Received January 24, 2017; accepted March 4, 2017, date of publication March 14, 2017, date of current version April 24, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2682107

Efficient Graph Similarity Search in External Memory

XIAOYANG CHEN[1], HONGWEI HUO[1], (Member, IEEE), JUN HUAN[2], (Senior Member, IEEE), AND JEFFREY SCOTT VITTER[3], (Fellow, IEEE)
[1]Department of Computer Science, Xidian University, Xi'an 710071, China
[2]Department of Electrical Engineering and Computer Science, The University of Kansas, Lawrence, KS 66045 USA
[3]Department of Computer & Information Science, The University of Mississippi, Oxford, MS 38677-1848 USA

Corresponding author: H. Huo (hwhuo@mail.xidian.edu.cn)

This work was supported in part by the National Science Foundation (NSF) of China under Grant 61173025 and Grant 61373044, and in part by the NSF USA under Grant CCF-1017623.

ABSTRACT Many real-world applications, such as bioinformatics, data mining, pattern recognition, and social network analysis, benefit from efficient solutions for the graph similarity search problem. Existing methods have limited scalability when they handle the large graph databases, for example, those with millions or billions of graphs that cannot fit in main memory. In this paper, we study the problem of graph similarity search under the graph edit distance constraint in external memory. We present an efficient framework for arbitrary *q-gram*-based representations of a graph. Specifically, we propose a *q-gram* matrix index stored in hybrid layout in external memory to achieve efficient query processing, by converting the *q-gram* counting filter into a sparse matrix-vector multiplication problem. Furthermore, we also boost the query performance by transforming the global filter to a 2-D query rectangle, which allows us to perform a query in a reduced region, significantly reducing the number of query I/Os in practice. Extensive experiments on real data sets confirm that 1) our method can compete with the state-of-the-art in-memory methods in index size and filtering ability, and outperform them on scalability of coping with the PubChem data set including 25 million chemical structure graphs and 2) compared with the popular *q-gram*-based external inverted index, our external index structure needs much fewer number of query I/Os on the PubChem data set.

INDEX TERMS Graph similarity search, matrix index, external memory.

I. INTRODUCTION

Graph search plays a central role in data mining, pattern recognition, databases, machine learning, and big data predictive analytics. Attributed graphs capture the structure of data points and the attribution of nodes and edges. Similarity search of attributed graphs is a core operation of graph data and has applications in many disciplines such as bioinformatics, social network analysis, semantic web, and pattern recognition [8].

The core problem of graph similarity search is well defined [5], [13], [14], [21]: Given a graph database G and a query graph h, the problem is to identify all the graphs in G that are similar to h. There are at least four metrics being investigated [14]: graph edit distance [11], [21], maximal common subgraph distance [3], graph alignment [4], and graph kernel functions [14]. In this paper, we focus upon the graph edit distance (GED) between graphs g and h, denoted by $ged(g, h)$, which is the minimal number of operations that we use to transform g to h (or vice versa). A user or a query index system may specify the set of operations. Typical choices are node label change, edge label change, adding a node, adding an edge, removing a node, removing an edge, or any subset of the operations.

There are a large number of algorithms supporting graph similarity search based upon GED [5], [11], [12], [21]. The critical limitation of existing GED-based approaches is that they do not work well when dealing with very large databases that do not fit in internal memory, such as PubChem, which stores information about roughly 50 million chemical structures. We empirically tested some of the previous state-of-the-art methods and found that they do not scale well, detailed in Section VI. For such large transaction databases, we argue that external memory based methods are important.

We present an efficient framework for graph similarity search in external memory for arbitrary *q-gram* based representations of a graph. Our contributions in this paper are summarized below.

- We propose a *q-gram*-based matrix index for a graph database G. It can scale well to the I/O model by converting the *q-gram* counting filter into a SpMV problem, to achieve efficient query processing.
- We transform the global filter derived based upon the differences of the number of vertices and edges of comparing graphs to a two-dimensional query rectangle, which helps us perform graph similarity search in a reduced region, greatly reducing the number of query I/Os in practice.
- We develop a hybrid *q-gram* filter combining both the label-based *q-gram* and branch-based *q-gram* counting filters, which has a better performance than the tree-based *q-gram* and path-based *q-gram* counting filters.
- We have conducted comprehensive experimental studies to evaluate the filtering capability, number of query I/Os, occupied space, and construction time. The result shows that our method can easily scale to the PubChem dataset contains 25 million chemical structure graphs.

The rest of this paper is organized as follows: In Section II, we introduce the problem definition and related work. In Section III, we present our framework and give an approach to reduce the query region. In Section IV, we introduce the hybrid *q-gram* filter and the *q-gram* matrix index. In Section V, we give an external query method for the *q-gram* matrix stored in hybrid layout. Comprehensive experimental studies appear in Section VI, and we make concluding remarks in Section VII.

II. PROBLEM DEFINITION AND RELATED WORK

A. PROBLEM DEFINITION

In this section, we first provide formal definitions of graph edit distance and graph similarity search and then briefly overview related work. For simplicity, we only focus on simple undirected graphs where they do not have multi-edge or self-loop. Specifically, a graph is a four tuple $g = (V_g, E_g, \lambda, \Sigma_g)$ where V_g is the set of vertices, $E_g \subseteq V_g \times V_g$ is the set of edges, Σ_g is the set of vertex and edge labels, λ is the function that maps vertices and edges to their labels. Clearly $\lambda(u)$ is the label of the vertex u and $\lambda(e(u, v))$ is the label of the edge $e(u, v)$. λ_{V_g} and λ_{E_g} denote the multi-sets of vertex and edge labels, respectively. $|V_g|$ is the number of vertices in g and $|E_g|$ is the number of edges in g, and the graph size refers to $|V_g|$ in this paper.

Definition 1 (Graph Isomorphism [16]): We say that a graph g is isomorphic to another graph h if there exists a bijection $f : V_g \rightarrow V_h$, such that (1) for all $v \in V_g$, we have $f(v) \in V_h$ and $\lambda(v) = \lambda(f(v))$, (2) for all $e(u, v) \in E_g$, we have $e(f(u), f(v)) \in E_h$ and $\lambda(e(u, v)) = \lambda(e(f(u), f(v)))$. If g is isomorphic to h, we denote $g \cong h$.

In this paper we consider six edit operations in transforming one graph to another [11], including that insert/delete an isolated vertex, insert/delete an edge between two vertices, and substitute the label of a vertex or an edge. Given two graphs g and h there always exists at least one edit operation list L that transforms one graph to another, such as, $g = g^0 \rightarrow g^1 \rightarrow \ldots \rightarrow g^d \cong h$. We call such a list a *transforming operation list* between g and h. For any graphs g and h, the number of possible transforming operation lists is infinite. A transformation operation list is *optimal* if it has the shortest length among all possible transforming operation lists.

Definition 2 (Graph Edit Distance, GED): Given two graphs g and h, the edit distance between g and h, denoted by $ged(g, h)$, is the length of an optimal transforming operation list between g and h, or the minimal number of operations to transform one graph to another.

Definition 3 (Graph Similarity Search, GSS): Given a graph database $G = \{g_1, g_2, \ldots, g_n\}$, a query graph h, and a distance upper-limit $\tau \geq 0$, by the graph similarity search we identify the set of graphs in G such that $ged(g, h) \leq \tau$, where $ged(g, h)$ is defined in Definition 2.

