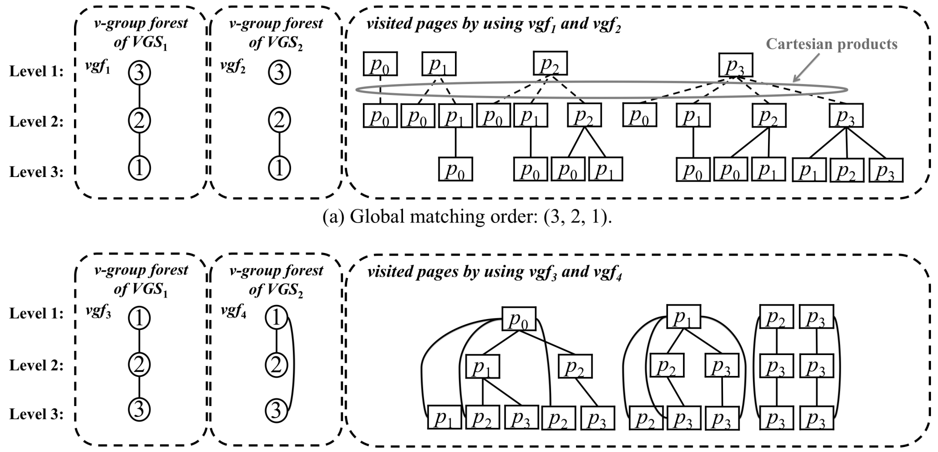
Property 1 states that a full-order query sequence matches an ordered data sequence of size $|V_R|$ element by element.

Property 1. *Given an* RQG *$q_R(V_R, E_R)$, when a full-order query sequence qs matches a data vertex sequence of size $|V_R|$ using a mapping (injection) m, $m[qs[1]] \prec m[qs[2]] \prec \cdots \prec m[qs[|V_R|]]$.*

Lemma 1. *Suppose that vertices are stored in pages according to $id(\cdot)$. Given an* RQG *$q_R(V_R, E_R)$, when a full-order query sequence qs matches a data vertex sequence of size $|V_R|$ using an injection m, $P(m[qs[1]]) \leq P(m[qs[2]]) \leq \cdots \leq P(m[qs[|V_R|]])$.*

PROOF. Refer to Appendix C. □

This way, we fix the page sequence of size $|V_R|$ in a non-decreasing order, varying all full-order query sequences. There can be many full-order query sequences of q_R, and hence, finding subgraph mappings for each such sequence may lead to a redundant matching process. To tackle this issue, we utilize the notion of *v-group sequence*.



(a) Global matching order: (3, 2, 1).

(b) Global matching order: (1, 2, 3).

Figure 4: An example of v-group forests.

v-group sequence. We first group the full-order query sequences by their topology based on *equivalence* of query sequences as defined below.

Definition 3. ***[Query Sequence Equivalence]*** *Let QS be the set of all full-order query sequences for an* RQG $q_R(V_R, E_R)$. *Let $\cong$ be an equivalent relation over QS such that $qs_i \cong qs_j$ if and only if $\forall 1 \leq k, k' \leq |V_R|$, if $(qs_i[k], qs_i[k']) \in E_R$, $(qs_j[k], qs_j[k'])$ $\in E_R$.*

The equivalent class of qs_i is a set of full-order query sequences which are equivalent to (*i.e.,* $\cong$) qs_i. This class is called a *v-group sequence*, since all full-order query sequences in a v-group sequence match the same data vertices. For example, in Figure 1, $qs_3(u_2, u_1, u_3)$ and $qs_4(u_2, u_3, u_1)$ constitute a v-group sequence. Thus, when (v_1, v_3, v_5) matches qs_3, it also matches qs_4.

Once all v-group sequences are obtained, we need to determine a matching order for each of them in order to traverse the data graph, *i.e.,* pages in the database. Since there can be multiple v-group sequences, we may traverse the data graph as many times as the number of v-group sequences. Thus, we provide a method for determining a global matching order which is used to traverse the data graph (*i.e.,* pages in disk) based on all v-group sequences. The term vgs_i denotes the i-th v-group sequence, and $\{vgs_i\}$ is a set of all v-group sequences.

v-group forest. In order to solve the explosive partial solution size problem, when we traverse the data graph, we maintain candidate vertices for each red query vertex rather than saving all possible partial solutions. For this, we construct an acyclic traversal structure called a *v-group forest* for each v-group sequence. This way, the total size of partial solutions for each v-group sequence is bounded by O($|V_R| * |V_G|$). If we use a bitmap, its size is significantly smaller than the graph size.

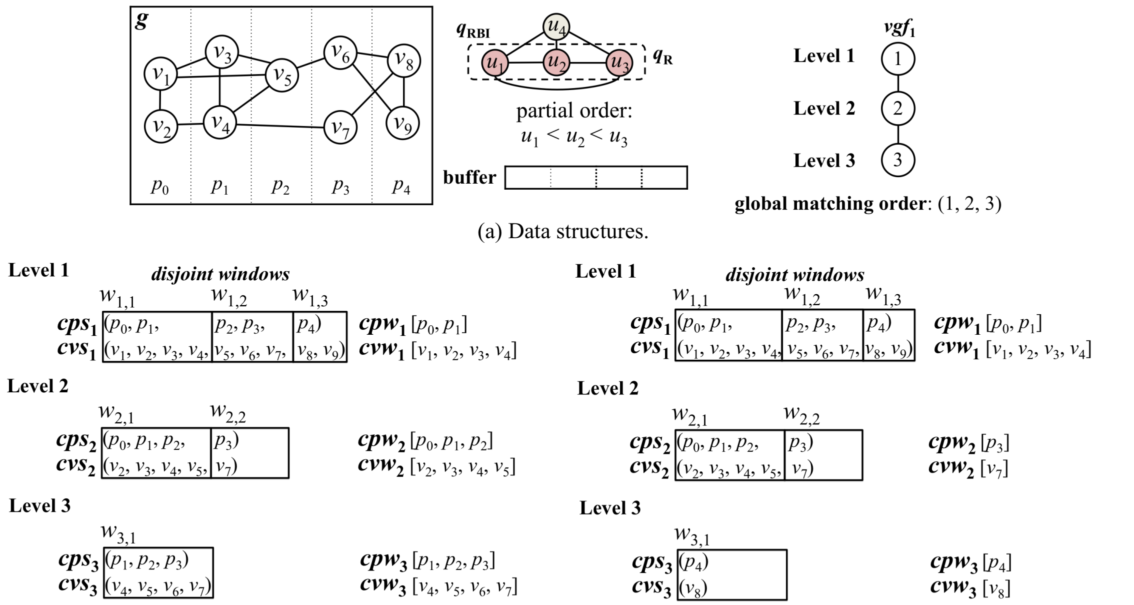
Intuitively, each node in a v-group forest stores an array index for the v-group sequences. Each node is associated with its level. At each level, we have only one node. Thus, the number of levels is equal to $|V_R|$. Note that the array index is also used to enforce the total order in the v-group sequences. Figure 4(a) shows the v-group forest for vgs_1 when the global matching order of array indexes is (3, 2, 1). There is only one tree in this forest having two edges, while the v-group forest for vgs_2 consists of two trees. In the second v-group forest, the node storing 2 as the array index is not connected from the node storing 3. The node storing 1 as the array index can be a child of the other two nodes; thus, we choose the one which is farthest from its root node with respect to the length of the path from the root node. The term vgf_i denotes the v-group forest for vgs_i, and $\{vgf_i\}$ is a set of all v-group forests. The term $vgf_i[j]$ denotes the node at level j of vgf_i.

Data graph traversal. Given a global matching order, we explain how we traverse the data graph considering the matching order as well as the topology information of every v-group forest. In order to do this, for each v-group forest, we first determine a set of candidate pages to explore at level l by considering its topology. Then, a set of pages to explore at level l should be a union of the sets of candidate pages for the nodes at level l of all v-group forests. Figure 4(a) shows the visited pages by using two v-group forests, vgf_1 and vgf_2. At the first level, we can match any page (p_0 to p_3) in the database. For each page at the first level, we need to access pages in the next level, considering the topologies of both v-group forests. If we consider only the first v-group forest, we only need to access pages connected from the page at the first level. However, if we consider the second v-group forest, we need to consider all pages for each page retrieved in the first level, incurring a Cartesian product. Note that the dotted lines between pages represent the Cartesian product. Thus, the set of pages to retrieve at the second level should be a union of candidate pages from both v-group forests. Note that, even in this case, we can still exploit the total order in each v-group sequence. Therefore, in this example, p_0 at level 1 does not match p_1, p_2, p_3 at level 2, while p_3 at level 1 matches p_0 to p_3. Figure 4(b) shows the visited pages by using vgf_3 and vgf_4 derived from the global matching order (1, 2, 3). In contrast to Figure 4(a), there is no Cartesian product needed when we traverse the data graph.

Now, we explain how to select a global matching order. Different matching orders can generate different numbers of Cartesian products. Thus, we enumerate all possible matching orders and choose the one generating the least number of Cartesian products. Since the number of red query vertices is very small, such enumeration cost is negligible compared to the subgraph enumeration itself.

