Wind-Bell Index: Towards Ultra-Fast Edge Query for Graph Databases

Rui Qiu*, Yi Ming‡, Yisen Hong*, Haoyu Li* and Tong Yang*†

*School of Computer Science, and National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University, Beijing, China
‡Academy for Advanced Interdisciplinary Studies, Peking University
†Peng Cheng Laboratory, Shenzhen, China

***Abstract*—Graphs are good at presenting relational and structural information, making it powerful in the representation of various data. For the efficient storage and processing of graph-like data, graph databases have been rapidly developed and extensively studied. However, graph databases mostly use adjacency lists as their basic data structure (*e.g.*, Neo4j), which could result in poor performance of edge due to the skewed degree distribution of graphs.**

We design the Wind-Bell Index to address this problem. Wind-Bell Index is a memory-efficient index data structure, which can be attached to existing graph databases to speed up the edge. We have fully implemented our data structure in Neo4j, the most popular graph database today, and conduct theoretical and experimental analysis to evaluate the performance. Theoretical results prove the high query efficiency of our algorithm. And experimental results show that the average edge query speed is increased by hundreds of times compared with the original query interface of Neo4j. We believe that the excellent performance and scalability of Wind-Bell Index make it suitable for the application in a variety of graph databases.

I. Introduction

A. Background and Motivation

In recent years, graph databases have grown in importance due to its ability of storing and processing graph-like data. Considering the sparsity of graphs in practice, adjacency lists are used as the basic storage structure in most graph databases. However, adjacency lists invalidate directly searching for an edge in the graph given the start point and the end point. Instead, we can only conform the existence of an edge by traversing the linked list of either the start point or the end point, which reduces the efficiency of the basic edge query.

Moreover, the skewed degree distribution of graphs also significantly affects the query performance of an edge. It is well known that the node degrees of graphs often follow the power-law distribution [1], [2]. In other words, graphs are usually composed of a majority of low-degree nodes and a minority of high-degree nodes [3], [4]. The skewness of graphs is an obstruction in the optimization of graph databases. Existing graph databases (*e.g.*, Neo4j [5]) take a long time when querying the neighbors of high-degree nodes, and take a short time for low-degree nodes. However, high-degree nodes have a higher probability to be queried and updated, leading to poor average performance. Therefore, it is highly desired to design a solution which can improve the performance of edge query, especially for high-degree nodes.

B. Prior Art and Their Limitations

Great efforts have been made on the construction and optimization of graph databases in both academic and industrial fields. However, previous studies mostly focus on some specific problems, such as discovering similarity [6], [7], finding subgraphs [8], [9], generating shortest paths [10], [11], [12], and so on. By contrast, we attempt to accelerate the fundamental edge query, which consequently influences the performance of specific query problems that include visiting the edges. To the best of our knowledge, no existing work directly lays emphasis on the improvement of edge query in graph databases.

Among all the existing graph databases, the most widely used open-sourced system is Neo4j [5], acknowledged for its high performance, robustness, and flexibility. Neo4j uses the adjacency list rather than the adjacency matrix to store the whole graph, making it more suitable for sparse graphs. A query for a particular edge $\langle u, v \rangle$ requires traversing the entire list related to the start point u. Therefore, the speed of a edge query mainly depends on the degree of node u. Unfortunately, Neo4j does not address the problem of querying high degree nodes, leaving room for improvement.

C. Our Proposed Solution

To address this problem, we propose a new data structure, namely Wind-Bell Index, to build an index for the edges in graph databases. Wind-Bell Index is characterized by its high query speed and efficient memory consumption, which can speed up the edge query by hundreds of times and consume only 8 bytes for each edge.

Wind-Bell Index is a hybrid structure of adjacency matrix and adjacency list, combining the advantages of the two basic data structures. Wind-Bell Index consists of two parts, the ceiling matrix and the hanging linked lists. The ceiling matrix is a concentrated adjacency matrix, while the hanging linked lists are balanced adjacency lists. Given a sparse graph with m nodes and S edges, we assume that S is close to m due to its sparsity, and assume the degree of the nodes follow

Co-primary authors: Rui Qiu, Yi Ming, Yisen Hong. Corresponding author: Tong Yang (yangtongemail@gmail.com).

the power-law distribution. By setting the size of our ceiling matrix to $\sqrt{m} * \sqrt{m}$, we can guarantee that the expected length of each linked list is approximately 1 in the case of well load balance. To achieve well load balance, we propose to use the multi-hashing technique and leverage the kick strategy of Cuckoo hashing (see details in Section III). After realizing load balance, all the linked lists are very short, and therefore querying each edge only needs a few memory accesses (*e.g.*, 1 $\sim$ 4), thus speeding up the edge query. In contrast, a traditional adjacency matrix need the size of $m * m$ (much larger than $\sqrt{m} * \sqrt{m}$) while adjacency lists do not address the problem of high-degree nodes.

We have derived theoretical results for our Wind-Bell and proved that Wind-Bell outperforms the adjacency list when the graph follows the power-law distribution. We have fully implemented our data structure in the community version of Neo4j, and released the source code on Github without author information[13]. The experimental results show that the edge query performance of Neo4j is improved by 2 orders of magnitude on average by using the Wind-Bell.

D. Key Contributions

In this paper, we make the following 3 key contributions.

- We propose the Wind-Bell Index, a relational index structure which can be implemented in graph databases to accelerate edge query. To the best of our knowledge, this is the first index structure focusing on the optimization of edge query in graph databases.
- We conduct mathematical analysis to evaluate our algorithm. Theoretical results prove the effectiveness and efficiency of our algorithm.
- We have fully implemented our Wind-Bell into the graph database Neo4j, and provide a new interface for edge query. The experimental results show that the average edge query speed is increased by hundreds of times compared with the original version of Neo4j's Java API.

II. Background and Related Work

In this section, we briefly introduce the background of query optimization in graph databases and the related algorithms that inspired our work.

