A Resilient Index Graph for Querying Large Biological Scientific Data

Liang Li*, Zhihong Shen*, Jianhui Li*, Dongjiang Liu*, Huajin Wang*, Lipeng Wang†, Qinglan Sun†
*Computer Network Information Center, Chinese Academy of Sciences, Beijing, China
†Institute of Microbiology, Chinese Academy of Sciences, Beijing, China
Email: liliang@cnic.cn, bluejoe@cnic.cn, lijh@cnic.cn, ldongjiang@cnic.cn, wanghj@cnic.cn, wanglp@im.ac.cn, sunql@im.ac.cn

***Abstract*—The biological scientific linked data is large graph that contains billions of triples representing links between massive microorganisms. Now it is challenged by the growing graph size and the costly queries that require massive traversals. This work designs a resilient index graph to model the query pattern. According to given query pattern, it indexes graph traversals between the starting and objective vertices and represents them as index edges. Through visiting index edges, the query can be completed in one hop without repeatedly traversing the graph. It can bound the query to limited traversals and thus could response in real time. Moreover, the index graph can be constructed using BSP computing model with constant rounds. We developed a prototype system based on Titan, and experimental results showed that the index graph can complete complex biological benchmark queries within 400 milliseconds on average.**

***Keywords*—Linked Data, Graph Index, Query Pattern, Big Graph.**

I. Introduction

Nowadays, many biological scientific datasets are large graphs that contain billions of nodes and edges[1], and the graph databases have become the first choice to address query problems of large biological datasets[2], [3], [4], [5]. One typical example is the WDCM that is a large microorganism graph and aims to provide integrated information for microbial resource centers and microbiologists all over the world[1]. It provides metadata information on 708 culture collections from 72 countries and regions. Currently, WDCM includes $>$368 000 strains from 103 culture collections in 43 countries and regions and provides free access to all these graph data at www.wdcm.org.

With the continually growing of the biological datasets, it is nearly impossible to use only a single machine to manage graph data and satisfy the real time access requirements. Current WDCM graph has more than 24 millions of vertices managed in a single-machine Virtuoso platform, and in the close future this size might be scaled up to 1 billion or more. To understand this trend, consider the web and social network data that have been explosively increased recent years. In 2000 the web graph had only 2.1 billion vertices and 15 billion edges[6]; now, nearly all major search and social net companies can support a web graph of 1 trillion vertices or more. Another bioinformatics example, work[5] attempts to solve the genome assembly problem by constructing the de Brujin graph that contains as many as 4^k vertices where k is at least 20. Distributed large graphs which serve as important data structures to manage billions of vertices[7], [8], should be the suitable approach to address the large biological dataset query problem.

Although a lot of indices efforts have been devoted to efficient graph analysis, few existing methods are designed for multi-hop real-time querying of very large graphs. To understand the challenge, consider the super-linear space and/or super-linear construction time most state-of-the-art works required to build the indices. For example, the R-Join approach[9] for subgraph matching is based on the 2-hop index. The time complexity of building such an index is O(n^4), where n is the number of vertices. In large graphs, the value of n is on the scale of 1 billion (10^9) and any super-linear approach will become unrealistic[7]. These features make it difficult to use big data systems such as MapReduce[10], BigTable[11] to obtain scale-out capability.

Consider another example, a graph database may contain too many traversal paths if the graph are large and diverse. Yan et. al randomly extracted $10,000$ graphs from the antiviral screening database and found out that there are totally around 100, 000 paths with length up to 10, but most of them are redundant[12]. Therefore, it is inefficient to index all the multi-hop traversals in the graph. In this paper, we propose the index graph that just indexes limited parts of the graph and can be constructed in limited BSP computation rounds.

There are also many database systems that relies on index to optimize the graph traversal or subgraph matching to support the real-time query of big graph, i.e., gStore[13], Titan, Virtuoso, Blazegraph. gStore firstly uses the graph matching to complete the graph query. Titan in the other side uses the Bigtable[11] systems such as HBase or Cassandra to manage the large graph, and relies on the external indices such as Elasticsearch or Solr to speedup the graph query. These indices focus on the query of rows or properties and perform well in count and sort operations. The systems that incorporate them perform well for one-hop query, but not for multi-hop queries that require massive accesses. However, in query processing of large biological datasets graph, the most time-consuming parts are the multi-hop graph traversals and/or the subgraph matching.

The resilient index graph proposed in this work is used to address these problems. It uses the graph itself to store and manage the index space and only covers limited parts of the biological graph to balance the query processing and the complexity. For graph query, it effectively improves the real-time query performance and reduces the number of hops that required in the original query. For the index data structures, it indexes the traversal paths rather than the vertex, and can be constructed in parallel using BSP computing model. In this work, we construct the index graph for WDCM biological datasets and obtain impressive real-time query performance compared to the well-known Virtuoso implementation. Besides its potential in improving the query performance of multi-machine graph databases, it can support optimization of single-machine graph database systems such as Titan-BDB as well.

This work is organized as follows. Section II discusses the backgrounds of the graph database, the graph query approach, and graph indices. Section III analyzes the performance problems of biological graph queries and proposes the index graph based optimization. Section IV constructs the index graph for these graphs to optimize the multi-hop queries. Section V uses biological datasets to verify the effectiveness of the index graph and discusses difference between graph index and existing indices. In Section VI we conclude this work.