Figure 1 shows two data graphs g_1 and g_2 and a query graph h. We can obtain that $ged(g_1, h) = 4$ and $ged(g_2, h) = 10$. If the edit distance threshold $\tau = 4$, only g_1 is the answer.

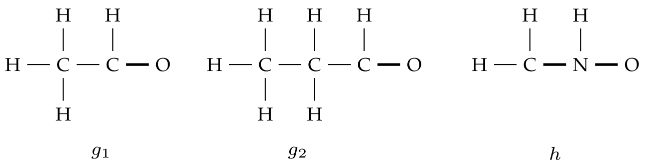

FIGURE 1. Graphs g_1 and g_2, and query graph h.

B. RELATED WORK

1) EXISTING FILTERS

Computing the graph edit distance is an NP-hard problem [18], and hence it is not a trivial task to seek for an efficient algorithm. Most of the existing methods adopt the filter-and-verify schema to speed up the search. With such a schema, we first filter data graphs that are not possible results to generate a candidate set, and then validate the candidate graphs with the expensive graph edit distance computations.

In general, the existing filters can be divided into four categories: global filter, *q-gram* counting filter, mapping distance-based filter and disjoint partition-based filter. Specifically, number count filter [18] and label count filter [20] are two global filters. The former is derived based upon the differences of the number of vertices and edges of comparing graphs. The later takes labels as well as structures into account, further improving the former. κ-AT [11] and GSimJoin [20] are two major *q-gram* counting filters. In κ-AT, a *q-gram* is defined as a tree consisting of a vertex v and the paths whose length no longer than κ starting from v. However, GSimJoin considered the simple path whose length is p as a *q-gram*. C-Star [18] and Mixed [21] are two mapping distance-based filters. The lower bounds are derived based on the minimum weighted bipartite graphs between the star and branch structures of g and h, respectively. SEGOS [12] introduced a two-level index structure to speed up the filtering process, which has the same filter ability with C-star. Pars [19] divided each data graph g into τ+1 non-overlapping substructures, and pruned the graph g if there exists no substructure that is subgraph isomorphic to h. The above methods show different performance on different databases and we can hardly prove the merits of them theoretically [5].

2) EXTERNAL MEMORY MODEL

The external memory model was introduced by Aggarwal and Vitter [1], which is also called the I/O model [10]. In this model, the CPU is connected directly to an internal memory of size M, which is in turn connected to a much larger and slower disk. The CPU can only operate on data inside the internal memory. So, when we need to operate the data stored in disk, we have to transfer data between internal memory and disk through I/O operations. Compared with the cost of the transfer, the cost of operations in internal memory can be negligible. Thus the performance of an algorithm in the external memory model is measured by the number of I/O operations used. Han et al. [6] made full disk-based implementations on representative indexing methods for the subgraph isomorphism problem on a common framework. Tian and Patel [9] presented a disk-based hybrid index, which uses existing common disk-based index structures. Bender et al. [2] presented several optimal algorithms for the variants of the sparse matrix dense vector multiplication in the combination of the I/O models of Aggarwal and Vitter [1], and of Hong and Kung [7].

III. A GENERAL FRAMEWORK

A. FRAMEWORK

Given a graph database G, a query graph h and an edit distance threshold τ, we propose a general framework for the graph similarity search problem in external memory detailed in the following three steps:

Step 1: Transform. We map each graph g in the graph database G to a two-dimensional point $(|V_g|, |E_g|)$. These points can form a rectangle region R. Similarly, the number count filter [18] can also form a query rectangle R_h. By partitioning R into subregions, we can reduce the query region from R to a reduced region Q_h. Both R_h and Q_h are defined in Section III-B.

Step 2: Index construction. For each subregion, we build the *q-gram* matrix between the *q-grams* and the graphs mapped into this subregion, and then store the *q-gram* matrix in hybrid layout in external memory.

Step 3: Query processing. For each subregion in Q_h, we calculate the common *q-grams* between data graph g and query graph h using the *q-gram* matrix, and then filter the graphs that do not satisfy the *q-gram* counting filter to obtain the candidate set $Cand$.

B. TRANSFORM

Given a graph database G, we consider each graph g in G as a point $(|V_g|, |E_g|)$ in the two-dimensional plane where the x-axis and y-axis denote the respective number of vertices and edges in g. Thus the graph database G can be considered as a set of points $S = \{(|V_{g_j}|, |E_{g_j}|) : 1 \le j \le n\}$, where n is the number of graphs in G. These points form a rectangle region $R = [x_{min}, x_{max}] \times [y_{min}, y_{max}]$, where $x_{min} = \min_j\{|V_{g_j}|\}$, $x_{max} = \max_j\{|V_{g_j}|\}$, $y_{min} = \min_j\{|E_{g_j}|\}$ and $y_{max} = \max_j\{|E_{g_j}|\}$ for $1 \le j \le n$. By partitioning R into subregions, we can perform a query in a reduced query region.

Given an initial division point (x_0, y_0) and a length l, we partition R into disjoint subregions as follows. First, we construct the initial subregion $R_{0,0} = [x_0 - l/2, x_0 + l/2] \times [y_0 - l/2, y_0 + l/2]$ of size $l \times l$. Then, we extend along the surrounding $R_{0,0}$ to obtain subregions $R_{i,j}$ of the same size $l \times l$, where i and j are the relative offsets with respect to $R_{0,0}$ in x-axis and y-axis, respectively. Finally, we repeat this process until all points in R are exhausted. Then R is partitioned into some disjoint subregions such that $R = \cup_{i,j} R_{i,j}$ and $R_{i,j} \cap R_{i',j'} = \emptyset$ for all $i \neq i'$ and $j \neq j'$. Note that i and j can be negative.

Definition 4 (Query Rectangle and Region): Given a query graph h and an edit distance threshold τ, query rectangle R_h of h is defined as the rectangle $[|V_h| - \tau, |V_h| + \tau] \times [|E_h| - \tau, |E_h| + \tau]$. The query region Q_h of h is the union of all subregions intersecting with R_h, i.e., $Q_h = \cup_{i,j} R_{i,j}$ such that $R_{i,j} \cap R_h \neq \emptyset$.

Given two graphs g and h, if $ged(g, h) \le \tau$, we know that $||V_g| - |V_h|| + ||E_g| - |E_h|| \le \tau$, then we have $|V_h| - \tau \le |V_g| \le |V_h| + \tau$ and $|E_h| - \tau \le |E_g| \le |E_h| + \tau$. According to the definition of R_h, we have $(|V_g|, |E_g|) \in R_h$. As $R_h \subseteq Q_h$, thus we have $(|V_g|, |E_g|) \in Q_h$ and hence only need to perform the query in the reduced region Q_h. Figure 2 gives an example to illustrate the concepts of region R, query rectangle R_h, and query region Q_h. In the example of Figure 2, we have $Q_h = \{R_{0,0}, R_{1,0}, R_{0,-1}, R_{1,-1}\}$. Thus we only need to perform the query on the subregions $R_{0,0}$, $R_{1,0}$, $R_{0,-1}$, and $R_{1,-1}$.

