Interval-Index: A Scalable and Fast Approach for Reachability Queries in Large Graphs

Fangxu Li, Pingpeng Yuan(✉), and Hai Jin

Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Huazhong University of Science and Technology, Wuhan 430074, China
{ppyuan,hjin}@mail.hust.edu.cn

Abstract. Now more and more large graphs are available. One interesting problem is how to effectively find reachability between any vertex pairs in a very large graph. Multiple approaches have been proposed to answer reachability queries. However, most approaches only perform well on small graphs. Processing reachability queries on large graphs requires much storage and computation and still remains challenges. In this paper, we propose a scalable and fast indexing approach called Interval-Index, based on traversal tree-based partitioning and relabeling scheme. Our approach has several unique features: first, the traversal tree-based partitioning ensures access locality and parallelism in computation; second, continuous relabeling ensures fast querying and saves search space; third, we convert the entire graph database into a traversal tree graph on a smaller scale, to reach a compact storage structure. Finally, we run extensive experiments on synthetic graphs and real graphs with different sizes, and show that Interval-Index approach outperforms the state-of-the-art Feline in both storage size and the performance of query execution.

1 Introduction

Highly connected data sets have increased exponentially over the past several years. For example, Facebook had more than 800 million users by the end of 2011 [2]. With the continued growth of graph-structured data, a common graph application is to query whether there exist paths between two vertices in a graph, namely reachability queries. In most cases of reachability queries, users always lack exact knowledge of the graph. They can only provide a rough query description. Thus, it is difficult to describe reachability queries using SQL-style (SPARQL, etc.) languages due to their uncertain patterns.

Diverse approaches have been proposed to answer reachability queries. However, most approaches only perform well on relatively small graphs, with hundreds of thousands of vertices and edges at most [3]. But for processing larger graphs, they are either too costly in storage or slow in query time. For example, Path-tree [11] has the restriction on scalability, as its index size on large dense graphs may be very large. So many approaches, e.g., PWAH [12], GRAIL [4], SCARAB [3], TF-Label [6] and Feline [7] are proposed to process large graphs recently. However, the approaches are still crucial for performance limits.

S. Zhang et al. (Eds.): KSEM 2015, LNAI 9403, pp. 224–235, 2015.
DOI: 10.1007/978-3-319-25159-2_21

For example, PWAH requires large memory to hold index in order to ensure efficiency. So it is not scalable for large graphs. For GRAIL, the inclusion relation between labels is necessary to indicate reachability, but not a sufficient one. So GRAIL can only answer un-reachability between two vertices, otherwise GRAIL requires to traverse the graph using DFS. Feline needs to keep the primitive graph in memory, since it cannot answer reachability merely by the index. So it requires long time to load data, and large memory. Like GRAIL, partial order-based scheme in Feline is not a sufficient condition for reachability. So DFS is also required to exclude those un-reachable vertex pairs in the worst case.

In this paper, we present a scalable and fast indexing approach for very large graphs, named as Interval-Index. Our approach is motivated by our observation that processing reachability queries need traverse graphs. But traversal through large graphs will bring immeasurable time and space consumption. Therefore, we first partition a large graph into tree-based partitions to improve access locality and reduce search space. In order to build interval indexes on partitions easily, we further assign vertices of each partition continuous IDs in parallel. By this means, we further maximize sequential access and minimize random access on storage media. Third, since vertex IDs of each partition are continuous, we build an interval index which helps to determine the reachability in each tree-partition. Our approach can prune search space greatly during answering reachability queries. The main contributions of our research are as follows.

- A traversal tree-based partitioning approach is proposed to search each partition in parallel. Partitioning a graph into multiple smaller trees does not break the relationships of graph. Furthermore, it ensures access locality and reduces the storage space.
- An efficient and scalable graph indexing technique - Interval-Index is proposed to help quickly determine which trees a vertex is in. In order to facilitate building interval index, vertices of each partition are assigned continuous IDs so that vertex IDs of each partition will not overlap. The relabeling scheme ensures high efficiency in pruning of search space during answering reachability queries.

