Efficient Graph Management Based On Bitmap Indices

Norbert Martínez-Bazan
DAMA-UPC
Universitat Politècnica de Catalunya
Barcelona, Spain
nmartine@ac.upc.edu

Victor Muntés-Mulero[*]
CA Labs
Barcelona, Spain
Victor.Muntes-Mulero@ca.com

Sergio Gómez-Villamor
Sparsity Technologies
Barcelona, Spain
sgomez@sparsity-technologies.com

M.Ángel Águila-Lorente
DAMA-UPC
Universitat Politècnica de Catalunya
Barcelona, Spain
maguila@ac.upc.edu

David Dominguez-Sal
DAMA-UPC
Universitat Politècnica de Catalunya
Barcelona, Spain
ddomings@ac.upc.edu

Josep-L. Larriba-Pey
DAMA-UPC
Universitat Politècnica de Catalunya
Barcelona, Spain
larri@ac.upc.edu

ABSTRACT

The increasing amount of graph like data from social networks, science and the web has grown an interest in analyzing the relationships between different entities. New specialized solutions in the form of graph databases, which are generic and able to adapt to any schema as an alternative to RDBMS, have appeared to manage attributed multigraphs efficiently. In this paper, we describe the internals of DEX graph database, which is based on a representation of the graph and its attributes as maps and bitmap structures that can be loaded and unloaded efficiently from memory. We also present the internal operations used in DEX to manipulate these structures. We show that by using these structures, DEX scales to graphs with billions of vertices and edges with very limited memory requirements. Finally, we compare our graph-oriented approach to other approaches showing that our system is better suited for out-of-core typical graph-like operations.

Categories and Subject Descriptors

E.1 [**Data Structures**]: *Graphs and networks*; H.2.8 [**Database Management**]: Database applications; H.3.2 [**Database Management**]: Information storage

General Terms

Algorithms, Design, Performance

[*]Victor Muntés collaborated in this project while he was at UPC and a member of DAMA-UPC.

Keywords

Data Representation, Graph Databases, Query Performance

1. INTRODUCTION

The increasing number of huge networks such as the Web, geographical systems, social networks, or those created to represent the interaction between proteins in biological systems, has brought the need to manage information with inherent graph-like nature [1]. In this scenario, the natural way to represent the information and the results is by means of very large graphs. The interest for analyzing the interrelation between the entities in these networks is rapidly increasing, forcing the creation of information management systems that are able to perform graph-oriented operations efficiently.

In the recent literature, we find two approaches to graph analysis. The first approach focuses on performing complex analysis computations on large graphs. For this purpouse, the usual architecture is a distributed system with large scalability and some graph oriented operations. Some examples are Pegasus [12] (a map-reduce library that computes graph operations by means of large scale matrix computations), or Pregel [16] (a graph computation model based on the exchange of messages between vertices). However, the architecture of these systems limits the set of available operations and requires the codification of graph algorithms specially adapted to such architectures. The second approach is focused on providing database functionalities to store and query graph-like data. As a consequence, several commercial products for graph management have appeared, such as Neo4J [19], HyperGraphDB [9], or OrientDB [22], which store the graph in a file system (which can be a local file system, a shared disk...) and perform computations in a computing node relying on a buffer pool and an efficient secondary memory access. For large sites, the system can scale its throughput using a distributed system where a front-end load balancer [4] distributes the load among available servers running the graph database such as Neo4J HA [20]. In this paper, we describe the internal implementation of DEX [6], which focuses on the second approach.

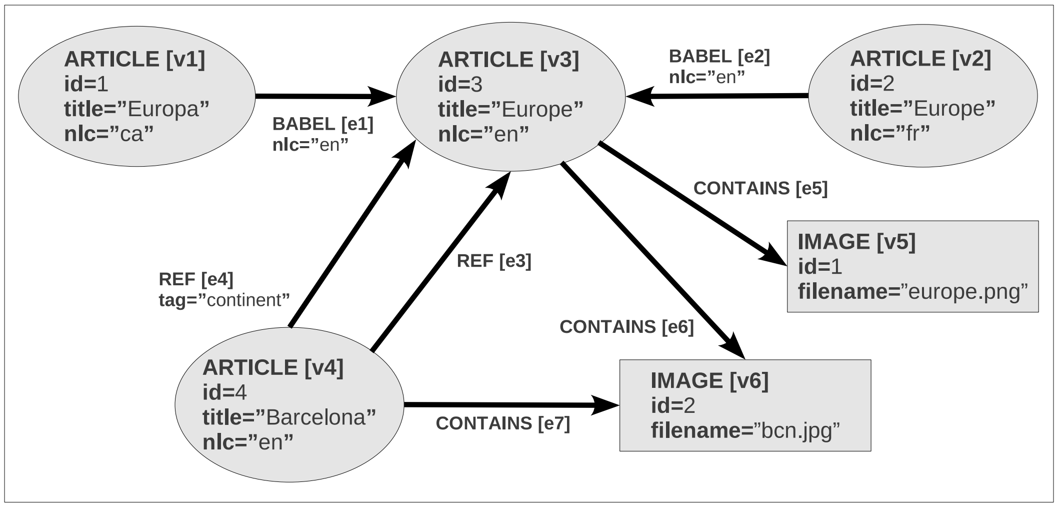

Figure 1: Example of a labeled and directed attributed multigraph.

The requirements for graph databases are varied to support the diversity of workloads. Graph queries often combine graph traversals with intensive attribute accesses. For instance, the best route in a network mapping a country roadmap can be constrained by several factors such as the road type or the existence of a gas station in that road. Therefore, the existence of attributes attached to both vertices and edges becomes necessary. Furthermore, most applications naturally demand for multigraphs: two authors might have collaborated coauthoring a paper more than once, two cities will be connected by more than two roads, etc. Moreover, the system must internally provide an efficient secondary memory management: (i) answering a query should not imply loading the whole graph into memory, (ii) vertices and edges should be very compact in memory and should be accessed very efficiently, (iii) typical graph-oriented operations such as edge navigation should be executed very efficiently, and (iv) all the attribute values in the graph should be accessed very fast.

In this paper, we detail DEX internal representation for large-scale labeled and directed attributed multigraphs based on collections of object identifiers. In a preliminary work [17], we presented the capabilities of DEX describing its application to information retrieval tasks and the integration of multiple data sources. However, in [17] the internal data structures and implementation of DEX were not described nor analyzed. In this paper, we show that, by using bitmaps for the implementation of these collections, we reduce the cost of the most common graph operations, improving the overall performance of a graph database system. The two important aspects that make bitmaps suitable are that: (i) they allow keeping a large amount of information in a relatively reduced amount of space, usually by using presence bits or compression techniques, and (ii) they can be operated very efficiently by using binary logic operations. Finally, we show how a set of basic operations exploit the efficiency provided by our internal organization for very large graphs.