5. DUALITY-BASED SUBGRAPH ENUMERATION

DUALSIM traverses the data graph level by level. However, due to the limited buffer size, we can load a subset of pages in memory for each level. Here, we need to ensure that the same sets of pages are loaded only once in order to minimize the number of disk I/Os. For this purpose, we propose novel concepts of *current vertex/page*


g
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8
v_9
p_0
p_1
p_2
p_3
p_4
q_{RBI}
u_4
u_1
u_2
u_3
q_R
partial order:
$u_1 < u_2 < u_3$
buffer
vgf_1
Level 1
Level 2
Level 3
1
2
3
global matching order: (1, 2, 3)
Level 1
disjoint windows
$w_{1,1}$ $w_{1,2}$ $w_{1,3}$
cps_1 $(p_0, p_1,$ $p_2, p_3,$ $p_4)$
cvs_1 $(v_1, v_2, v_3, v_4,$ $v_5, v_6, v_7,$ $v_8, v_9)$
cpw_1 $[p_0, p_1]$
cvw_1 $[v_1, v_2, v_3, v_4]$
Level 2
$w_{2,1}$ $w_{2,2}$
cps_2 $(p_0, p_1, p_2,$ $p_3)$
cvs_2 $(v_2, v_3, v_4, v_5,$ $v_7)$
cpw_2 $[p_0, p_1, p_2]$
cvw_2 $[v_2, v_3, v_4, v_5]$
Level 3
$w_{3,1}$
cps_3 (p_1, p_2, p_3)
cvs_3 (v_4, v_5, v_6, v_7)
cpw_3 $[p_1, p_2, p_3]$
cvw_3 $[v_4, v_5, v_6, v_7]$
Level 1
disjoint windows
$w_{1,1}$ $w_{1,2}$ $w_{1,3}$
cps_1 $(p_0, p_1,$ $p_2, p_3,$ $p_4)$
cvs_1 $(v_1, v_2, v_3, v_4,$ $v_5, v_6, v_7,$ $v_8, v_9)$
cpw_1 $[p_0, p_1]$
cvw_1 $[v_1, v_2, v_3, v_4]$
Level 2
$w_{2,1}$ $w_{2,2}$
cps_2 $(p_0, p_1, p_2,$ $p_3)$
cvs_2 $(v_2, v_3, v_4, v_5,$ $v_7)$
cpw_2 $[p_3]$
cvw_2 $[v_7]$
Level 3
$w_{3,1}$
cps_3 (p_4)
cvs_3 (v_8)
cpw_3 $[p_4]$
cvw_3 $[v_8]$

(a) Data structures.

(b) When current page windows for $w_{1,1}$, $w_{2,1}$, $w_{3,1}$ are chosen. (c) When current page windows for $w_{1,1}$, $w_{2,2}$, $w_{3,1}$ are chosen.

Figure 5: An example of candidate vertex/page sequences and current vertex/page windows.

window and *candidate vertex/page sequence* in Section 5.1, which are used together with the concept of v-group forest in Section 4. When we load a set of pages for each level, we issue page requests in a sequential scan fashion without incurring random disk page accesses. This strategy is important especially when we use HDDs.

Subgraph enumeration is a CPU intensive task. Thus, the important issue is how to overlap CPU processing with I/O processing effectively. Specifically, as soon as a page is loaded at the last level, we issue I/O requests for the next pages and execute subgraph enumeration using pages in memory at the same time. This is how our overlapping strategy can effectively hide the I/O cost. For this purpose, we propose novel concepts of *internal and external subgraphs* under the dual approach in Section 5.2.

5.1 Candidate Sequence and Current Window

We traverse the data graph based on the v-group forest by retrieving vertices/pages level-by-level. Suppose that m_j buffer frames are allocated for the j-th level in vgf_i. Assume that there is one v-group forest for ease of explanation and then explain how to handle multiple v-group forests.

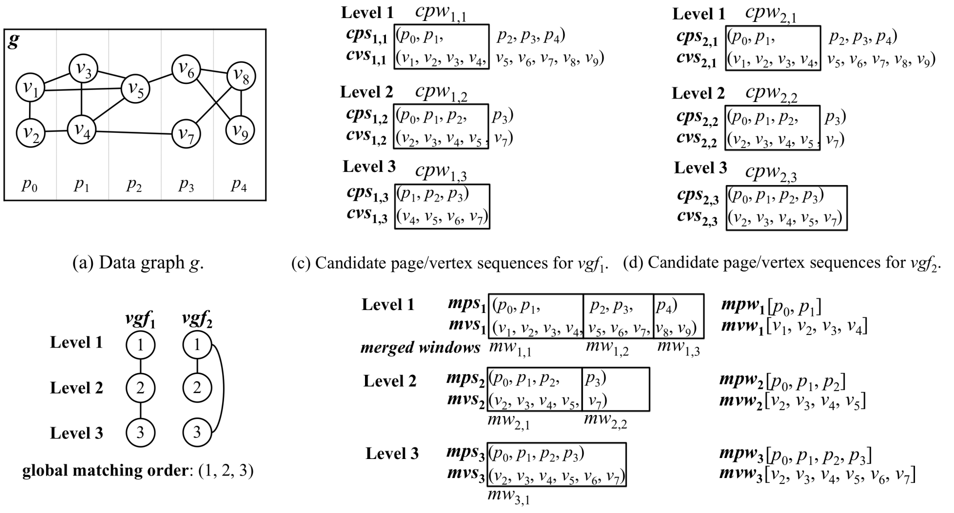
Candidate sequence and current window. A vertex/page window currently in the buffer for processing is called the current vertex window (CVW) and current page window (CPW). The candidate vertex/page sequence (CVS and CPS) for $vgf_i[j]$ is recursively defined as an ordered sequence of vertices/pages connected from the current vertex/page window corresponding to the parent node of $vgf_i[j]$. As a base case, the candidate vertex/page sequence at level 1 is defined as a sequence of all vertices/pages.

Now, we explain how we load a subset of pages, i.e., the current page window, from the candidate page sequence at each level so that we can ensure that the same arrangement of page sets for all levels is made only once. For this purpose, we divide the sequence into disjoint windows. Since some vertices/pages in the sequence can be in the buffer already, disjoint windows can be variably sized. Once disjoint windows are identified, we can iterate over them one by one. When we load a set of pages for each level, we issue page requests in a sequential scan fashion without incurring random disk page accesses.

Figure 5 shows an example of the current vertex window and the current page window at each level. The data graph is stored in five pages $p_0 \sim p_4$, while we have four memory buffer frames. Due to the partial order ($u_1 < u_2 < u_3$), we have only one v-group sequence and, accordingly, one v-group forest vgf_1. Consider Figure 5(b). At level 1, we need to retrieve all vertices. Thus, the candidate vertex sequence (denoted as cvs_1) is (v_1, v_2, $\cdots$, v_9), while the candidate page sequence (denoted as cps_1) is (p_0, p_1, $\cdots$, p_4). Suppose that two memory frames are allocated for the first level. Then, we load the first two pages p_0 and p_1 in the buffer. Thus, the current page window (cpw_1) is (p_0, p_1), while the current vertex window is (v_1, v_2, v_3, v_4). Once p_0 and p_1 are loaded, we can identify cps_2 and cvs_2. Here, cps_2 is (p_0, p_1, p_2, p_3), and cvs_2 is (v_2, v_3, v_4, v_5, v_7). Suppose that we allocate one memory frame for the second level. Then, the current page window at the second level (cpw_2) is (p_0, p_1, p_2), while the second page window is p_3. Note that p_0 and p_1 in cpw_2 are already in the buffer. After loading cpw_2, we can identify the candidate vertex/page sequences at the last level. Figure 5(c) shows CPSs, CVSs, CPWs, and CVWs at each level, when $w_{1,1}$, $w_{2,2}$, and $w_{3,1}$ are chosen as current page windows.

Merged sequence and current merged window. In general, there can be multiple v-group forests. Thus, if we process these v-group forests one by one, we are unable to ensure that the same arrangement of page sets for all levels is made only once. In order to avoid this problem, for each level, we first find all candidate sequences for all v-group forests and merge them into one sorted sequence, called *merged sequence*, removing any duplicate in the merged sequence. We then divide this merged sequence by disjoint windows called *merged window*. After loading a merged, disjoint window, we assign loaded vertices/pages to relevant v-group forests accordingly. This way, we can avoid repeated loading of pages from disk.