2 Preliminary

Since undirected graphs can be converted into directed connected graphs and disconnected graphs can be split into connected subgraphs, thus, we will mainly discuss directed connected graph in the following.

Definition 1 (Graph). *A graph $G = (V, E)$ is defined by a finite set V of vertices, $\{v_i | v_i \in V\}$ $(0 \leq i < |V|)$, and a set E of directed edges which connect certain pairs of vertices, and $E = \{e = (u, v) \in E | u, v \in V\}$.*

The in-degree of vertex u is denoted as $deg^-(u) = |\{v|(v, u) \in G\}|$. Its out-degree $deg^+(u) = |\{v|(u, v) \in G\}|$, and $d(u) = deg^-(u) + deg^+(u)$.

Definition 2 (Traversal Tree). *A traversal tree $T = (V_T, E_T)$ of $G = (V, E)$ is defined as follows: 1) $E_T \subseteq E$; (2) $\exists v_r \in V_T$, $\forall u \in V_T$,$(u, v_r) \notin E_T$. v_r is the root of T; (3) $\forall u \in V_T$, u satisfies the following conditions: (i) $\exists v \in V$, u is v or u is a dummy copy of v; (ii) $\exists v \in V_T$ and v is a child of u s.t. $(u, v) \in E_T$; (iii) if u is a leaf of T, u is a dummy vertex or $deg^+(u) = 0$. if u is a non-leaf of T, u has at most $deg^+(u)$ children.*

Definition 3 (Rooted Traversal Tree). *An Rooted Traversal Tree (RT-Tree) $R = (V_T,\ E_T)$ of $G = (V, E)$ is a traversal tree starting from a vertex $v \in V$ whose $deg^-(v)$=0.*

For instance, the tree indicated by the dotted line in Fig. 1(a) is an RT-Tree. G may have multiple RT-Trees. The number of RT-Trees of G depends on the number of vertices with zero in-degree. The intersection of two RT-Trees may be not empty. Here, we define the intersection of two RT-Trees as pivotal tree because two or more RT-Trees share it.

Definition 4 (Pivotal Tree). *Given a directed graph $G = (V, E)$, assume $R_m, R_n (m \neq n)$ are RT-Trees of G. The pivotal tree P_T of R_m and R_n is defined as follows: (i) $P_T = R_m \bigcap R_n$; (ii) $\exists < v_0, v_1, ..., v_k >$, such that $v_0 = v_{rt}(R_i)(i = m, n), v_k = v_{rt}(P_T), \forall j \in [0, k-1] : (v_j, v_{j+1}) \in R_i$. The root of pivotal tree is named as pivotal vertex.*

Achieving high performance in processing a large scale graph requires partitioning of the graph. Here, we partition a graph into multiple non-overlaping traversal trees. In the following, traversal trees indicate non-overlapping traversal trees if we do not state explicitly.

Definition 5 (k-way Partition). *Let $P_1, P_2, \ldots, P_k$ denote a set of traversal trees of $G = (V, E)$, where (1) $P_i = (V_{P_i}, E_{P_i})$, $V_{P_1} \cup V_{P_2} \cup \ldots \cup V_{P_k} = V$, $V_{P_i} \cap V_{P_j} = \emptyset (i \neq j)$; (2) $\forall (u, v) \in E, u \in V_{P_i}$, $v \in V_{P_i} \Rightarrow (u, v) \in E_{P_i}$. $P_1, P_2, \ldots, P_k$ $(1 \leq k \leq |V|)$ are named as k partitions of G.*

Definition 6 (Graph of Traversal Trees). *Given a directed graph $G = (V, E)$, and $\mathcal{T}$ is the set of non-overlapping traversal trees, which is a partition of G. $|\mathcal{T}| \geq 1$. The graph of traversal trees $G_T = (V_{G_T}, E_{G_T})$ of G is defined as follows: (i) The function $f : \mathcal{T} \leftrightarrow V_{G_T}$ is bijective. Namely, each traversal tree in $\mathcal{T}$ is mapping into a vertex of V_{G_T}; (ii)$\forall T_i, T_j \in \mathcal{T} (i \neq j)$, if the root of T_j is reachable from the root of T_i, $(f(T_i), f(T_j)) \in E_{G_T}$.*