A. Query Optimization

A number of graph databases have been developed to satisfy the requirements of different applications, such as Neo4j [5], OrientDB [14], DEX [15], HyperGraphDB [16], and so on. Among all these graph databases, Neo4j has been the most popular one in recent years [17].

As for graph databases, query performance is an especially crucial evaluation indicator. Therefore, great effort has been made on the optimization of query performance [18]. Existing technologies include data distribution [19], query decomposition [20], incremental processing [21], and graph sketching [22], [23]. However, previous studies of query optimization mostly focus on specific application scenarios and optimizing any kind of query with one general technique still remains a challenge [18].

B. Related Algorithms

Our Wind-Bell Index is inspired by the following three algorithms.

d-random scheme [24]: Hash table is a commonly-used flexible data structure for performing fast look-ups. However, hash collisions may significantly undermine the actual performance in practice, and using multiple hash functions instead of single hash functions is a solution to this problem [25], [26]. The d-random scheme is one of the multi-hashing algorithms. By using d independent hash functions, we get d buckets for each item and hash the item to the least loaded one. Although searching for an item requires examining d buckets, the probability of long linked lists is significantly reduced.

Fast Hash Table [27]: Fast Hash Table regards Bloom filter [28] as a multi-hashing technique and presents an efficient hash table data structure by extending Bloom filters. In a Fast Hash Table, every item is stored in the shortest linked list among the d corresponding ones. In order to maintain this property, the items in the table need to be adjusted during each insert and delete operation.

Cuckoo hashing [29]: Cuckoo hashing is widely accepted as one significant recent advancement in the field of hash tables. If an item is to be inserted and the two candidate buckets are both occupied, it will kick out one of the existing item, and the kicked-out item will be reinserted. This kick operation will be repeated until every item has its own bucket. We modify the kick mechanism from Cuckoo hashing in the optimized version of our Wind-Bell Index.

III. The Wind-Bell Index

In this section, we first introduce the data structure and basic operations of our Wind-Bell Index, then propose an optimization strategy to further improve the performance.

A. Data Structure

First of all, we present the data structure of our Wind-Bell Index. We may compare the Wind-Bell Index to a real wind bell, which consists of a ceiling and multiple pendants. In our data structure, the ceiling of the index is a matrix, while the pendants represent linked lists. Each bucket in the ceiling matrix keeps a pointer of one linked list, just like the top of the wind bell with hanging pendants.

Ceiling Matrix: The ceiling of our Wind-Bell Index is an adjacency matrix, marked as A, and the buckets in the ceiling matrix are marked as $A[i][j]$. Every time there comes a new edge e with $\langle u, v \rangle$, the row where it should be is determined by the start point u, and the column is determined by the end point v. We use N hash functions to map e into N rows and N columns, then we combine them to obtain the N^2 candidate buckets for storing the element, marked as $A[i_1, i_2 ... i_N][j_1, j_2 ... j_N]$.

Each bucket in the matrix has two fields, a counter CNT and a pointer PTR. The pointer points to the hanging linked list attached to the bucket, while the counter records the length of the list. Despite the extra memory cost, maintaining the counter field[1] is essential in our algorithm because it enables us to balance the load of whole structure and accelerate the querying process.

Hanging Linked Lists: We have some designs for the linked list hanging from the ceiling matrix. Every element in the hanging linked list represents an edge in the graph, and there are five basic fields for every element, marked as $\langle u, v, nextRel, preRel, nextRec\rangle$. u and v stand for the start point and end point of the edge, $nextRel$ points to the next element in the hanging linked list, and $preRel$ points to the previous element. $nextRec$ is not used in this section, it will be introduced in the Section V later. By making the hanging linked list a doubly linked list, we can better support the flexible operations of graph databases.

In addition to the five basic fields, extra fields can be added to the elements to indicate the properties of an edge. As for multigraphs with more than one edge between the same start point and end point, we simply store these edges together and add a pointer to the element to link the edges with one another just use the $NextRec$ pointer. And for graphs with different types of edges, we can build multiple Wind-Bell indexes for each type of edge if needed. Building multiple Wind-Bell does not necessarily incur large amount of extra space because the elements in the hanging linked lists would not be repeatedly stored.

B. Operations

Insertion: For each newly added edge e with $\langle u, v\rangle$, we need to insert it into the Wind-Bell. Before the insertion, we use the query operation (it will be shown below) to check whether the point pair $\langle u, v\rangle$ exists. If it does exist, the new edge will be updated into the existing element.

Otherwise, we insert a new element in the Wind-Bell for e. We use N hash functions to calculate the N^2 candidate buckets, put e into the least loaded bucket among them, and increase the CNT of the chosen bucket by one. We can flexibly adjust N, the number of hash functions used, according to the application scenario. If we lay more emphasis on data load balancing, we can use a larger N to select the least loaded bucket from more candidates when inserting elements; if we emphasize more on query performance, we can use a smaller N to reduce the number of buckets to be traversed during the query.

Example (Figure 1): Suppose that we want to insert an edge e marked as $\langle 18, 06\rangle$, which is shown in green. In this example, we set N to 2 and calculate the 4 candidate buckets using the 2 hash functions, and traverse all the buckets to confirm that e is a new edge for this graph, as shown in Step 1-4. After that, we choose the bucket with the smallest counter, which is 0 in the example, increase the CNT of the chosen bucket by one and put e on the top of the hanging linked list of PTR.

Query: To query an edge by $\langle u_q, v_q\rangle$, we first use the N hash functions to calculate the N^2 candidate buckets that may contain the edge $\langle u_q, v_q\rangle$. Then we traverse the linked lists attached to these N^2 buckets to find out whether the target edge exists. If it does, we directly return the element; Otherwise, we report that the query target does not exist and return NULL.

Example (Figure 1): Suppose that we want to query an edge marked as $\langle 20, 21\rangle$, which is shown in orange. We calculate the 4 candidate buckets using the 2 hash functions as well, and traverse all the buckets to check whether the element $\langle 20, 21\rangle$ exists in the graph. In this example, we query the 4 candidate buckets in the ascending order[2] of CNT. Luckily, the query hit the element E_q in the first hanging linked list, so we directly return E_q without checking the rest of the linked lists.