Main Contributions. In this work, (i) we detail our model to represent labeled and directed attributed multigraphs; (ii) we propose a simple but efficient implementation for this model by using a vertical partitioning based on a single structure composed by collections of oids; and (iii) we present an analysis of the complexity of both time and space of our proposed data structures. We also prove experimentally the querying performance and scalability of our system, showing that it is better suited than other approaches for the most typical graph-like operations.

This paper is organized as follows. In Section 2, we present our proposal of a graph representation based on object identifier collections. Section 3 presents the specialized structures proposed in this paper and several examples of the construction and querying of a graph using bitmap-based operations. Experimental results are presented in Section 4. Section 5 presents some related work in graph databases. Finally, in Section 6 we draw some conclusions and future work.

2. GRAPH MODEL

In our proposal, the basic logical data structure is a labeled and directed attributed multigraph. A *labeled* graph has a label for each vertex and edge. This label is used as the type for that vertex or edge. A *directed* graph allows for edges with a fixed direction, from the *tail* or source vertex to the *head* or destination vertex. An *attributed* graph allows a variable list of attributes for each vertex and edge, where an *attribute* is a value associated to a name, simplifying the graph structure. A *multigraph* allows multiple edges between two vertices. This means that two vertices can be connected several times by different edges, even if two edges have the same tail, head and label.

Instead of the classical definitions of a graph based on matrices or collections representing the materialization of adjacency between vertices, we propose a new representation that splits the graph into multiple lists of pairs of values, where the first element of the pair is always a vertex or an edge, and the second element of the pair is a value. This is the basis of our implementation based on an efficient management of bitmaps and other auxiliary structures, the *value sets*, as explained later on in Section 3.

Table 1: Wikipedia transformation into value sets

Collection	Graph G	Transformed Graph G
L	$(v_1, \texttt{ARTICLE}), (v_2, \texttt{ARTICLE}), (v_3, \texttt{ARTICLE}), (v_4, \texttt{ARTICLE}), (v_5, \texttt{IMAGE}), (v_6, \texttt{IMAGE}), (e_1, \texttt{BABEL}), (e_2, \texttt{BABEL}), (e_3, \texttt{REF}), (e_4, \texttt{REF}), (e_5, \texttt{CONTAINS}), (e_6, \texttt{CONTAINS}), (e_7, \texttt{CONTAINS})$	$(\texttt{ARTICLE}, \{v_1, v_2, v_3, v_4\}), (\texttt{BABEL}, \{e_1, e_2\}), (\texttt{CONTAINS}, \{e_5, e_6, e_7\}), (\texttt{IMAGE}, \{v_5, v_6\}), (\texttt{REF}, \{e_3, e_4\})$
T	$(e_1, v_1), (e_2, v_2), (e_3, v_4), (e_4, v_4), (e_5, v_3), (e_6, v_3), (e_7, v_4)$	$(v_1, \{e_1\}), (v_2, \{e_2\}), (v_3, \{e_5, e_6\}), (v_4, \{e_3, e_4, e_7\})$
H	$(e_1, v_3), (e_2, v_3), (e_3, v_3), (e_4, v_3), (e_5, v_5), (e_6, v_6), (e_7, v_6)$	$(v_3, \{e_1, e_2, e_3, e_4\}), (v_5, \{e_5\}), (v_6, \{e_6, e_7\})$
A_{id}	$(v_1, 1), (v_2, 2), (v_3, 3), (v_4, 4), (v_5, 1), (v_6, 2)$	$(1, \{v_1, v_5\}), (2, \{v_2, v_6\}), (3, \{v_3\}), (4, \{v_4\})$
A_{title}	$(v_1, \texttt{Europa}), (v_2, \texttt{Europe}), (v_3, \texttt{Europe}), (v_4, \texttt{Barcelona})$	$(\texttt{Barcelona}, \{v_4\}), (\texttt{Europa}, \{v_1\}), (\texttt{Europe}, \{v_2, v_3\})$
A_{nlc}	$(v_1, \texttt{ca}), (v_2, \texttt{fr}), (v_3, \texttt{en}), (v_4, \texttt{en}), (e_1, \texttt{en}), (e_2, \texttt{en})$	$(\texttt{ca}, \{v_1\}), (\texttt{en}, \{v_3, v_4, e_1, e_2\}), (\texttt{fr}, \{v_2\})$
$A_{filename}$	$(v_5, \texttt{europe.png}), (v_6, \texttt{bcn.jpg})$	$(\texttt{bcn.jpg}, \{v_6\}), (\texttt{europe.png}, \{v_5\})$
A_{tag}	$(e_4, \texttt{continent})$	$(\texttt{continent}, \{e_4\})$

We annotate formally the previously presented description of a graph as a collection of sets. Let $V = \{v_1, \ldots, v_n\}$ be a finite set of vertices and $E = \{e_1, \ldots, e_m\}$ a finite set of directed edges. A non directed edge can be represented as two opposing directed edges. We represent the relations among vertices as two sets: $T = \{(e_1, t_1), \ldots, (e_m, t_m)\}$ is the set of tail pairs (e_i, t_i), which indicates that the tail of e_i is the vertex $t_i \in V$, and $H = \{(e_1, h_1), \ldots, (e_m, h_m)\}$ is the set of head pairs (e_i, h_i), which indicates that the head of e_i is the vertex $h_i \in V$. Given an *object* o, which is either an edge or vertex ($o \in \{V \cup E\}$), we map a single label to each object $L = \{(o, l) \mid o \in (V \cup E), l \in \textit{string}\}$. We represent attributes as sets of pairs, $A_i = \{(o_1, c_1), \ldots, (o_r, c_r)\}$, which assign an attribute value $c_i \in \mathcal{D}$ (where $\mathcal{D}$ are the valid data types such as int, boolean, timestamp, etc.) to a set of objects, $r \leq n + m$. Finally, we describe formally the previously presented directed attributed multigraph with p attributes as $G = (V, E, L, T, H, A_1, \ldots, A_p)$.

Note that there is not any restriction on the domain of the data type of all the values of an attribute. In our proposal, the attributes are not constrained to a single data type, and a more restricted vision with strong-typed attributes is left as an integrity constraint to be guaranteed by the higher level operations which insert or update attribute values. Note also that an edge with the same vertex as tail and head, a *directed reflexive* edge, is also supported with this representation due to the separation of tails and heads into two different lists of pairs.