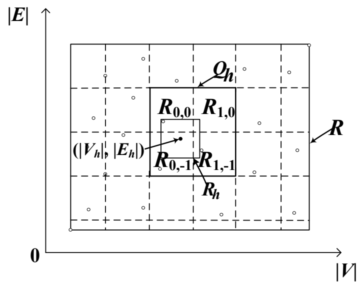

FIGURE 2. Illustration of R, R_h, and Q_h.

For a given query graph h, since the subregions in Q_h are adjacent, we just need to find the boundaries of subregions intersecting with R_h using the following formula:

$$Q_h = \cup_{i,j} R_{i,j} \text{ for all } i_1 \le i \le i_2 \text{ and } j_1 \le j \le j_2. \quad (1)$$

where $i_1 = \lfloor(|V_h| - \tau - (x_0 - l/2))/l\rfloor$ and $j_1 = \lfloor(|E_h| - \tau - (y_0 - l/2))/l\rfloor$ are the relative positions of the subregion in the lower left corner of Q_h with respect to $R_{0,0}$ in x-axis and y-axis, respectively. $i_2 = \lfloor(|V_h| + \tau - (x_0 - l/2))/l\rfloor$ and $j_2 = \lfloor(|E_h| + \tau - (y_0 - l/2))/l\rfloor$ are the relative positions of the subregions in the top right corner of Q_h with respect to $R_{0,0}$ in x-axis and y-axis, respectively.

IV. MATRIX INDEX

In this section, we give two *q-gram*-based counting filters and an index structure: *q-gram* matrix index, which is a matrix index that has been used to similarity search [17].

A. HYBRID *q-gram* FILTERS

*Definition 5 (Branch-Based **q-gram**):* Given a graph g and a vertex v in V_g, the branch structure [21] of v, denoted by b_v, is a tuple $b_v = (\lambda(v), adj(v))$, where $\lambda(v)$ is the label of v and $adj(v)$ is the multi-set of labels for edges adjacent to v. The branch-based *q-gram* of v is the branch structure of v. The branch-based *q-gram* set of graph g is defined as $B(g) = \{b_v : v \in V_g\}$.

It is trivial to see that (1) vertex insertion/deletion/substitution will affect one branch-based *q-gram*, (2) edge insertion/deletion/substitution will affect two branch-based *q-grams*. Thus, one operation changes at most two branch-based *q-grams*. Therefore, for any two graphs g and h when we transform one graph to another after $ged(g, h)$ edit operations, they must share at least $\max\{|V_g|-2\cdot ged(g, h), |V_h|-2\cdot ged(g, h)\}$ common *q-grams*. So, we can obtain the **branch-based *q-gram* counting filter as follows**: if $ged(g, h) \le \tau$, then we have $|B(g) \cap B(h)| \ge \max\{|V_g| - 2\tau, |V_h| - 2\tau\}$.

*Definition 6 (Label-Based **q-gram**):* Given a graph g, a label-based *q-gram* is the label of a vertex or an edge of g. For the graph g, the set of its label-based *q-gram* is $L(g) = \{\lambda(u) : u \in V_g \text{ or } u \in E_g\}$, where $\lambda(u)$ is the label of u.

For the label-based *q-gram*, each edit operation affects one label-based *q-gram*, thus we can obtain the **label-based *q-gram* counting filter as follows**: if $ged(g, h) \le \tau$, then we have $|L(g) \cap L(h)| \ge \max\{|V_g|, |V_h|\} + \max\{|E_g|, |E_h|\} - \tau$.

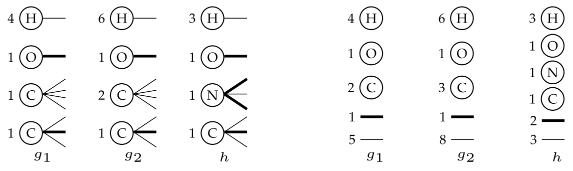

FIGURE 3. Branch-based ***q-gram*** (left) and label-based ***q-gram*** (right) sets.

In Figure 3, we show the branch-based *q-gram* and the label-based *q-gram* sets of the graphs shown in Figure 1. The number on the left of each subgraph is the times of the *q-gram* occurring in the graph.

B. *q-gram* MATRIX INDEX

In this subsection, we give a matrix index structure, referred to as *q-gram* matrix index, which is able to efficiently implement *q-gram*-based counting filters, including the branch-based and the label-based *q-gram* counting filters used in this paper.

Let $\mathcal{U}$ be the set of all distinct *q-grams* occurring in G and $Q(g)$ be the *q-gram* multi-set of a graph g. For the graph g, we use a *q-gram* vector $w(g)$ to represent its *q-gram* set $Q(g)$, where $w_j(g)$ is the number of occurrences of the *q-gram* $\mathcal{U}_j$ in $Q(g)$. The *q-gram* matrix W built on the graph database G is an $n \times |\mathcal{U}|$ matrix, where $W_{ij} = w_j(g_i)$. A schematic representation of W is given below:

$$W = \begin{pmatrix} w_1(g_1) & w_2(g_1) & \dots & w_{|\mathcal{U}|}(g_1) \\ w_1(g_2) & w_2(g_2) & \dots & w_{|\mathcal{U}|}(g_2) \\ \vdots & \vdots & \vdots & \vdots \\ w_1(g_n) & w_2(g_n) & \dots & w_{|\mathcal{U}|}(g_n) \end{pmatrix}$$

Given a graph g, a query graph h and a *q-gram* $\mathcal{U}_j$, only when $\mathcal{U}_j \in Q(g) \cap Q(h)$, the shared *q-grams* between g and h increases by the amount of $\min\{w_j(g), w_j(h)\}$. Thus the number of common *q-grams* between graphs g and h is $|Q(g) \cap Q(h)| = \sum_{j=1}^{|\mathcal{U}|} \min\{w_j(g), w_j(h)\}$.

In the following sections, we use W_B and W_L to denote the branch-based and the label-based *q-gram* matrices, respectively, $w_B(h)$ and $w_L(h)$ to denote the branch-based and the label-based *q-gram* query vectors of h, respectively. Table 1 shows the branch-based *q-gram* matrix W_B and the label-based *q-gram* matrix W_L built on the graphs g_1 and g_2 shown in Figure 1, where 1 and 2 in the first row of Table 1 represent the label of fine and thick edges in the graphs shown in Figure 1, respectively.

TABLE 1. W_B**(left) and** W_L**(right).**

	H1	O2	C1111	C112
g_1	4	1	1	1
g_2	6	1	2	1

	H	O	C	1	2
g_1	4	1	2	5	1
g_2	6	1	3	8	1

V. QUERY PROCESSING

In this section, we give the query method of *q-gram* matrix index stored in hybrid layout in external memory.