Algorithm 1: Insertion-Wind-Bell

```
Procedure Insertion(e:⟨u, v⟩):
    $ptr \leftarrow Query(e)$;
    if $ptr$ != $null$ then
        update the element $e$ in $ptr$;
    end
    for each $i \in [1, N]$ do
        for each $j \in [1, N]$ do
            $Buc \leftarrow$ bucket with the least counter $A[Hash_i(u)][Hash_j(v)]$;
        end
    end
    put $e$ into $Buc$;
```

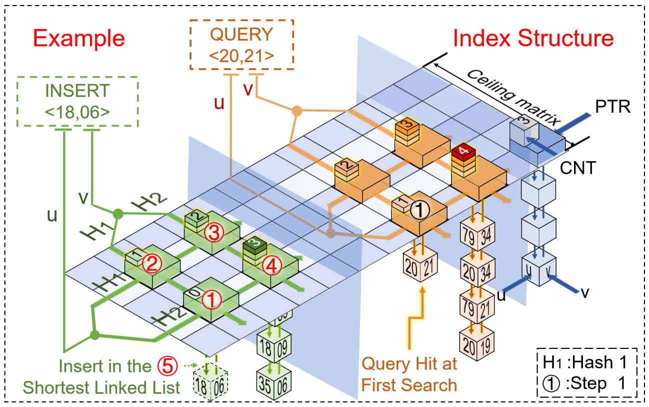


Figure 1: Basic index structure and operations.

C. Optimization

In order to achieve better load balance, we propose an optimization strategy and provide an adjustment operation.

We will adjust the longest linked list among the N^2 candidate buckets during an insertion when it meets the following two conditions:

1) The length of the longest list exceeds a predefined threshold T_{abs}.

[1]Using the counter field is inspired by the Fast Hash table [27].

[2]We prove the efficiency of quering in the ascending order in Section IV

Algorithm 2: Query-Wind-Bell

```
1 Function Query($\langle u_q, v_q \rangle$):
2     for each $i \in [1, N]$ do
3         for each $j \in [1, N]$ do
4             if element $E_q$ is found in
                $A[Hash_i(u_q)][Hash_j(v_q)]$ then
5                 return $E_q$;
6             end
7         end
8     end
9     return null;
```

2) The ratio of the length of the longest list to the shortest list exceeds a predefined threshold T_r.

If so, a randomly-chosen element e_r will be kicked out from the longest linked list and reinserted. This operation will move e_r to the shortest linked list in N^2 candidate buckets of e_r, so as to balance the load. This operation may cause a series of elements to be reinserted, and therefore we make a limit to the maximum times of reinsertion in a single operation to ensure that the insertion of one edge won't take too long.

How should we choose the element e_r to be reinserted? Intuitively, adjusting the oldest element in the linked list may be most effective because the shortest list among the N^2 candidate buckets of the oldest element is likely to have changed after many operations. So we maintain an additional pointer PTR_t in each bucket of the ceiling matrix pointing to the tail of the hanging linked list. By maintaining the tail pointer, we can directly find the oldest element in the list and reinsert it.

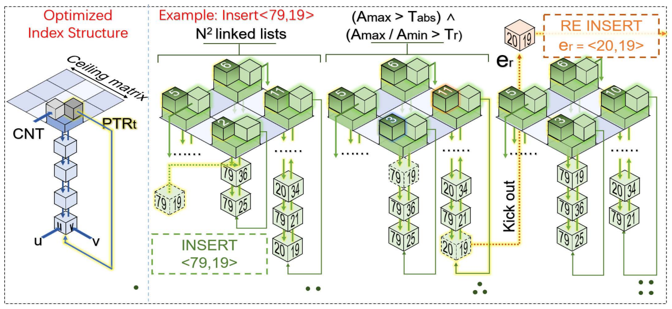


Figure 2: Optimization: kick strategy for load balance.

Example (Figure 2): To better show our optimization, we present an example of the adjustment operation. In this example, we define $N^2 = 4$, $Tabs = 10$, and $Tr = 2$. Suppose that we want to insert an edge e marked as $\langle 79, 19 \rangle$. First, we calculate the 4 candidate buckets as before, and check the least loaded bucket A_{min} (3) and the most loaded bucket A_{max} (11). Noticing that the CNT of A_{max} is bigger than T_{abs}, and the ratio of the CNT of A_{max} to A_{min} is also bigger than T_r, so we kick out the tail element e_r $\langle 20, 19 \rangle$ from A_{max}, and reinsert it after the insertion of e is completed.

Analysis: Adding the tail pointer will increase the space consumption of the ceiling matrix by an additional $1/2$, but the space consumption of the ceiling matrix takes a very small part in the whole Wind-Bell, which is acceptable compared with its optimization effect. Besides, the extra time consumed by the adjustment operation is constant since there's no need to perform the query operation when reinserting the kicked-out element. In other words, we only need to use N hash functions and select the bucket with the smallest counter to reinsert the kicked-out element.

IV. Time complexity analysis

In this section, we conduct theoretical analysis of the Wind-Bell Index. Our goal is to evaluate the memory access time T when QUERY to show Wind-Bell Index has a high efficiency.

Assuming that we have m nodes and S edges in total. The out- and in- degrees of node i are marked as X_i and Y_i respectively, and they are independently and identically distributed[3] in random variable X (i.e. $X_{[1..m]}, Y_{[1..m]} \overset{i.i.d.}{\sim} X$). We define the size of the ceiling matrix is $K * K$ $(K \leqslant m)$. Suppose there are G_i start nodes $X_{i[1..G_i]}$ mapped to row i and H_j end nodes $Y_{j[1..H_j]}$ mapped to column j $(0 \leqslant G_i, H_j \leqslant m)$. Every time we insert an element, we select N rows and N columns to find N^2 candidate buckets, inserting the element into the shortest one. For simplicity, in this section, we fix $N = 1$.

