I/O Cost Minimization: Reachability Queries Processing over Massive Graphs

Zhiwei Zhang
Chinese Univ. of Hong Kong
Hong Kong, China
zwzhang@cuhk.edu.hk

Jeffrey Xu Yu
Chinese Univ. of Hong Kong
Hong Kong, China
yu@cuhk.edu.hk

Lu Qin
Chinese Univ. of Hong Kong
Hong Kong, China
lqin@cuhk.edu.hk

Qing Zhu
Renmin University of China
Beijing, China
zq@ruc.edu.cn

Xiaofang Zhou
University of Queensland
Queensland, Australia
zxf@uq.edu.au

ABSTRACT

Given a directed graph G, a reachability query (u, v) asks whether there exists a path from a node u to a node v in G. The existing studies support reachability queries using indexing techniques, where both the graph and the index are required to reside in main memory. However, they cannot handle reachability queries on massive graphs, when the graph and the index cannot be entirely held in memory because of the high I/O cost. In this paper, we focus on how to minimize the I/O cost when answering reachability queries on massive graphs that cannot reside entirely in memory. First, we propose a new *Yes-Label* scheme, as a complement of the *No-Label* used in *GRAIL* [23], to reduce the number of intermediate results generated. Second, we show how to minimize the number of I/Os using a heap-on-disk data structure when traversing a graph. We also propose new methods to partition the heap-on-disk, in order to ensure that only sequential I/Os are performed. Third, we analyze our approaches and show how to extend our approaches to answer multiple reachability queries effectively. Finally, we conducted extensive performance studies on both large synthetic and large real graphs, and confirm the efficiency of our approaches.

1. INTRODUCTION

Many emerging real applications deal with a large graph due to the expressive power of a graph to handle complex relationships among objects. Instances include social network analysis, biological network analysis, navigation behavior analysis, and web site analysis. The sizes of such graph structured data are rapidly increasing, and become so large that cannot reside in main memory.

Among all the graph queries, as a fundamental type of queries, a graph reachability query answers whether a node is reachable from another node in a large directed graph, and has being extensively studied [1, 11, 8, 14, 15, 16, 10, 6, 22, 18, 5, 7, 3, 13, 12, 23, 20]. Consider a social network that nodes represent people and edges represent relations between people. There are needs to understand whether two people are related for security reasons [2]. On biological networks, where nodes are either molecules, or reactions, or physical interactions of living cells, and edges are interactions among them, there is an important question to "find all genes whose expressions are directly or indirectly influenced by a given molecule" [19]. All those questions can be answered based on reachability queries.

Reachability Queries: Let $G = (V, E)$ be a large directed graph that has n nodes and m edges. A *reachability query*, denoted (u, v), asks whether there exists a path from node u to node v in G. For simplicity, we use $u \rightsquigarrow v$ to denote yes, and $u \not\rightsquigarrow v$ to denote no. A reachability query over a directed graph G can be answered over a corresponding directed acyclic graph (DAG) of the graph G by condensing strongly connected components of G into nodes. Two nodes, u and v, co-exist in a strongly connected component, if and only if both $u \rightsquigarrow v$ and $v \rightsquigarrow u$ are true. In other words, in a strongly connected component, for every two nodes, u and v, $u \rightsquigarrow v$ and $v \rightsquigarrow u$. Given a directed graph $G(V, E)$, its strongly connected components, $C_1, C_2, \cdots$, can be efficiently identified in $O(n+m)$ time [9]. With the strongly connected components identified, a DAG of the graph G, denoted G', can be constructed as follows. First, a strongly connected component C_i in G is replaced by a representative node v in G'. Second, all the edges between the nodes in the strongly connected component C_i are removed while all incoming edges and outgoing edges of C_i will be represented as incoming edges and outgoing edges of the representative node v in G'. Upon the DAG G', a reachability query (u, v) over G can be processed over G' by checking whether the corresponding strongly connected component, where v resides, is reachable from the corresponding strongly connected component, where u resides. In the following, without otherwise specified, we assume G is a DAG.

Table 1 shows a summary on the time/space complexity of different approaches for reachability queries processing. There are two extreme approaches to process a reachability query, (u, v), in a graph G. It can be processed as to traverse from u to v using breadth/depth-first search (*BFS*/*DFS*) over the graph G on demand, when a reachability query is issued. It incurs high query processing cost in $O(n + m)$ time, without any preprocessing cost. On the other hand, it can be processed as to check whether (u, v) exists in the edge transitive closure (TC) of a graph G in $O(1)$ time by first precomputing the edge transitive closure TC on disk. Such precomputing is also called as index construction. *TC* results in high

	Query Time	*Index Construction*	*Index size*
BFS/DFS	$O(n+m)$	-	$O(1)$
TC [17, 20]	$O(1)$	$O(nm)$	$O(n^2)$
Tree+SSPI [4]	$O(m-n)$	$O(n+m)$	$O(n+m)$
GRIPP [18]	$O(m-n)$	$O(n+m)$	$O(n+m)$
Dual-Labeling [22]	$O(1)$	$O(n+m+t^3)$	$O(n+t^2)$
Chain Cover [5]	$O(\log k)$	$O(n^2+kn\sqrt{k})$	$O(nk)$
Tree Cover [1]	$O(\log n)$	$O(nm)$	$O(n^2)$
Path-Tree Cover [13]	$O(\log^2 k')$	$O(mk')$ or $O(nm)$	$O(nk')$
2-Hop Cover [8]	$O(m^{1/2})$	$O(n^3 \cdot \lvert TC \rvert)$	$O(nm^{1/2})$
3-Hop Cover [12]	$O(\log n + k)$	$O(kn^2 \cdot \lvert Con(G) \rvert)$	$O(nk)$
GRAIL [23]	$O(d)/O(n+m)$	$O(d(n+m))$	$O(dn)$

Table 1: Time/Space Complexity (n and m are # of nodes and edges, k is # of paths/chains, $t = m - n$ (non-tree edges), and d is # of codes.)

computational cost to be constructed in $O(nm)$ time, and high storage consumption using $O(n^2)$ space. All the existing work attempt to reach a reasonable query time by minimizing the index construction time and the space consumed, and have the difficulties to deal with a massive graph with one exemption [23]. In [23], Yildirim et al. propose an approach called *GRAIL* (Graph Reachability indexing via rAndomized Interval Labeling). In brief, for a given graph G, it randomly generates d interval codes using *DFS*. During the i-th *DFS* over G, it generates a code u, denoted as L_u^i, over G, in linear time and space. Let $L_u = (L_u^1, L_u^2, \cdots, L_u^d)$ be a list of d interval codes, for node u. It ensures that if there is at least one $L_v^i \not\subseteq L_u^i$, for $1 \leq i \leq d$, then $u \not\rightsquigarrow v$ over G. However, such an approach does not ensure that if every $L_v^i \subseteq L_u^i$ for $1 \leq i \leq d$, then $u \rightsquigarrow v$. In the worst scenario, it needs to answer a reachability query, (u, v), using *DFS* to traverse G assuming all the d-codes and the graph reside in main memory.

The main issue we study in this paper is I/O cost minimization in processing reachability queries, because all the existing work keep the entire index and the graph in main memory, and do not consider I/O cost. The I/O cost occurring in the existing approaches can be extremely high when the entire index and graph cannot be kept in main memory, since reachability queries processing requires random I/Os and it is very difficult to predict what disk pages will be needed in the near future. The baseline of our work is *GRAIL* [23] since it is the currently known only approach that can handle a massive graph in terms of the index construction time/space. We show the importance of the memory allocation regarding the number of I/Os in Fig. 1. Fig. 1 shows the total number of I/Os needed for processing 20,000 reachability queries over a graph G, based on *GRAIL* [23], when only a percentage of the entire index constructed for G and the graph itself can be held in main memory. The generated graph G has 10,000,000 nodes, 50,000,000 edges, and an average degree 5, which results in 11,939 of 64KB-sized pages. The *GRAIL* index built is 11,940 64KB-sized pages with the default dimension of index 5 as used in [23]. With a LRU buffer replacement strategy, the total number of I/Os is 38,808 64K-sized pages when 80% of the needed memory is available, and becomes 322,811 when only 20% of the needed memory is available. I/O cost is a dominant factor that affects the query processing time. In this paper, we propose a new approach to process a batch of reachability queries together, as shown in Fig. 1, the total number of I/Os by our approach is 24,418, which is the additional 539 disk page accesses plus the number of disk page accesses for scanning the entire index and graph once (23,879). In addition, all the I/Os using *GRAIL* are random disk accesses. Our approach is a random I/O free approach. As an indicator, based on the SATA Hard Disk Drive we used to test, with 64KB sized disk pages, the bandwidth (MB/sec) of sequential I/O is 4 times of that of random I/O; with 8KB sized disk pages, the bandwidth (MB/sec) of sequential I/O is 19 times of that of random I/O.[1]

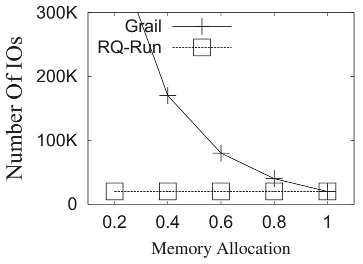


Figure 1: The Number of I/Os

The main contributions of this work are summarized below. First, we introduce a new labeling scheme called *Yes-Label*, as a complement of the *No-Label* used in *GRAIL*, to reduce the number of intermediate results generated when traversing the graphs. Second, we propose an algorithm to generate *Yes-Label* using linear time and space, and we show how to combine the *Yes-Label* with *No-Label* to answer reachability queries in memory efficiently. Third, we study how to minimize the number of I/Os when answering a reachability query given limited memory using a new *YNG-Index*. We discuss how to make all I/Os sequential by using a partition based heap-on-disk. We analyze the number of I/Os and show how to extend our method to answer multiple reachability queries. Finally, we conducted extensive performance studies on both large synthetic and large real graphs, to confirm the efficiency of our approaches.

The remainder of the paper is organized as follows. We discuss the related work in Section 2. In Section 3, we introduce *GRAIL* as the up-to-date approach to deal with a massive graph in main memory, and discuss the I/O issues when main memory is limited. In Section 4, we introduce a new labeling called an *Yes-Label*, and show that we can generate such *Yes-Label* in linear time and space. Together with *No-Label* used in *GRAIL*, we can further reduce *BFS/DFS* cost. We focus on I/O minimization in Section 5. We show our experimental studies in Section 6, and conclude our paper in Section 7.

