SGDB - Simple graph database optimized for activation spreading computation

Marek Ciglan and Kjetil Nørvåg

Dept. of Computer and Information Science, NTNU, Trondheim, Norway.
{marek.ciglan,kjetil.norvag}@idi.ntnu.no

Abstract. In this paper, we present SGDB, a graph database with a storage model optimized for computation of Spreading Activation (SA) queries. The primary goal of the system is to minimize the execution time of spreading activation algorithm over large graph structures stored on a persistent media; without pre-loading the whole graph into the memory. We propose a storage model aiming to minimize number of accesses to the storage media during execution of SA and we propose a graph query type for the activation spreading operation. Finally, we present the implementation and its performance characteristics in scope of our pilot application that uses the activation spreading over the Wikipedia link graph.

1 Introduction

The graph data structure is one of the most important data structures in computer science. In addition to that, it is also a useful structure for data modeling. Many real-world objects can be naturally described by graphs. The most straightforward examples are those of various types of networks; e.g., transportation networks, delivery networks, hypertext networks, citation networks, social networks or communication networks (e.g., e-mail).

Although the relational data model is dominant in nowadays information systems, modeling data in the graph structure is gaining noticeable interest. Graph databases are information systems providing graph abstraction for modeling, storing and accessing the data. In the graph data model, relations between modeled objects are as important as the data describing the objects. This is the most distinctive feature from other database models - graph databases aim to provide efficient execution of queries taking into account the topology of the graph and the connectivity between stored objects.

Graph traversal and graph analysis operations are traditionally implemented by preloading whole graph in the memory and process it in memory, due to performance reasons. This approach naturally suffers from memory size limits and is unusable when the amount of data exceeds the limits of the physical memory available on the processing machine. Graph databases aim to provide efficient persistent storage for graph data that also allow for fast graph traversal processing.

In this paper we present the storage model for the graph structure together with the architecture and the implementation of a graph database, named Simple Graph Database[1] (SGDB). The storage model of SGDB is optimized for execution of Spreading Activation (SA) algorithm over stored graphs. Spreading Activation algorithm was designed for searching semantic and association networks. Its basic form, as well as the most common variations, are extensively described in [6]. The nodes in graph structure represents objects and

[1] https://sourceforge.net/projects/simplegdb/

links denotes relations between objects. The algorithm starts by setting activation of the input nodes and the processing consists of iterations, in which the activation is propagated from activated nodes to their neighbor nodes. Mechanism of SA utilize breadth first expansion from activated nodes to identify other nodes in the network that are strongly associated with initial nodes (have high activation value). The breadth first traversal, utilized in SA, can be characterized by random accesses to the underlying graph structure, to retrieve edges of activated nodes.

To motivate the use of Spreading Activation algorithm over graph databases, we provide several use-cases. In the domain of *Semantic Web*, the SA technique has been successfully used for mining socio-semantic networks [18]. In the enterprise environment, SA can be used for *product recommendation systems* to identify the products that the customer have not purchased yet, but are highly associated with the products he already bought. *Social networks* can be naturally represented by graphs. Also in this application domain, we can find uses for SA technique. E.g., recommendation of people that the user might know.

Motivated by presented use-cases, we propose a graph database optimized for execution of SA algorithm. The main contributions of the paper are the following:

- proposal of a storage model for graph data, aiming at supporting the random access pattern for retrieval of the connectedness information of nodes activated during SA process.
- proposal of the query type for the Spreading Activation operation for graph databases, to facilitate the usage of the SA over stored graph data.

The paper is organized as follows. In Section 2, we discuss related work, we then describe the SA procedure in Section 3 to define the context of the work. The storage model for SGDB is proposed in Section 4 and we propose the graph query type for the SA operation over graph database in Section 5. The architecture of proposed system is described in 6, providing also few important implementation details. In Section 7 study performance characteristics of the system, evaluated using a pilot application for finding connections in Wikipedia link graph, Finally, in Section 8 we conclude the paper and outline issues for further work.

2 Related work

The concept of a graph databases has been popular for some time. An extensive survey of the graph database models proposed in this period is presented in [2]. Proposed models ranged in the complexity, from simple directed graphs with labeled nodes and edges [8, 9] to complex representations of nested objects [12, 13]. Also variety of graph query languages were proposed and ranged from SQL-like languages [1] to graphical queries [16].