Figure 1 shows a small example of a graph extracted from Wikipedia. In this example, the model has two types of vertices: Wikipedia articles (`ARTICLE`), image files (`IMAGE`), and three types of edges: references between articles (`REF`), links to images (`CONTAINS`) and translations of articles (BABEL). Each node and vertex can have a variable number of attributes associated. The representation of this graph is shown in the left column of Table 1.

3. GRAPH REPRESENTATION

In this section, we present a proposal to implement the graph model presented in the previous section. We summarize the key aspects necessary to obtain a successful structure to implement the model: (i) our system must be able to store huge amounts of object sets and access them efficiently, (ii) given a key, it must be able to retrieve the set of objects associated to that key and (iii) given an object, it must be able to efficiently retrieve the set of values associated to that object.

3.1 Bitmap based representation

We transform the sets presented in the previous section to a structure that we call *value set*, which contains the same information but is more compact. The value set groups all pairs of the original set with the same value as a pair between the value and the set of objects with such value. The righmost column of Table 1 shows the transformation of the running example graph into eight value sets.

We assume that each vertex $v \in V$ and edge $e \in E$ is identified by a unique *vid* $\in \mathbb{N}$ or *eid* $\in \mathbb{N}$, respectively. The *vids* and *eids* are also known as *oids* or object identifiers. For the sake of simplicity, we consider oids as logical, unique and immutable. We assume also that the graph database manager has an oid generator to provide a unique oid to each new vertex or edge when they are created.

We represent a value set as two maps: one maps each object to a value, while the other maps each different value to a bitmap (see Figure 2). In order to represent a vertex or edge set, we propose to use a bitmap structure. A bitmap is a variable-length sequence of 0's and 1's. Each position in a bitmap corresponds to the identifier of an object *oid* (a *vid* or an *eid*). If the object is included in the set its corresponding position is set to 1, and 0 otherwise. As previously introduced, the motivation is twofold. First, bitmaps keep a large amount of information in a reduced amount of memory. Second, by using bitmaps, binary logic operations can be performed very efficiently to manipulate object sets.

All queries and modifications of the graph are performed by combining five basic operations on value sets: *domain*, which returns the set of distinct values; *objects*, which returns the set of vertices or edges associated to a value; *lookup*, which returns the set of values associated to a set of objects containing a particular vertex or edge; *insert*, to add a vertex or edge to the collection of objects of a value; and, finally, *remove*, which removes a vertex or edge from the collection of objects of a value. For space constraints, we do not report the whole mapping of the classical graph operations to these five value set operations, but we show by examples the mapping of some functions to these operations in Section 3.3.

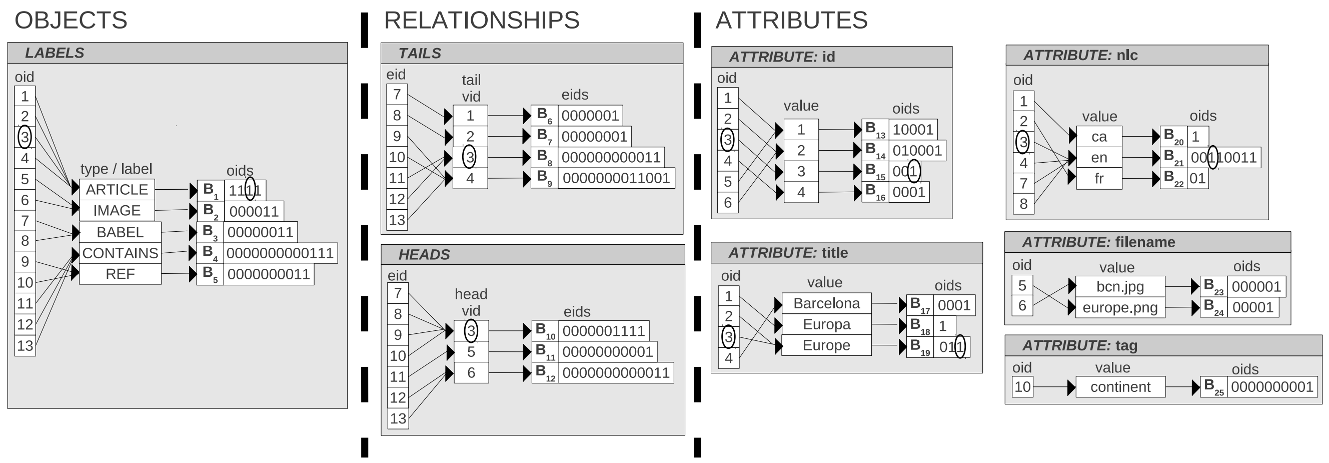

Figure 2: Example of the representation of a graph through the proposed structures.

3.2 Bitmap-based Representation of the Running Example

Figure 2 depicts the structures that represent the graph in the running example presented in Figure 1. We have graphically divided the structures into three different groups. The first group contains the structure that allows for accessing the objects in the graph (OBJECTS group), including vertices and edges. The second group is divided into two sets of structures that are related to the connectivity between objects in the graph (RELATIONSHIPS group). And, the third group contains all the structures for efficiently accessing attribute values in an object or obtaining the objects related to a certain value (ATTRIBUTES group).

In this example, we assign arbitrary oid values to the vertices and to the edges. As a convention, in this figure all bitmaps start their sequence in position 1 (the first valid *oid*) and finish in the last position set to 1 (the highest *oid* considered in the bitmap).

The figure shows eight value sets and their bitmaps, following the definitions introduced in Section 2: one for the labels (or types) of vertices and edges; one for the tails; one for the heads; and, finally, one for each of the 5 attributes. Remember also that attribute "ID" is a unique key value only for articles and images (the vertices) and it is not the same as the vertex *oid*.

In this example, we have marked value "3" in all the maps and bitmaps of all value sets to identify the label, attributes and relationships of the English article for "Europe", which corresponds to the vertex with *vid*=3. Thus, bitmap B_1 shows that its label is `ARTICLE`. Its attribute values are "Europe" for `title` in bitmap B_{19} and "en" for `nlc` in bitmap B_{21}. Finally, bitmap B_8 shows that it is the tail of edges 11 and 12, and bitmap B_{10} contains the *eids* of the four edges where it is the head.