Figure 6 shows an example of merged vertex/page sequences and their windows at each level using g in Figure 5(a). Now we have two v-group forests vgf_1 and vgf_2 as shown in Figure 6(b). Figure 6(c) and Figure 6(d) represent CPSs and CVSs for each v-group forest and their current page windows. Consider Figure 6(e). The


g
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9
p_0 p_1 p_2 p_3 p_4
Level 1 $cpw_{1,1}$
$cps_{1,1}$ $(p_0, p_1,$ $p_2, p_3, p_4)$
$cvs_{1,1}$ $(v_1, v_2, v_3, v_4,$ $v_5, v_6, v_7, v_8, v_9)$
Level 2 $cpw_{1,2}$
$cps_{1,2}$ $(p_0, p_1, p_2,$ $p_3)$
$cvs_{1,2}$ $(v_2, v_3, v_4, v_5,$ $v_7)$
Level 3 $cpw_{1,3}$
$cps_{1,3}$ (p_1, p_2, p_3)
$cvs_{1,3}$ (v_4, v_5, v_6, v_7)
Level 1 $cpw_{2,1}$
$cps_{2,1}$ $(p_0, p_1,$ $p_2, p_3, p_4)$
$cvs_{2,1}$ $(v_1, v_2, v_3, v_4,$ $v_5, v_6, v_7, v_8, v_9)$
Level 2 $cpw_{2,2}$
$cps_{2,2}$ $(p_0, p_1, p_2,$ $p_3)$
$cvs_{2,2}$ $(v_2, v_3, v_4, v_5,$ $v_7)$
Level 3 $cpw_{2,3}$
$cps_{2,3}$ (p_0, p_1, p_2, p_3)
$cvs_{2,3}$ $(v_2, v_3, v_4, v_5, v_7)$
vgf_1 vgf_2
Level 1
Level 2
Level 3
1 2 3
global matching order: (1, 2, 3)
Level 1 mps_1 $(p_0, p_1,$ $p_2, p_3,$ $p_4)$
mvs_1 $(v_1, v_2, v_3, v_4,$ $v_5, v_6, v_7,$ $v_8, v_9)$
merged windows $mw_{1,1}$ $mw_{1,2}$ $mw_{1,3}$
$mpw_1[p_0, p_1]$
$mvw_1[v_1, v_2, v_3, v_4]$
Level 2 mps_2 $(p_0, p_1, p_2,$ $p_3)$
mvs_2 $(v_2, v_3, v_4, v_5,$ $v_7)$
$mw_{2,1}$ $mw_{2,2}$
$mpw_2[p_0, p_1, p_2]$
$mvw_2[v_2, v_3, v_4, v_5]$
Level 3 mps_3 (p_0, p_1, p_2, p_3)
mvs_3 $(v_2, v_3, v_4, v_5, v_6, v_7)$
$mw_{3,1}$
$mpw_3[p_0, p_1, p_2, p_3]$
$mvw_3[v_2, v_3, v_4, v_5, v_6, v_7]$

(a) Data graph g.

(c) Candidate page/vertex sequences for vgf_1.

(d) Candidate page/vertex sequences for vgf_2.

(b) V-group forests vgf_1, vgf_2.

(e) Merged vertex/page sequences and their current windows.

Figure 6: An example of merged vertex/page sequences and their windows.

merged vertex sequence for level i (denoted as mvs_i) is a merged sequence using $cvs_{1,i}$ and $cvs_{2,i}$. The merged page sequence for level i (denoted as mps_i) is a merged sequence using $cps_{1,i}$ and $cps_{2,i}$. Suppose that two memory frames are allocated for level 1. The current merged page window (denoted as mpw_1) is (p_0, p_1), while the current merged vertex window (mvw_i) is (v_1, v_2, v_3, v_4). Once p_0 and p_1 are loaded, we can identify $cps_{1,2}$, $cps_{2,2}$, and $cps_{2,3}$. All three candidate page sequences are (p_0, p_1, p_2, p_3) in this case, and thus mps_2 is (p_0, p_1, p_2, p_3). We can also identify $cvs_{1,2}$, $cvs_{2,2}$, and $cvs_{2,3}$ as (v_2, v_3, v_4, v_5, v_7). Accordingly, mvs_2 is (v_2, v_3, v_4, v_5, v_7). Suppose that we allocate one memory frame for level 2. Then, the current merged page window (mpw_2) is (p_0, p_1, p_2) since p_0 and p_1 are already in the buffer. After loading mpw_2, we can identify the candidate vertex/page sequences at the last level of vgf_1 ($cvs_{1,3}$ and $cps_{1,3}$). If we allocate one memory frame for level 3, the current merged page window (mpw_3) is (p_0, p_1, p_2, p_3) since $p_0 \sim p_2$ are already in the buffer.

5.2 Internal and External Subgraphs

In order to support overlapped and parallel subgraph enumeration in our dual approach, we define concepts of *internal and external subgraphs*, which are generalized concepts of internal and external triangles [17]. Before introducing the concepts in detail, we first explain how we overlap CPU processing with I/O processing at a high-level. DUALSIM achieves two types of overlapping: 1) Once the current window at level 1 is loaded, we can start to find data subgraphs within the window (i.e., internal subgraphs). At the same time, we find subgraphs connected from the vertices in the current window at level 1 by traversing the data graph level by level as we explain in the previous section. Those subgraphs are called external subgraphs. 2) As soon as we load a page at the last level, we can find subgraphs using pages in memory. At the same time, we can issues I/O requests for the next pages at the last level. DUALSIM splits the buffer by the number of levels of the v-group forest. The j-th area in the buffer consists of m_j memory frames. The first area, corresponding to level 1, is called *internal area* while all the other areas together are called *external area*.

We assume that the adjacency list of each vertex can be loaded in each buffer area. This assumption is called the small-degree assumption [15]. However, what if we have vertices whose adjacency lists are larger than an area? This can easily be handled by using [15]. First, identify all such vertices. For each large-degree vertex u, load as many edges of u as the first area can hold. Then, we report all subgraphs that involve at least one edge in the first area. The key issue here is how to identify candidate vertex/page sequences from a partly loaded adjacency list. This can be achieved by scanning the incoming edges of all vertices in the data graph. If we can identify the candidate vertex/page sequence, we can use the dual approach. We then remove all edges in the first area from the data graph. We repeat this process until no large-degree data vertices remain.

We now define the concept of the internal subgraph and the external subgraph. If a data subgraph which matches the RQG is in the internal area, then such a subgraph is called an *internal subgraph*. The data subgraph matching the RQG is called *external subgraph* if there is at least one data vertex in the internal area and there is at least one data vertex in the external area. We assume that, after finding subgraphs which match the RQG, we further match the other query vertices. Figure 7 shows internal subgraphs and external subgraphs when the query graph is q in Figure 1. Suppose that $v_1 \prec v_2 \prec v_3$. The first two cases correspond to internal subgraphs while the others correspond to external subgraphs.

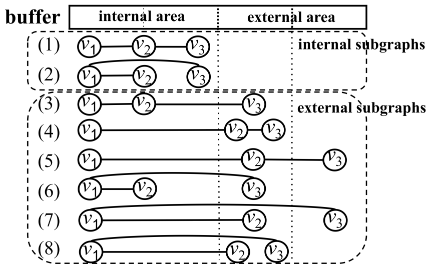


Figure 7: An example of internal and external subgraphs.

Note also that DUALSIM is a true generalization of the-state-of-the art triangulation [17], which handles the triangle-shaped query having two red query vertices. Thus, in our framework, the buffer space is divided into two areas. Like OPT [17], DUALSIM also supports overlapped and parallel subgraph enumeration. That is, as soon as a subset of disk pages is loaded into the internal area, we overlap internal subgraph enumeration with external subgraph enumeration. The internal subgraph enumeration finds all internal subgraphs in the pages loaded into the internal area, while the external subgraph enumeration finds all external subgraphs. Since we can use any in-memory subgraph enumeration algorithm for finding internal subgraphs, we focus on enumerating external subgraphs.

More importantly, during the external subgraph enumeration, we overlap CPU processing with I/O processing. That is, as soon as a page for the last level of the v-group forest is loaded from disk, DualSim finds external subgraphs using pages in memory, and issues asynchronous I/O requests for the next pages at the same time. This way, DualSim effectively hides I/O cost.

5.3 The DualSim Algorithms

DualSim consists of two steps: 1) the preparation step and 2) the execution step. Algorithm 1 summarizes how DualSim works. The elapsed time for the preparation step is at most 1 msec. for all queries we tested, which is negligible.

Algorithm 1. DualSim

```
Input: A data graph $g$, A query graph $q$
/* 1.  Preparation step                                      */
1:  $PO \leftarrow$ FindPartialOrders($q$);
2:  $\langle q_{RBI}, q_R \rangle \leftarrow$ GenerateRBIQueryGraph($q$, $PO$);
3:  $\{vgs_i\} \leftarrow$ FindVGroupSequences($q_R$, $PO$);
4:  $mo_g \leftarrow$ FindGlobalMatchingOrder($\{vgs_i\}$));
5:  $\{vgf_i\} \leftarrow$ BuildVGroupForests($\{vgs_i\}$, $mo_g$);
/* 2.  Execution step                                        */
6:  InitializeCandidateSequences(root nodes in $\{vgf_i\}$);
7:  foreach (merged vertex/page window $(mvw_1, mpw_1)$ from $\{vgf_i[1]\}$) do
8:      foreach (page id $pid \in mpw_1$) do
9:          AsyncRead($pid$, ComputeCandidateSequences($pid$,
              $\{cvw_{i,1}\}$, all child nodes of $\{vgf_i[1]\}$);
10:     end
11:     wait until ComputeCandidateSequences executions are finished
12:     $level \leftarrow 2$;
13:     DelegateExternalSubgraphEnumeration($level$, $q_{RBI}$,
          $\{vgs_i\}$, $\{vgf_i\}$, $\{mvw_j\}$, $\{mpw_j\}$);
14:     InternalSubgraphEnumeration($mvw_1$, $mpw_1$);
15:     UnpinPages($mpw_1$);
16:     ClearCandidateSequences(the children of $\{vgf_i[1]\}$);
17: end
```

In the preparation step, DualSim first finds a set PO of partial orders using the symmetry-breaking algorithm [12] (Line 1). Then, it generates q_{RBI} and q_R using PO (Line 2). Next, it finds all v-group sequences (Line 3). It then finds a global matching order considering all v-group sequences (Line 4). It next constructs a v-group forest for each v-group sequence using the global matching order (Line 5).