Let $w(h)$ be the *q-gram* query vector of h and y_i be the number of common *q-grams* between g_i and h, thus we have $y_i = \sum_{j=1}^{|\mathcal{U}|} \min\{w_j(g_i), w_j(h)\} = w(g_i) \otimes w(h)$, where $\otimes$ is a generalized min operator, defined as follows:

$$a \otimes b = \begin{cases} \min\{a, b\} & \text{if } a \text{ and } b \text{ are integers;} \\ \sum_j a_j \otimes b & \text{if } b \text{ is an integer and} \\ & \quad a \text{ is a vector;} \\ \sum_j a_j \otimes b_j & \text{if both } a \text{ and } b \text{ are vectors.} \end{cases}$$

Let $Y_i = \max\{|Q(g_i)| - \gamma_{g_i} \cdot \tau, |Q(h)| - \gamma_h \cdot \tau\}$, where $Q(g_i)$ and $Q(h)$ denote the multi-sets of *q-grams* in g_i and h, respectively, γ_{g_i} and γ_h are the respective maximum number of *q-grams* that can be affected by an edit operation. According to the principle of the *q-gram* counting filter [5], [20]: if $ged(g_i, h) \le \tau$, graphs g_i and h must share at least Y_i common *q-grams*. Thus, it must satisfy $y_i \ge Y_i$ when $ged(g_i, h) \le \tau$. So, the matrix representation of the *q-gram* counting filter is given as follows:

$$W \otimes w(h) \geq Y \tag{2}$$

where $W \otimes w(h) = [y_1, y_2, \cdots, y_n]$ and $Y = [Y_1, Y_2, \cdots, Y_n]$. Applying formula (2) to the branch-based *q-gram* counting filter and the label-based *q-gram* counting filter, respectively, we obtain their corresponding matrix representation as follows:

$$W_L \otimes w_L(h) \geq Y_L$$
$$W_B \otimes w_B(h) \geq Y_B$$

where $Y_L[i] = \max\{|V_{g_i}|, |V_h|\} + \max\{|E_{g_i}|, |E_h|\} - \tau$ and $Y_B[i] = \max\{|V_{g_i}| - 2\tau, |V_h| - 2\tau\}$. $w_L(h)$ and $w_B(h)$ are the label-based and the branch-based *q-gram* query vectors of h, respectively. For the graphs g_1 and g_2 shown in Figure 1, $W_B = \begin{pmatrix} 4 & 1 & 1 & 1 \\ 6 & 1 & 2 & 1 \end{pmatrix}$ and $w_B(h) = [3\ 1\ 0\ 1]$ by Table 1, we have $W_B \otimes w_B(h) = [5, 5]$. If $\tau = 2$, we can get $Y_B = [3, 6]$ and then filter g_2 out. Similarly, $W_L \otimes w_L(h) = [9, 9]$ and $Y_L = [11, 17]$, then we can filter g_1 out. Thus none of graphs pass the hybrid *q-gram* filter in this example.

We generally assume graphs to be sparse and so is the corresponding matrix W. For instance, we sample 10 million graphs at random from PubChem, and count the number of zero entries in each row of W_B and W_L. The result shows that more than 90% entries in each row of both W_B and W_L are zeros. The sparsity can be also observed from other studies such as those from NCI/NIH whose graph databases include millions of molecular structures with tens of nodes in each graph [13]. Therefore, with a sparse W, the *q-gram* counting filter that computes the number of common *q-grams* can be converted into a sparse matrix-vector multiplication (SpMV) problem: $W \otimes w(h) \geq Y$.

We use a list of triples (i, j, W_{ij}) to store the nonzero entries W_{ij} at position (row i, column j). The order of this list corresponds to the layout of the matrix in main memory. Compared with the transaction graph database and the *q-gram* matrix W, $w(h)$ and Y are typically small, thus we store them in main memory.

A. QUERY WITH COLUMN-MAJOR LAYOUT

Let $x = w(h)$ be the *q-gram* query vector of the query graph h. When we store W in column-major layout, only the nonzero entries in the jth ($x_j \neq 0$) column of W are needed in computation, since when $x_j = 0$, $W_{ij} \otimes x_j$ equals to zero that has no contribution to y_i ($y = W \otimes x$). For instance, the branch-based *q-gram* vector $w_B(h) \neq 0$ for $j = 1, 2,$ and 4, namely only the nonzero entries of the columns 1, 2, and 4 of W_B are needed in computation.

Let $I = \{j : x_j \neq 0\}$ be the set of nonzero entries of x, we maintain an array y of size n in main memory to compute $W \otimes x$, where y_r stores the sum of all nonzero entries W_{rc} such that $c \in I$. The query algorithm is shown in Algorithm 1, where W is the *q-gram* matrix stored in external memory in column-major layout and x is the *q-gram* query vector resided in main memory. Y is the *q-gram* vector that stores the least number of common *q-grams* between graph g_i and h, resided in main memory.

Algorithm 1 *QMatrix*-C(W, x, Y)

```
Input: $W, x, Y$
Output: $Cand = \{g_i : W(i, .) \otimes x \geq Y_i\}$
1  $Cand \leftarrow \emptyset, I \leftarrow \{j : x_j \neq 0\}$
2  $y[1..n] \leftarrow 0$
3  for $j \leftarrow 1$ to $|I|$ do
4      $left \leftarrow N_{I_j}$
5      while $left > 0$ do
6          Read $\min\{B, left\}$ triples $(r, c, W_{rc})$
           into main memory, s.t, $c = I_j$
7          $y_r \leftarrow y_r + \min\{W_{rc}, x[I_j]\}$
8          $left \leftarrow left - \min\{B, left\}$
9  for $i \leftarrow 1$ to $n$ do
10     if $y_i \geq Y_i$ then
11         $Cand \leftarrow Cand \cup \{g_i\}$
12 return $Cand$
```

In Algorithm 1, N_{I_j} denotes the total number of nonzero triples in column I_j and *left* denotes the number of nonzero triples have not been read into main memory in column I_j. We first read the triples (r, c, W_{rc}) in column I_j (i.e., $c = I_j$) into main memory to update the sum y_r in lines 3–8, and then determine whether a graph g_i is a candidate or not by $y_i \geq Y_i$ in lines 9–11.

B. QUERY WITH HYBRID LAYOUT

We can also store W in row-major layout and sequentially read all nonzero entries to directly perform an update $y_i = y_i + W_{ij} \otimes x_j$. Compared with row-major layout, the column-major layout can skip many columns of W, avoiding all triples to participate in computation. However, the column-major layout cannot properly support the random access of the triples of a given graph, since these triples are scattered on different disk blocks. In order to reduce the number of query I/Os, we have the following two key observations: (i) not all the triples in W are necessary in computation, especially for those of graphs do not meet the edit distance constraint. (ii) the distribution of nonzero entries in W may not be uniform, such as the occurrence of c-c structure in most of chemical structure makes the corresponding column very dense. This will lead to the number of nonzero entries in the dense part occupies the vast majority of the total number of nonzero entries. For the first case, we store the sparse part W_S of W in column-major layout and filter the graphs that do not meet the edit distance constraint to obtain a temporary graph candidate C_s. For the second case, we store the dense part W_D in row-major layout and filter the graphs in C_s to obtain the final candidate set *Cand*.

We divide the *q-gram* universal set $\mathcal{U}$ into two disjoint subsets D and S. If $\mathcal{U}_i \in D$, the entries in the column i of W are in the dense part. Otherwise, they are in the sparse part.