II. Background

In this section, we discuss the background of graph traversals and index structures in distributed database systems, and analyze the features of graph indices that support very large biological scientific datasets.

A. Graph Traversal and Gremlin

Graph traversal refers to the process of visiting (checking and/or updating) vertices in a graph. Graph traversal is an effective approach for querying large graphs that contain billions of vertices. It can limit the traversals only to interested vertices in large graphs without incurring much computation.

Gremlin[14] is a graph traversal language and virtual machine developed by Apache TinkerPop. Titan is natively integrated with TinkerPop graph stack and is open sourced with the liberal Apache 2 license. It uses existed Bigtable storage backends such as HBase to store the graph that contains hundreds of billions of vertices and edges, and is optimized for graph traversal querying in multi-machine cluster. In this work, we use Titan to manage the large biological datasets and gremlin to realize the corresponding graph query.

B. Multi-hop Query & Graph Index

For many graph-related biological applications, the graph query problem can be described as follows: given a graph database G and a graph query q, find all the matching in which q is a subgraph[12]. Multi-hop query is costly because one has to not only access the whole graph database but also check subgraph isomorphism which is NP-complete. Clearly, it is necessary to build graph indices in order to help processing graph queries.

XML query is one such type of graph queries built to model multi-hop expressions. Various indexing methods [15], [16], [17] have been developed. However, consider the super-linear construction complexity[9] and the redundant edges[12] in the real graphs, the construction of the path index across the whole graph is both hard and inefficient.

The index graph proposed in this work shares similar ideas, but focuses on using the graph structure to design the indices and scale up to the large biological scientific datasets.

TABLE I
WDCM Query Classification

Hops	Queries
one-hop	Q1, Q2, Q3, Q9, Q10, Q11, Q12, Q13, Q14, Q15, Q18, Q19, Q20
two-hop	Q4, Q5, Q6, Q7, Q8, Q16, Q17, Q21, Q22, Q23, Q24 ,Q25

III. Problem Analysis and Index Graph

In this section, we analyze the performance bottleneck when querying biological datasets and propose our index graph based approach. Further, we design algorithms for the index construction and updating.

A. Problems

In order to figure out the reasons that limit the query performance, we load a biological dataset WDCM into Virtuoso, Jena, and Titan, separately, and choose 26 standard queries from workloads. The classification of these queries is based on the length of traversal steps and the results are presented in Table I. In order to simplify the testing, Virtuoso, Jena, and Titan are run on a single machine.

As shown in Fig. 1, for all three query implementations, the one-hop query just uses 1 second or less, but the two-hop query uses more than 5 seconds, which cannot satisfy real-time query requirements. This is because two-hop queries traverse and/or match at least three different kinds of vertices that randomly distributed in the database. The more types of vertices, the more paths to traverse, which means the more times of the vertex accesses. Unluckily, in biological applications, these multi-hop queries take up nearly 50% of query workloads.

Note that for small datasets the queries can be optimized by loading all vertices into memory. When the data is too large to be loaded into memory, the traversals will suffer from the slow vertex access. In distributed systems, as the linked vertices and edges are distributed across machines, it is nearly impossible to complete all traversals and/or matching in milliseconds or seconds. This work proposes an index graph to address these kinds of queries.

B. The Index Graph

The index graph proposed in this work uses the graph itself to answer the multi-hop graph traversal problem. It indexes the multi-hop traversal paths via directly linking the starting vertices and the objective vertices through index edges. Thus, the query can be completed via traversing the index edges in one hop without indirectly traversing through internal vertices. Since most graph databases can complete the one-hop traversal in milliseconds, the index graph makes the existing graph

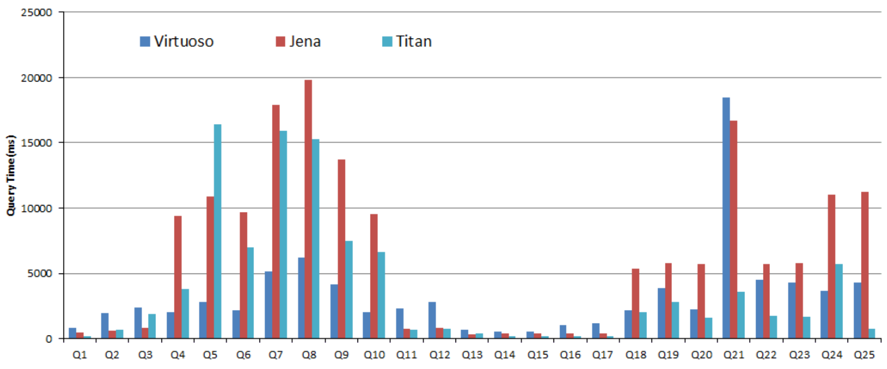


Fig. 1. The Query Time of Biological Datasets using Virtuoso, Jena, and Titan

database obtain the real-time query capability, without loading all data into memory.

The challenge of index graph is how to maintain the index graph space as small as possible while maintaining sufficient index capability. Before addressing these issues, we firstly define the index graph and model the main concepts of our approach. Assume the original graph $G(V, E)$, where V is the vertex set, E is the edge set, and each vertex and each edge have their own type, respectively. The index graph is defined as follows.

