Received December 30, 2019, accepted January 17, 2020, date of publication February 4, 2020, date of current version February 17, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2971639

Graph Compression Storage Based on Spatial Cluster Entity Optimization

DAWEI WANG[1], **WANQIU CUI**[2], **AND BIAO QIN**[1]
[1]School of Information, Renmin University of China, Beijing 100872, China
[2]School of Computer Science, Beijing University of Posts and Telecommunications, Beijing 100876, China

Corresponding author: Biao Qin (qinbiao@ruc.edu.cn)

This work was supported by the 2018 Outstanding China-Foreign Youth Exchange Program of the China Association for Science and Technology under Grant 311319000207.

ABSTRACT Graph storage technology is confronted with an enormous challenge as far as the compact and complex graph-structure data. This phenomenon is derived from social networks with spatially intensive data. Since a hot event can cause the generation of a network cluster, which consists of a massive duplicate associated entities in the social networks, the space utilization and processing speed of graph data is obstructed. Therefore, it is necessary to design a graph storage mechanism specifically for the above data. In this paper, we propose a **G**raph compression **S**torage engine based on spatial **C**luster entity **O**ptimization (**GSCO**), which improves the native graph storage model through the proposed the many-to-one mapping structure and a **H**eat **E**volution **E**limination algorithm (**H2E**). Firstly, we define the spatial cluster entity formally and confirm the compressed storage objects. Then, we introduce the many-to-one relationship to transfer the mapping structure between the node and property. It compresses the data to raise the space utilization of the graph database. Finally, we propose the H2E algorithm that allows the representative nodes to be anchored an extended period in memory according to the heat evolution acceleration. It increases the hit rate and throughput and reduces the I/O operation by deleting the redundancy of data. Extensive experiments results show that the proposed GSCO storage model is better than Neo4j for reading and writing data in spatial clustering entity. It significantly promotes the effectiveness of graph operation, including the data loading, the common queries, and the clustering test.

INDEX TERMS Graph compression storage, social networks, spatial cluster entity, heat evolution.

I. INTRODUCTION

Graph database has been extensively studied and applied with the emergence of abundant relational data such as knowledge mapping and social networks. Specifically, Neo4j [1] and Titan [2] are adopted by top enterprises like eBay, Cisco. FlockDB [3] is developed and used by Twitter. The graph storage model is an indispensable module and business in the construction of the graph database. It surfaces a tremendous challenge when dealing with space-intensive clustered data. The spacial clusters are usually highly correlated data incurred by the widespread dissemination of hot events on social networks, including reply, likes, and forwarding [4], etc. It sets off numerous interactions focused on the hot event and constructs an immense network that contains substantial duplicate data. The different popularity of events brings into a diverse density of graph structures in social networks. The spacial cluster structure caused by hot events is more concentrated than the usual events. It forms a spacial cluster of star-shaped topological in the discussion space of the topic [5], [6]. Although the network is often sparse ($|E| \ll |V|^2$), the neighborhood of the nodes is dense. Besides, the clustering coefficient and transitivity are also high. The structure produces considerable duplicate data that need to be handled in the graph database resulting in data redundancy and waste of storage space. Therefore, it is necessary to build a separate storage mechanism for such data in the graph database.

The associate editor coordinating the review of this manuscript and approving it for publication was Vlad Diaconita.

The goal of this paper is to design a new storage model in the graph database and compress the duplicate data presenting space-intensive in the social network. We eliminate redundant data to realize efficient utilization of storage space and improve related operational performance. Due to the characteristics of the social network data, the generous new data is created by the interaction behavior such as forwarding and comment, etc., which has a strong correlation with the original entity node. That is, the new node carries the partly content of the original node, but also has a different primary key from other entities. As a result, a large amount of duplicate data exists in the formation of a space-intensive structure in an interactive network. They are usually long text and frequently reused in social networks. The reason is that the nature of the semi-structured data. The structure of the data mixes with the content, and there is no noticeable difference between them [7]. Moreover, the operating speed of spatial clustering data is also greatly affected. How to discard redundant data to achieve compression, and reduce the I/O consumption caused by file loading is the main problem of this paper.

In order to store dense network data, the Neo4j graph database discards relational models. It first employs the native graph storage [8] with a labeled property graph model to store and process graph data. However, a native graph database does not recognize the demand for efficient storage toward clustered data that tends to be space-intensive in social networks. In the native graph storage engine, the data stores in a dynamic storage file that cannot be encoded as a property structure of the inline value. The dynamic storage file requires a pointer for further addressing and increases the path to read the property. Therefore, this model not only suffers from the problem of data redundancy but also slows down the speed of related operations as many associations and addressing. Moreover, the Hypergraphs model [9] allows the relationship to attach any number of nodes when storing the graph data, effectively reducing the number of relationships. However, the storage model is only suitable for the field of many-to-many ties. As a more generalized model, it needs to be more clearly defined and not easily tweaked while modeling. So it is not applied for the graph database based on the on-line transaction processing.

In this paper, we design a new graph storage engine GSCO. It optimizes native graph storage from two perspectives. On the one hand, we develop a many-to-one mapping relationship for the compression of data and structure. On the other hand, a heat evolution elimination algorithm is proposed to designate anchoring data in memory based on thermal evolution. Concretely, the model classifies the related data of hot events and extracts the spatially-intensive data in the social network as the spatial cluster entity. Then, we perform the compression of the original data by generating multiple representative nodes and the many-to-one mapping relationship. Finally, GSCO eliminates repetition data and improves the I/O speed by combining the novel mapping structure with the H2E algorithm.

Specifically, we overcome the existing obstacles to achieve two improvements. **1**) The storage space utilization. In the storage of the space-intensive data, the one-to-one relationships cause the repeated record of a substantial of the same data. We get rid of the limitation of the mapping structure and redesign many-to-one relations to map the same part of multiple nodes into the representative node for shared storage. We define the equivalence relationship in the entire data space and aggregate the public content to achieve the compression of redundant data. Moreover, we rate the access frequency and importance of nodes based on their priority and current popularity. The H2E algorithm dynamically allocates the anchor time for the data. Therefore, GSCO improves the utilization of storage space. **2**) Efficiency of graph operations. The abundance network structure of the space-intensive clusters increases the workload of the graph database, which can occupy the main system resources when traversing its nodes. We prune redundant data through a many-to-one relationship, which points the property storage files of a vast of nodes to the same representative node. The complexity of each clustering structure is compressed by mapping between the representative nodes and the remaining nodes. Therefore, we can perform traversal more efficiently in the graph model while reducing system resource consumption.

We evaluate the proposed GSCO model on a real-public dataset, which is collected from Sina Weibo and consists of 13 hot events. The extensive experimental results show that our GSCO model can achieve high space utilization in compressed storage and effectively improves the cache hit rate in the data read and query. Above these outperform, GSCO is significantly superior to the state-of-the-art Neo4j graph model adopting the labeled property graphs.

The contribution of this paper are summarized as follows:

- We propose GSCO, a graph storage engine through compression and shard cache optimization. GSCO can effectively storage spatial-intensive data caused by hot events in social networks, that raises the storage space utilization dramatically compared with Neo4j.
- We design a many-to-one relationship structure that enables the mapping between related sharing data and representative data in spatially cluster. It reduces redundancy caused by duplicate data and improves the space utilization of graph database.
- We develop a heat evolution elimination algorithm by defining the acceleration of heat evolution and the priority of representative nodes. Through the dynamic selection of the compression target and the timely update of memory, we greatly improve the memory hit ratio so as to realize more efficient and faster disk access.
- We conduct comprehensive experiments in a social network graph. The results indicate that the GSCO model delivers considerable performance on random element writes while being orders of magnitude faster on clustering operations compared with the Neo4j. We also demonstrate the H2E algorithm reaches high space utilization and the cache hit rate compared to the LRU algorithm using in Neo4j.

The rest of the paper is organized as follows. In section 2, we review the related work of graph storage. We propose the many-to-one mapping structure and H2E algorithm, as the two main components of GSCO in Section 3. In Section 4, we report and display broad-scale and multi-angle experimental results by comparing the state-of-the-art Neo4j storage model. Finally, we conclude the paper in Section 5.

II. RELATED WORK

In this section, we review the related works about graph storage technology. It divides into the graph storage model and graph compression method.

A. GRAPH STORAGE MODELS

As the main research directions of databases, data storage has received extensive attention. The traditional storage mechanisms include the storage model of the row, column, key [10], document and object exchange [11], etc. Besides, since complex relational data forms a network structure, a graph model is needed to the commitment to the management of rich related data [12]. Compared with other data models, the graph storage model can set different property sets for specific types of data.

The triplestores model [13] is derived from the Semantic Web. It is a data structure that contains the subject, predicate, and object. It can infer large-scale knowledge by connecting network resources with semantic tags. The graph databases can seem a superset of the simple triplestores, and the processed data only tends to be logically related in the triplestores model. Since it does not support polygon index adjacencies and its storage engine is not optimized for storing property graphs, the triplestores is not native graph databases. So the model is best at analysis. The hypergraph model is a generalized graph model in which a link (called a hyperedge) can be associated with any number of nodes [14]. The hypergraph model is suitable for many-to-many relationships [15], which visualizes the modeling of the problem and be more faithful to the details of the data.

