DEX: High-Performance Exploration on Large Graphs for Information Retrieval

Norbert Martínez-Bazan[1] Victor Muntés-Mulero[1] Sergio Gómez-Villamor[1]
Jordi Nin[2] Mario-A. Sánchez-Martínez[1] Josep-L. Larriba-Pey[1]

[1]DAMA-UPC, Computer Architecture Dept.
Universitat Politècnica de Catalunya,
Campus Nord UPC, C/Jordi Girona 1-3
08034 Barcelona, (Catalonia, Spain)
{nmartine,vmuntes,sgomez,msanchem,larri}
@ac.upc.edu

[2]IIIA, Artificial Intelligence Research Institute
CSIC, Spanish National Research Council
Campus UAB s/n
08193 Bellaterra, (Catalonia, Spain)
jnin@iiia.csic.es

ABSTRACT

Link and graph analysis tools are important devices to boost the richness of information retrieval systems. Internet and the existing social networking portals are just a couple of situations where the use of these tools would be beneficial and enriching for the users and the analysts. However, the need for integrating different data sources and, even more important, the need for high performance generic tools, is at odds with the continuously growing size and number of data repositories.

In this paper we propose and evaluate DEX, a high performance graph database querying system that allows for the integration of multiple data sources. DEX makes graph querying possible in different flavors, including link analysis, social network analysis, pattern recognition and keyword search. The richness of DEX shows up in the experiments that we carried out on the Internet Movie Database (IMDb). Through a variety of these complex analytical queries, DEX shows to be a generic and efficient tool on large graph databases.

Categories and Subject Descriptors

H.2.5 [**Database Management**]: Heterogeneous Databases; H.3.2 [**Information Storage and Retrieval**]: Information Storage; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Design, Performance

Keywords

Data Representation, Graph Databases, Information Retrieval, Social Networks, Query Performance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'07, November 6–8, 2007, Lisboa, Portugal.
Copyright 2007 ACM 978-1-59593-803-9/07/0011 ...$5.00.

1. INTRODUCTION

The development of huge networks such as the Internet, geographical systems, transportation or automatically generated social network databases, has brought the need to manage information with inherent graph-like nature [4]. In these scenarios, users are not only keen on retrieving plain tabular data from entities, but also relationships with other entities using explicit or implicit values and links to obtain more elaborated information. In addition, users are typically not interested in obtaining a list of results, but a set of entities that are interconnected satisfying a given constraint. Under these circumstances, the natural way to represent results is by means of graphs. As a consequence, classical database management systems (DBMS), typically based on the relational model, may not be the most suitable option to answer queries with these objectives.

Cases like bibliographic databases are a clear example where a more complex querying system would be beneficial. In these scenarios, the user might not be only interested in finding an specific author or publication, but to analyze the relationships within a group of authors, to understand the relevance of an specific paper or any other query implying the exploration of the relationships between entities.

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 Information Retrieval (IR) queries with different flavors ranging from keyword search to the complex mining of patterns in graphs, and (iii) the need to integrate data coming from different sources to enrich the answers to complex queries over incomplete databases.

Several Graph Database Models (GDMs) and IR systems have been proposed in the literature to tackle these problems partially. GDMs can be characterized as those where data structures for the schema and instances are modeled as graphs or generalizations of them, and data manipulation is expressed by graph-oriented operations and type constructors. Examples of GDMs are Good [6], Gram [2], GraphDB [9] and Gras [11] which is also an implementation for small to medium sized databases.

In this paper we propose and evaluate DEX, a high performance graph database querying system oriented to handle large graph databases coming from, among others, data in IR environments. DEX processes a database graph formed by a unique schema subgraph integrating all the involved sources and a unique data subgraph containing the entities and links between them. DEX includes a small set of graph operations that allow for querying the database in a versatile way. This makes DEX generic and allows for multiple query types including link analysis, social network analysis, pattern recognition and keyword search. In general, we show that DEX can be used both in scenarios where the user has detailed information of the data schema and, also, in those scenarios where the information about the data organization is not available. In addition, we also present the most important structures used in DEX, which are the key for DEX to perform efficiently with large data sets. The evaluation of DEX is done using the Internet Movie Database (IMDb) and it is performed with 5 different queries in multiple IR areas, showing the performance of the system in two out-of-core and in-memory configurations. Also, we perform an analysis of the use of different strategies to solve the same type of query, showing the need for research in the area of graph query optimizers. Finally, we evaluate the load times for databases ranging from a few MB to more than 12 GB.

This paper is organized as follows. In Section 2, we present the general framework of DEX, introducing the data model, the query processing and the core operations. Section 3 presents the architecture of DEX. Results are presented in Section 4. Finally, in Sections 5 and 6, we present related work and draw some conclusions and future work.

2. FRAMEWORK

DEX is based on a graph database model, that is basically characterized by three properties: data structures are graphs or any other structure similar to a graph; data manipulation and queries are based on graph-oriented operations; and there are data constraints to guarantee the integrity of the data and its relationships. DEX fulfills these conditions 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 relationships and attribute domains.

2.1 Graph Model

The most basic logical data structure in DEX is a labeled and directed attributed multigraph $G = \{T, N, E\}$, where T is the collection of labels, N is the collection of nodes and E is the collection of directed edges. A *labeled* graph has a label for each node and edge, that denotes the object type. A *directed* graph has all 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 name, simplifying the graph structure. Finally, a *multigraph* allows multiple edges between two nodes. This means that two nodes can be connected several times by different edges, with the only restriction that it is not allowed to have two edges with the same tail, head and label. A directed multigraph helps to represent the richness of the multiple relationships between objects as this is a usual situation into complex social networks.

A label or object type $t \in T, t = \{typename, A\}$, has a unique name *typename* and a collection of attribute definitions, where each attribute $a \in A$, $a = \{attrname, domain\}$ has a unique name *attrname* and is restricted to a *domain* (for instance strings, numbers or timestamps). Thus, the objects of a certain type can only contain values within the same set of strong-typed attributes.

A node $n \in N, n = \{key, type, depth, text, V\}$ has an optional unique *key* that can serve as a unique identifier for the original data source, like a ROWID in a relational database or an URL into the WWW. It belongs also to a *type* $\in T$, and has a *depth* that represents its level in the hierarchy of nodes. It can also have an optional *text* to store a user-defined string: a description, an HTML page, the text of an XML element, etc. Each node has a collection $V = \{(a, v)\}$ of values for the attributes of its type, where $a \in A(type) \wedge v \in domain(a)$. An edge $e \in E, e = \{tail, head, type, depth, V\}$ has a *tail* or source node, a *head* or destination node, and like the nodes, it belongs to a *type* $\in T$, has the *depth* into the graph hierarchy and a collection $V = \{(a, v)\}$ of values for the attributes of its type.