Definition 1: If $G_{index}(V_i, E_i)$ *is the index graph of* $G(V, E)$*, it holds that* $V_i \subset V$ *and if there is one path* $\langle v_1, v_2\rangle, \langle v_2, v_3\rangle, \cdots, \langle v_{n-1}, v_n\rangle$ *in origingal graph, then there is an edge* $\langle v_1, v_n\rangle$ *in* E_i *and vice versa.*

The index graph is used to maintain index edges for the original graph. The index edges here cover the starting and objective vertices that the traversal might search in the original graph. If there is an edge $\langle v_i, v_j\rangle \in E_i$, it holds that $v_i, v_j \in G$ is connected by a traversal path.

The two-hop index graph is a special case of the index graph. It only covers the vertices related to 2-hop traversals and reduces the traversals from the original two hops to one hop. A typical example about the two-hop index graph is presented in Fig. 2(b). It indexes the two-hop traversals that pass vertices with type c. Therefore, the $\langle b, e\rangle$ in the index graph represents $\langle b, c\rangle$ and $\langle c, e\rangle$ in the original graph. Generally, the 2-hop index graph G_{2i} space that covers the whole G is still too large as it has a larger number of edges but most of them are useless. Because users might just query traversals that pass vertices with fixed biological type and/or microorganism id.

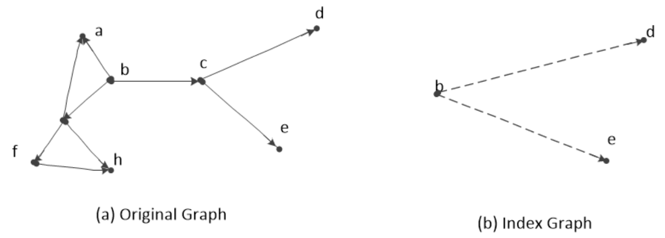


Fig. 2. Two-Step Index Graph.

Therefore, we give the type-fixed two-hop index graph. Before definition, we model the fixed type traversal pattern. The traversal $V_1{-}(xEdge){-}{>}V_T{-}(yEdge){-}{>}V_2$ represents all traversals that start from vertices of type V_1 and reach vertices of Type V_T through the *out* edge with type $xEdge$, and then reach vertices of Type V_2 through the *out* edge $yEdge$.

Definition 2: A type-fixed two-step index graph $G_{2i}(V_1, V_2, E, V_T)$ *is a two-step index graph that only indexes traversals* $V_1{-}(xEdge){-}{>}V_T{-}(yEdge){-}{>}V_2$.

C. Features of the Index Graph

As discussed in the previous section, index graph is related to the traversal pattern. Generally, there are many multi-hop traversal patterns in an application system. It might lead to large memory consumption if we build index graph for each of them. In this section, we present features to enable the maintaining of multiple index graphs in a single graph.

Now consider the G_{ij} that is the union of index graphs G_i and G_j of the same original graph G, and check whether it is a complete index graph.

Feature 1: For any index graph G_i, G_j *of* G, $G_{ij} = G_i \cup G_j$*, it holds that* G_{ij} *is an index graph of* G.

It is very easy to prove that the union operations can maintain the completeness of the index graph. We can store multiple index graphs together to improve space utilization.

Feature 2: For any index $G_i(V_i, E_i)$ *of* G, $G_s = G \cup G_i$*, it holds that for any* v_1, $v_2 \in V_i$*, if there is a path that satisfies the query pattern of* G_i*, there is an index edge* $\langle v_1, v_2\rangle \in E_i$ *and vice versa.*

Feature 2 makes it feasible to store the index graph and the original graph in a single graph. Although G_s is not a complete index graph, G_s fully inherits the index capability of G_i. This feature facilitates the management and the use of the index graph to help accelerating the graph querying.

With the filtering and the merging, we can maintain the index graph in a small size. Next, we will discuss the construction and the updating of the index graph based on these definitions and features.

In order to satisfy the WDCM two-hop query requirement, this work focuses on the two-hop index graph and designs the

Algorithm 1 Two-hop Index Graph Construction

```
Input: G(V, E), $V_T$, $xEdge$, $yEdge$
Description: $xEdge$ is the edge type from source vertices to $V_T$ vertices, and $yEdge$ is the edges from $V_T$ vertices to dest vertices
Output: index graph $G_i(V_i, E_i, V_T)$
1: for each v ∈ V do
2:     if v.outE($xEdge$).inV().type == $V_T$ then
3:         v.broadcast(v.outE($xEdge$).inV(), V.id())
4: bsp sync and comm.
5: for each v ∈ V($V_T$) do
6:     receive msgs
7:     for each msg ∈ msgs) do
8:         v.broadcast(v.outE($yEdge$).inV(), msg)
9: bsp sync and comm.
10: for each v ∈ V do
11:     receive msgs
12:     for each msg ∈ msgs) do
13:         addEdge(msg.vertex, v)
14: bsp sync and comm.
15: construct the index graph $G_i$ based on the added edges
16: return $G_i$
```

corresponding algorithms. Researchers can extend the index graph to optimize other types of multi-hop querying and processing, i.e., three-hop querying.

D. The Construction and Update Algorithm

This section presents the construction and update algorithms of the index graph. We can implement these algorithms on existed systems such as Spark to accelerate the construction and support the online data collecting and updating.