In the execution step, DualSim initializes candidate vertex/page sequences for all root nodes in the v-group forests (Line 6). This initialization makes every root node correspond to all vertices/pages. Then, we traverse the data graph level-by-level with the v-group forest by using the concept of merged vertex/page window (Lines 7 $\sim$ 16). Here, we use ComputeCandidateSequences($\cdot$) (in Appendix A) to compute the candidate vertex/page sequences for each page in the current merged window.

Specifically, DualSim first obtains the current, merged vertex/page window at level 1, mvw_1/mpw_1 (Line 7). For each page p in mpw_1, we asynchronously read the page (Line 8 $\sim$ Line 10). Note that we can pass a callback function when we request an asynchronous I/O. Here, the callback function, ComputeCandidateSequences($\cdot$) is invoked as soon as the requested page is loaded into the buffer. Since the current level is set to 1, the callback function computes the candidate vertex/page sequences for level 2. We need to wait until all requested pages are loaded in the buffer. Then, we invoke a recursive function, DelegateExternalSubgraphEnumeration($\cdot$) in order to find all external subgraphs connected from the current, merged vertex window (Line 13). This way, the main thread delegates the external subgraph enumeration to other threads. The main thread executes internal subgraph enumeration using the pages loaded in the internal area (Line 14). When either internal subgraph enumeration or external subgraph enumeration ends, then we perform *Thread Morphing* [17], which assigns the available threads to the unfinished subgraph enumeration. Execution of thread morphing is omitted in Algorithm 1 due to space limitation. This is how we overlap the internal subgraph enumeration with the external subgraph enumeration.

For example, consider Figure 6. mpw_1 of the first iteration (Line 8) is (p_0, p_1), and mvw_1 of the first iteration is (v_1, v_2, v_3, v_4). At the first iteration, two pages p_0 and p_1 in mpw_1 are loaded in the internal area by calling AsyncRead. ComputeCandidateSequences($\cdot$) is invoked when each of the requested pages is loaded into the buffer. Then, ComputeCandidateSequences($\cdot$) computes $cvs_{1,2}$ with $adj(v)$ and $cps_{1,2}$ with $\{P(v')|v' \in adj(v)\}$ for each v in $cvw_{1,1}$. This function also computes $cvs_{2,2}$, $cps_{2,2}$, $cvs_{2,3}$, and $cps_{2,3}$. We then delegate the external subgraph enumeration to the other threads by calling DelegateExternalSubgraphEnumeration($\cdot$). After delegation of external subgraph enumeration, the main thread invokes InternalSubgraphEnumeration($\cdot$). After executions for both functions are finished, the pages p_0 and p_1 are unpinned for page replacement. These steps are repeated until all merged vertex/page windows are processed.

Now, we explain DelegateExternalSubgraphEnumeration($\cdot$). The inputs to this function are l, q_{RBI}, $\{vgs_i\}$, $\{vgf_i\}$, mvw_l, and mpw_l: Here, l is the current level to process: We have a loop over merged vertex/page windows. At each iteration, we identify the current, merged vertex/page window, mvw_l, and mpw_l. If l is the last level, we invoke read requests with the callback ExtVertexMapping($\cdot$) (in Appendix A), which finds all vertex-level mappings using data vertices loaded in the buffer. That is, once the current vertex window at every level is loaded in the buffer, we can find the external subgraphs using ExtVertexMapping($\cdot$). At the same time, by invoking AsynRead($\cdot$) calls for the next pages, we can overlap CPU processing of ExtVertexMapping($\cdot$) with I/O processing. Here, since all internal subgraphs are enumerated in the internal area, this function avoids matching all red query vertices with data subgraphs in the internal area in order not to generate duplicate results. Instead, for each page in mpw_l, we request an asynchronous I/O with the callback function ComputeCandidateSequences($\cdot$). Then, we recursively invoke DelegateExternalSubgraphEnumeration($\cdot$) for the next level.

For example, consider Figure 6. Here, l is set to 2. Thus, at level 2, we compute the current merged vertex/page sequence, mpw_2 (= (p_0, p_1, p_2)) and mvw_2 (= (v_2, v_3, v_4, v_5)). We then request three asynchronous reads with the callback function ComputeCandidateSequences($\cdot$) for p_0, p_1, and p_2. We recursively delegate external subgraph enumeration to available threads. When DelegateExternalSubgraphEnumeration($\cdot$) is called at level 3 (the last level in this example), we request asynchronous reads with another callback function ExtVertexMapping($\cdot$) so that we can find external subgraphs, starting from each page loaded at level 3.

I/O Cost of DualSim. Now, we analyze the I/O cost of DualSim. Let $|E|$ be the number of edges in the data graph, M be the memory buffer size, and B be the page size. We assume that each edge is stored in one memory word, and M is equally divided into ($|V_R|$-1) regions. Note that we need only one memory frame for scanning pages at the last level. This additional memory frame is not considered in the asymptotic analysis. s_j represents the average reduction factor ($0 \leq s_j \leq 1$) for level j. Since we access only pages con-

Algorithm 2. DELEGATEEXTERNALSUBGRAPHENUMERATION

```
Input: Current level $l$, RBI query graph $q_{RBI}$, $v$-group sequences $\{vgs_i\}$,
       $v$-group forests $\{vgf_i\}$, merged current vertex windows $\{mvw_j\}$,
       merged current page windows $\{mpw_j\}$

foreach (merged vertex/page window $(mvw_l, mpw_l)$ from $\{vgf_i[l]\}$) do
    if ($l < |G_R|$) then
        foreach (page id $pid$ in $mpw_l$) do
            AsyncRead($pid$, COMPUTECANDIDATESEQUENCES($pid$,
              $\{cvw_{i,l}\}$, children of $\{vgf_i[l]\}$);
        wait until COMPUTECANDIDATESEQUENCES executions are
          finished;
        DELEGATEEXTERNALSUBGRAPHENUMERATION($l + 1$, $q_{RBI}$,
          $\{vgs_i\}$, $\{vgf_i\}$, $\{mvw_j\}$, $\{mpw_j\}$);
        UNPINPAGES($mpw_l$);
        CLEARCANDIDATESEQUENCES(the children of $\{vgf_i[l]\}$);
    else
        foreach ($pid$ in $mpw_l$) do
            AsyncRead($pid$, EXTVERTEXMAPPING($pid$, $q_{RBI}$, $\{cvw_i\}$,
              $\{vgs_i\}$, $\{vgf_i\}$));
    end
end
```

nected from the pages loaded at level 1. Then, at level l, DUALSIM requires $s_1(=1) \times \frac{|E|}{\frac{M}{|V_R|-1}} \times s_2 \times \frac{|E|}{\frac{M}{|V_R|-1}} \times \cdots \times s_l \times \frac{|E|}{B}$ disk I/Os. Thus, the total I/O cost is the summation of disk I/Os for each level as follows.

$$\sum_{1 \le l \le |V_R|} \prod_{i=1}^{l} s_i \times \left(\frac{|E|}{\frac{M}{|V_R|-1}}\right)^{l-1} \times \frac{|E|}{B} \qquad (1)$$

Buffer allocation strategy. Until now, we assume that the buffer is divided into equal-sized memory frames. We propose a better buffer allocation strategy where only two buffer frames * # of threads are allocated for the last level, and two third of the remaining buffer frames are allocated for the first level and one third are equally divided for the other levels. One buffer frame is used when we load a page at the last level, while the other buffer frame is used for asynchronous I/O processing. When the number of red vertices is 2 (i.e., triangulation), all the remaining buffer frames are allocated for the first level. This allocation enables the effective overlapping of CPU processing with I/O processing. By using this strategy, the number of iterations for the first level is much smaller than the equal-sized allocation strategy, which is very effective especially when we use HDDs. We use this buffer allocation strategy for all experiments in Section 6. Experimental results show that DUALSIM effectively hide the expensive I/O cost for subgraph enumeration using either SSDs or HDDs.

6. EXPERIMENTS

In this section, we present the performance of DUALSIM. Specifically, the key goals of this experimental study are as follows. Additional performance results are reported in Appendix B due to space constraints.

- We show the insensitivity of the elapsed time of DUALSIM to varying buffer size (Section 6.2.2).
- We show the linear speedup of DUALSIM with an increasing number of CPU cores (Appendix B.1).
- We show significantly better performance of DUALSIM compared to the state-of-the art subgraph enumeration methods in a *single* machine (Section 6.2.3).
- We show significantly better performance of DUALSIM in a single machine compared to the state-of-the art distributed methods using 51 machines (Section 6.2.4). We also show that DUALSIM is the only method to enumerate subgraphs in massive graphs.

Table 1: Statistics of Datasets.

	WG	WT	UP	LJ	OK	WP	FR	YH
$\lvert V \rvert$	$0.9M$	$2.4M$	$3.8M$	$4.8M$	$3.1M$	$25M$	$66M$	$1.4B$
$\lvert E \rvert$	$8.6M$	$9.3M$	$33M$	$86M$	$234M$	$234M$	$3610M$	$12.9B$
$Size$	$115MB$	$140MB$	$504MB$	$1.28GB$	$3.47GB$	$4.07GB$	$66.9GB$	$263GB$

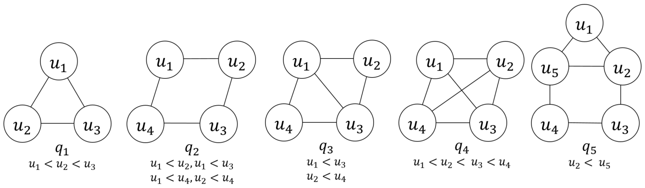


Figure 8: Queries.