After a period of high interest in the 90s, for a period of time the interest in graph databases disappeared. The emergence of *Semantic Web* shifted attention to RDF[2] stores. RDF can be also viewed as a graph data structure. Early RDF stores were design to operate in-memory or used relational database back-ends for the data persistence [17]. Later, specialize persistent stores optimized for semantic data were developed [11] [7]; those are designed as triple stores to support RDF model.

The focus on graph databases recently re-emerged. Several companies in the industry have developed graph databases systems (e.g. Freebase [4], DirectedEdge [3], Neo4j [4]). There

[2] http://www.w3.org/RDF/

[3] http://www.directededge.com (visited: 10.12.2009)

[4] http://neo4j.org/ (visited: 10.12.2009)

is also an effort for providing systems for large-scale graph processing in distributed environment (e.g. Google Pregel [14], graph package in Hama [5], or [10]). The data in [10] as well as Hama is stored in a distributed key value store, used in conjunction with Map-Reduce systems and the graph structure is modeled as the adjacency matrix.

3 Preliminaries

In this section, we first describe the structure that is being modeled, we than describe the Spreading Activation algorithm to define the context for the proposed approach. We discuss in detail the modified SA technique that allows to observe the value of activation received from distinct initial nodes. We highlight important points that influence the design of presented system.

3.1 Modeled data structure

The aim of this work is to support the SA technique over a graph with weighted and typed edges, stored on a persistent medium. We can define the modeled structure using equation

$$G = (V, E, f, w, T, t)$$

where G is the graph label, V is a set of nodes, E is a set of edges, f is a function $f : V \times V \rightarrow E$ defining mapping between nodes and edges, w is a function defining edge weights $w : E \rightarrow \langle 0, 1\rangle$, T is a set of edge type labels and t is a function defining edges types $t : E \rightarrow T$.

The operations considered for this data structure are insertion,deletion of nodes and edges, retrieval of outgoing and incoming edges for a given node and iteration over node and edges sets. Due to the space limitation, we define only insertion operations. Similarly, operations for nodes and edge deletion, edge weight, type modification and others can be defined. Operation of node insertion can be defined as

$insert(G = (V, E, f, w, T, t), v) = (V' = \{V \cup v\}, E, f' : V' \times V' \rightarrow E, w, T, t)$; edge insertion operation is

$insert(G = (V, E, f, w, T, t), (e_{new}, i, j, w_{val}, t_{val})) =$
$(V, E' = \{E \cup e_{new}\}, f', w', T, t') \mid i, j \in V; f(i, j) = \perp; w_{val} \in \langle 0, 1\rangle; t_{val} \in T$

where $f' : V \times V \rightarrow E'$, $w' : E' \rightarrow \langle 0, 1\rangle$, $t' : E' \rightarrow T$ and

$$f'(k,l) = \begin{cases} f(k,l)\ ;\ k \neq i \wedge l \neq j \\ e_{new}\ ;\ k = i \wedge l = j \end{cases} ; w'(e) = \begin{cases} w(e)\ ;\ e \in E \\ w_{val}\ ;\ e = e_{new} \end{cases} ; t'(e) = \begin{cases} t(e)\ ;\ e \in E \\ t_{val}\ ;\ e = e_{new} \end{cases}$$

Edge retrieval operations can be defined as follows:
$outgoing(G, n) = \{e \mid n, i \in V; e : f(n, i) \neq \perp\}$ and
$incoming(G, n) = \{e \mid n, i \in V; e : f(i, n) \neq \perp\}$

In addition to the graph topology, we want to store user defined attributes that can be associated with nodes and edges. In our approach, the user defined data (node and edges attributes) are stored in a separate structure, linked with graph by node identifiers. The storage of the user defined data is out of the scope of this paper, as it does not influence the graph traversal operations.

