DEX: a High-Performance Graph Database Management System

Norbert Martínez-Bazan [#1], Sergio Gómez-Villamor [*2], Francesc Escalé-Claveras [#3]

[#]*DAMA-UPC, Computer Architecture Dept., Universitat Politècnica de Catalunya*
Campus Nord UPC, C/Jordi Girona 1-3, 08034 Barcelona, (Catalonia, Spain)
[1]nmartine@ac.upc.edu
[3]fescale@ac.upc.edu

[*]*Sparsity Technologies*
Barcelona, (Catalonia, Spain)
[2]sgomez@sparsity-technologies.com

***Abstract*— The amount of applications calling for efficient large graph management is dramatically increasing. Social network analysis, Internet or biocomputation are just three examples of such applications. In these cases, the interest focuses on the structural analysis of the relationships between different entities organized in huge networks or graph-like structures. Being able to efficiently handle such graphs becomes essential, placing graph database management systems in the eye of the storm. Among the different challenges posed by graph databases, finding an efficient way to represent and manipulate huge graphs that do not entirely fit in memory is still an unresolved problem.**

In this work, we present DEX, a high performance graph database management system based on bitmaps and other secondary structures. We show that by using bitmap structures for graph representation it is possible to improve the performance of a graph database system, allowing for the efficient manipulation of very large graphs containing thousands of millions of nodes and edges.

I. Introduction

The increasing number of huge networks such as the Internet, geographical systems and social network databases, 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 particular, the study of very large scale-free networks characterized by power-law distributions is one of the most active areas of research at this moment. Thus, 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 this scenario, the natural way to represent the information and the results is by means of very large graphs.

Those environments impose three important problems: (i) the continuous growth of the data sources, (ii) the need for a versatile querying system that allows for queries with different flavors such as link analysis, pattern recognition, graph mining or keyword search, among others, and (iii) the need for more flexible schemas in order to integrate structured and semi-structured data coming from different sources.

Finding structures for representing such kind of graphs containing hundreds of millions of objects that allow for its efficient manipulation is essential, especially when the graph size prevents the system from accommodating it entirely in-memory. In this situation, the internal representation of the graph must ensure several crucial aspects: (i) evaluating a query should not imply loading the whole graph into memory, (ii) nodes 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 values in the graph should be accessed very fast. Therefore, these requirements demand for a very versatile, light and compact structure that can be manipulated without jeopardizing performance.

The first proposals of graph-based data models (GDM) were made at the beginning of the 1980's for the formalization and comparison with other database models (LDM [2]). At the beginning of the 90's appeared *GOOD* [3], which set the theoretical basis of a system were both the representation and manipulation of data was based on graphs. A complete survey of these and other graph storage systems can be found in [1].

After several years without significant research activity, there was a reactivation at the beginning of this decade due to the increasing interest in the exploration of large networks, keyword-based search, link analysis and graph mining, among other new research areas. For example, in the WEB area, RDF [4] has become the *de facto* standard for the representation of metadata and Internet resources, where data is represented as a graph of triples subject-predicate-object.

Recently, there has been a renewed interest in the field of general-purpose graph databases that has given rise to several commercial graph database systems such as Neo4J (neo4j.org), a Java-based graph database; HyperGraphDB (www.kobrix.com), a key-value storage supporting generalized hypergraphs; Sones (www.sones.com), an object-oriented data management system based on graphs; or InfiniteGraph (www.infinitegraph.com), a distributed graph database for large-scale graphs.

In this work, we present DEX [5], [6], a proposal to efficiently implement and manipulate very large graphs in a graph database system. We show that, by using bitmap structures, we reduce the cost of the most common graph operations, improving the overall performance of a graph

database system. There are two important aspects that make bitmaps very suitable: (i) they allow keeping a large amount of information in a relatively reduced amount of space, usually by using compression techniques, and (ii) they can be operated very efficiently by using binary logic operations.

II. DEX

DEX is a new graph database defined and implemented using a combination of several specialized structures that allow for an efficient management of very large graphs. It fulfills the conditions of a graph database model since its data representation is in the form of a large graph; the query operations are based on graph operations or extensions to graph operations; query results are also in the form of new graphs; and, finally, there are constraints based on node and edge types, explicit and implicit relationships, and attribute domains.

The logical data structure in DEX is a labeled and attributed multigraph $G = \{L, N, E, A\}$, where L is the collection of labels, N is the collection of nodes, E is the collection of edges, directed or undirected, and A is the collection of attributes. A *labeled* graph has a label for each node and edge, that denotes the object type. A *directed* graph allows for edges with a fixed direction, from the *tail* or source node to the *head* or destination node. An *attributed* graph allows a variable list of attributes for each node and edge, where an *attribute* is a value associated to a string which identifies the attribute. A *multigraph* allows multiple edges between two nodes. This means that two nodes can be connected several times by different edges, even if two edges have the same tail, head and label. DEX deals with two different types of graphs: the $DbGraph$, a single persistent graph that contains the whole database, and the $RGraphs$ that can be used to create temporary query results. The information contained in the DbGraph is used as source data by DEX queries to obtain the results in the form of one or more RGraphs.