2.2 Data Representation

DEX deals with two different types of graphs: the *DbGraph*, a single graph that contains the schema and data extracted from external data sources and stores them in a persistent storage; and the *RGraph*, the result of a DEX query, which is stored in the temporal storage. 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.

The DbGraph includes two subgraphs: the *schema* subgraph, which contains the metadata of the data sources, and the *data* subgraph, which contains the data loaded from the data sources as defined in the schema subgraph. Both are connected through edges between the entity definitions of the schema subgraph and their instance objects in the data subgraph. The schema subgraph contains the following node types:

datasource: contains information of a data source and its type (ODBC, CSV, XML, WEB, etc.). It is connected to one or more *datasets*.
dataset: the definition of an entity or collection of data units (rows, XML elements, HTML pages, etc.). It is defined by one or more *attributes*.
attribute: a characteristic or property of a dataset (column, xml attribute, etc.). It has a name and a data type and can be part of one or more *relationships*.
relationship: it can represent either any edge between two instances in any dataset, or the definition of a constraint between two attributes (for example a foreign key).

Figure 1 shows an example of the mapping between a simple bibliographic database stored in a RDBMS, and the corresponding schema subgraph through its DEX representation. Attributes belong to datasets and datasets to datasources, creating a hierarchy that depends on the original data sources. However, relationships may link datasets from a single or across multiple data sources (intra- or inter-relationships, respectively). This goes beyond the classical data models, allowing the DbGraph to connect and relate heterogeneous data sources that share common identifiers, as if they were foreign keys across different data sources. Thus, if there are two distinct databases that have the entity PERSON, and both entities have a public person identifier like the social security number (SSN) as a primary

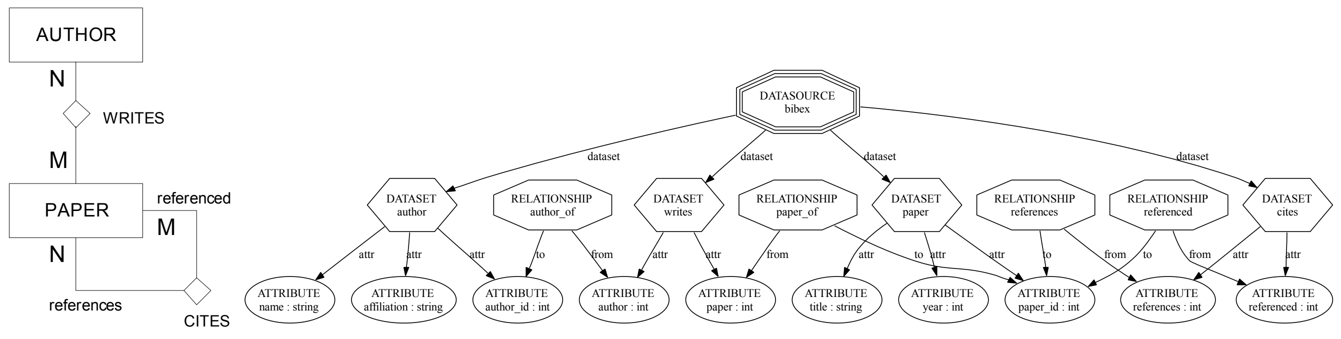


Figure 1: Example of E/R and DbGraph for a Bibliographic Database

key, a relationship between them can be created within the DbGraph to join all the information of each single person. This provides the illusion of a single and integrated storage. In the future these relationships can also serve as integrity constraints between multiple data sources.

As explained before, the DbGraph allows for the integration of multiple data sources mapped from different data models. This data sources can be relational databases, tabular files with rows and columns, XML documents, HTML pages, RDF graphs, etc. For each different data model there exists an specific mapping to the DbGraph schema and data representations. Table 1 shows a brief summary of a sample of these mappings.

Table 1: Mapping of data models to DbGraph

type	RDBMS	CSV	XML	HTML	RDF
datasource	schema	path	path	URL	URL
dataset	table	file	DTD or xsd:schema	URL prefix	graph
attribute	column	column	element attribute	-	-
relationship	foreign key	-	-	-	-
data node	row	row	elements	web page	subject object
data edge	-	-	hierarchy parent-child	hyperlinks	predicate

2.3 Query Processing

Conventional data models, like the relational, are well suited for queries based on values, like equalities or range search, but in these models the exploration of relationships must be always set explicitly by the joins even if foreign keys have been declared, and it becomes really difficult to explode all the potential relationships of a node. Not only this: for self-relationships, the relational queries require recursive extensions that are more difficult to create and manage. On the opposite side, the natural query mechanism of DEX is the automatic exploration of the relationships in a graph, represented in the form of edges between nodes. It is a *relationship* rather than a *value* oriented analysis.

However, in a real scenario with complex analysis where the source data does not represent the semantics of the query, or where some hidden patterns must be detected in the structures, a more sophisticated query process is required. Our proposal to solve DEX queries is the *Exploral* process, as shown in Figure 2. The different phases of this process are defined as follows:

Querying. In this phase, the DbGraph is queried during the *SELECT* task to return only the N nodes that match a filter or a set of conditions. Each node selected is then processed by the *EXPLODE* task to create a new RGraph from that node. Then the RGraph is exploded by navigating the DbGraph relationships taking into account optional constraints in the edge type, the edge direction or the explosion depth. The result is a collection of N RGraphs.

Preparing. In this phase, zero or more *TRANSFORM* tasks can modify the structure of the RGraphs, by adding or removing edges, grouping nodes in clusters or hubs, splitting the RGraph in multiple subgraphs, inferring new information or relationships, etc. The result is a collection of $M \geq N$ RGraphs. Then, in the *FILTER* task RGraphs are checked to validate if they are valid, for example, by detecting if they contain a certain pattern in their structure. The result is a subset of $P \leq M$ RGraphs.

Mining. While in the *FILTER* task each RGraph was analyzed individually, sometimes we need to compare the candidate results to check if they match some pattern between them. For instance, we would be interested in returning the best RGraph or only a subset of RGraphs. This is done during the *MATCH* phase by applying graph-mining and pattern-matching techniques, and the result is a subset of $Q \leq P$ RGraphs.

Browsing. In this phase, each RGraph may be modified during the *DISPLAY* task, and new visual attributes may be set for nodes and edges (e.g. line styles, colors, etc.).