2. RELATED WORK

All the existing work focus on time/space complexity on index construction, and in particular time complexity for querying provided all the index and graph can reside in memory. Table 1 shows a summary on the time/space complexity of different approaches, for a graph $G(V, E)$ with $n = |V|$ and $m = |E|$.

Simon proposes an algorithm to compute *TC* for a DAG, G, in $O(nm)$ time [17], with $O(n^2)$ space in the worst case. With the edge *TC* constructed, the query time is constant $O(1)$. Schaik and Moor [20] study a bit-vector compression approach, the time/space complexity are the same to [17], computing *TC* is needed.

In [4], Chen et al. propose an index by utilizing a spanning tree of the graph G. It takes $O(n + m)$ time to construct an index in

[1]The testing is measured using `http://freshmeat.net/projects/fio/`

$O(n+m)$ size. Given two nodes u and v in G, it can answer (u, v) in $O(1)$ time if there is a path from u to v in the spanning tree. However, because the index is generated based on the connections over the spanning tree of the graph G, it cannot say $u \not\rightsquigarrow v$ if there does not exist a corresponding index entry. To handle the edges that do not appear in the spanning tree, Chen et al. use an additional data structure called SSPI (Surrogate&Surplus Predecessor Index) to answer a reachability query in run time, which takes $O(m-n)$ time in the worst case. Like [4], in [18], Trißl and Leser build an index, called GRIPP (GRaph Indexing based on Pre- and Postorder numbering), using a spanning tree of the graph G. Trißl and Leser discuss traversal strategies using the proposed GRIPP. The time and space complexities are the same to Tree+SSPI.

Wang et al. propose a dual-labeling approach in [22] for sparse graphs based on the observation that the majority of large graphs in real applications are sparse. It implies that the number of non-tree edges in the graph G, that do not appear in a spanning tree of G, is small. Let $t = m - n$ be the number of such non-tree edges. Wang et al. use a tree coding scheme for tree edges and a graph coding scheme for non-tree edges for sparse graphs where $t \ll n$. It handles the edge transitive closure over non-tree edges. The dual-labeling approach achieves $O(1)$ query time with an index of size $O(n + t^2)$ that is constructed in $O(n + m + t^3)$ time.

Jagadish in [11] proposes a chain cover approach. The chain cover is to decompose a graph G into pairwise disjoint chains, where a chain based is more general than a path based. Jagadish proposes an algorithm in $O(n^3)$ to find the minimal number of chains, in G. The number of chains for G is called the width of G, denoted by k. Based on the chain cover, an index in $O(nk)$ size can be constructed. The query time is $O(\log k)$. In [5], Chen and Chen propose a new approach that can further reduce the time complexity of constructing the index based on the chain over to $O(n^2+kn\sqrt{k})$.

Agrawal et al. in [1] study a tree cover approach to assign labels to nodes in a DAG. In brief, if a node u can reach a node v, then u can reach any nodes in the subtree rooted at v. Agrawal et al. propose an optimal tree cover that maximally compresses the edge transitive closure. The index size is $O(n^2)$ in the worst case, but in practice, it can compress edge transitive closure which results in an even better compression rate than a chain cover [11, 5]. The time complexity for index construction is $O(nm)$. It can construct an index for a large graph efficiently. The query time is $O(\log n)$.

Jin et al. propose path-tree cover in [13] along the line of tree cover [1]. Jin et al. decompose G into pairwise disjoint paths and build a tree over the paths by treading a decomposed path as a node in the tree. Let k' be the number of pairwise disjoint paths in G. Two algorithms are proposed, namely, PTree-1 and PTree-2. Both construct an index in $O(nk')$ space. PTree-1 constructs the index in $O(nm)$ time, whereas PTree-2 constructs it in $O(mk')$ time. The query time is in $O(\log^2 k')$.

Cohen et al. in [8] propose an index called 2-hop cover. A node, u, in a graph G is assigned two sets of nodes, as its label, called $L_{in}(u)$ and $L_{out}(u)$. $L_{in}(u)$ contains a set of nodes that can reach u and $L_{out}(u)$ contains a set of nodes that u can reach. The labels assigned to nodes are done in a way to ensure $u \rightsquigarrow v$ to be true if and only if $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. It turns out to be a set cover problem. Cohen et al. propose an approximate algorithm to construct an index in $O(nm^{1/2})$ space. The time complexity for constructing such an index remains open. In [13], the conjecture is $O(n^3 \cdot |TC|)$ where $|TC|$ is the size of the edge transitive closure of G. Several efficient algorithms are proposed to compute 2-hop cover [15, 6, 7]. The 2-hop cover maintenance is studied in [16, 3]. Jin et al. in [12] further study a new approach, called 3-hop, that combines chain cover and 2-hop cover. The index construction time is $O(kn^2 \cdot |Con(G)|)$. Here k is the number of pairwise disjoint paths in G, and $Con(G)$ is transitive closure contour of G defined in [12]. Yildirim et al. in [23] propose *GRAIL* to deal with a massive graph, which we will discuss in detail in next section.

Algorithm 1 *No-Label* (G, d)

```
 1: Let $R$ be a set of roots in $G$;
 2: for each $i$ from 1 to $d$ do
 3:     $p \leftarrow 1$;
 4:     DFS $(i, r, G)$ for every $r \in R$ in random order;

Procedure DFS $(i, u, G)$
 5: if $u$ has been visited then
 6:     return
 7: for each $v$ of $(u, v) \in E(G)$ in random order do
 8:     DFS $(i, v, G)$;
 9: let $q$ be the smallest post-order in any child of $u$;
10: $L_u^i[u_q, u_p] \leftarrow [\min\{q, p\}, p]$;
11: $p \leftarrow p + 1$;
```

None of the existing approaches consider I/O cost as a dominant factor in their approaches. Vitter surveyed the state of the art of the algorithms and data structures for external memory in [21].

3. GRAIL: A NO-LABELING APPROACH

We introduce *GRAIL* [23] in this section, which serves the baseline of the work presented in this paper. Given a directed acyclic graph $G(V, E)$, where V is a set of nodes and E is a set of edges. We use $V(G)$ and $E(G)$ to denote all nodes and all edges in G, respectively. *GRAIL* randomly generates d codes by traversing G for d times using *DFS*. Let $L_u^i = [u_q, u_p]$ be an interval code assigned to u in the i-th *DFS* traversal, $L_u^i = [u_q, u_p]$ is assigned as follows: u_p is the post-order of the node, and u_q is the smallest post-order of a node that u can reach in the i-th *DFS* traversal. The *No-Label* algorithm is shown in Algorithm 1 which generates d interval codes for every node u in G. Here, for a node u, the range of its code covers the maximum range of all its reachable nodes. The smallest post-order corresponds to the first node that u can reach during the depth first search. The code L_u^i can possibly cover many nodes which u cannot reach.

Given the codes assigned to two nodes u and v, the containment, $L_u^i \subseteq L_v^i$, is defined as $v_q < u_q$ and $v_p \geq u_p$. And given $L_u = (L_u^1, L_u^2, \cdots, L_u^d)$ and $L_v = (L_v^1, L_v^2, \cdots, L_v^d)$, it is defined as $L_v \subseteq L_u$ if $L_v^i \subseteq L_u^i$ for every $1 \leq i \leq d$, and $L_v \not\subseteq L_u$ if there is at least $L_v^i \not\subseteq L_u^i$ for $1 \leq i \leq d$. In a theorem given in [23], if $L_v \not\subseteq L_u$ then $u \not\rightsquigarrow v$. We call it a *No-Label* approach, because in *GRAIL* $L_v \subseteq L_u$ does not necessarily mean $u \rightsquigarrow v$. In the worst case, *GRAIL* needs to answer a reachability query (u, v) using *DFS* to traverse the whole graph.

Example 3.1: A DAG G is shown in Fig. 3(a), and its *No-Label*s are shown in Table 2. Assume that we only use L_u^1 (the third column) as the *No-Label*s in this example. Consider a reachability query (c, g), where the *No-Label*s for c and g are $L_c^1 = [1, 2]$ and $L_g^1 = [8, 8]$. The answer is $c \not\rightsquigarrow g$, because $L_g^1 \not\subseteq L_c^1$. Consider another reachability query (l, f), where the *No-Label*s for l and f are $L_l^1 = [1, 12]$ and $L_f^1 = [3, 9]$. The answer is $l \not\rightsquigarrow f$, even though

Algorithm 2 *GRAIL* $(G, (u, v))$

```
1: if $L_v \not\subseteq L_u$ then
2:     return false; $\{u \not\rightsquigarrow v\}$
3: if $u = v$ then
4:     return true; $\{u \rightsquigarrow v\}$
5: for every $w$ of $(u, w) \in E(G)$ do
6:     if $L_v \subseteq L_w$ then
7:         if GRAIL $(G, (w, v))$ = true then
8:             return true;
9: return false; $\{u \not\rightsquigarrow v\}$
```

$L_f^1 \subseteq L_l^1$. In other words, given the existence of the *No-Label*s (the third column in Table 2 only), if $L_g^1 \not\subseteq L_c^1$, it can conclude that $c \not\rightsquigarrow g$ immediately without further checking; but if $L_f^1 \subseteq L_l^1$, it cannot make any conclusion immediately whether f is reachable from l or not, and needs to traverse the graph to check. □

Remark 3.1: *Given a specific DFS, the interval of a* No-Label L_u^i *for a node u is maximum to ensure $u \not\rightsquigarrow v$ for another node v if $L_v^i \not\subseteq L_u^i$, and is minimum to verify $u \rightsquigarrow v$ using the DFS if $L_v^i \subseteq L_u^i$ otherwise.* □

In Remark 3.1, by maximum, we mean that given a specific *DFS*, the interval represented by the *No-Label* of a node u covers the maximum number of intervals for all nodes it can reach. The reason is that for a node v that u can reach, the post-order of v must be in the *No-Label* of u, because the *No-Label* of u is in the range from the smallest post-order among its descendants to its own post order. By minimum, we mean that, if a node u can reach a node v, then the *No-Label* for u must contain the *No-Label* of v. This is because the boundary of the *No-Label* is taken from the post-order of reachable nodes. None post-order of the unreachable nodes are set as the boundary.

The *GRAIL* algorithm is shown in Algorithm 2. In Algorithm 2, line 5-8, deals with the depth-first search (*DFS*). In the original *GRAIL* algorithm in [23], it has a simple mechanism to reduce such *DFS* with some additional data structure. We omit that part, because it is shown not effective in the performance studies in [23]. The *GRAIL* algorithm performs very well when the graph G and the entire index (every L_u for $u \in G$) are held in main memory. However, as given in Fig. 1, the I/O cost can be very high when it cannot be held in main memory.