3 The Interval-Index Approach

In our approach, we first traverse a graph and construct its RT-Trees, and then split these RT-Trees into multiple non-overlapping traversal trees based partitions. So parallelism and locality of the computation can be improved. The vertices in the same partition will be assigned continuous IDs in order to build a compact index easier. So we can quickly determine which partition or RT-Trees

a vertex is in by comparing vertex id with the ranges of partitions. Thus, the reachability between two vertices can be determined by the relationship between two IDs. In the following, we will first introduce how to search pivotal vertex, partitioning a graph into traversal trees, relabeling. Finally, we will build interval index and store it in an efficient manner.

3.1 Searching Pivotal Vertices

Traversal of large graphs will cause huge cost. Consider the case to answer reachability queries, we always have to frequently access the neighborhood of the recent visited vertices, or the paths where the recent visited vertices are. Therefore, to improve access locality and reduce search space, we need place those neighboring vertices in paths into a same partition. Each partition can be an RT-Tree. However, there may exist overlapping subgraphs, namely pivotal trees among RT-Trees of a graph. These pivotal trees may also overlap with each other. Since the intersection of traversal trees is not empty, it is difficult to index traversal trees and answer reachability. So we need to further partition RT-Trees into disjoint traversal trees.

Algorithm 1. Searching Pivotal Vertex

Require: Graph G, root set $\mathcal{S}$
1: Pivotal vertex set $\mathcal{P} \leftarrow \emptyset$;
2: **for** each $root \in \mathcal{S}$ **do**
3: **if** *root* is unvisited **then**
4: $enqueue(root)$;
5: **while** *queue* is not empty **do**
6: $j = dequeue()$;
7: **if** j is unvisited **then**
8: $label(j) \leftarrow root$;
9: **for** each direct successor u of j **do**
10: $enqueue(u)$,$deg^-(u) \leftarrow deg^-(u) - 1$;
11: **else if** $label(j) \neq root$ and $deg^-(u) == 0$ **then**
12: $\mathcal{P} \leftarrow \mathcal{P} \cup \{j\}$;

To partition RT-Trees further into disjoint parts, we first find out pivotal vertices using BFS. The traversal starts from the root of each RT-Tree. Each time when a vertex is visited, the in-degrees of its direct successors should be reduced by one. Assume we visit vertex j, there are three situations depending on whether vertex j was visited before (Algorithm 1). Consider the situation in which vertex j has not been visited yet, we label it with the vertex ID of the current tree's root. If j has been visited before and its label is not equal to the current root ID, j belongs to a pivotal tree. If the in-degree of j is zero, j is a pivotal vertex. If j has been visited yet, and its label is equal to current root ID, it indicates that the RT-Tree has cross edges (e.g., edge: $v_{27} \rightarrow v_{98}$). For example, in Fig. 1(a), all the pivotal vertices are v_7, v_9.

(a) a graph

(b) Graph after partitioning and relabeling

Fig. 1. Tree-based partitioning and relabeling

3.2 Traversal Tree-Based Partitioning

After finding all the pivotal vertices, we then partition the graph using traversal tree-based partitioning (Algorithm 2). If we can reach a vertex u (e.g. v_{30} in Fig. 1(a)) via pivotal vertex s_1 (e.g. v_9 in Fig.1(a)), or a vertex s_2 (e.g.,v_{99} in Fig.1(a)), the topological traversal process will assign u to the partition s_1 resides due to its in-degree restriction on visiting order. However, DFS or BFS can not do it correctly. For instance, in BFS, if the traverse from s_2 to u is earlier than from s_1 to u, u will be assigned to the partition s_2 resides, otherwise, the partition s_1 resides. Actually, since u is a successor of a pivotal vertex, it should be as a part of the pivotal tree to be partitioned.

