An introduction to Graph Data Management

Renzo Angles[1] and Claudio Gutierrez[2]

[1] Dept. of Computer Science, Universidad de Talca, Chile
rangles@utalca.cl
[2] Dept. of Computer Science, Universidad de Chile, Chile
cgutierr@dcc.uchile.cl

Abstract. A graph database is a database where the data structures for the schema and/or instances are modeled as a (labeled)(directed) graph or generalizations of it, and where querying is expressed by graph-oriented operations and type constructors.
In this lecture we present the basic notions of graph databases, give an historical overview of its main development, and study the main current systems that implement them.

Table of Contents

Introduction

It has been long recognized that graphs are a natural way to represent information and knowledge. In fact, graph database (abbreviated graph-db) models have a long development, at least since the 1980's. But it is only recently that several technological developments have made it possible to make this abstract idea a reality. Powerful hardware to store and process graphs; powerful sensors to record directly the information; powerful machines that allow to analyze and visualize graphs, among other factors, have given rise to the current flourishing in the area of graph data management.

Graph database models. In this lecture we will introduce the notion of graph-db model. As is well known, a data model can be characterized by three basic components, namely data structures, query and transformation language, and integrity constraints. Following this definition, a graph-db model is a model where data structures for the schema and/or instances are modeled as graphs (or generalizations of them), where the data manipulation is expressed by graph-oriented operations (i.e. a graph query language), and appropriate integrity constraints can be defined over the graph structure

The main characteristic of a graph database is that the data are conceptually modeled and presented to the user as a graph, that is the data structures (data and/or schema) are represented by graphs, or by data structures generalizing the notion of graph (e.g. hypergraphs or hypernodes). One of the main features of a graph structure is the simplicity to model unstructured data. Therefore, in graph-db models the separation between schema and data (instances) is less marked than in the classical relational model.

Regarding *data manipulation and querying*, it is is expressed by graph transformations, or by operations whose main primitives are based on graph features like paths, neighborhoods, subgraphs, graph patterns, connectivity, and graph statistics (diameter, centrality, etc.). Some db-models define a flexible collection of type constructors and operations, which are used to create and access the graph data structures. Another approach is to express all queries using a few powerful graph manipulation primitives. Usually the query language is what gives a database model its particular flavor. In fact, the differences among graph data structures are usually minors as compared to differences among graph query languages.

Finally, *integrity constraints* enforce data consistency. These constraints can be grouped in schema-instance consistency, identity and referential integrity, and functional and inclusion dependencies. Examples of these are labels with unique names, typing constraints on nodes functional dependencies, domain and range of properties, etc.

In this lecture we will concentrate in the data structure and language facets of graph database models.

Graph Database Systems There are two categories of graph database systems: graph databases and graph processing frameworks. The former are database systems, much like the relational ones, which aim at storing and querying graph data. The latter are frameworks that batch process big graphs, putting emphasis on throughput, hence taking advantage of multiple machines. These systems provide two perspectives for storing and querying graph data, each one with their own goals.

Contents and Organization of this lecture. In this lecture we present an overview of the basic notions, the historical evolution and the main current developments of this area. There are three main topics, distributed by sections. First, an overview of the field and its development, which we hope can be of help to look for ideas and past experiences. Second, we review the main graph database models in order to give a perspective on actual developments. Third, a similar review is done with graph database query languages. Finally, we present current database systems in a comparative manner.

Disclaimer. These are notes of an introductory course in the subject. Their goal is to serve as complementary material of the lecture. It is important to note that, on one hand, historical references are mentioned to help the student, rather than establish historical facts. On the other hand, the material is not novel and we relied heavily our own previous surveys on the subject, sometimes, copying literally whole paragraphs from them.

1 Motivation and Overview of the Field

In this section we present motivations for graph data management and briefly review the developments of it. There is an emphasis on models, because we think that it is important that the students know this material in order to build over current experiences and past background in the field.

What follows is taken almost literally from [60], and modified for better reading.

1.1 Why graph database models?

Why choose a graph-db model instead of a relational, object-oriented, or semi-structured one? Graph-db models are designed to deal with data in areas where the main concern of data management has to do with its interconnectivity or topology. In these applications, the data and the relations amongst the data are usually at the same level.

Graph-db present the following advantages over other types of models:

– Allow for a natural modeling of data when it has graph structure. Graph structures become visible to the user and they allow a natural way of handling applications data. Graphs have the advantage of being able to keep all the information about an entity in a single node and show related information by arcs connected to it. Graph objects (like paths, neighborhoods) may have first order citizenship.

– Queries can address direct and explicitly this graph structure. Associated with graphs are specific graph operations in the query language algebra, such as finding shortest paths, determining certain subgraphs, and so forth. Explicit graphs and graph operations allow users to express a query at a high level of abstraction. Another advantage is that it is not important to require full knowledge of the structure to express meaningful queries.
– Implementation-wise, graph databases may provide special graph storage structures, and efficient graph algorithms available for realizing specific graph operations over the data.

1.2 Comparison with other models

There are manifold approaches to model information and knowledge, depending on application areas and user needs. In this section, we review the most influential models and compare them to graph db-models.

The *Relational db-model* [83, 84] was introduced by Codd and is based on the simple notion of relation, which together with its associated algebra and logic, made the relational model a primary model for database research. In particular, its standard query and transformation language, SQL, became a paradigmatic language for querying. It popularized the concept of abstraction levels by introducing a separation between the physical and logical levels. Gradually the focus shifted to modeling data as seen by applications and users (that is, tables) [146]. The differences between graph db-models and the relational db-model are manifold. The relational model is geared towards simple record-type data, where the data structure is known in advance (airline reservations, accounting, inventories, etc.). The schema is fixed, which makes it difficult to extend these databases. It is not easy to integrate different schemas, nor is it automatable. The query language cannot explore the underlying graph of relationships among the data, such as paths, neighborhoods, patterns.

Semantic db-models [152] focus on the incorporation of richer and more expressive semantics into the database, from a user's viewpoint. Database designers can represent objects and their relations in a natural and clear manner (similar to the way users view an application) by using high-level abstraction concepts such as aggregation, classification and instantiation, sub- and super-classing, attribute inheritance and hierarchies [146]. In general, the extra semantics support database design and evolution [114]. A well-known and successful case is the entity-relationship model [78], which has become a basis for the early stages of database design. Semantic db-models are relevant to graph db-model research because the semantic db-models reason about the graph-like structure generated by the relationships between the modeled entities.

Object-oriented (O-O) db-models [122] are designed to address the weaknesses of the relational model in data intensive domains (Knowledge bases, engineering applications). This research is motivated by applications involving complex data objects and complex object interactions, such as CAD/CAM software, computer graphics and information retrieval. According to the O-O programming paradigm on which these models are based, they represent data as a collection of objects that are organized into classes, and have complex values and methods. O-O db-models are related to graph db-models in their explicit or implicit use of graph structures in definitions [131, 58, 108]. Nevertheless, there are important differences with respect to the approach for modeling how to model the world. O-O db-models view the world as a set of complex objects having certain state (data), where interaction is via method passing. On the other hand, graph db-models view the world as a network of relations, emphasizing data interconnection, and the properties of these relations. O-O db-models focus on object dynamics, their values and methods. Graph db-models focus instead on the interconnection, while maintaining the structural and semantic complexity of the data.

Semistructured db-models [72, 53] were motivated by the increased existence of semistructured data (also called unstructured data), data exchange, and data browsing mainly on the Web [72]. In semistructured data, the structure is irregular, implicit and partial; the schema does not restrict the data, it only describes it, a feature that allows extensible data exchanges; the schema is large and constantly evolving; the data is self-describing, as it contains schema information [53]. Among the most representative original models are OEM [149], Lorel [54], UnQL [73], ACeDB [168] and Strudel [99]. Many of these ideas can be seen in current semi-structured languages like XML or JSON. Generally, semistructured data is represented using a tree-like structure. However, cycles between data nodes are possible, which leads to graph-like structures like in graph db-models. Some authors characterize semistructured data as rooted directed connected graphs [73].