The labeled property graph model consists of nodes, links, properties, and labels [16]–[18]. The node contains properties, which can act as a container. The property value can be in the form of any key-value pairs. At the same time, nodes can be labeled with one or more tags. The node structure is semantically linked by direction and name. Connections can also be used as containers to store properties. The native graph has an indexed adjacency.

The management of large-scale graph data is a tricky problem of graph computation and graph storage. It is involved in graph mining and analysis, including related subgraph matching, subgraph isomorphism testing, hypergraph matching, etc. There is a lot of discussion in [19]–[22]. Based on these studies, a large number of excellent group databases have been generated, such as Neo4j,[1] InfiniteGraph, FlockDB, OrientDB, Affinity, Allegro-Graph, HypergrophDB,[2] FranzInc, Titan,[3] Trinity, etc.

[1]https://neo4j.com/
[2]http://www.hypergraphdb.org/
[3]http://titan.thinkaurelius.com/

B. GRAPH COMPRESSION METHODS

With the increase of large-scale graph data, graph storage mechanisms have been devoted to research on graph compression techniques [23]–[25] to reduce the storage burden and improve the operation efficiency of graph data.

They encode a graph or its transitive closure into compact data structures via node ordering and document similarity, hosts [25], and linkage similarity [26], etc. Shrink [24] reduces the structure size of the graph and mainly considers the distance-query. Other studies aim at specific classes of operations, which mainly contain neighborhood queries [27]–[29], reachability queries [30], and path queries [26], etc. Furthermore, most of them need to decompression before querying the graph [1], [31]. Moreover, the query evaluation algorithms on original graphs have to be modified to answer queries in their compact structures. Therefore, existing methods are not universal and dose not consider the nature of the structure and content of the constantly changing graph data. However, we analyze the flexibility of dynamic memory occupation in this paper.

As an important implementation of the property graph model, Neo4j is excellent and widely adopted. Each property record stores up to four property blocks. Each property block contains property types, property index files, and property values. For each value of the property, the record contains a pointer or inline value to the dynamic store record. The property index file stores all the property names. The value of the property is given priority inline into the property store file [32]. However, it is dragged to a dynamic store when the value can not be encoded as an inline value. Due to the Neo4j is a one-to-one mapping between node and property, queries for such data require multiple accesses to one or more files which increases I/O operations and reduces throughput in Neo4j. This phenomenon has a direct relationship with the separation strategy of graph structure and property data. Therefore, we make improvements based on Neo4j. We fully use the superiority of the complete attribute and structure information of the graph provided by the label property graph. Then, we add a new mapping mechanism and memory replacement algorithm to achieve the compression storage of space-intensive graph data. Finally, we increase the memory utilization and accelerate the processing speed and throughput of graph data.

III. OUR MODEL

In this section, we firstly describe the problem statement. Then the definition of the spatial cluster entity is introduced. The main component of the graph data storage model GSCO is proposed, which consists of the many-to-one mapping structure and the designation of H2E algorithm.

The proposed GSCO aims to optimize the graph storage model when handling with space-intensive data, thereby improving the storage space utilization and operational efficiency. It first identifies the spatial cluster entities that can cause duplicate data. Then, GSCO extracts representative

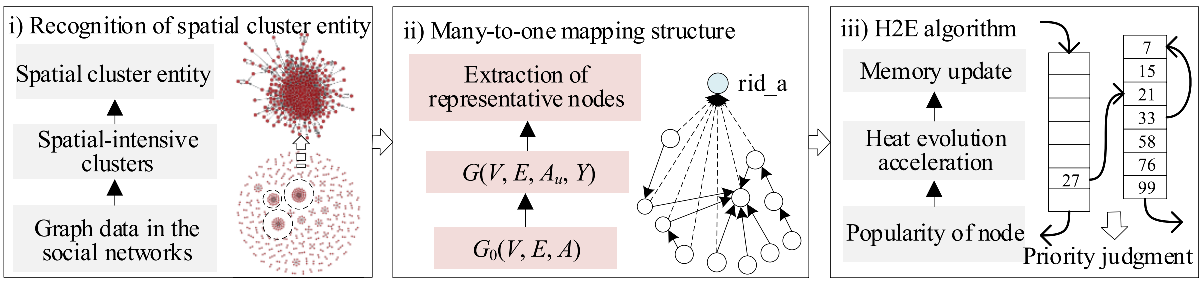

i) Recognition of spatial cluster entity
Spatial cluster entity
Spatial-intensive clusters
Graph data in the social networks
ii) Many-to-one mapping structure
Extraction of representative nodes
G(V, E, A_u, Y)
G_0(V, E, A)
rid_a
iii) H2E algorithm
Memory update
Heat evolution acceleration
Popularity of node
27
7
15
21
33
58
76
99
Priority judgment

FIGURE 1. **The framework of the proposed GSCO model.**

elements and designs a new many-to-one structure mapping. Finally, the H2E algorithm dynamically updates the memory according to the access popularity. Fig. 1 illustrates the framework of GSCO. It consists of three main components: the recognition of spatial cluster entity, many-to-one mapping structure, and H2E algorithm.

A. PROBLEM STATEMENT

Given microblog data with different hot events, we define the social networks graph structure as $G = (V, E, A_u, Y)$, where $V(G)$ is a set of message nodes, $E(G)$ is a set of edges which represents the forwarding relationship between nodes, $A_u(G)$ is the property in the form of a $key - value$ pair which has no duplicate data, and Y is the representative nodes. The property of nodes in Y is a shared content that is extracted from $A(G)$. We use $|V(G)|$, $|E(G)|$ and $|A(G)|$ to denote the number of nodes, edges and properties in G, respectively. For each node $u \in V(G)$, we use $N(u, G)$ to denote the neighborhood of u in G, which is $N(u, G) = \{v|(u, v) \in E\}$ [20]. $D(u, G)$ represents the degree of node $u \in V(G)$, that is, the number of neighbors of u in G, expressed as $D(u, G) = |N(u, G)|$. The G is a simple directed graph that has no self-looping and no parallel edges.

Since the nature of forwarding operations in social networks, the forwarding node contains a part of the same information as the original node. In the graph database Neo4j, all nodes are stored in the same way [8]. As a result, a lot of the same data that discusses the identical hot event is dumped into memory multiple times as different physical nodes. The structure formed by loading copies of the same content into the graph as multiple independent nodes contributes to maintaining the graph structure. To address this problem, we optimize the storage engine by redesigning the relationship between the node and the property storage, while ensuring the integrity of the graph structure and information.

Followed by the Neo4j, we store the independent storage files as $V(G)$, $E(G)$ and $A(G)$. This way ensures the separation of graph structure $G = (V, E)$ and property data $A(G)$. Among them, $A(G)$ is made up of a fixed-size record, which is referenced by a node $v \in V(G)$ and a relational record $e \in E(G)$. For each property value, the record contains a pointer to the inline value or the dynamic store record. In particular, we judge the data in $A(G)$ and elevate the dominance of data that cannot be encoded as inline values. This data acts as a node in $V(G)$, which in turn becomes a representative node Y_i, $Y_i \in Y$ to express the shared content contained in other properties in the G. The purpose is to compress the data node which contain the same content by many nodes projecting into a representative node, which denotes as $\{v_1, v_2, \ldots, v_n\} \rightarrow \{Y_1, Y_2, \ldots, Y_m\}$, where the $m \ll n$. Moreover, the H2E algorithm is designed and the hit rate is improved by caching and eliminating the spatial cluster entity according to the priority.

Concretely, we compress redundant data in forwarding nodes centered on the original node. By many-to-one structure mapping, we merge the multiple related data into several representative nodes that indicate the different hot events. To improve the efficiency of the operation of the graph, we choose the representative nodes with the most attention and long duration of heat to be anchored in memory for a long time. At the same time, we eliminate outdated information dynamically according to the priority of the node. Finally, space utilization and I/O efficiency are improved in our storage model.

B. SPATIAL CLUSTER ENTITY

According to the aggregation effect of data from hot events, the social network graph presents multiple dense sub-graphs around different hot events. Due to the phenomenon that the entities in the group are dense and sparse between groups, the graph model is divided into several cluster structures of data. We formalize the classification and definition of these space-intensive clusters.

Let E_p represents the node describing a hot event, M_ϵ is the length of the content property contained in E_p. R_ϵ indicates the number of forwards of E_p. C_ϵ and L_ϵ is the frequency of comments and likes of E_p. Since the size of the message content exceeds 32 Byte, the data cannot be encoded as inline values and occupies the storage space in the property graph data model [4]. We set the threshold of content size $\Gamma_\iota = 32$ Byte. The interaction times as Γ_κ and Υ_κ.