When we reach an unvisited vertex u (not a pivotal vertex) from a root, we assign it to the partition where this root is, and delete u and its direct out-edges from the graph. Otherwise, if u is a pivotal vertex, the RT-Trees where u appears will be recorded in a data structure (e.g. Table 1). Thus, we can easily find the RT-Trees where a pivotal vertex appears. When we finish the topological traversal from one root, we can get a partition. The above steps are performed for each root of RT-Trees repeatedly until all vertices are assigned.

Consider the graph in Fig. 1(a) again, initially, partition $part_1 = \{V_1, E_1\}$, $V_1 = E_1 = \emptyset$. The traversal begins at root vertex v_1 and v_1 is inserted into the queue. Now, $V_1 = V_1 \cup \{v_1\} = \{v_1\}$. If *queue* is not empty, execute *dequeue* (v_1) and delete its out-edges $(v_1, v_5), (v_1, v_7)$ from the graph. Among v_1's direct successors $\{v_5, v_7\}$, v_5 is unvisited and not a pivotal vertex, and its current $deg^-(v_5)$ is zero. So we visit v_5, $enqueue(v_5)$ and $V_1 = \{v_1\} \cup \{v_5\} = \{v_1, v_5\}$. Then, we reach v_7. Since v_7 is a pivotal vertex, we just record (v_1, v_7) and (v_3, v_7), where v_1,v_3 are the roots of RT-Trees where v_7 resides. Now $queue = \{v_5\}$. The above steps will be executed until the queue is empty. Finally, we can get three partitions $part_1$, $part_2$, and $part_3$.

After traversing all RT-Trees, we perform the same steps as the above repeatedly to get pivotal trees from pivotal vertices. The partitioning result is indicated using dotted lines in Fig. 1(b) (IDs in Fig. 1(b) are assigned using Algorithm 2).

Algorithm 2. Tree-Based Partitioning

Require: Graph G=(V,E), root set $\mathcal{S}$, pivotal vertex set $\mathcal{P}$
1: $i \leftarrow 1$, $\mathcal{D} = (\emptyset, \emptyset)$;
2: **for** each unvisited vertex $root \in \mathcal{S}$ **do**
3: $enqueue(root)$;
4: $part_i = (V_i, E_i)$, $E_i \leftarrow \emptyset$, $V_i \leftarrow \{root\}$;
5: **while** *queue* is not empty **do**
6: $j = dequeue()$;
7: **for** each successor u of j **do**
8: $deg^-(u) \leftarrow deg^-(u) - 1$;
9: **if** $u \in \mathcal{P}$ **then**
10: **for** each $t \in Root(u)$ **do**
11: $\mathcal{D} \leftarrow \mathcal{D} \cup (t, u)$;
12: **else**
13: **if** u is unvisited and $deg^-(u) == 0$ **then**
14: $enqueue(u)$; $V_i \leftarrow V_i \cup \{u\}$; set u as visited;
15: **for** each edge $(u, v) \in E$ and $u, v \in V_i$ **do**
16: $E_i \leftarrow E_i \cup (u, v)$;
17: $i \leftarrow i + 1$;
18: **for** each vertex $j \in \mathcal{P}$ **do**
19: execute step 3-17;

Our traversal tree-based partitioning scheme has three salient advantages. First, since neighboring vertices and paths are in same partitions, it can improve memory and disk access locality. Second, it prunes unnecessary search space, and ensure a quick index lookup. Third, smaller tree-partitions facilitate parallel processing.

3.3 Relabeling Trees

Each vertex of a graph has an ID. Since the initial vertex IDs are randomly assigned and the vertices are distributed over the graph, it is difficult to build the index on vertices. To improve locality for search and facilitate index construction, vertices belonging to the same partition are assigned with continuous IDs. So each partition can be indicated by a range (interval). We can efficiently determine which partitions a vertex is in merely using intervals, and further determine the corresponding RT-Trees it resides. Then, the reachability of two vertices can be partially determined by checking whether their IDs fall in a range.