1.3 Historical overview

The origins of graph databases can be dated at least to the nineties, where much of the theory developed. Probably due to the lack of hardware support to manage big graphs, this line of research declined for a while until a few years ago, when a second wave of research was initiated.

The first wave. In an early approach, facing the failure of contemporary systems to take into account the semantics of a database, a semantic network to store data about the database was proposed [158] . An implicit structure of graphs for the data itself was presented in the Functional Data Model [166], whose goal was to provide a "conceptually natural" database interface. A different approach proposed the Logical Data Model (LDM) [126], where an explicit graph db-model intended to generalize the relational, hierarchical and network models. Later [125] proposed a graph db-model for representing complex structures of knowledge called G-Base.

In the late eighties an object-oriented db-model based on a graph structure, called O_2, was introduced [128]. Along the same lines, GOOD [108] is an influential graph-oriented object model, intended to be a theoretical basis for a system in which manipulation as well as representation are transparently graph-based. Among the subsequent developments based on GOOD are: GMOD [58] that proposes a number of concepts for graph-oriented database user interfaces; Gram [57] which is an explicit graph db-model for hypertext data; PaMaL [100] which extends GOOD with explicit representation of tuples and sets; GOAL [113] that introduces the notion of association nodes; G-Log [151] which proposed a declarative query language for graphs; and GDM [112] that incorporates representation of n-ary symmetric relationships.

There were proposals that used generalization of graphs with data modeling purposes. [130] The Hypermodel [128] (which we will develop in more detail) was a model based on nested graphs on which subsequent work was developed [153, 129]. The same idea was used for modeling multi-scaled networks [136] and genome data [102].

Another generalization of graphs, hypergraphs, gave rise to another family of models. GROOVY [131] is an object-oriented db-model based on hypergraphs. This generalization was used in other contexts: query and visualization in the Hy+ system [86]; modeling of data instances and access to them [175]; representation of user state and browsing [171];

There are several other proposals that deal with graph data models. Güting proposed GraphDB [155] intended for modeling and querying graphs in object-oriented databases and motivated by managing information in transport networks. Database Graph Views [106] proposed an abstraction mechanism to define and manipulate graphs stored in either relational object-oriented or file systems. The project GRAS [121] uses attributed graphs for modeling complex information from software engineering projects. The well known OEM [149] model aims at providing integrated access to heterogeneous information sources, focusing on information exchange.

Another important line of development has to do with data representation models and the World Wide Web. Among them are data exchange models like XML [69], metadata representation models like RDF [123] and ontology representation models like OWL [141].

The second wave. We are witnessing the second impulse of development of graph data management which is focused on one hand, in practical systems, and on the other, in theoretical analyses particularly of graph query languages. We will review the former in Section 4 concentrating in database systems and will leave the latter out of this lecture. The interested reader will find the Barcelo's tutorial [65] helpful for the subject.

2 Graph Database Models

All graph db-models have as their formal foundation variations on the basic mathematical definition of a graph, e.g., directed or undirected graphs, labeled or unlabeled edges and nodes, properties on nodes and edges, hypergraphs, hypernodes.

Representing the database as a simple flat graph has the drawback that it is difficult to modularize the information it represents. Hypernodes and hypergraphs address this problem by allowing a concept to grow from a simple

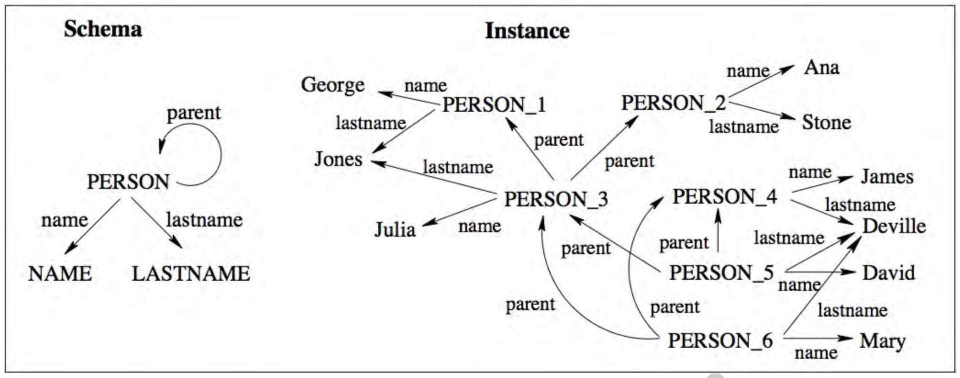

Fig. 1. Gram. At the schema level we use generalized names for definition of entities and relations. At the instance level, we create instance labels (e.g. PERSON_1) to represent entities, and use the edges (defined in the schema) to express relations between data and entities.

undefined concept to one defined by multiple complex relations. Note that, hypergraphs can be modeled by hypernodes by (i) encapsulating the contents of each undirected hyperedge within a further hypernode and (ii) replacing each directed hyperedge by two hypernodes related by a labeled edge. In contrast, multi-level nesting provided by hypernodes cannot be easily captured by hypergraphs.

Next, we will present these data structures and show a paradigmatic example of each. We will also present the most popular graph data structures today, RDF for the Web, and the property graph model for graph databases.

2.1 The Basic Graph Data Model

The most basic data structure for graph database models is a directed graph with nodes and edges labeled by some vocabulary. A good example is Gram [57], a graph db-model motivated by hypertext querying.

A schema in Gram is a directed labeled multigraph, where each node is labeled with a symbol called a *type*, which has associated a domain of values. In the same way, each edge has assigned a label representing a relation between types (see example in Figure 1). A feature of Gram is the use of regular expressions for explicit definition of paths called *walks*. An alternating sequence of nodes and edges represent a walk, which combined with other walks conforms other special objects called *hyperwalks*.

For querying the model (particularly path-like queries), an algebraic language based on regular expressions is proposed. For this purpose a hyperwalk algebra is defined, which presents unary operations (projection, selection, renaming) and binary operations (join, concatenation, set operations), all closed under the set of hyperwalks.

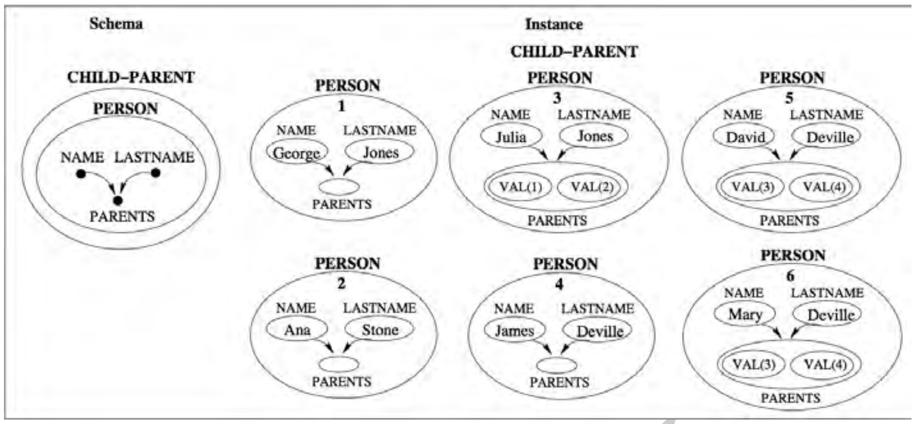

Fig. 2. GROOVY. At the schema level (left), we model an object *PERSON* as an hypergraph that relates the attributes *NAME*, *LASTNAME* and *PARENTS*. Note the value functional dependency (VDF) *NAME,LASTNAME* → *PARENTS* logically represented by the directed hyperedge ({NAME,LASTNAME} {PARENTS}). This VFD asserts that NAME and LASTNAME uniquely determine the set of PARENTS.