Correspondingly, we divide W into two parts: the dense part W_D and the sparse part W_S such that $W = [W_D, W_S]$ and so as the query vector $x = [x_D, x_S]$. Therefore, $W \otimes x = [W_D, W_S] \otimes [x_D, x_S] = W_D \otimes x_D + W_S \otimes x_S$. By the definition of $\otimes$, $W_D(i, .) \otimes x_D = \sum_{\mathcal{U}_j \in D} \min\{W_{ij}, x_j\} \leq \sum_{\mathcal{U}_j \in D} x_j$, namely, $W_D \otimes x_D \leq I_D$, where $I_D[i] = \sum_{\mathcal{U}_j \in D} x_j$. If we compute $W_S \otimes x_S \geq Y - I_D$ to obtain the temporary graph candidate C_s, then we have $Cand \subseteq C_s$. The reason is as follows: for any graph $g \in Cand$, it must satisfy $y_i = W(i, .) \otimes x = W_D(i, .) \otimes x_D + W_S(i, .) \otimes x_S \geq Y_i$, thus $W_S(i, .) \otimes x_S \geq Y_i - W_D(i, .) \otimes x_D \geq Y_i - I_D[i]$, and hence we have $g_i \in C_s$. So we can filter the graphs in C_s to obtain the final candidate set $Cand$ without pruning those graphs satisfying the graph edit distance constraint.

There exists $2^{|\mathcal{U}|}$ ways to divide $\mathcal{U}$ into two disjoint subsets D and S. We use a simple partition here. First, we sort W by the number of nonzero entries of each column, such that $\delta(i_1) \leq \delta(i_2) \leq \ldots \leq \delta(i_{|\mathcal{U}|})$, where $\delta(i_j)$ denotes the number of nonzero entries in column i_j of W, $1 \leq i_j \leq |\mathcal{U}|$ and $i_1, i_2, \ldots, i_{|\mathcal{U}|}$ is a permutation of $1, 2, \ldots, |\mathcal{U}|$. Then $\mathcal{U}$ is divided by columns into two parts: $D = \{\mathcal{U}_{i_k} : k \geq \alpha|\mathcal{U}| + 1\}$, and $S = \mathcal{U} - D$, where α is referred to as dense factor.

The query algorithm on hybrid layout is shown in Algorithm 2, where W_S is the sparse part of W stored in column-major layout, and W_D is the dense part of W stored in row-major layout, x is the query vector and Y is the vector that stores the least number of common *q-grams* resided in main memory.

```
Algorithm 2 QMatrix-SR($W_S, W_D, x, Y$)
   Input: $W_S, W_D, x, Y$
   Output: $Cand = \{g_i : W(i, .) \otimes x \geq Y_i\}$
 1 $Cand \leftarrow \emptyset$
 2 $[x_D, x_S] \leftarrow x$
 3 $I_D[1..n] \leftarrow \sum_{\mathcal{U}_j \in D} x_j$
 4 $[y_S, C_S] \leftarrow$ QMatrix-C($W_S, x_S, Y - I_D$)
 5 for $g_i \in C_S$ do
 6     Read nonzero entries of $g_i$ in $W_D$ into memory
 7     $y_D[i] \leftarrow W_D(i, .) \otimes x_D$
 8     if $y_D[i] + y_S[i] \geq Y_i$ then
 9         $Cand \leftarrow Cand \cup \{g_i\}$
10 return Cand
```

In Algorithm 2, we first divide the query vector x into two parts x_D and x_S in line 2, and then use Algorithm 1 to obtain the temporary candidate set C_s in line 4. Finally, we read the nonzero entries of graphs in C_s to obtain $Cand$ in lines 5–9.

When both W_B and W_L are used to filter, the naive method is to use W_B and W_L to obtain the branch-based candidate set C_B and the label-based candidate set C_L by Algorithm 2, respectively, thus the candidate set $Cand = C_B \cap C_L$. However, we can use the obtained temporary candidate set to reduce the number of query I/Os as follows. First, we obtain the temporary candidate sets C_B^S and C_L^S by the sparse parts W_B^S of W_B and W_L^S of W_L, respectively. Then, we read the nonzero entries of graphs in $C_B^S \cap C_L^S$ in W_L^D to obtain the label-based candidate set C_L. Finally, we read the nonzero entries of graphs in C_L in W_B^D to obtain $Cand$, where W_B^D and W_L^D are the dense parts of W_B and W_L, respectively.

C. QUERY ALGORITHM

Algorithm 3 (*QMatrix*-MSR) gives the whole query algorithm, where $W_D^{i,j}$ is the dense part of the *q-gram* matrix index $W^{i,j}$ corresponding to subregion $R_{i,j}$, and $W_S^{i,j}$ is the sparse part of $W^{i,j}$. $x^{i,j}$ is the *q-gram* query vector of h, and $Y^{i,j}$ stores the least number of common *q-grams*.

```
Algorithm 3 QMatrix-MSR($h, \tau, l, x_0, y_0$)
   Input: $h, \tau, l, x_0, y_0$
   Output: Cand
 1 $Cand \leftarrow \emptyset$
 2 $Q_h \leftarrow \cup_{i,j} R_{i,j}$ for all $i_1 \leq i \leq i_2$ and $j_1 \leq j \leq j_2$
 3 foreach $R_{i,j} \subseteq Q_h$ do
 4     $C_{i,j} \leftarrow$ QMatrix-SR($W_S^{i,j}, W_D^{i,j}, x^{i,j}, Y^{i,j}$)
 5     $Cand \leftarrow Cand \cup C_{i,j}$
 6 return Cand
```

For a query graph h, we first compute the query region Q_h in line 2 using formula 1, where $i_1 = \lfloor(|V_h| - \tau - (x_0 - l/2))/l\rfloor$, $j_1 = \lfloor(|E_h| - \tau - (y_0 - l/2))/l\rfloor$, $i_2 = \lfloor(|V_h| + \tau - (x_0 - l/2))/l\rfloor$ and $j_2 = \lfloor(|E_h| + \tau - (y_0 - l/2))/l\rfloor$. Then, we only need to use the matrices corresponding to the subregions $R_{i,j}$ such that $R_{i,j} \subseteq Q_h$, to obtain the candidate set $Cand$ in lines 3–5.

Query I/O Complexity. Given a *q-gram* matrix W with N nonzero entries storing in hybrid layout, the query region Q_h might contain all points in R, namely, in the worst case all graphs are needed in the query. Thus, the query I/O complexity of *QMatrix*-MSR is $O(N/B)$.

VI. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of our proposed method and compare it with κ-AT [11], GSimJoin [20], C-Star [18], and Mixed [12] on two real datasets. The efficiency of our method in external memory is evaluated on the large PubChem dataset. We randomly select 50 graphs from each dataset as its query graphs.

A. DATA SETS AND SETTINGS

We choose two publicly available real datasets in our experiment, described as follows.

(1) AIDS.[1] It is an antivirus screen compound dataset from the Development and Therapeutics Program in NCI/NIH to discover compounds capable of inhibiting the HIV virus, which contains 42,687 chemical compounds. We generate the labeled graphs from these chemical compounds and omit

[1]`http://dtp.nci.nih.gov/docs/aids/aidsdata.html`

Hydrogen atoms as did in [15]. It has an average number of 25 vertices and 27 edges.

(2) PubChem.[2] It is a National Institute of Health (NIH) funded project to record experimental data of chemical interactions with biological systems in NIH. It contains more than 50 million chemical compounds and records their biological activities until today. We also follow the same procedure as did in [15] to transform the chemical compounds to labeled graphs. The average number of vertices and edges are 23 and 25, respectively.