4. A NEW YES-LABELING APPROACH

As shown in Algorithm 2, the only way that returns true without recursion is $u = v$ which means the whole path has been searched. Consider a worst case of using *GRAIL* to answer (w_0, w_n) when w_n is reachable from w_0 in G over a long path $(w_0, w_1)(w_1, w_2)\cdots(w_{n-1}, w_n)$. By *GRAIL*, it needs to scan the path using *DFS* because $L_{w_i} \subseteq L_{w_n}$ for $0 \le i \le n$. It can possibly end up a large number of random I/Os.

Fig. 2 shows the distribution on traversal depth for yes/no-queries. By a yes query, we mean the answer of a reachability query (u, v) is positive, and by a no query, we mean the answer of a reachability query (u, v) is negative. We use a generated graph with $n = 100,000$ nodes and $m = 300,000$ edges. We use *GRAIL* to answer the reachability of every pair of nodes in the graph. For each pair, we record the maximum depth traversed using *GRAIL*. We divide the $n \times n$ pairs into two parts. One part includes no-

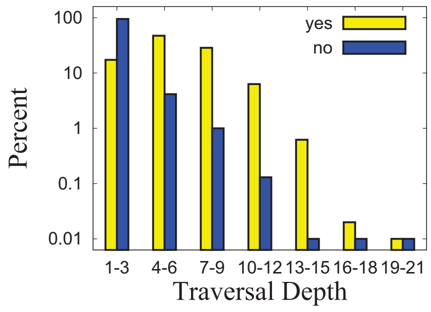


Figure 2: Distribution on max depth traversed for yes/no-queries

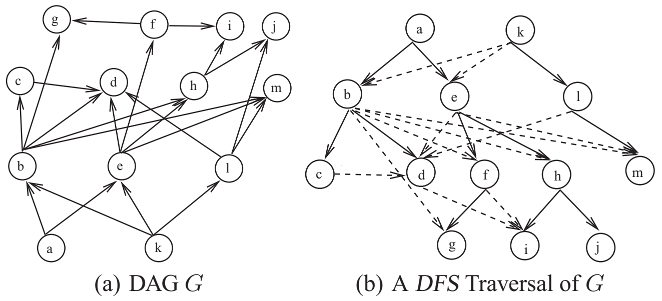


Figure 3: An Example

queries, and the other part includes yes-queries. For each part, we calculate the distribution of queries on the max depth traversed. As shown in Fig. 2, nearly 95% of no-queries stop at depth 1-3, while 17% of yes-queries stop at depth 1-3. There are 47% of yes-queries stop at depth 4-6 and 29% of yes-queries stop at depth 7-9. When expanded to depth 9, all nodes with depth no larger than 9 will probably be visited. It means that *GRAIL* can handle yes-queries properly but cannot answer no-queries efficiently.

In order to resolve the problem of *DFS* when answering yes-queries ($u \rightsquigarrow v$), we design a new label, called *Yes-Label*. In a similar fashion, given a DAG $G(V, E)$, we randomly generate t interval codes for every node in G in linear time and space. Let $\mathcal{L}_u^i = [u_s, u_t]$ and $\mathcal{L}_v^i = [v_s, v_t]$ be the i-th interval code generated. We define $\mathcal{L}_v^i \subseteq \mathcal{L}_u^i$ as $v_s > u_s$ and $v_t \le u_t$. We will show that $u \rightsquigarrow v$ if $\mathcal{L}_v^i \subseteq \mathcal{L}_u^i$. Let $\mathcal{L}_u = (\mathcal{L}_u^1, \mathcal{L}_u^2, \cdots, \mathcal{L}_u^t)$ and $\mathcal{L}_v = (\mathcal{L}_v^1, \mathcal{L}_v^2, \cdots, \mathcal{L}_v^t)$. We define $\mathcal{L}_v \subseteq \mathcal{L}_u$ if there is at least $\mathcal{L}_v^i \subseteq \mathcal{L}_u^i$ for $1 \le i \le t$, and $\mathcal{L}_v \not\subseteq \mathcal{L}_u$ if $\mathcal{L}_v^i \not\subseteq \mathcal{L}_u^i$ for every $1 \le i \le t$. We call it *Yes-Label*, because we ensure $u \rightsquigarrow v$ if $\mathcal{L}_v \subseteq \mathcal{L}_u$. On the other hand, $\mathcal{L}_v \not\subseteq \mathcal{L}_u$ does not necessarily mean $u \not\rightsquigarrow v$. In the worst case, we need to answer a reachability query (u, v) using *BFS*. We will explain the choice of *BFS* instead of *DFS* later when we discuss sequential I/Os.

In order to construct a *Yes-Label* $\mathcal{L}_u^i = [u_s, u_t]$, we adapt a simple tree cover approach, based on the i-th *DFS* traversal of a graph, which works as follows: u_s is the pre-order of the node during *DFS*, and u_t is the largest pre-order of a node that u can reach in the i-th *DFS* traversal. Since the *Yes-Label*s generated over a tree can only be used to answer a small subset of yes-queries over the original graph, in our algorithm, we try to explore possible expanding the interval code for every node in a certain *DFS* traversal, in order for the *Yes-Label* to answer more yes-queries. In short, we allow overlapping between *Yes-Label*s.

Node	$\mathcal{L}^1_u$	L^1_u	L^2_u	Children
a	[1,20]	[1,11]	[1,13]	be
k	[2,26]	[1,13]	[1,12]	bel
b	[2,18]	[1,7]	[1,11]	cdhmg
e	[5,24]	[1,10]	[1,9]	dhmf
l	[22,25]	[1,12]	[1,4]	dmj
c	[3,4]	[1,2]	[3,10]	d
d	[5,6]	[1,1]	[3,3]	∅
h	[13,18]	[3,5]	[2,6]	ij
m	[23,24]	[6,6]	[1,1]	∅
f	[9,15]	[3,9]	[5,8]	gi
g	[10,11]	[8,8]	[7,7]	∅
i	[14,15]	[3,3]	[5,5]	∅
j	[16,17]	[4,4]	[2,2]	∅

Table 2: *Yes-Label*/*No-Label* for G in Fig. 3(a)

The *Yes-Label* algorithm is shown in Algorithm 3. We explain it using an example. A DAG G is shown in Fig. 3(a). In Algorithm 3, with the help of a virtual root $\top$, it assigns a node u using the similar approach used in [18]. In the *Initial* phase (line 6), it assigns the initial *Yes-Label* to nodes as follows: $\mathcal{L}^1_a = [1, 20]$, $\mathcal{L}^1_b = [2, 7]$, $\mathcal{L}^1_c = [3, 4]$, $\mathcal{L}^1_d = [5, 6]$, $\mathcal{L}^1_e = [8, 19]$, $\mathcal{L}^1_f = [9, 12]$, $\mathcal{L}^1_g = [10, 11]$, $\mathcal{L}^1_h = [13, 18]$, $\mathcal{L}^1_i = [14, 15]$, $\mathcal{L}^1_j = [16, 17]$, $\mathcal{L}^1_k = [21, 26]$, $\mathcal{L}^1_l = [22, 25]$, and $\mathcal{L}^1_m = [23, 24]$, The order of traversing is "abccddbefggfhiijjheaklmmlk". Note that we can do *DFS* from the roots in a random order. By *DFS* traversing the solid edges following a topological order. The reachability queries along the solid edges can be answered using the initial *Yes-Label*s, but the reachability queries involved some dashed edges cannot. The next step is to expand the initial *Yes-Label*s to answer more reachability queries involving dashed edges. The *Yes-Label* allows overlapping between intervals on trees. The main idea for the *Yes-Label* is based on the following intuition. In a directed acyclic graph, if the child nodes of two adjacent sibling nodes are also adjacent to each other, then the intervals for the two sibling nodes can be combined. In the example, for the two sibling nodes "e" and "b", their child nodes, namely, "c", "d", "f", and "h" are also adjacent. In this situation, It is possible to extend the interval of "e" and "b". In the *Yes-Expand* phase (line 8), it expands some intervals generated in the *Initial* phase to cover more *Yes-Label*. Consider node "e", assume its initial $\mathcal{L}^1_e = [8, 19]$. As shown in Fig. 3(b), "e" has an immediate left sibling "b" which has two children "c" and "d" linked by the solid edges following the topological order in the *Initial* phase. Because "e" links "d" in the DAG G (Fig. 3(b)), $\mathcal{L}^1_e = [8, 19]$ is expanded to $\mathcal{L}^1_e = [5, 19]$ to contain $\mathcal{L}^1_d = [5, 6]$. This expansion is possible, because there does not exist any *Yes-Label* in the expanded interval $[5, 19]$ that "e" cannot reach. In a similar fashion, "e" has an immediate right sibling "l" which has one child "m". Because "e" links "m" in the DAG G (Fig. 3(b)), $\mathcal{L}^1_e = [5, 19]$ is again expanded to $\mathcal{L}^1_e = [5, 24]$ to contain $\mathcal{L}^1_m = [23, 24]$. Furthermore, $\mathcal{L}^1_k = [21, 25]$ is expanded to $\mathcal{L}^1_k = [2, 25]$ for the same reason. Comparing the initial interval codes, the expansion covers additional 19 pairs which can be answered immediately using *Yes-Label*.

As discussed above, Algorithm 3 generates *Yes-Label* even if it does not call *Yes-Expand*. The expansion of the interval code of node u to some children of its sibling node w will cover all the nodes in the subtrees rooted at those children. Consider a special case, when the tree cover (the tree traversed in a certain *DFS*) of a graph is a complete binary tree of height h. Suppose on average, the *Yes-Label* for each node is expanded by 1 node. Then, in the level $h - 1$, the *Yes-Label* in each node can include one more reachable pair. In the level $h - 2$, the *Yes-Label* in each node can include two more reachable pairs, and in the level $h - k$, the *Yes-Label* in each node can include 2^k more reachable pairs. Fig. 4 shows the number of *Yes-Label*s that take effect when answering a set of 100,000 yes-queries. By taking effect, we mean a yes-query (u, v) is answered effectively using a *Yes-Label* before it traverses to the node v. The number of nodes in the generated graphs vary from 5,000,000 to 30,000,000, and the average degree for nodes in each graph is five. As shown in Fig. 4, using tree labels by *Initial* without *Yes-Expand*, only 33% of the yes-queries are answered using the *Yes-Label* on average, and by the additional *Yes-Expand*, more than 60% of the yes-queries can be answered using the *Yes-Label* on average.