2.2 The Hypergraph Data Model

The notion of hypergraph is a generalization of graphs where the notion of edge is extended to *hyperedge*, which relates an arbitrary set of nodes [67]. Hypergraphs allow the definition of complex objects (using undirected hyperedges), functional dependencies (using directed hyperedges), object-ID and (multiple) structural inheritance.

A good representative case is GROOVY (Graphically Represented Object-Oriented data model with Values [131]), an object-oriented db-model which is formalized using hypergraphs. An example of hypergraph schema and instance is presented in Figure 2.

The model defines a set of structures for an object data model: value schemas, objects over value schemas, value functional dependencies, object schemas, objects over object schemas and class schemas. The model shows that these structures can be defined in terms of hypergraphs.

Groovy also includes a hypergraph manipulation language (HML) for querying and updating hypergraphs. It has two operators for querying hypergraphs by identifier or by value, and eight operators for manipulation (insertion and deletion) of hypergraphs and hyperedges.

2.3 The Hypernode Data Model (Nested Graphs)

A hypernode is a directed graph whose nodes can themselves be graphs (or hypernodes), allowing nesting of graphs. Hypernodes can be used to represent *simple* (flat) and *complex objects* (hierarchical, composite, and cyclic) as well

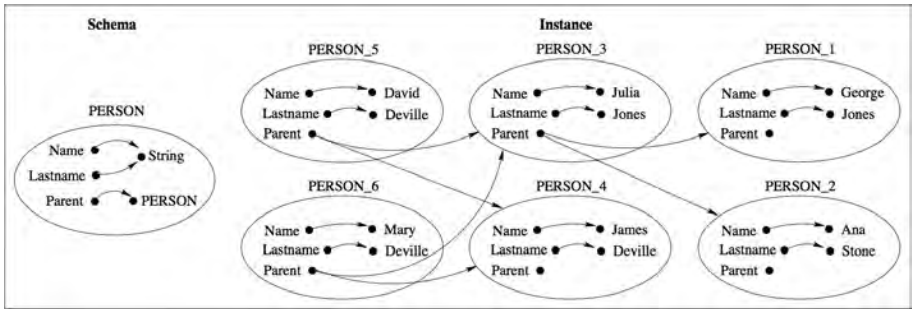

Fig. 3. Hypernode Model. The schema (left) defines a *person* as a complex object with the properties *name* and *lastname* of type string, and *parent* of type person (recursively defined). The instance (on the right) shows the relations in the genealogy among different instances of person.

as mappings and records. A key feature is its inherent ability to *encapsulate information*.

The hypernode model which we will use as example was introduced in [130]. It defines the model and a declarative logic-based language structured as a sequence of instructions (hypernode programs), used for querying and updating hypernodes. A more elaborated version [153] includes the notion of schema and type checking, introduced via the idea of types (primitive and complex), that are also represented by nested graphs (See an example in Figure 3). It also includes a rule-based query language called *Hyperlog*, which can support both querying and browsing with derivations as well as database updates, and is intractable in the general case. A third version of the model [129] discusses a set of constraints (entity, referential and semantic) over hypernode databases. In addition it presents another query and update language called HNQL, which use compounded statements to produce HNQL programs.

Summarizing, the main features of the Hypernode model are: a nested graph structure which is simple and formal; the ability to model arbitrary complex objects in a straightforward manner; underlying data structure of an object-oriented data model; enhancement of the usability of a complex objects database system via a graph-based user interface.

2.4 The RDF Data Model

Resource Description Framework (RDF) [123] is a recommendation of the W3C designed originally to represent metadata. One of the main advantages (features) of the RDF model is its ability to interconnect resources in an extensible way using graph-like structure for data.

An atomic RDF expression is a triple consisting of a subject (the resource being described), a predicate (the property) and an object (the property value). Each triple represents a statement of a relationship between the subject and the object. A general RDF expression is a set of such triples, which can be intuitively

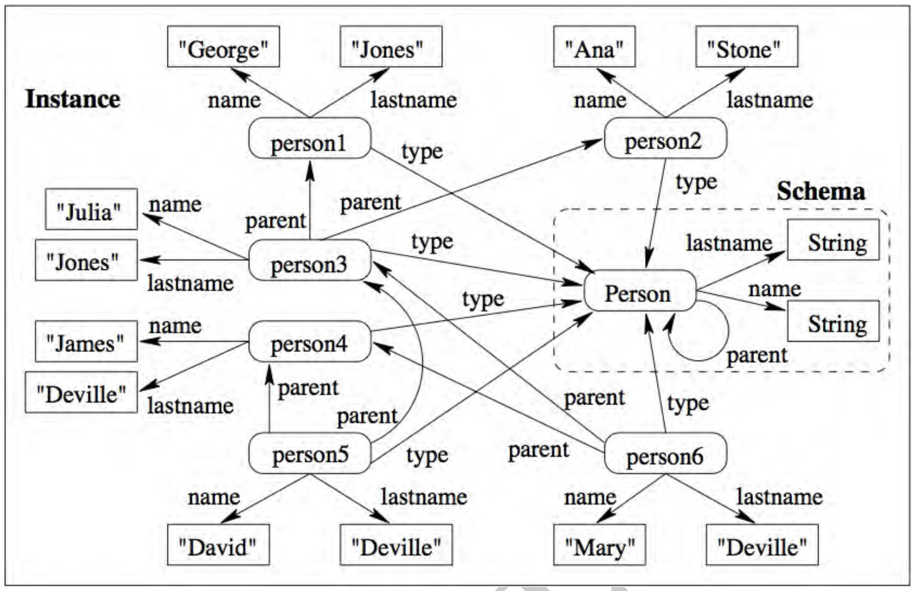

Fig. 4. RDF. Schema and instance are mixed together. In the example, the edges labeled *type* disconnect the instance from the schema. The instance is built by the subgraphs obtained by instantiating the nodes of the schema, and establishing the corresponding parent edges between these subgraphs.

considered as a labeled graph, called an RDF Graph (see example in Figure 4), although formally is not a graph [111].

SPARQL [154] is the standard query language for RDF proposed by the W3C. It is able to express complex graph patterns by means of a collection of triple patterns whose solutions can be combined and restricted by using several operators (i.e. AND, UNION, OPTIONAL, and FILTER). The latest version of the language, SPARQL 1.1 [98], includes explicit operators to express negation of graph patterns, arbitrary length path matching (i.e. reachability), aggregate operators (e.g. COUNT), subqueries, and query federation.

2.5 The Property Graph Data Model

A *property graph* is a directed, labelled, attributed multigraph. That is, a graph where the edges are directed, both nodes and edges are labeled and can have any number of properties (or attributes), and there can be multiple edges between any two vertices [156]. Properties are key/value pairs that represent metadata for nodes and edges. An example of property graph is shown in Figure 5.