Obviously the consumption of a single adjacency list is

$$T_1 = \mathbb{E}\left[\sum_{i=1}^{m} \mathbb{E}\left(X_i \mathbb{E} Y_{X_i}\right)\right] = \mathbb{E}\left(\sum_{i=1}^{m} \frac{X_i^2}{2}\right) = \frac{m}{2}\mathbb{E}X^2.$$

We want to prove that for a Wind-Bell Index, if the variance of $r.v.X$ satisfies $\lim\limits_{m\to+\infty} \mathbb{D}X/m = 0$, $\mathbb{E}X > \mu > 0$, we have

$$T_2 = \frac{K^2}{2m^2\left(\mathbb{E}X\right)^2}\left(\mathbb{D}X\frac{m}{K} + \mathbb{E}X^2\frac{m^2}{K^2}\right).$$

To prove this theorem, we first give two observations.

In the first step, noticing the distribution of X may change with the nodes number m, we can't directly use any existing SLLN. However, we observe that given $\lim\limits_{m\to+\infty} \mathbb{D}X/m = 0$ and $\mathbb{E}X > \mu > 0$, we can use Chebyshev inequality and get

$$\mathbb{P}\left(\left|\frac{S}{m\mathbb{E}X} - 1\right| \geqslant \frac{\epsilon}{\mathbb{E}X}\right) \leqslant \frac{1}{\epsilon^2}\mathbb{D}\left(\frac{S}{m}\right) = \frac{\mathbb{D}X}{m\epsilon^2} \to 0\,(m \to +\infty).$$

In other words, $m\mathbb{E}X$ can be used to approximately replace the edge number S.

[3]In fact, the out- and in- degrees must satisfy $\sum\limits_i X_i = \sum\limits_i Y_i = S$. We roughly use the $i.i.d.$ condition to simplify the proof.

In the second step, we observe that if $\lim\limits_{m\to+\infty} K/m = 0$, we have

$$\mathbb{E}G_g^2 \xlongequal{G_g\sim B(m,\frac{1}{K})} \sum_{i=0}^{m} i^2 C_m^i \left(\frac{1}{K}\right)^i \left(1-\frac{1}{K}\right)^{m-i}$$

$$= \frac{m}{K}\sum_{i=0}^{m-1}[iC_{m-1}^i \left(\frac{1}{K}\right)^i \left(1-\frac{1}{K}\right)^{m-1-i}$$

$$+ C_{m-1}^i \left(\frac{1}{K}\right)^i \left(1-\frac{1}{K}\right)^{m-1-i}]$$

$$= \frac{m^2}{K^2}\left(1+\frac{K}{m}-\frac{1}{m}\right) = \frac{m^2}{K^2}\,(m\to+\infty).$$

Based on above two observations, we can easily prove our main theorem. Since X_i and Y_j are independent, $\sum\limits_{g=1}^{G_i} X_{ig}$ and $\sum\limits_{h=1}^{H_h} Y_{jh}$ are independent as well. Therefore

$$T_2 := \mathbb{E}\sum_{i=1}^{K}\sum_{j=1}^{K}\left(\frac{1}{2}\sum_{g=1}^{G_i} X_{ig}\frac{\sum\limits_{h=1}^{H_j} Y_{jh}}{\sum\limits_{i=1}^{m} X_i}\right)^2$$

$$\xlongequal{\text{Observation 1}} \mathbb{E}\sum_{i=1}^{K}\sum_{j=1}^{K}\left(\frac{1}{2}\sum_{g=1}^{G_i} X_{ig}\frac{\sum\limits_{h=1}^{H_j} Y_{jh}}{m\mathbb{E}X}\right)^2$$

$$\xlongequal{i.i.d.} \frac{K^2}{2(m\mathbb{E}X)^2}\left[\mathbb{E}\left(\sum_{g=1}^{G_g} X_{ig}\right)^2\right]^2$$

$$\xlongequal{\mathbb{E}(\cdot)=\mathbb{E}(\mathbb{E}(\cdot))} \frac{K^2}{2(m\mathbb{E}X)^2}\times$$

$$\left\{\mathbb{E}\left(\sum_{g=1}^{G_g}\mathbb{E}X_{ig}^2\right) + \mathbb{E}\left(\sum_{g=1}^{G_g}\sum_{g'=1,g'\neq g}^{G_g}\mathbb{E}\left(X_{ig}X_{ig'}\right)^2\right)\right\}^2$$

$$\xlongequal{i.i.d.} \frac{K^2}{2(m\mathbb{E}X)^2}\times$$

$$\left\{\mathbb{E}\left(G_g\mathbb{E}X^2\right) + \mathbb{E}\left[\left(G_g^2-G_g\right)(\mathbb{E}X)^2\right]\right\}^2$$

$$= \frac{K^2}{2(m\mathbb{E}X)^2}\left(\mathbb{D}X\mathbb{E}G_g + \mathbb{E}X^2\mathbb{E}G_g^2\right)^2$$

$$\xlongequal{\text{Observation 2}} \frac{K^2\left(\mathbb{D}X\frac{m}{K} + \mathbb{E}X^2\frac{m^2}{K^2}\right)^2}{2m^2(\mathbb{E}X)^2}.$$

To conclude, we have

$$\frac{T_1}{T_2} = \frac{m(\mathbb{E}X)^2\mathbb{E}X^2}{\left[\mathbb{D}X + \frac{m(\mathbb{E}X)^2}{K}\right]^2}.$$

With the condition of $K*K = m$, we can prove that the memory access time of Wind-Bell Index is always less than that of an adjacency list. When $K=\sqrt{m}$, we have

$$\frac{T_1}{T_2} = \left(\frac{1}{\sqrt{m}}\left(\sqrt{\frac{\mathbb{E}X^2}{(\mathbb{E}X)^2}} - \sqrt{\frac{(\mathbb{E}X)^2}{\mathbb{E}X^2}}\right) + \sqrt{\frac{(\mathbb{E}X)^2}{\mathbb{E}X^2}}\right)^{-2}.$$

Let $x := \mathbb{E}X^2/(\mathbb{E}X)^2$. On the one hand, according to Jensen inequality, we have $x \geqslant 1$; on the other hand, according to $\sum\limits_{i=1}^{m} X_i^2 \leqslant (\sum\limits_{i=1}^{m} X_i)^2$ we have $x \leqslant m$. Let $F(x) := T_1/T_2$, we can get $\frac{dF(x)}{dx}|_{x=\sqrt{m}-1} = 0$. So we claim $F(x)$ increases from 1 to $\sqrt{m}-1$, decreases from $\sqrt{m}-1$ to m. Therefore $F(x) \geqslant F(1) = F(m) = 1$, which means Wind-Bell Index has a better query efficiency than adjacency linked list.