Algorithm 3 *Yes-Label* (G, t)

```
 1: Let $R$ be a set of roots in $G$;
 2: for each $i$ in 1 to $t$ do
 3:     create a virtual root, $\top$, that links to every node in $R$;
 4:     sort $G$ in topological order starting from $\top$;
 5:     $p \leftarrow 0$; {$p$ is a global variable.}
 6:     Initial $(i, r, G)$ for every $r \in R$ in topological order;
 7:     $\mathcal{L}^i_\top = [0, p]$;
 8:     Yes-Expand $(i, \top, G)$;

Procedure Initial $(i, v, G)$
 9: if $v$ has been visited then
10:     return
11: $q \leftarrow p$; $p \leftarrow p + 1$;
12: for each $v$ of $(u, v) \in E(G)$ in topological order do
13:     Initial $(i, v, G)$;
14: $\mathcal{L}^i_u[u_s, u_t] \leftarrow [q, p]$;
15: $p \leftarrow p + 1$;

Procedure Yes-Expand $(i, u, G)$
16: if $u$ has been visited then
17:     return
18: for each $v$ of $(u, v) \in E(G)$ in topological order do
19:     Yes-Expand $(i, v, G)$;
20: assume $u$ has $(v_1, v_2, \cdots)$ children in order;
21: for each child $v_j$ do
22:     suppose $v_{j-1}$ has $(w_1, w_2, \cdots, w_n)$ children in order;
23:     suppose $v_{j+1}$ has $(w'_1, w'_2, \cdots, w'_m)$ children in order;
24:     if there exists $w_k$ such that $(v_j, w_l) \in E(G)$ for $k \le l \le n$ then
25:         set $v_{j_s}$ in $\mathcal{L}^i_{v_j} = [v_{j_s}, v_{j_t}]$ to be $w_{k_s}$ in $\mathcal{L}^i_{w_k} = [w_{k_s}, w_{k_t}]$;
26:     if there exists $w'_k$ such that $(v_j, w'_l) \in E(G)$ for $1 \le l \le k$ then
27:         set $v_{j_t}$ in $\mathcal{L}^i_{v_j} = [v_{j_s}, v_{j_t}]$ to be $w'_{k_t}$ in $\mathcal{L}^i_{w'_k} = [w'_{k_s}, w'_{k_t}]$;
```

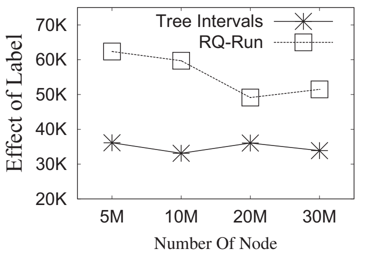


Figure 4: The effectiveness on label expansion

We discuss some properties of *Yes-Label*. Like *No-Label* used in *GRAIL*, the *Yes-Label* we use for a node is a single interval code. Unlike *No-Label*, overlapping is allowed between *Yes-Label*s. More precisely, in *No-Label*, there are two cases: $L^i_v \subseteq L^i_u$ and $L^i_v \not\subseteq L^i_u$, and when $L^i_v \not\subseteq L^i_u$, $L^i_v \cap L^i_u = \emptyset$. In *Yes-Label*, when $\mathcal{L}^i_v \not\subseteq \mathcal{L}^i_u$, $\mathcal{L}^i_v \cap \mathcal{L}^i_u$ is possibly non-empty.

Theorem 4.1: *$u \leadsto v$ if $\mathcal{L}_v \subseteq \mathcal{L}_u$.*

Proof Sketch: The *Yes-Label*s generated in the *Initial* phase are correct following a specific *DFS* in the topological order. In other words, in *DFS*, a node will visit its children in the topological order. The *DFS* results a tree. We prove it level by level from the bottom to the top. Here, we assume that at the k-th level, $u \leadsto v$ if $\mathcal{L}_v \subseteq \mathcal{L}_u$, on the condition that u appears at the k-th level, and v appears in any level below the k-th level. It is easy to shown that it is true when k is the height of the tree minus 1. We prove it for the (k-1)-th level (one level up). Assume u is at the (k-1)-th level, and v is in any level below the (k-1)-th level. Let $\mathcal{L}_u$ (= $[\mathcal{L}_{us}, \mathcal{L}_{ut}]$) and $\mathcal{L}_v$ (= $[\mathcal{L}_{vs}, \mathcal{L}_{vt}]$) denote the *Yes-Label* of u and v. There are three cases: u expands its *Yes-Label* to left, to right, and both. Below, we discuss the case when u expands its *Yes-Label* to left. The other two cases can be proved in a similar fashion. Assume u expands its *Yes-Label* according to the immediate left sibling u_0 where u_0 has children $(w_1, w_2, \cdots, w_n)$. For simplicity, suppose the *Yes-Label* of u is expanded to contain the *Yes-Label* of w_n, and assume w_n's *Yes-Label* is $\mathcal{L}_{w_n}$ (= $[\mathcal{L}_{w_n s}, \mathcal{L}_{w_n t}]$). We have a new *Yes-Label* for u which is $\mathcal{L'}_u$ (= $[\mathcal{L}_{w_n s}, \mathcal{L}_{ut}]$). Then, consider a node v below the (k-1)-th level. There are three cases. First, suppose $w_n \leadsto v$ since $\mathcal{L}_v \subseteq \mathcal{L}_{w_n}$ as proved at the k level. When u expands its *Yes-Label* to contain w_n due to the fact that there exists an edge in the graph from u to w_n, obviously $u \leadsto v$ and $\mathcal{L}_v \subseteq \mathcal{L'}_u$. Second, suppose one of u's child, ω, can reach v ($\omega \leadsto v$) using the *Yes-Label*s as proved at the k-th level. Obviously, we have $u \leadsto v$ and $\mathcal{L}_v \subseteq \mathcal{L'}_u$ at the (k-1)-th level. Third, consider (u, v) on the condition such that $w_n \not\leadsto v$ and $\omega \not\leadsto v$ at the k-th level. Here, the case is, by expanding the *Yes-Label* of $\mathcal{L}_u$ to $[\mathcal{L}_{w_n s}, \mathcal{L}_{ut}]$, whether we may wrongly answer yes, because $\mathcal{L}_v \subseteq \mathcal{L}_u$, even though $u \not\leadsto v$. The third case is impossible, because this implies that $\mathcal{L}_{vs}$ appears in $[\mathcal{L}_{w_n s}, \mathcal{L}_{w_n t}]$, and $\mathcal{L}_{vt}$ appears in $[\mathcal{L}_{us}, \mathcal{L}_{ut}]$. Due to the interval of v, its pre-order must be set before u_0 and its post-order be set after u. In the specific *DFS*, it cannot end up $\mathcal{L}_v \subseteq \mathcal{L}_u$ in the third case. □

Example 4.2: A DAG G is shown in Fig. 3(a), and one *Yes-Label* is shown in Table 2. Consider $\mathcal{L}^1_a = [1, 20]$ and $\mathcal{L}^1_c = [3, 4]$. Since $\mathcal{L}^1_c \subseteq \mathcal{L}^1_a$, $a \leadsto c$. Consider $\mathcal{L}^1_c = [3, 4]$ and $\mathcal{L}^1_d = [5, 6]$. Even though $\mathcal{L}^1_d \not\subseteq \mathcal{L}^1_c$, $c \leadsto d$. □

Remark 4.2: *The Yes-Label L^i_u for node u is the maximum to ensure $u \leadsto v$, if $L^i_v \subseteq L^i_u$, for another node v, and is minimum to verify $u \not\leadsto v$ using BFS/DFS.* □

Lemma 4.1: *Given No-Label and Yes-Label, and let the codes for u and v be $(L_u, \mathcal{L}_u)$ and $(L_v, \mathcal{L}_v)$. $L_v \not\subseteq L_u$ and $\mathcal{L}_v \subseteq \mathcal{L}_u$ do not co-occur.* □

Proof Sketch: Suppose $L_v \not\subseteq L_u$ and $\mathcal{L}_v \subseteq \mathcal{L}_v$ co-occur. As discussed above, $L_v \not\subseteq L_u$ implies that $u \not\leadsto v$, and $\mathcal{L}_v \subseteq \mathcal{L}_u$ implies that $u \leadsto v$. Since $u \not\leadsto v$ and $u \leadsto v$ contradict with each other, we conclude that $L_v \not\subseteq L_u$ and $\mathcal{L}_v \subseteq \mathcal{L}_u$ do not co-occur. □

The combination of *Yes-Label* and *No-Label* can effectively reduce the number of *BFS*/*DFS*, because the *Yes-Label* $\mathcal{L}_v \subseteq \mathcal{L}_u$ reduces those that need to check $L_v \subseteq L_u$.

The *Yes-GRAIL* algorithm is given in Algorithm 4 which uses both *Yes-Label* and *No-Label* to answer a reachability query (u, v).

Algorithm 4 *Yes-GRAIL* $(G, (u, v))$

```
1: if $L_v \not\subseteq L_u$ then
2:     return false; {$u \not\leadsto v$}
3: if $\mathcal{L}_v \subseteq \mathcal{L}_u$ then
4:     return true; {$u \leadsto v$}
5: for every $w$ of $(u, w) \in E(G)$ do
6:     if $L_v \subseteq L_w$ then
7:         if Yes-GRAIL $(G, (w, v))$ = true then
8:             return true;
9: return false; {$u \not\leadsto v$}
```

5. I/O COST MINIMIZATION

In this section, we discuss how to minimize I/O cost, since the *Yes-GRAIL* algorithm with both *Yes-Label* and *No-Label* still ends up a large number of random I/O accesses. We use $\mathcal{L}$ and L to denote the sets of *Yes-Label* and *No-Label* for every node in G. The sizes of $\mathcal{L}$ and L are denoted as $|\mathcal{L}|$ and $|L|$ in addition to the size of graph G, denoted as $|G|$. The problem we study here is how to minimize I/O cost when the memory allocated is M and the graph G and the codes generated is much larger than M, such as $M \ll |G| + |\mathcal{L}| + |L|$. Reconsider *Yes-GRAIL* in this scenario, given graph G and both $\mathcal{L}$ and L are stored on disk using B-sized blocks, the number of I/Os cannot be even bounded when answering a single reachability query (u, v). We explain the reason below. Let the total number of blocks be $(|G| + |\mathcal{L}| + |L|)/B$. Because $M \ll |G| + |\mathcal{L}| + |L|$, in the worst case, every block is missing in the buffer when needed. As shown in *Yes-GRAIL*, when both $\mathcal{L}_v \subseteq \mathcal{L}_u$ and $L_v \not\subseteq L_u$ are not true, it needs to check reachability from every child node of u, w, by issuing a set of intermediate reachability queries (w, v). If one of the children, w, can reach v, then u can reach v. In the worst case, the number of I/Os can be possibly, $P_d \cdot d_m^{P_d}$, where P_d is the diameter of graph G (the longest possible path) and d_m is the maximum degree of a node in G. Even though in real cases when considering the average degree d_a and the shortest path P_s, the possible number of I/Os is still as large as $P_s \cdot d_a^{P_s}$. Such I/O cost will be multiplied by $|Q|$ if there are $|Q|$ reachability queries to be answered.