- If $M_\epsilon > \Gamma_\iota$, and $R_\epsilon > \Gamma_\kappa$, $C_\epsilon + L_\epsilon > \Upsilon_\kappa$, then E_p is Large Entity which denoted as E_p^Θ.
- If $M_\epsilon > \Gamma_\iota$, and $R_\epsilon > \Gamma_\kappa$, but $C_\epsilon + L_\epsilon < \Upsilon_\kappa$, then E_p is Big Entity, expressed as E_p^θ.
- If $M_\epsilon > \Gamma_\iota$, but $R_\epsilon < \Gamma_\kappa$, and $C_\epsilon + L_\epsilon < \Upsilon_\kappa$, then E_p is Wide Entity. We represent it as E_p^+.

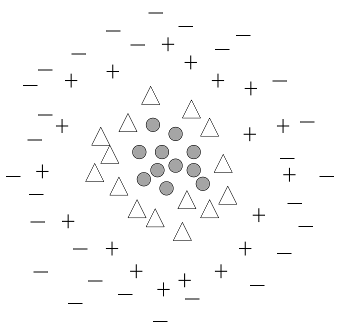


FIGURE 2. The structural relationship between spacial cluster entity.

- If $M_\epsilon < \Gamma_\iota$, then we define E_p as Short Entity and represented as E_p^-.

Therefore, the entity in the space-intensive cluster can be summarized as:

$$E_p = \begin{cases} E_p^\Theta & M_\epsilon > \Gamma_\iota \; R_\epsilon > \Gamma_\kappa \; \& \; C_\epsilon + L_\epsilon > \Upsilon_\kappa \\ E_p^\theta & M_\epsilon > \Gamma_\iota \; R_\epsilon > \Gamma_\kappa \; \& \; C_\epsilon + L_\epsilon < \Upsilon_\kappa \\ E_p^+ & M_\epsilon > \Gamma_\iota \; R_\epsilon < \Gamma_\kappa \; \& \; C_\epsilon + L_\epsilon < \Upsilon_\kappa \\ E_p^- & M_\epsilon < \Gamma_\iota \end{cases} \tag{1}$$

Case 1. Large Entity:

The large entity E_p^Θ is a circular representation in the Fig. 2, which gathers in the center. Due to the amount of text volume and microblog interactions, including forwarding, likes, and comments exceeded the given threshold Γ_κ, a substantial data can not be encoded as inline values and frequently refresh to memory. Therefore, the storage space is wasted. It is easy to increase the I/O operation and reduce the throughput of the database.

Case 2. Big Entity:

E_p^θ is distributed on the periphery of the large entity, which is represented by a triangle in Fig. 2. Because text volume and the forwarding number exceed the preset threshold and the number of likes and comments does not reach Γ_κ, there is also a large amount of data that can not be encoded as inline values. However, It does not need to be refreshed frequently in memory. It can result in wasted storage, but with minimal effect on throughput.

Case 3. Wide Entity:

E_p^+ is denoted as "+" in Fig. 2. Its text volume exceeds Γ_ι, but the number of interactions does less than the threshold Γ_κ. This data does not belong to the hot data, which has little impact on the storage model.

Case 4. Short Entity:

E_p^- is dispersed at the outer end of the cluster structure, which is represented by "_" in Fig. 2. It can be encoded inline into the storage records when stored.

Definition 1 (Spacial Cluster Entity): The entities in *Case* 1 and *Case* 2 can cause massive nodes to contain redundant data in Neo4j. They waste space and influence the processing efficiency of the graph. Therefore, we define E_p^Θ and E_p^θ as *Spacial Cluster Entity*, which is the object compressed in GSCO model. It denotes as $E_p^\Theta \cup E_p^\theta = \{M_\epsilon > \Gamma_\iota \; \& \; R_\epsilon > \Gamma_\kappa\}$. Γ_κ is set to the mean value of message forwarding times in each event of the dataset based on the statistical analysis.

C. GSCO STORAGE MODEL

In this section, we introduce the GSCO storage mechanism by presenting its many-to-one mapping structure and the heat evolution elimination algorithm H2E.

1) MANY-TO-ONE MAPPING STRUCTURE

We adopt the native graph storage and file storage to reduce the independence of the storage structure, thereby ensuring the separation of graph structure and property. The existing storage model utilizes a one-to-one mapping relationship between nodes and properties. To prevent multiple dumps of the same content, we redesign the many-to-one structure between nodes and property storage. We create a representative node that contains the common data of all node properties in the graph. To more clearly describe the novel structure relationship in the graph model, we introduce a sample exhibited in Fig. 3. By comparing the storage pattern of GSCO with the property graph model, the problem of redundant data can be highlighted.

The node *id_a* is the original microblog message, other nodes are the forwarding node of *id_a* in Fig. 2(a). The edges among nodes denote the repost relationship. The dotted line indicates the relationship between entity and property storage, which forms a one-to-one structure. The node *rid* is the representative node of the shared content extracted from the properties of all nodes in Fig. 2(b). The nodes containing common data point to the same representative node. The dynamic storage file of the representative node is the shared content of all connected nodes. It creates a many-to-one structure mapping between different entities and the representative nodes. The GSCO storage model greatly reduces the copies of the property records. Through the conversion of inline values and pointers, the dynamic storage records of the entity node with the common property are shared. Therefore, it is not necessary to store each node for relation scheduling in the graph mode, thereby achieving compression of storage space.

Specifically, the implementation process of the many-to-one mapping structure is as follows.

a: DATA STRUCTURE

Suppose $C = \{E_p^\Theta, E_p^\theta\}$, the each object is a spatial cluster entity which denotes a *key*−*value* pair. For any two mappings in C, $f(key)$ and $g(key)$ is continuous in the domain of the position $key = M_{id} \in C$. If there is a sufficiently small neighborhood N of $key = M_{id}$, it makes $f(key) \equiv g(key)$ on N, then $f(key)$ and $g(key)$ are equivalent at the point. We mark it as $f \sim g$.

If $S(M_{id})$ represents the set of all contiguous mappings in the neighborhood of the origin, then $\sim$ is an equivalent relationship on $S(M_{id})$. Each equivalent class of $S(M_{id})$ cor-

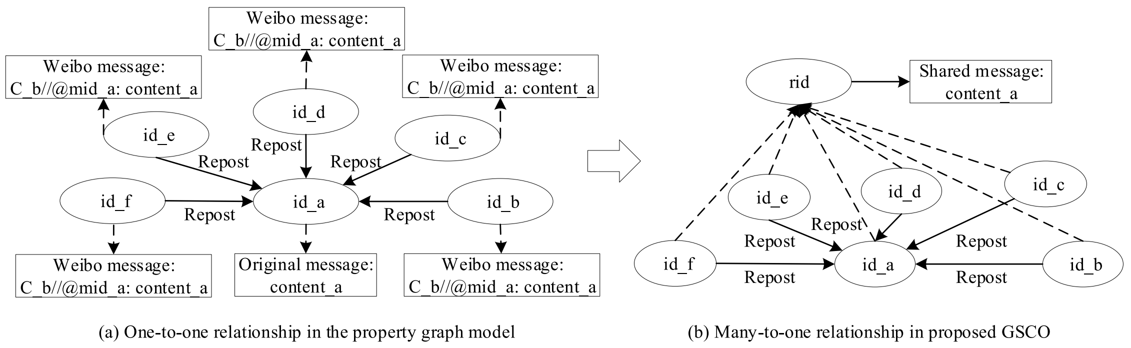


FIGURE 3. **The interaction relationship between the storage files.**

responds to $\sim$ is called a germ that is continuously mapped at the origin. The equivalent class is the same *value* that corresponds to *key* in the different collections of C. Due to the non repeatable nature of the set, there are different *value* in $S(M_{id})$, that is, $|S(M_{id})| \ll |\sum f(M_{id})|$. We use θ_n to represent the whole row of germs that are continuously mapped at the origin, which is $\theta_n = S(M_{id})/\sim$.

Based on the above germ theory, we define the data structure of many-to-one in GSCO. It decouples the spatial cluster entity in the social network graph G. The evolution process of the data structure of the GSCO storage model is described in detail as follows.

Firstly, the graph structure is represented as a triple $G_0 = (V, E, A)$ based on the nature of the separation of properties from nodes and relationships. $|A(G_0)|$ represents the number of properties in G_0. There are the mappings among the node, relationship, and property.

- Between the node and the property: $\forall v \in V(G_0), \exists\, !r \in A(G_0)$ corresponds to v, referred to as: $f : V(G_0) \to A(G_0)$.
- Between the relationship and the property: $\forall e \in E(G_0), \exists\, !r \in A(G_0)$ corresponds to e, referred as: $f : E(G_0) \to A(G_0)$.

Let $V_s = \{v_{s0}, v_{s1}, v_{s2} \ldots\}$ is a node sequence of graph G_0, which each $v_{si} \in V_s \subset V$. Let v_{s0} is the innermost node in the node sequence. As i increases, the distance of the node v_{si} from the node v_{s0} becomes farther. The level becomes lower, that is, the in-degree $|ID(v_{si})|$ decreases.