To take a step further, if the density function of $r.v.X$ obeys the power-law distribution

$$p_X(x) = \begin{cases} \frac{\alpha}{1-m^{-\alpha}}\frac{1}{x^{\alpha+1}}, & x\in[1,m]; \\ 0, & x\notin[1,m], \end{cases}$$

through some mathematical tricks, we can get a more precise theoretical result (shown in Table I, note that $m\to+\infty$). We can also draw Figure 3 to demonstrate that our Wind-Bell Index can be up to 200 times faster than the adjacency linked list when $m=10^6$.

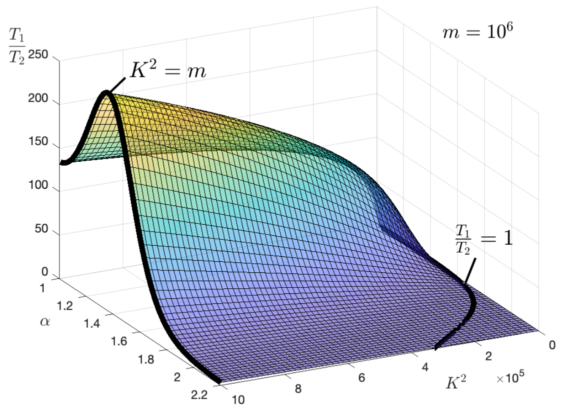


Figure 3: T_1/T_2 when $m=10^6$.

In Section VI, we find that the experimental results consist with the theoretical analysis.

V. Implementation in Neo4j

In this section, we detail the interface we have made in the open source version 4.2.0 of Neo4j.

First we introduce the basic data structure in Neo4j. In Neo4j, graph consists of nodes and relations; nodes have their labels, while relations have their types. Besides, other information about nodes and relations is described as properties. Each relation has its source node and destination node and the two nodes will save the relation in their own adjacency linked list. Neo4j provides iterators to traverse the adjacency linked lists.

Table I: Estimate of T_1/T_2.

$F := T_1/T_2$	$\alpha \in (1, 2)$	$\alpha \approx 2$	$\alpha \in (2, +\infty)$
$K \ll m\mathbb{E}^2 X/\mathbb{D}X$	$(\alpha-1)^2 K^2 m^{1-\alpha}/\alpha(2-\alpha)$	$K^2 \ln^2 m/m$	$(\alpha-1)^2 K^2/\alpha(\alpha-2)m$
$K \approx m\mathbb{E}^2 X/\mathbb{D}X$	$\alpha(2-\alpha)m^{\alpha+1}/(\alpha-1)^2$	$2m/\ln m$	$\alpha(\alpha-2)m$
$K \gg m\mathbb{E}^2 X/\mathbb{D}X$	$\alpha(2-\alpha)m^{\alpha-1}/(\alpha-1)^2$	$m/\ln^2 m$	$\alpha(\alpha-1)^2(\alpha-2)$

Neo4j does not provide an edge index. Therefore, the original transaction of Neo4j does not support directly getting an edge by its start point and end point. A particular edge can be indirectly visited through the function $getRelationships$ provided by the class $Node$, which returns all the edges connected to the node. In other words, in order to get an edge by its start point u_q and end point v_q in Neo4j, we need to first find the node of the start point u_q, and then traverse the linked list of u_q to get the edge we want. If the degree of u_q is high, traversing its linked list is time consuming. Therefore, we add the new interface to the transaction of Neo4j, which can get an edge without traversing the adjacency list of the related nodes. The latest open-source version of Neo4j is 4.4.12. This version has a interface which could query an edge by its start point and end point, but as fast as the query function we mentioned above.

We make the following three modifications in the interface implementation of our Wind-Bell Index.

1) To realize data persistence, we add read and write functions for the Wind-Bell. We append a field for our Wind-Bell Index in the class of $TransactionImpl$ in Neo4j. If the graph database is restarted, we create and initialize a new Wind-Bell; otherwise, we load the previous Wind-Bell from the disk. When the current transaction is committed, we write the Wind-Bell into the corresponding file in the disk. Through this modification, we support the open and close operations of Neo4j and establish the consistency between the graph database and the Wind-Bell Index.
2) To realize real-time update of the relational changes in the graph, we set a monitor in the edge creation function $CreateRelaionshipTo$ of class $Node$. Every time a node u creates a new edge connected to node v, we can capture the action and insert the edge e with the pair $\langle u, v\rangle$ into our Wind-Bell.
3) To realize fast multi-edge query, we modify our index structure. In the implementation, we notice that Neo4j allows multiple relationships between the same start point u and end point v [4]. Therefore, we organize all the edges with the same $\langle u, v\rangle$ pair as a linked list and regard the whole linked list as one element, which is attached to the field $nextRec$ in the hanging linked list. We call the element an $edgelinkedlist$ to distinguish it from the hanging linked list. As a result, what our interface returns is actually an iterator, by which we can traverse the edge linked list and get all the edges between point $\langle u, v\rangle$.

[4]In practical graphs, two adjacent nodes could have many edges. For example, two bank accounts could have many transactions.

Through the modifications mentioned above, we realize the data persistence, real-time update, and multi-edge query for our Wind-Bell Index. Through the interface we provided in the $TransactionImpl$ of Neo4j, we can directly query any particular edge given the start point and end point at a higher speed.