Algorithm 3 shows the relabeling process using DFS. Global variable *idcount* indicate the starting ID assigned to every partition. In each partition, atomic *__sync_fetch_and_add* operation is used to seize the interval of IDs first. Then vertices in each partition are assigned ids according to theirs visiting order. For example, $V_1 = \{v_1, v_5, v_{27}, v_{98}\}$ of $part_1$ is relabeled as $\{v_1, v_2, v_3, v_4\}$ (Fig. 1(b)).

Distribution of pivotal vertices has been recorded during partitioning. So we can easily search for their corresponding RT-Trees to merge partitions. For example, Table 1 records distribution of pivotal vertices in Fig. 1, $part_4$ with old

Algorithm 3. Relabeling Traversal Trees

Require: Partition set $\mathcal{P}$, vertex size set $\mathcal{C}$ of partitions, global variable *idcount*

1: **parfor** each $part_i = (V_i,\ E_i) \in \mathcal{P}$ **do**
2: new_id $\leftarrow$ $\_\_sync\_fetch\_and\_add$(&idcount, $\mathcal{C}_i$);
3: **for** each vertex $v \in V_i$ in DFS order **do**
4: old_id(v) $\leftarrow$ new_id;
5: new_id $\leftarrow$ new_id+1;

root v_7 resides in two RT-Trees with old root v_1 and v_3 (v_1 and v_5 are their new root IDs). So $part_4$ should be merged into these two RT-Trees. Table 2 is the collection of partitions described by intervals for each RT-Tree (after relabeling). For example, $[5, 6], [11, 12]$ and $[13, 15]$ compose a RT-Tree with new root v_5.

Table 1. Distribution of pivotal vertices

old root ID	pivotal vertex
v_1	v_7
v_3	v_7
v_1	v_9
v_3	v_9
v_4	v_9

Table 2. Partition-based RT-Tree

new root ID	partition
v_1	[1, 4], [11, 12], [13, 15]
v_5	[5, 6], [11, 12], [13, 15]
v_7	[7, 10], [13, 15]

3.4 Interval-Index Construction

After partitioning a large graph into a set of tree-partitions, we may get a large number of tree partitions. It is important to lookup a tree in order to answer reachability. Since vertices of each partition are assigned the continuous IDs, each partition can be indicated by a minimal id and a maximal id. The minimal id of a partition is a root of traversal tree. Thus, we can quickly locate a traversal trees where a vertex is. It greatly reduces the search space. We will construct the graph of traversal tree. Specifically, we consider the partition with continuous IDs as a vertex in the graph of traversal trees. If two partitions are in the same RT-Tree, we add an edge between them into the graph of traversal trees. The traversal tree graph of Fig. 1(b) is shown in Fig. 2. Now, only direct edges between two vertices should be taken into consideration. That is, there exists no transitive relation in a traversal tree graph. Vertices v_3, v_5 reside in different RT-Trees, even though the two partitions denoted by interval [1, 4] and [5, 6] are indirectly connected by partition [11, 12].

We build an adjacency list index for the graph of traversal trees. Table 3 shows the adjacency list index of Fig. 2. Because it indicates a range, we call it interval index. We can determine the reachability of vertex pairs by checking the adjacency list index. The reachability of any two vertices can be determined

using the interval index. Concretely, we first search the partition where the first vertex resides and locate the corresponding row in the adjacency list. All the neighboring partitions of this partition can be found in the list. Second, we check whether the second vertex resides in the neighboring partitions. By this way, we can quickly answer the reachability of vertex pairs.

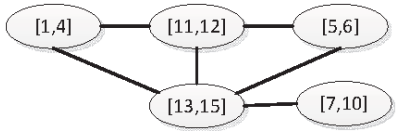

Fig. 2. The graph of traversal trees

Table 3. Adjacency list index

vertex	list of edges
[1, 4]	[11, 12],[13, 15]
[5, 6]	[11, 12], [13, 15]
[7, 10]	[13, 15]
[11, 12]	[1, 4], [5, 6],[13 15]
[13, 15]	[1, 4],[5, 6], [7, 10],[11, 12]

With interval index, the reachability of vertex pairs can be determined quickly. It requires two binary searches. First, we search *offset_index* for the offset address of the partition where the first vertex resides and locate the corresponding row in the adjacency list. And all the neighboring partitions of this partition can be found in the list. Second, we search whether the second vertex resides in these neighboring partitions to answer the reachability.