Our graph model satisfies the following integrity rules:

- The union of all the bitmaps in LABELS represents all the objects of the graph:

$$V = B_1 \cup B_2; E = B_3 \cup B_4 \cup B_5; O = V \cup E$$

- An object only has one label, i.e. all the bitmaps $\{B_1, B_2, B_3, B_4, B_5\}$ in LABELS are pairwise disjoint.
- The bitmaps in TAILS and HEADS only contain edges, and each edge has both a tail and a head:

$$B_6 \cup B_7 \cup B_8 \cup B_9 = B_{10} \cup B_{11} \cup B_{12} = E$$

- Each edge has one single tail and one single head, i.e. all the bitmaps $\{B_6, B_7, B_8, B_9\}$ in TAILS are pairwise disjoint, and all the bitmaps $\{B_{10}, B_{11}, B_{12}\}$ are also pairwise disjoint.
- Each object, vertex or edge, has zero or one value for each attribute, i.e. all the bitmaps in the value set of an attribute are pairwise disjoint. For example, in attribute `id`, all its bitmaps $\{B_{13}, B_{14}, B_{15}, B_{16}\}$ are pairwise disjoint.

3.3 Bitmap-based Operations with the Running Example

With these structures, it is very easy to define graph-based operations just by combining one or more bitmaps and map accesses. For example:

- Number of articles:

$$|objects(\texttt{LABELS}, \texttt{'ARTICLE'})| = 4$$

- Articles in French or Catalan:

$$(objects(\texttt{NLC}, \texttt{'fr'}) \cup objects(\texttt{NLC}, \texttt{'ca'})) \\ \cap\, objects(\texttt{LABELS}, \texttt{'ARTICLE'}) = \{1, 2\}$$

- Out-degree of English article "Europe":

$$|objects(\texttt{TAILS}, objects(\texttt{TITLE}, \texttt{'Europe'}) \\ \cap\, objects(\texttt{NLC}, \texttt{'en'}) \\ \cap\, objects(\texttt{LABELS}, \texttt{'ARTICLE'}))| = 2$$

- Articles with references to the image with `filename` "bcn.jpg":

$$\{lookup(\texttt{TAILS}, x) \mid \\ x \in objects(\texttt{HEAD}, objects(\texttt{FILENAME}, \texttt{'bcn.jpg'}) \\ \cap\, objects(\texttt{LABELS}, \texttt{'IMAGE'}))\} = \{3, 4\}$$

- Count the articles of each language:

$$\{(x, y) \mid x \in domain(\texttt{NLC}) \\ \wedge\ y = |(objects(\texttt{NLC}, x) \\ \cap\ objects(\texttt{LABELS}, \texttt{'ARTICLE'}))|\}$$

- Update the filename of vertex 6 to "barcelona.jpg":

$$remove(\texttt{FILENAME}, lookup(\texttt{FILENAME}, 6), 6)$$
$$insert(\texttt{FILENAME}, \texttt{'barcelona.jpg'}, 6)$$

3.4 Implementation Details and Complexity Analysis

All the structures presented have been implemented in the current version of the DEX core engine [17]. In DEX, vertices and edges have unique logical oids. These logical oids have no dependencies on the physical location of the object, allowing a dynamic reorganization of the graph for performance purposes.

Each oid is represented as a single bit in a bitmap, using a word aligned scheme, which compresses long sequences of zeroes. Bitmaps are compressed by grouping the bits into clusters of 32 consecutive bits. Only clusters with at least one bit set are stored. A bitmap is then a sequence of pairs with the 32-bit integer cluster identifier followed by their 32-bit integer cluster data. The pairs are stored in a sorted tree structure to speedup the *lookup*, *insert* and *remove* operations. In order to know the length of the bitmap, the number of actual bits set to 1 in the structure is kept updated. The current version supports 37-bit unsigned integer *vids* and *eids*, allowing more than 137 billion objects per graph ($2^{37} = 2^{32}\ clusters * 2^5\ bits\ per\ cluster$). These identifiers are clustered in groups for each vertex or edge type. This approach makes it easier to find which is the type of one object, and improves the locality in the bitmaps due to a higher density of consecutive bits set to 1. The space complexity of this bitmap representation is $O(n)$, since in the worst case, a constant number of bits is needed to represent an oid. The word alignment facilitates the implementation of efficient set operations, such as the union or the intersection. Although there exists many other efficient bitmap representations for bitmap indices that have very good compression ratios such as WAH [25], the representation that we implemented experimentally proved to be enough efficient and space-wise for our purpouses. It is out of the scope of the present work to study the best bitmap representation for the proposed architecture.

Maps are implemented using B+ trees. For attributes that are strings, we store in the map only the identifier of the string. The full string is retrieved from a sequential storage of utf-8 char sequences. The space complexity of a B+ tree with n keys is $O(n)$. Other options for maps may be hash tables, or perfect hash functions for compressed read-only graphs.

A value set is built with the combination of two B+ trees plus a collection of bitmaps. The space complexity of a value set with x active oids and v diferent values where $v \leq x$ is $O(x)$, which is the largest of the space complexity $O(x)$ for the B+ tree between oids and values, and $O(v)$ for the B+ tree between the values and their bitmaps $O(x)$. Thus, the space complexity of a graph with n vertices, m edges, l labels and a attributes is $O(n + m)$, which is the space complexity of the largest value set with $n + m$ oids. Also, the time complexity of the five value set operations is $O(v)$ for domain, $O(\log v)$ for *objects* and $O(\log x)$ for *lookup*, *insert* and *remove*, where x is the number of *oids* and v the number of distinct values.

The tail and head value sets have been split into specific value sets for each edge type that contains only the references to edges of the type, achieving a more efficient memory management. Attributes are strong-typed and they have been split also for each different type with the same attribute name. Attribute values do not have specific compression because they are represented with the same basic data structure as any other piece of data in the graph, and this structure already contains only the distinct values and the compressed representation of bitmaps.

4. EXPERIMENTAL RESULTS

In this section, we present several experiments in order to test the capability of the structures proposed in this paper to analyze and query graph-structured data in a single computer with limited physical memory. These experiments are classified into three categories: (i) comparison of our approach with other solutions; (ii) core operations performance and memory usage analysis and (iii) scalability.

4.1 Experimental setup

The experiments are performed using a computer with an Intel(R) Xeon(R) E5440 CPU at 2.83 GHz. The memory hierarchy is organized as follows: 6144 KB second level cache, a 64 GB main memory and a disk with 1.7 TB. The operating system is Linux Debian etch 4.0. For all the experiments, each query has been executed five times, and the slowest and the fastest results have been discarded. The reported resulting time is the average of the remaining three results. The first execution of each experiment acts like a warm-up phase, being discarded as the slowest result.