Note that most of the tasks in the process allow for inter-task parallelism. Specifically, the first four tasks from SELECT to FILTER and the DISPLAY task can be pipelined and/or parallelized because each graph has no dependencies on the others.

2.4 Core graph-oriented operations

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. DEX aims at maintaining the list of operations as small as possible and to leave the implementation of more complex algorithms to a higher level. The current version has only 18 operations. Table 2 gives a summary with the name and a brief description of each operation. It is out of the scope of this paper to give a detailed definition of these core operations, its signature and implementation.

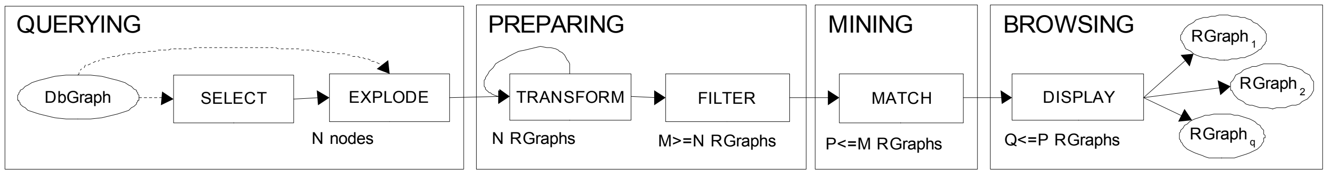


Figure 2: Exploral Process

Table 2: Core graph-oriented Operations

Operation	Description
CreateGraph	Creates a new empty graph
DropGraph	Removes a graph
RegisterType	Registers a new node or edge type
GetTypes	Returns the list of all node or edge types
FindType	Returns the id of a node or edge type
NewNode	Creates a new empty node of a given node type
AddNode	Copies a node
AddEdge	Adds a new edge of some edge type from a head node to a tail node
DropObject	Removes a node or an edge
Scan	Returns all nodes or edges of a given type
Select	Returns the nodes or edges of a given type which have an attibute that evaluates true to a comparison operator over a value
Related	Returns all neighbors of a node following edges members of an edge type in a specific direction
Neighbors	Returns all nodes of a node type that are neighbors of a node regardless of the edge type or direction
Combine	Returns the union, intersection or difference of two collections of nodes or edges
SetAttribute	Changes the value for an attribute of an object
GetAttribute	Returns the value of an attribute of an object
SetProperty	Changes an object property
GetProperty	Returns an object property

2.5 Query Example

In the previous sections, we detailed the different phases and tasks of the Exploral process and the core operations provided by the DEX engine. A simple query example can help to understand how the Exploral process works and which is the mapping between the Exploral tasks and the core operations. Let us take the bibliographic database example in Figure 1. There are four data sets: AUTHOR, PAPER, WRITES and CITES. A simple query could be: **Find the ten authors with the largest number of publications that have published at least two papers in year 2006**. The Exploral process for this query will contain the following tasks:
Select. It scans the data set AUTHOR to return all the authors that fulfill the query requirements.
Explode. For each author, it builds a graph and explode the relationships to WRITES and PAPER.
Transform. For each graph, it groups the papers by its publication year, by creating new grouping nodes.
Filter. It drops all the graphs where group '2006' does not exist.
Match. It sorts the graphs in descending order by the number of nodes of type 'PAPER', keeping only the first 10.
Display. It removes the grouping nodes.

Each tasks is implemented with a combination of core graph-oriented operations. An example for two of them follows.

Algorithm 1 SELECT
```
return Scan($DbGraph$,
            FindType($DbGraph$,
                     "AUTHOR", <node>))
```

Algorithm 2 FILTER
```
for all $gr$ : $RGraph$ do
  $nodes \leftarrow$ Select($gr$,
                FindType($gr$, "YEAR", <node>),
                <eq>, "2006")
  if length($nodes$) = 0 then
    DropGraph($gr$)
  else
    $papers \leftarrow$ Neighbors($gr$, $nodes$[1],
                    FindType($gr$, "PAPER",
                             <node>))
    if length($papers$) < 2 then
      DropGraph($gr$)
    end if
  end if
end for
```

3. DEX ARCHITECTURE

The first prototype of DEX has been built using an architecture of three layers as shown in Figure 3: the core, the lower layer, that manages and queries the graph structures; an inner API layer to provide an application programming interface; and the higher layer applications, to extend the core capabilities and to visualize and browse the results.

Core. This is the most internal part of the architecture and it is responsible for the data management and query resolution. It contains three important modules: the *GraphPool*, the *DbGraphManager* and the *QueryEngine*. The *GraphPool* manages the graphs and their data structures, the buffer pool and the I/O file operations. It stores the DbGraph data in a persistent repository and maintains a temporary repository for the RGraph cache. The *DbGraphManager* is the interface between the external data modules and the DbGraph. It has different plug-ins for each different data source format. Finally, the *QueryEngine* provides optimized methods to query and browse graphs. For efficiency purposes, this layer has been written in portable C++ and successfully tested under different versions of GNU/Linux and Windows.

API. This layer provides a Java interface to the core capabilities, basically with public calls to the core operations plus some extra functions to start, shutdown and maintain a DEX instance.

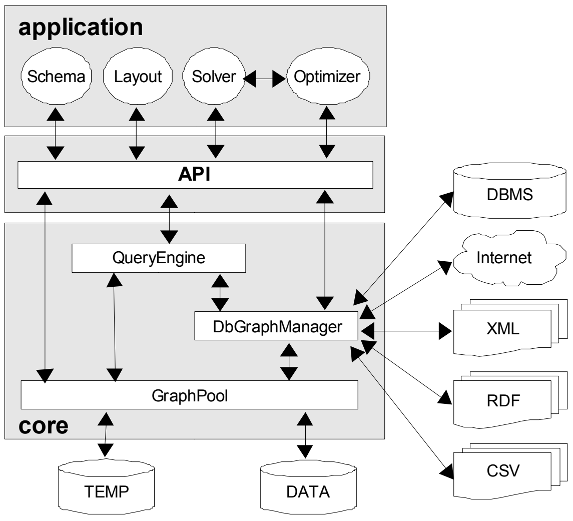
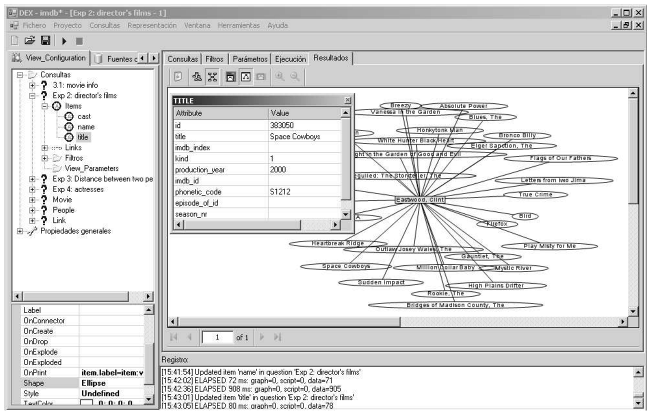