1) Index Graph Construction Algorithm: With conceptions and features discussed above, this section designs an index graph construction algorithm $Algorithm$ 1 to support the two-hop index graph construction using BSP parallel model. The BSP(Bulk Synchronous Parallel model)[18] abstracts the parallel processing as the iterative computation steps and synchronization steps, and is used in many batch based graph processing such as Pregel[19]. Here we use the BSP to design the index graph construction algorithm.

As shown in $Algorithm$ 1, the construction includes the following three rounds. In the first round, each vertex that has edges $xEdge$ to vertices of V_T broadcasts its vertex as msg to all linked V_T. In the second round, each V_T vertex receives msg from its $in\ xEdge$ edge and then redirects each msg to all $out\ yEdge$ edges. In the third round, for each msg vertex v_2 receives from the V_T vertex, add the corresponding index edge $\langle msg.vertex, v_2\rangle$. Then, based on generated index edges, we construct the index graph. Obviously, the complexity of this algorithm is bounded by the $xEdge$ and $yEdge$ edges of the V_T vertices.

Since $Algorithm$ 1 is developed based on BSP model, it can be easily parallelized in Hadoop or Spark. In order to reuse the space, we store the edges of the index graph in the original graph and specifically label them to distinguish from original edges. When some indices will not be used in the future, users can delete them and leave space for creating new indices.

Algorithm 2 Two-hop Index Graph Updating

```
Input: $G(V, E)$, $G_i(V_i, E_i, V_T)$, $e$
Description: $e$ is the edge that needs to be inserted in to $G$
Output: output result: graph index
1: $E_{set}$ = $null$
2: while 1 do
3:     add $e$ to $E$ and $E_{set}$
4:     if $E_{set}$.collect_time≥Max or $E_{set}$ is full then
5:         for each $e \in E_{set}$ do
6:             if e.inV ∈ $V_T$ then
7:                 for $v \in$ V & $\langle e.inV, v\rangle \in E$ do
8:                     add $\langle e.outV, v\rangle$ to $E_i$
9:             if e.outV ∈ $V_T$ then
10:                for $v \in$ V & $\langle v, e.outV\rangle \in E$ do
11:                    add $\langle v, e.inV\rangle$ to $E_i$
12:        reset $E_{set}$
13: return $G_i$
```

2) Index Graph Updating Algorithm: In order to support the online biological scientific data collecting and updating, we design an incremental algorithm $Algorithm$ 2 in this section. Incremental computing, is a software feature which, whenever a piece of data changes, attempts to save time by only recomputing those outputs which depend on the changed data[20], [21]. In this algorithm, we use the incremental computing to compute the indexes for the newly added data.

The algorithm is designed to implement in a periodic way. It continually receives the edges in E_{set} until the collecting time reach a bound, i.e, Max = 10s, then it checks the status of the original graph and the index graph and updates the index graph when really needed. After that, it resets the E_{set}, continues to receive edges, and so on. That the algorithm only monitors the adding of edges is because the adding of vertices will not change the status of the index graph.

Moreover, because of its periodic processing, it can be easily integrated into existed systems, i.e., Spark, and allows the data processed in a streaming way.

IV. The Type-Fixed Two-Step Index Graph for Biological Datasets

The WDCM is a popular data resource in the microorganism research. As shown in Fig. 3, it includes microorganisms such as taxonomy, protein, gene, genome, and the relations between these organisms. In this section, we discuss the graph traversal query example in detail that starts from taxonomy vertices to other vertices such as pathway, enzyme, or gene ontology, and show how the type-fixed two-hop index graph optimize the queries of the WDCM.

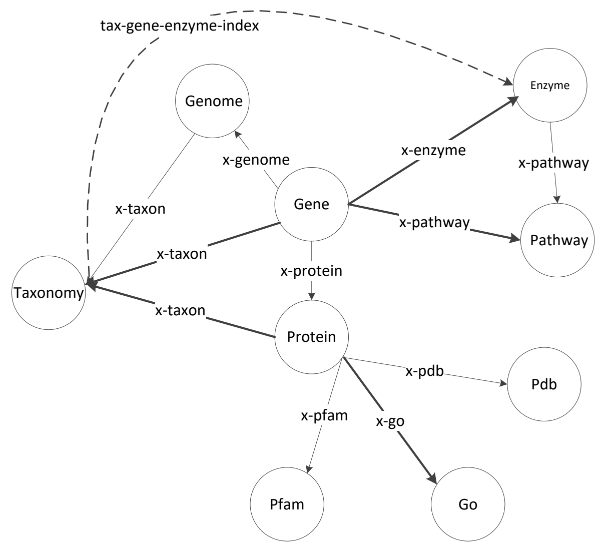


Fig. 3. The WDCM Datasets Graph

A. *The Model of the Index Graph*

This section first abstracts the query pattern of the WDCM biological datasets. For example, query the enzyme using the fixed taxonomy id. Through analyzing the datasets in Fig. 3, the access pattern can be defined as follows.

V_{taxon}<−(x-$taxon$)-V_{gene}-(x-$enzyme$)->V_{enzyme}.

The V_{taxon}, V_{gene}, and V_{enzyme} represent the vertices and the $<-(x\text{-}taxon)-$ and $-(x\text{-}enzyme)->$ represent edges that connected these vertices. The traversal that passes gene is starting from taxonomy to gene through the *in x-taxon* edges and then to enzyme through the *out x-enzyme* edges. Consider the query pattern of enzyme, we construct a two-hop fixed type index graph G_{enzyme}.