[5] http://wiki.apache.org/hama (visited: 10.12.2009)

3.2 Spreading activation algorithm

The Spreading Activation algorithm is based on the breadth first expansion from activated nodes in the graph data structure. Nodes in the graph represent modeled objects and edges represent relationships between the objects. Edges can be weighted or typed (or both) and can be directed or undirected. The input of the SA algorithm is a set of initially activated nodes and a set of parameters influencing the activation process, the output is a set of nodes activated by the SA process. The SA process consists of iterations in which the activation is spread in breadth first manner. Each iteration is composed of a spreading phase and a pre-adjustment or post-adjustment phases. In pre/post-adjustment phases the activation decay can be applied on activated nodes. In the spreading phase, activated nodes send impulses to their neighbors. The value of the impulse propagated from an activated node is a function of the node's input value. In the basic SA variant, the input value of a node n is equal to the sum of weighted output values of nodes connected with n by outgoing edges. The output values are weighted by the edge weights. Let T be a set of edge types of the graph and $Q \subseteq T$ be the types allowed in the SA computation. Function a is

$$a(t, Q) = \begin{cases} 1 \; ; t \in Q \\ 0 \; ; t \notin Q \end{cases}$$

The output value can be described by following formula:

$$I_n = \sum_i O_i w(e_{i,n}) a(t(e_{i,n}), Q)$$

where I_n is the input value of the node n; O_i is the output value of the node i connected to n by an outgoing edge and $w(e_{i,n})$ is the weight of the edge connecting i and n; $w \in \langle 0, 1 \rangle$; $t(e_{i,n})$ is the type of the edge $e_{i,n}$. The most commonly used output function is the threshold function, where the output is zero when the node input value is below the user defined threshold th. In case that $I_n > th$ the output is equal to one.

The activation thus spreads from initial nodes over the network. The algorithm finishes when the there are no nodes with $O_n > 0$ in an iteration. The convergence and the fix point of the SA process has been studied in [3]. In practice, some additional termination conditions are used (e.g., distance constraint).

3.3 Activation vector spreading

In the standard SA algorithm, we can not distinguish whether a node received an activation from one or multiple initial nodes. To obtain richer information about activation spread, we have introduced a modification of the standard SA technique in [5], called Activation Vector Spreading (AVS).

We store the node activation as a vector, its length is equal to the number of input nodes and the n-th element of the vector represents the amount of the activation originating from the n-th input node. The activation vector n-th input node is initiated as follows: all the values in the vector are equal to zero, expect n-th element, which is initially set to one. The activation spread is computed individually for each element of the activation vector. Informally, the activation spread is computed individually for each input node. In addition to that, in each iteration, if the individual elements of node's input vector are lower than the defined threshold th but the sum of all the elements is greater than t, we spread an output

activation vector with a non-zero element, which is the element with highest value in the input activation vector. Let $I_n = (I_{n_1}, I_{n_2}, \ldots, I_n)$ be the input activation vector of node n, where I_{n_i} is the amount of activation received from i-th initial node. The output function in AVS is:

$$O_n = \begin{cases} (thr(I_{n_1}, th), thr(I_{n_2}, th), \ldots, thr(I_{n_m}, th)) & \textit{iff } \exists i : thr(I_{n_i}) \geq th \\ (ismax(I_{n_1}, I_n), \ldots, ismax(I_{n_m}, I_n)) & \textit{iff } \nexists i : thr(I_{n_i}) \geq th \wedge \sum_i I_{n_i} \geq th \\ (0, 0, \ldots, 0) & \textit{iff } \nexists i : thr(I_{n_i}) \geq t \wedge \sum_i I_{n_i} < th \end{cases}$$

$thr(x)$ is a threshold function

$$thr(x,t) = \begin{cases} 1 \; ; x \geq t \\ 0 \; ; x < t \end{cases} \quad \text{and} \quad ismax(x, I_n) = \begin{cases} 1 \; ; \forall i : I_{n_i} \leq x \\ 0 \; ; \exists i : I_{n_i} > x \end{cases}$$

This modification allows us to observe which sources the node received the activation from (non-zero values in the activation vector) and the amount of activation from each source.

Important aspect of the SA algorithm for the graph storage design is the use of breadth first expansion from activated nodes. The activation value of a node n dependents on activation values of the connected nodes and weights and/or types of connecting edges. Those are the only values necessary to compute the activation value of a node.

