Chapter 2
Massive Graph Management for the Web and Web 2.0

Maria Giatsoglou[1], Symeon Papadopoulos[1,2], and Athena Vakali[1]

[1] Aristotle University, 54124, Thessaloniki, Greece
mgiatsog@csd.auth.gr,
avakali@csd.auth.gr
[2] Informatics and Telematics Institute, CERTH, 57001, Thermi, Greece
papadop@iti.gr

Abstract. The problem of efficiently managing massive datasets has gained increasing attention due to the availability of a plethora of data from various sources, such as the Web. Moreover, Web 2.0 applications seem to be one of the most fruitful sources of information as they have attracted the interest of a large number of users that are eager to contribute to the creation of new data, available online. Several Web 2.0 applications incorporate *Social Tagging* features, allowing users to upload and tag sets of online resources. This activity produces massive amounts of data on a daily basis, which can be represented by a *tripartite graph* structure that connects users, resources and tags. The analysis of *Social Tagging Systems* (STS) emerges as a promising research field, enabling the identification of common patterns in the behavior of users, or the identification of communities of semantically related tags and resources, and much more. The massive size of STS datasets dictates the necessity for a robust underlying infrastructure to be used for their storage and access.

This chapter contains a survey of existing solutions to the problem of storing and managing massive graph data focusing particularly on the implications that the underlying technologies of such frameworks have on the support/operation of Web 2.0 applications using them as back-end storage solutions, as well as on the efficient execution of web mining tasks. Considering the category of STS as an example of Web 2.0 applications, the requirements that are posed for the management of STS data are thoroughly discussed. On the basis of these requirements three frameworks have been developed, using state-of-the-art technologies as backbones. The results of benchmarks conducted on the developed frameworks are presented and discussed.

1 Introduction

The widespread adoption of Web 2.0 tools and technologies that took place during the last years has fundamentally changed the way information is published on the Web. A plethora of Web 2.0 applications, including Social Tagging Systems, Wikis, and Blogs, have emerged, amongst which there are some that recently gained

A. Vakali & L.C. Jain (Eds.): New Directions in Web Data Management 1, SCI 331, pp. 19–58.
springerlink.com

profound success. Some of the most well-known examples of successful Web 2.0 applications are: *Facebook*[1], a social networking website counting hundreds of millions of users, *Flickr*[2], a photo management and sharing application that allows users to tag pictures and form communities, and *delicious*[3], a social bookmarking web service where users can store, share and retrieve bookmarks. What is common between all Web 2.0 applications is that the activity of users results in data that are interconnected through associations, thus forming a network.

The breakthrough of Web 2.0 applications was accompanied by the eagerness of a large proportion of people to join them and to actively contribute to the generation and publishing of Web content. This type of user activity produces massive amounts of data on a daily basis, regarding the uploaded content itself, as well as the relations formed between (a) users, (b) users and shared/uploaded content, and (c) content and metadata (such as *tags*) associated to it by users. A rough idea of the amount of these data can be drawn taking Facebook as an example, where each week more than 3.5 billion pieces of content (such as web links, news stories, blog posts, notes, photos) are shared [67]. The data magnitude, the need to model their relation structure, as well as to efficiently store and retrieve them, have created new challenges in the field of data management. Classic data management solutions, such as data warehouses, seem to be inadequate to store efficiently massive sets of relational data. Moreover, emphasis has been moved from traditional entry-based data access, e.g. customer records, to *navigational access* that allows reaching e.g., the references of an article, the friends of a user via friendship links, etc. The design and implementation of a robust data management framework that manages to maintain a stable performance as the size of data increases, and support navigational queries in a optimal way is still a challenging task for web-scale retrieval systems.

The existence of such massive amounts of data containing complex and emerging structures has also given new impetus to the field of data mining. The information of how users or online resources relate to each other, as well as how users react to resources has captured the interest of researchers, as it was soon realized that it could be exploited to deduce interesting conclusions about how groups of people characterize resources and interpret content, or even what pieces of information tend to be more popular among them. The analysis of Web 2.0 data is further motivated by the notion that the collaboration and contribution of many individuals results in the "formation" of a shared or group intelligence, characterized as *collective intelligence*. Collective intelligence is a new source of information that can be utilized in a variety of applications, as it is produced by the contribution of multiple people representing different views and ideas. For example, it can be exploited in order to uncover groups of either users that share common interests, resources that seem to belong to the same thematic region, or tags (usually referred to as *communities* of users, resources, or tags, respectively). The discovery of such meaningful communities can be utilized in applications such as recommender systems, in order to

[1] `http://www.facebook.com/`
[2] `http://www.flickr.com/`
[3] `http://delicious.com/`

increase their efficiency. For example, *tag communities* can be used in a system that recommends tags to users that they would possibly find relevant to a given resource.

The analysis of relational data produced by Web 2.0 applications, however, requires the use of special methods and poses a question on the data structure that should be used for storing and accessing them. A natural way to model Web 2.0 data seems to be the *network* or *graph* model where *nodes* represent entities/objects and *edges* represent the relations that exist between them. In order to enable the progress of research on such relational datasets continuously increasing in size, a prerequisite is the availability of a robust framework, appropriate for storing and accessing graph-based data. Some of the most challenging issues that should be carefully taken into consideration are: the storage of large graphs (e.g. of 10^9 nodes and 10^{10} edges) in a form that will be as compact as possible, the support for reasonably fast graph traversals and updates, and the design of a framework that will be easy to use and adaptable to the specifications of individual applications.

This chapter provides a review of several solutions and infrastructures used for the storage and analysis of very large graphs, and also discusses and compares their individual characteristics and limitations. Moreover, the special case of using a Social Tagging Systems (STS) as a source of data that can be modeled as a tripartite graph is thoroughly discussed, as an interesting application area. After considering and summarizing the requirements for the storage and analysis of data from STS, we present the results of a set of benchmark experiments that have been designed to compare the performance of three STS data management frameworks built upon different graph persistence technologies, with respect to the storage and management of graph-based data derived from social tagging applications.

The rest of the report is structured as follows. Section 2 discusses the challenges presented by the analysis of massive graphs and includes a categorization of the different available solutions for the management of graph-structured data. Section 3 and Section 4 provide an overview of some of the most recent graph management solutions that belong to the category of transaction graph databases and data mining-oriented solutions, respectively. Section 5 presents STS as an application setting, describing some of the state-of-the-art data mining tasks that are currently being applied in the area, and also summarizes the requirements that these tasks impose on the underlying framework used. Section 6 describes the architecture of three frameworks that have been developed for the management of STS data, presents a set of benchmarks experiments designed to test and compare their performance, and discusses the benchmark results. Finally, Section 7 concludes the chapter.

2 Handling Massive Graphs on the Web

The study of the Web has recently emerged as a new research field. Researchers started to model the Web as a network consisting of nodes representing web pages and edges representing the hyperlinks that connect them, forming the so-called *Web Graph*. The edges in such a model can be: directed (e.g. a hyperlink leading from web page x to web page y), or undirected (e.g. a hyperlink leading from web page x

to web page *y* and vice versa). One of the earliest application domains that exploited the graph model of the Web to extract knowledge was the domain of the Web search engines [10,37]. However, as the Web gained more and more popularity, the number of web pages made available for analysis, acquired usually with the help of web crawlers, was rapidly increasing. While the Web started to reach the size of billions of web pages with ten or hundred times more edges, technological advances made it possible to collect for analysis datasets of sizes proportional to the aforementioned numbers. The availability of such large datasets posed new questions on what techniques and algorithms should be employed to analyze the data.

The obvious problem is that as sizes are getting bigger, the main memory of an average personal computer does not suffice anymore in order to load and manipulate all of the data at once. This has created the need for the development and employment of alternative storage and analysis techniques (Figure 1). Some of the most straightforward approaches are:

- to compress the data so as to reach a size small enough in order to fit in an average computer's RAM and then analyze them,
- to store the data in an external memory repository, and fetch them in batches when required by the analysis algorithm, combining possibly a caching schema to increase performance,
- to use a cluster or a grid of computer nodes in order to distribute the data so as to fit into each node's RAM for faster analysis, and then aggregate the result.

A prerequisite for efficient access to Web and Web 2.0 data within information retrieval scenarios or during the execution of demanding analysis operations is the existence of a robust underlying graph management framework. Frameworks for large graphs' management are usually disk-based, enabling the persistent storage of the large amounts of graph data. There are numerous approaches as to how to store and provide access to such data, that make use of existing infrastructures. Figure 2 depicts a categorization of existing persistent graph frameworks. Existing solutions can be distinguished in two generic categories depending on the reason why the

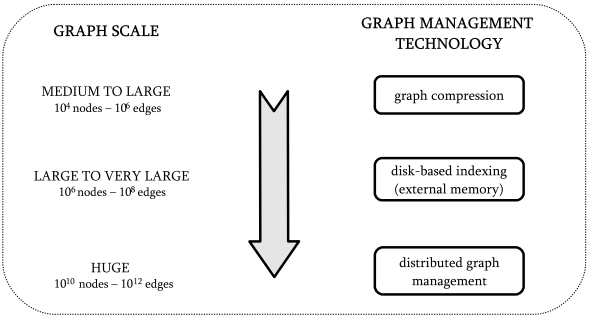

Fig. 1. Techniques to store and analyze graph data depending on graph scale

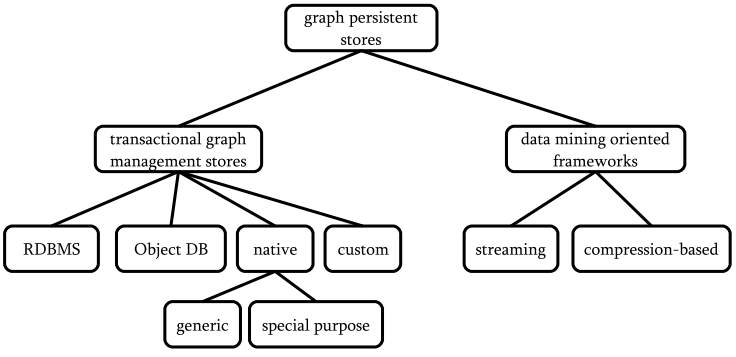

Fig. 2. Categories of graph management frameworks

storage and availability of the data is required: (a) transactional graph databases, and (b) data-mining oriented solutions.

Transactional graph databases can be used for the management of graphs where data (modeled as graph nodes or edges) can be inserted, deleted or updated on demand. This type of frameworks support ACID transactions to ensure reliable processing of database operations. The underlying infrastructures are disk-based enabling the persistent storage of large graphs. The infrastructures that can be used in a transactional graph database can be classified as follows:

- Frameworks based on Relational Database Management Systems (RDBMS),
- Frameworks based on Object Databases,
- Native graph stores, characterized as either: (i) generic, or (ii) special-purpose,
- Custom solutions.

In addition, as mentioned above, there is a requirement for frameworks to store and allow access to web graphs for data mining purposes. The most usual case in graph data mining (or graph mining) is to examine static datasets, and analyze data with algorithms that involve random navigational access to graph nodes and edges. Graph mining operations therefore pose different requirements to the respective data management framework, e.g. there is no need for graph updates, and also the graph accessing mechanisms should be as fast as possible in order for the algorithms to execute in a reasonable time. In general, data mining-oriented solutions can be distinguished in two subcategories:

- Streaming,
- Compression-based.

The categorizarion of graph frameworks depicted in Figure 2 is based on their suitability for a particular application setting. However, frameworks belonging to these categories may address the problem of scalability of the graph data in a different way. In particular, when the size of data is very large, persistent graph frameworks based on distributed computing infrastructures can be used in order to exploit the

storage capacity of multiple computer nodes. In Sections 3 and 4 the categories of: (i) transactional graph databases and (ii) data mining-oriented graph management solutions, respectively, are thoroughly discussed. Each section presents representative examples of frameworks and methods belonging to the respective category, including examples of the special case of distributed graph management solutions.

3 Transactional Graph Databases

Transactional graph databases are disk-based dynamic graph management solutions that operate on the basis of *transactions*. The following subsections intend to provide a thorough insight in the different types of back-end infrastructures that can be used in a transactional graph database.

3.1 RDBMS-Based Frameworks

One early approach for storing networks has been the use of RDBMS, such as *MySQL*. The obvious reason is that RDBMS have been established as the dominant choice for storing data due to their simplicity, robustness, and flexibility as a generic data storage and manipulation mechanism, compared to their alternatives. Moreover, they provide native support for integrity constraint checking, removing this burden from the application side. However, nowadays relational databases receive criticism based on the argument that they are not efficient for managing relational data.

Critics claim that the RDBMS structure is too rigid for storing networks of data, considering that they store both data and their relationships in the form of tables. In particular, the use of tables makes it difficult to fit new kind of data, as their structure should be strictly defined from the beginning and cannot be altered later. Moreover, their most serious limitation is that relational databases are not scalable enough for graph access operations, especially when the size of data is continuously increasing.