The property graph model is very popular in current graph database systems, for example Neo4j [30], Sparksee/dex [39] and InfiniteGraph [27]. In practice, each vertex of a property graph has an identifier (unique within the graph) and zero or more labels. Node labels could be associated to node typing in order

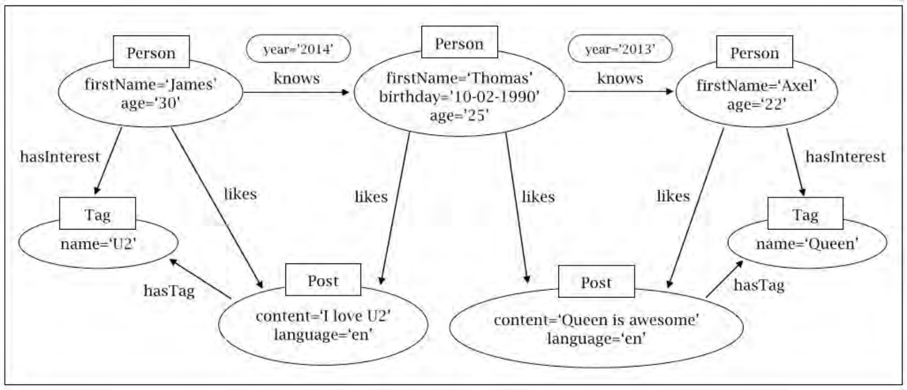

Fig. 5. Property graph model. In the example, nodes are typed (squares) and both nodes and edges contain properties (rounded squares).

to provide schema-based restrictions. Additionally, each (directed) edge has a unique identifier and one or more labels.

Property graphs are used extensively in computing as they are more expressive than the simplified mathematical objects studied in theory. However, note that expressiveness is defined by ease of use, not by the limits of what can be modeled [157]. In fact, the property graph model can express other types of graph models by simply abandoning or adding particular bits and pieces [156].

There is no standard query language for property graphs although some proposals are available. Blueprints [11] was one of the first libraries created for the property graph data model. Blueprints is analogous to the JDBC, but for graph databases. Gremlin [7] is a functional graph query language which allows to express complex graph traversals and mutation operations over property graphs. Neo4j [30] provides Cypher [15], a declarative query language for property graphs. The syntax of Cypher, very similar to SQL via expressions match-where-return, allows to easily express graph patterns and path queries. PQL [36] is a path query language for property graphs which was derived from Lorel. PQL is based on a mapping from path patterns to nested relational algebra extended with transitive closure.

3 Graph Database Query languages

Graph databases system address two main kinds of query workloads: (i) low-latency online graph query processing (e.g., social network transactions), and (ii) high-throughput offline graph analytics (e.g., PageRank computation). The former are the focus of graph databases whereas the latter are the speciality of graph processing frameworks.

In this section we will present examples of both types of graph queries. Additionally, we present examples of queries expressed in three different graph query languages.

3.1 Essential Graph Queries

In this section we present a set of essential queries that are supported by different graph query languages.

Pattern matching queries A graph pattern matching query consists of an expression representing a pattern (very much like a conjunctive query).

The basic block of such queries is the basic pattern. In SPARQL for example, a basic pattern is a triple that includes variables. The semantics of such a query is the set of all sub-graphs of the data graph that are isomorphic to the given pattern. For example, in SPARQL the pattern `{?x p ?y}` looks for the pairs `a b` such $a \xrightarrow{p} b$ is an edge in the data graph.

Pattern query languages combine basic patterns using algebraic structures (union, difference, filtering, etc.), given rise to complex and expressive expressions.

Pattern matching has attracted a great deal of attention in database theory [182, 165, 64, 79, 96, 94], data mining [174, 180, 181], bioinformatics [173], the semantic Web [76], social networks [61], and user-interfaces [58]. Today it is at the core of most graph query languages.

Adjacency queries This type of queries are special cases of the most elementary patterns. The primary notion in this type of queries is node/edge *adjacency*. Two nodes are adjacent (or neighbors) when there is an edge between them. Similarly, two edges are adjacent when they share a common node. Think of these queries as $a \xrightarrow{p} ?x$ or $a \xrightarrow{x?} b$, etc.

This class includes the following types of operations:

- basic node/edge adjacency [124]: tests whether two nodes (edges) are adjacent.
- k-neighborhood of a node [148]: for a node v, the k-neighborhood of v is the set K of all nodes that are reachable from v via a path of k edges (or "hops"). For instance, the k-neighborhood of v where $k = 1$ is the set of nodes containing v and immediate neighbors to v.
- k-hops[91]: returns all the nodes that are at a distance of k edges from the root node. Note that a k-neighborhood query can be expressed as a composition of k-hops queries (1-hops $\cup \cdots \cup$ k-hops), but removing duplicates.

Some application areas include spatial databases [160], molecular biology [162], information retrieval (for web ranking using hubs and authorities)[77], semantic Web [104], and recommendation systems (to obtain a particular user's neighborhood with similar interest) [91].

Reachability queries (connectivity) These types of queries are characterized by path or traversal problems. The problem of *reachability* tests whether two given nodes are connected by a path. A reachability query may involve the generation of a boolean result, a set of nodes, a single possible solution path, or a set of possible paths. From the literature we identify the following reachability queries:

- Single-source, single-sink problem (or reachability, or graph accessibility problem – GAP) [182]: it is a decision problem that looks for a path path between two nodes. Given a source node s and a sink node t, we want to decide whether there is a path from s to t.
- All-pairs transitive closure [182]: to compute the set of all pairs of nodes (u, v) such that u can reach v, i.e., there is a path form node u to node v.
- Single-source transitive closure [182, 169]: to find the nodes that are reachable from an specific source node u. This problem is solved via two main traversal strategies: depth-first search (DFS) or breadth-first search (BFS).
- Dual single-sink problem [182]: to find all the nodes that can reach a given sink node v.
- Fixed-length paths: find a path which contain a fixed number of nodes and edges.
- Simple paths [142]: it implies that the path does not use any node of edge more than once.
- Regular path queries [142, 55, 133, 90, 70, 169, 95, 132]: find a path which allow some node and edge restrictions (i.e., regular expressions).
- Conjunctive regular path queries [177, 74, 63]: a query that combines conjunctive queries and regular path queries.
- Shortest path [62]: compute the quickest/shortest route between two nodes. Works studying this problem [62]

Applications. Reachability queries are studied and required in several application domains. Among them we can mention: Spatial databases [150], biological databases [139], and the semantic Web [103].

Summarization queries These types of queries do not consult the graph structure; instead they are based on special operators that allow to summarize the query results, normally returning a single value. Aggregate functions (e.g., average, Count, maximum, etc.) are included in this group. Summarization queries can be used to answer the following queries:

- the order of the graph (i.e., the number of vertices)
- the degree of a node (i.e., the number of neighbors of the node),
- the minimum, maximum and average degree in the graph,
- the length of a path (i.e., the number of edges in the path)
- the distance between nodes (i.e., the length of a shortest path between the nodes),

3.2 Graph Analytical Queries

There are several important algorithms for graph analysis and mining (see [56] for a extensive review). However, there exists no standard categorization of such algorithms. Next, we briefly review some categories and graph analytical algorithms described in articles that compare graph processing frameworks.

In 2010, the authors of [91] presented a very complete categorization by dividing graph operations into: traversals (shortest path and k-hops), graph analysis (hop-plot, diameter, eccentricity, density and clustering coefficient), components (connected components, bridges and cohesion), communities (dendrogram, max-flow min-cut and clustering), centrality measures (degree centrality, closeness centrality and betweenness centrality), pattern matching (graph and subgraph matching), graph anonymization (k-degree and k-neighborhood anonymization) and other operations (PageRank and structural equivalence).

For an empirical evaluation of graph-processing algorithm [105], the graph processing algorithms were categorized into five groups: general statistics, graph traversal, connected components, community detection, and graph evolution. The *general statistics* (STATS) [105] algorithm computes the number of vertices and edges, and the average of the local clustering coefficient of all vertices. *Graph traversals* are characterized by their random memory access pattern and data driven computation [81]. Breadth-first search (BFS) is a widely used algorithm in graph traversals, which is often a building block for more complex algorithms, such as item search, distance calculation, diameter calculation, shortest path and longest path. *Connected component* (CONN) is an algorithm for extracting groups of vertices that can reach each other via graph edges. *Community detection* (CD) is important for social network applications, as users of these networks tends to form communities, that is, groups whose constituent nodes form more relationships within the group than with nodes outside the group. For *graph evolution* (EVO) [105], an accurate algorithm not only can predict how a graph structure will evolve over time, but can also help to prepare for these changes (for example data size increase).