VI. Experimental Results

In this section, we present our experimental results of the Wind-Bell Index. First, we describe the experimental setup in Section VI-A. Second, we present the time improvement and space performance of implementing Wind-Bell in Section VI-B and Section VI-C, respectively. Finally, we summarise and analyze the overhead of Wind-Bell in Section VI-D.

The query operation in Neo4j can be made by using Java API, TraverserFrameWork, or Cypher query language. Experimental results show that the query speed of using Java API is the fastest among the three query methods [30]. Therefore, we choose the original Java API provided by Neo4j as the baseline in our experiments.

A. Experimental Setup

Setup Details: We implement our Wind-Bell in Neo4j with java, and then compare the storage and query performance of our algorithm with the original query interface for relations provided by Neo4j. The default parameters are set as follows. The number of hash functions N is set to 2. The length and width of matrix K is set to 100. The default data size is set to $1 * 10^6$. The maximum data size[5] is set to $1.3 * 10^6$.

Datasets: We use the dataset from CAIDA[31] network data in our experiment. CAIDA datasets identifies each flow of IP trace streams by the five-tuples: source and destination IP address, source, and destination port, protocol. With 13 bytes of five-tuples, there are another 8 bytes in each record for timestamp.

We use the source and destination IP address in the traces as the start point and end point for the graph, respectively. We create timestamps ourselves, although there are timestamps in the dataset of CAIDA, the 8-byte timestamp is not fit for JAVA experiments, filtering by the timestamps would be difficult on ranging, so we make they ourselves. We make timestamp by random function, 80% on $(0, 10000)$, and the other on $(0, 1000000)$, simulating that there comes a burst on time $(0, 10000)$. The dataset of CAIDA we used contains 1067013 edges concerning 59304 different nodes, more details on Table II.

[5]Data size refers to the number of edges we insert in the experiments

Table II: Details of CAIDA dataset.

Node Number	59304
Edge Number	1067013
Max Node Degree	486
Min Node Degree	2
P95 Node Degree	59
P99 Node Degree	457

Computation Platform: Our experiments are performed on a server with 18-core CPUs (36 threads, Intel(R) Core i9-10980XE CPU @ 3.00 GHz) and 128GB DRAM memory.

Queries: We run three different queries in Neo4j:

$Q1$: MATCH (a)-$[e]$->(b)
RETURN e

$Q2$: MATCH (a)-$[e]$->(b)
WHERE $e.time$>1000&$e.time$<100000
RETURN e

$Q3$: MATCH (a)->(b)->(c)->(a)
RETURN a, b, c

$Q3$ is query of finding triangle.

Metrics: We compare our Wind-Bell with the original interface of Neo4j using the following metrics and describe the time distributions of corresponding experiments.

1) ***Average Query Time (AQT):*** *The average time needed for performing one edge query.*
2) ***Average Insertion Time (AIT):*** *The average time needed for inserting one edge.*
3) ***Loading Rate (LR):*** *The proportion of buckets in which the counter is not zero.*
4) ***Average List Length (ALL):*** *The average length of the buckets in which the counter is not zero.*
5) ***Longest List Length (LLL):*** *The length of the longest list in all buckets.*
6) ***Average Memory Usage (AMU):*** *The average memory used for Wind-Bell Index to store one edge.*

B. Experiments on Time

Average Query Time vs. Data Size (Figure 4(a)). We find that the *average query time* of the optimized Wind-Bell is *at least* 36.85 *times shorter* than that of the original Neo4j. The results show that the ratio of the query time of original Neo4j to the optimized Wind-Bell increases when the data size grows larger. When the data size is 1300K, the average query time of the optimized Wind-Bell is 7.88×10^{-4} ms, which is 557.38 times shorter than that of the original Neo4j.

Average Query Time vs. Matrix Width (Figure 4(b)). We find that the *average query time* of the optimized Wind-Bell is *at least* 328.33 *times shorter* than that of the original Neo4j. The results show that when setting the matrix width to 200, the average query time of the optimized Wind-Bell is 1.047×10^{-3} ms, which is 328.78 times shorter than that of the original Neo4j.

Average Query Time on Finding Triangles (Figure 4(c)). We find that the *average query time* of finding triangles of the optimized Wind-Bell is *at least* 17601.49 *times shorter* than that of the original Neo4j. The results show that when querying 900K node pairs, the average query time of Wind-Bell is 2.36×10^{-4} ms, which is 33718.93 times shorter than that of the original Neo4j. In the experiment of finding triangles, we use the simplest approach: Enumerating groups of three nodes like $\langle u, v, w \rangle$, and then checking whether the three points form a triangle. We enumerate 100K, 300K, 500K, 700K, 900K triangles, and then get the average query time of finding a triangle.

Summary and Analysis. In summary, Wind-Bell Index achieves high speed of query in different situations, it performs about hundreds of times better than the original Neo4j on average, and even 10,000 times better in the problem of finding triangles. The reason for such a good performance is that the Wind-Bell avoids the steps of finding nodes from the database and traversing the long adjacency lists of high-degree nodes to confirm the edges. Besides, maintaining the load balance of Wind-Bell is another reason for the well performance, which reduces the times of memory access in each query.

C. Experiments on Space

Loading Rate vs. Data Size (Figure 5(a)). We find that the *loading rate* of the optimized Wind-Bell is *at least* 5.61% *higher* than that of the original Wind-Bell. The results show that when using data size of 1300K, the loading rate of the optimized Wind-Bell is 93.30%, which is 32.42% higher than that of the original Wind-Bell.

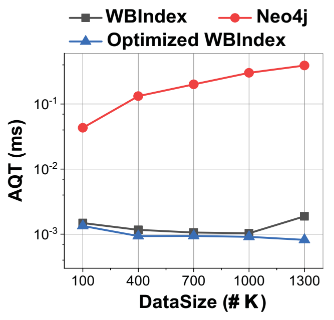
Average List Length vs. Data Size (Figure 5(b)). We find that the *average list length* of the optimized Wind-Bell is *at least* 24.11% *shorter* than that of the original Wind-Bell. The results show that when using data size of 1300K, the average list length of the optimized Wind-Bell is 2.79, which is 24.48% shorter than that of the original Wind-Bell.