However, some people support the use of traditional RDBMS for storing and analyzing graphs. For example, one recent approach [57] proposes:

- the storage of the graph nodes in an SQL table, using an integer identifier for each node as the primary key of the respective record, and also
- the storage of the graph edges in a separate table, using the source and destination nodes for each edge as foreign keys to the nodes table.

Requirements such as the uniqueness of an edge or the prevention of self-loops are ensured with the use of SQL *CHECK* constraints. The graph can then be traversed by either SQL querying, SQL standards *Common Table Expressions* (CTEs) that enable recursions though the nodes, or by using temporary tables [28]. In addition, the construction of the graph's *transitive closure*[4] with the use of CTEs is proposed. A graph's closure can be used to answer queries related to social networks, such as the degree of separation or the possible paths between two nodes. Nevertheless, the

[4] The transitive closure of a graph is a graph which contains an edge (u,v) whenever there is a directed path from u to v.

proposed methods might be too slow depending on the size of the graph and the application performance requirements, so there may be a need for employing caching schemes on top of such a framework. In the following paragraphs two RDBMS are presented, namely (i) H2 and (ii) Oracle DB, which are considered as a suitable basis for graph management frameworks.

H2 database. Using a fast database engine can partially mitigate the performance shortcomings of RDBMS-based graph frameworks. The H2 database engine [69] is a native Java RDBMS that appears a promising choice. Benchmark results show that not only the memory usage of H2 database is smaller, but also its query optimizer results in query times shorter than the times achieved by most competing RDBMS. Moreover, H2 is considered to be scalable as it creates both in-memory and disk-based tables, using hash table and tree indexing or B-tree indexing, respectively. Another important asset is that with H2 there is no limit on the size of the result set of a query, as it buffers the results to disk after a certain size of data is exceeded.

Oracle Database. This database supports modeling networks of data as graphs and analyzing them (since the 10*g* version). These functionalities are included in Oracle's *Network Data Model* (NDM)[5] [49]. NDM enables the storage of the network nodes, links (directed or undirected), as well as ordered lists of links that contain no repeating links or nodes, and are referred to as *paths*. Graphs are represented in object-relational form in the database, using separate tables, whereas queries and updates are performed via PL/SQL. NDM allows posing certain network constraints such as minimum bounding rectangle, path cost, and path depth, and also supports graph operations including shortest path between nodes, minimum cost spanning tree, k-nearest neighbors, k-shortest paths, as well as node and link buffering.

NDM analyzes networks after loading them entirely in memory, thus posing boundaries on the size of network that it can support. Its network analysis capabilities were enhanced in the 11*g* version of Oracle, with the introduction of the *load-on demand* (LOD) approach that made the analysis of larger networks possible [62]. With LOD, the network is not loaded in memory from the beginning, but is partitioned and after that, only the partitions that are required for analysis are loaded in memory automatically. Moreover, partition loading can be accelerated by generating and using BLOB representations.

3.2 Object Database-Based Frameworks

Object or *Object-oriented* (OO) *databases* constitute an alternative solution to RDBMS, combining object-oriented programming language capabilities with traditional persistent data storage and management features. Their use enables developers to model and store complex data as objects, without the need of defining and abiding to a specific relational schema, and simplifies the modification process that is required in case the data model changes. Another argument in favor of object databases with respect to RDBMS is their support for an object schema for data representation both within the application as well as for persistent storage, without

[5] Part of the Oracle Spatial component.

the need for an Object-Relational mapping [60], which is usually a rather cumbersome task. However, the use of *object* instead of *relational* databases results in bigger files for the same data, as they do not separate the structure from the data themselves. Regarding relationships between data, relational and object databases follow two different approaches; (i) relational relationships are usually based on set theory idioms, while (ii) object relationships are mainly based on idioms adopted from graph theory, such as trees, thus depending on the approach, information is accessed in different ways [74]. Moreover, object databases are in general considered to be faster than relational databases for specific access patterns such as navigational access, whereas this is not the case for direct queries to objects.

Object databases can be readily used for storing graph data, mapping the graph structure on an object schema. With such a mapping, e.g. each graph node can be represented by an object of the class *node* with the edges being represented as relationships between the appropriate *node* objects. This constitutes a simpler and more natural way of storing graph data than using a relational database, and is expected to be a faster solution due to the navigational nature of the graph access patterns. One shortcoming of using object databases is the bigger size of the database files.

Although not so widely used as RDBMS, there is a variety of object databases available. In the following paragraphs three popular open-source object databases are presented: (i) Oracle Berkeley DB, (ii) db4o, and (iii) Neodatis ODB.

Oracle Berkeley DB or *Berkeley DB* is an open-source object database, that can be embedded in applications developed in various programming languages, such as Java, C++, Perl, and Python. The use of the Berkeley DB library allows developers to freely decide how data will be stored in a record, without enforcing any constraints on the data. The database comes in three different editions that are also configurable to fit any application's special requirements, with some editions/configurations supporting traditional database features such as ACID transactions, locking, concurrency management, and replication [75].

Berkeley DB stores data as key/data pairs and supports B-tree, hash table, record and queue access methods. It does not support SQL queries, whereas queries can be performed with the use of indexes to each record. According to its developers, Berkeley DB is very scalable, supporting small databases that fit entirely in memory, as well as extremely large disk-resident databases of sizes up to 256 terabytes of data. In order to speed up access to data that are frequently accessed, Berkeley DB offers an in-memory cache [68].

db4o is another open-source object database library, that can be embedded in Java and .NET applications. Similar to Berkeley DB, db4o combines traditional database features, such as robustness, reliability, replication, concurrency support, with simplification of the data storage procedure. An interesting feature is that db4o not only creates automatically the data model that is required to store data objects during a transaction, but also updates the models on-demand [66]. db4o supports *Native Queries* (NQ) instead of string-based APIs, such as SQL, in order to enable database access using the programming language that has been used for the development of the application. Moreover, it supports the *Query by Example* (QbE) API to enable

easy searching for matching objects, as well as the LINQ extensions for .NET. db4o uses B-trees for indexing, supports caching for efficient access to objects, and also provides an in-memory mode. As far as scalability is concerned, db4o can create database files of up to 254 GB.

After conducting the Poleposition database benchmark [6], between db4o and other relational databases, such as MySQL, JavaBD and SQLite, combined with object-relational mappers (JDBC or *Hibernate*), db4o was found to perform better than its competitors for read, write, query, and delete operations, when they involve accessing complex object structures or deep hierarchies. Moreover, its performance was acceptable, although worse than one competitor, for simple flat objects [65].

NeoDatis ODB is also an open-source object database library, embeddable in Java and .NET applications, that supports ACID transactions and can be used in a multi-threaded environment. In ODB every entity (class or object) is characterized by an Object Identifier (OID), which is associated with the respective physical position of the entity in the database file. OIDs are used by pointers in the database for accessing directly a specific object, or for storing relations between objects. They are grouped in blocks that contains the OIDs of the objects that are instances of a given class, in order to enable quick access to them. ODB has also a caching mechanism for mappings from OIDs to objects and reversely, and supports B-tree indexing. ODB provides the following query possibilities for data retrieval: (i) all objects of a specific class, (ii) a subset of objects of a specific class via CriteriaQuery, (iii) a subset of objects of a specific class via NativeQuery, (iv) direct id-based object retrieval, or (v) specific object value retrieval [70].

Based on the results on the Poleposition benchmark, it appears that ODB performs on average better than db4o on most circuits, although there are also some results that indicate that *db4o* is slightly faster than ODB for some circuits [77].

3.3 Native Graph Stores

A natural way to store large graph-shaped datasets seems to be through the use of a persistence engine that directly encodes the graph structure. This type of graph store can be characterized as *native* and should in general support the representation and storage of both nodes including node-related properties, as well as attributed links connecting pairs of data nodes. In the following sections some examples of existing native graph stores will be given, including stores that are generic, i.e. designed to enable the storage of various types of graphs, or are intended for the storage of special graph types, e.g. RDF or XML.

Generic graph stores: *Graph databases* have been recently presented as an efficient way to handle networks of data. Unlike RDBMS, graph databases are designed with inner support for entities that represent *nodes* and *relationships* (or edges), thus making it possible to store and access data in a more efficient and simple way. They aim to provide a complete environment that will make the storage, indexing and quick retrieval of graph data easy, and at the same time retaining traditional

[6] `http://www.polepos.org/`

database properties such as: transactions, durable persistence, concurrency control, and transaction recovery. Graph databases have also been designed taking seriously into consideration the matter of scalability.

One of the first and more complete efforts towards the direction of a generic native graph store has been the development of Neo4j [73] and its release as an open-source graph database. Neo4j is an embedded, disk-based, transactional graph persistence engine that stores data in the form of graphs. Apart from the capabilities of storing nodes and edges and also properties related to them (they are collectively referred to as *primitives*), Neo4j has an easy-to-use, rather straightforward API and provides a variety of extra graph manipulation facilities, such as checks for possible inconsistencies and support for both directed and undirected edges. Moreover, it requires constant time for adding, removing, or accessing a property and creating, deleting, or accessing a node or relationship, whereas it requires linear time for accessing the relationships that involve a given node. However, although in general Neo4j can be considered as fast when concurrent reads take place, it is slower with concurrent updates. This requires careful consideration of the number of operations that will be packed in a Neo4j transaction, which is also affected by the available size of RAM. Moreover, transactions may be useful for ensuring data integrity, but sometimes they can seriously decrease the speed of operations.

It is claimed that Neo4j can scale up to billions of nodes, relationships and properties, but this is a maximum capability relevant only for servers with more than 16 GB of RAM. In general, the scalability of Neo4j is greatly affected by the hardware specifications of the computer station hosting it. For example, it is claimed that an average laptop with 1-2 GB RAM handles tens of millions of primitives, whereas a standard server of 4-8 GB RAM handles hundreds of millions of primitives. However, our experiments with Neo4j (see Section 6) did not give proof for such scalability.

Although Neo4j does not provide a native indexing mechanism yet, it supports indexing facilities by use of the *Apache Lucene* text indexing library. This utility allows indexing nodes with key-value pairs, just like properties, so that they can be queried and retrieved using a given key. The querying process can be accelerated via a LRU cache that holds the most recently accessed results. A limitation of this indexing scheme, however, is that it does not allows indexing relationships.

Another example of native graph store is grDB [30], In grDB graph data are stored grouped in *blocks*, with the block being the smallest amount of information inserted or extracted from the database. The information that grDB stores for a graph is structured in the form of adjacency lists for each node using an integer identifier per node. A grDB instance consists of the *storage component*, that stores the blocks containing parts of the adjacency lists of one or more nodes, and the *block cache component*, that caches some storage blocks in order to improve performance. It also supports multiple levels of storage files.

Special-purpose graph stores: Data encoded in the XML format exhibit an innate tree-like structure that could be used for modeling certain relations that exist in web graphs. More specifically, since a tree is by definition a connected graph

that does not contain any cycles, XML could be possibly used for modeling data nodes with relations that conform to these limitations, or at least can be normalized in more than one trees. For the efficient storage and management of XML data, special databases have been developed. Although the design and functionality of these special-purpose graph stores have been optimized for the storage and retrieval of XML data, they could provide a framework for the storage of graph data (with the aforementioned limitations). Native XML databases, such as *Apache Xindice* and *Tamino XML Server* [79], constitute an interesting alternative to RDBMS, as they do not require the definition of a schema (*schema-free*), thus allowing storing records (XML documents) including semi-structured data that do not necessarily follow a strict predetermined structure. In such databases the storage and retrieval of XML documents takes place according to a (logical) model, such as the XPath model, whereas data retrieval is usually performed by means of the XQuery language. On most occasions, indexing is used to accelerate the querying process [61]. XML databases have, however, received criticism about not being very scalable, as in general XML queries and other mechanisms result in very slow retrievals across large document repositories [58,59]. It also should be mentioned that XML databases are not required to have any particular underlying physical storage model, as they can be built on top or other data storage infrastructures.

Apart from XML databases, structured data can also be stored in RDF[7] or OWL[8] repositories. In general, RDF is a semantically richer way to represent graph-based data, in the form of RDF statements, i.e.subject-predicate-object expressions, known as *triples*, that connect with a specific relationship the subject to the object of the statement. OWL is an extension of RDF that exhibits more expressive power than RDF and enables efficient reasoning. RDF repositories are frameworks dedicated to the management of RDF data in general, that could also be used for the management of web graph data. OWL repositories could also be used for the same purpose, however they are considered to be more specialized than RDF repositories, with the expressive power of OWL being rather needless for the modeling of simple web graph data. Some of the most efficient repositories that support the storage of graph-shaped data either in RDF, OWL, or both, as well as SPARQL queries are Jena [71], Sesame [78], AllegroGraph [64], Virtuoso [80] and OWLIM [76].