4 Storage model

The aim of this work is to design a persistent graph database system allowing for fast execution of the spreading activation algorithm, without pre-loading the whole graph to the memory prior to the execution. As the access to the persistent medium is the most time costly operation, we aim at minimizing the number of accesses to the storage medium. The SA procedure utilize the breadth first expansion, characterized by a number of random accesses to the graph data. The addressed problem can be formulated as follows: Propose a persistent storage system for representation of a directed, weighted graph with typed edges that allows for an implementation of the spreading activation algorithm with the minimum number of accesses to the persistent storage media.

We can not avoid the random access pattern in general; however we organize the data in a way to reduce the number of disk access operations for retrieving the information needed to compute the activation spreading.

This section describes the storage model proposed for SGDB system, aiming at reducing storage lookups for the SA technique. Adjacency list is an ideal representation of a graph structure for breath first traversals. Adjacency list is a graph representation, where each node n has an associated list of nodes that are connects to n by an edge. The adjacency list can be viewed a set of tuples, where first element of each tuple is a node n and the second element is the list of nodes adjacent to n.

A practical data structure for adjacency list is $key - value$ map, where key is the identifier of the node and $value$ is the list of identifiers of the adjacent nodes. As the $key - value$ map is a practical data structure, there has been already a considerable amount of work performed and there are numerous persistent $key - value$ stores available (e.g. Berkeley DB,

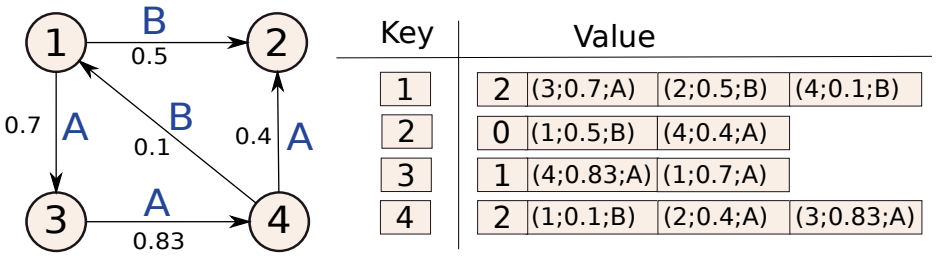

Fig. 1. Example of graph representation in the proposed storage model.

JDBM[6], BabuDB [7]), able to store large amount of data. A relational database could be used to store $key - value$ pairs; however, specialized $key - value$ stores have superior performance characteristics for the given task.

Using this representation, given a starting node n, we need $1 + d$ lookups (where d is the number of nodes adjacent to n) in a $key - value$ store to obtain a set of identifiers of nodes distant two hops from n. Spreading Activation method requires more than structural information. As stated in 3.2, to compute activation values in the SA algorithm we need additional data - weights and/or edge types. To keep the number of lookups low and to avoid additional retrieval from the data storage, we propose to keep the edges weights and types directly in the adjacency list as they are required by the SA algorithm. This simple, even trivial, change brings important time savings for SA processing, oppose to the approach where the edge weights and types are modeled as edge attributes and are stored in a separate structure.

In our storage model, the graph is stored as a set of $key - value$ pairs (i, N_i), where i is the identifier of the node and N_i is the representation of edges adjacent to node i. Each edge e is represented by a triple $e = (j, w_{(i,j)}, t_{(i,j)})$ where j is the identifier of adjacent node, $w_{(i,j)}$ is the weight of the edge connecting i and j and $t_{(i,j)}$ is the type of the edge. We model the weight by a float value and the type by an integer value. As we need to model directed graphs, we must distinguish the direction of edges. We model adjacent edges N_i as a tuple $N_i = (k, \{e_1, e_2, \ldots, e_m\})$, were k denotes the number of outgoing edges; $\{e_1, e_2, \ldots, e_m\}$ is a list of edges and all $e_l : l < k$ represent outgoing edges and all $e_l : l > k$ represent incoming edges. An example of a graph represented using proposed storage model is depicted in Fig. 1. Let us examine the record encoding node 1; the first element of the *Value* part of the record indicates that there are two outgoing edges from node 1 (those are the first two in the list - $(3, 0.7, A)$ and $(2, 0.5, A)$) and the rest of the list represents incoming edges (in this case only the edge $(4, 0.1, B)$).