6.1 Experimental Setup

Datasets/Queries. Eight real-world graph datasets were used in the experiments (WebGoogle (WG), WikiTalk (WT), USPatents (UP), LiveJournal (LJ), Wikipedia (WP), Friendster (FR), Yahoo (YH), and Orkut (OK)). All datasets can be downloaded from their original web sites. In order to show the performance over varying graph sizes, we generated different sizes by choosing 20%, 40%, 60%, 80%, and 100% of vertices from Friendster. Table 1 summarizes the statistics of these datasets. We use the same queries as in [24]. Figure 8 depicts the five queries.

Competitors. We compare the performance of DUALSIM with two state-of-the-art distributed subgraph enumeration frameworks, PSGL [24] and TWINTWIGJOIN [20]. We use binary files obtained from the authors of both methods. Note that since TWINTWIGJOIN is based on Hadoop, it can run in a single machine. PSGL failed in most experiments in a single machine environment due to memory overruns. Hence, we only discuss its performance in a distributed setting. In addition, we implemented two variants of TWINTWIGJOIN using PostgreSQL (TTJ-PG) and Spark SQL [4] (TTJ-SparkSQL) for comprehensive performance comparison. That is, DUALSIM is compared to TTJ-PG for single machine experiments and to TTJ-SparkSQL for cluster experiments. We also added experimental results using the Samsung 850 Pro 1TB SSD when DUALSIM is investigated in a distributed framework.

Running Environments. In our experiments, we use either a single-machine environment or distributed environment. In the single machine environment, we conducted experiments using two machines having identical hardware – an Intel Core i7-3930K CPU (6 CPU cores), 24GB RAM, and one 1TB HDD (Western Digital VelociRaptor). DUALSIM was executed on Windows 7, while TWINTWIGJOIN was executed on Linux since DUALSIM currently supports the Windows platform only, while TWINTWIGJOIN currently supports the Linux platform only. Note that due to a faster file system support in Linux, binaries executed on Linux are typically faster than those executed on Windows [14, 17]. In the distributed environment, we conducted experiments on a cluster of 51 machines, one master and 50 slaves, interconnected by an InfiniBand 40G network. Each machine has two Intel Xeon E5-2650 CPUs (a total of 16 cores), 32GB RAM, and one 300GB HDD (Hitachi Ultrastar). We use one machine in the cluster to execute DUALSIM. Note that using a high speed network is important for boosting the performance of distributed frameworks. Whenever we load a database page from disk, we bypass OS cache. Thus, every access to the database incurs a real disk I/O. This holds true even if we use small datasets.

Measure. To measure the cost, the elapsed time is used. To measure the parallelization effect, the speed-up is used, which is the elapsed time of the single thread execution over that of the multiple thread execution.

6.2 Experimental Results

We now report the performance of DualSim. Table 2 shows the various parameter values used for our experiments. Unless specified otherwise, values in boldface are used as default parameters in each experiment.

Table 2: Parameters used in the experiments.

Parameter	Description	Values Used
DS	data graphs	**WebGoogle**, **WikiTalk**, USPatents, **LiveJournal**, Orkut, Wikipedia, Friendster, Yahoo
q	query graphs	$\boldsymbol{q_1}$, q_2, q_3, $\boldsymbol{q_4}$, q_5
t	# of threads	1, 2, 3, 4, 5, **6** (single machine) **16** (cluster)
buf	buffer size (% of data graph size)	5, 10, **15**, 20, 25 (single machine)
s	# of slaves	8, 16, 32, **50** (cluster)

6.2.1 *Preprocessing Cost*

We first study the impact of preprocessing cost associated with subgraph enumeration techniques. Note that all existing subgraph enumeration methods use the degree-based ordering [1,20,24]. Table 3 reports the results for different datasets. Note that this preprocessing incurs an external sort of the original database, whose I/O cost is $O(n_p log n_p)$ where n_p is the number of pages in the database. At the last level in external-merge sort, we also update adjacency lists of all reordered vertices before they are finally written to disk. Thus, the preprocessing cost is theoretically cheaper than the cost for q_1. In fact, the preprocessing time is significantly smaller compared to the execution time of complex query patterns. For example, the processing time is 8.5 times smaller than processing q_4 for the FR dataset.

Table 3: Elapsed time of preprocessing (in sec.).

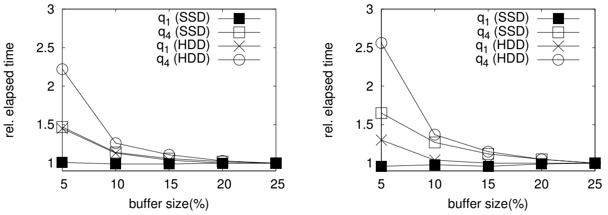
	WG	WT	UP	LJ	OK	WP	FR	YH
Preprocessing time	0.892	1.280	4.126	9.423	26.70	28.54	439.6	1951

Even if we consider dynamic graphs, we can still sort the database whose cost is typically smaller than the cost for processing q_1. Once we have a database and have small updates to the database, we do not need to reorder the whole database. In order to simulate evolving graphs, we make 95% of vertices fully sorted, and append 5% of them to the database. We tested with FR dataset using queries $q1$ and $q4$. The performance degradation is only 14.7% to 15.9%. Thus, in order to process complex queries (*e.g.,* q_4 and q_5), we always reorder the database as the reordering cost is significantly smaller than the cost for processing queries. Regarding q_1, we do not need to reorder the database until there is a significant number of changes to the database.

6.2.2 *Varying the Buffer Size*

To see the effect of the memory buffer size on the elapsed time, we varied it from 5% of the graph size to 25% in 5% increments. We measure the relative elapsed time of DualSim with a given memory buffer size compared to that with 25% of the graph size.

Figure 9 plots the relative elapsed times of DualSim using q_1 and q_4 for LJ and OK. DualSim showed almost constant performance except in the case where the buffer is very small (5% of the graph size). Specifically, for q_1, the relative elapsed times of DualSim remain 1 over varying buffer sizes. For the query q_4, when we reduce the buffer size from 25% to 5% (5 times), DualSim shows 2.22 and 2.56 times performance degradation. Overall, DualSim achieves robust performance even under the limited memory budget. When we use the SSD instead of the HDD, DualSim shows almost no performance degradation for q_1 while it shows at most 1.65 times the performance degradation for q_4 when the buffer size is set to 5% of the graph size.



(a) LJ.

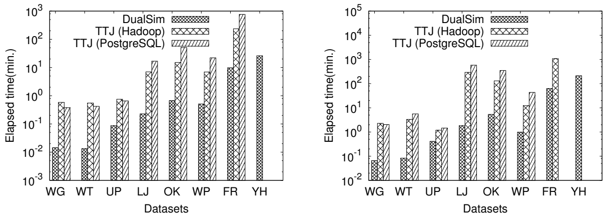
(b) OK.

Figure 9: Relative elapsed times for varying the buffer size.

6.2.3 *Comparison in a Single Machine Environment*

We now compare DualSim with TwinTwigJoin, which can execute subgraph enumeration in a single machine. Note that DualSim is configured to use a limited size of memory buffer (15% of the graph size), while TwinTwigJoin uses all available memory (*i.e.,* 24G memory size). We follow experimental parameters for TwinTwigJoin as in [24].

Varying datasets. Figure 10 shows the elapsed times of DualSim and TwinTwigJoin for q_1 and q_4 on different datasets. DualSim consistently shows better performance than TwinTwigJoin for all datasets. Specifically, DualSim outperforms both variants of TwinTwigJoin by up to 318.34 times even if DualSim uses much less memory. Also, note that TwinTwigJoin failed to process all queries on the largest data set (*i.e.,* YH).



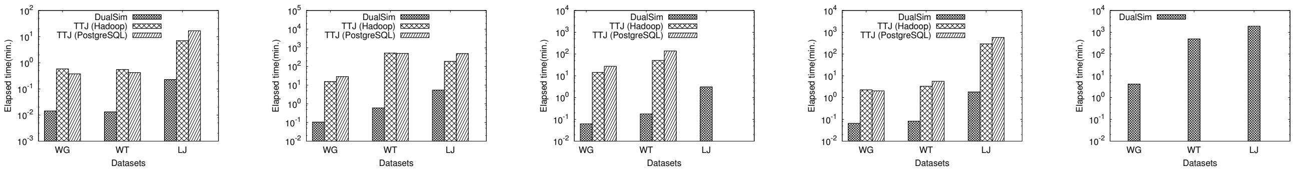
(a) Query q_1.

(b) Query q_4.

Figure 10: Varying datasets in a single machine.

The performance gain of DualSim is primarily due to two key causes. First, it enables significant reduction in CPU processing time using efficient graph traversal with the dual approach rather than expensive join operations. Second, it achieves significant reduction in the number of disk I/Os since DualSim does not need to write enormous partial results to disk at all. We shall report the estimated and actual sizes of partial results of TwinTwigJoin and PSgL in Appendix B.4. Although accurate estimation of intermediate result sizes is an interesting future research topic, it is beyond the scope of this paper; Table 5 in Appendix B.4 uses estimation formulae in [20, 24] in order to estimate the number of intermediate results. However, due to unrealistic assumptions, we will see that these formulae lead to significant estimation errors.