$G_{enzyme}(\{V_{taxon}, V_{enzyme}\}, E_{wdcm}, V_{gene})$.

As shown in Fig.3, the *tax-gene-enzyme-index* is the index edges of G_{enzyme}. Since the edges between the taxonomy and the gene can only be of *x-taxon* and the edges between the gene and the enzyme can only be of *x-enzyme*. Consider the definition of the index graph, if there is an *tax-gene-enzyme-index* edge in E_{enzyme}, then there must be a traversal path from taxonomy to enzyme via gene and vice versa.

Similarly, this work defines $G_{pathway}$ for the multi-hop pathway query pattern and G_{go} for the multi-hop gene ontology query pattern, respectively.

For pathway, the query pattern is represented as $V_{taxon}<-(x\text{-}taxon)-V_{gene}-(x\text{-}pathway)->V_{pathway}$. The index graph is $G_{pathway}(\{V_{taxon}, V_{pathway}\}, E_{wdcm}, V_{gene})$.

For gene ontology, the query pattern is represented as $V_{taxon}<-(x\text{-}taxon)-V_{protein}-(x\text{-}go)->V_{go}$. The index graph is $G_{go}(V_{taxon}, V_{go}, E_{wdcm}, V_{protein})$.

We run the index graph construction algorithm to generate these index graphs. As illustrated by $Feature$ 2, we store all these index graphs and the original WDCM datasets in the same graph space. Compared to the overall existed edges, the increased index edges are limited.

B. *The Optimization of Graph Traversal*

Based on the constructed index graph, this work takes the query of enzyme as an example to show how to optimize the graph traversal.

Here is a graph traversal query derived from WDCM. It starts from a taxonomy vertices, and then to gene vertices through the *x-taxon* edges, and then to the enzyme vertices through the *x-enzyme* edges. The optimization of the graph query based on the index graph G_{enzyme} is quite easy.

Without index graph, the query needs to traverse first through *x-taxon* edges and then through *x-enzyme* edges. Here is a gremlin traversal sample for enzyme count operations. Because the vertices might be randomly distributed across multiple machines, this traversal is time-consuming and will incur many communications.

$g.V().has(\text{“}vertexID\text{”}, \text{“}.../taxonomy/1270\text{”})$
$\quad .in(x\text{-}taxon).has(\text{“}type\text{”}, \text{“}GeneNode\text{”})$
$\quad .out(x\text{-}enzyme).dedup().count()$

With index graph, the query can directly reach the object enzyme vertices through traversing the *tax-gene-enzyme-index* index edge. We replace the two-hop graph traversal with the one-hop index traversal and thus reduce the amount of communications and the query time. The corresponding gremlin code is presented as follows.

$g.V().has(\text{“}vertexID\text{”}, \text{“}../taxonomy/1270\text{”})$
$\quad .out(tax\text{-}gene\text{-}enzyme\text{-}index).dedup().count()$

Since the one-hop traversal is efficient in most distributed graph databases, with the index graph proposed in this work, the graph query can be completed in real time without further index optimization and hardware enhancement.

V. Experimental Evaluation

This section chooses typical biological WDCM multi-hop queries to verify the effectiveness of the proposed index graph. We separately run these queries on both the single-machine system and the multi-machine system, and compare their results with that of Virtuoso.

A. *Experimental Environments*

As shown in Table II, the experimental platform is composed of five servers connected by infiniband. We allocate 4 machines for Hadoop and HBase, and 1 for Elasticsearch. Titan which runs on Hbase is used to store both the original datasets graph and the index graph. In order to simplify the design, we do not apply the explicit graph partition and rely on the Titan to distribute vertices to multiple machines. Although these systems are not new, the index graph proposed in this work makes these subsystems work in a coordinated way. If not specified specifically, following experiments are run on this platform.

The basic information about WDCM biological datasets is presented in Table III. It is a very large graph, its current Dataset I has 300 million triples and its future Dataset II has 3

TABLE II
EXPERIMENTS ENVIRONMENTS

Processors	E5-2603 v3 @1.60GHz x 12
Memory	32GB
Disk File System	XFS
Hadoop	1.2.1
Gremlin Query Engine	developed based on titan 1.0.0
Single-Node Graph Database	Titan with BerkeleyDB
Multi-Node Graph Base	Titan with hbase-0.98.0.23-hadoop1

TABLE III
WDCM DATASET STATISTICS

Name	WDCM
Author	WFCC-MIRCEN World Data Centre for Microorganisms
Discipline	Microorganisms
Type	RDF
Dataset I	300 millions Triples
Dataset II	3 Billion Triples

billion triples. It manages relations between micro organisms such as taxonomy, protein, gene, pathway, and genome and represents them as edges in graph. According to our analysis, there are many super vertices in this datasets, some even have more than 100 thousand edges.

This work chooses 18 two-hop queries, including information query and count query between taxonomy and other micro organisms such as enzyme, pathway, gene ontology, etc. We label these queries as enzyme1, enzyme2, enzyme3, pathway1, pathway2, pathway3, go1, go2, and go3, separately. Each query starts from the fixed taxonomy vertex and ends in the objective vertices. The query implementations using the index graph in this work are developed based on gremlin. The corresponding query patterns are defined in Section IV.