This representation allows us to retrieve outgoing and incoming edges of a node n together with edge weights and types in one lookup. The disadvantage of this approach is that information on edges are redundant; i.e., edge $e_{(i,j)}$ is stored as an outgoing edge in the record of node i and as an incoming edge in the node j record. This necessitates to modify both records, in case of edge manipulation (e.g., update of the weight, type values or deletion).

[6] http://jdbm.sourceforge.net/ (visited: 10.12.2009)
[7] http://code.google.com/p/babudb/ (visited: 10.12.2009)

5 Spreading activation queries

As mentioned in Section 3.2, the input of the SA algorithm is a set of initially activated nodes and set of parameters influencing the activation spread. In this section, we propose a query syntax for executing the SA operation over the stored graph with the aim to allow the definition of SA process using simple plain text string. The purpose of the SA query is to facilitate the usage of the system, the execution of SA operation over stored graph.

The set of parameters considered for the SA algorithm is the following: activation threshold (Th), activation decay ($Decay$) , allowed edge types ($Types$) , use of incoming edges for spreading ($Incoming$) and maximal number of SA iterations ($MaxIteration$). The specification of activated nodes is done in terms of node properties that identify nodes. Thus, the proposed syntax for the SA query is the following:

([node([prop=val;]);]*) ; SAProperties: Th:threshold_val; Decay: decay_value;*
Types=([[edge_type] | all]); Incoming=[true|false]; MaxIteration=max_iterationt*

We explain the SA query on the following example. In our pilot application, we use the graph constructed from Wikipedia articles (modeled as nodes) and links (modeled as edges). Following example query executes the SA algorithm from nodes representing articles 'Spreading Activation' and 'Wikipedia', with activation threshold 0.4 and decay 0.8, using all edge types and both directions of edges, constrained to three SA iterations:

(node(name='Spreading Activation'); node(name='Wikipedia')) ; SAProperties:
Th:0.4; Decay: 0.8; Types=all; Incoming=true; MaxIteration=3

Query execution is performed in two phases. In the first phase, the input for SA operation is constructed. This involves identification of the initial activation nodes, using attributes specified in the node definition part of the query. Nodes with given attributes and attribute values are selected and the initial nodes vector is constructed. From the node definition part of the query, we construct a vector of initial nodes. E.g., initial nodes vector constructed from the example query would be ['Spreading Activation', 'Wikipedia'] (for simplicity, article names represent nodes of the graph).

Initial nodes vector, together with other parameters, is than used as inputs for SA operation. In the second step the AVS algorithm is executed, taking advantage of the underlying storage model for fast retrieval of information required to compute the activation spread.

The result of the SA query is a set of tuples; each tuple contains following elements: (Activated node, activation, partial activations vector, number of impulses vector, distance vector). *Activated node* is the node activated in the process of activation spreading, *activation* is the node's total activation, *partial activations vector* contains partial activations received from distinct initial nodes (n-th element of the vector corresponds to the activation received from the n-th element of initial nodes vector). *Number of impulses vector* contains information on the number of impulses received by the node from distinct initiators and the distance vector contains information on the distance of the node from distinct initiators. For example, part of the result set for the example query is:

(Semantic Web; 12.42; [2.4182642, 10.0,]; [3, 10,]; [2,2,])
(Web search engine; 12.12; [0.7190919, 11.407564,]; [1, 21,]; [1,3,])

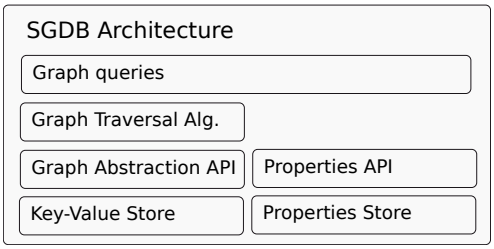

Fig. 2. Architecture of SGDB.

(Knowledge management; 10.80; [4.8060675, 6.0,]; [5, 6,]; [2,2,])

The SA query can be formulated in the plain text and its result set can be also communicated in the plain text. Another advantage of the proposed SA query is that the upper bound of query selectivity can be estimated based on query specification (use of outgoing and incoming links) and the information about initial nodes in/out-degrees.