3.4 Custom

Apart from the previous infrastructures, custom disk-based solutions that do not belong to a specific category can be employed for the management of web graph data. For instance, the use of a framework based on *Lucene* is proposed. Lucene is text search engine library, that can be easily incorporated in any application that requires text indexing and searching. Indexing with Lucene offers high scalability, cross-platform support, rather small memory requirements, and also fielded search capabilities. Apart from indexing and searching data from other sources, Lucene

[7] `http://www.w3.org/RDF/`

[8] `http://www.w3.org/TR/owl-features/`

also provides the possibility of storing the data in their original form. In general, data are indexed in Lucene as *documents* that contain *fields* of text.

The generic nature of Lucene in combination with its scalability renders it a promising candidate back-end infrastructure for a graph storage framework. A possible implementation would index and store nodes and edges with Lucene, creating a separate document for each entity, and using *terms* to store the properties of each entity. Depending on the application and type of data, for each document, the terms that store the properties that are intended to be used as keywords for querying, will be indexed. In Section 6.1, an implementation of a framework for managing STS data based on Lucene is described in more detail.

3.5 Distributed Transactional Databases

Distributed graph management frameworks are recent solutions that try to solve the problem of limited memory, by distributing the graph in more than one computer nodes that form a cluster. In order to achieve this and for the resulting framework to be efficient, distributed frameworks should employ an appropriate graph partitioning policy and also a query mechanism that will seek and retrieve data from the appropriate computer nodes, minimizing needless queries to irrelevant nodes.

An early research work in distributed transactional graph management, namely MSSG [30] presents a middleware framework for storing, accessing and analyzing massive-scale semantic graphs with update capabilities. The development of MSSG aims to support the storage and analysis of very large graphs reaching trillions of vertices and edges. In order to handle such massive datasets, MSSG has been designed as a distributed database, that supports a large cluster architecture of computer nodes for storing data. Moreover, the framework utilizes the grDB graph database (described in subsection 3.3). The framework, combined with a new parallel external memory breadth-first search algorithm enables fast query responses to the database. The way that MSSG functions is described in brief in the following paragraphs, however it should be stressed that little information has been made available as to how MSSG partitions the graph in order to enable distributed storage.

MSSG was designed using *DataCutter* [5], a development and deployment framework for establishing "filter" services that operate on data "streams" between storage systems and user applications, as a base infrastructure, with the *Ingestion Service*, the *Query Service*, and the *GraphDB Service* modules having been added as integrated components and interfaces. In brief:

- the Ingestion Service is used for entering graph data that are stored to the back-end storage nodes after having been clustered,
- the Query Service allows the analysis of the stored graph,
- the GraphDB Service provides an interface for the available methods implemented for storing and accessing graph data.

The adjacency list of a node can be stored in either a single computer of the cluster, or it can be distributed in more than one computers. Experimental results indicate that the MSSG framework can handle large graph datasets, managing to store

and query a graph of 100,000,000 nodes and 1,999,999,640 directed edges, even though a query with length 5 between the source and destination node is answered in about 12 minutes, which is relatively slow. Experiments also showed that grDB outperforms BerkeleyDB and MySQL in storage and retrieval, considering the tested graphs. Moreover, the performance of grDB on a search query is relatively close to the performance of the implemented in-memory methods under test.

4 Data Mining-Oriented Solutions

In situations where the storage and analysis of static graphs is required, database transactions can be omitted for the sake of performance, and alternative solutions are usually employed. The most efficient solution seems to be to manage to fit the graph structure in main memory by means of graph compression techniques. Another more scalable possibility is to encode the graph's structure in human-readable text files stored in the computer's filesystem and stream the data into memory for analysis. However, this approach requires the adaptation of data mining algorithms to the streaming or semi-streaming model.

In the following subsections both compression-based as well as streaming solutions for the analysis of graph data will be discussed. Moreover, some recent distributed solutions for the management of massive graphs will be discussed.

4.1 Compression-Based Databases

When the available main memory does not suffice to load the whole graph dataset, but fast access to data is required, an efficient graph data compression method is necessary. In the following paragraphs we will present some examples of compression-based graph databases.

WebGraph. One of the earliest and more successful efforts in the compression of web data has been the *WebGraph* framework [6], a suite of codes, algorithms and tools for storing and manipulating large web graphs. The algorithms of WebGraph were based on the *Link* Database [33], an earlier work employing compression techniques to store web graphs that can fit in main memory. Both *Link* and WebGraph perform well in compressing large graphs, combining a number of techniques, such as *referentiation* and *intervalization*. However, WebGraph outperformed its predecessor achieving compression rates of e.g. 3.08 bits per link for a graph consisting of 118 million nodes and 1 billion links.

The success of the WebGraph compression approach is justified considering that the properties of *locality*, *similarity* and *consecutivity* that are typical on the Web were seriously taken into consideration during its development. The property of locality describes the fact that pages belonging to the same host often point to each other via navigational links. Therefore, if we consider a lexicographical ordering of URLs, the source and destination URLs of a link are "close". The property of similarity expresses the observation that pages whose URLs are lexicographically "close", tend to have links to common destination pages (*successors*). Consecutivity

means that the successors of a web page also tend to be lexicographically "close", as they usually belong to the same level of site hierarchy. In order to exploit the aforementioned properties of the Web, WebGraph applies the following technique.

- Given a set of URLs and the information that some of them are linked, URLs are sorted lexicographically and assigned integer identifiers.
- The successor lists of each node are created and sorted by the node identifier.
- The successor list of a node x is expressed with respect to the successor list of a node y with smaller identifier via a *reference list* comprising (a) the *copy list*, i.e. a list of the two nodes' common successors and (b) the *list of extra nodes*, i.e. the set of the successors of x not present in the successor list of y.
- Applying the technique of *differential compression* with encoding methods such as γ coding to the copy list, WebGraph manages to code a link in less that one bit. The list of extra nodes is also compressed using *integer intervalization* and *gap encoding*.

After the compressed graph has been created, WebGraph provides methods for accessing the graph either *randomly* (selecting to access random nodes) or *sequentially* (iterating over all nodes defining the sequence by increasing number identifier). The provided access algorithms are very efficient as they employ lazy techniques to access the compressed graph, thus delaying decompression until it is actually needed. Moreover, WebGraph offers a number of parameterization options in order to allow a trade-off between the *compression ratio* and the time needed to compress the graph, as well as between the decompression speed and the size of the offset array. The compressed graph can either be loaded to RAM, or accessed offline.

The developers of WebGraph also investigated and experimented with different codes to encode the *gaps* that exist between nodes belonging to an ordered successor list [7], after proving empirically that they follow a power-law distribution. They introduced a new set of flat codes for integers, the ζ codes, and proved experimentally that on most cases they are superior to traditional coding methods such as, *Elias* γ, *Elias* δ and *variable-length nibbling*, when they are applied to integers that follow a power-law distribution similar to the distribution of the successor list gaps.

Extensions of WebGraph. A recent work [8], focuses on determining whether WebGraph could be used for efficiently compressing graphs created by data from sources other than web graphs, such as a Social Tagging System. Based on the notion that the compression rate achieved when compressing a web graph depends greatly on the ordering of the nodes, several ordering methods either: *extrinsic* (using information other than that conveyed in the graph itself), or *intrinsic* (using only the information conveyed by the graph structure), have been investigated to determine their effect on the compression rate. As the efficiency of an extrinsic method, such as URL ordering, is doubtful for the case of a network other than a web graph, finding an intrinsic ordering that yields good compression rates is generally considered a challenging problem. In [8], the proposed method is to:

1. order the nodes of the graph randomly,
2. create the adjacency matrix considering each row as a sequence of bits, with 0 denoting the absence, and 1 the existence of a link between two nodes,
3. permute the rows and columns of the matrix so that in the resulting matrix two rows are similar only if they appear consecutively, or almost consecutively.

The two methods that were applied were to either find a permutation that sorts the rows of the adjacency matrix based on the lexicographic ordering, or find a permutation based on the Gray ordering of the row bit vectors. Moreover, two mixed methods combining extrinsic as well as intrinsic characteristics were tested. Both methods use the Gray ordering but limit its application based on the information of the distribution of the nodes within hosts. Experiments with the aforementioned methods using URL ordering showed that (a) the efficiency of each method depends on the structure of the graph itself, (b) intrinsic methods perform very well for the inverse graph, and (c) mixed methods yield better performance on every tested graph.

A recent study [17] focuses on determining whether social networks can be efficiently compressed. This work was motivated by the approach followed in WebGraph considering the properties of locality and similarity that exist for web graphs in order to improve compression ratios. The question posed in this study is whether social networks in general can be effectively compressed by a method similar to the one followed in WebGraph. An easy observation is that the URL lexicographic ordering of the nodes that is a part of the WebGraph compression technique and also a reason for its success cannot be applied to generic social networks. Thus, a new node ordering technique is proposed that uses a simple heuristic based on *shingles*. If we consider two sets A and B, and σ as a random permutation of the elements in $A \cup B$, then $M_\sigma(A) = \sigma^{-1}(min_{a \in A}\{\sigma(a)\})$ is the smallest element in A according to σ, and is called a shingle. The probability that the shingles of set A and set B are identical equals to the *Jaccard coefficient* of the two sets, which is a measure of their similarity. The proposed method regards the out-neighbors of each graph node as separate sets, computes their shingles for an appropriate permutation and then orders the graph nodes according to their corresponding shingles. As a result, nodes with many out-neighbors in common will end up to be close to each other. An alternative technique, *double shingle ordering*, has also been proposed that uses a second shingle for breaking ties produced by the first one.

After the nodes have been ordered, their adjacency lists are compressed using a technique similar to the one employed in WebGraph. Apart from referential and gap encoding, the technique introduces an alternative method for encoding the links that are *reciprocal*, that is the links that are undirected. In particular, this method encodes the reciprocal links in the adjacency list of the node with the smallest integer identifier, and also adds a bit flag for each neighbor encoded in the adjacency list, that declares whether the link is reciprocal or not. With this approach, reciprocal links are encoded only once, thus improving the compression ratio, but this also causes slower queries in the compressed graph.

Experimental results on various datasets indicate that the proposed compression method yields better compression ratios than WebGraph when applied to social networks that are highly reciprocal in structure. Moreover, after experimenting with

various ordering techniques such as, *Gray*, *natural*, and *random* orderings, the *double shingle ordering* managed to achieve the best compression performance. The success of *shingle ordering* with respect to the other methods is attributed to: (a) the reduction of the lengths of the gaps that exist between the neighbors of the adjacency lists, and (b) the exploitation of the properties of locality and similarity. Finally, experimental results indicated that social networks appear to be less compressible than web networks, mainly due to the presence of nodes with low degree.

Taking the above into consideration, WebGraph seems to be a very effective solution as it manages to store a graph in limited disk space and also fetches the neighbors of a node when requested in little time. However, one drawback of the WebGraph framework is that it represents each node with a number without giving the possibility of compressing more information related to a given node. Moreover, it does not provide edge indexing capabilities.

Re-Pair. Several researchers, motivated by the WebGraph compression approach and using the WebGraph framework as a basis for comparisons, tried to find methods with improved performance in terms of compression rates or graph access speed. One such effort [18] proposed a method based on the *Re-Pair* compression technique [36] in order to store a representation of a given graph. Re-Pair is a phrase-based compressor that receives as an input a sequence of symbols, finds the most frequent pair of symbols in it and replaces it with a new symbol, storing the corresponding mapping in a dictionary. This procedure is repeated until every pair in the sequence is unique. Although it is a rather fast (linear-time) technique, it requires a large amount of memory, especially when the initial sequence is long. Therefore, an approximate technique is proposed [18] that can be applied in external memory. In any case, an in-memory hash-table is required to hold the unique pairs of symbols occurring in the sequence (represented by their position in it) along with their frequency. After the hash-table is filled up to a load threshold, no new pairs are inserted, although the traversal is completed so as to calculate the frequencies of the pairs that have already been inserted. Afterwards, the k most frequent pairs are selected and replaced in the sequence with new symbols with a new traversal. The process is repeated traversing the sequence from the position where the insertion of new pairs in the hash-table had previously stopped. When the sequence of symbols resides on secondary memory, the hash-table can store more pairs of symbols as it can occupy all the main memory that is available and also special techniques have been employed so as to avoid unnecessary random access to disk.

In order to apply the Re-Pair technique on a graph representation, the graph is modeled as a sequence of integers representing the graph nodes, each one followed by its adjacency list. However, each node maps to two different distinct integers; one that is used when the integer is placed in the sequence before its adjacency list and one that is used when it is included in the adjacency list of other nodes, thus preventing the integers that mark the start of an adjacency list from being replaced. These alternative representations are removed from the sequence after Re-Pair has been applied, and they are stored in main memory along with pointers to the beginning of their adjacency list. In general, the proposed method takes advantage of the

similarity of the adjacency lists. Moreover, in order to achieve better compression rates, differential compression can also be applied to the lists.