In a comparison study [184] of parallel processing systems, the authors considered three graph algorithms commonly used in network analysis studies, namely PageRank, Shortest Path and Triangle Counting. *PageRank* and its variants such as Personalized PageRank are effective methods for link prediction based on finding structure similarities between nodes in a network. The *clustering coefficient* of a vertex expresses the chance of how likely its neighbors are also connected to one another. The (global) clustering coefficient is based on triplets of nodes, where a triplet consists of three nodes that are connected by either two (open) or three (closed) undirected ties. The global clustering coefficient is therefore defined as the fraction of the number of closed triplets over number of total connected triplets of vertices. Therefore, the method of calculating the global clustering coefficient is also often called triangle counting [176]. The characteristic path length is the average *shortest path* length in a network [176], which measures the average degree of separation between nodes in a network (or network component). The clustering coefficient and the characteristic path

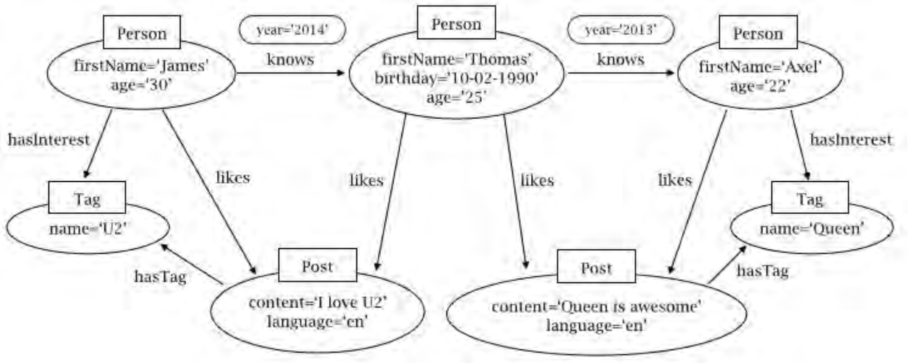

Fig. 6. A sample property graph database which contains social network data.

length are important network measures that are often used to determine the type of a network (e.g., random, small-world and scale-free).

In a recent experimental comparison of Pregel-like graph processing systems, the authors [109] considered four categories of graph algorithms: random walk, sequential traversal, parallel traversal, and graph mutation. *Random walk* algorithms perform computations on all vertices based on the random walk model. Examples of algorithms in this category are PageRank and HITS. The category of *sequential traversal* includes algorithms like single-source shortest path, breadth-first search, depth-first search and reachability. Algorithms like weakly connected components, label propagation and graph clustering are in the group of *parallel traversal*. Finally, the category of *graph mutation* contains algorithms for computing the minimum spanning tree, graph coarsening and graph aggregation.

3.3 Graph Query Languages in Practice

In this section, we will present a brief description of three graph query languages (SPARQL, G-SPARQL and Cypher) by presenting their syntax and main features. Additionally, we compare the expressivity of the three languages by presenting several types of queries in natural language, and showing how they are expressed (if possible) in each query language.

The queries discussed in this section will be based on a graph dataset similar to the property graph showed in Figure 6. The sample property graph uses nodes to represent Persons, Tags and Posts, which are common entities in the social network use-case. Two persons can be related by using the relationship (edge) “knows”. Persons could be related with Tags and Posts via relationships “hasInterest” and “like” respectively. A Post can be associated with a Tag via the “hasTag” relation. Persons, Tags and Posts can contain different properties. The edges of type “knows” include the property “year” to register the year when the relationship was created.

Pattern Matching Queries A pattern matching query is based on the definition of a graph pattern and the objective is to find subgraphs (in the database graph) satisfying the graph pattern. We consider several types of pattern matching queries depending on the complexity of the graph pattern.

– *Single node graph patterns* This type of query looks for nodes having a given attribute or a condition over an attribute.

Example: return the persons whose attribute first name is "James".

```
## SPARQL 1.0 and SPARQL 1.1
SELECT ?X
FROM <http://www.socialnetwork.org>
WHERE { ?X sn:firstName "James" }

## G-SPARQL
SELECT ?X
WHERE { ?X @firstName "James" }

## CYPHER
MATCH (person:Person)
WHERE person.firstName="James"
RETURN person
```

– *Single graph patterns* A single graph pattern consists of a single structure node-edge-node where variables are allowed in any part of the structure. A single graph pattern is oriented to evaluate adjacency between nodes.

Example: return the pairs of persons related by the "knows" relationship.

```
## SPARQL 1.0 and SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?X ?Y
WHERE { ?X sn:knows ?Y }

## G-SPARQL
SELECT ?X ?Y
WHERE { ?X knows ?Y }

## CYPHER
MATCH (person1:Person)-[:knows]->(person2:Person)
RETURN person1, person2
```

– **Complex graph patterns** A complex graph pattern is a collection of single graph patterns connected by special operators, usually join, union, difference and negation. In the literature of graph query languages (see for example GraphLog [87]), a complex graph pattern is graphically represented as a graph containing multiple nodes, edges and variables, and special conditions can be defined over all of them (e.g value-based conditions over nodes, negation of edges, summarization, etc.). The evaluation of graph patterns is usually defined in terms of subgraph isomorphism [88, 107].

Example (Join of graph patterns): return the first name of persons having a friend named "Thomas" .

```
## SPARQL 1.0 and SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?N
WHERE { ?X sn:type sn:Person . ?X sn:firstName ?N .
        ?X sn:knows ?Y . ?Y sn:firstName "Thomas"  }

## G-SPARQL
SELECT ?N
WHERE { ?X @type "Person" . ?X @firstName ?N .
        ?X knows ?Y . ?Y @firstName "Thomas"  }

## CYPHER
MATCH (person:Person)-[:knows]-(thomas:Person)
WHERE thomas.firstName="Thomas"
RETURN person.firstName
```

Example (Union of graph patterns): return the persons interested in either "Queen" or "U2".

```
## SPARQL 1.0 and SPARQL 1.1
## This query introduces duplicates which are eliminated
## by the DISTINCT operator
PREFIX sn: <http://www.socialnetwork.org/>
SELECT DISTINCT ?X
WHERE { ?X sn:type sn:Person . ?X sn:hasInterest ?T . ?T sn:type sn:Tag .
        { { ?T sn:name "Queen"} UNION { ?T sn:name "U2" } } }

## SPARQL 1.0 and SPARQL 1.1
## This query avoids duplicates by using a FILTER condition
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?X
WHERE { ?X sn:type sn:Person . ?X sn:hasInterest ?T . ?T sn:type sn:Tag .
        ?T sn:name ?N . FILTER ( ?N = "Queen" || ?N = "U2" ) }

## G-SPARQL
SELECT ?X
WHERE { ?X @type "Person" . ?X hasInterest ?T . ?T @type "Tag" .
        ?T @name ?N . FILTER ( ?N = "Queen" || ?N = "U2" ) }

## CYPHER
MATCH (person:Person)-[:hasInterest]->(tag:Tag)
WHERE tag.name="Queen" OR tag.name="U2"
RETURN DISTINCT person
```

Example (Difference/negation of graph patterns): return the first name of persons which do not like any post.

```
## SPARQL 1.0
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?N
WHERE {
 ?X sn:type sn:Person .
 { ?X sn:firstName ?N . OPTIONAL { ?X sn:likes ?P } }
 FILTER (!bound(?P))
}

## SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?N
WHERE {
 { ?X sn:type sn:Person . ?X sn:firstName ?N }
 MINUS
 { ?X sn:likes ?P }
}

## G-SPARQL
SELECT ?N
WHERE {
 ?X @type "Person" .
 { ?X @firstName ?N . OPTIONAL { ?X likes ?P } }
 FILTER (!bound(?P))
}

## CYPHER
MATCH (person:Person)
WHERE NOT((person)-[:likes]->(:Post))
RETURN person.firstName
```

Filter conditions in graph patterns A graph pattern can be extended to allow filter boolean restrictions over node and edge labels.