6 SGDB architecture

This section presents the overall high-level architecture of SGDB system and its implementation. The architecture of the SGDB is depicted in Figure 2. The system is decomposed into modules with distinct functionality; modules are organized in layers, where each layer provide a simplified abstraction for the upper layer, making the complexity of underlying layers transparent.

The base stone of SGDB is the **key-value store** that maintains the adjacency lists of the graph structure in form of the key-value tuples. The main responsibility of this module is to provide fast lookups for adjacency lists based on the given key (node identifier).

Properties store is responsible for storing data (properties) related to the objects modeled by nodes and relationships modeled by edges. E.g. let us suppose that node n in the graph represents a person; properties associated with n could be name of the person, address of the person. The property store is independent of the graph structure store and allows for retrieval of the graph nodes based on their attributes.

Graph Abstraction API (GAAPI) provides graph abstraction layer, so that users can access and manipulate the graph data using graph theory concepts - nodes and edges, instead of node identifiers and adjacency lists. GAAPI provide functions for iterating over collection of graph nodes, retrieval of graph nodes based on node identifiers or user defined properties, retrieval of outgoing and incoming edges. In addition GAAPI provide functionality to modify the graph structure - insertion and removal of nodes and edges.

Properties API provides access to properties store. **Graph traversal layer** contains implementations of graph traversal operations, such as the Spreading Activation algorithm or path finding algorithms. It exploits (GAAPI) to access the graph structure. Finally, the **graph queries layer** is a presentation layer, providing access to the database functionality using graph queries (current implementation provides SA queries as described in Section 5).

6.1 Implementation

SGDB[8] is implemented in JAVA. The current implementation does not support transactions and can be used as an application embedded database (it can not be run in a standalone mode). SGDB is available as an open source software. It exploits Oracle Berkeley DB Java Edition [9] as a key-value store.

The storage model used in SGDB allows to retrieve outgoing and incoming edges together with edge weight and type data in one lookup. This is convenient for breath first traversals, especially the SA. The drawback of this approach, from the implementation point of view, is that for update operations on the graph (insertion, deletion of nodes and edges) the edge list must be retrieved from the storage, modified and than stored back, rewriting the whole original record. In addition, the data describing edge $e_{i,j}$ are stored twice in the storage – it is stored as an outgoing edge in the record of node i and as an incoming edge in the record of node j. This is the trade-off of proposed storage model, more demanding update operations are compensated by efficient retrieval operations.

7 Evaluation

In this section we first compare performance of the proposed approach with a general purpose graph database for retrieval of weighted and typed links using the random access pattern. Second, we provide performance characteristics of SGDB in the scope of our pilot application. We describe the application and properties of used graph data, describe the evaluation setting and present achieved performance characteristics.

Experiments were conducted over data set of Wikipedia link graph. The graph structure was generated by a custom parser from Wikipedia XML dump (dump from 03.11.2009 was used). The resulting graph contained 3.3 million nodes and over 91 millions edges. As shown by previous research [15], Wikipedia link graph exhibits small-world properties. Small-world networks are characterized by small average path length between nodes and high values of the clustering coefficient.

In the first part of the evaluation, we have studied the performance of a general purpose graph database for SA technique, in which the link weights and types are modeled as edge attributes. We have compared the time required to retrieve edges for a randomly chosen node without and with weight and type data. We have used Neo4j[10]), an open source graph database as a general purpose graph database for the tests. The default settings of Neo4j database were used in the tests. The experiment was conducted in a black-box testing fashion.

Wikipedia link graph data set was used in the experiment. All the experiments were conducted on a PC with 2GHz Intel Core 2 Duo-processor and 7200 RPM hard drive.