Secondly, we further divide $A(G_0)$ into two parts A_u and Y, which extends the new graph representation to $G = (V, E, A_u, Y)$. Here $A_u \subset A$, Y is a set of $< key, S(M_{id}) >$, $Y \subset A$, $A_u \cap Y = \emptyset$ and $A_u \cup Y \subset A$. The Y is a germ sequence, so we have $Y = \{Y_1, Y_2, Y_3 \ldots\} = \{< key_1, S(M_{id1}) >, < key_2, S(M_{id2}) >, \ldots\}$. Where $< key_i, S(M_{idi}) > \subset Y$, $Y \subset G$, but $Y_i \neq < key_i, S(M_{idi}) >$. Based on the above inference, we conclude that the Y is a set formed by the germ group. The element in Y is referred to as a representative node. Y satisfies the properties of closure, associative law, unit element, and inverse element. The dynamic storage in multiple nodes is projected into a property value. It cuts down the coupling among nodes and reflects the many-to-one mapping pattern of the graph structure.

b: BASIC OPERATIONS

To extract representative nodes, we implement some basic graph operations. They are *spanSequence*, *readProperty*, and *extractGerm*, respectively.

spanSequence. Given the graph G, we firstly identify the space-intensive clusters containing the spacial cluster entities. Because the in-degree represents the popularity of the node, it is more important than the out-degree to reflect the heat of a node. We only consider the in-degree in the algorithm. We obtain the node with the highest degree by breadth-first traversal. The node with the largest degree is pushed into the root node queue ℓ_r. Meantime, a new queue ℓ_n with the current node as the head is generated. As a root node of a hot event, all child nodes are iteratively traversed and stored into the ℓ_n according to the label and degree. If there is a node whose degree is greater than the current root node, we judge the content similarity of the two nodes. When the similarity is larger than the given threshold, the new node act as the root node, while the original one dequeue. On the contrary, the node is placed in the root node queue and become into the root node of a new hot event. The above operations are repeated until all nodes are traversed in the graph.

Finally, a root node queue ℓ_r is generated. Each element in ℓ_r as the head node corresponds to a node queue ℓ_n. Each ℓ_n indicates a hot event and is determined by the root. G is partitioned into some subgraphs centered on one hot event. The *spanSequence* is summarized in Algorithm 1.

readProperty: We read the node properties in turn by the label, and then establish the index on the label property. We obtain the property value in the dataset that needs the maximum storage space. The property name and its mapping value are denoted as pn_{Max} and $value_{Max}$. We retrieve the nodes set D_v corresponding to the node v through the global index B, and then perform property traversal. If there is a property that satisfies the condition, a label is added to the name of the property. Then, we create an index B_a^v based on the node v and its properties on this label property. We show the algorithm of *readProperty* in Algorithm 2.

extractGerm: According to the node sequence ℓ_n generated by *spanSequence*, we traverse the all node by calling the *readProperty* algorithm. In this case, we get a property B-tree based on the graph G and its property index file f. We traverse

Algorithm 1 *spanSequence*(G, v_0)

```
Input: graph $G$, start node $v_0$
Output: a root queue $\ell_r$, several node queues $\{\ell_n\}$
1  while $\exists$ node $v$ are not visited do
2      $L \leftarrow$ all the unvisited neighbors of node $v_0$;
3      $root \leftarrow maxDegree(v_0, V(L))$ & $[root]$ = traversed;
4      if $root$ not in $\ell_n$ with $\ell_r$.top as the front node then
5          if $root$ has a high similarity with the $\ell_r$.top then
6              if node is not a spacial cluster entity then
7                  delete $\ell_n$ with front node root;
8          else
9              $\ell_r$.push($root$);
10             create a new queue $\ell_n$ based on current root;
11             $\ell_n \leftarrow insertSort(V(\ell_n))$;
12             foreach $v$ in $\ell_n$ & [v]! = traversed do
13                 $spanSequence(G, v)$;
14     else
15         $\ell_n \leftarrow insertSort(V(\ell_n))$;
16 if node $x$ in $\ell_r$ is not a spacial cluster entity then
17     delete $\ell_n$ node queue with front node $x$;
18 return $\ell_r$, $\{\ell_n\}$;
```

Algorithm 2 *readProperty(v, f)*

```
Input: node $v$, property index file $f$
Output: index $B_a^v$ installed in the property of node $v$
1  $D_v \leftarrow B.query(v)$ ;
2  foreach $i$ in $f$ do
3      if $f_i$ is Inlining then
4          skip;
5      else
6          foreach $A_i$ in $A$ do
7              create index on:$A_i$;
8              $B_a^v \leftarrow index(A_i)$;
```

Algorithm 3 *extractGerm*(G,f)

```
Input: $G = (V, E, A)$, property or field $f$
Output: B-tree $B_a$ installed in the property of $G$; NULL
        otherwise
1  foreach $v$ in $V$ do
2      readProperty$(v, f)$;
3  if $B_a \neq \emptyset$ then
4      foreach $B_a^v$ in $B_a$ do
5          $S_G \leftarrow B_a^v.value$;
6      createRelation();
7      return $S_G$;
8  else
9      return NULL;
```

the data for the second time according to the B-tree index. The data pointed by the index is extracted as an independent node. We use pattern matching to determine the degree of repetition of the data. If it first appears, a new node is created with the association to the node. Otherwise, we create a new association with the node. As a result, it forms a many-to-one structure mapping. Finally, we delete the B-tree index and the dynamic data storage file created in the property after converting it to a new node and connection. The representative nodes are generated. The process is shown in Algorithm 3.

c: COMPLEXITY ANALYSIS

The above three algorithms realize the conversion of data structure in GSCO model. These algorithms are implemented in the graph database which supports graph calculation. The graph traversal interface is used to ensure the effectiveness of the algorithm. *SpanSequence* is a recursive algorithm whose contains the traversing (line 12) and insertion sort (line 16) of the nodes queue, the time complexity is about $O(|V|^2 + |V| \times |E|)$. The *readProperty* algorithm takes $O(log(m, |V|))$ time by requiring a traversal of the nodes in the graph, where m indicates that the B-tree is an m-ary tree. In extractGerm algorithm, the query on the B-tree can be carried out in $O(log(m, |V|) \times |V|)$ time (lines 3-7). In the worst case, the value of m can be taken as 1, so that time complexity reaches $O(|V|^2)$.

2) HEAT EVOLUTION ELIMINATION ALGORITHM (H2E)

The GSCO model modulates local data into multiple *small* communities by representative nodes. It aggregates nodes belonging to the same spatial cluster and establishes a new network. In this situation, the graph application performs multiple operations in one logical subgraph of the entire dataset. In the many-to-one structure built by the GSCO model, the operations can be accumulated into a more cohesive space. Therefore, it is necessary to optimize the model to reduce the burden of the spatial cluster. First, the boundary of each community provides the smallest cut point for the segmentation of the large graph. It is more conducive to the cut graph data for the cache operation. Moreover, due to the spatial cluster entity is triggered by a hot event, It can keep the attention heat for a while. We are inspired by the heat evolution of the event and further propose the H2E algorithm, which achieves the iterative updating of heat data.

Specifically, we analyze the popularity of each representative node. The hot data is determined by the number of being reposted, commented, and liked. We anchor the nodes with persistent heat in memory for a while, which can update the memory constantly and improve the hit rate. The proposed H2E algorithm determines and updates important data continuously. The data stored in the GSCO storage model comprises of the regular nodes, related relationships and properties, representative nodes, multi-degree relationships, and relative properties. Based on the storage characteristics of each part of the data, they present a star-shaped distribution with representative nodes as the center. The data closer to the cluster center are more important in the graph structure. Therefore, we set the center point, i.e., the representative nodes have the highest priority and outwards in turn lower. According to Definition 1, we define the priority of each element.

$$\begin{aligned} P(Y) &= P({}_1^S R) = 5 \\ P({E_p}^{\Theta}) &= P(\rho {E_p}^{\Theta}) = 4 \\ P({E_p}^{\theta}) &= P(\rho {E_p}^{\theta}) = 4 \\ P(E_p^{+}) &= P(\rho {E_p}^{+}) = 3 \\ P(E_p^{-}) &= P(\rho {E_p}^{-}) = 2 \\ P(E^N) &= P(\rho E^N) = 1 \end{aligned}$$

where, ρ denotes the property of entity. ${}_1^S R$ is the many-to-one relationship. E^N represents the normal node except for the spatial cluster entity. Combining Definition 1, we formalize that the priority relation of each data structure in the GSCO. The priority comparison decides to the dynamic evolution principle of the memory. The relation is as follows:

$$\begin{aligned} P(Y) &= P({}_1^S R) \\ &> P({E_p}^{\Theta}) = P(\rho {E_p}^{\Theta}) = P({E_p}^{\theta}) = P(\rho {E_p}^{\theta}) \\ &> P(E_p^{+}) = P(\rho {E_p}^{+}) \\ &> P(E_p^{-}) = P(\rho {E_p}^{-}) \\ &> P(E^N) = P(\rho E^N) \end{aligned}$$

By relevant statistics, the spread of the hot event usually lasts about a week on social networks. Therefore, we set the heat duration as a week and denote as T. The maintenance time of a hot event is following normal distribution or skew distribution. It is not constant in its life cycle T whether it conforms to the normal distribution or the skewed distribution. To measure the state of heat change, we define the heat evolution acceleration and propose H2E based on α_φ. The heat evolution acceleration of node can be denoted as:

$$\alpha_\varphi = \frac{\Delta v}{\Delta t} = \log\left(\frac{ID_{t_2} - ID_{t_1}}{t_2 - t_1} + 1\right) \tag{2}$$

where $ID = R_\epsilon + (C_\epsilon + L_\epsilon)$, t_1 and t_2 are any points in the cycle, where $t_2 > t_1$. ID_{t_1} and ID_{t_2} represent the penetration of the hot event at t_1 and t_2, respectively. They are affected by the number of forwards of node and the frequency of comments and likes.