The main ideas of our approach are as follows to process a single reachability query (u, v).

- First, we store all nodes of graph G in a topological order on disk. For a node u in G, we store all its information in an entry, denoted $\mathsf{entry}(u) = (L_u, \mathcal{L}_u, A_u)$, where L_u is the d *No-Label* $L_u = (L^1_u, L^2_u, \cdots, L^d_u)$, $\mathcal{L}_u$ is the t *Yes-Label* $\mathcal{L}_u = (\mathcal{L}^1_u, \mathcal{L}^2_u, \cdots, \mathcal{L}^t_u)$, and A_u is the adjacency list of u (a list of its children) in topological order. We denote it *YNG-Index*, because it contains all *Yes-Label*, *No-Label*, as well as G itself. The total $|V|$ entries are stored in B-sized pages on disk.
- Second, in order to minimize the number of I/O accesses to access *YNG-Index* in possible sequential I/Os, we use *BFS* (Breadth-First Search) instead of *DFS*. With *BFS*, it consumes memory space to maintain nodes in a heap in *BFS*. We maintain the heap in pages in the same buffer used for holding pages for *YNG-Index* in the same manner. In this way, the access to index and graph is bounded by |*YNG-Index*| with additional I/O cost to access the heap-on-disk.
- Third, the heap-on-disk reduces the number of I/Os accessing

Algorithm 5 *RQ-Run* (Q)

```
Input: A set of reachability queries $Q = \{(u_i, v_i)\}$
Output: The answers to $Q$
 1: Sort all the queries in $Q = \{(u_i, v_i)\}$ by $u_i$;
 2: Keep all the Yes-Label/No-Label for every $v_i$ in memory;
 3: Divide all $Q$ into $N$ partitions;
 4: Let $D_j$ be the $j$-th partition in which every entry is in the form of
    $(u_i, v_i, w_i)$ where initially $w_i = u_i$;
 5: for $j = 1$ to $N$ do
 6:   for each $(u_i, v_i, w_i)$ in topological order of $w_i$ in $D_j$ do
 7:     if the answer to $(u_i, v_i)$ has not been marked yes then
 8:       Access the index of $w_i$;
 9:       if $\mathcal{L}_{v_i} \subseteq \mathcal{L}_{w_i}$ then
10:         mark the answer to $(u_i, v_i)$ as yes;
11:       else if $L_{v_i} \not\subseteq L_{w_i}$ then
12:         mark the answer to $(u_i, v_i)$ as no;
13:       else
14:         for each $c_i$ of children of $w_i$ do
15:           if $c_i$ is before $v_i$ by topological order then
16:             put $(u_i, v_i, c_i)$ to the corresponding partition $D_k$;
```

YNG-Index, but it still ends up random I/O accesses. In order to maximize sequential I/O accesses, we divide the heap-on-disk into N partitions, $D_1, D_2, \cdots, D_N$. Recall all nodes in G are numbered based on a topological order. The D_i partition is used to keep user-given reachability query (u, v) or its intermediate reachability queries (w, v) in *BFS* if w is put into partition D_i. These intermediate reachability queries (w, v) are issued during *BFS* to check (u, v) by checking (w, v) instead given $u \rightsquigarrow w$ is ensured. The N partitions allow us to delay checking of some reachability queries in order to ensure sequential I/Os.

- Fourth, we expand the same partitioning strategy using sequential I/Os, to process a batch of user-given reachability queries Q. The main benefit of our approach to process a set of reachability queries Q is that it only needs to scan *YNG-Index* once at most.

The algorithm, called *RQ-Run*, is shown in Algorithm 5.

First, line 1, we sort all queries in $Q = \{(u_i, v_i)\}$ by increasing order of u_i for $1 \leq i \leq |Q|$. Recall in *YNG-Index*, all entries are stored following such an order. If $u < v$ then u is stored in a disk-page before the disk-page where v appears.

Second, line 2, for a pair of (u_i, v_i) in Q, u_i is the source and v_i is the destination. We retrieve all the *Yes-Label* and *No-Label* $(\mathcal{L}_{v_i}, L_{v_i})$ from $\mathsf{entry}(v_i)$ for every destination v_i in memory. It is important to note that we do not need to maintain the adjacency list A_{v_i} in $\mathsf{entry}(v_i)$ in main memory, and the size of $(\mathcal{L}_{v_i}, L_{v_i})$ is $d + t$ pairs of integers which is small in size. These codes are used to process reachability queries. It is important to notice that it is possible that the label of every v_i cannot hit the memory cache. In case when a cache miss occurs, we issue one more sequential scan on the *YNG-Index*, by sorting all queries in $Q = \{u_i, v_i\}$ by increasing order of v_i for $1 \leq i \leq |Q|$. Since labels in *YNG-Index* are sorted in topological order, we can ensure that one sequential scan is enough to load the labels of all v_is.

Third, line 3-4, we prepare N partitions, $(D_1, D_2, \cdots, D_N)$, of heap-on-disk, which will reside in memory when possible. Suppose there are $n = |V|$ nodes and the topological order is in the range of 1 to n. The j-th partition D_j is used to keep the reachability queries, if its source node u_i is in the range of $[(j-1)\lceil \frac{n}{N} \rceil + 1, j\lceil \frac{n}{N} \rceil]$. Initially, we put a reachability query (u_i, v_i) into D_j, if u_i is in its range, as (u_i, v_i, w_i) where $w_i = u_i$. We explain it below. For a given reachability query (u_i, v_i), if we cannot answer it by either *Yes-Label* or *No-Label* immediately, we need *BFS* by issuing a set of intermediate reachability queries $\{(w_i, v_i)\}$ if w_i is a child of u_i, because u_i can reach w_i and then u_i can reach v_i if w_i can reach v_i. In the triple of (u_i, v_i, w_i), the last two form a reachability query (w_i, v_i) to check reachability for the original reachability query (u_i, v_i), provided $u_i \rightsquigarrow w_i$ is known.

Step	D_1	D_2	D_3	D_4	*YNG-Index*
1	a				a
2	ab	e			kb
3	ab	ec	dhm	g	e

Table 3: The Traces

Fourth, line 5-16, in order to sequential access the *YNG-Index*, we process the partitions D_i in order, because all nodes are stored in *YNG-Index* in such an order. We load each partition D_j in order (line 5). Each time we load a partition D_j into memory, we access all entries (u_i, v_i, w_i) in D_j in topological order of w_i (line 6). Recall all D_js are partitioned based on topological order of w_i. In line 8, we ensure all w_is are accessed sequentially on *YNG-Index*. After loading the label of w_i, we can compare the label of w_i and the label of v_i since the label of all v_is have been loaded beforehand (line 2). There are three situations. (1) If $\mathcal{L}_{v_i} \subseteq \mathcal{L}_{w_i}$, we ensure that the answer of the original query (u_i, v_i) is yes and we stop expansion (line 9-10). (2) If $L_{v_i} \not\subseteq L_{w_i}$, we can also stop expansion because we can make sure the answer of any expanded query should be no (line 11-12). (3) If neither of the above two situations happens, we should expand the current result by adding all children of w_i. Since for each child c_i of w_i, the label of c_i is not loaded and loading the label of c_i in the current step will issue random I/Os. We simply put it into the corresponding partition D_k of c_i sequentially and the label of c_i will be loaded after loading the partition D_k later on.

We explain the sequential I/O using G in Fig. 3(a). Here, the nodes of G are stored in the *YNG-Index* in a topological order. We focus on the sequential I/Os by ignoring all the possible *Yes-Label*/*No-Label*. Assume $N = 3$, D_1, D_2, D_3 and D_4 are used to keep source nodes in akb, elc, dhm, and fgi respectively. Consider processing a reachability query (a, m). The steps are shown in Table 3. First, we load the labels and child nodes of a from the *YNG-Index*. We check that neither $\mathcal{L}_m \subseteq \mathcal{L}_a$ nor $L_m \not\subseteq L_a$ holds. We append the child b of a into D_1 (D_1 is now in memory) and append the child e of a into D_2 (on disk). We pop b from the memory heap D_1, and visit k and b sequentially on *YNG-Index* in order to load the labels and child nodes of b. We check that neither $\mathcal{L}_m \subseteq \mathcal{L}_b$ nor $L_m \not\subseteq L_b$ holds, and thus we continue to expand b to append its child nodes into the corresponding partitions. Next, we load partition D_2 with two elements e and c into memory. We load the labels and child nodes of e from the *YNG-Index*. We find that $\mathcal{L}_m \subseteq \mathcal{L}_e$. We stop expansion and conclude that $a \rightsquigarrow m$.

5.1 The Number of I/Os

Consider a single reachability query $q = (u, v)$. We use $T(q)$ to denote a subset of nodes in G that satisfy all of the following conditions:

1. $u \in T(q)$.

2. For each $w \in T(q)$, $u \rightsquigarrow w$.
3. For each $w \in T(q)$, $L_v \subseteq L_w$.
4. For each $w \in T(q)$, $\mathcal{L}_v \nsubseteq \mathcal{L}_w$.
5. For each $w \in T(q)$, if $w \neq u$, then there exists $(w', w) \in E(G)$, such that $w' \in T(q)$.

$T(q)$ is actually the maximum number of intermediate reachability queries needed to be expanded when answering the query q using Algorithm 5. Usually, $T(q) \ll |V(G)|$. When the answer to q is yes, the number of intermediate reachability queries can be even less, because once there is a certain $w \in T(q)$ such that $\mathcal{L}_v \subseteq \mathcal{L}_w$, we can stop all expansions of intermediate queries and return yes as the final answer. We have the following lemma.