Example: find the persons whose age is between 18 and 30.

```
## SPARQL 1.0 and SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?X
WHERE {
 ?X sn:type sn:Person . ?X sn:age ?A . FILTER (?A > 18 && ?A < 30)
 }

## G-SPARQL
SELECT ?X
WHERE {
 ?X @type "Person" . ?X @age ?A . FILTER (?A > 18 && ?A < 30)
}

## CYPHER
MATCH (person:Person)
WHERE person.age>18 and person.age<30
RETURN person
```

Fixed-length path queries A *fixed-length path query* is a special type of graph pattern which represents a traversal from a source node to a target node, by including a fixed number of nodes and edges.

Example: Find the names of people at distance 2 from "James" by following "knows" links.

```
## SPARQL 1.0 and SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?N
WHERE { ?X sn:type sn:Person . ?X sn:firstName "James" .
        ?X sn:knows ?Z . ?Z sn:knows ?Y . ?Y sn:firstName ?N .
        FILTER (!(?Y = ?X || ?Y = ?Z)) }

## G-SPARQL
SELECT ?N
WHERE { ?X @type "Person" . ?X @firstName "James" .
        ?X knows ?Z . ?Z knows ?Y . ?Y @firstName ?N .
        FILTER (!(?Y = ?X || ?Y = ?Z)) }

## CYPHER
MATCH
 (james:Person{firstName:"James"})-[:knows*2..2]-(person:Person)
WHERE
 NOT(person=james) AND NOT((person)-[:knows]-(james))
RETURN
 DISTINCT person.firstName
```

Reachability queries Reachability queries are characterized by path or traversal problems, and the objective is to test whether two given nodes are connected by a path. Reachability queries are usually expressed by using regular path queries.

Regular path queries A regular path query is usually represented as a single graph pattern (N_s E N_t) where N_s is the source node (value or variable), N_t is the target node (value or variable), and E is a regular expression representing the path pattern (see property paths in the description of SPARQL 1.1). A reachability query may involve the generation of a boolean result, a set of nodes, a single possible solution path, or a set of possible paths. Regular path queries are not supported in SPARQL 1.0.

Example: find the first name of people that can be reached from "James" by relation "knows".

```
## SPARQL 1.1
 PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?N
WHERE { ?X sn:type sn:Person . ?X sn:firstName "James" .
        ?X sn:knows* ?Y .
        ?Y sn:firstName ?N }
```

```
## G-SPARQL
SELECT ?N
WHERE { ?X @type "Person" . ?X @firstName "James" .
        ?X knows* ?Y .
        ?Y @firstName ?N }

## CYPHER
MATCH (james:Person)-[:knows*]-(reachablePerson:Person)
WHERE james.firstName="James"
RETURN DISTINCT reachablePerson
```

Consider reachability queries where the computed paths are required to be returned as a sequence of nodes and edges. Only G-SPARQL and Cypher are able to answer this type of queries. SPARQL 1.1 is restricted to return the source node and the target node of the path.

Example: find the people that can be reached from "James" by following the relation "knows", and return the corresponding path.

```
## G-SPARQL
SELECT ?N ??P
WHERE { ?X @type "Person" . ?X @firstName "James" .
        ?X ?PP(knows*) ?Y .
        ?Y @firstName ?N }

## CYPHER
MATCH path = (james:Person)-[:knows*]-(reachablePerson:Person)
WHERE james.firstName="James"
RETURN DISTINCT reachablePerson, path
```

Note that these queries will return one path per "reachable" person but not the shortest path.

Regular path queries with path-length restrictions Some languages allow to restrict the length of the paths returned by a regular path query.

G-SPARQL allows expressions of the form `Length(??P, <3)` which allows to filter the path variable `??P` with the length condition `<3`. In Cypher, path-length restrictions can be defined in the recursive relation, for example `[:KNOWS*1..3]`. SPARQL 1.1 is not able to express path-length restrictions.

Example: return the paths of length 5-10, along the "knows" relation, that connect 'James' and 'Axel'.

```
## G-SPARQL
SELECT ??P
WHERE { ?X @type "Person" . ?X @firstName "James" .
        ?X ??P(knows*) ?Y . ?Y @firstName "Axel" .
        FILTERPATH(Length(??P, >=5)) . FILTERPATH(Length(??P, <=10)) }

## CYPHER
MATCH path = (james:Person)-[:KNOWS*5..10]-(axel:Person)
WHERE james.firstName="James" AND axel.firstName="Axel"
RETURN path
```

Regular path queries with value-based restrictions This type of queries implies the introduction of value-based restriction over the paths returned by a regular path query, i.e. value conditions over the attributes of nodes and edges belonging to the resulting paths.

SPARQL 1.1 does not support this kind of restrictions. G-SPARQL is characterized by allowing valued-based restrictions over specific nodes and edges (i.e. AtLeastNode, AtMostNode, AllNodes, AtLeastEdge, AtMostEdge, AllEdges). In the case of Cypher, this type of restrictions can be defined in the `WHERE` clause.

Example: find the first name of people that can be reached from "James" by following relations "knows" created during the year 2012 (assume that each relation "knows" contains an attribute "year" of creation).

```
## G-SPARQL
SELECT ?N
WHERE { ?X sn:firstName "James" .
        ?X ??P(sn:knows+) ?Y .
        ?Y sn:firstName ?N .
        FILTERPATH(AllEdges(??P, @year "2012")) }

## CYPHER
MATCH (james:Person)-[r:KNOWS]-(other:Person)
WHERE james.firstName="James" AND r.year=2012
RETURN other.firstName
```

Regular path queries with structural restrictions This type of queries implies the introduction of structural restrictions over the paths returned by a regular path query, i.e. conditions over the edges of the nodes in the resulting paths.

Only Cypher is able to provide this kind of restrictions. They can be defined in the clause `WHERE` by using the collection function `nodes(path)` which returns the nodes occurring in the resulting paths associate to the given `path`.

Example: find the first name of people that can be reached from "James" by following relations "knows" and satisfying that each people in the sequence also knows "James").

```
## CYPHER
MATCH path = (james:Person)-[:KNOWS*]-(other:Person)
WHERE filter(node IN nodes(path) WHERE (node)-[:KNOWS]-(james))
RETURN other.firstName
```

Shortest path queries A shortest path query means to compute the quickest/shortest route between two nodes in the graph. Most languages provide ad-hoc functions to calculate shortest paths queries. In some cases the shortest path can be calculated by combining a reachability query with aggregate operators (e.g. COUNT + MIN), although it requires that the reachability query results in a set of paths.

SPARQL 1.1. is not able to calculate shortest path queries. G-SPARQL and Cypher provide special predicates to return the shortest path.

Example: Return the shortest path between "James" and "Axel" by following the relation "knows".

```
## G-SPARQL
SELECT ?*P
WHERE { ?X @firstName "James" .
        ?X ?*P(knows+) ?Y .
        ?Y @firstName "Axel"}

## CYPHER
MATCH (jame:Person{firstName:"James"}),(axel:Person{firstName:"Axel"})
MATCH path=shortestPath((james)-[:knows*]-(axel))
RETURN path
```

Aggregate queries and grouping Aggregate queries are based on special operators, non related to the data model, that permit to summarize or operate on the query results. Common aggregate operators include: COUNT, SUM, AVG, MIN, and MAX. Aggregate operators are very useful to calculate special information about nodes and edges in the graph, for example the degree of a node (i.e. by counting the neighbors of the node) or the length of the shortest path between two nodes.