For a hot event, the acceleration is at the half-cycle of its duration. There are three cases:

- $a_{T/2} = 0$, it indicates that hot events conform to the normal distribution.
- $a_{T/2} > 0$, it demonstrates that the hot events meet the negative skewed distribution.
- $a_{T/2} < 0$, it shows that the hot events are in positive skewed distribution.

We determine the time interval of the next time node base on the above three cases. In summary, the flow chart of the H2E algorithm shown in Fig. 4. Given the graph G, we first extract data that conforms to spatial cluster entity and generate the sequence ℓ_n using *spanSequence* algorithm (details in the first basic operation). H2E maintains two queues ℓ_h and ℓ_c. ℓ_h is used to record the history of all cache data being accessed, while ℓ_c saves the cached data. According to the sequence ℓ_n, when the data in it is accessed, we calculate its priority and add it to the access history list ℓ_h. However, if the data priority is **5**, it is directly stored in the cache. The cache list ℓ_c is sorted again according to the heat evolution acceleration. After part of data is retransmitted in the ℓ_h, if some data do not reach the threshold of heat elimination, then the data with the minimum product of priority and acceleration $P \times \alpha_\varphi$ is eliminated. When the ℓ_c is accessed again, the heat evolution acceleration α_φ is updated to be reordered. When the list is full, the data at the end of the ℓ_c is eliminated, that is, the data with the *least heat acceleration* can be replaced. We summarize the H2E in Algorithm 4.

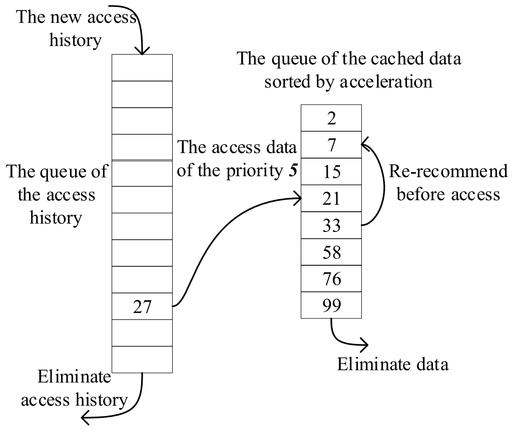

The new access history
The queue of the cached data sorted by acceleration
The access data of the priority 5
The queue of the access history
27
2
7
15
21
33
58
76
99
Re-recommend before access
Eliminate access history
Eliminate data

FIGURE 4. The flow chart of H2E algorithm.

LRU algorithm eliminates data based on the access history list in one queue, and its complexity is $O(1)$. LRU-K maintains two queues on this basis, namely the FIFO queue and the LRU queue, which improves the hit rate. LRU-K transfers data in the access history queue into the cache queue according to the access frequent of data K. The H2E algorithm maintains two LRU queue by directly judging the priority of the data and does not record the access times of data so that reducing overhead. Therefore, the compared complexity: LRU-K > H2E > LRU.

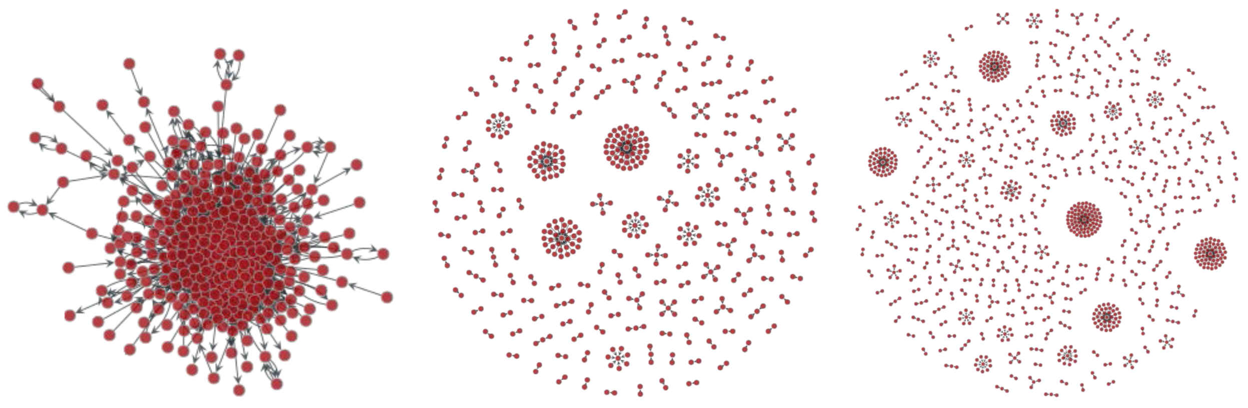
IV. EXPERIMENT AND EVALUATION

In this section, we experimentally evaluate the proposed GSCO model on the standard datasets. The proposed GSCO and the baseline database Neo4j (v3.1.3) are compared from three aspects, including data loading, some common queries, and clustering tests, respectively. In particular, we select several common clustering algorithms to evaluate the storage model in the clustering test. These clustering algorithms contain K-means [33], DBSCAN [34], spectral clustering(SC) [35], and hierarchical clustering(HC) [36].

A. EXPERIMENTAL ENVIRONMENT AND DATASET

1) EXPERIMENTAL ENVIRONMENT

We perform all experiments on Lenovo ThinkCentre Desktop. It is an Intel ® Core^{TM} x86_64 platform with a 3.40 GHz 8-core i7-6700 CPU and 16 GB of RAM, running CentOS 7



(a) 500 nodes around a hot event (b) random 1000 nodes (c) random 2000 nodes

FIGURE 5. **The distribution of partial data in the dataset.**

Algorithm 4 H2E Algorithm

```
   Input: $\ell_h$, $\ell_c$
   Output: the new $\ell_h$ and $\ell_c$
1  Through heat acceleration and priority, it determines the
   data anchored in memory, the length of time;
2  foreach $i$ in $G$ do
3      if $P(i) == 5$ then
4          put $i$ into $\ell_c$;
5          compute all $\alpha_\varphi$ by using Equation (2);
6          update $\ell_c$;
7          if $\ell_c$ is full then
8              delete Min($\alpha_\varphi$);
9      else
10         put $i$ into $\ell_h$;
11         computer $\alpha(i) = \alpha_\varphi(i) * P(i)$;
12         if $\alpha(i) < \Gamma$ then
13             delete $i$ from $\ell_h$;
14         else if $\ell_h$isfull then
15             delete Min($\alpha(i)$);
```

with kernel-4.4.110. We utilize a size of 1 TB, 7200 rpm Western Digital HDD equipped with the ext4 file system. Because the HDD is I/O bound even with a single thread, we run serial experiments on the HDD. The environment provides equal and effective operating conditions for the storage model GSCO and the baseline graph storage model Neo4j.

2) DATASET

Our experimental data from datatang[4] with a data format of json.bz. The dataset composes 13 hot events from the social networks. Totally, it includes 84,168 microblog information, 27,759 forwarding relationship, 63,641 user information. We mainly deal with the microblog content and forwarding relationship. We first remove the punctuation and special symbols without affecting the semantics. According to the microblog id, we crawl the number of likes, comments and forwards for each message. Then, we construct the connection edges to forming the graph G by repost relations among nodes in the social networks.

To display the form of the spatial cluster, we extract part of data from the dataset. The distribution of data is shown in Fig. 5. Among them, Fig. 5(a) is the distribution of 500 nodes around a hot event, Fig. 5(b) and Fig. 5(c) is the distribution of random 1,000 and 2,000 nodes. We can see that spatial clusters are local dense structures centered on events. Only nodes that reach a certain popularity can form spatial clusters, such as being frequently forwarded, like, and comment. These entities are the target of GSCO compression.

B. EXPERIMENTAL COMPARISON

In this section, we conduct massive experiments to verify the effectiveness of GSCO in graph data storage. They mainly include three aspects of workloads to compare the effect of GSCO and Neo4j. Each test is a simulation designed to be a common operation in a graph database system. We describe each workload in detail and make a comprehensive evaluation of GSCO on the dataset.