Note that PostgreSQL uses merge join extensively and hence must sort intermediate results. Consequently, if these results are stored in memory, TTJ-PG outperforms the Hadoop-based TwinTwigJoin, while it performs slower than the latter if the intermediate results need to be spilled to disk, which leads to external sort.



(a) Query q_1. (b) Query q_2. (c) Query q_3. (d) Query q_4. (e) Query q_5.

Figure 11: Varying queries in a single machine.

Varying queries. We now vary queries by fixing datasets. Figure 11 reports the elapsed times in a single machine for all queries using WG, WT, and LJ. Observe that DualSim shows much shorter elapsed times compared to TwinTwigJoin. DualSim outperforms TwinTwigJoin by up to 77.83, 866.61, 779.39, and 318.34 times for q_1, q_2, q_3, and q_4 respectively. Note that the binary executable of TwinTwigJoin fails to handle q_5.

It is worthwhile to note that DualSim significantly outperforms TwinTwigJoin as the number of solutions increases. For example, DualSim outperforms TwinTwigJoin by 866.61 times in WT using q_2. DualSim efficiently traverses the graph using the dual approach while TwinTwigJoin generates excessive partial solutions so that the join cost for them is very expensive. There is a relatively small number of solutions in UP for q_4. Thus, DualSim outperforms TwinTwigJoin by 3.54 times. Since WP is a bipartite graph, there is no solution for q_4 using WP. Even in this case, DualSim outperforms TwinTwigJoin by 44.23 times due to efficient graph traversal based on the dual approach. Lastly, observe that TwinTwigJoin fails to complete q_3 on LiveJournal. This is due to spill failure in Hadoop since TwinTwigJoin generates excessive partial results.

Varying graph size. Lastly, we demonstrate the scalability of DualSim in a single machine over varying graph sizes. In order to vary the number of vertices, we generate datasets by randomly sampling 20%, 40%, 60%, and 80% vertices from the original FR dataset according to [24]. Note that we choose FR over YH in this experiment because TwinTwigJoin failed to process queries on the latter dataset.

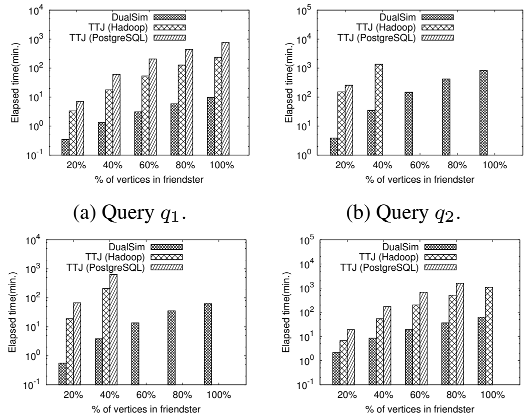
Figure 12 shows the performance over varying graph sizes. DualSim outperforms TwinTwigJoin in all cases. Furthermore, the performance gap between DualSim and TwinTwigJoin increases as the graph size increases. Regarding q_1, the performance gaps are 20.25, 46.17, 17.26, 75.35, and 24.13 respectively. TwinTwigJoin fails in several cases again. For q_2 and q_3, TwinTwigJoin fails to report results whenever the graph size is larger than 40% of the original FR.

6.2.4 Comparison on Distributed Environment

We now compare the performance of DualSim to the state-of-the-art distributed subgraph enumeration methods (PSgL and TwinTwigJoin). Recall that both distributed methods use 51 machines, while DualSim use only a single machine.

Varying datasets. Figure 13 plots the elapsed times of DualSim, PSgL, and TwinTwigJoin for various real graphs. In all cases, DualSim significantly outperforms competitors. Specifically, DualSim is faster than PSgL and TwinTwigJoin by up to 6.53 and 162.05 times for q_1, respectively. Regarding q_4, DualSim outperforms PSgL and TwinTwigJoin by up to 12.96 and 24.64 times, respectively.

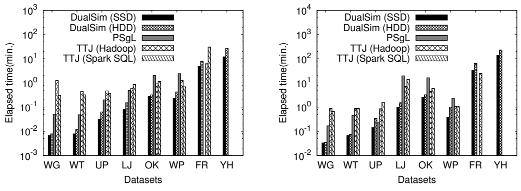
Observe that TTJ-SparkSQL typically performs better than Hadoop-based TwinTwigJoin when intermediate results can be stored in memory. Otherwise, the former performs slower than the latter when queries generate large intermediate results as Spark SQL must spill those results into disk. Importantly, TTJ-SparkSQL fails for some queries which generate a large size of intermediate results, where those results are divided into partitions across machines in a cluster. Specifically, it fails to process if the size of a partition block becomes larger than 2 GB. A typical workaround for this problem is to manually set a larger number of partition blocks. However, if we arbitrarily increase the number of partition blocks, the performance would significantly decrease. For instance, when on FR dataset, TTJ-SparkSQL fails with the default number of partition blocks. Even when we increase the number of partition blocks by up to 3200, it still fails due to the same problem. Both PSgL and all variants of TwinTwigJoin implementations fails to process this Yahoo dataset due to the out of memory problem.



(a) Query q_1. (b) Query q_2.

(c) Query q_3. (d) Query q_4.

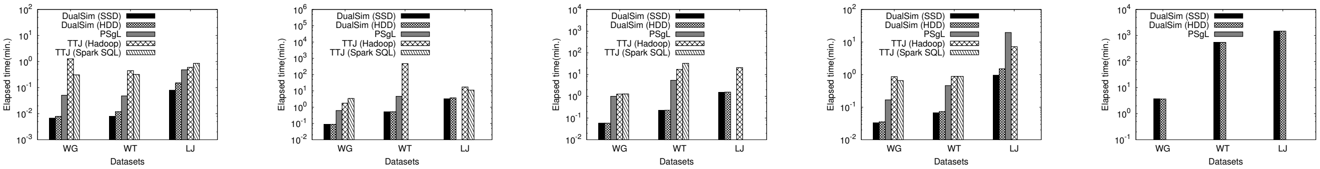
Figure 12: Varying graph size in a single machine.



(a) Query q_1. (b) Query q_4.

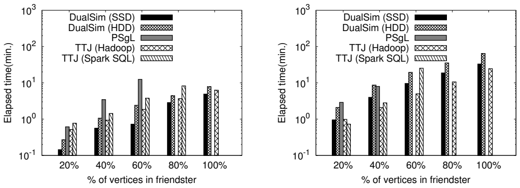
Figure 13: Varying datasets in a cluster.

Varying queries. Figure 14 shows elapsed times of DualSim, TwinTwigJoin, and PSgL for all queries using WG, WT, and LJ. Again, DualSim outperforms its competitors for all queries. Specifically, it outperforms TwinTwigJoin by up to 162.05, 903.47, 144.40, and 24.64 times for q_1, q_2, q_3, and q_4, respectively. DualSim outperforms PSgL by up to 6.53, 8.88, 23.87, and 12.96 times for q_1, q_2, q_3, and q_4, respectively. Recall that TwinTwigJoin cannot handle q_5. Furthermore, since PSgL maintains partial solutions in memory, PSgL fails to process q_2 and q_3 for LJ, and q_5 for all three datasets. In these cases, the sizes of partial solutions are enormous.



(a) Query q_1. (b) Query q_2. (c) Query q_3. (d) Query q_4. (e) Query q_5.

Figure 14: Varying queries in a cluster.



(a) Query q_1. (b) Query q_4.

Figure 15: Varying graph size in a cluster.

Varying graph size. Figure 15 shows elapsed times with varying the number of vertices in the data graph. DUALSIM outperforms PSGL and TWINTWIGJOIN by up to 5.34 and 2.86 times, respectively, for q_1. TWINTWIGJOIN using 51 machines shows better performance than DUALSIM using a single machine for q_4 since (a) TWINTWIGJOIN is optimized to process clique queries and (b) there are relatively small numbers of results for FR. Specifically, TWINTWIGJOIN requires two join operations for a clique query while DUALSIM requires three-level graph traversal. PSGL fails to complete queries for q_1 using 80% and 100% datasets, and fails to answer for q_4 using 60%, 80% and 100% datasets, respectively.

We also perform experiments using q_2 and q_3 for varying graph sizes (detailed in Appendix B.3). Note that all state-of-the-art techniques fail as the graph size increases whereas DUALSIM successfully processes all queries and outperforms TWINTWIGJOIN and PSGL by up to 5.27 and 35.04 times.

7. RELATED WORK

In-memory Methods. Most of the previous work on subgraph enumeration assumes that the graph would fit in main memory. [7] proposes a simple edge-searching based method with the time complexity of $O(\alpha(g)|E|)$ for three query shapes, where $\alpha(g)$ is the arboricity of the data graph g. [7] may incur significant disk reads if it is applied to external subgraph enumeration. [12, 34] propose enhanced algorithms compared to [7]. In contrast, in this paper we propose a scalable and efficient disk-based approach to address the subgraph enumeration problem.

Disk-based Methods. Many external triangulation methods [15, 17, 19, 29] have been proposed. However, all these methods only deal with triangle enumeration, a special case of the subgraph enumeration. [17] proposes an overlapped and parallel, disk-based triangulation framework, OPT. In this paper, we generalize OPT using the concept of the dual approach so that we can support any shaped queries efficiently.