Figure 3: DEX architecture

Applications. Multiple Java tools or applications can be built on top of the DEX API. The current DEX front-end for Windows provides a user-friendly interface and contains several extension modules to increase the capabilities of the core: a *schema* module for the maintenance of the DbGraph, a *solver* module to implement queries following the Exploral process, an *optimizer* module to improve the access plans generated by the solver, a *layout* module to display RGraphs using different graph layouts like hierarchical, radial, etc. The displayed graphs can also be browsed to retrieve extra information or to refine the queries to obtain more accurate results. Figure 4 shows a screenshot of an RGraph in a radial layout being browsed to extract details of one of its nodes.



Attribute	Value
id	383050
title	Space Cowboys
imdb_index	
kind	1
production_year	2000
imdb_id	
phonetic_code	S1212
episode_of_id	
season_nr	

Label	
OnConnector	
OnCreate	
OnDrop	
OnExplode	
OnExploded	
OnPrint	item.label=item:v
Shape	Ellipse
Style	Undefined
TextColor	0; 0; 0; 0

```
[15:41:54] Updated item 'name' in question 'Exp 2: director's films'
[15:42:02] ELAPSED 72 ms: graph=0, script=0, data=71
[15:42:36] ELAPSED 908 ms: graph=0, script=0, data=905
[15:43:01] Updated item 'title' in question 'Exp 2: director's films'
[15:43:05] ELAPSED 80 ms: graph=0, script=0, data=78
```

Figure 4: DEX Interface

3.1 Core Internals

The construction of a large DbGraph for multiple datasets and the management of a huge volume of concurrent RGraphs with a limited amount of memory requires an implementation of specialized structures. A basic way to represent a directed multigraph could be a sparse oriented incidence matrix. However, this representation may not satisfy our requirements of high-performance storage and retrieval. Therefore, DEX splits the graph matrix 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, but an introductory definition of the DEX internals follows.

Let $oid \in \mathbb{O} \subset \mathbb{N}$ be a unique object identifier. A local *oid* is unique only in a local context (e.g. an array index), while a global *oid* is unique regardless the context where it is being used (e.g. a node id). Unique integer *oid*s help in the construction and compression of structures. Let $\mathbb{T} = \{\texttt{int}, \texttt{long}, \texttt{real}, \texttt{string}, \texttt{boolean}, \texttt{timestamp}, \mathbb{O}\}$ be the supported data types.

A set $\mathbf{S}\langle\mathbb{T}\rangle = \{s_1, s_2, \ldots, s_n\}$ is an unordered collection of values of type $\mathbb{T}$ where each value occurs at most once. A bit vector $\mathbf{B} = \mathbf{S}\langle\mathbb{O}\rangle$ is a set of *oid*s implemented using compressed bitmaps. Bit vectors are used to store and retrieve collections of objects because they require less memory and are easier to manage, combine and iterate.

A map $\mathbf{M}\langle\mathbb{T}_k, \mathbb{T}_v\rangle = \{(key, value)\}$ is a collection of keys and a collection of values, where each $key \in \mathbb{T}_k$ is associated with one $value \in \mathbb{T}_v$. Keys can only appear once while values can be associated to multiple keys. Maps are implemented using trees. A reversible map $\mathbf{R}\langle\mathbb{T}\rangle = \mathbf{M}\langle\mathbb{T}, \mathbb{O}\rangle$ is a bijective map where each value is an *oid* associated with one single key. It is implemented using a tree where each node can be accessed also by its unique index.

Let $vid = oid$ be a unique value identifier. Let $\mathbb{V} \subset \mathbb{O}$ be the subset of *vid*s. A dictionary $\mathbf{D}\langle\mathbb{T}\rangle = \{v, o\}$ is a collection that maps key values with their lists of associated *oid*s, where $v = \mathbf{R}\langle\mathbb{T}\rangle$ contains the unique *vid* associated to each value, and $o = \mathbf{M}\langle\mathbb{V}, \mathbf{B}\rangle$ contains the list of *oid*s linked to each different value. The basic idea of the dictionary is to map any type of values to a unique value identifier type that simplifies the structures, improves the performance of operations and in many cases reduces the storage requirements (e.g. variable length string). A link $\mathbf{L}\langle\mathbb{T}\rangle = \{v, d\}$ is a binary association between an *oid* and its value, where $v = \mathbf{R}\langle\mathbb{O}\rangle$ contains the *vid* of each *oid* and $d = \mathbf{D}\langle\mathbb{T}\rangle$ is a dictionary with the *oid*s of each *vid*.

Let $nid \in \mathbb{O}$ be a unique local node id in the graph. Let $eid \in \mathbb{O}$ be a unique local edge id in the graph. Let $tid \in \mathbb{O}$ be a unique local node or edge type id in the graph. Let $\mathbf{P} = \{\texttt{level}, \texttt{tail}, \texttt{head}, \texttt{type}, \ldots\}$ be the different properties for graphs or graph objects. A property can be *materialized* if its values are stored in a structure (for example the `level` in edges but not in nodes), or *virtual* if its values can be extracted or inferred from other graph structures (for example the `level` of a node or the `head` of an edge). Let $\mathbf{T} = \{a, p\}$ be a type, where $a = \mathbf{M}\langle\texttt{string}, \mathbf{L}\langle\mathbb{T}_a\rangle\rangle$ is the variable list of attributes that contains the values of each object for each attribute, and $p = \mathbf{M}\langle\mathbf{P}, \mathbf{L}\langle\mathbb{T}_p\rangle\rangle$ is the predefined list of materialized properties that contains the values of each object for each materialized property. This combi-

Table 3: Load experiments

Dataset	format	raw	DEX	ratio	nodes	t(h:m:s)
DBLP	xml	0.35	0.23	1.52	3.6	00:03:17
CITESEER	xml	1.90	0.42	4.52	4.6	00:04:00
IMDB	csv	1.19	1.14	1.04	25.1	00:20:46
OCP (pack)	csv	3.04	3.18	0.96	99.6	01:06:58
OCP (full)	csv	12.40	9.87	1.27	128.6	03:40:02

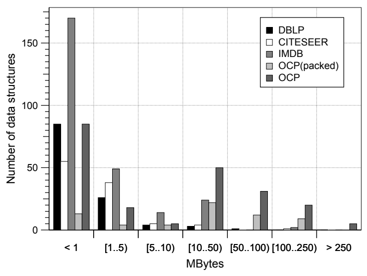


Figure 5: Structures in MB

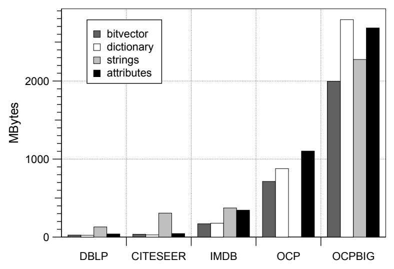


Figure 6: Memory

nation of structures for each attribute is like a full-indexing of the data that provides a fast access to all values in the graph.