Lemma 5.2: *The number of I/Os needed to answer a single reachability query $q = (u, v)$ is bounded by $O(\frac{|G|+|L|+|\mathcal{L}|+|T(q)|}{B}+N)$, where $|G| = |V(G)| + |E(G)|$ is the size of the graph, $|L|$ and $|\mathcal{L}|$ are the sizes of* No-Label *and* Yes-Label *respectively, B is the block size, and N is the number of partitions.* □

Proof Sketch: Since we need to scan the whole *YNG-Index* with size $O(|L| + |\mathcal{L}| + |Q|)$ once sequentially. The number of I/Os used in index scan is at most $O(\frac{|G|+|L|+|\mathcal{L}|}{B})$. We also need to scan the N partitions which keep intermediate queries sequentially. Suppose the sizes of the N partitions are P_1, P_2, ..., P_N, the total number of I/Os by scanning the partitions is at most $\sum_{i=1}^{N}(\lfloor \frac{P_i}{B} \rfloor + 1) \leq \frac{\sum_{i=1}^{N} P_i}{B} + N = \frac{T(q)}{B} + N$. All together, we conclude that the number of I/Os is bounded by $O(\frac{|G|+|L|+|\mathcal{L}|+|T(q)|}{B} + N)$. □

As shown in Lemma 5.2, we need to scan the whole *YNG-Index* to answer a single query. Actually, we can do even better as follows. In order to answer a single query, we do not need to scan every block of the *YNG-Index*. We can jump to the next intermediate query in the heap-on-disk with smallest topological order directly. Since intermediate queries in the heap-on-disk are visited in topological order. We only need to jump in a forward manner and never need to go back. In such a way, the number of I/Os can be bounded by $O(|T(q)|)$ when $|T(q)| < \frac{|G|+|L|+|\mathcal{L}|}{B}$.

An important feature of Algorithm 5 is that, when answering multiple queries, the number of I/Os caused on scanning *YNG-Index* does not increase. The only thing that influences the number of I/Os is the number of intermediate queries generated. We have the following lemma.

Lemma 5.3: *The number of I/Os that are needed to answer a set of reachability queries, $Q = \{(u_i, v_i)\}$, for $1 \leq i \leq |Q|$, is bounded by $O(\frac{|G|+|L|+|\mathcal{L}|+\sum_{q \in Q}|T(q)|}{B} + N)$, where $|G| = |V(G)| + |E(G)|$ is the size of the graph, $|L|$ and $|\mathcal{L}|$ are the sizes of* No-Label *and* Yes-Label *respectively, B is the block size, and N is the number of partitions.* □

Proof Sketch: In Algorithm 5, we scan *YNG-Index* once to load the L and $\mathcal{L}$ labels for every intermediate query generated by the multiple queries when needed. The I/O cost on *YNG-Index* scan is still $O(\frac{|G|+|L|+|\mathcal{L}|}{B})$. For I/Os cost caused by the intermediate queries, suppose the sizes of the N partitions are P_1, P_2, ..., P_N to keep all the intermediate results for all queries. The I/O cost is at most $\sum_{i=1}^{N}(\lfloor \frac{P_i}{B} \rfloor + 1) \leq \frac{\sum_{i=1}^{N} P_i}{B} + N$, since $\sum_{i=1}^{N} P_i = \sum_{q \in Q}|T(q)|$. By putting all together, we conclude that the number of I/Os is bounded by $O(\frac{|G|+|L|+|\mathcal{L}|+\sum_{q \in Q}|T(q)|}{B} + N)$. □

The optimal I/O for the reachability queries depends on (a) the quality of the *Yes-Label*/*No-Label*, and (b) the traversal order. The factor (a) determines the total size of the intermediate results which may need to be stored on disk, and the factor (b) determines how to access such intermediate results. For the factor (a), the *No-Label* is well studied, and we introduce the *Yes-Label* which further reduces the number of intermediate results when processing queries. For the factor (b), we minimize the I/O cost by sequentially scanning the intermediate results. The reason is that, for reachability queries, if we have obtained the answers, no traversing is needed, otherwise all the intermediate results need to be checked. With all the intermediate results on disk, *Yes-Label* reduces the intermediate results, and at the same time *RQ-Run* scan the intermediate results only once. In total, *RQ-Run* makes the I/O cost minimized.

5.2 Memory Consumption and Sequential I/Os

We discuss the least memory consumption in order to ensure the sequential I/Os. In order to answer multiple queries $Q = \{(u_i, v_i)\}$ using Algorithm 5, where only sequential I/Os are allowed. The memory consumption consists of three parts: 1) the memory used for the initial queries, 2) the memory used for the *YNG-Index*, and 3) The memory used for the intermediate queries.

For 1), first, we need to load the labels of all v_i into memory. Because each time we generate a new intermediate query (w_i, v_i), after loading the label of w_i sequentially, we need to check whether $L_{v_i} \nsubseteq L_{w_i}$ or $\mathcal{L}_{v_i} \subseteq \mathcal{L}_{w_i}$ in order to prune (w_i, v_i). If we load the label of v_i from disk, we issue a random I/O on the *YNG-Index*, which is not expected. Second, we need to know the updated result of every query $(u_i, v_i) \in Q$. It is because every time if we know $\mathcal{L}_{v_i} \subseteq \mathcal{L}_{w_i}$ for a certain intermediate result (u_i, v_i, w_i), we know the answer of the original query (u_i, v_i) is yes and we can prune all intermediate results generated from the query (u_i, v_i). This part can be done using a bitmap of size $|Q|$. The total memory used for the initial queries should be $O(|Q|)$.

For 2), we only need to scan *YNG-Index* once. Each time, we load B bytes from *YNG-Index*. The memory used for the *YNG-Index* is bounded by $O(B)$.

For 3), for a set of queries Q and a certain query $q \in Q$, we define $T_j(q) = T(q) \cup D_j$. We have $D_j = \sum_{q \in Q} T_j(q)$. The memory used for the intermediate queries should be $\max_{j=1}^{N}\{D_j\} = \max_{j=1}^{N}\{\sum_{q \in Q} T_j(q)\}$. Suppose on average, the number of intermediate queries generated in each partition for a certain query q is I. The average memory used for the intermediate queries should be $O(I \times |Q|)$. We have the following lemma.

Lemma 5.4: *In order to ensure sequential I/Os when answering a set of reachability queries Q, the memory consumption is $O(|Q| + B + I \times |Q|)$ on average.* □

Proof Sketch: The result can be easily derived from the above discussions. □

When a certain $T_j(q)$ becomes too large, it is possible that $|T_j(q)|$ is not bounded by $O(I)$. We will discuss such issues in the next

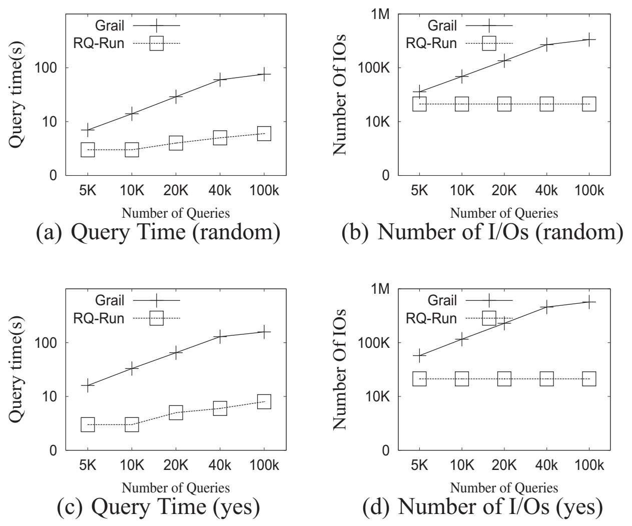


Figure 5: Vary Number of Queries $|Q|$ (Synthetic)

Parameter	Range	Default
Number of Queries $\|Q\|$	5K, 10K, 20K, 40K, 100K	20K
Memory Ratio M	0.2, 0.4, 0.6, 0.8, 1.0	-
Block Size B	16K, 32K, 64K, 128K	64K
Yes-Query Ratio R	0, 0.2, 0.4, 0.6, 0.8, 1	-
Graph Size $\|V(G)\|$	5M, 10M, 20M, 30M	10M
Average Degree D	1, 2, 3, 4, 5	3

Table 4: Parameters

part.

5.3 Dealing with Possible Cores

In our experiments, we find that for some real graphs, there are some nodes with extremely large out degrees. For example, in the *citeseerx* dataset used in our experiments, there are 6,540,399 nodes and 15,011,259 edges. The maximum out degree for a certain node is larger than 180,000. We call each node with extremely large out degree a core. The number of cores in each graph is usually small.

Consider an intermediate query (u_i, v_i, w_i), where w_i is a core. If neither $L_{v_i} \not\subseteq L_{w_i}$ nor $\mathcal{L}_{v_i} \subseteq \mathcal{L}_{w_i}$, we will expand w_i and put a large number of intermediate queries into the heap-on-disk. In such a situation, when answering $q = (u_i, v_i)$, it is possible that $|T_j(q)|$ is not bounded by $O(I)$ for a certain partition D_j.

In order to handle cores, for each core $u \in V(G)$, we precompute all the nodes that u can reach and manage them using a bitmap. We load the bitmaps of cores into memory before processing queries. As an indicator, suppose there are $10,000,000$ nodes and 5 cores in the graph. We need $5 \times \frac{10,000,000}{8}$ bytes to store all 5 bitmaps, which is less than 6 MB memory. After loading all bitmaps into memory, when processing queries, for each intermediate query (u_i, v_i, w_i) where w_i is a core, we can answer the query (w_i, v_i) in $O(1)$ time using the in-memory bitmaps, and do not need to expand w_i. In such a way, we can avoid adding a large number of child nodes of w_i into the heap-on-disk and thus bound the total memory consumed.