Distributed Methods. Subgraph enumeration on a large-scale graph is computationally expensive, and thus various, distributed approaches have been proposed.

[23] proposes SGIA-MR, which is based on an edge-based join. However, a reducer can run out of memory since it could generate a large number of intermediate results [23]. [1] proposes a single map-reduce round method for subgraph enumeration. The method duplicates edges in multiple machines at the map phase so that each machine can perform independent subgraph enumeration. However, when the query graph is complex, it can cause out-of-memory error [20] due to the large amount of intermediate results. [20] is the-state-of-the-art map-reduce algorithm. It first partitions a query graph into a set of of TwinTwigs, each of which is an edge or two incident edges. It then constructs a left-deep join plan. [20] shows its superiority compared with [1, 23], but it still suffers from a large number of partial solutions.

[24] proposes a distributed subgraph enumeration method which is based on Apache Giraph. At each superstep, it expands partial subgraph instances. This method stores all partial solutions in memory. Thus, it easily fails for many queries due to memory overruns.

Our disk-based DUALSIM algorithm not only employs a novel dual approach on a *single machine* instead of a distributed setup to address the subgraph enumeration problem but also addresses the limitation of explosive intermediate result size effectively.

There are several, distributed triangulation methods. [11] proposes a method based on PowerGraph, while [3] proposes an MPI-based distributed algorithm. [30] proposes a map-reduce triangulation. [9] presents a distributed triangulation method using efficient external memory access.

Approximate Methods. There are several methods [2, 5, 6, 10, 36] which can estimate the number of subgraphs for a given query. Although they are efficient for massive graphs, such approximate counts may lead to erroneous conclusions [24].

8. CONCLUSION

In this paper, we proposed an efficient and scalable, disk-based subgraph enumeration framework, DUALSIM, in a single machine. For this framework, we proposed a novel notion of the dual approach for subgraph enumeration, which completely solves the notorious, excessive partial solution size problem. In order to efficiently traverse the data graph, we proposed the novel concepts of v-group forest and candidate data/page sequence. The dual approach together with these concepts minimizes the number of disk I/Os while it uses a bounded storage space for saving candidate results. We also proposed the concept of v-group sequence in order to avoid repeated pattern matching in the dual approach. We then proposed detailed parallel algorithms for DUALSIM which support overlapped and parallel subgraph enumeration under the dual approach. We also derived the cost model for DUALSIM. Experiments using various real datasets showed that DUALSIM significantly outperformed the-state-of-the-art methods both in a single machine and in a distributed environment. DUALSIM showed robust performance for all queries, while the-state-of-the-art methods failed for many queries we tested. Overall, we believe our overlapped and parallel subgraph enumeration provides comprehensive insight and a substantial framework for future research.

9. ACKNOWLEDGMENTS

This work was supported in part by an industrial gift from Microsoft. This work was also supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIP) (No. NRF-2014R1A2A2A01004454), the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the "ICT Consilience Creative Program" (IITP-2015-R0346-15-1007) supervised by the IITP(Institute for Information & communications Technology Promotion), and Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Plannig (NRF-2012M3C4A7033342). Sourav S Bhowmick was supported by the Singapore-MOE AcRF Tier-1 Grant RG24/12.

10. REFERENCES

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

[2] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp. Biomolecular network motif counting and discovery by color coding. In *ISMB*, pages 241–249, 2008.

[3] S. Arifuzzaman, M. Khan, and M. V. Marathe. PATRIC: a parallel algorithm for counting triangles in massive networks. In *CIKM*, pages 529–538, 2013.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[5] I. Bordino, D. Donato, A. Gionis, and S. Leonardi. Mining large networks with subgraph counting. In *ICDM*, pages 737–742, 2008.

[6] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting triangles in data streams. In *PODS*, pages 253–262, 2006.

[7] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM J. Comput.*, 14(1):210–223, 1985.

[8] B. Escoffier, L. Gourvès, and J. Monnot. Complexity and approximation results for the connected vertex cover problem in graphs and hypergraphs. *J. Discrete Algorithms*, 8(1):36–49, 2010.

[9] I. Giechaskiel, G. Panagopoulos, and E. Yoneki. Pdtl: Parallel and distributed triangle listing for massive graphs. Technical Report, University of Cambridge, 2015, https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-866.pdf.

[10] M. Gonen, D. Ron, and Y. Shavitt. Counting stars and other small subgraphs in sublinear time. In *SODA*, pages 99–116, 2010.

[11] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30, 2012.

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

[13] 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.

[14] W. Han, S. Lee, K. Park, J. Lee, M. Kim, J. Kim, and H. Yu. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD*, pages 77–85, 2013.

[15] X. Hu, Y. Tao, and C. Chung. Massive graph triangulation. In *SIGMOD*, pages 325–336, 2013.

[16] S. R. Kairam, D. J. Wang, and J. Leskovec. The life and death of online groups: predicting group growth and longevity. In *WSDM*, pages 673–682, 2012.

[17] J. Kim, W. Han, S. Lee, K. Park, and H. Yu. OPT: a new framework for overlapped and parallel triangulation in large-scale graphs. In *SIGMOD*, pages 637–648, 2014.

[18] W. Kim, M. Li, J. Wang, and Y. Pan. Biological network motif detection and evaluation. *BMC Systems Biology*, 5(S-3):S5, 2011.

[19] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a PC. In *OSDI*, pages 31–46, 2012.

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

[21] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[22] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[23] T. Plantenga. Inexact subgraph isomorphism in mapreduce. *J. Parallel Distrib. Comput.*, 73(2):164–175, 2013.

[24] 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.

[25] N. Shervashidze, S. V. N. Vishwanathan, T. Petri, K. Mehlhorn, and K. M. Borgwardt. Efficient graphlet kernels for large graph comparison. *AISTATS*, pages 488–495, 2009.

[26] J. Cardinal and E. Levy. Connected vertex covers in dense graphs. *Theor. Comput. Sci.*, 411(26–28):2581–2590, 2010.

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

[28] M. R. Garey and D. S. Johnson. The rectilinear Steiner tree problem is NP-complete. *SIAM Journal of Applied Mathematics*, 32(4), 826–834, 1977.

[29] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *KDD*, pages 672–680, 2011.

[30] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.

[31] J. Ugander, L. Backstrom, and J. M. Kleinberg. Subgraph frequencies: mapping the empirical and extremal geography of large graph collections. In *WWW*, pages 1307–1318, 2013.

[32] J. Wang and J. Cheng. Truss decomposition in massive networks. *PVLDB*, 5(9):812–823, 2012.

[33] D. J. Watts and S. H. Strogatz. Collective dynamics of ‘small-world‘ networks. *nature*, 393(6684):440–442, 1998.

[34] S. Wernicke. Efficient detection of network motifs. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 3(4):347–359, 2006.

[35] E. A. Wong and B. Baur. On network tools for network motif finding: a survey study, 2010.

[36] 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 *ICPP*, pages 594–603, 2010.

[37] P. Erdös and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci*, 5:17–61, 1960.

APPENDIX

A. ALGORITHMS DETAILS

COMPUTECANDIDATESEQUENCES$(\cdot)$ computes the candidate vertex/page sequences for all child nodes of $\{vgf_i[cur]\}$. It first pins page pid (Line 1). Then, for each v-group forest, for each data vertex v in page pid, we check if v is in the current vertex window, $cvw_{i,cur}$ (Lines 2 $\sim$ 4). If so, we compute candidate vertex/page sequences for all child nodes of $vgf_i[cur]$ (Lines 5 $\sim$ 7). All pinned pages in this function are unpinned in Algorithm 1 (for level 1) or Algorithm 2 (for the other levels except the last level).

Algorithm 3. COMPUTECANDIDATESEQUENCES

```
Input: page ID $pid$, the current vertex windows for current level
       $\{cvw_{i,cur}\}$, all child nodes of $\{vgf_i[cur]\}$

1:  Pin page $pid$;
2:  foreach ($vgf_i$) do
3:      foreach ($v$ in data vertices of page $pid$) do
4:          if ($v \in cvw_{i,cur}$) then
5:              foreach (the child node $n_c$ of $vgf_i[cur]$) do
6:                  add $adj(v)$ to $cvs$ for $n_c$;
7:                  add $\{P(v')|v' \in adj(v)\}$ to $cps$ for $n_c$;
8:              end
9:          end
10:     end
11: end
```

EXTVERTEXMAPPING$(\cdot)$ enumerates the vertex mappings for all possible external subgraphs, starting from page pid loaded at the last level. It first pins page pid (Line 1). Then, for each v-group forest, for each data vertex v in page pid, we check if v is in the current vertex window, $cvw_{i,cur}$ (Lines 2 $\sim$ 4). If so, we map v to the corresponding node $vgf_i[|V_R|]$ (Line 5). Note that we use another mapping m_{vgs} which can be used for enumerating all full-order query sequences in the v-group sequence. Then, we recursively map data vertices in the buffer to the remaining nodes in vgf_i by invoking RECEXTVERTEXMAPPING (Line 7). At the end of EXTVERTEXMAPPING, we unpin page pid (Line 12).