Dataset and queries: The dataset selected for the experiments is Wikipedia (wikipedia.org), the well known web-based, multilingual encyclopedia project. The Wikipedia articles have hyperlinks to other articles, to images or to the translation of the articles. From the original XML files of the January 2010 Wikipedia dump, we have extracted near 55 million articles from 254 different languages, 2.1 million images and more than 321 million links. For each article, we store the title and the language; for the images, we store the filename and the language; and finally, for the links, we create an edge between the source article, the destination article or image, the link type and, optionally, the language for its translations. Figure 1 shows an example of a Wikipedia graph following the previous representation format.

Instead of running specialized algorithms for restricted domains or problems, we run in all the solutions five different queries which combine the navigation through the graph relationships with the filter by attribute values; some of them aggregate the results, while others update attributes. All queries access a large part of the Wikipedia graph by traversing many edges or by retrieving attribute values for vertices or edges. These queries are:

Query 1: *Find the article with the largest outdegree and find a shortest-path tree (SPT).* This query finds the article a with the largest number of ARTICLES referenced. Then, it traverses the graph with a breadth first search (BFS) starting in a and only considering the edges of type REF (taking into account the direction). The output of this query is the ID, TITLE and outdegree of a.

Query 2: *Recommend articles related to the most popular one.* This query finds the article a with the largest indegree. Then, it performs a two hop exploration following the REF edge type: let $N(x)$ be the set of neighbours of x, the query computes $S_a = N(a)$ and $S_r = N(S_a)$. The query returns the attributes ID and TITLE, as well as the outdegree of a, and the top-5 articles in S_r sorted in descending order by the count of paths.

Query 3: *Find new images for articles from articles in other languages.* This pattern matching query returns the number of articles written in Catalan that are translated to English and that do not contain any of the images of the English version.

Query 4: *Find, for each different language, the number of articles and the number of images referenced by those articles without repetition.* This grouping query returns a list of all the languages sorted alphabetically by name in ascending order together with the number of articles and the number of distinct referenced images.

Query 5: *For each article, materialize an attribute indicating the number of images contained, only if it contains more than one.* The output of this update query is the number of articles with at least one image.

These queries are representative of different classes of graph queries: k-hops and path traversals (Q1, Q2), graph pattern matching (Q3), aggregations and edge connectivity (Q4), and graph transformation (Q5). Without the use of specific indexes, the classical graph problems are a combination of these query classes.

Systems tested. We compare the following implementations to DEX:

Memory-based incidence matrix: We implement a sparse incidence matrix using incidence lists. It is a simple but fast representation for in-memory large graphs, which we use as baseline. The solution has two large memory vectors, one for the vertices and another one for the edges. Each vertex is a structure with the vertex type (the label identifier), a pointer to the attributes record, and two incidence lists: the outgoing edges and the ingoing edges. The attributes record contains the length and a sequence of pairs, the attribute identifier and the attribute value. Each incidence list is a vector of incident edges and its total number. Also, each edge is a structure with the edge type (the label identifier), the tail and the head vertex identifiers. This representation allows for updates as well as efficient edge traversal but it is restricted to the physical memory available for the process. By using graph compression techniques, we could reduce the memory requirements (but increase the computational cost) and the out-of-core scenario would still exist for larger graphs.

MonetDb: MonetDB is an open-source database system for high performance applications in data mining, OLAP, and XML Query. It implements a vertical fragmentation storage model with automatic and self-tuning indexes and run-time query optimization, oriented to in memory processing. The main reasons for the comparison with MonetDB are its efficient vertical fragmentation representation, close to ours, and its wide acceptance by the research community as a database storage even for relational data. In particular, the basic data structure of MonetDB is the Binary Association Table (BAT), which is a collection of pairs (OID, value). Following our multigraph representation in Section 2 based also on sets of pairs (OID, value), we use one BAT for each of these collections. Thus, we have one BAT for the labels, two for the edge tails and the edge heads, and one more for each attribute. The MonetDB queries have been written using MAL, an assembly-like language for the interface with the MonetDB kernel.

MySQL: MySQL is a well-known and widely used relational database engine. We used the public domain version of MySQL 5.0.51a for Debian Linux, and for the different memory configurations for the experiments we tuned the *my.cnf* file following the MySQL tuning server recommendations (`http://dev.mysql.com/doc/refman/5.6/en/server-parameters.html`). Some of the server parameters modified have been the *read*, *key* and *sort* buffers, among others. We modeled the graph into five tables: one entity for each vertex type and its attributes, and one many-to-many relationship for each edge type and its attributes. Each table has a primary key and the other columns used in queries are indexed to provide a full indexed access. The whole indexing was made in a second phase after the bulk load to make all the database creation process more efficient.

Neo4J: Neo4J is a fully transactional commercial graph database implemented in Java. It has its own disk-based native storage manager and an edge traversal framework for query resolution. Three specific Lucene v3.1.0 indexes were created for three attributes to run the queries faster.

Our query implementations are single threaded with calls to the corresponding system. In the case of the systems implemented by us (DEX and incidence matrix) we are not either exploiting internal thread parallelism in the calls.

4.2 Performance analysis

In this experiment, we check the four architectures that have a buffer pool management. We limit the available memory for the buffer pool to 1 GB for all systems, which is approximately one order of magnitude smaller than the data source files in plain text (we give more memory to Neo4J because their file image was significantly larger than the data source files). This experiment tests the main environment for which DEX is oriented: graph queries in databases that do not fit in memory. The load time, the graph size and the execution time for the five queries are reported in Table 2.

Table 2: Wikipedia Benchmark out-of-core, 1 GB buffer pool.