Figure 1 shows an example of a graph database with labels or types for nodes and edges (*labeled*), attributes in nodes and edges (*attributed*), and single and multiple directed or undirected edges (*directed multigraph*). In the graph database instance of this example, there are three node types (*PERSON*, *MEDIA* and *TAG*), two directed edge types (*APPEARS* and *HAS*) and one undirected edge type (*CONTACT*). Each node or edge type has a different set of attributes. For example, the node type *PERSON* has two attributes (*name* and *age*) and the edge type *CONTACT* has only one attribute (*since*). The current DEX implementation supports the following data types for attribute domains: character string literals, large texts, integer numbers, long integer numbers, real numbers, boolean values, timestamps, and node or edge identifiers (oids).

Bitmap Representation

The construction of a huge DbGraph and the management of a large volume of concurrent RGraphs with a limited amount of memory requires a new way to represent a graph for graph database querying systems that allows for out-of-core

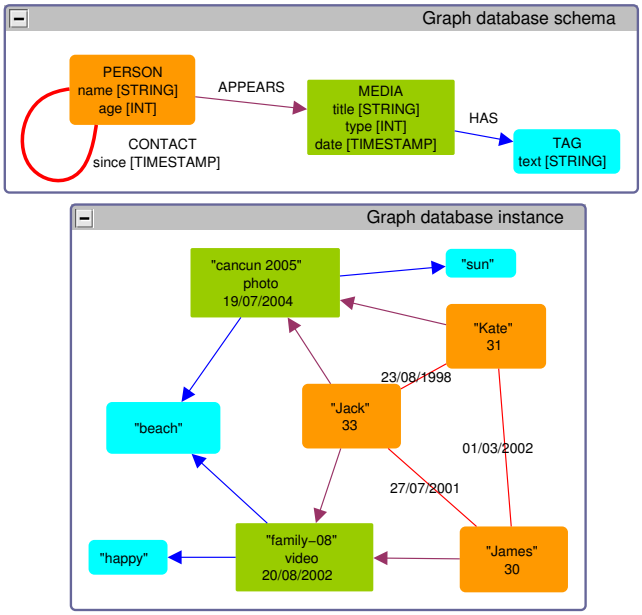

Fig. 1. Example of labeled and directed multigraph database.

data management. Our approach follows four basic principles: (i) the graph must be split into smaller structures to favor the caching of significant parts of the data with a small memory usage, reverting in a more efficient storage and query performance; (ii) object identifiers for nodes and edges have to be used to reduce the memory requirements and to speed-up the graph operations; (iii) specific structures must be used to help in the navigation and traversal of edges; and finally, (iv) attributes have to be fully indexed to allow queries over nodes and edges based on value filters.

Thus, our proposal is based on *oids* and two different types of structures: *bitmaps* and *maps*. As explained above, a DEX graph contains types, attributes, nodes and edges. Each object (a node or an edge) has a unique identifier in the graph, called *oid*. An oid is represented by a positive integer number. A collection of objects is a set of distinct oids. A bitmap or bit-vector is a collection of presence bits that denotes which objects are selected or related to other objects. They are essential for speeding-up the query execution and reducing the amount of space required to store and manipulate the graph. A map is similar to an inverted index with key values associated to bitmaps or data values, and it is used as an auxiliary structure to complement bitmaps, providing full access to all the data stored in the graph.

These two types of structures are combined to build a more complex one: the *link*. A link is a binary association between unique identifiers and data values. It provides two basic functionalities: given an identifier, it returns the value; and given a value, it returns all the identifiers associated to it. Finally, a graph is built as a combination of links, maps and bitmaps to provide a logical view of a labeled and directed attributed multigraph.

Thus, DEX splits the graphs into multiple small indexes to improve the management of out-of-core workloads, with the use of efficient I/O and cache policies. It is out of the scope of this paper to give a detailed description of the implementation and performance of each DEX structure. A complete definition of the DEX internals can be found in [6].

DEX queries are implemented as a combination of low-level graph-oriented operations, which are highly optimized to get the maximum from the data structures. The internal calls to the structures are encapsulated into a library of public operations to simplify the programming of queries. DEX aims at maintaining these list of public operations as small as possible and leaving the implementation of more complex algorithms to a higher level.