Algorithm 4. EXTVERTEXMAPPING

```
Input: Page id $pid$, RBI query graph $q_{RBI}$, current vertex windows
       $cvw$, $v$-group sequences $\{vgs_i\}$, $v$-group forests $\{vgf_i\}$

1:  Pin page $pid$;
2:  foreach ($vgf_i$) do
3:      foreach (data vertex $v$ in page $pid$) do
4:          if ($v \in cvw_{i,|V_R|}$) then
5:              $m_{vgs}[vgf_i[|V_R|]] \leftarrow v$;
6:              $idx \leftarrow 2$ ; // idx represents the index in
                  the matching order.
7:              ; RECURSIVEEXTVERTEXMAPPING($idx, m_{vgs}$,
                  $q_{RBI}, cvw_i, vgs_i, vgf_i$);
8:              $m_{vgs}[vgf_i[|V_R|]] \leftarrow$INVALID;
9:          end
10:     end
11: end
12: Unpin page $pid$;
```

RECEXTVERTEXMAPPING$(\cdot)$ recursively maps data vertices in the buffer to the remaining nodes in vgf_i. It basically follows GenericQueryProc in [13] using the matching order qo_i. The main differences are that 1) RECEXTVERTEXMAPPING maps all red query vertices to the corresponding data vertices, and then maps non-red query vertices and 2) RECEXTVERTEXMAPPING uses two-level mapping functions (m_{vgs} and m) to reduce the CPU processing time. U_{CON} represents a set of query vertices where each query vertex is already matched and connected from $qo_i[idx]$.

Algorithm 5. RECEXTVERTEXMAPPING

```
Input: Current matching index $idx$, $v$-group sequence mapping
       $m_{vgs}$, RBI query graph $q_{RBI}$, $vgf_i$'s current vertex window
       $cvw_i$, $v$-group sequence $vgs_i$, $v$-group forest $vgf_i$

1:  $U_{CON} \leftarrow adj(qo_i[idx]) \cap qo_i[1 : idx - 1]$;
2:  foreach ($v$ in $\bigcap_{n \in U_{CON}} adj(m_{vgs}[qo_i[n]])$) do
3:      if (IsJoinable($qo_i[idx], v, m_{vgs}, vgs_i, cvw_i$)) then
4:          $m_{vgs}[qo_i[idx]] \leftarrow v$;
5:          if ($idx < |q_R|$) then
6:              RECEXTVERTEXMAPPING($idx + 1, m, q_{RBI}$,
                  $vgs_i, vgf_i$);
7:          else
8:              foreach ($qs \in vgs_i$) do
9:                  $m \leftarrow \emptyset$;
10:                 for ($1 \leq j \leq |V_R|$) do
11:                     $m[qs[j]] \leftarrow m_{vgs}[qo_i[k]]$ where the array
                          index of the $qo_i[k]$ is $j$;
12:                 end
13:                 NONREDVERTEXMATCHING($m$);
14:             end
15:         end
16:         $m_{vgs}[qo_i[idx]] \leftarrow$ INVALID;
17:     end
18: end
```

B. ADDITIONAL EXPERIMENTAL RESULTS

B.1 Varying the Number of Threads

The purpose of this experiment is to assess the CPU parallelization effect. We measure the elapsed times and speed-ups by varying the number of execution threads. In order to simulate hot-run, we preload the whole graph in memory so that the speed up due to CPU parallelism can be measured. Figure 16 plots the speed-ups of DUALSIM in LJ by varying the number of execution threads from 1 to 6. DUALSIM achieves almost linear speed-up. Specifically, the speed-up using 6 threads for q_1 is 5.46 while the speed-up for q_4 is 5.53. The experimental results substantiate that DUALSIM fully utilizes multi-core parallelism.

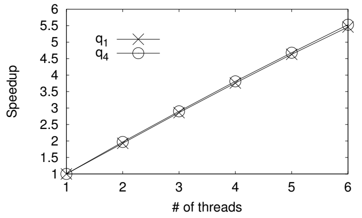


Figure 16: Varying the number of execution threads.

B.2 Comparison to OPT

We compare DUALSIM with OPT [17], a state-of-the-art disk-based triangulation. We use LJ, FR and YH datasets to this end. Figure 17 shows the results demonstrating superiority of DUALSIM for enumerating triangles. DUALSIM is a generalization of OPT but outperforms OPT since DUALSIM has a better buffer allocation strategy where most of buffer frames are allocated for internal triangulation and only two buffer frames * # of threads are allocated for external triangulation. By using this strategy, the number of iterations for the first level is much smaller than OPT, which is very effective when we use HDDs.

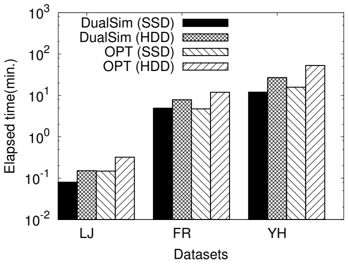
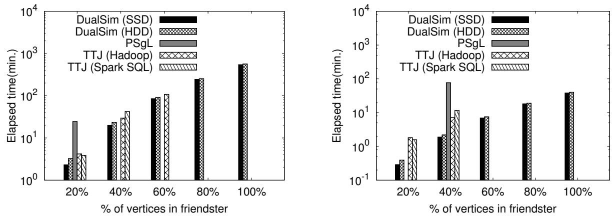


Figure 17: Comparison to OPT.

B.3 Varying Graph Size for Q2 and Q3

Figure 18 shows experimental results for $q2$ and $q3$ in a distributed enviroment. Regarding q_2 and q_3, DUALSIM outperforms TWINTWIGJOIN and PSGL by up to 5.27 and 35.04 times. TWINTWIGJOIN (Hadoop), TWINTWIGJOIN (Spark SQL), and PSGL fail to process q_2 when the input graph reaches 80%, 60%, and 40% of the original FR, respectively. Regarding q_3, they fail when the input graph reaches 60% of the original graph size. We also note that PSGL successfully processes q_3 for 40% of vertices but fails to process for 20% of vertices, since one slave machine has three times more intermediate results even if the dataset size is smaller, depending on partitioning results.



(a) Query q_2.

(b) Query q_3.

Figure 18: Varying graph size in a cluster.

B.4 Intermediate Result Size

Table 4 shows the *actual* number of intermediate results generated by TWINTWIGJOIN and PSGL for q_1 and q_4 across different datasets. As remarked earlier, the massive number of these results have significant adverse impact on the performances of these approaches.

Table 4: The actual number of intermediate results.

	TWINTWIGJOIN		PSGL	
	q_1	q_4	q_1	q_4
WG	17 728 487	120 080 516	14 267 638	80 638 724
WT	13 986 397	161 379 149	11 597 979	132 275 038
UP	24 233 735	30 466 127	11 290 035	10 776 978
LJ	330 307 042	20 761 697 361	290 581 577	19 871 938 971
OK	760 453 448	8 495 795 053	631 448 564	6 450 113 417
WP	135 418 732	42 568 284	26 134 800	25 236 087
FR	6 338 480 209	32 419 482 939	**fail**	**fail**
YH	**fail**	**fail**	**fail**	**fail**

Table 5 shows the *estimated* number of intermediate results using the estimation formulae in [20, 24]. We observe that there are significant estimation errors in their estimation. In [24], when a query vertex u is selected to match a data vertex v, it assumes that every data vertex in $adj(v)$ can be mapped to any non-matched query vertex in $adj(u)$. However, some vertices in $adj(v)$ could have been matched with query vertices already. Thus, this assumption leads to over-estimation. In [20], the estimation model assumes the Erdös-Rényi random graph model. However, this model does not capture characteristics of real-world graphs. Furthermore, neither formula takes into account of bloom filters or partial orders in their estimation. Even if we do not use bloom filters or partial orders, their estimation still leads to significant errors due to the aforementioned, unrealistic assumptions. Accurate estimation of intermediate result sizes would be an interesting future research topic.

Table 5: The estimated number of intermediate results.

	TWINTWIGJOIN		PSGL	
	q_1	q_4	q_1	q_4
WG	89 647 379	255 985 479	728 292 937	40 912 661 890
WT	40 923 518	108 792 088	8 300 949 511	137 340 284 542
UP	305 676 473	867 478 443	339 556 041	11 701 483 501
LJ	1 595 424 330	4 656 266 644	7 274 351 324	930 439 994 468
OK	17 995 274 430	53 668 127 117	45 628 539 012	6 177 760 914 437
WP	2 475 955 253	7 061 648 888	355 660 021 848	2 156 268 223 798
FR	200 675 735 441	596 618 192 868	720 720 573 330	137 456 527 555 023
YH	123 599 731 090	351 495 517 035	13 589 362 651 547	132 438 400 752 902

B.5 Preparation Cost

Table 6 shows elapsed time for the preparation step for varing queries. The preparation step takes at most 1 msec, which is negligible.

Table 6: Elapsed time of preparation step (in msec.).

query	q_1	q_2	q_3	q_4	q_5
time	0	0	0	1	1

C. PROOF TO LEMMA 1

PROOF. According to Property 1, $m[qs[1]] \prec m[qs[2]] \prec .. \prec m[qs[|V_R|]]$. Vertices are stored according to $id(\cdot)$. Thus, if $id(v_i) < id(v_j)$, $P(v_i) \leq P(v_j)$. Therefore, the condition, $P(m[qs[1]]) \leq P(m[qs[2]]) \leq \cdots \leq P(m[qs[|V_R|]])$, is satisfied. $\square$