Experiments showed that the proposed method yields compression rates comparable to WebGraph providing faster graph navigation. For example, a graph with 22,744,080 nodes and 639,999,458 edges was compressed into 420 MB (a plain representation would require around 2.4 GB of RAM), achieving two times faster navigation to the compressed graph than when using the WebGraph compression. When differential compression is also applied, slightly better compression rates are achieved, but graph navigation is somewhat delayed.

Virtual Node Miner. Another approach, the *Virtual Node Miner* [14], provides a solution for web graphs that need to be updated after having been compressed, and that also performs well without requiring URL sorting, which is a relatively time-consuming process. The main innovation is that it employs a *pattern mining* approach in order to compress a web graph, using an effective itemset mining algorithm that finds directed bipartite cliques. Moreover, the fact that this method is not based on URL encoding, indicates that it may possibly be used in application domains other than web graphs, such as social networks. The proposed algorithm considers the outlinks (or inlinks) of each graph node as an itemset and aims to identify frequent subsets that in fact represent common links between the graph nodes. The algorithmic steps are described roughly below:

- The graph nodes are clustered on basis of the similarity of their adjacency lists. This step uses *k min-wise independent hash functions* to sample the adjacency lists of each node and then sort the rows of the resulting adjacency matrix lexicographically, in order to bring closer similar adjacency lists and form clusters.
- For every cluster, the algorithm searches for frequent recurring patterns of neighbors, which are actually directed bipartite cliques.
- Every pattern is replaced by a new node, called a *Virtual Node*, that has outlinks to the nodes that formed each specific pattern. After that, the nodes that demonstrated the pattern in their adjacency lists, replace all the outlinks to the nodes that belong to the pattern with just a single outlink to the Virtual Node.
- The process is repeated allowing Virtual Nodes to be reused as actual graph nodes.
- The remaining edges are compressed with an appropriate compression method, such as: ζ or *Huffman* coding.

The resulting graph has a number of extra nodes, the Virtual Nodes, but significantly less links, therefore it is considered to be *compressed*. Experiments indicated that the Virtual Nodes added are about 20% of the original number of nodes, and therefore a moderate overhead to the offset array. Moreover, when compared with WebGraph, experiments showed that the above methods are comparable regarding the compression they achieve. The proposed method has been proven to be rather scalable, as it manages to compress a graph with 3 billions of edges on a computer with 16 GB of RAM in about 2.5 hours. The time required for compression also scales linearly with the graph's size. It is also of interest that if the available memory does not

suffice, Virtual Node Miner can run in batches, thus enabling incremental updating of the compressed graph.

Research with the Virtual Node Miner continued with an effort that used this compression technique to adjust several web graph algorithms so that they could run directly on the compressed graphs and thus demonstrate reduced time complexity [32]. The basis for these algorithms was the invention of a method operating on the compressed graph, that speeds up the multiplication of a graph's adjacency matrix. This multiplication routine was used for computing random walk distributions, finding top singular vectors, estimating the size of neighborhoods, and others, and the resulting methods were used to speed up the implementation of well-known link analysis algorithms such as PageRank [10,37], SALSA [38] and HITS [35]. Experiments showed that the performance of the proposed algorithms was better than the traditional implementations, increasing speed almost up to a ratio close to the respective compression ratios.

4.2 Streaming Solutions

Probably the most intuitive way to encode a graph in a human readable file is either in the form of adjacency lists [9], or in the form of an edge list (e.g. in its simpler form an edge can be expressed as a pair of nodes that are related). The existence of disk-resident files satisfies the persistence requirement for the storage of the graph data, with their sizes being limited only in terms of the available size of external memory. However in order to perform graph analysis tasks, data should be loaded in main memory in an efficient way. In the *streaming model* graph data are streamed from the disk into memory as a sequence of edges. However, the streaming model poses some constraints on the graph mining algorithms, which should be designed so that they can process the edges of a graph in an arbitrary order given only a limited RAM space and desirably making only one pass over them [24]. In order to achieve this, algorithms should be able to make space-efficient data summaries in RAM as data are streamed. This is a considerable challenge, since in general the streaming model poses the constraint of using $O\{polylogN\}$ space and per-item processing time for a given graph with N nodes [41].

There have been some efforts in trying to solve simple graph problems in the stream model, such as the problem of counting triangles in a graph [15].The triangle counting problem in the streaming model is defined as finding the ε-approximation of the number of triangles in a graph with probability at least 1-δ, making one pass over the data stream. The method proposed in [15] assumes that the set of the graph's nodes is known in advance and the graph's edges appear as a stream. It manages to calculate approximately the number of triangles via a technique that uses reservoir sampling, requiring $O(\frac{1}{\varepsilon^2} \times \log \frac{1}{\delta} \times (1 + \frac{|T_1|+|T_2|}{|T_3|}))$ memory cells, where T_i stands for the number of triples of nodes in the graph which have i edges between them. In [13] authors try to provide lower bounds for the more complex problem of finding pairs of vertices that share c neighboring nodes. They give proof that any

[9] The adjacency list of a node is a sequence of its neighboring nodes.

one-pass, randomized data-stream algorithm that determines if a pair of nodes in a directed graph with N nodes shares more than c neighbors requires $O(\sqrt{c} \times n^{\frac{3}{2}})$ bits of space. The large memory bound of the aforementioned method indicates that the application of the streaming model for general graph problems seems to be difficult due to the strict space constraint it imposes [23].

A more relaxed model is the *semi-streaming model* that was initially suggested in [41], as a solution for graph problems where the available main memory suffices for the storage of the graph's nodes, but not for the storage of the graph's edges. This model bounds the storage space for an algorithm that operates on a graph stream by $O(N \times polylogN)$. In [23] the semi-streaming model was further elaborated, allowing also a small number of sequential passes over the graph data. Authors in [23] dealt with various graph problems in the semi-stream model such as the computation of the shortest-path distances between the graph's nodes, as well as the diameter and girth of a graph. They showed that these problems can be approximately solved even with one-pass over the data, via an approximation technique that uses *graph spanners*[10] to calculate shortest distances. In addition to the aforementioned problems, algorithms for the problem of graph matching in the semi-streaming model were also presented in the same research paper. Feigenbaum et al. [24] later improved their method for constructing graph spanners decreasing the processing time per edge from $O(N)$ to $O(polylogN)$. Moreover, they proved that the computation of Breadth-First-Search (BFS) trees is not efficiently executed in the semi-streaming model.

Another recent work [2], studies the problem of *graph sparsification* in the one-pass semi-streaming model. Graph sparsification involves the construction of a compact representation of a given graph through which the size of any cut can be estimated. This problem is therefore connected to the problem of finding an approximate min-cut in a graph. The method proposed in [2] constructs and stores a summary of the graph in main memory, that is updated based on the newly-arrived edges. The original algorithm for finding the sparsification of a graph involves the calculation of the connectivity of every new edge which is impossible unless all the graph's edges are available. However, the proposed method calculates the connectivity of each new edge on the current sparsification, achieving an $1 \pm \varepsilon$ approximation of the cut values of a graph with N nodes and M edges, while requiring $O(N(\log N + \log M) \times \log \frac{M}{N} \times (1+\varepsilon)^2/\varepsilon^2)$ edges in main memory.The semi-streaming model has also been applied to the problem of local triangle counting in graphs [4] (i.e. given a node u, count the number of triangles that are incident to node u). In this research work, apart from the space constraint of the semi-streaming model, algorithms are allowed to $O(logN)$ passes over the data that reside in the external memory. Two algorithms are proposed: one that requires the storage of some intermediate counters in external memory and another that maintains all information in main memory. Given a newly-arrived edge (u, v), both algorithms are based on the approximate calculation of the Jackard coefficient between the two sets of nodes that are adjacent to nodes u and v, respectively.

[10] A subgraph $G'(V,E')$ is a t-spanner of graph $G(V,E)$ if the distance between any pair of nodes in G' is at most t times the distance in G.

A similar graph access approach that is mentioned here, but not presented in detail, is the semi-external model [1]. This model allows for enough main memory to store the graph nodes, but not the graph's edges as well. On the contrary, the graph's edges are stored in external memory, with the model allowing random access to them. However, random access to the disk-residing edges can make the whole process seriously slow.

4.3 Distributed Data Mining-Oriented Solutions

The requirement to perform data mining on massive graphs in a relatively short time has also motivated research in the field of distributed data mining-oriented solutions. Bader and Madduri have recently presented a study including combinatorial techniques for the analysis of large-scale dynamic networks [39]. Their innovation is that they have designed and implemented efficient graph data structures and kernels for modeling temporal graphs of massive sizes that are processed on parallel systems. Temporal information related to e.g. the update or insertion of a node or an edge, are handled by assigning time-stamps to the respective nodes or edges. After experimenting with a number of structures, they proposed a hybrid data structure combining dynamic resizable adjacency arrays for low-degree vertices, with simple self-balancing binary trees, referred to as "treaps" [46], for high-degree vertices. This structure was found to achieve good performance for both insertions as well as deletions, that may be batched or streaming. The data models as well as the algorithms have been designed for multithreaded servers, with multiple cores and a significant amount of both shared cache and main memory. These architectures have been proven to perform much faster in graph analysis algorithms than optimized external memory based architectures [3].

In order to solve or avoid conflicts when e.g. multiple threads try to add data to the adjacency list of the same node, various methods are proposed, such as: following the simple lock-based approach, or allowing each processor to have access to the adjacency lists of only a subset of the graph nodes. In addition, several algorithms have been designed and implemented to execute efficiently graph operations such as: connectivity, path-related and centrality queries. Experiments show that the proposed algorithms scale well on parallel architectures, with e.g. an algorithm based on an implementation of the link-cut tree being able to process queries in time proportional to the diameter of the network. It is also important that the proposed implementation can answer queries related to the evolution of the graph during time.

MapReduce: MapReduce [20] is a programming model with an associated implementation for processing large data sets that may be stored in a distributed filesystem or database. The proposed model, introduced by Google, is applicable for computational problems that can be formed as a set of key-value pairs, e.g. web page indexing based on keywords. The computational process is in general divided into two steps: *map* and *reduce*. Programmers are responsible for creating an application-specific map function that processes the input key/value pair to generate a set of intermediate key/value pairs, and also implement a reduce function that merges all intermediate values associated with the same intermediate key. Each operation

initializes with the splitting of the input files and continues with the assignment of different map and reduce tasks to worker nodes by a special master node. The master node is also responsible for the final aggregation of all results and the production of the output to the original computational problem.

This model has proven to be very efficient for problems that involve accessing large sets of data, however it is disputable whether it can be applied for graph related problems. Some graph related problems can be successfully solved by use of MapReduce. For instance, the computation of PageRank over the Web can be implemented as a chained MapReduce application. However, the main difficulty with solving graph-related problems with MapReduce is that it is is very inefficient for graph traversals, as map workers have access to only a part of the graph. A recent research work [19] investigates the possibility of decomposing graph operations, such as graph simplification, triangle and rectangle enumeration, finding trusses and components, and performing Barycentric clustering, into a sequence of MapReduce processes. In order to overcome the problem that exists with graph traversals, techniques such as the use of multiple map and reduce iterations, or the use of custom optimized graph representations, such as sparse adjacency matrices are proposed.

5 A Case for Web 2.0 Graph Stores: Social Tagging Systems

In this section we focus on a recently evolved research area: the analysis of *Social Tagging Systems*. An introduction to Social Tagging Systems is provided, along with a short review of the most current progress made in several related analysis tasks, and we discuss their special characteristics. Social Tagging Systems are presented as an application setting for massive graph data management frameworks, due to the special requirements that their analysis imposes on the underlying infrastructure.

5.1 Introduction to Social Tagging Systems

An important functionality that has been embraced by many on-line applications is *Social Tagging*. Social Tagging Systems (STS) enable their users to upload content, and to annotate it by means of freely chosen keywords, called *tags*. By relating resources with tags, users enrich them with a semantic meaning that can be of use to other people that come across it. Moreover, the information from STS can be exploited by use of data mining in order to provide enhanced services to users, e.g. recommendations, sophisticated content navigation (e.g. by means of a concept hierarchy representing a resource collection). The study of STS has led to the formalization of *folksonomies*, i.e. lightweight knowledge structures that emerge from the use of a shared vocabulary to characterize resources (*emergent semantics*) [31,40]. The folksonomy model has been established as the most widely-used means to represent and analyze STS-related information, thus its definition is given below.

Definition 1. A folksonomy is defined as the tuple $\mathbb{F} = (U, T, R, Y)$, where U, R and T are the disjoint sets of users, resources and tags, respectively, and $Y \subseteq U \times R \times T$ is a triadic relation between them, representing the annotation of a resource

with a tag by a user. Another way to represent the folksonomy is as an undirected hypergraph $G = \{V, E\}$ consisting of a set of nodes $V = U \cup T \cup R$ that are connected by hyperedges that formulate the set $E = \{\{u,r,t\} | (u,r,t) \in Y\}$.