3.5 Delta Compression and Integer Encoding

Relabeling scheme makes differences among IDs residing in the same partition smaller to further improves locality. We store the interval-based adjacency list using byte-level delta compression scheme [8], in which the minimum number of bytes are used to label the delta value between IDs. The relabeling scheme makes the difference between IDs or intervals smaller. For one row of the adjacency list like $[id_1, id_2] : [s_1, e_1], [s_2, e_2]...$, we store it as $id_1, id_2 - id_1, s_1, e_1 - s_1, s_2 - e_1, e_2 - s_2, ...$ We then append the first interval $[id_1, id_2]$ and the offset of its corresponding row into a file named *offset_index* in order to efficiently locate its row in subsequent query processing.

The ID (including the delta value) is an integer. The size for storing an integer is typically 4 bytes. Not all integers require the whole word space to store them. For example, in small graphs like *linkedmdb*, the number of vertices is about 1 million. The storage for each id is at most 3 bytes. So it is wasteful to store them with a larger number of bytes when a smaller number of bytes are sufficient. However, for bigger graph, 4 bytes may be not enough to store an ID. Thus, we use the flexible approach - variable integer encoding [8].

3.6 Complexity Analysis

Index construction involves four steps: we need search pivotal vertices first, then partition the graph using traversal trees and relabel vertices. Finally, we build

interval index. The complexity of Algorithm 1 is $O(|E|)$, because both of them enumerate all vertices once. The complexity of Algorithm 2 is $O(|V| + |E|)$, because each vertex is visited when every edge is visited. The index construction can be executed when the relabeling process finishes (Algorithm 3). The complexity of the last two steps is $O(|V|)$. Thus, its final complexity is $O(|V| + |E|)$.

Assume vertex pairs (u, v), our approach first locate the partition where u appears. The way is to perform binary search in the storage block in which u's partition is stored. Thus, the complexity depends on the average number of partitions in a storage block. In our approach, the block size is 4KB. Thus, there are not many partitions stored in a block. If u, v belong to the same partition, then v is reachable from u. If u, v are not in the same partition, after determining the location of a partition, we binary search its adjacent list to check whether v appears. If v is in the list, then v is reachable from u. Otherwise, they are not reachable. Thus, our approach takes time $O(log(\text{block\_size}) + log(average\_len))$, where $average\_len$ is the average length of adjacent lists of partitions. $average\_len$ is less than the average depth of rooted traversal trees.

4 Experimental Evaluation

Since Veloso et al. [7] showed that Feline outperforms the recently systems, such as GRAIL [4], we choose Feline as the competitor. Both Feline and our method are complied using g++. We run all experiments on a server with an Intel 2.13GHz CPU, 48GB memory, and CentOS 6.5 (2.6.32 kernel).

4.1 Performance on Real Graphs

Real Data Sets. We use six real-world data sets from a wide spectrum of domains: *linkedmdb*[1], *de wiktionary*[2], *dblp*[3], *pagelinks_en*[4], *yago*[3] and *wikidata*[2]. These real graphs vary both in size and in average degree (i.e.,d_{avg}). The characteristics of the data sets are shown in Table 4.

Table 4. Characteristics of real data sets

Data sets	linkedmdb	de wiktionary	dblp	pagelinks_en	yago	wikidata
vertices	1,404,454	3,499,638	30,812,730	18,268,992	101,722,334	164,223,339
edges	3,087,311	8,518,892	67,840,417	136,591,822	205,638,803	377,173,710
d_{avg}	2.20	2.43	2.20	7.48	2.02	2.30

Storage. Table 5 reports the results. Interval-Index approach requires less storage than Feline does. The reason is that Interval-Index approach is based on

[1] http://queens.db.toronto.edu/~oktie/linkedmdb/
[2] http://www.rdfhdt.org/datasets
[3] http://datahub.io/zh_CN/dataset
[4] http://data.dws.informatik.uni-mannheim.de/dbpedia

local partitioning and byte-level delta compression. Moreover, relabeling makes the IDs residing in the same partition continuous, significantly improving compactness of the index.