1) DATA LOADING

This section mainly includes single insertion workloads and massive insertion workloads, as well as the comparison of space occupied after the data is imported. In the preprocessed dataset, there are 26% of the data that meets the conditions of the spatial cluster entity. Neo4j treats all data equally, whereas GSCO deals with the spatial cluster entity differently from the rest data of the 74%. The following shows the impact and compression effect of duplicate data in the storage model GSCO and Neo4j under different operations.

a: SINGLE INSERTION WORKLOADS

We simulate a real-time scenario with the single insertion workloads, in which case the graph is created progressively. We assume that the growth of the graph follows the steps

[4]http://www.datatang.com/data/46758

TABLE 1. Dataset used in single insertion workloads.

Dataset	Nodes	Edges
Weibo1	10000	3277
Weibo2	20000	6655
Weibo3	30000	9993
Weibo4	40000	13269
Weibo5	50000	16805
Weibo6	60000	20086
Weibo7	70000	23441
Weibo8	80000	26522

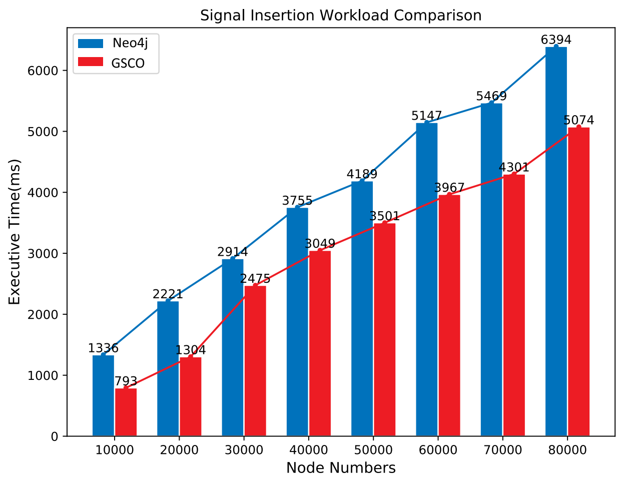


FIGURE 6. The comparison of single insertion workloads.

by the single insertion. We segment the dataset into several blocks to comprehensively evaluate the loading performance. Each block consists of ten thousand nodes and the edges that appear when these nodes are inserted. The statistics of data inserted in batches are shown in Table 1.

Firstly, we create a graph database and load the experimental data shown in Table 1 into it. Each object (*i.e.*, node or edge) is inserted directly and the graph is constructed incrementally. The single insertion workloads are processed by creating a single node and a relationship related to the node that has already been created. If the end node does not exist in the dataset, a node with only node *id* and no property is created. We measure the insertion time of each block and plot the results in Fig. 6.

Fig. 6 shows the comparison of operands per second in GSCO and Neo4j after all the data has been created. The X axis indicates the size of each data block and the Y axis is the time required to import the data block. We can conclude that the executive time of GSCO is significantly lower than Neo4j. This is mainly because 26% of the data meets the conditions of spatial cluster entities in the dataset. They are the compressed object of GSCO, which are implemented many-to-one mapping and storage. Therefore, in the process of importing nodes and edges, GSCO reduces the reimport of a large number of nodes and edges and greatly saves the load of single insertion. In the process of importing all nodes into the GSCO in turn, the operands per second are 27391, while the Neo4j is 12523. It is more obvious that the part of data

TABLE 2. The results of massive insertion workloads (in ms).

Graph Comp.	Quantity	Neo4j(v3.1.3)	GNode
Nodes	84168	7983	2802
Edges	27759	1615	1475
Total	*	9598	4277

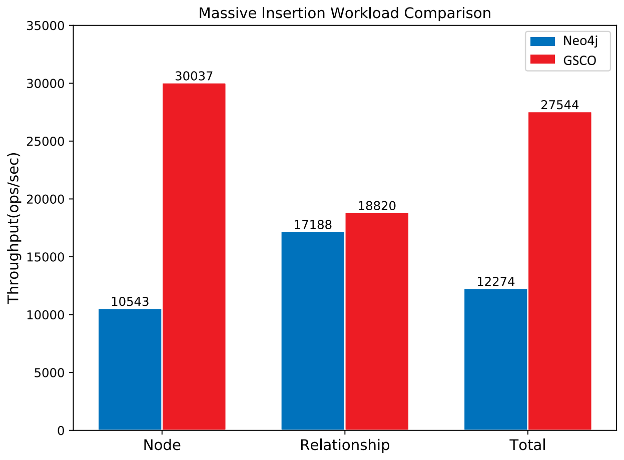


FIGURE 7. The comparison of massive insertion workloads.

that is spatial cluster entity achieve 37037, and the average speed of the remaining data is 23809 ops/sec.

b: MASSIVE INSERTION WORKLOADS

In this test, we simulate the massive insertion workloads. We first create a graph database and configure it to be bulk loaded mode. The dataset is loaded into the graph database GSCO and Neo4j, respectively.

The massive insertion workloads are processed by batch importing all nodes and all relationships correlated to the node that has already been imported. If the end node does not exist, then we create a node with only the assigned node *id* and no property. We batch import all the node data first, then load all the relationships into the database by the interface Batch Inserter. Simultaneously, we measure the time to create the entire graph. The results of the massive insertion test are shown in Table 2.

Table 2 shows the advantages of the GSCO model compared to Neo4j in massive insertion workloads. We plot Fig. 7 to display the performance comparison of GSCO and Neo4j in throughput. We plot Fig. 7 to display the performance comparison of GSCO and Neo4j in throughput. The abscissa axis is divided into three dimensions to compare the import efficiency. They are nodes batch import, relationships batch import, and total data batch import, respectively.

By analyzing the results in Fig. 7, we make some conclusion. In the processing of batch importing the nodes into Neo4j, the operands per second are 10543, while the operands per second of batch importing the edges are 17484. To summarize, the entire batch importing rate is 12274 ops/s, and the total executive time is 9.6s. The counterpart of the previous situation is that the import speed reached 43478 ops/s when the 26% node data that satisfies the spatial cluster entity is imported into GSCO. The operands per second of batch importing the remaining nodes are 24894. So, the speed of importing all nodes is 30037 ops/sec. And the operands per second of batch importing the edges are 18820. Consequently, the entire batch importing rate is 27544 ops/s, and the total execution time is 4.3s.

c: DATA SPACE OCCUPANCY

After all the data is imported into the database, the occupied space in GSCO and Neo4j are 79155.2 KB and 141516.8 KB, respectively. The new graph storage engine GSCO saved 43% more storage space than Neo4j. According to the above numerical comparison, it can be found that GSCO reduces space utilization. The results indicate that the GSCO can effectively implement data compression in the dataset with spatial clusters.

2) QUERY WORKLOAD, QW

We perform three common graph queries on graph database GNode and Neo4j, which are FindNeighbours, FindAdjacentNodes and FindShortestPath. These three queries are applied to most of the existing social network data. For the FindNeighbours, we can find some friends or followers on Facebook or Twitter. Similarly, we can find out whether two users have joined a specific community by the find adjacent nodes query (FindAdjacentNodes). Through the find shortest path query (FindShortestPath), we can find two users connected to each other in social networks. Therefore, it is crucial whether these queries can be efficiently implemented in the shortest time.

a: FINDNEIGHBOURS

To demonstrate the effectiveness of the proposed GSCO in FindNeighbours, we find the neighbor nodes of `INCOMING` and `OUTGOING` corresponding to all the nodes using breadth-first traversal through the interface Traversal Description. We set the depth to `1`. We obtain the results drawled in Fig. 8 that the execution time of nodes are greater than `1` in both `INCOMING` and `OUTGOING`. The axis marked as *Node Id* represents the number of each node stored in the graph database. The *Degree* denotes the number of neighbors corresponding to each node. The ordinate is the time required to find all neighbors.

We use the blue line and the orange line to represent the `INCOMING` and `OUTGOING` of Neo4j, respectively. In the same way, we use the green line and the red line to show the `INCOMING` and `OUTGOING` of GSCO. In addition, the blue part and green part are `INCOMING` data. The "id" number of the query results mainly ranges from the $[1.1\times10^6, 1.9\times10^6]$, the relationship concentration is the degrees within [1, 5]. The time used by the GSCO is concentrated in the 0-2 ms, and the time of Neo4j is concentrated with the 1-4 ms. Therefore, GSCO takes less time than Neo4j. While the time used by the GSCO and Neo4j are similar in `OUTGOING` data. This proves the importance and influence of the in-degree of the node in the GSCO model.