A DEX graph $\mathbf{G} = \{n, e, o, t, h\}$ is a directed and labeled attributed multigraph, where $n = \mathbf{M}\langle \texttt{string}, \mathbf{T}\rangle$ contains all attributes and materialized properties of each node type; $e = \mathbf{M}\langle \texttt{string}, \mathbf{T}\rangle$ contains all attributes and materialized properties of each edge type; $o = \mathbf{L}\langle tid, nid\rangle$ contains the list of objects, nodes or edges, for each type; $t = \mathbf{L}\langle nid, eid\rangle$ contains the list of edges where each node is the tail; and, finally, $h = \mathbf{L}\langle nid, eid\rangle$ contains the list of edges where each node is the head. This implementation model based on multiple indexes favors the caching of significant parts of the data with a small memory usage, reverting in a better efficient storage and query performance. As an example, using this representation a DEX graph with 3 node types with 12 attributes and 2 edge types without attributes is stored into 71 structures plus the bit vectors.

4. EXPERIMENTS

In order to test the capability of DEX to manage large amounts of data as well as the ability to analyze and query graph-structured data using different approaches, it becomes mandatory to test our tool on a wide range of different scenarios. In this section, we explain in detail a sample of different experiments executed using DEX. These experiments can be classified into three categories: *(i)* bulk load and data compression; *(ii)* exploratory analysis, with different queries for link analysis, social network analysis, pattern recognition and keyword search; and *(iii)* a performance analysis based on different query approaches.

The experiments are performed using a computer with a single AMD Sempron 2400+ at 1.67 GHz CPU. The memory hierarchy is organized as follows: it contains a 128 KB L1-cache, a 256 KB L2-cache and 3 GB of RAM using a maximum of 2 GB per process. We use a 120 GB disk configured in RAID-1. The operating system is MS Windows Server 2003 R2, Enterprise Edition, including SP-1.

4.1 Load Analysis

The purpose of our first set of experiments is to evaluate the bulk load and compression capabilities of DEX. In order to check the required time to load, full index and compress data, five different data sources were selected and loaded separately into a DEX database. The data sets are obtained from the following sources:

DBLP (dblp.uni-trier.de): the widely used computer science bibliography database, with more than 852,000 scientific references and their authors. The original database is in a single XML document.

CITESEER (citeseer.ist.psu.edu): another well-known scientific literature digital library (citeseer.ist.psu.edu), with more than 716,000 articles, authors and cross references between articles. Data is available in 72 XML files.

IMDb (www.imdb.com): the Internet Movie Database, the largest movie database with information for more than 845000 titles and 2 million people. The download format selected is a plain text data file. These data files are then processed and converted to well-formatted CSV files using IMDbPY (imdbpy.sourceforge.net) python package. Some attributes in the database contain large text fields of several KB each.

OCP(full): this is the first industrial use of the DEX core engine and query capabilities developed for ANCERT, the Spanish Notarial Certification Agency, and it is being used by the Spanish Patrimonial Control Office to detect fraud in real estate and share transmission operations.

OCP(packed): a subset of the full OCP database. This one does not have any confidential information about properties, people, notaries or transactions. All strings have been removed and confidential id's like passports or SSN have been mapped to random unique integer numbers. This version is used for the development of the OCP software because the full database is protected by an NDA agreement.

Table 3 contains the results of the loading process for the five previous data sources, where *raw* is the original data size in GB, *DEX* is the size of the resulting DbGraph in GB, *ratio* is the compression ratio between the raw data and the DbGraph data, *nodes* is the total amount of nodes of the DbGraph in millions, and finally *time* is the total elapsed time for loading the data, indexing and compressing.

Results show that DEX is able to create large DbGraphs in an average speed of about 3 GB/h, regardless of the data set. Data is in general compressed compared to its original size, with a ratio of up to 4.52 in Citeseer. The compress ratio in Citeseer is very large compared to the others because data is formatted in XML with a lot of redundant data containing long string identifiers, which are highly compressed by DEX using its string dictionaries. On the other side, the OCP packed data source is not compressed and it even presents a slight increase in space, with a compression ratio of 0.96. This is due to the fact that this data set does not contain strings and all its values have already been simplified to small integers, drastically reducing the compressing possibilities. Finally, it is also important to remember that the process was performed on a single CPU computer with 2 GB of RAM for the load process and, even in these conditions, DEX is able to manage and compress a 12.4 GB data source into a 9.87 GB compressed DbGraph in less than 3 hours and 45 minutes.

Following, we present a more detailed analysis of the memory requirements and distribution of the internal structures used to store the DbGraph contents. We can observe that, for all the data sources, the DbGraph has been split into a large set of small structures. Figure 5 shows the distribution of the structures explained in Section 3.1 as a function of their size for all the data sets studied. Observe that, in general, the memory group containing a larger number of structures corresponds to those fitting in less than 1MB. If we now look at the memory distribution in terms of usage, in Figure 6, we can see that, in general, bit vectors represent less than 20% of the DbGraph memory requirements, leaving the rest of the memory for dictionaries and attribute data. This means that, given a query, only a small part of these bit vectors will be loaded into memory and then queried, thus reducing the I/O and the memory cache requirements, and improving the performance if we run the experiments in a concurrent query environment.

4.2 Queries

The objective of this subsection is to verify the capability of DEX to solve different kinds of queries, not only from the performance point of view, but also trying to answer real complex questions in a public database. Through the different examples, we also show the generic applicability of DEX to very different scenarios.