We conducted all experiments on a HP Z800 PC with a 2.67 GHz CPU and 24GB memory, running Ubuntu 12.04 operating system. We implemented our algorithm in C++, with $-O3$ to compile and run. To ensure our index being maintained in external memory during query, we ran the shell command: "$sh\ -c\ sync\ \&\&\ echo\ 3 > /proc/sys/vm/drop\_caches$" to clear the in-memory cache data before each query. We set the simple path $p = 4$ in GSimJoin and $\kappa = 1$ in κ-AT, which are the recommended values [21]. We set disk block size $B = 4$KB, subregion length $l = 2$, and dense factor $\alpha = 0.06$. In the following sections, we refer LBMatrix to our index structure.

B. EVALUATING TRANSFORMATION

In this section, we randomly select 25 million data graphs from PubChem, and vary τ from 1 to 5 to evaluate our proposed transformation in Section III-B. We use Basic LBMatrix to denote the basic implementation of the branch-based and the label-based *q-gram* matrices, both of which are built on the whole region. Figure 4 presents the average number of query I/Os and total filtering time for the fifty query graphs.

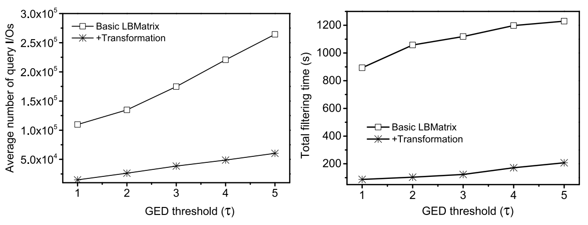

FIGURE 4. Average query I/Os and total filtering time on PubChem.

Compared with Basic LBMatrix, the number of query I/Os required for +Transformation reduces by more than 80%. Regarding the filtering time, +Transformation can achieve 6.5x speedup on the average. Thus, the transformation can greatly reduce the number of query I/Os.

In addition, we fix $\tau = 3$ and vary the subregion length l from 1 to 5, to evaluate the effect of l on the query performance. Figure 5 shows the average number of query I/Os and total filtering time. We know for sure that the average number of I/Os first decreases and then increases. This is because that: (1) Small l will produce too many subregions, making each of them only contains few graphs, thus most of disk

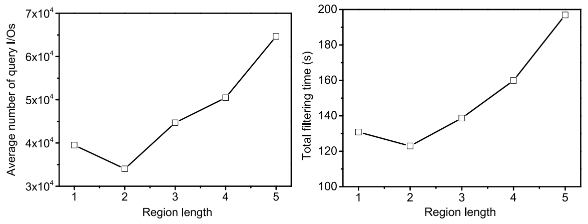

FIGURE 5. Average number of query I/Os and total filtering time on PubChem.

blocks storing the triples are not full. This will lead to more query I/Os. (2) Large l will produce a large query region Q_h, and hence also needs more query I/Os.

C. EVALUATING HYBRID LAYOUT

To evaluate the effectiveness of our proposed hybrid layout, we randomly select 25 million data graphs from PubChem and compare it with the other two matrix layouts, i.e., row-major and column-major layout.

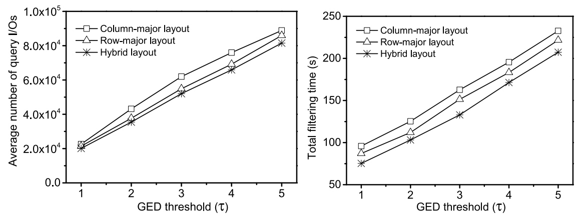

FIGURE 6. Average number query I/Os and total filtering time on PubChem.

Figure 6 shows the average number of query I/Os and total filtering time. By Figure 6, we can see that the column-major layout needs the most I/Os. This is because that the column-major layout do not support random access of triples, making the temporary candidate set C_L obtained by computing $W_L \otimes w_L(h) \geq Y_L$ cannot be used to reduce the number of query I/Os in computation of $W_B \otimes w_B(h) \geq Y_B$. Hybrid layout achieves a reduction of I/Os over column-major layout by 15% and 1.2x speedup in filtering time on the average.

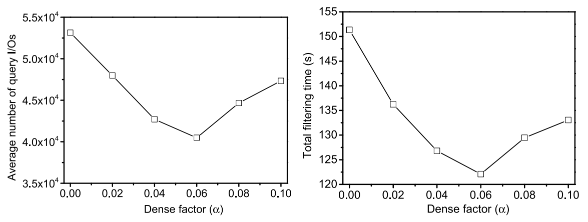

FIGURE 7. Average number query I/Os and total filtering time on PubChem.

In addition, we fix $\tau = 3$ and vary the dense factor α from 0 to 0.1, to evaluate the effect of α on the query performance. Figure 7 plots the average number of query I/Os and total filtering time. The average number of I/Os required by hybrid layout first decreases and then increases, and achieves the minimum when $\alpha = 0.06$. There are several factors contributing to this trend: (1) Small α indicates that more columns

[2]http://pubchem.ncbi.nlm.nih.gov/

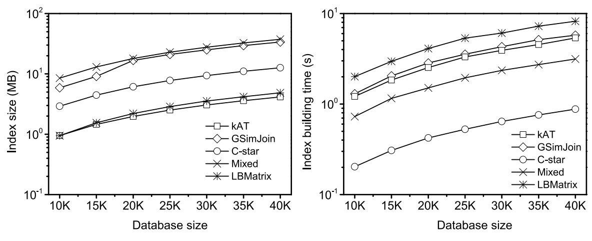

FIGURE 8. Building time and index size on AIDS.

are stored in column-major layout. This will lead to more query I/Os involved in computation of the temporary candidate set C_s by using W_S. (2) Large α indicates that more columns are stored in row-major layout. This will produce more temporary candidates, thus more query I/Os are needed in computation of the final candidate set *Cand* by using W_D.

D. COMPARING WITH IN-MEMORY METHODS

In this subsection, we compare LBMatrix with the state-of-the-art in-memory graph similarity search methods, including κ-AT, C-Star, GSimJoin and Mixed.

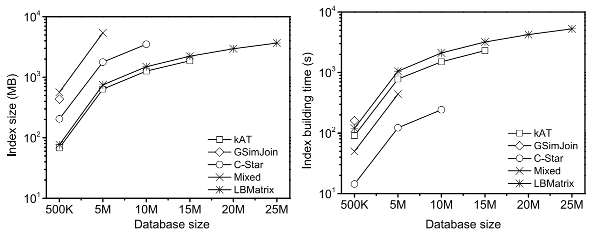

FIGURE 9. Building time and index size on PubChem.

1) EVALUATING INDEX CONSTRUCTION

We vary the size of datasets to evaluate the index construction performance of the above methods on the small dataset AIDS and large dataset PubChem, and show the results in Figure 8 and Figure 9.

Regarding the index size, κ-AT has the least storage cost. Mixed does not perform well, since it builds a U-tree where the internal nodes contain the information of the leaf nodes. Since we only need to store the nonzero entries of W_B and W_L, the index size of LBMatrix is smaller than GSimJoin, C-Star and Mixed. For the large dataset PubChem, all tested methods that maintain the index structure in main memory cannot properly run when the database size is more than 15M, while LBMatrix resided in the external memory can easily scale to such large dataset. Note that, although κ-AT has a smaller index size than LBMatrix, its main memory consumption is larger than LBMatrix during index building. This is because that the index structure of κ-AT has been residing in main memory during index building, while LBMatrix continually write the nonzero entries into disk blocks to reduce the main memory consumption.