6. PERFORMANCE STUDIES

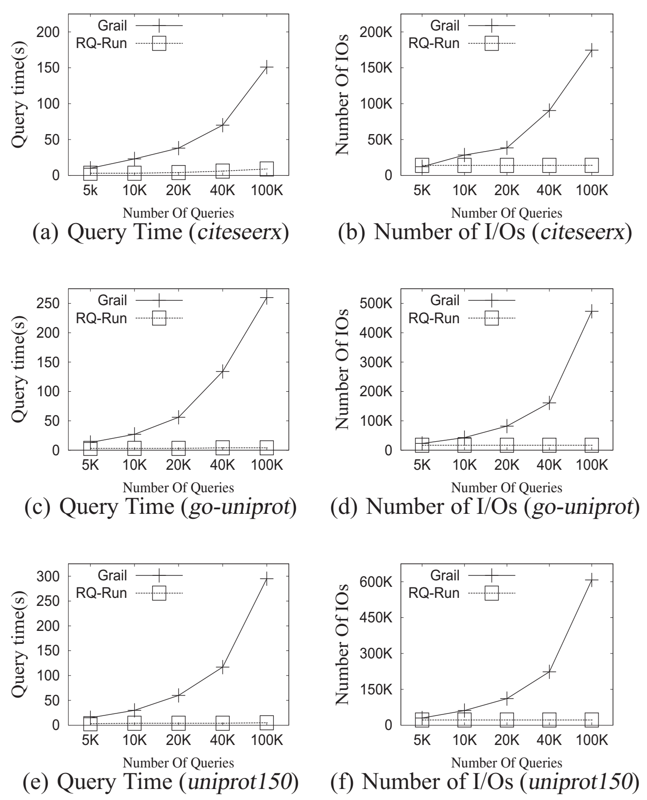


Figure 6: Vary Number of Queries $|Q|$ (Real)

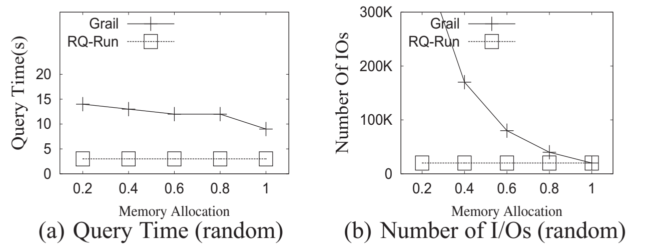
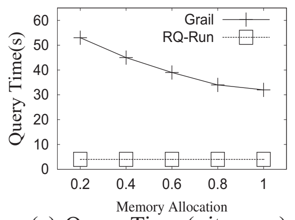


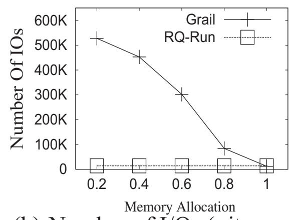
Figure 7: Vary Available Memory Ratio M (Synthetic)

We conducted extensive performance studies to test our approaches. We compare two algorithms, denoted as *RQ-Run* and *GRAIL*. *RQ-Run* denotes our approach to process multiple reachability queries using Algorithm 5. *GRAIL* denotes the approach introduced in [23]. The LRU (Least Recently Used) buffer replacement strategy is used when the memory is not enough. We only compare our approach with *GRAIL* because it is the currently known only approach that can handle massive graphs in terms of index construction time/space. For each test, we record the query processing time as well as the total number of I/Os. All the algorithms were implemented using Visual C++ 2005 and tested on a PC with 2.66GHz CPU and 3.43GB memory running Windows XP.

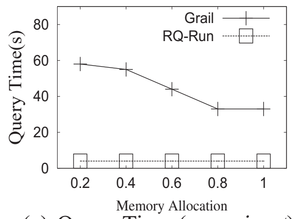
Datasets: We use three large real datasets, namely, *citeseerx*, *go-uniprot*, and *uniprot150*. *citeseerx* (citeseerx.ist.psu.edu) includes citations among papers. The graph of *citeseerx* contains 6,540,399 nodes (papers) and 15,011,259 edges (citations), with average degree 2.30 for each node. *go-uniprot* includes the Gene ontology and annotations from the UniProt dataset (www.uniprot.org). The



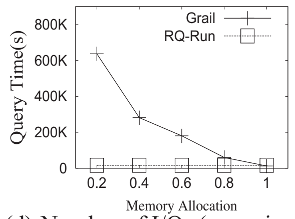
(a) Query Time (*citeseerx*)



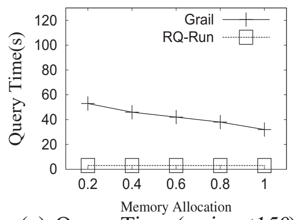
(b) Number of I/Os (*citeseerx*)



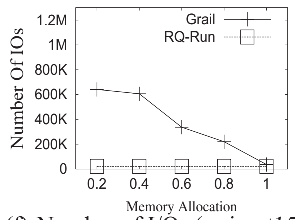
(c) Query Time (*go-uniprot*)



(d) Number of I/Os (*go-uniprot*)



(e) Query Time (*uniprot150*)



(f) Number of I/Os (*uniprot150*)

Figure 8: Vary Available Memory Ratio M (Real)

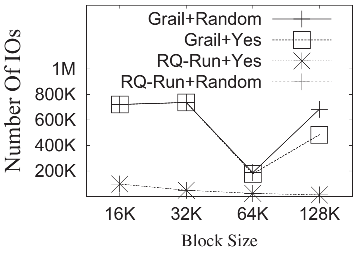
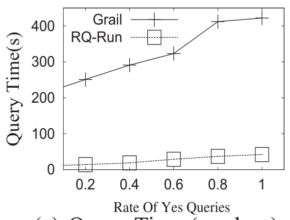


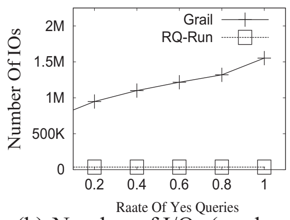
Figure 9: Vary Block Size B (Synthetic)

graph of *go-uniprot* contains 6,967,956 nodes and 34,770,235 edges, with average degree 4.99 for each node. The *uniprot150* dataset is obtained from the RDF graph of the UniProt dataset. The graph of *uniprot150* contains 25,037,600 nodes and 25,037,598 edges, with average degree 1.0 for each node. Among the three datasets, *go-uniprot* is a dense graph and *uniprot150* is a sparse graph. The density of the *citeseerx* dataset is between *go-uniprot* and *uniprot150*. The *citeseerx* dataset contain cores (nodes with very large degree) and the *go-uniprot* and *uniprot150* datasets do not contain cores. When dealing with cores, we use the bitmap technique in both *GRAIL* and our *RQ-Run* algorithm. We also generate several large synthetic datasets with number of nodes ranging from 5,000,000 to 30,000,000. We use the graph generation algorithm used in *GRAIL* [23] to generate synthetic graphs.

Queries: For each dataset, we randomly select a set of queries such that every pair (u, v) will be selected with the same probability. We found that the answers for randomly selected queries are almost no-queries for all datasets. We prepare another set of yes-

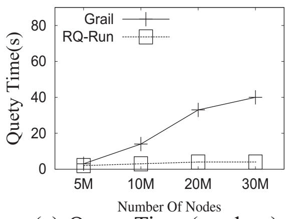


(a) Query Time (random)

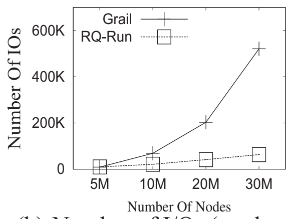


(b) Number of I/Os (random)

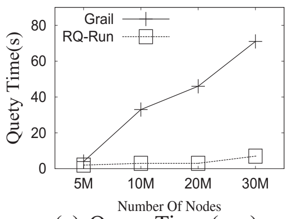
Figure 10: Vary Ratio of Yes-Queries R (Synthetic)



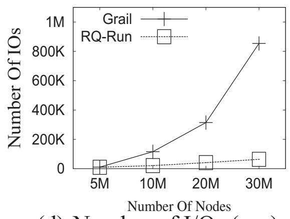
(a) Query Time (random)



(b) Number of I/Os (random)



(c) Query Time (yes)



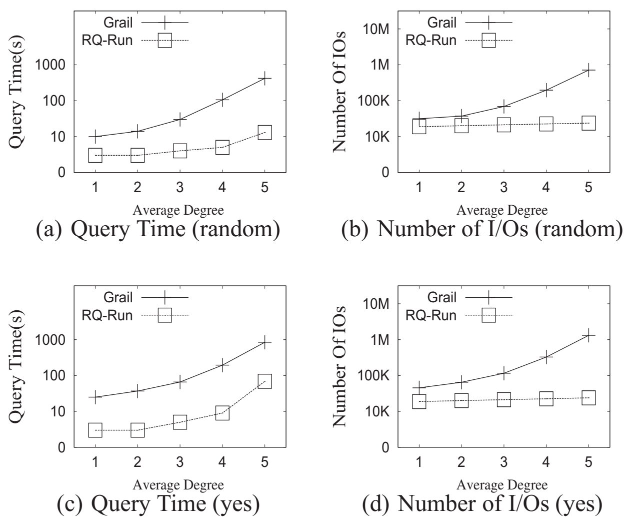
(d) Number of I/Os (yes)

Figure 11: Vary Graph Size $|V(G)|$(Synthetic)

queries. Every pair (u, v) with $u \rightsquigarrow v$ will be selected with the same probability. We do our testing on the two query sets for every case. We also vary the ratio of yes-queries by randomly selecting a certain number of queries from each of the two query sets.

Index Construction: For *GRAIL*, the index includes the graph information and the *No-Label* for each node in the graph. For each node, we create five *No-Label* which is the same as used in *GRAIL*. For *RQ-Run*, the index include the graph information, the *No-Label* for each node, as well as the *Yes-Label* for each node. For each node, we create two *Yes-Label* and three *No-Label*, which has best performance considering the size of the index, the effectiveness of pruning intermediate results, and the memory usage when processing queries. The time for creating each index is less than five minutes. For each dataset, the index size for *GRAIL* is the same with the index size for *RQ-Run*. For the *citeseerx* dataset, the size of the original graph is 172,413,264 bytes and the size of the index is 347,822,592 bytes for both *GRAIL* and *RQ-Run*. For the *go-uniprot* dataset, the size of the original graph is 278,273,152 bytes and the size of the index is 445,671,004 bytes for both *GRAIL* and *RQ-Run*. For the *uniprot150* dataset, the size of the original graph is 400,601,584 bytes and the size of the index is 1,201,804,792 bytes for both *GRAIL* and *RQ-Run*. For synthetic dataset, the default graph contain 10,000,000 nodes and 30,000,000 edges. The size of original graph is 320,000,000 bytes and the size of index is 720,000,000 bytes.

Parameters: We vary several parameters to test the scalability of



(a) Query Time (random) (b) Number of I/Os (random)

(c) Query Time (yes) (d) Number of I/Os (yes)

Figure 12: Vary Average Degree D (Synthetic)

our approach. (1) The number of queries $|Q|$ is the number of reachability queries used in each test. (2) The available memory ratio M is the ratio of the available memory used to the size of the index. When the available memory ratio is 1, it means the whole index including the graph can be kept in memory. Since the index sizes of *GRAIL* and *RQ-Run* are the same. For a certain M, the available memory used for both *GRAIL* and *RQ-Run* is the same. When not varying memory ratio, we set our default available memory to be 128MB for all datasets. (3) The block size B is the size of each disk page. (4) The graph size $|V(G)|$ is the number of nodes in the graph. (5) The average degree D is the average number of adjacent edges for each node in the graph. We have $D = \frac{|E(G)|}{|V(G)|}$. The ranges for all parameters and their default values are shown in Table 4. Unless otherwise stated, when varying a certain parameter, we will use the default value for all the other parameters.