The data set selected for our experiments is the IMDb movie database. The reason for choosing IMDb is threefold. First, it provides well-known metadata, data and concepts that allow to present clear and easily understandable examples. Second, it allows for multiple types of queries: asking for movie information, cast roles, exploring the relationship between cast members, searching on the history of cinema, etc. And third, it is the largest data set to which we can access among those not affected by an NDA agreement. Figure 7 shows the E/R diagram for the IMDb database with the number of rows for each entity. The most important data sets are NAME, which contains all people involved in any title; TITLE, which contains basic information of each movie or TV chapter; and, finally, CAST, which contains the relationships between a person (NAME) and a movie (TITLE).

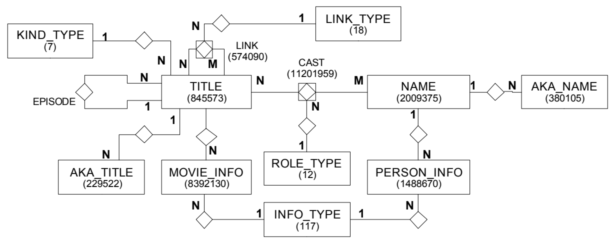


Figure 7: IMDb E/R Diagram

We execute five queries on IMDb in the four different areas of interest mentioned before: *link analysis*, showing that DEX is able to explore the relationships between objects; *social networks*, in order to analyze the relationships between groups of affine objects; *pattern recognition*, showing the ability of DEX to identify patterns into the relationship links between entities; and, finally, *keyword search*, showing how our tool is able to find the relationship between objects that share, in some way, a given keyword, and returns the context of such objects.

Note also, that we do not use a graph query language but write the queries by directly calling the core functions provided by the kernel. Translating a more sophisticated high-level query language into core operations is the task of a graph query optimizer and is out of the scope of this paper.

Following, we present a detailed description for each area of interest. We repeat all the executions using two different memory configurations. First, we execute each query having all the system memory available (2 GB). This allows to allocate enough memory for the whole DbGraph, making it possible for IMDb 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 as shown in Figure 6, to fit in memory. For each memory configuration, each query has been executed five times, and the slowest and fastest results have been discarded. The reported resulting time is the average of the remaining three results. Table 4 contains the details of the execution of each query, where *potential* is the number of nodes selected from the DbGraph during the SELECT phase and, as a consequence, the potential number of results or graphs; *graphs* is the actual number of results; *nodes* is the total number of nodes in all the *graphs*; *edges* is the total number of edges in all the *graphs*; *operations* is the total number of select operations (*scan*, *filter*, *related* and *neighbors*) performed by the DEX query engine; *t(s)* is the total elapsed time in seconds; and, finally, *t(s)/graph* is the average time per resulting graph, in seconds.

4.2.1 Link Analysis

As we have seen in previous sections, DEX is prepared to deal with linked objects in a graph. Intuitively, our tool fits perfectly for solving typical link analysis problems where the focus is on the relationship between the entities of a virtual network. Translated to the DEX domain, we are interested on exploring the relationships between the nodes of the DbGraph by navigating the edges between them.

***Query 1 (Q1)*: Get all the information of movie m.** An example of a link analysis query is Q1, where all the information of a movie is obtained in a single graph. Following the DEX execution explained in Section 3, the root node will be a node from the data set MOVIE. From this initial node, DEX will explode a graph containing all casts, people, information and extra data related to that movie, including the chain of film references to the root. Note that, although the general schema is assumed to be known, we are not sure neither about the amount of information retrieved by the DbGraph nor about the length of the chain of film references. This could pose serious problems to traditional relational systems that would have to resort to recursive queries including a large number of join operations, increasing significantly the complexity of such queries.

We make two different executions of this query Q1a and Q1b. In Q1a, we obtain all the information related to movie 'Crash', awarded with an Oscar to the best movie in 2005. The resulting graph has 2054 nodes and 3692 edges, and it takes less than 0.134 seconds to be obtained. Note that

Table 4: Query Results

							128MB		in-memory	
group	query	potential	graphs	nodes	edges	operations	t(s)	t(s)/graph	t(s)	t(s)/graph
Link	Q1a	1	1	2054	3692	2403	0.133	0.133	0.133	0.133
Analysis	Q1b	845573	845573	122471954	189748956	126980177	8513.871	0.010	8735.637	0.010
Social	Q2	1	1	8199	8198	10865	1.790	1.790	1.518	1.518
Networks	Q3	1	1	1052	552826	1331833	207.228	207.228	207.183	207.183
Pattern Rec.	Q4	2009375	4705	110624	315355	6048463	385.035	0.082	383.876	0.082
Keyword	Q5a	3464575	384	4754	4370	18	2.624	0.007	1.914	0.005
Search	Q5b	24547488	3308	14614	11306	41	21.217	0.006	16.492	0.005

time results are identical when we limit the available memory to 128 MB, most possibly because the bit vectors used for this query fit in memory in both cases. The second version, Q1b, is an explosion of all the information for all the movies, implying near 40% of all the titles in the DbGraph (the remaining 60% are TV chapters). The final result is a set of 845573 graphs, one for each movie, with hundreds of millions of nodes and edges in total. The execution requires near 127 millions of core operations, and it takes around 10 milliseconds per graph on average. Note that, in this particular case, the average time for the executions where the memory is limited to 128 MB is lower than those where the whole memory is available. This results make sense if we take into account that, in the first scenario, the GraphPool size is small, and therefore its management is fast, while in the second case it occupies 2 GB, making the management more complex. In general, the extra I/O incurred by the system in a memory-limited scenario, would compensate for the time improvements obtained by reducing the GraphPool size. However, in this case, each movie and all its information is read only once and is not used again, thus eliminating any extra I/O.

4.2.2 Social Networks

Previous experiments show that DEX can be used to explore the links between different entities in a graph. Now, we depict an example where DEX is used to analyze social networks. In this case the focus is the relationship between different groups of nodes with the same affinity.

Let us consider all the actors and actresses in IMDb who have participated in the same movie to form a group in a social network. Specifically, we define a *partnership* as the relationship between two actors or actresses who have performed in the same movie. Additionally, we impose two restrictions to this query. First, we restrict the explored titles to only those titles corresponding to a movie, excluding TV chapters. Second, we restrict the cast roles to 'actor' or 'actress'. We apply these two conditions because (i) we want to increase the query complexity rather than always exploring everything and, (ii) the IMDb data set contains a lot of information extracted from interview TV programs that could provide unrealistic relationships between actors or actresses, who participated on different days in the same program and never actually met.