B. The Basic Info and Count Query

This section compares the index graph based traversal implementation and the original graph traversal implementation using Dataset I that includes 300 million triples and Dataset II that includes 3 billion triples. In order to strictly remove the affection of database cache, we restart the graph database Titan for each query implementation. The $original\ graph$ method refers to the original two-hop graph traversal based implementation, and the $index\ graph$ method refers to the proposed index graph based implementation.

We first run the queries on Dataset I that has 300 million triples and present the results in Fig. 4. Fig. 4 presents the query time for both the index graph based implementation and the original graph implementation. The figure shows that the graph index based implementation uses much less time than the original graph implementation. This is because the amount of traversals with index graph is reduced to a much lower level, mostly no more than 10% of the original queries. So for the enzyme and pathway query, the indexed implementation is nearly accelerated more than 10 times. For the x-go where the traversals are a little complex and cannot be fully covered by the index graph, the acceleration is not obvious, but we still obtain more than 3 times speedup for info query. Although we do not use Titan's sort-favor property index and not specifically optimize the index graph for the count operations, the count query time is still significantly reduced.

We further run the same queries on Dataset II that has 3 billion triples and obtain similar results. As shown in Fig. 5, the index graph can make the graph database system response the query in milliseconds mostly through decreasing the traversal hops.

This section further compares the query time for both the small dataset Dataset I and the large dataset Dataset II using the same query benchmark. The results are presented in Table IV and show that the query time is maintained constantly when the data size scales from 300 million to 3 billion. It illustrates the potential of the proposed index graph in tackling the continually growing biological datasets such as WDCM.

C. The Query Comparison with Virtuoso

In this section, we further compare the query implementations between Virtuoso and the proposed index graph using Dataset I. The $virtuoso$ method refers to the Virtuoso based query implementation and the $index\ graph$ method refers to the index graph based query implementation. The WDCM Virtuoso database is well tuned and runs on a server that has 64 GB memory. Because it is providing online services, it cannot be reconfigured to remove the affection of the cache. The index graph is run on the original platform and is restarted each time to remove the cache affection. We compare the query time of both the Virtuoso and the index graph implementation.

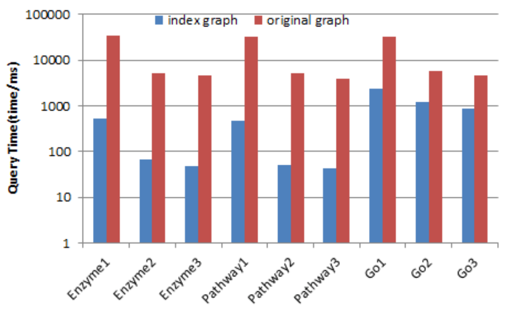
Fig. 6 shows that the time of graph index is mostly smaller than that of the Virtuoso implementation, with performance improvement ranging from 1.3x to 10x for most 18 queries. That the use of more time for go's count query is because it requires more traversals than that the index graph covered.

We further analyze the single-node implementations and the multi-node implementations of the index graph, and compare their query time with that of Virtuoso. The query is to find out the count of enzyme, pathway, and go vertices that related to fixed taxonomy. The single-machine index graph is run on Titan with BerkeleyDB. The multi-machine index graph is run on Titan with Hbase.

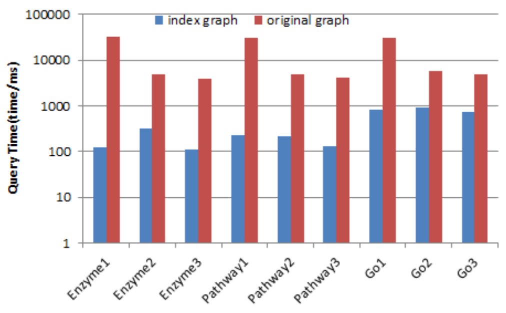
As shown in Fig.7, for both the count query of enzyme and pathway, the query time is nearly the same for both the single-machine and the multi-machine implementations, respectively. For go query, the Hbase based multi-machine implementation uses more time to complete than both BerkeleyDB and Virtuoso single-machine implementations. It shows that in one side the query of go has many traversals not covered by the traversal pattern employed; in the other side, the traversal of multi-machine based graph database is more expensive. In these cases, the index graph and its ideas are especially important for the real time query optimizations.

D. Index Maintainance and Update Overheads

This section evaluates the maintaining and updating overheads of the graph index using Dataset I and Dataset II.