Table 5. Storage for real data sets (in MB)

Data sets	linkedmdb	de wiktionary	dblp	pagelinks_en	yago	wikidata
Interval-Index	108.6	108.0	308.0	2798.2	3133.5	10250.2
Feline	153.3	182.3	524.6	4831.0	4690.0	13230.0
Reduction	40.8%	41.3%	42.1%	33.2%	22.5%	44.3%

Query Answering Time. We randomly select 1,000k pairs of vertices for each data set. Table 6 reports the total time taken to run queries in real graphs. The experimental results are the average values of 5 runs. Since Feline uses a memory-based access mechanism, its query time includes the loading time of the graph and the time for processing 1,000k pairs of vertices. On the contrary, Interval-Index's access mechanism is based on external memory operation, implemented by *mmap virtual storage* technique. So it only needs to load the required data into memory. More importantly, its local partitioning holds neighbors together, so the search space is well pruned. Continuous IDs in each partition further reduce query time. The results clearly show that Interval-Index outperforms Feline in all data sets.

Table 6. Query time for real data sets (in second)

Data sets	linkedmdb	de wiktionary	dblp	pagelinks_en	yago	wikidata
Interval-Index	139.43	69.65	532.87	132.32	926.73	22037.29
Feline	181.82	86.73	685.91	438.25	1267.26	32159.30

4.2 Scalability

We also perform experiments on six synthetic random graphs with different size generated using LUBM data generator [1]. So we can evaluate the scalability of our approach. The characteristics of six data sets are shown in Table 7.

Table 7. Characteristics of synthetic data sets

Data sets	LUBM10M	LUBM50M	LUBM100M	LUBM200M	LUBM300M	LUBM500M
vertices	3,303,724	16,349,317	32,905,170	65,764,621	98,640,459	164,416,780
edges	13,409,395	66,751,196	133,613,894	267,027,610	400,512,826	667,592,614

Fig. 3 reports the results on synthetic data sets. The storage of Interval-Index increases more slowly than Feline does as the graph sizes grow. Since Feline needs to keep the primitive graph in memory, thus larger storage restricts

its scalability. The right part of Fig. 3 shows that our approach is again faster than Feline. As the number of edges increases, the query time of Interval-Index increases slowly, while the time of Feline increases more quickly. The reason is that Interval-Index's partition-based index sharply reduces search space to ensure efficient query performance. Its external memory-based access mechanism further ensures robust performance against the growth in graph size. Since it only needs to load the required data into memory, the graph size does not have much effect on IO and memory read/write. However, Feline requires much time to load all data into memory for computation. Thus, the performance of Feline is not better than our approach.

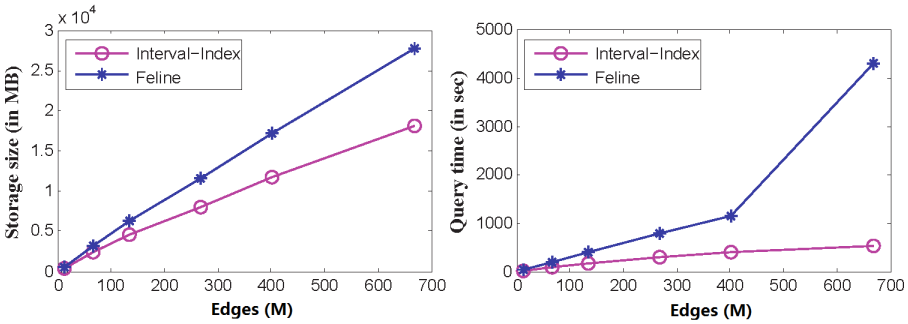

Fig. 3. Performance on varying edges ($M = 10^6$)

5 Related Work

Reachability querying is a fundamental graph operation with numerous applications both in research and in industry. Due to the emergence of large graph-structured data sets, reachaility queries have attracted enormous attention currently. Although many efforts have been devoted to it, it is still a challenge whether we can do faster and more scalable to answer reachability queries over even larger graphs with minimum cost (time and/or space).