	MonetDb	MySQL	Neo4j*	DEX
Graph Size (GB)	**12.00**	15.72	42.00	16.98
Load (h)	*Error*	**1.36**	8.99	2.89
Q1 (s)	4,801.6	> *12 h.*	> *12 h.*	**120.5**
Q2 (s)	3,788.4	13,841.6	> *12 h.*	**205.4**
Q3 (s)	458.9	33.0	481	**10.8**
Q4 (s)	279.3	**45.0**	> *12 h.*	144.9
Q5 (s)	267.4	930.3	> *12 h.*	**140.9**

(*) Java VM with 45 GB

Table 3: Query Results

Query	results	edge trav.	edge trav./sec	total mem.	bitmaps
Query 1	624,525	236,387,207	1,987,616.30	832.19	42.97%
Query 2	5	261,735,954	1,270,747.94	2,974.50	48.59%
Query 3	51,780	1,536,698	143,885.58	320.81	48.00%
Query 4	254	4,987,879	33,984.32	245.13	77.67%
Query 5	2,401,597	5,934,724	42,072.39	319.00	80.64%

Table 4: Bitmap memory usage (in MB)

	Size	Q1	Q2	Q3	Q4	Q5
LABELS	13.56	11.60	11.60	11.60	11.60	11.60
TAILS	1,272.32	1,030.90	857.09	229.67	164.79	164.79
HEADS	633.98	0.00	506.98	0.00	0.00	0.00
Attr. ID	122.77	0.00	0.00	0.00	0.00	0.00
Attr. TITLE	835.92	0.00	0.00	0.00	0.00	0.00
Attr. NLC	3,618.49	0.00	0.00	791.29	833.64	0.00
Attr. FILENAME	769.79	0.00	0.00	0.00	0.00	0.00
Attr. TAG	31.94	0.00	0.00	0.00	0.00	0.00
TOTAL	7298.77	1,042.50	1375.67	1032.56	1010.03	176.39
Memory used		357.63	1445.31	154.00	190.38	257.25

The best results, in boldface, denote that DEX performs very well in a scenario where there is not enough memory to load the whole database. The use of only the bitmaps required to solve the queries reduces the memory requirements as expected. The other proposals perform, in general, worse than DEX. MonetDb is not able to load the graph (we build the database giving 8 GB to MonetDB, and 1GB to the queries), because it is basically a main-memory database based on memory mapping techniques. All queries are slower than in DEX, specially the traversals (Q1 and Q2) that require scans over the large TAILS and HEADS BATs. MySQL has a good performance for join-based and aggregate queries (Q3 to Q5), but is also very slow for traversals (Q1 and Q2) that involves large nested joins. Finally, Neo4j is not able to run four of the five queries in less than 12 hours even with 45 GB of Java VM memory.

We observe DEX specialization in queries with traversals and attribute accesses through an execution profile. Table 3 shows the results of the execution process: the elapsed *time* in seconds, the number of *results*, the number of *edge traversals*, the average number of *edge traversals per second*, and the size of the *bitmaps* and the *memory* used in MB. Note that all the queries have been solved in a few minutes or even in seconds. Also, all the queries have traversed tenths of thousands of edges per second, and in particular Query 1 has traversed near two million edges per second. The last two columns show in detail the memory usage of the buffer pool in MB for the graph structures and the temporary storage, and the percentage of the buffer pool used by the bitmaps. Four of the queries require less than 1 GB of physical memory: this means that all data fits easily into main memory. In consequence, it reduces I/O and reverts in a more efficient query resolution.

Next, we present a more detailed analysis of the memory requirements and distribution of the bitmaps used to store the graph contents. The graph has been split into a combination of more than 175 million bitmaps of different sizes which occupy 7.2 GB of data. Figure 3 shows the distribution of the bitmaps as a function of the number of set bits. The log-log plot shows that the distribution follows a power-law, with a very large number of small bitmaps with 1 or less than 10 set bits, and only a few large bitmaps. In more detail, the distribution of the count of the number of bitmaps of each size that are used by each query is almost the same as the original bitmap size distribution in Figure 3.

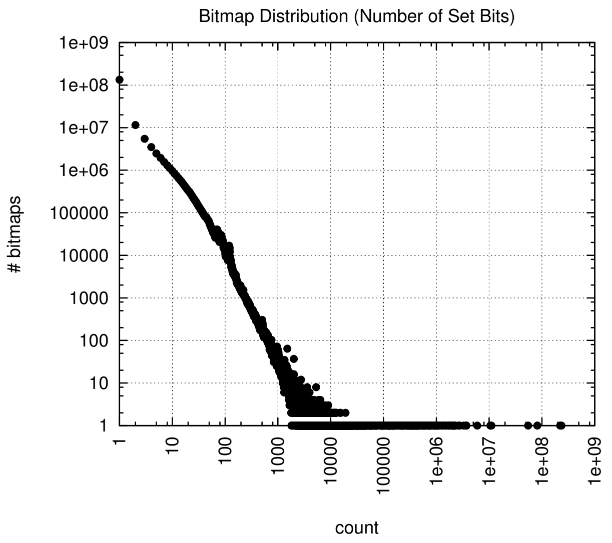

Figure 3: Bitmap size distribution.

For a more detailed analysis of bitmap activity during query processing, Table 4 shows in each row the *size* of the bitmaps of one value set and the size of the bitmaps involved in each query. The last row shows the total amount of memory (buffer pool) used for bitmaps, including some extra memory for padding purposes.

Table 5: Wikipedia Benchmark In-memory

	Matrix	MonetDb	MySQL	Neo4J	DEX
Memory (GB)	20.00	20.00	12.00	45.00	20.00
Graph Size (GB)	15.09	12.00	15.92	42.00	16.98
Load (h)	0.32	0.74	1.15	8.99	2.25
Q1 (s)	12.97	106.14	> *12 h.*	> *12 h.*	118.93
Q2 (s)	51.86	120.50	8,896.00	> *12 h.*	205.97
Q3 (s)	6.28	7.56	29.83	481.00	10.68
Q4 (s)	31.65	84.97	39.71	> *12 h.*	146.77
Q5 (s)	76.15	48.34	909.24	> *12 h.*	141.06

We see that an important number of bitmaps are not used in the queries. Also, in a page-based representation of bitmaps, when only a part of the value set is being used then only some of the total data pages for the bitmaps are loaded into the buffer pool, reducing memory requirements significantly. For example, Q1 uses bitmaps with a total size of 1042.5 MB but only 357.63 MB of them are loaded into memory. On the other hand, Q2 and Q5 use extra memory due to the sparse distribution of object identifiers explored, and the padding of bitmaps into secondary storage fixed size data pages.

In general, all graph queries traverse edges by visiting the TAILS and/or the HEADS bitmaps, but the attribute bitmaps are only used in attribute filters (Q3 and Q4). This means that these bitmaps do not need to be loaded into the buffer pool, which reduces the memory requirements for the complete resolution of the query. For example, Q1 only accesses TAILS but not HEADS, because it only follows the outgoing edges to count the maximum out-degree and to obtain the shortest path tree. Instead, Q2 first uses HEADS bitmaps to look for a specific article with the largest in-degree, but then it follows outgoing edges in TAILS to obtain the related articles.