SPARQL 1.0 and G-SPARQL do not support aggregate operators. SPARQL 1.1 and Cypher defines all the common aggregate operators. Additionally, Cypher includes special aggregate operators for paths, for example `length(path)` allows to obtain the length of the given `path`.

Example: Return the number of friends of "James"

```
## SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT (COUNT(?Y) AS ?Friends)
WHERE { ?X sn:firstName "James" . ?X sn:knows ?Y }

## CYPHER
MATCH (james:Person)-[:FRIENDS]->(friend:Person)
WHERE james.firstName='James'
RETURN count(friend)
```

Example: Return the length of the shortest path between "James" and "Axel".

```
## CYPHER
MATCH (jame:Person{firstName:"James"}),(axel:Person{firstName:"Axel"})
MATCH path=shortestPath((james)-[:knows*]-(axel))
RETURN length(path)
```

Grouping. The result sequence of an aggregate query can be grouped by values of different attributes or relationships. This is the function of the GROUP BY operator in SQL.

Operators for grouping are defined in SPARQL 1.1 and Cypher. G-SPARQL does not support grouping.

Example: Return the number of friends of each person

```
## SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?X, (COUNT(?Y) AS ?Friends)
WHERE { ?X sn:type sn:Person . ?X sn:knows ?Y }
GROUP BY X?

## CYPHER
MATCH (person:Person)
RETURN person, length((person)-[:knows]-())
```

Group conditions. Some query languages allows to filter the groups according to a given condition. This is the function of the HAVING operator in SQL.

Example: for each person having 100 friends or more, returns their first name and friends' number.

```
## SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT ?N, (COUNT(?F) AS ?FriendsNumber)
WHERE { ?X sn:type sn:Person . ?X sn:firstName ?N . ?X sn:knows ?F }
GROUP BY ?X
HAVING (COUNT(?F) >= 100)

## CYPHER
MATCH (person:Person)
WITH person, length((person)-[:knows]-()) AS friendsNumber
WHERE friendsNumber>=100
RETURN person.firstName, friendsNumber
```

Restrictions over result sequences The result sequence of a query can be restricted by using the following operators:

- DISTINCT: to return only distinct (different) values.
- ORDER BY: to sort (ascending or descending) the result sequence by a node, edge or property.
- OFFSET: to define where the solutions start from in the sequence of solutions (i.e. a given position in the sequence).
- LIMIT: to restrict the number of solutions to a given number.

Most of the above operators are supported by SPARQL 1.1 and Cypher.

Example: Return the youngest top-5 distinct friends of "James".

```
## SPARQL 1.1
PREFIX sn: <http://www.socialnetwork.org/>
SELECT DISTINCT ?F ?N
WHERE { ?X sn:type sn:Person . ?X sn:firstName "James" .
        ?X sn:knows ?F . ?F sn:birthday ?B }
ORDER BY ASC(?B)
LIMIT 5
```

```
## CYPHER
MATCH (james:Person)-[:knows]-(friend:Person)
WHERE james.name="James
RETURN friend
ORDER BY friend.birthday ASC
LIMIT 5
```

4 Graph Database Management Systems

The systems for graph data management can be classified in two main categories, graph databases and graph processing frameworks. Although the problems addressed for both groups are similar, they provide two different approaches for storing and querying graph data, with their own advantages and disadvantages.

Graph databases aim at persistent management of graph data, allowing to transactionally store and access graph data on a persistent medium. In this sense, these provide efficient single-node solutions with limited scalability. On the other hand, graph processing frameworks aim to provide batch processing and analysis of large graphs often in a distributed environment with multiple machines. These solutions usually process the graph in memory, but different parts of the graph are managed by distinct, distributed nodes.

Very related to graph databases are the systems for managing RDF data. These systems, called RDF Triple Stores or RDF databases, are specifically designed to store collections of RDF triples, to support the standard SPARQL query language, and possibly to allow some kind of inference via semantic rules. Although Triple Stores are based on the RDF graph data model, they are specialized databases with their own characteristics. Therefore, we will study them separately.

Next we present a review of current systems in the above categories, including a short description of each of them.

4.1 Graph databases

A *graph database* is a system designed specifically to store and query graph data. The design of any graph database is based on a graph data model which determines the data structures, query operations and integrity constraints used by the entire system. Currently, most graph databases implement the property graph data model [156].

Considering their level of maturity, we assume that a graph database must provide most of the major components in database management systems, being them: external interfaces (user interface or API), database languages (for data definition, manipulation and querying), query optimizer, database engine (middle-level model), storage engine (low-level model), transaction engine, management and operation features (tuning, backup, recovery, etc.).

Considering their internal implementation, we classify graph databases in two types: native and non-native graph databases. Native graph databases implement ad-hoc data structures and indexes for storing and querying graphs. Non-native graph databases make use of other database systems to store graph data and implement query interfaces to execute graph queries over the back-end system.

In the group of native graph databases we found AllegroGraph [3], Bitsy [9], Cayley [13], GraphBase [22], Graphd [23, 143], HyperGraphDB [25, 115], IBM System G [26, 75, 178], imGraph [116], InfiniteGraph [27], InfoGrid [28], Neo4j [30], Sparksee/DEX [39], Trinity [164, 44]) and TurboGraph [110, 45]

In the group of non-native graph databases we consider: Titan [43], which supports Apache Cassandra, Apache HBase and Oracle BerkeleyDB as storage backends; FlockDB [18] is a distributed graph-oriented database which uses MySQL as the storage engine; OrientDB [34] and ArangoDB [8], which are document-store databases adapted to graphs; OQGRAPH [32] which is a graph computation engine for MySQL, MariaDB y Drizzle; VelocityGraph [46] an object database supporting graphs; Horton [50, 161], which is based on the cloud programming infrastructure Orleans; and Oracle extensions for managing spatial and graph data [33].

There are several papers comparing the features [92, 59, 71, 170, 140] and performance [172, 81, 66, 117] of graph databases. In Table 1, we present a general view of the main features of current graph databases, including data model, storage model and query facilities. Next, we briefly describe the systems we consider more relevant.

AllegroGraph[3] is one of the precursors in the current generation of graph databases. Although it was born as a graph database, its current development is oriented to meet the Semantic Web standards (i.e., RDF/S, SPARQL and OWL). Additionally, AllegroGraph provides special features for GeoTemporal Reasoning and Social Network Analysis.

Sparksee (formely DEX)[138, 16] is a native graph database for persistent storage of property graphs. Its implementation is based on bitmaps and other secondary structures, and provides libraries (APIs) in several languages for implementing graph queries. Sparksee is been used in social, bibliographical and biological networks analysis, media analysis, fraud detection and business intelligence applications of indoor positioning systems

HyperGraphDB [25, 115] is a systems that implements the hypergraph data model (i.e. edges are extended to connect more than two nodes). This model allows a natural representation of higher-order relations, and is particularly useful for modeling data of areas like knowledge representation, artificial intelligence and bio-informatics. Hypergraph stores the graph information in the form of key-value pairs which are stored on BerkeleyDB.

InfiniteGraph [27] is a database oriented to support large-scale graphs in a distributed environment. It aims the efficient traversal of relations across massive and distributed data stores. Its focus of attention is to extend business, social and government intelligence with graph analysis.

Neo4j [30] is based on a network oriented model where relations are first class objects. It is fully written in java and implements an object-oriented API,

Table 1. Short review of graph databases. The systems are sorted alphabetically.