Vary Number of Queries $|Q|$: We vary the number of queries $|Q|$ from 5,000 to 100,000. The results on the synthetic dataset are shown in Fig. 5. Fig. 5 (a) and Fig. 5 (b) show the processing time and number of I/Os using the random query set. When the number of queries $|Q|$ increases, the processing time for both *GRAIL* and *RQ-Run* increase. The processing time for *GRAIL* increases sharply when $|Q|$ is large, because processing a large number of reachability queries will produce a large number of random I/O accesses. When $|Q|$ is 100,000, *RQ-Run* is more than ten times faster than *GRAIL*. When $|Q|$ increases, the I/O cost for *GRAIL* also increases sharply, while the *RQ-Run* algorithm keeps stable. This is because *RQ-Run* scan the index only once no matter how many queries in the query set Q need to be processed. The extra I/O cost using *RQ-Run* is only on scanning the intermediate tuples in the heap-on-disk sequentially. Also the gap between *RQ-Run* and *GRAIL* for yes-queries is larger than the gap between *RQ-Run* and *GRAIL* for random queries. This suggests that the *Yes-Label* has advantages when the number of yes-queries is large. Fig. 5 (c) and Fig. 5 (d) show the processing time and number of I/Os using the yes-query set. The curves for *RQ-Run* are similar to those of random queries. For *GRAIL*, answering yes-queries take much more time and cost much more I/O accesses than answering random queries. This is because *GRAIL* is a *No-Label* based algorithm, when answering yes-queries, it needs to expand the whole path from the source node to destination node for every yes-query. The expansion takes a lot of time, and consumes a lot of random I/Os. As shown in this figure, the effectiveness of the breadth first traversal and *Yes-Label* are obvious.

The results on real datasets using the random query set are shown in Fig. 6. Fig. 6 (a) and Fig. 6 (b) show the processing time and number of I/Os when varying $|Q|$ on the *citeseerx* dataset. When the number of queries $|Q|$ is small, the gap between *GRAIL* and *RQ-Run* is small, but when $|Q|$ increases, the processing time and number of I/Os for *GRAIL* increase sharply while those for *RQ-Run* increase slowly. This is because when $|Q|$ is small, the dominant part for *RQ-Run* is on scanning the *YNG-Index*, which is costly. When $|Q|$ becomes large, the cost on scanning the *YNG-Index* does not increase with the number of queries. So the extra I/O and time cost are only spent on scanning the intermediate tuples sequentially. Such cost increases linearly with the number of queries. The processing time and I/O cost for the *go-uniprot* dataset are shown in Fig. 6 (c) and Fig. 6 (d) respectively, and the results for the *uniprot150* dataset are shown in Fig. 6 (e) and Fig. 6 (f). The results for both *go-uniprot* and *uniprot150* are similar to those on the *citeseerx* dataset. The results for real datasets using the yes-query set are similar to those on the synthetic dataset. We do not show them for the lack of space.

Vary Available Memory Ratio M: We vary the available memory ratio M from 0.2 to 1. Fig. 7 shows the results on the synthetic dataset. When M increases, the processing time and the number of I/Os for *GRAIL* decrease while the processing time and the number of I/Os for *RQ-Run* are nearly unchanged. It is because *RQ-Run* only produces sequential I/Os, and the memory consumption for *RQ-Run* is very small in order to ensure sequential I/Os. When M increases to 1, the number of I/Os for *GRAIL* is nearly zero, but the processing time for *GRAIL* is still 4 times larger then *RQ-Run*. This is because, other than keeping the index in memory, *GRAIL* needs extra I/Os to keep the intermediate queries, and needs cost to process LRU in order to reduce the cache missing. Fig. 8 (a) and Fig. 8 (b) show the results on the *citeseerx* dataset, Fig. 8 (c) and Fig. 8 (d) show the results on the *go-uniprot* dataset, and Fig. 8 (e) and Fig. 8 (f) show the results on the *uniprot150* dataset. The performances on all real datasets are similar to those on the synthetic dataset.

Vary Block Size B: Fig. 9 shows the number of I/Os generated when varying the block size B from 16K to 128K for both yes-query set and random query set on the synthetic dataset. Since our *RQ-Run* algorithm only needs sequential accesses on disk, a large block size will reduce the number of I/Os significantly. For *GRAIL*, a large number of random accesses will be produced. When the block size B is small, each jump on the disk will possibly ends up an I/O. When the block size B is large, the probability of cache miss will be high. As shown in Fig. 9, the block size of 64KB produces the least number of I/Os. The results on the real datasets are similar to those on the synthetic dataset.

Vary Ratio of Yes-Queries R: We vary the ratio of yes-queries R from 0 to 1. The processing time and number of I/Os for the synthetic dataset are shown in Fig. 10 (a) and Fig. 10 (b) respectively. When the ratio of yes-queries increases, the processing time and the number of I/Os for both *GRAIL* and *RQ-Run* increase. This is because for *GRAIL*, it needs to traverse the whole path from source to destination for each yes-query, and for *RQ-Run*, the cost of answering a yes-query is also higher than the cost of answering a no query. *GRAIL* increases more sharply because *RQ-Run* can stop half way in the path from source to destination for a yes-query, as long as the *Yes-Label* containment condition is satisfied. The results on the real datasets are similar to those on the synthetic dataset.

Vary Graph Size $|V(G)|$: We vary the number of nodes $|V(G)|$ in the graph G from 5,000,000 to 30,000,000 and generate four synthetic datasets. Fig. 11 (a) and Fig. 11 (b) show the processing time and number of I/Os used to process the random queries respectively. Fig. 11 (c) and Fig. 11 (d) show the processing time and number of I/Os used to process the yes-queries respectively. When the graph size increases, the processing time and number of I/Os for all test cases increase. *GRAIL* increases sharply while *RQ-Run* increases slowly in all situations. When $|V(G)|$ increases to 30M, *RQ-Run* is 10 times faster than *GRAIL* for random queries and more than 15 times faster than *GRAIL* for yes-queries.

Vary Average Degree D: We vary the average degree D of nodes in a graph from 1 to 5. The results are shown in Fig. 12. Fig. 12 (a) and Fig. 12 (b) show the results for random queries, and Fig. 12 (c) and Fig. 12 (d) show the results for yes-queries. When D increases, the processing time and number of I/Os for all cases increase. This is because, when D increases, each node will generate more intermediate queries. The number of I/Os for *RQ-Run* increases slowly. This is because although it needs time to check intermediate queries, the actually number of intermediate queries written on disk dose not increase sharply. *RQ-Run* is more than 8 times faster than *GRAIL* for random queries and more than 10 times faster than *GRAIL* for yes-queries in all cases.

Single Query Testing: We test the performance for single queries on a graph with 10M nodes, 30M edges, and an average degree 3. The graph is 320,000,000 bytes and 640,000,000 bytes with index. We set the main memory to be 128M. For the initialization, *GRAIL* takes 13 seconds and *RQ-Run* takes 1 second, since *GRAIL* needs to load part of a graph into main memory. We set the page size to be 64K and select 100 single queries to be tested one by one, using both *RQ-Run* and *GRAIL*. To process an yes-query, on average, *RQ-Run* needs 10 I/Os and *GRAIL* needs 14 I/Os. To process a random query, on average, *RQ-Run* needs 6 I/Os and *GRAIL* needs 9 I/Os. We report the I/Os because the running time is unstable due to the effect of system buffer controled by the operating system. In general, *RQ-Run* outperforms *GRAIL* when processing a single query.

7. CONCLUSIONS

In this paper, we study how to answer reachability queries on massive graphs which cannot be kept entirely in memory. We analyze *GRAIL*, a *No-Label* based in memory algorithm, which is the only existing solution to answer reachability queries on massive graphs. We find that when the graph and the labels can not be kept entirely in memory, the *GRAIL* approach will issue a large number of random I/Os. We introduce a new *Yes-Label* based labeling scheme, as a complement of the *No-Label* used in *GRAIL*, to reduce the number of random I/Os when answering yes-queries. We introduce a *YNG-Index* to store all *No-Label*, *Yes-Label*, as well as the graph itself. We study how to minimize the number of I/Os using a heap-on-disk when scanning the *YNG-Index* sequentially. We propose methods to partition the heap-on-disk, in order to ensure that only sequential I/Os are performed. We show how to extend our approaches to answer multiple queries effectively and we give analysis on the number of I/Os. We conducted extensive performance studies on large synthetic and real graphs, and confirm the efficiency of our approaches.

Acknowledgment: The work was supported by grants of the Research Grants Council of the Hong Kong SAR, China, No. CUHK 419008 and CUHK 419109.

8. REFERENCES

[1] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *Proc. of SIGMOD'89*, 1989.

[2] K. Anyanwu and A. Sheth. ρ-queries: enabling querying for semantic associations on the semantic web. In *Proc. of WWW'03*, 2003.

[3] R. Bramandia, B. Choi, and W. K. Ng. On incremental maintenance of 2-hop labeling of graphs. In *Proc of WWW'08)*, 2008.

[4] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *Proc. of VLDB'05*, 2005.

[5] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *Proc. of ICDE'08*, 2008.

[6] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. In *Proc. of EDBT'06*, 2006.

[7] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *Proc. of EDBT'08*, 2008.

[8] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proc. of SODA'02*, 2002.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 2001.

[10] H. He, H. Wang, J. Yang, and P. S. Yu. Compact reachability labeling for graph-structured data. In *Proc. of CIKM'05*, 2005.

[11] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.

[12] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-HOP: A high-compression indexing scheme for reachability query. In *Proc. of SIGMOD'09*, 2009.

[13] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *Proc. of SIGMOD'08*, 2008.

[14] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proc. of STOC'04*, 2004.

[15] R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex XML document collections. In *Proc. of EDBT'04*, 2004.

[16] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the HOPI index for complex XML document collections. In *Proc. of ICDE'05*, 2005.

[17] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theor. Comput. Sci.*, 58(1-3):325–346, 1988.

[18] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *Proc. of SIGMOD'07*, 2007.

[19] J. van Helden, A. Naim, R. Mancuso, , M. Eldridge, L. Wernisch, D. Gilbert, and S. Wodak. Reresenting and analysing molecular and cellular function using the computer. *Journal of Biological Chemistry*, 381(9-10), 2000.

[20] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *Proc. of SIGMOD'11*, 2011.

[21] J. S. Vitter. Algorithms and data structures for external memory. *Found. Trends Theor. Comput. Sci.*, 2:305–474, January 2008.

[22] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proc. of ICDE'06*, 2006.

[23] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *PVLDB*, 3(1), 2010.