Rather than working on a hypergraph, on many occasions and depending on the corresponding analysis task, a simplified bipartite graph is produced representing the associations between either: (a) users and resources, (b) users and tags, or (c) resources and tags.

This technique makes the graph analysis easier, as it transforms the hyperedges of the tripartite hypergraph into simple edges. The resulting edges are usually weighted, e.g. in the user-tag bipartite graph, an edge exists between a user and a tag if the user has used this tag to annotate at least one resource, and is weighted by the number of resources that have been annotated with this tag. This graph can be symbolized as: $UT = \{U \times T, E_{ut}\}, E_{ut} = \{(u,t) | \exists u \in U : (u,r,t) \in E\}, w : E_{ut} \rightarrow \mathbb{N}, \forall e : (u,t) \in E_{ut}, w(e) := |\{r : (u,r,t) \in E\}|$ [31]. Relevant expressions can be formulated for the bipartite graph between resources and tags (RT), as well as for the graph between users and resources (UR). A bipartite graph can be represented with a model, such as an adjacency matrix, with each row relating to a member of the first entity type and each column to a member of the other, whereas the value of a cell stores the number of co-occurrences of the respective entity members.

A further simplification can take place, resulting in a graph that represents the co-occurrences between members of the same entity type only. For example, considering the user-tag bipartite graph, two graphs can be produced; one that comprises tags as vertices and edges that represent the annotation of some resource with two tags by a common user, and another that comprises users as vertices and edges that represent the annotation of some resource with a common tag by two users.

If required, a tripartite graph can be also produced, combining the three bipartite graphs, where all the resource-tag, resource-user, tag-user co-occurrences are represented with simple weighted edges. However, the use of bipartite graphs is more often than the use of tripartite, as most algorithms focus on the correlation between the members of two or one entities. For example, the associations between resources and tags is of most interest for a tag recommendation system, whereas the information about which user tagged a resource is not that interesting in this scenario. However, the associations between resources and users would be useful for an application e.g. that recommends resources that may interest users.

5.2 Social Tagging Systems: Analysis Tasks

Ontology extraction: One of the first expectations of researchers was to take advantage of the emerging folksonomies in order to construct ontologies for the Semantic Web [40]. However, early works indicated that the derivation of ontologies from folksonomies presented some serious difficulties, especially because tagging is not necessarily hierarchical such as the ontology structure, meaning that unless it is otherwise stated, an assignment of tags to a resource signifies that the latter is equally characterized by all tags, but it does not imply a hierarchical relationship between the tags. Moreover, there is the widely-discussed problem of tag *ambiguity*

and *polysemy*, i.e. tags that have ambiguous meaning and are used by users to annotate resources that are not relevant to each other. Another issue is the existence of synonyms, that should be identified as tags with a common meaning [26].

Tag meaning disambiguation: Tag ambiguity poses serious challenges to applications that analyze the information included in the STS structure. The annotation of two resources that belong to semantically different categories with common polesymous tags creates a relation between them that is not intended. Therefore, recent research has attempted to address the problem of ambiguous tags. An early effort tried to discover the different dimensions of knowledge in a folksonomy, and after calculating the conditional probabilities of tags in different conceptual dimensions, ambiguous tags were found to have high probabilities on more than one dimensions [52]. In [53], a clustering technique based on the community identification algorithm of Girvan and Newman [42] was employed to find clusters of tags that indicate the different meanings of ambiguous tags in a folksonomy. This work was continued in [55] where the different contexts in which a tag can be used were again on focus, and therefore analysis was conducted for every tag on the associated subset of the folksonomy. Several kinds of network representation were tested and experimental results indicated that tag co-occurrence networks that explicitly incorporate the user-tag associations provide better results in identifying the different contexts a tag can appear in. The results of the proposed automatic tag clustering technique were successfully applied to classify documents retrieved by Web searches.

Study of usage dynamics: Research was also directed towards unveiling the dynamics that characterize the evolution of an STS. Research on the users' behavior in *delicious* showed that users tag collections are growing and evolving over time, due to new interests [26]. However, it was discovered that the set of tags that were used for annotating most of the bookmarks (the resources in delicious) tended to stabilize after a while, exhibiting a stable pattern with fixed frequencies for each tag. This indicates the existence of shared knowledge amongst users, as well as imitation. In addition, the tags that were used to annotate a bookmark by larger numbers of users (the most popular) and also the ones that were used earlier were found to be more representative of the larger category the resource belongs in, therefore have great significance for further analysis. In [29] it was proven, based again on data from delicious, that the distribution of tags is indeed stabilized after some time, following a power law distribution. Moreover, it appeared that after stabilization, analysis of the high-frequency tags of an STS can reveal the collective categorization scheme. Similar results have also been found in [51], where it is stated that tags used to annotate a specific resource are relatively strongly semantically related.

Statistics analysis: It is also interesting to find out the distribution of the user participation in an STS. Earlier research results in social networks in general indicated that user participation follows a power law [50], however subsequent works showed that there were more users contributing content in social networks than those expected from a power law distribution [34]. A recent work [27] showed that the distribution of different users participation follows the stretched exponential distribution, which means that top users are distributed much flatter than those in power law networks.

However, this distribution depends also on the type of content; for example, the distribution of user contribution on content that is more "difficult" to create is more skewed towards a few core users. It should be noted though that the results from this last work have not been based on results from STS.

Clustering: Another direction that has won vivid research interest is clustering, either in users, resources or tags of an STS. The discovery of clusters within a STS has been mainly approached as a *community identification* problem in a graph-structured network. There are different approaches, however, that use either: (a) a bipartite or tripartite graph representation [21], or (b) a simplified tag-tag, user-user, or resource-resource co-occurrence graph. On the first case, the resulting communities are strongly-knit connected components that exist in the graph and are formed by two (or three) kinds of entities, whereas on the second case communities comprise of members of just one type of entity (e.g. tags). Due to their complexity, there have been few methods that have applied clustering for community identification to the induced tripartite hypergraph [9,11].

Tag clustering is a research subject with numerous interesting applications. E.g., the tag clusters resulting from a tag-tag network can be used in a system that recommends to users tags for annotating resources, as they comprise of tags that are semantically "close". Similarly, resource clusters can be used to group objects belonging to the same category, whereas user clusters group people that have exhibited similar behavior patterns in an STS. A tag clustering approach is based on the application of classical community identification methods in the implied graph featuring tag relationships, such as in [16] where a spectral community identification method is employed, in [48] which identifies communities based on graph *modularity* [42] and in [44,45] where a *seed-based community expansion* method has been applied. Moreover, some efforts dealt with the problem of tag clustering, using vector-based agglomerative hierarchical clustering methods rather than the structural properties of the STS graph [12,47]; however they are very slow for large sets of tags. Apart from clustering methods, tags have also been used for the classification of web resources, using optimization techniques that use tag annotations as a feature space for resources and also exploit the link relationships between resources and tags [56].

5.3 Application Setting

All the analysis and mining tasks that are applied to STS and have been discussed in the previous subsection, require a robust graph management infrastructure providing a number of features, dictated by the special characteristics of these systems. The sizes of the graphs formed in the context of STS render them an excellent application setting and motivation for massive graph storage and access frameworks. In the following, the most characteristic STS properties are summarized in order to derive the requirements for a framework developed for their analysis and storage.

- STS users are increasing in numbers and also tend to contribute more in either providing new or annotating existing content. This results in the gradual development of massive folksonomies from STS data that can be available for

analysis. Folksonomy data are encoded in graph structures of hundreds of millions of nodes and ten times or even more edges. delicious, for example, was estimated in 2008 to have 462,168,833 bookmarks and 1,632,204 monthly visitors [43]. These numbers combined with the number of tags used for annotation in delicious is indicative of the massive size of tripartite graphs induce from STS (where both resources, users and tags are considered as nodes).

- STS entities follow power law or skewed distributions. This means that the induced graph exhibits scale free characteristics, i.e there are few nodes that have high frequency and many nodes that are infrequent, thus the network is on its larger proportion rather sparse.
- Information in an STS is updated on a daily basis. However, the number of tags that have been used for annotating a resource is not constantly increasing. On the contrary, after some time the tag distribution stabilizes and each tag used to annotate a resource is characterized by a stable frequency [29,26]. This implies that users often follow common tagging patterns [26].
- STS graphs are often used as an application area for various mining tasks, such as community identification. During graph analysis, algorithms need to access random nodes, extract information that is related to them (e.g. the name of a resource), and also find the edges that are attached to them along with their destination nodes. Taking into consideration the size of the graphs and also the frequency of node and edge accesses that are required in mining algorithms, it is evident that these operations should happen as fast as possible.
- When the induced tripartite hypergraph is simplified in a simple e.g. tag-tag co-occurrence network, some information is lost and cannot be recreated. This information, however, may be useful or necessary for some applications. For example, it is possible that community identification in a bipartite graph will result in more semantically "correct" communities than when using its projection in a simplified entity-entity graph [44]. There is also evidence that explicit information about e.g. user contribution [55] helps dealing with the problem of tag ambiguity.
- Depending on the STS analysis application, the graph representation may include directed on undirected edges. For example, maybe an edge with a resource node as source and a tag node as destination is desirable but the reverse edge does not need to be stored, because it is not useful for the application.
- The STS related data include a number of parameters that may differ, e.g. resources in Flickr may have different attributes than resources in delicious (Flickr resources are images that may have attributes like dimensions, file type, and file size, apart form their URL, whereas delicious resources are bookmarks that may have less attributes such as a title).

On the basis of the above characteristics, the requirements of the framework for the analysis and storage of STS graphs are formulated below.

Graph access methods. The basic graph access operations should be supported, namely node and edge lookup, insertion/update and deletion. Since the stored graphs represent an STS, specializations of the above access operations depending on node

and edge types (U/R/T and UR/UT/RT respectively) should be exposed. In addition, specializations of neighbourhood access operations should be available (i.e. get all neighbour tags for a given user). Finally, the framework should provide graph nodes and edges iterators (predicated with the type of node/edge). In addition, node and edge properties (e.g. frequency values) should be possible to store and access along with the corresponding nodes/edges.

Memory constraints. The framework should support storage and analysis of graphs that do not fit in the main memory of a typical workstation. Partial graph load, external node and edge indices, as well as caching schemes are desired attributes for the foreseen framework.

Support for graph analysis. Since most graph mining techniques require fast access to the graph's structure, it is necessary to hold in memory the largest possible portion of the graph's structure in order to support fast random node and edge access. Such information takes precedence over additional node/edge property values which can be stored in external memory.

6 STS Data Management Framework Benchmark

In order to test the performance of different infrastructures when used as underlying technologies for the management of STS data, we implemented three STS data management frameworks. The design of the frameworks was based to some extent on the requirements stated in the previous section. The developed frameworks, that can be characterized as transactional graph databases, are based on H2, Lucene, and Neo4j, representing the categories of RDBMS, custom, and native graph stores, respectively. The rest of the section is structured as follows. Subsection 6.1 describes the three implemented frameworks in details, subsection 6.2 presents the benchmark tests that have been designed in order to evaluate the frameworks' performance, and subsection 6.3 presents and discusses and results of the benchmarking procedure.

6.1 Participating Frameworks Description

In general, the interesting information that can be drawn from an STS can be expressed as statements, with a given statement representing the assignment of an online resource with a tag by a given user. We made the assumption that the STS-related information (statements) is provided to the data management frameworks in the format of triplets consisting of labels. The first label of each triplet refers to the username of the user that characterized a resource, the second refers to the hash value of the URI of the resource, whereas the third refers to the tag that was assigned by the specific user to the resource. Triplets of STS data can be provided as input to the frameworks either separately (one at a time) or in batches.

Graph Model Description. All frameworks support the management of a graph consisting of *user* (U), *resource* (R) and *tag* (T) nodes. Each node entity includes: (i) a string value (label) denoting a username, the hash value of a URI or a keyword if the node represents a user, a resource or a tag respectively, and also (ii) an

arithmetic value that denotes the node's frequency of appearance in the STS dataset. For example, if a certain user has made 10 tag assignments to resources then the respective node's frequency would be equal to 10.

The nodes of the graph are interconnected via three types of directed edges: (a) User-to-resource edges (UR), (b) User-to-tag edges (UT), and (c) Resource-to-tag edges (RT). Each edge entity also includes an arithmetic value denoting its frequency, e.g. if an edge starting from a resource R and ending to a tag T has frequency 10, this means that tag T has been assigned 10 times to resource R.

Supported Functionality. The developed frameworks support node-, edge-, and graph-based operations. The main operations are lookup, insert/update and delete. More specifically:

- Node lookup, insertion/update and deletion. A node can be of any of the three supported types (U/R/T). A node insertion entails a node lookup (in case of existence, instead of node insertion, a node frequency update is performed). A node deletion entails the deletion of the node's inlinks and outlinks.
- Node neighborhood iteration.
- Edge lookup, insertion/update and deletion. An edge can be of any of the three supported types (UR/UT/RT). An edge insertion entails an edge lookup (in case of existence, instead of edge insertion, an edge frequency update is performed.
- Graph node/edge iteration.
- Graph statistics (number of nodes/edges per type of node/edge.