4.3 Performance in-memory

In this experiment, we configure the buffer pool of the systems with enough memory to store the full database in memory and compare them to an in memory solution for reference. We observe in Table 5 that in-memory oriented solutions, incidence matrices and MonetDB, are generally faster than systems that emphasize the buffer pool management. However, we observe that the execution times are still comparable to in memory solutions. Furthermore, we observe here the scalability of MySQL and DEX. Comparing these results with the experiments executed with a limit of 1 GB of memory, we see that DEX query times are almost the same for both configurations, without an important performance penalty. MySQL also keeps a very good performance except in Q2 that involves edge traversals through large joins.

4.4 Scalability

Finally, we perform a set of tests in order to evaluate the scalability of our proposal. We perform two different experiments. First, we repeatedly execute Q1 for different synthetic graphs with different sizes generated using R-MAT [5], where the generated vertices simulate the Wikipedia articles and the directed edges represent the references between articles. Specifically, we generate graphs with an average degree 8 from scale 20, containing almost one million vertices and more than 8 million edges, to scale 28, containing 230 million vertices and more than 2 billion edges. Second, we fix the input graph to scale 27 and reduce the amount of available memory, to understand the behavior of our proposal as we limit system resources. We have chosen Q1 because it implies reading the graph twice, first it gets all the vertices to get the degree and, afterwards it traverses the whole graph to calculate the SPT.

Figure 4 depicts the execution time of this query as a function of the scale used to generate the graph. The execution time grows linearly as we increase the scale (equivalent to multiplying by 2 the size of the graph at each step). This growth is justified by the increase at each scale of the height of the B+ trees used for the maps, which have a time cost of $\log(n)$ for the *objects* operation over the TAILS value set. Note that these experiments have been run on a buffer pool configured with 60 GB of memory, and only the graph generated with scale 28 does not fit in memory entirely.

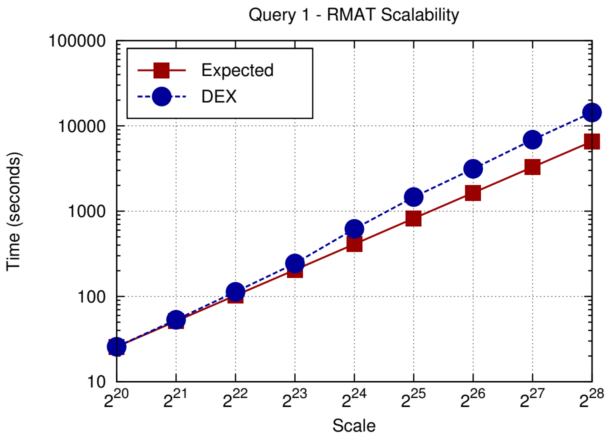

Figure 4: Scalability Test - Changing RMAT scale.

Although the increase of time in this last case follows the same trend as in the previous cases, this experiment is not exhaustive enough to conclude that our proposal is scalable in an out-of-core scenario. For this, we perform a second experiment fixing the graph scale to 27 and reducing the memory from 64 GB to 1 GB. This graph fits in 41 GB of memory, and thus only in the largest memory configuration, the graph fits entirely in memory. Figure 5 shows the scaled execution time for Q1 and different amounts of available memory, i.e. the execution time divided by the execution time obtained when the whole graph fits in memory. From the plot, we observe that by reducing the memory to 1/64th of the original memory we only increase the execution time by a factor of 9. This means that the system scales very well, even when limiting the amount of memory drastically. This is because the query only requires a sequential access over all the bits of the bitmaps of vertex labels in LABELS, and for each of these bits (a vertex) it scans the bitmaps of their TAILS, which in turn are visited only once. Thus, even if all these TAILS bitmaps do not fit in memory, the amount of I/O is small because once replaced they never come back to main memory.

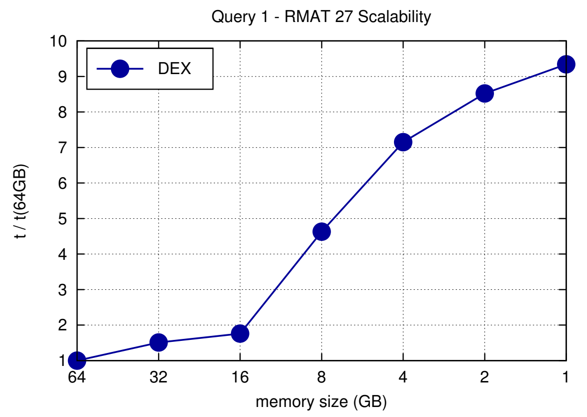

Figure 5: RMAT sf=27 reducing memory size.

5. RELATED WORK

The theoretical basis of a system where both the representation and manipulation of data are based on graphs is introduced in GOOD [7]. With the explosion of the search mechanisms based on keywords for the WEB, there have been several new proposals based on graph data representation like Banks [3] and Précis [14]. Still in the WEB area, several approaches aim at modeling Internet as a graph, like in [15]. More recently, RDF [13] has become the standard for the representation of metadata and Internet resources in the form of a graph, represented by a set of triplets. Some well-known examples of RDF storages are HP-Jena [18] and OpenLink Virtuoso [10]. In the genomics area, a graph database management system to support genome databases is developed in [8], and some specific index structures for graph databases are defined in GRACE [24].

Recently, there has been a renewed interest in the field of general-purpose graph databases that has yielded to several industrial solutions such as Neo4J [19], a Java-based open-source graph database engine; HyperGraphDB [9], an embeddable graph database with generalized hypergraphs; OrientDB [22], an open source document-graph database; or InfiniteGraph [11], a distributed and cloud-enabled graph database. Even more remarkable, Google Inc. has recently presented Pregel[16], a distributed model to compute graph operations on highly scalable distributed systems.

The use of bitmaps indices for efficient data management, since its introduction in the Model 204 DBMS [21], has been exhaustively exploited in very different scenarios such as commercial relational DBMSs (IBM, Informix, Oracle, RedBrick, Sybase IQ). Recently, bitmaps have been used also in the OpenLink Virtuoso [10] RDF triple storage, or for XML indexing in BitCube [27] to obtain statistical measurements and clustering. For a more specific use of bitmap in graph queries, US Patent 2008/0034074 A1, from Unisys Corporation, uses bit-vectors to provide a logical indication of connectivity between vertices to determine different metrics. A more recent proposal in the same area is BitMat [2], which is an in-memory RDF graph storage where the compressed representation serves as the primary storage without being necessary to use any additional indexing technique.

Finally, while in our case the bitmap indices are used for an efficient storage and data retrieval, other graph indexing techniques are widely used to retrieve relevant graphs quickly from very large graph databases, for example in the area of graph substructure search, as summarized in [26], or more specifically for keyword search in graphs [23].