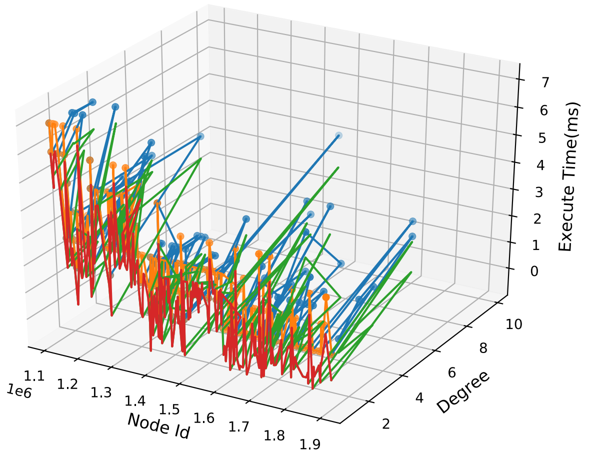


FIGURE 8. **The data range used in FindNeighbours.**

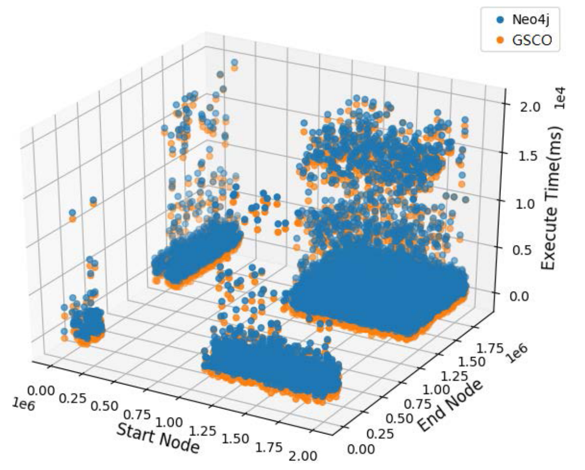


FIGURE 9. **The comparison of finding adjacent nodes on the main sides.**

b: FINDADJACENTNODES

In the same way, we traversed each edge, in turn, to perform the *FindAdjacentNodes* operation through the interface Traversal Description. Since each edge has a start node and an end node, they are the adjacent nodes of this edge. So we are looking for the starting and ending nodes of the edge. Due to the nature of the dataset containing hot events, the relevant nodes of the edge are relatively concentrated. The time required for Neo4j and GSCO to find the adjacent nodes of each edge is shown in Fig. 9.

The *Start Node* and *End Node* in Fig. 9 represent the number of two adjacent nodes of an edge, respectively. The ordinate represents the time required to traverse the edge. The start node and end node of each edge are mainly located in the scope formed by node "id" are $[7.5 \times 10^5, 1.75 \times 10^6]$ and $[1.5 \times 10^6, 1.9 \times 10^6]$. Finally, the traversal time is mainly concentrated ranges from 500 to 2000 ms. From the distribution of results in Fig. 9, the GSCO has less execution time than Neo4j.

c: FINDSHORTESTPATH

We find the shortest path between the given starting node and 100 randomly chosen nodes by the *FindShortestPath* method. We randomly select a node (that is *id* = "1926965″) as the

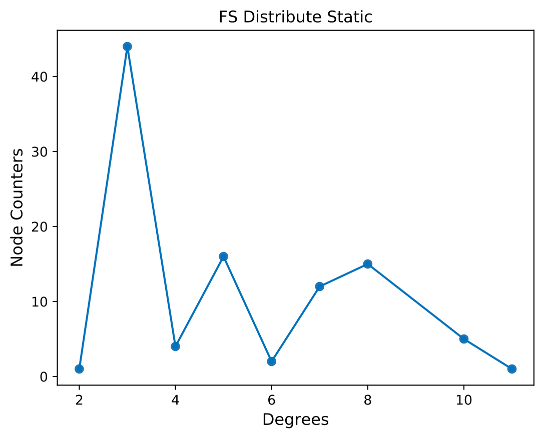


FIGURE 10. The degree distribution of 100 node.

TABLE 3. The execution time of three QW (in ms).

Workloads	Neo4j	GSCO
FindNeighbours	50895	34783
FindAdjacentNodes	1129	781
FindShortestPath	9539	5400

starting node. The length of the shortest path between it and random 100 nodes is mainly 9 different values, which degrees are $\{2, 3, 4, 5, 6, 7, 8, 10, 11\}$. The shortest path distribution between 100 nodes with the given node is shown in Fig. 10.

The distance between the 100 nodes and the majority of the nodes at the starting node is 3. We display the execution time in Table 3. It summarized the execution time of the above three common queries. In the three query operation, the query time of GSCO has the minimum consumption in executive time. This is because GSCO is designed a many-to-one structure, which can save round trip lookup time compared with one-to-one mapping in the property graph storage model Neo4j. The results of the query workload prove the GSCO can prominently improve the graph database effective in dealing with the space-intensive graph.

3) CLUSTERING WORKLOADS, CW

Clustering is a commonly used technique for statistical data analysis in many fields [37]–[39]. The typical operations in social networks include event discovery [40] and community mining [41]–[43]. Therefore, it is a critical criterion that the database can provide fast clustering operations. Currently, the traditional clustering algorithms include K-means [33], DBSCAN [34], spectral clustering [35], and hierarchical clustering [36], etc. We conduct the four clustering algorithms in dataset storage by GSCO and Neo4j respectively and evaluate the performance of the storage mechanisms.

Since the dataset contains 13 hot events, we set the number of clusters to 13 and select the corresponding initial centroid for K-means. Then, we adopt the vector space model (VSM) to quantify the message content. The text similarities act the distance measure of the nodes in the clustering, which is calculated by the cosine angle. Note, we select the representative nodes as the initial centroids of K-means in the GSCO storage model. We set the parameter radius *Eps* and *MinPts* in advance in DBSCAN. According to the provided *MinPts* and the value of the radius *Eps*, all the core points are calculated. It is perceived from the nature of DBSCAN that every representative node is the core object. For the spectral clustering (SC) algorithm, we also set the number of clusters as 13. The K-nearest neighbor act as the composition method and Ncut as the cut method. Besides, we use the cohesion method in hierarchical clustering (HC) and set the number of clusters to 13. For the above four clustering algorithms, we further employ the H2E algorithm to anchor each representative node in memory until the end of the cluster operation in the GSCO storage model.

TABLE 4. The result of clustering workloads (in ms).

Database	Algorithm	Accuracy	Quality	Running time
Neo4j	*K-means*	81.60	45.35	45.23
	DBSCAN	53.15	37.85	72.97
	SC	82.77	77.88	65.80
	HC	49.21	62.85	87.91
GSCO	*K-means*	87.90	73.41	16,63
	DBSCAN	68.27	62.02	25.76
	SC	93.67	81.39	18.46
	HC	60.22	69.85	38.79

We carry out 20 experiments by the four clustering algorithms on the microblog dataset stored in GSCO and Neo4j, respectively. The average results about the accuracy, quality, and running time are manifested in Table 4. It reveals the efficiency of GSCO in the clustering task. The units of accuracy and quality are percentages. The running time is estimated by the millisecond.

From the comparison results in Table 4, we observe that GSCO makes each clustering algorithm achieve the best efficiency in all metrics. This is because the many-to-one mapping structure and H2E algorithm play an important role in the GSCO. Based on the many-to-one mapping structure, the representative node can directly determine the classes of multiple adjacent nodes. Through the process of mapping creation, the related nodes have been aggregated according to the forwarding rule. It is conducive to the enhancement of accuracy and speed of the clustering algorithms. Additionally, H2E can anchor the representative node (i.e., the 13 initial centroids) in the memory. It reduces the space footprint of nodes that operate infrequently. Therefore, the GSCO reduces the iterations, thereby significantly improving the clustering effect in time. In summary, the GSCO reaches superior performance in the clustering.

To conclude, we explore the advantages of the GSCO model compared with the Neo4j in storing social network graph data through the abundant experimental results and analysis.

- Write performance: GSCO throughput up to 50,000+ ops/sec, the delay in a few milliseconds. The main bottleneck is the proportion of datasets that satisfy the spatial cluster entity.
- Reading performance: The H2E algorithm designed in GSCO makes it possible to achieve a throughput of

about 25000+ when reading spacial clustered data, with a delay of a few to 20 milliseconds.

- Clustering performance: The GSCO model compresses the redundancy data that originated from the same category. Therefore, the clustering operation reaches distinct effects.

V. CONCLUSION

In this paper, we propose the new graph storage engine GSCO, considering that the existing storage model did not scale well to spatial cluster entity in the social networks. We decouple the data structure and improve the space utilization rate and processing speed. Firstly, a new many-to-one mapping structure is proposed based on the one-to-one the labeled property graph model of the Neo4j graph database. Then, we future design the H2E algorithm to dynamically anchor data in memory based on the heat evolution acceleration. Finally, we conduct extensive experiments on a real dataset, which indicates that GSCO outperforms the state-of-the-art Neo4j in the space utilization and operation speed for the spacial clustered data of social network.

REFERENCES

[1] M. Qiao, H. Zhang, and H. Cheng, "Subgraph matching: On compression and computation," *Proc. VLDB Endowment*, vol. 11, no. 2, pp. 176–188, 2017.

[2] C. Wang, Y. Feng, Q. Guo, Z. Li, K. Liu, Z. Tang, A. K. Tung, L. Wu, and Y. Zheng, "ARShop: A cloud-based augmented reality system for shopping," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1845–1848, 2017.

[3] R. Angles and C. Gutierrez, "An introduction to graph data management," 2017, *arXiv:1801.00036*. [Online]. Available: https://arxiv.org/abs/1801.00036

[4] C. S. Park and B. K. Kaye, "The tweet goes on: Interconnection of Twitter opinion leadership, network size, and civic engagement," *Comput. Hum. Behav.*, vol. 69, pp. 174–180, Apr. 2017.