H2-based Framework. This RDBMS-based framework uses three SQL tables for the storage of the graph's nodes: the USER, RESOURCE and TAG tables. Each table includes three fields: (a) an integer identifier, (b) a string label, and (c) an integer frequency value. All tables storing nodes support the ON DELETE CASCADE SQL feature, so that in case a node is deleted, its outlinks and/or inlinks are also automatically deleted. Moreover, three tables are dedicated to the storage of the graph's edges: the USER-RESOURCE, USER-TAG and RESOURCE-TAG tables. Each table includes three fields: (a) the integer identifier of the source node, (b) the integer identifier of the destination node and (c) an integer frequency value.

Apart from the functionalities mentioned in the previous paragraph, the framework supports also the retrieval of the integer identifier of a node for a given label. Integer identifiers are used in general in order to follow the classical relational database model, and most of all, to reduce the required amount of space for the storage of the graph data. Each communication with the database, whether it is a read, write or delete operation is handled as a separate SQL transaction.

Lucene-based Framework. The Lucene framework uses three separate indexes for indexing and storing the U/R/T nodes and also three indexes for indexing and storing the three types of directed edges. Each entity (either node of edge) is represented in Lucene as a document that contains a number of fields. In our implementation each node document contains a *key* field that stores the node's label and is indexed so that it can be used for retrieving the document when needed. Moreover, each document contains a *frequency* field to store the number of node's occurrences.

The structure of an edge document includes: (i) a key field created by combining the labels of the source and destination nodes, (ii) a field storing the label of the source node (iii) a field storing the label of the destination node, and (iv) a field storing the edge's frequency. The key field is indexed to enable efficient queries for determining whether a specific edge exists. However, the fields storing the labels of the source and destination nodes are also indexed in order for the implementation to support the retrieval of the outlinks and inlinks of a given node. Writes are committed to the indexes in batch, in order to limit the time consuming disk accesses. During subsequent commits the intermediate writes are stored in a cache memory.

Neo4j-based Framework. The Neo4j-based Framework stores all the graph nodes and edges in a common database. However, it allows the definition of a number of relationships types that in our implementation allow distinguishing the category of the graph entities. In total, six relationship types are defined for characterizing UR, UT, and RT edges, and also defining that a node is a user, resource, or tag[11]. Each node includes two properties: the node's label and frequency. Each edge is represented with a relationship that also includes a property storing the edge's frequency. Nodes are indexed using the Lucene-based index implementation provided by Neo4j, so as to allow retrieving a node with its label. Moreover, in order to increase performance the most frequently queried nodes are cached, and also multiple database operations (reads, writes, updates, deletes) are grouped in a database transaction.

6.2 Benchmark Tests Description

The frameworks described in the previous subsection participate in a number of benchmark tests. These tests have been designed to provide an indication of the frameworks' performance with respect to various operations. In particular, the proposed benchmark suite includes the following measurements:

- graph load time (from a triples file)
- disk space usage
- node/edge insertion time (for batches of 1,000 insertions)
- node/edge deletion time (in case of nodes, their in-/out-links are also deleted)
- batch random node query execution times
- batch random edge query execution times (for existing and non-existing edges)
- graph node/edge iteration times
- neighborhood fetch and iteration for a number of randomly selected nodes.

The tests described above are conducted on graphs that contain: (i) real data from a well known STS (Flickr), or (ii) synthetic random data generated by the Erdős-Rényi model [22]. For the synthetic random graph a string generator is used that allows the generation of strings of up to 10 characters. The main difference between the synthetic and real graph data is that the nodes of the synthetic graph are connected with a fixed probability value, whereas the edges of a real STS graph follow

[11] User nodes are connected via the **user** relationship type to a special root node. Similar connections are created for the resource and tag nodes.

the power law distribution. Apart from the type of data there are also some other differences between the loading and insertion tests on real and synthetic graphs. In particular, when the tests are executed on synthetic graphs, a given node or edge is supplied as input only once (along with a frequency value), therefore no updates take place during the testing procedure, and thus there is no need for checking whether the input node or edge exists. Therefore, the nodes and edges are simply added to the graph with the specified frequency parameter as soon as they appear as input. The experiments described above are summarized in Table 1, which also presents the notation that will be used for each type of experiment throughout the rest of the chapter. For example, the notation for a node iteration experiment that runs on a synthetic graph with 1 million edges would be IN-S-1M.

Table 1. Benchmark test notation

Symbol	Position	Meaning	Comments
L	1	Load graph	Load a graph into the graph store.
DN	1	Delete graph nodes	Deletes 10,000 nodes (and their associated edges) from the graph.
DN	1	Delete graph edges	Deletes 10,000 edges from the graph.
QN	1	Query nodes	Executes 10,000 random node queries on the graph.
QEx	1	Query edges 1	Executes 10,000 random edge queries for existing edges on the graph.
QEn	1	Query edges 2	Executes 10,000 random edge queries for non-existing edges on the graph.
DS	1	Disk space	Reports the disk space usage by the graph under test.
IN	1	Node iteration	Iterates over all nodes of the graph.
IE	1	Edge iteration	Iterates over all edges of the graph.
INN	1	Node neighborhood iteration	Iterates over 10,000 random node neighborhoods of the graph.
R	2	Real	A graph created from real-world data is used.
S	2	Synthetic	A graph generated based on the E-R model is used.
K	3	Thousands	Quantifies the size of the graph under test.
M	3	Millions	Quantifies the size of the graph under test.

6.3 Benchmark Results

In the following paragraphs the performance of the developed frameworks based on the results of the benchmark tests will be discussed. Tables 2, 3, 4 and 5 present the experimental results for the disk usage, load, delete, and query experiments, respectively, whereas Figures 3, 4, 5, and 6 illustrate in a a diagrammatic way the results for the node and edge insertion experiments.

The disk usage test results (Table 2) indicate that the H2-based framework has the lowest disk usage for all sizes of real as well as synthetic graphs. This however was somewhat expected as an edge entity stored in the H2-based framework includes the integer identifiers of the source and destination nodes rather than their string labels that naturally occupy more disk space. Between the other two frameworks, the one based on Neo4j seems to be more compact for real graph data. However, when synthetic data are used the disk space usage remains the same for the Neo4j-based framework, whereas it is reduced for the Lucene-based framework. One difference between the synthetic and real graph data is that the label's length for the synthetic

graph nodes is limited to 10 characters the most, whereas the labels of the real graph nodes can contain more characters. For example, the URI hash values that are used as labels for the R nodes have a high possibility of containing more than 10 characters. From the above, it can be concluded that the disk space required for the storage of the documents of the Lucene-based framework has a stronger dependency on the labels' length in relation to the space required for the storage of the entities of the Neo4j-based framework.

Table 2. Disk space usage results

		disk space usage (in Mbytes)		
Disk space test	nodes	H2	LUCENE	NEO4J
DS-R-100K	28,388	6,9	14,5	13,2
DS-R-500K	125,942	36,3	72	61,8
DS-R-1M	235,984	72,8	143,1	119,6
DS-R-5M	1,032,947	379,3	712	559,9
DS-R-10M	1,983,803	766,7	1433,6	1126,4
DS-S-100K	28,388	5,1	8,1	12,8
DS-S-500K	125,942	28	38,9	60
DS-S-1M	235,984	56,2	78,4	116,1
DS-S-5M	1,032,947	283	391,5	542,9
DS-S-10M	1,983,803	570,9	782,4	1126,4

Table 3 presents the total time required to build a graph given either a set of triples of U/R/T labels (real graph), or a set of synthetically generated U/R/T nodes with the respective edges (synthetic graph). According to the benchmark results, the Lucene-based framework is the fastest, whereas the Neo4j-based framework is the slowest of the three, with the Lucene-based framework loading: (i) the largest real graph (10 million edges) 6 times quicker than the Neo4j-based one, and (ii) the largest synthetic graph 4.5 times quicker. In general, synthetic graphs are built in less time than real graphs of the same size which is explained by the differences in the experimental procedure (as it has been stated in subsection 6.2), when loading a synthetic graph there is no need to check whether a node or edge has already been added to the graph). Another observation is that the H2-based framework seems to perform relatively well, with the time required to build a graph being almost proportional to the graph size on most occasions.

Table 3. Load test results

Load test	nodes	H2	LUCENE	NEO4J
L-R-100K	28,388	24,277sec	2,166sec	1,44min
L-R-500K	125,942	2,19min	42,326sec	9,19min
L-R-1M	235,984	5,2min	2,1min	19,13min
L-R-5M	1,032,947	28,49min	17,37min	1,55hr
L-R-10M	1,983,803	1,1hr	43,34min	4,25hr
L-S-100K	28,388	7,670sec	874,700ms	21,816sec
L-S-500K	125,942	37,869sec	9,601sec	2,27min
L-S-1M	235,984	1,15min	35,807sec	5,4min
L-S-5M	1,032,947	7,9min	5,5min	30,6min
L-S-10M	1,983,803	15,10min	14,6min	1,5hr

The results of the node and edge insertion experiments for the real graph data are presented as diagrams in Figures 3 and 5, respectively, whereas the respective results for the synthetic graph data are presented in Figures 4, and 6. Each figure includes three diagrams (one for each framework) that plot the average time for the insertion of a new node or edge calculated for every 1,000 insertions. The diagrams also illustrate the dispersion of the values around the average, with use of the standard deviation values that have also been calculated for every 1,000 insertions. The results of the node insertion benchmark for both real and synthetic graph data indicate that the H2-based framework is the most effective for node insertion requiring on average less than 20msec for the insertion of a node, and managing to maintain a rather stable performance as the size of the graph increases. The slowest solution is the Neo4j-based framework, which also exhibits the highest values of standard deviation. The Lucene-based framework, on the other hand, has the lowest values of standard deviation, thus proving to be very stable. As it can be observed from the diagrams, the average node insertion time for the Lucene-based framework seems to be rising until a number of insertions is reached, and then rapidly fall. This rapid fall indicates that at this point the framework cache is full so the Lucene writer commits the changes that have been cached so far to the disk. Afterwards, the cache is emptied to enable the storage of the new insertions (and possible updates), so the performance of the framework improves. The results for the edge insertion tests (Figures 5, and 6) yield similar findings. Both the H2-based and Lucene-based frameworks are much faster than the Neo4j-based framework. It is noticable that the average times per edge insertion for the latter framework reach very large values as the size of the graph increases. This serious delay of the Neo4j-based framework did not allow us to complete the edge insertion experiment. Between the H2-based and Lucene-based frameworks, although their performance is comparable, the H2-based framework seems to maintain a more stable performance regardless of the graph size. The Neo4j-based framework had an even worse performance for the edge insertion test on synthetic graph data. Apart from the average times calculated for the first 2,000 insertions of edges, the successive results for the first next thousands of insertions were approximately 10 times larger than the respective results for the H2-based and Lucene-based frameworks, however they soon began to increase exponentially. In order to present the comparative results for the three frameworks, Figure 6 has been included using a logarithmic scale for the time axis.