***Query 2 (Q2)*: Find the minimum collaboration distance between two actors or actresses.** The first query exploring partnerships, Q2, tries to find the minimum distance between two partners. If distance is 1 it means that both have worked in the same movie; a distance of 2 means that they never played a movie together, but it exists at least another partner who has performed a movie with both of them. Note that this query requires a virtual transformation of the DbGraph, where DEX simplifies the original graph and generates a more simple graph containing actors and actresses exclusively, connected by an edge if they have performed in the same movies. Assuming this graph, the problem is reduced to find the minimum number of edges to find a path between two nodes.

Before entering in the details of Q2, we show an example extracted from IMDb using DEX to understand the degree of relationships in the data set. Tom Hanks shares a partnership relation with 2293 actors or actresses at distance 1. He is related to 205333 more at distance 2, 531939 at distance 3, and so on. This means that only exploring 3 levels, more than 730000 actors and actresses are related to Tom Hanks.

Q2 is executed using the famous Hale Berry and John Wayne, who played his last western in 1976. The distance between them is 2. This means that they never worked together, but there is at least one person, Martin Landau in this case, who played films with both. The query is executed in less than 1.52 seconds when the memory is not limited and involves near 11000 operations over the NAME, TITLE and CAST data sets. The execution time obtained after limiting the available memory is 1.79 seconds, that is 15% slower than the in-memory case due to the intense use of bit vectors.

***Query 3 (Q3)*: Find the full relationships network of all the partners of actor *a*.** Q3 is a more complex query. Instead of looking for relationships between two partners, we are now interested in knowing the full network of relationships of all the partners of one actor or actress. This kind of analysis provides us with a lot of information about partnership patterns, groups of actors based on the casting agencies, director preferences, or others.

To test the query we select the actress Scarlett Johansson, who has worked in 31 movies. After 6 minutes of execution, DEX returned a dense graph with 1051 partner nodes and near 5.5 million partnership edges. The query required more than 4 million operations on the DbGraph, and it shows the capability of DEX to run complex queries and manage huge volumes of data in a reasonable amount of time. Note that the average execution time is identical for the two tested memory configurations. In this case, the subset of bit vectors necessary to solve the query is smaller than 128 MB. Therefore, thanks to the structures used by DEX, it is possible to efficiently answer this query on a 1.14 GB database.

4.2.3 Pattern Recognition

Pattern recognition defines a different kind of queries, where a lot of potential graphs can be created and explored, but only a few of them will qualify because they match a certain pattern. While previous queries were executed using the SELECT, EXPLODE and TRANSFORM phases of the Exploral process, in this case the phases FILTER and MATCH will decide which graphs will become part of the final result.

***Query 4 (Q4)*: Find all the directors that have worked with the same actress in three different movies made in a period of time of five years.** As an example of pattern recognition, Q4 tries to find all directors with three movies in less than five years with the same actress, i.e., it would be some kind of 'muse' detector query. This is a complex query that not only requires the pattern detection but also involves several data filters like director role, movie kind or actress role.

The results of this query show that up to 4705 directors have these pattern in their casts. An example is given in Figure 8, were Peter Jackson has collaborated with Cate Blanchet, Liv Tyler and Katie Jackson in the 'Lord of the Rings' trilogy. The runs required more than 6 million operations over the DbGraph in near 6.5 minutes, giving an average of less than 0.09 seconds per graph. The differences between the two memory configurations are not significant as for Q3.

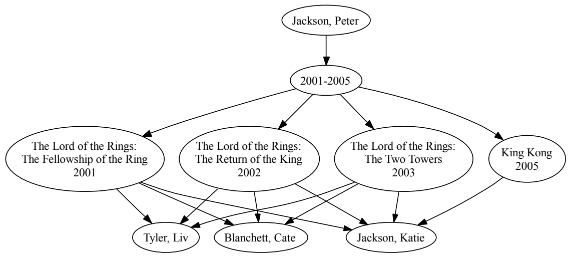

Jackson, Peter
2001-2005
The Lord of the Rings: The Fellowship of the Ring 2001
The Lord of the Rings: The Return of the King 2002
The Lord of the Rings: The Two Towers 2003
King Kong 2005
Tyler, Liv
Blanchett, Cate
Jackson, Katie

Figure 8: RGraph for Q4 ('Peter Jackson')

4.2.4 *Keyword search*

We have shown the ability of DEX to return a set of results when the user knows the schema or at least part of it, i.e., we assume that the user has knowledge on how the data is structured. However, this assumption may be unrealistic in some scenarios like the WEB or documental databases. In this last experiment we show that DEX is also suitable to perform a keyword search, where the user is assumed not to know anything about the organization of the data.

Note that most of keyword search engines are based on data storages where data has been tokenized and indexed to provide fast retrieval of documents related to keywords. In conventional database models like the relational model, this type of search requires a full indexing of all the string columns and it becomes unfeasible due to the high cost in terms of storage size and performance.

In this scenario, DEX must find objects that contain a given keyword in the DbGraph. DEX can take advantage of the dictionaries and compressed structures mentioned before to scan for keywords in many or all the attributes of the DbGraph. Obviously, the chances for DEX to perform better than a documental or a tokenized storage are low, but we show that DEX can provide the functionality in a reasonable time without special configurations or extra data as in other database models. In addition, DEX does not only return a list of entities including the set of keywords, but an entity-context graph containing interesting extra information for that entity, enriching the quality of the results.

***Query 5 (Q5)*: Return all the context information of all the entities containing the keyword *woody*.** In order to test the ability of DEX to answer a keyword search query, we run Q5 to try to find all occurrences of "woody", in any attribute and ignoring the case, and its context information. With this query, we expect to find occurrences like for instance "Allen, Woody" or "Woody Woodpecker" in the whole DbGraph. For each occurrence, DEX will build a small graph to link it to the closest entities in the DbGraph. This way, in the case of Woody Allen, we will not only obtain the person "Allen, Woody", but other information like his movies or information related to his person. For this experiment, we explode the entities that are directly related to the entity containing the keyword.

We executed two versions of the query: Q5a restricts the scan attributes to "name" and "title", to provide an example of the cost of an oriented search, while Q5b is a full database scan without any restriction. Q5a returns 384 graphs, out of more than 3 million candidate graphs, that contain on average more than 10 nodes and edges each. It requires only 18 core operations that are executed in less than 1.9 seconds having the whole memory available and 2.6 seconds using limited memory. Query Q5b returns 3308 graphs out of more than 24 million candidates. The time required for this query is 16.4 seconds using 41 core operations, and 21.2 seconds when memory is limited. This second query takes more time not only because of the number of attributes scanned is larger, but also because of two specific attributes in movie_info and person_info that contain very long text strings difficult to scan in a non-tokenized implementation.