Longest List Length vs. Data Size (Figure 5(c)). We find that the *longest list length* of the optimized Wind-Bell is *at least* 59.46% *shorter* than that of the original Wind-Bell. The results show that when using data size of 1300K, the longest list length of the optimized Wind-Bell is 23, which is 62.30% shorter than that of the original Wind-Bell.

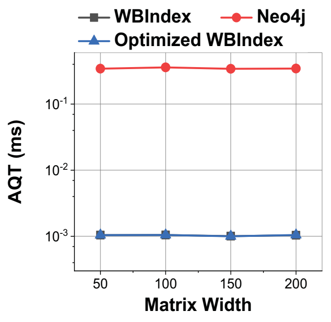
Summary and Analysis. In summary, the experiments on space show that the optimized Wind-Bell achieves better load balance in different metrics comparing with the original Wind-Bell. It proves that our optimization of kicking out the tail element is a good way to balance the load of the ceiling matrix. Therefore, we may find that the query speed of the optimized Wind-Bell is higher than the original Wind-Bell. The improvement of speed may be not so obvious in our experiments, but we believe that the optimization scheme will have a significant improvement effect when the skewness of the degree distribution increases.

D. Overhead

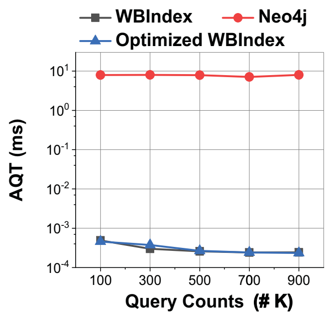
Average Insertion Time vs. Data Size (Figure 4(d)). We find that the *average insertion time* of the optimized Wind-Bell is



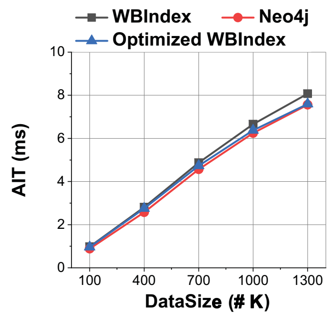
(a) AQT vs. Data Size.



(b) AQT vs. Matrix Width.

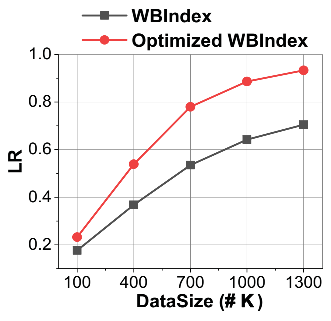


(c) AQT on Finding Triangles.

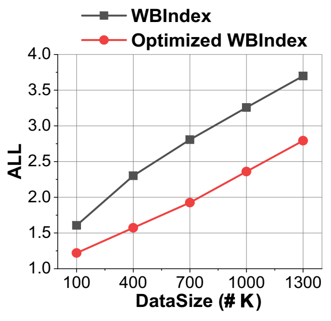


(d) AIT vs. Data Size.

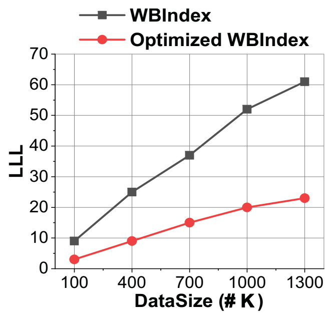
Figure 4: Experiments on time.



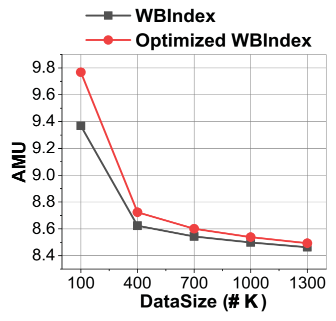
(a) LR vs. Data Size.



(b) ALL vs. Data Size.



(c) LLL vs. Data Size.



(d) AMU vs. Data Size.

Figure 5: Experiments on space.

at most 8.65% *longer* than that of the original Neo4j. The results show that when using data size of 1300K, the average insertion time of the optimized Wind-Bell is 7.60 ms, which is 0.51% longer than that of the original Neo4j.

In addition to the metrics shown in the figures, we also describe the time distribution of insertion of $1000K$ in Table III. We find that the difference between the maximum insertion time and the average insertion time is not significant, showing that Wind-Bell index is a balanced data structure. Besides, we find that the 99% quantile of the time distribution is very close to the average time, which also prove the balanced performance of Wind-Bell index.

Table III: Details of Time Distribution.

	Insertion time
Average Time(ms)	7.44
Min Time(ms)	6
Max Time(ms)	28
P99 Time(ms)	8
P95 Time(ms)	8

Average Memory Usage vs. Data Size (Figure 5(d)). We find that the *average memory usage* of the optimized Wind-Bell is *at most* 4.27% *larger* than that of the original Wind-Bell. The results show that the ratio of the memory usage of the optimized Wind-Bell to the original Wind-Bell decreases when the data size grows larger. When the data size is 1300K, the average memory usage of the optimized Wind-Bell is 8.49 byte, which is 0.36% larger than that of the original Wind-Bell.

Summary and Analysis. In summary, Wind-Bell does not bring a large overhead in the insertion time. The overhead of insertion is relatively small, which can be ignored when compared with the insertion time of the original Neo4j. The space Wind-Bell consumes is not so small to be ignored, but it is not so big for an index. For every edge in the graph, we take about 8 bytes to maintain it in the Wind-Bell. When the graph grows larger, the space consumption of the Wind-Bell will become larger, but we believe it is acceptable and worthwhile compared with the significant speed up effect. In practical application, user can flexibly adjust the parameters in the data structure. A larger K results in a larger ceiling matrix which consumes more space but shortens the average length of the hanging linked lists. A larger N indicates using more hash functions which leads to more balanced hanging linked lists but increases the insertion and query time. It would be better to use a larger K if the data volume is large. However increasing N is more time-consuming than we expected, so we suggest that N be set to 2.