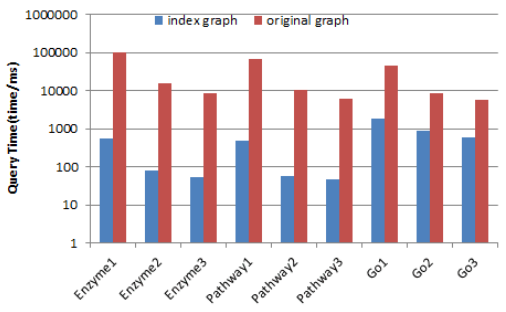


(a) count

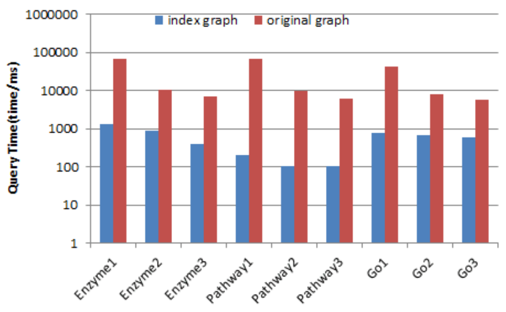


(b) info

Fig. 4. The Query Implementations using Dataset I



(a) count

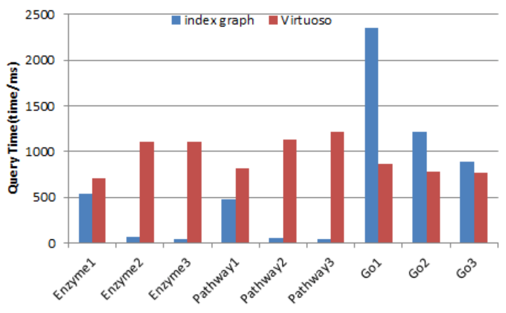


(b) info

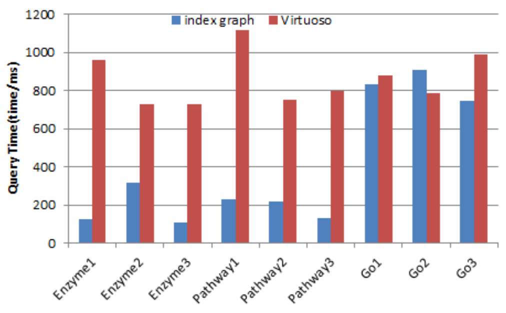
Fig. 5. The Query Implementations using Dataset II

TABLE IV
INDEX GRAPH BASED WDCM COUNT QUERY TIME(TIME/MS)

	Enzyme1	Enzyme2	Enzyme3	Pathway1	Pathway2	Pathway3	Go1	Go2	Go3
Dataset I	534	67	47	480	51	43	2355	1211	894
Dataset II	539	80	54	494	57	47	1790	877	608



(a) count



(b) info

Fig. 6. Index Graph vs Virtuoso on Dataset I

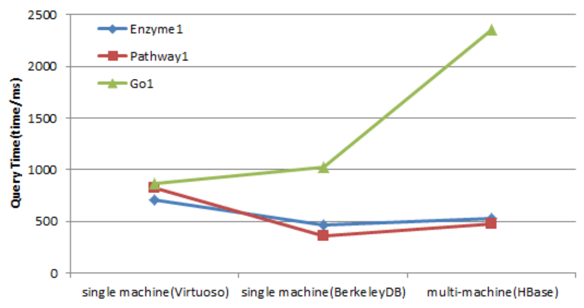


Fig. 7. The Query Time When Scale from Single Node to Multiple Nodes

TABLE V
THE STORAGE OVERHEAD - INDEX EDGES

	enzyme	pathway	go
Dataset I	86387	61882	497190
Dataset II	138280	61882	497190

The index maintainance overheads mainly come from the storing of the index edges. We count the generated index edges for the three query patterns discussed in Section IV and mark them as $enzyme$, $pathway$, and go, respectively. We present them in Table V. Table V shows that the number of index edges of the type-fixed index graph is comparatively small. Each index edge just uses one or several bytes, thus the overall storage overhead for query patterns of enzyme, pathways, or go is at megabyte level. It means the storing and the querying of these edges will be efficient.

We further evaluate update overheads of the index graph when inserting new edges. The benchmark data are 1,482 $enzyme$ edges, 17,267 $pathway$ edges, and 3,801 go edges generated based on WDCM datasets. The update algorithm is $Algorithm$ 2 that inserts edges at the granularity of E_{set} that contains up to 1000 edges. The performance bottleneck of $Algorithm$ 2 is the throughput of inserting edges and updating the index graph. We evaluate the average update time for inserting 1000 edges using single thread. The results are presented in Table VI. Table VI shows that the update time needed is not significantly increased for larger Dataset II. It needs at most 2 seconds to update with a E_{set} of new edges. Moreover, the update time depends on the query pattern. For simple query pattern(i.e.,enzyme), the overheads are small. But for complex query pattern(i.e, go), the overheads are of multiple times larger.

TABLE VI
THE INDEX UPDATING OVERHEADS PER EACH 1000 EDGES (TIME/SECONDS)

	enzyme	pathway	go
Dataset I	0.002	0.759	0.931
Dataset II	0.004	1.495	1.201

E. Further Discussion

These experimental results show that the proposed index graph can effectively reduce the biological multi-hop query time when the query's graph traversals are fully covered. The index graph is suitable for the multi-hop query of very large biological datasets that requires massive costly graph traversals, i.e.,WDCM, especially on distributed graph databases.

During the optimization of the graph traversals, we find that for some vertices that have more than 1,000,000 edges, the graph traversals that pass these edges are time-consuming and always be failed due to backend and network errors. To our surprise, the index graph addresses these problems by filtering or redirecting the traversals of these edges and thus obtains the much stable query implementation. Note that we do not manually partition the graph to optimize the edge-cut and vertex-cut. It seems that the index graph can lower the expectations of graph partition, which is hard to handle for non-professional users.