Table 4 presents the results for the node and edge deletion experiments. The average times for node deletion indicate that the H2-based framework performs better than the other two frameworks, whereas the slowest framework for every test has proved to be the one based on Lucene. The Neo4j-based framework maintained the same performance for every graph size. In general, the deletion of a node causes the deletion of all edges that are adjacent to it. Therefore, the time measured for a node deletion includes an extra delay required for fetching and deleting the node's neighbors. The deletion of the neighbors of a node seems to be a serious overhead for the Lucene-based framework, whereas it is automatically executed in the H2-based framework due to the SQL ON DELETE CASCADE constraint. However, it can be observed that the times measured for the Lucene-based framework have the

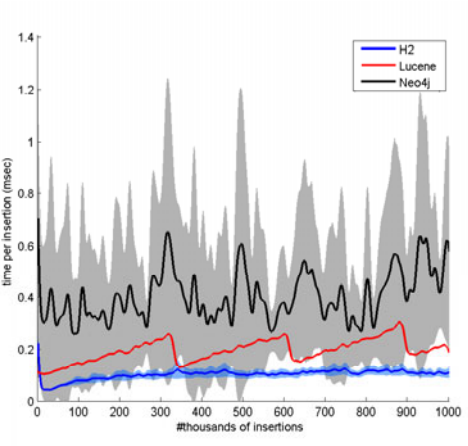

Fig. 3. Diagram of mean time per node insertion in a real graph (per 1,000 insertions)

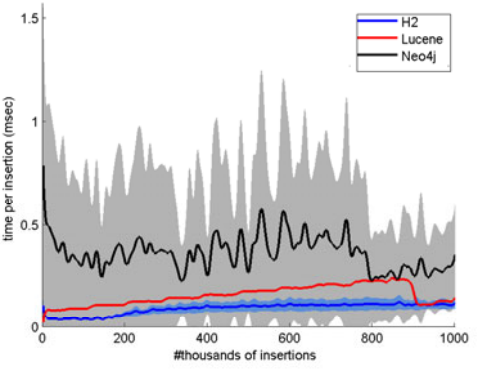

Fig. 4. Diagram of mean time per node insertion in a synthetic graph (per 1,000 insertions)

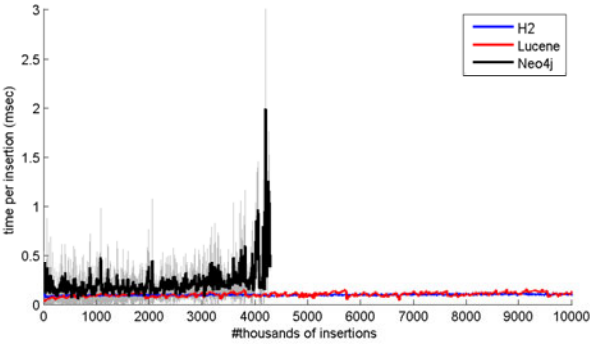

Fig. 5. Diagram of mean time per edge insertion in a real graph (per 1,000 insertions)

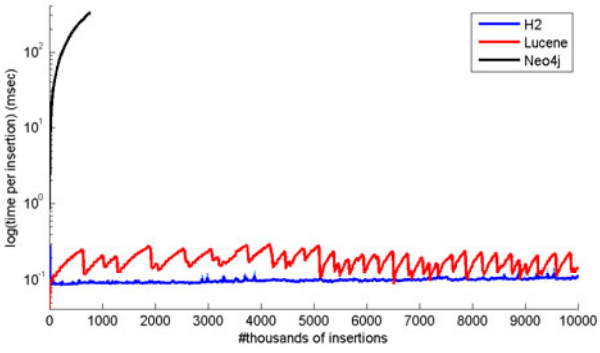

Fig. 6. Diagram of mean time per edge insertion in a synthetic graph (per 1,000 insertions). The time axis is plotted in a logarithmic scale as the time values for the Neo4j-based framework are much higher than the corresponding times for the other two frameworks.

Table 4. Delete test results

	mean time			standard deviation		
Delete test	H2	LUCENE	NEO4J	H2	LUCENE	NEO4J
DN-R-100K	357,947us	567,391ms	3,349ms	5,287ms	0ns	30,187ms
DN-R-500K	1,427ms	1,768sec	3,574ms	13,500ms	0ns	30,162ms
DN-R-1M	1,502ms	3,290sec	3,481ms	7,164ms	0ns	30,171ms
DN-S-100K	182,778us	82,178ms	2,407ms	4,846ms	0ns	30,277ms
DN-S-500K	717,940us	102,943ms	1,766ms	26,585ms	0ns	26,614ms
DN-S-1M	1,102ms	119,799ms	2,602ms	30,351ms	0ns	30,259ms
DE-R-100K	140,674us	1,778us	484,522us	959,511us	9,493us	8,697ms
DE-R-500K	214,985us	2,313us	405,613us	1,735ms	17,285us	8,244ms
DE-R-1M	87,791us	1,808us	348,789us	704,322us	5,711us	7,416ms
DE-S-100K	109,413us	1,624us	704,770us	989,403us	4,371us	8,969ms
DE-S-500K	165,929us	1,638us	465,800us	1,219ms	4,338us	7,418ms
DE-S-1M	62,773us	1,564us	455,313us	17,778us	4,7us	8,47ms

lowest values of standard deviation among all frameworks, something that has been observed in the results of the previous test as well.

The results for the edge deletion experiments clearly show the superiority of the Lucene-based framework. The results of these experiments in combination with the results for the node deletion experiments indicate that this framework is particularly efficient in deleting an edge (and probably a node as they are indexed in a similar way), however it is not very efficient in retrieving the neighbors of a node. Moreover, the average time for an edge deletion does not seem to change linearly with the graph size, but it is affected by the randomness of the edge selection.

The results for the query experiments are presented in Table 5. The experimental results of the tests that involve querying random nodes of real graphs show that the Lucene-based framework has the best performance between all frameworks. Comparing the performance of the other two frameworks, it can be observed that the Neo4j-based framework scales better than the one based on H2 for large graphs. In

addition, the results concerning the H2-based framework show a particularly high value of standard deviation for the graph of 1 million edges. However, the same experiments on the synthetic graph generated results that show that the H2-based framework conducts queries faster than the other two frameworks, whereas Neo4j has the worse performance.

Table 5. Query test results

	mean time			standard deviation		
Query test	H2	LUCENE	NEO4J	H2	LUCENE	NEO4J
QN-R-100K	162,16us	94,74us	271,48us	991,29us	562,32us	1,48ms
QN-R-500K	352,649us	132,264us	374,78us	1,504ms	615,906us	1,155ms
QN-R-1M	1,281ms	182,273us	476,680us	3,247ms	1,151ms	1,966ms
QN-S-100K	28,620us	63,881us	203,272us	179,664us	205,149us	691,344us
QN-S-500K	60,307us	93,835us	226,427us	599,284us	370,230us	528,356us
QN-S-1M	99,469us	115,996us	253,828us	1,103ms	642,891us	942,900us
QEx-R-100K	126,499us	170,284us	96,998us	14,933us	686,622us	70,622us
QEx-R-500K	280,310us	259,337us	211,398us	1,361ms	1,161ms	277,977us
QEx-R-1M	4,483ms	397,403us	233,614us	8,536ms	2,739ms	119,956us
QEx-S-100K	181,804us	222,825us	7,200ms	489,827us	3,99ms	5,95ms
QEx-S-500K	302,62us	180,291us	30,204ms	1,375ms	685,576us	3,178ms
QEx-S-1M	2,329ms	288,105us	55,556ms	7,214ms	1,391ms	0ns
QEn-R-100K	72,585us	4,576us	30,95us	445,929us	1,416us	56,208us
QEn-R-500K	70,521us	6,160us	21,57us	171,65us	1,677us	53,164us
QEn-R-1M	73,515us	7,225us	21,774us	149,983us	2,164us	54,961us
QEn-S-100K	61,904us	7,147us	12,530us	39,135us	6,77us	49,602us
QEn-S-500K	52,799us	6,391us	16,113us	30,711us	28,855us	167,289us
QEn-S-1M	53,358us	3,675us	21,633us	64,305us	1,721us	709,977us
IN-R-100K	44,425us	9,131us	7,848us	48,857us	147,153us	41,66us
IN-R-500K	30,842us	1,599us	4,926us	0ns	29,726us	85,871us
IN-R-1M	30,859us	1,387us	4,789us	0ns	24,403us	155,118us
IN-S-100K	5,181us	1,901us	6,668us	19,959us	74,364us	45,639us
IN-S-500K	4,408us	1,544us	5,947us	28,378us	41,969us	86,139us
IN-S-1M	3,824us	1,362us	6,53us	15,509us	34,185us	125,898us
IE-R-100K	6,174us	2,587us	2,740us	17,906us	78,859us	5,81us
IE-R-500K	7,77us	2,466us	3,862us	13,814us	58,637us	964,668us
IE-R-1M	7,611us	2,592us	2,849us	11,598us	107,551us	13,986us
IE-S-100K	6,129us	1,390us	2,638us	16,993us	2,432us	86,702us
IE-S-500K	4,452us	1,468us	2,434us	14,588us	65,639us	128,164us
IE-S-1M	3,110us	1,410us	3,160us	12,598us	45,841us	696,530us
INN-R-100K	124,460us	329,298us	166,693us	528,406us	593,44us	412,642us
INN-R-500K	171,186us	445,346us	270,393us	831,722us	1,49ms	124,973us
INN-R-1M	194,627us	522,203us	281,274us	939,848us	1,884ms	241,676us
INN-S-100K	100,471us	280,704us	171,670us	314,960us	726,270us	351,630us
INN-S-500K	145,330us	384,964s	195,835us	823,617us	2,335ms	386,801us
INN-S-1M	276,626us	422,135us	226,636us	1,192ms	1,649ms	1,521ms

The results of the tests involving the query of existing edges from real graphs indicate that the Neo4j-based framework has the best performance, as it conducts queries faster than its competitors and also has the lowest value of standard deviation. The H2-based framework, on the contrary, was proven to be very slow for larger graphs, having also a high value of standard deviation. However, the same experiment on synthetic graphs generated completely different results for the Neo4j-based framework, as it was the slowest, with the measured average times being much larger than the times measured for the real graph tests. The framework that had the best performance in these experiments for the larger graphs is the Lucene-based one.

The Lucene-based framework was proved to be the fastest when querying non-existing edges for both real and synthetic graphs, as well as the most stable as it had the lowest standard deviation values. The slowest framework for both types of tests was the one based on H2. Another observation is that the standard deviation for the Neo4j-based framework was much larger for the synthetic graph tests, whereas quite the opposite is true for the H2-based framework.

In all iteration tests (including both node as well as edge iteration) the Lucene-based framework had the best performance. For the node iteration tests it is worth noticing that this framework generated the same average times for all real and synthetic graphs, on the exception of the smallest real graph for which it generated a larger average time. Between the other two frameworks the Neo4j-based proved to be faster for real graphs, whereas their performance was comparable for synthetic graph tests. In general, both the Neo4j-based and H2-based frameworks were faster when querying nodes of synthetic rather than real graphs. However, the Neo4j-based framework was observed to have a very high value of standard deviation for the larger graphs. The Neo4j-based framework performed relatively well in edge iteration tests, with the calculated average times for the real graph tests being comparable to the respective average times for the Lucene-based framework. An observation about the edge iteration results for the Lucene-based framework is that it performed two times faster for the synthetic graph tests in relation to real graph tests.

Finally, the tests that involved querying the neighbors of random nodes, indicate that the H2-based framework is the most efficient for such type of queries for most of the tests, whereas the Lucene-based framework has the worst performance. The single test for which the H2-based framework was outperformed by the Neo4j-based framework is the test conducted on the largest synthetic graph (1,000,000 edges). An observation that gives proof of the poor performance of the Lucene-based framework in relation to the other frameworks is that its performance, when the test was conducted on the largest real graph, was 2.5 times worse than the best performance for the same test, whereas it was approximately 2 times worse than than best performance when the test was conducted on the largest synthetic graph. This again indicates that the Lucene-based framework is particularly slow when retrieving the neighbors of a node.

7 Conclusions and Outlook

The abundance of Web data has created the need for more efficient scalable graph data management structures. With this problem in mind, we presented various solutions for the management of massive web graphs. We considered the special case of STS as an application setting and we defined the requirements that STS data impose on the underlying management framework. Moreover, we developed three different STS data management frameworks and presented their structure and functionality. The developed frameworks were benchmarked in terms of the disk space required for data storage, as well as in terms of how fast they perform data insertion, update, and deletion operations. The experimental results showed that both frameworks have

their pros and cons, and that the choice of a suitable framework for the management of STS data (or web graph data in general) depends on the type of operations that are expected to be performed more often. In general, the custom Lucene-based framework seems to be an efficient solution for the majority of operations, except for those that involve accessing the neighbors of a node, where the H2-based and Neo4j-based frameworks proved to be better solutions. Moreover, although the proposed frameworks have been designed for the case of the STS, they can be easily adjusted so that they are applicable for other types of Web and Web 2.0 graph data. The possibility of testing the performance of frameworks that use other technologies as underlying infrastructures (such as object databases, presented in subsection 3.2, or data mining-based solutions, presented in Section 4) is also worthwhile to be explored as future work.

From the above, it appears that there are different requirements for the management of Web and Web 2.0 graph data, depending on the reasons why their storage, and management in general, is desirable. A Web graph management framework should consequently either be centered on a specific application, or be adaptable to suit many application requirements. An interesting vision would be to combine the characteristics of data mining-based solutions, such as the compression-based databases, with the update functionalities of transactional databases, in a framework able to support applications both for static graph analysis and for managing time-varying graphs. In general, graph data management frameworks should be developed to maximally exploit the available main memory. Approaches towards this direction would be to e.g. store part of the graph structure in main memory and the rest of the data in external memory, or use a computer cluster to increase the size of available main memory to fit the entire graph structure. An alternative approach would be to differentiate between the way the adjacency lists of high-degree and low-degree nodes are stored, so that e.g. the adjacency lists of high-degree nodes are stored in main memory to decrease the time required for their access.

The possibility of developing a framework for handling temporal graphs that would maintain information about the graph's state in different time steps constitutes another interesting future work area. Such functionality could be included in a graph management framework e.g. information about when a node or edge was added to (or deleted from) the graph is stored as an extra attribute of the node or edge. Managing temporal graph data with such a framework would enable querying about e.g. when an edge was added to the graph, which nodes were adjacent to a node at a given time, etc.