First, we have generated lists of identifiers of randomly chosen nodes from the graph. We have created lists containing 10, 100, 1 000 and 10 000 node identifiers; we used 10 identifier sets for each. This setup was used to test the random access pattern that is typical in SA computation. Pregenerated lists of identifiers were used to ensure the same conditions for distinct tests. We have then measured the time required to retrieve both outgoing and incoming edges for the nodes in the lists. We used generic graph database to retrieve edges without retrieving weight and type attributes. Next, we have measured the time required to

[8] https://sourceforge.net/projects/simplegdb/

[9] http://www.oracle.com/technology/products/berkeley-db/je/index.html (visited: 10.12.2009)

[10] http://neo4j.org/ (visited: 10.12.2009)

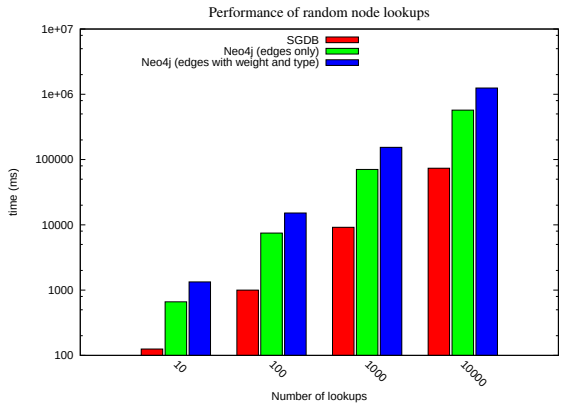

Fig. 3. Time required to perform sets of random lookups; X-axis depicts the number of lookups per set; Y-axis represents the time required to perform lookups. Log scale is used for both axes.

retrieve the edges for the same nodes, but with weight and type data. In average, the time required to retrieve weighted and typed edges was 2.1 times higher than the retrieval of edges without weight and type attributes.

We have performed the same experiments using SGDB, where the weight and type data are stored together with link targets. The edges retrieval (with weight and type data) was 16.9 times faster compared to the retrieval of typed and weighted edges from general purpose graph database. The reason is that in SGDB the weight and type data are coupled together with edges definitions, so only one disk access can be used to read all the data for the SA expansion from a given node; in addition, SGDB random access operation for retrieval of node's edges was slightly faster. Figure 3 shows histogram of the time required to retrieve the edges using SGDB, and general purpose graph database (with and without retrieving weight and type attributes).

In the second series of tests, we have run our pilot application. The pilot application aims at finding connections between two or more given input nodes in Wikipedia link graph. Nodes in the Wikipedia link graph model articles and edges represent links between articles. The pilot application uses the activation spreading over the Wikipedia link graph to find highly activated nodes (named connecting nodes), and identifies the paths with the highest sum of activation on the nodes between initial nodes to connecting node.

In each test, we have measured the time required for finding connections between two randomly chosen nodes, constrained to two iterations of activation spread. Under the two iterations constraint, we can identify the connections of the maximal length of 4 between two initial nodes. Because of the small average distance between nodes in the test set (one of the properties of the small-world graphs), we found connections for randomly chosen nodes in 86.4% of cases.

We have performed 1000 queries, both incoming and outgoing links were used, decay parameter was set to 1 (meaning no decay in iterations) and activation threshold was set to 0.000001. The effect of this setting was the full breath first expansion from the initial nodes in two iterations.

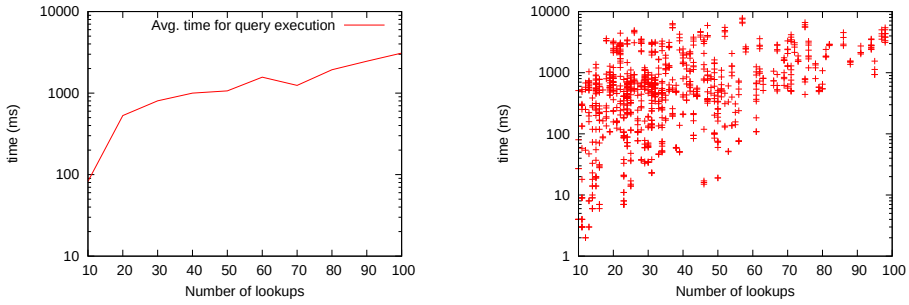

Fig. 4. X-axis represents number of node lookups in queries. Y-axis (log scale) represents time in ms the queries took to execute. Plot on the left represent averaged values and figure on the right depicts points representing values for individual queries.

Figure 4 depicts the time required to execute queries with an increasing numer of node lookups for a query. The average execution time was 1136.4 ms, the activation values for 181820.1 nodes in average was computed for a single query. The average number of edge retrieval operations from the SGDB storage was 49.7.