The size of the index graph is related to the volume of vertices that satisfy the query pattern. Complex queries always impose sophisticated type-fixed constraint, the size of the index graph is often small. But for general two-hop queries that impose nearly no constraint, the size of the index graph can be very large as any two-hop connected vertex could satisfy the pattern. The challenging points are how to build the index graphs in acceptable time and maintain these index graphs efficiently. For small graphs, the sequential implementation is enough. For very large graphs, we suggest incorporating existed BSP frameworks such as Hadoop-gremlin or Spark GraphX to generate the index. It may take several hours or more to complete the computation, which depends on the vertices volume and the query complexity. For the maintaining of multiple index graphs, we present $Feature$ 2 that shows how to merge the index graphs and the original graph efficiently.

The shortcoming of the type-fixed index graph is that users have to abstract the query pattern and submit it to the system for generating index graph. For the temporal queries that are not fully indexed, it might suffer from the massive graph traversals. This is because for very large graphs, the complexity of automatically generating the temporal multi-hop traversal indices is super-linear[7] and thus we instead choose to generate the fixed type indices defined by users.

Compared to the state of the art works that use memory to realize the real-time query of large graphs[22], our index graph is much lighter and should be easier to carried out on systems without large memory. Compared to the join approaches that use both subgraph matching and graph traversal[23], the index graph works at a higher level and uses index to reduce the graph traversals.

VI. CONCLUSION

In this paper, we propose an index graph to address performance problems of the multi-hop query of large biological datasets. It is used to index relations between the starting vertices and the object vertices. With the index graph, we can complete the query in one hop without repeatedly traversing all related vertices between them. Besides traversal benefits, the index graph has the potential to decrease the inter-node communication in distributed graph database and has better scale-out capability to tackle larger graph with billions of vertices. Experimental results demonstrate that our approach makes the biological multi-hop queries response in milliseconds or seconds, even without using graph database cache. We will further experiments on even larger graphs (i.e., trillions of vertices) to verify the query speedup and throughput bounds on massive data set. As another future work, some extra works will be also performed to simplify the index graph and support more graph databases (i.e., gStore, Neo4J).

Acknowledgment

This work was supported by the National Key Research and Development Plan under grant No. 2016YFB1000600 and The Strategic Priority Research Program of the Chinese Academy of Sciences under grant No. XDA06010307.

References

[1] L. Wu, Q. Sun, P. Desmeth, H. Sugawara, Z. Xu, K. Mccluskey, D. Smith, V. Alexander, N. Lima, and M. Ohkuma, "World data centre for microorganisms: an information infrastructure to explore and utilize preserved microbial strains worldwide," *Nucleic Acids Research*, vol. 45, no. D1, pp. D611–D618, 2017.

[2] C. T. Have and L. J. Jensen, "Bioinformatics editorial," *Bioinformatics*, vol. 29, no. 24, pp. 3107–3108, 2013.

[3] A. Lysenko, I. A. Roznovăţ, M. Saqi, A. Mazein, C. J. Rawlings, and C. Auffray, "Representing and querying disease networks using graph databases," *BioData mining*, vol. 9, no. 1, p. 23, 2016.

[4] V. Touré, A. Mazein, D. Waltemath, I. Balaur, M. Saqi, R. Henkel, J. Pellet, and C. Auffray, "Ston: exploring biological pathways using the sbgn standard and graph databases," *BMC bioinformatics*, vol. 17, no. 1, p. 494, 2016.

[5] D. R. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de bruijn graphs," vol. 18, pp. 821–829, 2008.

[6] M. Levene and A. Poulovassilis, "Web dynamics," *Computer Networks*, vol. 2, pp. 60–67, 2001.

[7] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 788–799, 2012.

[8] DataStax, "Titan documentation," http://s3.thinkaurelius.com/docs/titan/1.0.0/, 2017.

[9] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang, "Fast graph pattern matching," in *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ser. ICDE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 913–922.

[10] M. Vaidya, "Mapreduce: a flexible data processing tool," *Communications of the Acm*, vol. 53, no. 1, pp. 72–77, 2010.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable:a distributed storage system for structured data," *Acm Transactions on Computer Systems*, vol. 26, no. 2, pp. 1–26, 2008.

[12] X. Yan, P. S. Yu, and J. Han, "Graph indexing: a frequent structure-based approach," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004, pp. 335–346.

[13] L. Zou, J. Mo, L. Chen, M. T. Zsu, and D. Zhao, "gstore: Answering sparql queries via subgraph matching," *Proceedings of the Vldb Endowment*, vol. 4, no. 8, pp. 482–493, 2011.

[14] M. A. Rodriguez, "The gremlin graph traversal machine and language (invited talk)," in *Proceedings of the 15th Symposium on Database Programming Languages*. ACM, 2015, pp. 1–10.

[15] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes, "Exploiting local similarity for indexing paths in graph-structured data," in *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE, 2002, pp. 129–140.

[16] D. Shasha, J. T. Wang, and R. Giugno, "Algorithmics and applications of tree and graph searching," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2002, pp. 39–52.

[17] Q. Chen, A. Lim, and K. W. Ong, "D (k)-index: An adaptive structural summary for graph-structured data," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 134–144.

[18] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[20] M. Carlsson, "Monads for incremental computing," in *ACM SIGPLAN Notices*, vol. 37, no. 9. ACM, 2002, pp. 26–35.

[21] C. Demetrescu, I. Finocchi, and A. Ribichini, "Reactive imperative programming with dataflow constraints," in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 407–426.

[22] F. Checconi and F. Petrini, "Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 425–434.

[23] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li, "Efficient subgraph matching on billion node graphs," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 788–799, 2012.