[5] H.-J. Li, Q. Wang, S. Liu, and J. Hu, "Exploring the trust management mechanism in self-organizing complex network based on game theory," *Phys. A, Stat. Mech. Appl.*, Nov. 2019, Art. No. 123514.

[6] H.-J. Li, Z. Bu, Z. Wang, J. Cao, and Y. Shi, "Enhance the performance of network computation by a tunable weighting strategy," *IEEE Trans. Emerg. Topics Comput. Intell.*, vol. 2, no. 3, pp. 214–223, Jun. 2018.

[7] G. Tillmann, *Usage-Driven Database Design*. Berkeley CA, USA: Springer, 2017, pp. 301–314.

[8] E. E. I. Robinson and J. Webber, *Graph Databases*. Newton, MA, USA: O'Reilly Media, 2016.

[9] P. Purkait, T.-J. Chin, A. Sadri, and D. Suter, "Clustering with hypergraphs: The case for large hyperedges," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 9, pp. 1697–1711, Sep. 2017.

[10] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, May 2017, pp. 79–94.

[11] S. Bharadwaj, L. Chiticariu, M. Danilevsky, S. Dhingra, S. Divekar, A. Carreno-Fuentes, H. Gupta, N. Gupta, S.-D. Han, and M. Hernández, "Creation and interaction with large-scale domain-specific knowledge bases," *Proc. VLDB Endowment*, vol. 10, no. 12, pp. 1965–1968, 2017.

[12] R. K. Kaliyar, "Graph databases: A survey," in *Proc. Int. Conf. Comput., Commun. Autom. (ICCCA)*, 2015, pp. 785–790.

[13] M. Arenas and M. Ugarte, "Designing a query language for RDF," *ACM Trans. Database Syst.*, vol. 42, no. 17, p. 2, 2017.

[14] I. Kabiljo, B. Karrer, M. Pundir, S. Pupyrev, and A. Shalita, "Social hash partitioner: A scalable distributed hypergraph partitioner," *Proc. VLDB Endowment*, vol. 10, no. 11, pp. 1418–1429, 2017.

[15] J. Chen, Y. Lin, G. Lin, J. Li, and Y. Zhang, "Attribute reduction of covering decision systems by hypergraph model," *Knowl.-Based Syst.*, vol. 118, pp. 93–104, Feb. 2017.

[16] P. Yang, W. L. Faro, R. S. Arvapally, and C. J. Merz, "Systems and methods for generating relationships via a property graph model," U.S. Patent 14 829 219, Feb. 23, 2017.

[17] R. Raman, S. Hong, and H. Chafi, "Graph data processing system that supports automatic data model conversion from resource description framework to property graph," U.S. Patent 14 812 819, Feb. 2, 2017.

[18] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu, "Graph-Flow: An active graph database," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1695–1698.

[19] R.-H. Li, J. X. Yu, R. Mao, and T. Jin, "Recursive stratified sampling: A new framework for query evaluation on uncertain graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 2, pp. 468–482, Feb. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7286806/

[20] B. Lyu, L. Qin, X. Lin, L. Chang, and J. X. Yu, "Scalable supergraph search in large graph databases," in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 157–168. [Online]. Available: http://ieeexplore.ieee.org/abstract/document/7498237/

[21] K. Zhao and J. X. Yu, "All-in-one: Graph processing in RDBMSs revisited," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, May 2017, pp. 1165–1180.

[22] R.-H. Li, L. Qin, and J. X. Yu, "Efficient and progressive group steiner tree search," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, Jun. 2016, pp. 91–106.

[23] P. Goyal and E. Ferrara, "Graph embedding techniques, applications, and performance: A survey," *Knowl.-Based Syst.*, vol. 151, pp. 78–94, Jul. 2018.

[24] A. Sadri, F. D. Salim, Y. Ren, M. Zameni, J. Chan, and T. Sellis, "Shrink: Distance preserving graph compression," *Inf. Syst.*, vol. 69, pp. 180–193, Sep. 2017.

[25] R. A. Rossi and R. Zhou, "GraphZip: A clique-based sparse graph compression method," *J. Big Data*, vol. 5, no. 1, p. 10, 2018.

[26] R. Reas, S. Ash, R. Barton, and A. Borthwick, "SuperPart: Supervised graph partitioning for record linkage," in *Proc. IEEE Int. Conf. Data Mining (ICDM)*, Nov. 2018, pp. 387–396.

[27] N. R. Brisaboa, S. Ladra, and G. Navarro, "Compact representation of Web graphs with extended functionality," *Inf. Syst.*, vol. 39, pp. 152–174, Jan. 2014.

[28] M. Nelson, S. Radhakrishnan, A. Chatterjee, and C. N. Sekharan, "Queryable compression on streaming social networks," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2017, pp. 988–993.

[29] J. Han, K. Zheng, A. Sun, S. Shang, and J.-R. Wen, "Discovering neighborhood pattern queries by sample answers in knowledge base," in *Proc. IEEE 32nd Int. Conf. Data Eng. (ICDE)*, May 2016, pp. 1014–1025.

[30] A. Khan and C. Aggarwal, "Query-friendly compression of graph streams," in *Proc. IEEE/ACM Int. Conf. Adv. Social Netw. Anal. Mining*, Aug. 2016, pp. 130–137.

[31] L. D. Valstar, G. H. Fletcher, and Y. Yoshida, "Landmark indexing for evaluation of label-constrained reachability queries," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 345–358.

[32] J. Guia, V. G. Soares, and J. Bernardino, "Graph databases: Neo4j analysis," in *Proc. 19th Int. Conf. Enterprise Inf. Syst. (ICEIS)*, Porto, Portugal, Jan. 2017, pp. 351–356.

[33] Z. Huang, N. Li, K. Rao, C. Liu, Y. Huang, M. Ma, and Z. Wang, "Development of a data-processing method based on Bayesian K-means clustering to discriminate aneugens and clastogens in a high-content micronucleus assay," *Hum. Exp. Toxicol.*, vol. 37, no. 3, pp. 285–294, Mar. 2018.

[34] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN," *ACM Trans. Database Syst.*, vol. 42, no. 3, p. 19, 2017.

[35] N. Binkiewicz, J. T. Vogelstein, and K. Rohe, "Covariate-assisted spectral clustering," *Biometrika*, vol. 104, no. 2, pp. 361–377, Jun. 2017.

[36] A.-A. Liu, Y.-T. Su, W.-Z. Nie, and M. Kankanhalli, "Hierarchical clustering multi-task learning for joint human action grouping and recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 1, pp. 102–114, Jan. 2017.

[37] Z. Bu, H.-J. Li, C. Zhang, J. Cao, A. Li, and Y. Shi, "Graph k-means based on leader identification, dynamic game and opinion dynamics," *IEEE Trans. Knowl. Data Eng.*, to be published.

[38] H.-J. Li, Z. Bu, Z. Wang, and J. Cao, "Dynamical clustering in electronic commerce systems via optimization and leadership expansion," *IEEE Trans. Ind. Informat.*, to be published.

[39] Z. Bu, H.-J. Li, J. Cao, Z. Wang, and G. Gao, "Dynamic cluster formation game for attributed graph clustering," *IEEE Trans. Cybern.*, vol. 49, no. 1, pp. 328–341, Jan. 2019.

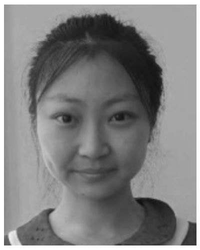
[40] N. Ko, B. Jeong, S. Choi, and J. Yoon, ''Identifying product opportunities using social media mining: Application of topic modeling and chance discovery theory,'' *IEEE Access*, vol. 6, pp. 1680–1693, 2017.

[41] J. Obregon, M. Song, and J.-Y. Jung, ''InfoFlow: Mining information flow based on user community in social networking services,'' *IEEE Access*, vol. 7, pp. 48024–48036, 2019.

[42] J. Cao, Z. Bu, Y. Wang, H. Yang, J. Jiang, and H.-J. Li, ''Detecting prosumer-community groups in smart grids from the multiagent perspective,'' *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 49, no. 8, pp. 1652–1664, Aug. 2019.

[43] H.-J. Li and J. J. Daniels, ''Social significance of community structure: Statistical view,'' *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 91, no. 1, 2015, Art. no. 012801.

DAWEI WANG was born in Shandong, China. He is currently pursuing the Ph.D. degree with the School of Information, Renmin University of China, Beijing. His research interests include design and analysis of algorithms, social network mining, databases, and graph query.



WANQIU CUI was born in Liaoning, China. She is currently pursuing the Ph.D. degree in computer science and technology with the Beijing University of Posts and Telecommunications, China. Her main research interests include artificial intelligence, social network analysis, machine learning, and information retrieval.

BIAO QIN received the Ph.D. degree in computer science from the Huazhong University of Science and Technology, in 2003. He is currently an Associate Professor with the Renmin University of China. His main research interests include probabilistic database, uncertain data mining, and Bayesian networks.