Wei et al. [10] classifies existing approaches into two categories: Label-Only and Label+G. The label-only methods only utilize the labels to answer reachability queries, e.g., 3-Hop [9] and TF-Label [6]. Some of these methods [9] compress TC (transitive closure) to reduce index size. But majority of them are still space-consuming, and thus they are not so promising with regard to scalability on large graphs. SCARAB [3] is recently proposed as a general framework to represent a reachability backbone. It is used to further improve the scalability of existing methods. Since the backbone is much smaller than the graph itself, querying can generally be faster. However, if the size of backbone remains too large, these methods probably do not work [3,6]. TF-Label [6] is constructed based on a topological folding structure that recursively folds a target graph into half to restrict the size of label. The Label+G methods will answer reachability queries

only by label if possible, otherwise DFS will be required. GRAIL [4], Ferrari [5] and Feline [7] are classified to this category. GRAIL is a reachability index formed by multiple intervals obtained by the traditional min-post strategy. Randomized interval labeling is applied to it. Ferrari employs a selective interval set compression and a topological ordering to prune search space.

6 Conclusions

In this paper, we propose a scalable and fast graph indexing scheme, Interval-Index to answer reachability queries in large graphs. The Interval-Index approach, converts a large graph into a collection of fewer tree-partitions. The reachability of any two vertices can be determined effectively and accurately using the interval index. With extensive experiments, we demonstrate that Interval-Index outperforms Feline, one of the recent best systems in storage and query answering. Furthermore, Interval-Index is scalable proven to be not only efficient, but also robust against the increase in graph size.

Acknowledgments. The research is supported by National High Technology Research and Development Program of China (863 Program) under grant No.2012AA011003.

References

1. LUBM. http://swat.cse.lehigh.edu/projects/lubm/
2. http://www.facebook.com/press/info.php?statistcs
3. Jin, R., Ruan, N., Dey, S., Yu, J.X.: SCARAB: scaling reachability computation on large graphs. In: Proc. of SIGMOD 2012, pp. 169–180 (2012)
4. Yildirim, H., Chaoji, V., Zaki, M.J.: GRAIL: Scalable reachability index for large graphs. PVLDB **3**(1), 276–284 (2010)
5. Seufert, S., Anand, A., Bedathur, S.J., Weikum, G.: FERRARI: flexible and efficient reachability range assignment for graph indexing. In: Proc. of ICDE 2013, pp. 1009–1020 (2013)
6. Cheng, J., Huang, S.L., Wu, H.H., Fu, A.W.-C.: TF-label: a topological-folding labeling scheme for reachability querying in a large graph. In: Proc. of SIGMOD 2013, pp. 193–204 (2013)
7. Veloso, R., Cerf, L., Junior, W., Zaki, M.: Reachability queries in very large graphs: a fast refined online search approach. In: Proc. of EDBT 2014, pp. 511–522 (2014)
8. Yuan, P., Liu, P., Wu, B., Liu, L., Jin, H., Zhang, W.: TripleBit: a fast and compact system for large scale RDF data. PVLDB **6**(7), 517–528 (2013)
9. Jin, R., Xiang, Y., Ruan, N., Fuhry, D.: 3-HOP: a high-compression indexing scheme for reachability query. In: Proc. of SIGMOD 1999, pp. 813–826 (2009)
10. Wei, H., Yu, J.X., Lu, C., Jin, R.M.: Reachability querying: An independent permutation labeling approach. PVLDB **7**(12), 1191–1202 (2014)
11. Jin, R., Ruan, N., Xiang, Y., Wang, H.: Path-tree: An efficient reachability indexing scheme for large directed graphs. TODS **36**(1), 7 (2011)
12. Schaik, S.J., Moor, O.D.: A memory efficient reachability data structure through bit vector compression. In: Proc. of SIGMOD 2011, pp. 913–924 (2011)