III. DEX Architecture

The DEX Engine is a highly efficient software development library which allows for the construction of applications for the exploration of very large graphs. DEX, currently in version 4.0, has been developed by the DAMA-UPC technology transfer team from Universitat Politècnica de Catalunya (Barcelona, Spain) since 2006. It is the result of the intensive research made by the group in the areas of database performance and very large graph analysis, and it has been successfully used in multiple industrial projects in different areas, such as fraud detection, medical treatment analysis, bibliographic search, or advertising brainstorming. DEX is currently beign comercialized through Sparsity Technologies, a spin-out of the DAMA-UPC Group, but the research and innovation of the DEX core engine is still being done by the DAMA-UPC research team.

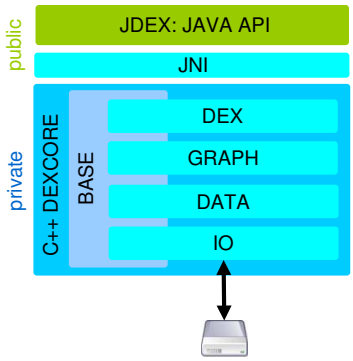

Fig. 2. DEX core architecture.

The current version of DEX is provided as a Java library with a complete API that supports the storage, extraction and query of large graphs. For efficiency purposes, the core engine has been written in portable C++ and successfully tested under different versions of GNU/Linux (32/64 bits) and Windows, and there is a transparent JNI interface which provides the Java access to this core. Figure 2 shows the different layers of the current DEX engine. It has two main components: DEXCORE, the most internal part of the architecture, responsible for the data management in the form of bitmaps, maps and links; and JDEX, the public Java API.

DEXCORE has been built in four different layers: *IO*, *DATA*, *GRAPH* and *DEX*. There is also an auxiliary library, *BASE*, which provides common functions and guarantees the portability of the whole engine. The *IO* layer manages the storage of the data structures and the I/O file operations to allow the out-of-core management of DEX graphs. There are two storages, one persistent and another one temporal. Each storage is divided into 64 KB pages. Pages are managed in multiple memory pools that can be configured by the final applications. The *DATA* layer implements all the structures: bitmaps, different kinds of maps, links, plus other auxiliary structures such as large vectors or long text buckets. The *GRAPH* layer combines different structures to build and manage the DbGraph and the RGraphs. Finally, the *DEX* layer is responsible for the organization of multiple databases, user sessions, transaction management, setup and monitoring. In detail, the current version 4.0 of DEX supports 37-bit unsigned integer $oids$, more than 137 billion objects per graph. These $oids$ are clustered in groups for each node 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. Bitmaps are compressed by grouping the bits in 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. Maps are implemented using B+ trees, and maps of UNICODE strings store the distinct key values in a separate storage of compressed UTF-8 character sequences. Also, the tail and head links are split into specific links for each edge type that contains only the references to edges of the type, achieving a more efficient memory management.

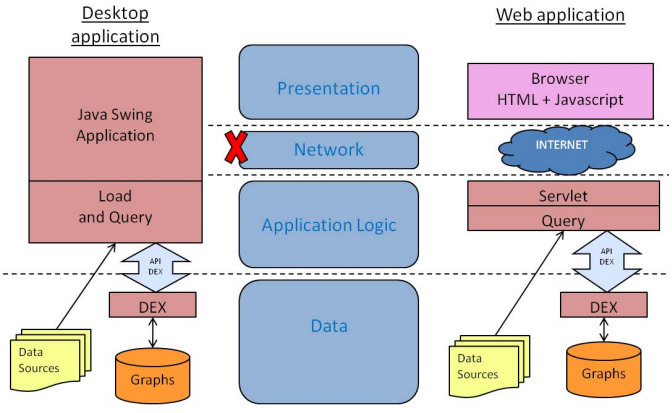

Fig. 3. Two examples of application architectures using DEX.

The DEXCORE C++ library is provided as a public Java API. The interface is implemented following the JNI specification to allow a transparent mapping between the Java calls to the C++ calls. This Java API simplifies the programming by providing a complete set of Java classes and interfaces,

and provides extra functionallity built directly in Java, like scripting for schema definition and bulk load, data loaders for comma-separated files and JDBC, output of subgraphs and query results in standard formats such as GraphML, and high-level graph operations such as connected components, BFS and DFS traversals, weighted and unweighted shortest paths and attribute statistics and histograms.

Figure 3 shows two examples of the integration of DEX in different Java applications. The left example is an embedding of DEX into a Swing desktop application, while the right one is a two layer web-based architecture, where DEX is used in the server side as the backend storage and application logic, and the client is a simple WEB browser.