6. CONCLUSIONS

In this paper, we have proposed and modeled a partitioned representation of a graph as a set of generic structures that allows for efficiently answering generic queries on large-scale graphs that may contain billions of attributed vertices and edges. We have also analyzed the space and time complexity of these data structures and its operations, and we have seen that bitmaps are essential for achieving a full-indexed view of a graph database. This is crucial in order to efficiently perform the most typical graph-oriented operations, showing that other models are not as well fitted in terms of efficiency, specially when the graph does not fit entirely in main memory.

The presented indexing technology can be used in other scenarios different from the proprietary graph database management system. For example, the representation of the graph relationships in the form of multiple bitmaps can be embedded in a relational database as auxiliary indexes, where the relational engine would be the responsible for the storage of the node and edge data and the relation-based queries, and these new graph indexes could be used to solve the specific graph-oriented queries as extensions of the standard SQL language. Another example could be RDF, where our bitmap-based approach could be used to improve the performance of the RDF storage and retrieval.

Our bitmap implementation is one among the many possible and it could be certainly improved. But even being naïve with B+trees and simple bitmaps, with these structures we can manage very large graphs (1 billion edges in 1GB of memory). Also, the experiments show that it is more efficient than other solutions currently available for generic attributed multigraphs. We are aware of the fact that, by using a more complex data structure, partitioning methods or bitmap compression, the results could be even better and we plan this research as part of our future work.

Finally, the growth of information will also lead us to the exploration of distributed graph databases, tackling the problem of efficient graph-like data distribution among several partitions. Our graph representation opens different areas of research, such as the vertical partitioning based on *value sets* or similar structures, the bitmap distribution and partitioning among partitions, or even the adaptation of our approach to massive parallel processing models such as MapReduce.

Acknowledgment

The members of DAMA-UPC thank the Ministry of Science and Innovation of Spain and Generalitat de Catalunya, for grant numbers TIN2009-14560-C03-03 and GRC-1187 respectively.

7. REFERENCES

[1] R. Angles and C. Gutiérrez. Survey of Graph Database Models. *ACM Computing Surveys (CSUR)*, 40(1):1–39, 2008.

[2] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit" Loaded: a Scalable Lightweight Join Query Processor for RDF Data. In *Proceedings of the 19th International Conference on World Wide Web*, WWW'10, pages 41–50, New York, NY, USA, 2010. ACM Press.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE'02, pages 431–440, San Jose, CA, USA, 2002. IEEE Computer Society.

[4] V. Cardellini, M. Colajanni, and P. Yu. Dynamic Load Balancing on Web-Server Systems. *Internet Computing, IEEE Computer Society*, 3(3):28–39, 1999.

[5] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the 4th SIAM International Conference on Data Mining*, ICDM'04, Lake Buena Vista, FL, USA, 2004.

[6] DEX. *http://sparsity-technologies.com/.*

[7] M. Gemis, J. Paredaens, I. Thyssens, and J. Van den Bussche. GOOD: A Graph-Oriented Object Database System. In *Proceedings of of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD'93, pages 505–510, Washington D.C., USA, 1993. ACM Press.

[8] M. Graves, E. R. Bergeman, and C. Lawrence. A Graph-Theoretic Data Model for Genome Mapping Databases. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, volume 5 of *HICSS'95*, pages 32–41, Kihei, Maui, HI, USA, 1995. IEEE Computer Society.

[9] HyperGraphDB. *http://www.hypergraphdb.org/index.*

[10] K. Idenhen. Introducing OpenLink Virtuoso: Universal Data Access Without Boundaries, White paper. *http://virtuoso.openlinksw.com/whitepapers/vdb/html/virt10/virtuosowp.pdf.*

[11] InfiniteGraph. *http://www.infinitegraph.com/.*

[12] U. Kang, C. Tsourakakis, and C. Faloutsos. Pegasus: A Peta-Scale Graph Mining System Implementation and Observations. In *Proceedings of the 9th IEEE International Conference on Data Mining*, ICDM'09, pages 229–238, Miami, FL, USA, 2009. IEEE Computer Society.

[13] G. Klyne and J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 10 February 2004. *http://www.w3.org/TR/2004/REC-rdf-concepts-2004.*

[14] G. Koutrika, A. Simitsis, and Y. Ioannidis. Précis: The Essence of a Query Answer. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE'06, page 69, Atlanta, GA, USA, 2006. IEEE Computer Society.

[15] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tompkins, and E. Upfal. The Web as a Graph. In *Proceedings of the 19th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS'00, pages 1–10, Dallas, TX, United States, 2000. ACM Press.

[16] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a System for Large-Scale Graph Processing. In *Proceedings of the 16th International Conference on Management of Data*, COMAD'10, pages 135–146, Nagpur, India, 2010. ACM.

[17] N. Martínez-Bazan, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M.-A. Sánchez-Martínez, and J.-L. Larriba-Pey. Dex: High-performance exploration on large graphs for information retrieval. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management*, CIKM'07, pages 573–582, Lisbon, Portugal, 2007. ACM Press.

[18] B. McBride. Jena: A Semantic Web Toolkit. *Internet Computing, IEEE*, 6(6):55–59, 2002.

[19] Neo4J. *http://neo4j.org.*

[20] Neo4J HA. *http://docs.neo4j.org/chunked/stable/ha-architecture.html.*

[21] P. E. O'Neil. Model 204 Architecture and Performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, HPTS'89, pages 40–59, London, UK, 1989. Springer-Verlag.

[22] OrientDB. *http://www.orientdb.org/index.htm.*

[23] D. Shasha, J. Wang, and R. Giugno. Algorithmics and Applications of Tree and Graph Searching. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS'02, pages 39–52, Madison, WI, USA, 2002. ACM.

[24] S. Srinivasa, M. Maier, M. R. Mutalikdesai, G. K. A., and G. P. S. LWI and Safari: A New Index Structure and Query Model for Graph Databases. In *Proceedings of the 11th International Conference on Management of Data*, COMAD'05, pages 138–147, Goa, India, 2005.

[25] K. Wu, E. Otoo, and A. Shoshani. Optimizing Bitmap Indices with Efficient Compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006.

[26] X. Yan and J. Han. *Managing and Mining Graph Data*, volume 40, chapter 5, pages 161–180. Springer-Verlag New York Inc, 2010.

[27] J. P. Yoon, V. Raghavan, V. Chakilam, and L. Kerschberg. Bitcube: A three-dimensional bitmap indexing for XML documents. *Journal of Intelligent Information Systems*, 17(2-3):241–254, 2001.