References

1. Abello, J., Buchsbaum, A.L., Westbrook, J.: A Functional Approach to External Graph Algorithms. Algorithmica 32(3), 437–458 (1998)
2. Ahn, K.J., Guha, S.: Graph Sparsification in the Semi-streaming Model. In: ICALP(2), pp. 328–338 (2009)
3. Bader, D., Madduri, K.: Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In: Proceedings of the ICPP 2006. IEEE Computer Society, Los Alamitos (2006)

4. Becchetti, L., Boldi, P., Castillo, C., Gionis, A.: Efficient semi-streaming algorithms for local triangle counting in massive graphs. In: Proceeding of the KDD 2008, pp. 16–24. ACM Press, New York (2008)
5. Beynon, M.D., Kurc, T., Catalyurek, U., Chang, C., Sussman, A., Saltz, J.: Distributed processing of very large datasets with DataCutter. Parallel Comput. 27(11), 1457–1478 (2001)
6. Boldi, P., Vigna, S.: The WebGraph Framework I: Compression Techniques. In: Proceedings of the WWW 2004, pp. 595–602. ACM, New York (2004)
7. Boldi, P., Vigna, S.: The WebGraph Framework II: Codes For The World-Wide Web. In: Proceedings of the DCC 2004, vol. 528. IEEE Computer Society, Los Alamitos (2004)
8. Boldi, P., Santini, M., Vigna, S.: Permuting Web Graphs. In: Avrachenkov, K., Donato, D., Litvak, N. (eds.) WAW 2009. LNCS, vol. 5427, pp. 116–126. Springer, Heidelberg (2009)
9. Bothorel, C., Bouklit, M.: An algorithm for detecting communities in folksonomy hypergraphs. Appeared in I2CS 2008, Schoelcher, Martinique, Sponsored by IEEE (2008)
10. Brin, S., Page, L.: The anatomy of a large-scale hypertextual Web search engine. Comput. Netw. ISDN Syst. 30(1-7), 107–117 (1998)
11. Brinkmeier, M., Werner, J., Recknagel, S.: Communities in graphs and hypergraphs. In: Proceedings of CIKM 2007, Lisbon, Portugal, pp. 869–872. ACM, New York (2007)
12. Brooks, C.H., Montanez, N.: Improved annotation of the blogosphere via autotagging and hierarchical clustering. In: Proceedings of the WWW 2006, pp. 625–632. ACM, New York (2006)
13. Buchsbaum, A.L., Giancarlo, R., Racz, B.: New results for finding common neighborhoods in massive graphs in the data stream model. Theor. Comput. Sci. 407(1-3), 302–309 (2008)
14. Buehrer, G., Chellapilla, K.: A scalable pattern mining approach to web graph compression with communities. In: Proceedings of the WSDM 2008. ACM, New York (2008)
15. Buriol, L.S., Frahling, G., Leonardi, S., Marchetti-Spaccamela, A., Sohler, C.: Counting triangles in data streams. Proceedings of the PODS 2006, pp. 253–262. ACM, New York (2006)
16. Cattuto, C., Baldassarri, A., Servedio, D.P.V., Loreto, V.: Emergent Community Structure In Social Tagging Systems. Advances in Complex Systems (ACS) 11(4), 597–608 (2008)
17. Chierichetti, F., Kumar, R., Lattanzi, S., Mitzenmacher, M., Panconesi, A., Raghavan, P.: On compressing social networks. In: Proceedings of the KDD 2009, pp. 219–228. ACM, New York (2009)
18. Claude, F., Navarro, G.: A Fast and Compact Web Graph Representation. In: Ziviani, N., Baeza-Yates, R. (eds.) SPIRE 2007. LNCS, vol. 4726, pp. 105–116. Springer, Heidelberg (2007)
19. Cohen, J.: Graph Twiddling in a MapReduce World. Computing in Science & Engineering 11(4), 29–41 (2009)
20. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
21. Du, N., Wang, B., Wu, B., Wang, Y.: Overlapping Community Detection in Bipartite Networks. In: Proceedings of the WI-IAT 2008, pp. 176–179. IEEE Computer Society, Los Alamitos (2008)
22. Erdős, P., Rényi, A.: On Random Graphs I. Publicationes Mathematicae 6, 290–297 (1959)
23. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: On graph problems in a semi-streaming model. Theor. Comput. Sci. 348(2), 207–216 (2005)

24. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: Graph distances in the streaming model. SIAM J. Comput. 38(5), 1709–1727 (2008)
25. Furtado, P.: Evolving Application Domains of Data Warehousing and Mining: Trends and Solutions. IGI Publishing (2009)
26. Golder, S.A., Huberman, B.A.: Usage patterns of collaborative tagging systems. J. Inf. Sci. 32(2), 198–208 (2006)
27. Guo, L., Tan, E., Chen, S., Zhang, X., Zhao, Y.: Analyzing patterns of user content generation in online social networks. In: Proceedings of the KDD 2009, pp. 369–378. ACM, New York (2009)
28. Guozhu, D., Leonid, L., Jianwen, S., Limsoon, W.: Maintaining Transitive Closure of Graphs in SQL. Int. J. Information Technology 5 (1999)
29. Halpin, H., Robu, V., Shepherd, H.: The complex dynamics of collaborative tagging. In: Proceedings of the WWW 2007. ACM, New York (2007)
30. Hartley, T.D.R., Çatalyürek, Ü.V., Özgüner, F., Yoo, A., Kohn, S., Henderson, K.W.: MSSG: A Framework for Massive-Scale Semantic Graphs. In: Proceedings of the 2006 IEEE International Conference on Cluster Computing, pp. 1–10. IEEE, Los Alamitos (2006)
31. Hotho, A., Robert, J., Christoph, S., Gerd, S.: Emergent Semantics in BibSonomy. GI Jahrestagung P-94, 305–312 (2006)
32. Karande, C., Chellapilla, K., Andersen, R.: Speeding up algorithms on compressed web graphs. In: Proceedings of the WSDM 2009, pp. 272–281. ACM, New York (2009)
33. Keith, H.R., Raymie, S., Janet, L.W., Rajiv, G.W.: The Link Database: Fast Access to Graphs of the Web. In: Data Compression Conference, vol. 0, p. 122. IEEE Computer Society, Los Alamitos (2002)
34. Kittur, A., Chi, E., Pendleton, B.A., Suh, B., Mytkowicz, T.: Power of the Few vs. Wisdom of the Crowd: Wikipedia and the Rise of the Bourgeoisie. World Wide Web 1, 2,19 (2007)
35. Kleinberg, J.M.: Authoritative sources in a hyperlinked environment. J. ACM 46(5), 604–632 (1999)
36. Larsson, N.J., Moffat, A.: Offline Dictionary-Based Compression. In: Data Compression Conference, vol. 0, p. 296. IEEE Computer Society, Los Alamitos (1999)
37. Lawrence, P., Sergey, B., Motwani, R., Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web. Technical Report. Stanford University (1998)
38. Lempel, R., Moran, S.: The stochastic approach for link-structure analysis (SALSA) and the TKC effect. Comput. Netw. 33(1-6), 387–401 (2000)
39. Madduri, K., Bader, D.A.: Compact graph representations and parallel connectivity algorithms for massive dynamic network analysis. In: Proceedings of the IPDPS 2009, pp. 1–11. IEEE Computer Society, Los Alamitos (2009)
40. Mika, P.: Ontologies Are Us: A Unified Model of Social Networks and Semantics. In: International Semantic Web Conference, pp. 522–536 (2005)
41. Muthukrishnan, S.: Data streams: algorithms and applications. In: Proceedings of the SODA 2003, pp. 413–413 (2003)
42. Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. Physical Review E 69(2), 26113+ (2004)
43. Papadopoulos, S., Menemenis, F., Vakali, A., Kompatsiaris, Y.: Analysis of Content Popularity in Social Bookmarking Systems. In: Evolving Application Domains of Data Warehousing and Mining: Trends and Solutions. IGI Publishing (2009)
44. Papadopoulos, S., Kompatsiaris, Y., Vakali, A.: Leveraging Collective Intelligence through Community Detection in Tag Networks. In: Proceedings of the CKCaR 2009 (2009)

45. Papadopoulos, S., Kompatsiaris, Y., Vakali, A.: A Graph-based Clustering Scheme for Identifying Related Tags in Folksonomies. In: Proceedings of the DaWaK 2010 (2010)
46. Seidel, R., Aragon, C.: Randomized search trees. Algorithmica 16, 464–497 (1996)
47. Shepitsen, A., Gemmell, J., Mobasher, B., Burke, R.: Personalized recommendation in social tagging systems using hierarchical clustering. In: Proceedings of the RecSys 2008, pp. 259–266. ACM, New York (2008)
48. Simpson, E.: Clustering Tags in Enterprise and Web Folksonomies. Technical Report. HP Labs (2008)
49. Stephens, S., Rung, J., Lopez, X.: Graph Data Representation in Oracle Database 10*g*: Case Studies in Life Sciences. IEEE Data Eng. Bull. 27(4), 61–66 (2004)
50. Voss, J.: Measuring Wikipedia. In: The 10th International Conference of the International Society for Scientometrics and Informetrics (2005)
51. Wu, C., Zhou, B.: Analysis of tag within online social networks. In: Proceedings of the GROUP 2009, pp. 21–30. ACM, New York (2009)
52. Wu, X., Zhang, L., Yu, Y.: Exploring social annotations for the semantic web. In: Proceedings of the WWW 2006, pp. 417–426. ACM, New York (2006)
53. Yeung, C.A., Gibbins, N., Shadbolt, N.: Tag Meaning Disambiguation through Analysis of Tripartite Structure of Folksonomies. In: Proceedings of the WI-IATW 2007, pp. 3–6. IEEE Computer Society, Los Alamitos (2007)
54. Yeung, C.A., Gibbins, N., Shadbolt, N.: Collective User Behaviour and Tag Contextualisation in Folksonomies. In: Proceedings of the WI-IAT 2008, pp. 659–662. IEEE Computer Society, Los Alamitos (2008)
55. Yeung, C.A., Gibbins, N., Shadbolt, N.: Contextualising tags in collaborative tagging systems. In: Proceedings of the HT 2009, pp. 251–260. ACM, New York (2009)
56. Yin, Z., Li, R., Mei, Q., Han, J.: Exploring social tagging graph for web object classification. In: Proceedings of the KDD 2009, pp. 957–966. ACM, New York (2009)
57. Alberton, L.: Graphs in the database: SQL meets social networks (2009), http://techportal.ibuildings.com/2009/09/07/graphs-in-the-database-sql-meets-social-networks
58. Bergman, M.K.: Scalability of the Semantic Web (2006), http://www.mkbergman.com/227/scalability-of-the-semantic-web
59. Bergman, M.K.: Enterprise Semantic Webs Demand New Database Paradigms (2006), http://www.mkbergman.com/185/enterprise-semantic-webs-esw-demand-new-database-paradigms
60. Obasanjo, D.: An Exploration of Object Oriented Database Management Systems (2001), http://www.25hoursaday.com/WhyArentYouUsingAnOODBMS.html
61. Staken, K.: Introduction to Native XML Databases (2001), http://www.xml.com/pub/a/2001/10/31/nativexmldb.html
62. Wang, J.C., Huiling, G., Betsy, G.: Oracle White Paper? A Load-On-Demand Approach to Handling Large Networks in the Oracle Spatial Network Data Model (2009), http://www.oracle.com/technology/products/spatial/pdf/11gr2_collateral/_ndmlod11gr2_wp_1009.pdf
63. Apache Xindice, http://xml.apache.org/xindice
64. AllegroGraph RDF store, http://www.franz.com/agraph/allegrograph
65. Benchmarks: Performance advantages to store complex object structures, http://www.db4o.com/about/productinformation/benchmarks
66. db4o, http://www.db4o.com/about/productinformation/db4o

67. Facebook Statistics (2010), http://www.facebook.com/press/info.php?statistics
68. Getting Started with Berkeley DB for Java - Release 4.8, http://www.oracle.com/technology/documentation/berkeley-db/db/gsg/_JAVA/BerkeleyDB-Core-JAVA-GSG.pdf
69. H2 database, http://www.h2database.com
70. How ODB Works, http://wiki.neodatis.org/how-odb-works
71. Jena Semantic Web Framework, http://jena.sourceforge.net
72. JUNG Graph Framework, http://jung.sourceforge.net
73. Neo4j graph database, http://neo4j.org
74. Object-relational impedance mismatch, http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch
75. Oracle Berkeley DB, http://www.oracle.com/technology/products/berkeley-db/index.html
76. OWLIM Repository, http://www.ontotext.com/owlim
77. PolePosition Benchmark NeoDatis1.9, http://switch.dl.sourceforge.net/project/neodatis-odb/NeoDatis%20ODB%20Performance/NeoDatis%201.9/PolePosition_NeoDatis-1.9.pdf
78. Sesame Framework, http://www.openrdf.org
79. Tamino XML Server, http://www.softwareag.com/corporate/products/wm/tamino
80. Virtuoso Server platform, http://www.openlinksw.com/virtuoso