IV. Experimental Results

It is out of the scope of this short paper to provide details of the performance of DEX with large-scale graphs in different areas of interest. These can be found in [6] with an analysis of the performance of the core engine and the bitmap representation; [5] for information retrieval; and, finally [7] with a survey of graph database performance between multiple graph database storages.

Just as an example, Table I contains the results of the loading process of three large datasets: IMDb (www.imdb.com), the Internet movie database; Wikipedia (en.wikipedia.org), a Web-based and multilingual encyclopedia project; and a synthetic dataset generated using RMAT [8], which naturally generates power-law degree distributions.

TABLE I
Data Load

	IMDb	Wikipedia	RMAT SF=28
DbGraph (GB)	2.40	7.60	83.00
physical memory (GB)	9.00	9.00	60.00
load time (hours)	0.36	2.17	15.04
nodes (millions)	13.34	19.50	230.85
edges (millions)	22.26	180.23	2147.49
values (millions)	48.14	283.73	230.85
insertions/sec.	65014.21	62373.31	48187.76

The results show that DEX is able to create large graphs with hundreds of millions of nodes, edges and attribute values at an average speed of thousands of objects and values per second. Notice that DEX 4.0 can handle a very huge graph with more than 2.3 billion nodes and edges (RMAT scale factor 28), generating a full indexed DbGraph of 83GB, which is 38% larger than the available memory.

Following, we present in Table II the result of the execution of three different queries over the IMDb graph database with different degrees of complexity: the first query, *Link Analysis*, is a full extraction of a movie using multiple 1-hop traversals; the second, *Social Networks*, calculates the distance between two actors; and, finally, *Pattern Recognition* extracts all movies that match a complex pattern involving aggregate calculations. In this experiment, we repeat all the executions using two different memory configurations. First, we execute each query having all the system in available memory. This allows to allocate enough memory for the whole DbGraph, making it possible to execute the whole operation in-memory. Second, we repeat all the executions limiting the memory capacity to 128 MB. We have chosen this memory size not to allow bit vectors, which occupy around 170 MB, to fit in memory.

TABLE II
Query Results

Query	M.nodes	M.edges	t(s) unlimited	t(s) 128MB
Link Analysis	0.002	0.004	0.13	0.13
Social Networks	0.008	0.008	1.52	1.79
Pattern Recognition	0.111	0.315	383.88	385.04

Looking at the results, we can see that DEX can solve complex queries that return thousands of nodes and edges in milliseconds or a few seconds. DEX adapts well to very low memory configurations by taking advantage of the intensive use of bitmaps and bitmap operations.

V. Conclusions

In this work, we have proposed DEX, a high performance graph database querying system. DEX is the first structural representation of a graph that allows for efficiently answering generic queries on graphs that may contain several hundred millions of attributed objects. We have seen that bitmaps are essential for achieving a full-indexed view of a graph database, still affordable in terms of space, which is crucial in order to perform the most typical graph-oriented operations.

Our work makes also a full-fledged graph database querying system proposal available for studying important challenges such as performance of graph-database operations, massive load of very large graphs in the range of trillions of objects, optimization of graph queries, preservation of integrity constraint, graph pattern matching or other generic high-level operations on graphs.

References

[1] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Computing Surveys (CSUR)*, vol. 40, no. 1, pp. 1–39, 2008.

[2] G. M. Kuper and M. Y. Vardi, "A new approach to database logic," in *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 2-4, 1984, Waterloo, Ontario, Canada*. ACM, 1984, pp. 86–96.

[3] M. Gemis, J. Paredaens, I. Thyssens, and J. Van den Bussche, "Good: A graph-oriented object database system," in *Proc. of the 1993 ACM SIGMOD, Washington, D.C., May 26-28, 1993*, P. Buneman and S. Jajodia, Eds. ACM Press, 1993, pp. 505–510.

[4] 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*.

[5] N. Martínez-Bazán, V. Muntés-Mulero, S. Gómez-Villamor, J. Nin, M. A. Sánchez-Martínez, and J. Larriba-Pey, "DEX: High Performance Exploration on Large Graphs for Information Retrieval," in *Proceedings of the 16th ACM Conference on Information and Knowledge Management Conference (CIKM), Lisbon*, 2007, pp. 573–582.

[6] N. Martínez, "DEX: High-Performance Graph Databases," Master's thesis, Univertitat Politècnica de Catalunya, Barcelona (Catalonia, Spain), 2008.

[7] D. Dominguez-Sal, P. Urbón-Bayes, A. Giménez-Vañó, S. Gómez-Villamor, N. Martínez-Bazán, and J. L. Larriba-Pey, "Survey of graph database performance on the hpc scalable graph analysis benchmark," in *IWGD*, 2010.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *SIAM Data Mining*, vol. 6, no. 1, 2004, pp. 6–1.