VII. Conclusion and Future Work

In this paper, we propose a new data structure Wind-Bell Index, which is used as an edge index to accelerate edge query in graph databases. We implement it in a widely-used graph database, Neo4j, and provide a new interface for edge query. In addition to the basic version of Wind-Bell Index, we also provide an optimization strategy to realize better load balance. In the experiments, our Wind-Bell achieves excellent performance: It speeds up the edge query by hundreds of times compared with the original query interface of Neo4j. The key idea of Wind-Bell index is to avoid traveling through long linked lists of high-degree nodes. We believe that Wind-Bell index can play a better role in the industry, where high degree nodes become a bottleneck. It is reasonable to believe that the performance of Neo4j on multiple tasks can be further accelerated by using our index and query interface. And all the other graph databases which also use adjacency lists to store the graphs can be improved by implementing Wind-Bell Index.

In our future work, we hope that we can further improve our code and add the Wind-Bell Index to the open source community of Neo4j to provide a convenient interface for efficient edge query. Moreover, we plan to implement the Wind-Bell Index in more graph databases, and try to extend our index structure to distributed graph databases to adapt to a wider range of application scenarios.

VIII. Ackowledgment

We would like to thank the anonymous reviewers for their thoughtful feedback. This work is supported by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, and National Natural Science Foundation of China (NSFC) (No. U20A20179, 61832001).

References

[1] Lada A Adamic, Bernardo A Huberman, AL Barabási, R Albert, H Jeong, and G Bianconi. Power-law distribution of the world wide web. *science*, 287(5461):2115–2115, 2000.

[2] Robert J Cimikowski. Graph planarization and skewness. *Congressus Numerantium*, pages 21–21, 1992.

[3] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *ACM SIGCOMM computer communication review*, 29(4):251–262, 1999.

[4] Deepayan Chakrabarti and Christos Faloutsos. Graph mining: Laws, generators, and algorithms. *ACM computing surveys (CSUR)*, 38(1):2–es, 2006.

[5] Neo4j website [online]. https://neo4j.com/.

[6] Zongyue Qin, Yunsheng Bai, and Yizhou Sun. Ghashing: Semantic graph hashing for approximate similarity search in graph databases. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2062–2072, 2020.

[7] Yongjiang Liang and Peixiang Zhao. Similarity search in graph databases: A multi-layered indexing approach. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 783–794. IEEE, 2017.

[8] Shixuan Sun and Qiong Luo. Scaling up subgraph query processing with efficient subgraph matching. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 220–231. IEEE, 2019.

[9] Yongjiang Liang and Peixiang Zhao. Workload-aware subgraph query caching and processing in large graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1754–1757. IEEE, 2019.

[10] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, Pingfu Chao, and Xiaofang Zhou. Efficient constrained shortest path query answering with forest hop labeling. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1763–1774. IEEE, 2021.

[11] Theodoros Chondrogiannis and Johann Gamper. Exploring graph partitioning for shortest path queries on road networks. In *26th GI-Workshop Grundlagen von Datenbanken: GvDB'14*, pages 71–76, 2014.

[12] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, and Ulf Leser. Exact and approximate algorithms for finding k-shortest paths with limited overlap. In *20th International Conference on Extending Database Technology: EDBT 2017*, pages 414–425, 2017.

[13] The source codes related to wind-bell index. https://anonymous.4open.science/r/Wind-Bell-Index-2CB7/.

[14] Orientdb website [online]. https://orientdb.com/.

[15] Norbert Martinez-Bazan, Sergio Gomez-Villamor, and Francesc Escale-Claveras. Dex: A high-performance graph database management system. In *2011 IEEE 27th International Conference on Data Engineering Workshops*, pages 124–127. IEEE, 2011.

[16] Borislav Iordanov. Hypergraphdb: a generalized graph database. In *International conference on web-age information management*, pages 25–36. Springer, 2010.

[17] Db-engines ranking of graph dbms [online]. https://db-engines.com/en/ranking/graph+dbms.

[18] Ali Ben Ammar. Query optimization techniques in graph databases. *arXiv preprint arXiv:1609.01893*, 2016.

[19] Daniel Nicoara, Shahin Kamali, Khuzaima Daudjee, and Lei Chen. Hermes: Dynamic partitioning for distributed social network graph databases. In *EDBT*, pages 25–36, 2015.

[20] Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. Fast query decomposition for batch shortest path processing in road networks. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1189–1200. IEEE, 2020.

[21] Wenfei Fan, Xin Wang, and Yinghui Wu. Incremental graph pattern matching. *ACM Transactions on Database Systems (TODS)*, 38(3):1–47, 2013.

[22] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 5–14, 2012.

[23] S. Acharya, B. S. Lee, and P. Hines. gsketch: On query estimation in graph streams.

[24] Yossi Azar, Andrei Z Broder, Anna R Karlin, and Eli Upfal. Balanced allocations. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 593–602, 1994.

[25] Andrei Z Broder and Anna R Karlin. Multilevel adaptive hashing. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 43–53, 1990.

[26] Yuhan Wu, Zirui Liu, Xiang Yu, Jie Gui, Haochen Gan, Yuhao Han, Tao Li, Ori Rottenstreich, and Tong Yang. Mapembed: Perfect hashing with high load factor and fast update. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 1863–1872, 2021.

[27] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. *ACM SIGCOMM Computer Communication Review*, 35(4):181–192, 2005.

[28] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[29] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[30] Hongcheng Huang and Ziyu Dong. Research on architecture and query performance based on distributed graph database neo4j. In *2013 3rd International Conference on Consumer Electronics, Communications and Networks*, pages 533–536. IEEE, 2013.

[31] The CAIDA Anonymized Internet Traces. http://www.caida.org/data/overview/.