Among all methods, C-Star performs best in index building time for it only needs to enumerate all star structures in each data graph without any complex index. LBMatrix has the longest index building time since it is an external index structure, which needs I/Os during index building.

2) EVALUATING FILTERS

For the small dataset AIDS, we vary threshold τ from 1 to 5 to evaluate the filter efficiency and ability. Figure 10 shows the average candidate size and total response time (i.e., the filtering time plus the verification time) for the fifty query graphs of κ-AT(denoted by "T"), C-Star(denoted by "C"), GSimJoin(denoted by "P"), Mixed(denoted by "M") and LBMatrix(denoted by "L"), where the line labeled with triangle gives the known empirical lower bound.

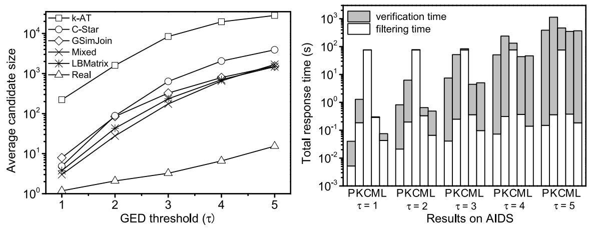

FIGURE 10. Average candidate size and total response time on AIDS.

Regarding the candidate size, Mixed has the smallest candidate size and shows the best filtering ability among all tested methods. κ-AT does not perform well for large τ, because there exists much more overlapping structures among its *q-grams*. LBMatrix has a close candidate size with Mixed, and performs better than GSimJoin, κ-AT, and C-Star. For the response time, Mixed has the shortest time in most case. C-Star performs occasionally worst because of its cost to construct the bipartite graph between each data graph and the query graph. Although LBMatrix is an external index structure, it performs better than GSimJoin, κ-AT, and C-Star in most case.

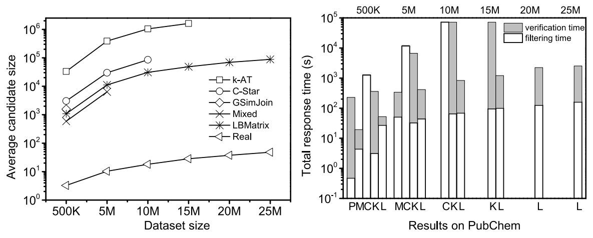

FIGURE 11. Average candidate size and total response time on PubChem.

For the large dataset PubChem, we fix $\tau = 3$ and vary the size of PubChem from 500K to 25M to evaluate the query performance of all tested methods, and show the results in Figure 11. Among all tested methods, Mixed has the smallest candidate size and the shortest response time when the database size is less than 5M. However, when the database size is 10M, both Mixed and GSimJoin cannot properly run for the memory error, and both the filtering time of C-Star and the verification time of κ-AT are longer than 24 hours, making all of them be unsuitable for such large dataset. LBMatrix can easily scale to it and obtain the required graphs in 1 hour.

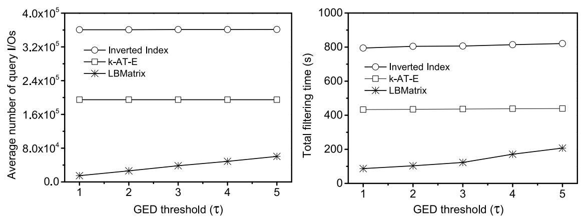

FIGURE 12. The average query I/Os and total filtering time on PubChem.

E. COMPARING WITH EXTERNAL MEMORY METHODS

In this section, we compare LBMatrix with κ-AT-E and LB-Inverted on the large PubChem dataset, where κ-AT-E and LB-Inverted are the *q-gram*-based external inverted indexes.

We first give a brief description of the implementation of the *q-gram*-based external inverted index as follows. For each *q-gram* u_j, we store a inverted list containing the tuples $(i, w_j(g_i))$, where $w_j(g_i)$ is the number of occurrence of *q-gram* u_j in g_i. The tuples in a inverted list are stored in continuous disk blocks. For κ-AT-E, we use the 1-*adjacent* subtrees as *q-grams* recommended in practice in [21] to construct the tree-based *q-gram* external inverted index. For LB-Inverted, we construct the branch-based *q-gram* external inverted index and label-based *q-gram* external inverted index, respectively.

For κ-AT-E, we query the tree-based *q-gram* inverted index to obtain the candidate set *Cand*. For LB-Inverted, we first query the branch-based *q-gram* inverted index to obtain the candidate set C_B, and then query the label-based *q-gram* inverted index to obtain the candidate set C_L, thus the final candidate set $Cand = C_B \cap C_L$. The query method on the *q-gram*-based external inverted index is similar with Algorithm 1 in Section V-A.

We randomly select 25 million data graphs from PubChem, and vary τ from 1 to 5 to evaluate the performance of the above three external index structures, i.e., LBMatrix, κ-AT-E and LB-Inverted. Figure 12 shows the average number of query I/Os and total filtering time for the fifty query graphs.

Among all methods, LBMatrix needs the least number of query I/Os and shortest filtering time. Compared with LB-Inverted, the number of query I/Os can be reduced by 85% on the average. Both κ-AT-E and LB-Inverted have a worse performance possibly for the following two reasons: (1) The query graph may contain most of the "dense" *q-grams*, leading to most of tuples $(i, w_j(g_i))$ involved in the computation. (2) All graphs are involved in the query. In addition, we also observe that the number of query I/Os required for LB-Inverted is greater than that for κ-AT-E. This is because that both the branch-based and label-based *q-gram* inverted indexes are used to obtain the candidate set in LB-Inverted, making the number of tuples involved in the query of LB-Inverted are greater than that needed in the query of κ-AT-E. The difference of the number of query I/Os between LBMatrix and κ-AT-E gets smaller under large τ setting, because the query region Q_h grows larger. Regarding the filtering time, LBMatrix can achieve 3.5x speedup compared with κ-AT-E and 6.5x speedup compared with LB-Inverted on the average.

VII. CONCLUSIONS

In the paper, we study the problem of graph similarity search under edit distance constraints in the I/O model. Unlike previous methods, our index structure works well with big data applications in an external memory setting. We build the *q-gram* matrix and convert the *q-gram* counting filter into a sparse matrix-vector multiplication problem to seek for an efficient query method. In addition, the transformation of the global filter into a two-dimensional query rectangle allows us to preform the query in a reduced region, which significantly reduces the number of query I/Os in practice. Comprehensive experiments on real data sets demonstrate that our method outperforms the state-of-the-art methods.

ACKNOWLEDGMENTS

The authors would like to thank Weiguo Zheng and Lei Zhou for providing their source files, and thank Xiang Zhao and Xuemin Lin for providing their executable files.