Note that the differences between the two memory configurations for this case are more noticeable compared to those in the other queries. The explanation is straightforward. To answer this keyword search query it is necessary to load dictionaries in memory in addition to the bit vectors. Since the memory has been strictly limited, the system has to incur in extra I/O to solve the query. Note also that, even in the case where memory is limited to $\frac{1}{20}$ of the total available system memory, execution times are only 37% and 27% slower for queries Q5a and Q5b, respectively.

4.3 Query Performance Analysis

In order to analyze in more detail the performance implications of the resolution method used to answer a query, we execute a simple query using different DEX configurations. The new query selects all the movies of a director. We test two different query plans, Qa and Qb, based on two alternative solution methods for the first constraint. In Qa, the person is selected and, for each of its casts, we filter that person depending on whether it is a director or not. In Qb, we first select all the 'director' casts. Then we select all the casts of the person, and finally, we make an intersection to know when he/she matches the role. Finally, in order to perform a comprehensive analysis, we execute two configurations of these queries: QS to get the movies of one single person, 'Woody Allen', and a full scan QF to identify all the directors and their movies.

Table 5 shows the results for the four combinations. As we expected, the filter version QSa is clearly faster than QSb. The reason for this is that the resulting selection has only one node and the cost of selecting all the potential directors is very high. On the other side, the full query QFb is close to 30% faster than QFa because it is trying to create more than 2 million graphs to return a result of more than 120000 query graphs. In this case, it is clear that the cost of the previous selection of potential casts is insignificant with respect to the total resolution time and the intersection operation is, on average, more efficient than checking individually each candidate node. The reason is that the intersection is implemented very efficiently because it implies the execution of a logical *and* operation between two bit vectors.

Table 5: Performance Analysis

query	potential	graphs	nodes	edges	selects	t(s)	t(s)/graph
QS_a	1	1	44	43	45	0.006	0.0060
QS_b	1	1	44	43	46	0.125	0.1250
QF_a	2e6	1.2e5	6.8e5	5.6e5	2.6e6	275.663	0.0023
QF_b	2e6	1.2e5	6.8e5	5.6e5	2.6e6	209.733	0.0017

These experiments show that the best strategy to solve a query depends on the data to be retrieved and, therefore, it becomes necessary to create a different execution plan to solve each query. In this situation, a graph database optimizer becomes necessary to find the optimum query execution plan.

5. RELATED WORK

The number of applications that needs for a graph data model to represent the information in their domains is very large. Examples of these are Genomic databases [8], software engineering design tools [11], and the web [14]. For all those examples, it is necessary to build graph model databases with open querying capabilities and not only keyword search engines that allow to extract pure tabular information.

There are many graph database models in the literature such as GOOD [6], GRAM [2], GOAL [10], GraphDB [9], GRAS [11], GRACE [15] and RDF [3] [12]. Among these, GRAS and GRACE propose an implementation of a graph database management system for small and medium sized databases. RDF has become a de-facto standard for describing resources in the form of a graph. Increasing the semantics of the results in keyword search has been addressed in several approaches, where special-purpose tools, like BANKS [5] or Précis [13], were built to map a relational database into a graph to be returned as a response for a keyword search query. Other interesting works have focussed on the performance of graph querying, like the work presented in [1] and GraphGrep [7].

6. CONCLUSIONS & FUTURE WORK

This paper places emphasis on the importance of complex graph querying in Information Retrieval environments. We show that the need for systems with high performance capabilities, integration capacity and querying versatility justify a whole area of research.

In particular, we presented DEX, a high performance graph database querying system that allows for the integration of multiple data sources. DEX shows to be a basic research tool and a practically usable piece of software for the experimentation on large graph databases in different areas of research. Next steps after this work include studying the performance of graph-database operations, optimizing graph queries, studying integrity constraint preservation and exploring the use of graph-databases in parallel and distributed environments, among others. Overall, DEX will be the vehicle for different approaches of our research in the future.

7. ACKNOWLEDGMENTS

The authors from UPC want to thank Generalitat de Catalunya for its support through grant number GRE-00352 and Ministerio de Educación y Ciencia of Spain for its support through grant TIN2006-15536-C02-02. Jordi Nin wants to thank the Spanish Council for Scientific Research (CSIC) for his I3P grant.

8. REFERENCES

[1] R. Agrawal and H. Jagadish. Algorithms for searching massive graphs. *IEEE Transactions on Knowledge and Data Engineering*, 06(2):225–238, 1994.
[2] B. Amann and M. Scholl. Gram: a graph data model and query languages. In *ECHT '92: Proceedings of the ACM conference on Hypertext*, pages 201–211, New York, NY, USA, 1992. ACM Press.
[3] R. Angles and C. Gutiérrez. Querying rdf data from a graph database perspective. In *ESWC*, pages 346–360, 2005.
[4] R. Angles and C. Gutierrez. Survey of graph database models. Technical Report TR/DCC-2005-10, Computer Science Department, Universidad de Chile, October 2005.
[5] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. *In Procs. of ICDE, 2002.*, pages 431–440, 2002.
[6] M. Gemis, J. Paredaens, I. Thyssens, and J. V. den Bussche. Good: A graph-oriented object database system. In P. Buneman and S. Jajodia, editors, *Proc. of the 1993 ACM SIGMOD, Washington, D.C., May 26-28, 1993*, pages 505–510. ACM Press, 1993.
[7] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs. *International Conference on Pattern Recognition*, 02:20112, 2002.
[8] M. Graves, E. Bergeman, and C. Lawrence. A graph-theoretic data model for genome mapping databases. *hicss*, 00:32, 1995.
[9] R. H. Güting. Graphdb: Modeling and querying graphs in databases. In *Proc. of the 20th VLDB Conference*, pages 297–308, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.
[10] J. Hidders and J. Paredaens. Goal: a graph-based object and association language. *CISM - Advances in Database Systems*, pages 247–265, 1993.
[11] N. Kiesel, A. Schuerr, and B. Westfechtel. Gras, a graph-oriented engineering database system. *Information Systems*, 20(1):21–51, 1995.
[12] 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.*
[13] G. Koutrika, A. Simitsis, and Y. Ioannidis. Précis: The essence of a query answer. In *Proc. of the ICDE'06 Conference*, page 69, Washington, DC, USA, 2006. IEEE Computer Society.
[14] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tompkins, and E. Upfal. The web as a graph. In *Proc. of PODS '00*, pages 1–10, New York, NY, USA, 2000. ACM Press.
[15] 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 *COMAD*, pages 138–147, 2005.