8 Conclusion

In this paper, we have proposed a storage model for a graph database, designed to provide fast data store for execution of the Spreading Activation (SA) technique. In addition, we have presented the architecture and implementation of SGDB, the graph database that utilize proposed storage model.

We have compared performance of our approach with the performance of a general purpose graph database, for the activation spreading over the stored graph. The evaluation showed important time savings using proposed approach. As our approach was designed for a specific problem, it is not surprising that it performs better (for that problem) than a generic one. However, we believe that the SA technique has a wide number of possible uses in context of graph databases and it is worth to exploit the optimization for the SA even at the graph structure storage level.

We have also proposed a query type for the Spreading Activation operation over the graph database. The SA query has an easily interpretable definition and results and the upper bound of query selectivity can be easily estimated. We have described the performance characteristics of SGDB, using proposed SA operation in the scope of our pilot application that exploits the Wikipedia link graph.

Future work will go in two directions: 1) extending the support for a more general query language, and 2) providing support for parallel implementations.

References

1. M. Amann, B. Scholl. Gram: A graph data model and query language. In *Proceedings of the European Conference on Hypertext Technology (ECHT)*, pages 201–211. ACM, 1992.

2. R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1):1–39, 2008.
3. M. R. Berthold, U. Brandes, T. Kötter, M. Mader, U. Nagel, and K. Thiel. Pure spreading activation is pointless. In *CIKM '09: Proceeding of the 18th ACM conference on Information and knowledge management*, pages 1915–1918, New York, NY, USA, 2009. ACM.
4. K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250, New York, NY, USA, 2008. ACM.
5. M. Ciglan, E. Rivière, and K. Nørvåg. Learning to find interesting connections in wikipedia. In *Proceeding of APWeb 2010*, 2010.
6. F. Crestani. Application of spreading activation techniques in information retrieval. *Artif. Intell. Rev.*, 11(6):453–482, 1997.
7. O. Erling and I. Mikhailov. RDF support in the virtuoso DBMS. In *Conference on Social Semantic Web*, volume 113 of *LNI*, pages 59–68. GI, 2007.
8. M. Gyssens, J. Paredaens, and D. V. Gucht. A graph-oriented object model for database end-user. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data.*, pages 24–33. ACM Press, 1990.
9. J. Hidders. A graph-based update language for object-oriented data models. In *Ph.D. dissertation*. Technische Universiteit Eindhoven, 2001.
10. U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM '09. Ninth IEEE International Conference on*, pages 229–238, Dec. 2009.
11. A. Kiryakov, D. Ognyanov, and D. Manov. OWLIM - a pragmatic semantic repository for OWL. In *Proc. Workshop Scalable Semantic Web Knowledge Base Systems*.
12. M. Levene and A. Poulovassilis. The hypernode model and its associated query language. In *Proceedings of the 5th Jerusalem Conference on Information technology.*, pages 520–530. IEEE Computer Society Press, 1990.
13. M. Mainguenaud. Simatic XT: A data model to deal with multi-scaled networks. In *Comput. Environ. Urban Syst. 16*, pages 281–288, 1992.
14. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 6–6, New York, NY, USA, 2009. ACM.
15. A. Mehler. Text linkage in the wiki medium: A comparative study. In *Proceedings of the EACL 2006 Workshop on New Text: Wikis and Blogs and Other Dynamic Text Sources*, pages 1–8, 2006.
16. J. Paredaens, P. Peelman, and L. Tanca. G-Log: A graph-based query language. In *IEEE Trans. Knowl. Data Eng. 7*, pages 436–453. IEEE, 1995.
17. K. Rohloff, M. Dean, I. Emmons, D. Ryder, and J. Sumner. An evaluation of triple-store technologies for large data stores. In R. Meersman, Z. Tari, and P. Herrero, editors, *OTM Workshops (2)*, volume 4806 of *Lecture Notes in Computer Science*, pages 1105–1114. Springer, 2007.
18. A. Troussov, M. Sogrin, J. Judge, and D. Botvich. Mining socio-semantic networks using spreading activation technique. In *International Workshop on Knowledge Acquisition from the Social Web (KASW'08)*.