REFERENCES

[1] A. Aggarwal and J. Vitter, "The input/output complexity of sorting and related problems," *Commun. ACM.*, vol. 31, no. 9, pp. 1116–1127, 1988.
[2] M. A. Bender, G. S. Brodal, R. Fagerberg, and R. Jacob, "Optimal sparse matrix dense vector multiplication in the I/O-model," *Theor. Comput. Syst.*, vol. 47, no. 4, pp. 934–962, 2010.
[3] M. L. Fernández and G. Valiente, "A graph distance metric combining maximum common subgraph and minimum common supergraph," *Pattern Recognit Lett.*, vol. 22, nos. 6–7, pp. 753–758, May 2001.
[4] H. Fröhlich, J. K. Wegner, and F. Sieker, "Optimal assignment kernels for attributed molecular graphs," in *Proc. ICML*, 2005, pp. 225–232.
[5] K. Gouda and M. Arafa, "An improved global lower bound for graph edit similarity search," *Pattern Recognit Lett.*, vol. 58, pp. 8–14, Jun. 2015.
[6] W. S. Han, J. Lee, M. D. Pham, and J. X. Yu, "iGraph: A framework for comparisons of disk-based graph indexing techniques," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 449–459, 2010.
[7] J.-W. Hong and H. T. Kung, "I/O complexity: The red-blue pebble game," in *Proc. STOC*, 1981, pp. 326–333.
[8] A. Robles-Kelly and E. R. Hancock, "Graph edit distance from spectral seriation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 3, pp. 365–378, Mar. 2005.
[9] Y. Tian and J. M. Patel, "TALE: A tool for approximate large graph matching," in *Proc. ICDE*, Apr. 2008, pp. 963–972.
[10] J. S. Vitter, "Algorithms and data structures for external memory," *Found. Trends Theor. Comput. Sci.*, vol. 2, no. 4, pp. 305–474, 2008.
[11] G. Wang, B. Wang, X. Yang, and G. Yu, "Efficiently indexing large sparse graphs for similarity search," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 3, pp. 440–451, Mar. 2012.
[12] X. Wang, X. Ding, A. K. H. Tung, S. Ying, and H. Jin, "An efficient graph indexing method," in *Proc. ICDE*, Apr. 2012, pp. 210–221.
[13] X. Wang, J. Huan, A. Smalter, and G. H. Lushington, "Application of kernel functions for accurate similarity search in large chemical databases," *BMC Bioinformat.*, vol. 11, no. 3, p. 1, 2010.
[14] X. Wang, A. Smalter, J. Huan, and G. H. Lushington, "G-hash: Towards fast kernel-based similarity search in large graph databases," in *Proc. EDBT*, 2009, pp. 472–480.
[15] D. W. Williams, J. Huan, and W. Wang, "Graph database indexing using structured graph decomposition," in *Proc. ICDE*, Apr. 2007, pp. 976–985.
[16] X. Yan, P. S. Yu, and J. Han, "Graph indexing: A frequent structure-based approach," in *Proc. SIGMOD*, 2004, pp. 335–346.

[17] X. Yan, F. Zhu, P. S. Yu, and J. Han, "Feature-based similarity search in graph structures," *ACM Trans. Database Syst.*, vol. 31, no. 4, pp. 1418–1453, 2006.
[18] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou, "Comparing stars: On approximating graph edit distance," *Proc. VLDB Endowment*, vol. 2, no. 1, pp. 25–36, 2009.
[19] X. Zhao, C. Xiao, X. Lin, Q. Liu, and W. Zhang, "A partition-based approach to structure similarity search," *Proc. VLDB Endowment*, vol. 7, no. 3, pp. 169–180, 2013.
[20] X. Zhao, C. Xiao, X. Lin, and W. Wang, "Efficient graph similarity joins with edit distance constraints," in *Proc. ICDE*, Apr. 2012, pp. 834–845.
[21] W. Zheng, L. Zou, X. Lian, D. Wang, and D. Zhao, "Efficient graph similarity search over large graph databases," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 4, pp. 964–978, Apr. 2015.

XIAOYANG CHEN received the B.S. degree from Xidian University in 2013, where he is currently pursuing the Ph.D. degree. His research interests include graph indexing and search, design and analysis of algorithms, external memory algorithms, and compressed indexes.

HONGWEI HUO (M'00) received the B.S. degree in mathematics from Northwest University, China, and the M.S. degree in computer science and the Ph.D. degree in electronic engineering from Xidian University, China. She is currently a Professor with the Department of Computer Science, Xidian University. Her research interests include the design and analysis of algorithms, bioinformatics algorithms, external memory algorithms and compressed indexes, parallel and distributed algorithms, and algorithm engineering.

JUN HUAN (SM'11) received the Ph.D. degree in computer science from the University of North Carolina. He is currently a Professor with the Department of Electrical Engineering and Computer Science, the University of Kansas. He directs the Data Science and Computational Life Sciences Laboratory, KU Information and Telecommunication Technology Center. He holds courtesy appointments at the KU Bioinformatics Center, the KU Bioengineering Program, and a Visiting Professorship from GlaxoSmithKline plc. He has authored over 120 peer-reviewed papers in leading conferences and journals and has graduated more than ten graduate students including seven PhDs. His research works on data science, machine learning, data mining, big data, and interdisciplinary topics including bioinformatics. He serves the Editorial Board of several international journals, including the *Springer Journal of Big Data*, the *Elsevier Journal of Big Data Research*, and the *International Journal of Data Mining and Bioinformatics*. He regularly serves the Program Committee of top-tier international conferences on machine learning, data mining, big data, and bioinformatics.

JEFFREY SCOTT VITTER (F'93) received the B.S. degree (Hons.) in mathematics from the University of Notre Dame in 1977, the Ph.D. degree in computer science from Stanford University in 1980, and an M.B.A degree from Duke University in 2002. He is currently a Chancellor and a Distinguished Professor of Computer and Information Science with the University of Mississippi. From 2010 to 2015, he was a Provost and Executive Vice Chancellor and a Roy A. Roberts Distinguished Professor with the University of Kansas. From 2008 to 2010, he was a Faculty Member with Texas A&M University, where he also served as a Provost and Executive Vice President for academics. From 2002 to 2008, he was the Frederick L. Hovde Dean of Science with Purdue University. From 1993 to 2002, he held the Gilbert, Louis, and Edward Lehrman Distinguished Professorship with Duke University, where he also served as a Chair of the Department of Computer Science and a Co-Director of Duke's Center for Geometric and Biological Computing. From 1980 to 1992, he advanced through the faculty ranks in computer science at Brown University. He is a fellow of Guggenheim, ACM, and AAAS, and a U.S. National Science Foundation Presidential Young Investigator and a Fulbright Scholar. He has authored the book *Algorithms and Data Structures for External Memory*, co-authored the books *Design and Analysis of Coalesced Hashing* and *Efficient Algorithms for MPEG Video Compression*, co-edited the collections *External Memory Algorithms* and *Algorithm Engineering*, and co-holder of patents in the areas of external sorting, prediction, and approximate data structures. His research interests span the design and analysis of algorithms, external memory algorithms, data compression, databases, compressed data structures, parallel algorithms, machine learning, random variate generation, and sampling. He has received the IBM Faculty Development Award, the ACM Recognition of Service Award (twice), and the 2009 ACM SIGMOD Test of Time Award. He has served on several boards and professional committees. He serves or has served on the Editorial Board of the *Algorithmica*, the *Communications of the ACM*, the IEEE TRANSACTIONS ON COMPUTERS, Theory of Computing Systems, and the *SIAM Journal on Computing*, and has edited several special issues.