	Data model				Storage method		Query facilities			Computing model	
Graph database	Simple graph	Property graph	Hypergraph	Nested graph	Native	Non-native	Query language	API	Graph algorithms	Single-node	Distributed
AllegroGraph	●				●		●	●	●	●	
ArangoDB		●			●	●	●	●		●	
Bitsy		●			●			●	●	●	
Cayley		●			●	●	●			●	●
FlockDB	●					●		●			●
GraphBase	●				●		●	●		●	
Graphd	●				●		●			●	
Horton		●				●	●				●
HyperGraphDB			●		●	●	●	●		●	●
IBM System G		●			●	●		●		●	●
imGraph	●				●			●			●
InfiniteGraph		●			●		●	●	●	●	●
InfoGrid	●				●	●		●		●	●
Neo4j		●			●		●	●	●	●	●
OrientDB		●				●	●	●	●		●
Sparksee/DEX		●			●			●	●	●	
Titan		●			●	●	●	●		●	●
Trinity		●	●			●	●	●	●		●
TurboGraph		●			●			●	●	●	

a native disk-based storage manager for graphs, and a framework for graph traversals.

Trinity [164, 44]) implements a general purpose graph engine over a distributed memory cloud. Trinity implements a globally addressable distributed memory storage, and provides a random access abstraction for large graph computation. Hence, it supports both online graph query processing and offline graph analytics. Its query languages, called TSL, allows users to declare data schema and communication protocols.

Additionally, we can find multiple tools related to graph databases, including small systems (G-Store [19],redis_graph [37], vertexdb [47]), in-development databases (CloudGraph [14], Orly [35], StigDB [41], JCoreDB Graph [29], Weaver [49]), graph libraries (Filament [17]), academic developments (SGDB [82], SylvaDB [?]) in-memory graph analysis tools (e.g., Cytoscape) and graph visualization tools (e.g., JUNG, IGraph, GraphViz, Gephi and NodeXL).

Oriented to support the development of graph applications, TinkerPop [7] provides several tools for graph databases. Blueprints provides a common API for the property graph data model; Pipes provides a data flow framework to join Blueprints with specific graph databases; Frames exposes the elements of Blueprints graphs as Java objects (i.e., implements an object-graph mapping) Gremlin is a domain specific language designed for traversing graphs. Rexster allows to expose any Blueprints graph by implementing a Restful interface.

4.2 RDF Database Systems

An *RDF database* (also called *Triple Store*) is a specialized graph database for managing RDF data. RDF defines a data model based on expressions of the form subject-predicate-object (SPO) called RDF triples. Therefore, an RDF dataset is composed by a large collection of RDF triples which implicitly form a graph.

SPARQL is the standard query language for RDF databases. It is a declarative language which allows to express several types of graph patterns. Its most recent version (SPARQL 1.1) supports advanced features like property paths, aggregate functions and subqueries.

There are several works comparing RDF databases [144, 167, 97, 89]. Similar to graph databases, RDF databases can also be classified into native and non-native RDF databases. Examples of native RDF databases are Jena [85], RDF-3X [147], 4store [2] and TripleBit [183]. Among the non-native RDF databases we can mention to OpenLink Virtuoso [48], Sesame [38] and DB2RDF [68], which are implemented on top of relational database systems.

Being more specific about native storage approaches, the RDF databases can be classified into four categories [183]: triples table, property table, column store with vertical partitioning and RDF graph based store.

A *triple table* refers to the approach of storing RDF data in a 3-column table with each row representing a SPO statement. Hence, the evaluation of SPARQL queries involve self-joins over this long table. A popular approach to improving performance of queries in this storage model is to use an exhaustive indexing method that creates a full set of SPO permutations of indexes. Among the systems implementing triple tables we can mention RDF-3X [147], Sesame [38], 3store [1], BrightstarDB [12].

A second approach is to store RDF data in a *property table* [52] with subject as the first column and the list of distinct predicates as the remaining columns. A single property table can be extremely sparse and contains many NULL values. Thus multiple-property tables with different clusters of properties are proposed as an optimization technique. Jena [85], Oracle [80], and BitMat are examples of systems implementing property tables.

RDF data can also be stored by using multiple two-column tables, one for each unique predicate. The first column is for subject whereas the other column is for object. This method, called *column store with vertical partitioning* [52], can be implemented over row-oriented or column-oriented database systems. This column-store approach is implemented by C-Store

Finally, a graph based approach focuses on storing RDF data as a graph. In this case, the RDF triples must be modeled as classical graph nodes and edges, and the SPARQL queries must be transformed into graph queries. Among the *RDF graph based stores* we can mention Ontotext GraphDB [31], gStore [185], Stardog [40], Blazegraph [10], TrinityRDF [51] and GEMS [145].

4.3 Graph Processing Frameworks

In addition to graph databases, a number of graph processing frameworks have been proposed to address the needs of processing complex and large-scale graph datasets. These frameworks are characterized by in-memory batch processing and the use of distributed and parallel processing strategies. Note that, distributed systems with more computing and memory resources are able to process large-scale graphs, but they can be less efficient than single-node platforms when specific graph queries are executed.

On the one hand, generic data processing systems such as Hadoop [5], YARN [6], Stratosphere [42] and Pegasus [118] have been adapted for graph processing due to their facilities for batch data processing. Most of these systems are based on the MapReduce programming model and implemented on top of the Hadoop platform, the open source version of MapReduce. By exploiting data-parallelism, these systems are highly scalable and support a range of fault-tolerance strategies. Though these systems improve the performance of iterative queries, users still need to "think" their analytical graph queries as MapReduce jobs. In fact, naively expressing graph computation and graph algorithms in these data-parallel abstractions can be challenging [179]. Additionally, these systems cannot take advantage of the characteristics of graph-structure data and often result in complex job chains and excessive data movement when implementing iterative graph algorithms [184].

On the other hand, graph-specific platforms such as Pregel [137], Apache Giraph [4], GraphLab [135, 134], Apache Hama, Catch de Wind [163], GPS [159], Mizan [120], PowerGraph [101], GraphX [24], TurboGraph [110] and GraphChi [127] provide different programming interfaces for expressing graph analytic algorithms. These platforms, also called *offline graph analytic systems*, perform an iterative, batch processing over the entire graph dataset until the computation satisfies a fixed-point or stopping criterion. Therefore, these systems are particularly designed for computing graph algorithms which require iterative, batch processing, e.g., PageRank, recursive relational queries, clustering, social network analysis, machine learning and data mining algorithms [119]. Next we briefly describe some of these systems.

Pregel [137] is a system that provides a native API specifically designed by Google for writing algorithms that process graph data. Pregel is a vertex-centric programming abstraction that adapts the Bulk Synchronous Parallel (BSP) model, which was developed to address the problem of parallelizing jobs across multiple workers for scalability The fundamental computing paradigm Pregel employs can be characterized as "think like a vertex". Graph computations are specified in terms of what each vertex has to compute; edges are communication channels for transmitting computation results from one vertex to another, and do not participate in the computation. To avoid communication overheads, Pregel preserves data locality by ensuring computation is performed on locally stored data. The input graph is loaded once at the start of a program and all computations are executed in-memory. As a result, Pregel supports only graphs that fit in memory [109].

Giraph [20] is an open source implementation of Pregel. Giraph runs workers as map-only jobs on Hadoop and uses HDFS for data input and output. Giraph also uses Apache ZooKeeper for coordination, checkpointing, and failure recovery schemes. Giraph has incorporated several optimizations, has a rapidly growing user base, and has been scaled by Facebook to graphs with a trillion edges. Giraph is executed in-memory, which can speed-up job execution, but, for large amounts of messages or big datasets, can also lead to crashes due to lack of memory.

GraphLab [135, 134] is an open-source, graph-specific distributed computation platform implemented in C++. GraphLab uses the GAS decomposition (Gather, Apply, Scatter), which is similar to, but fundamentally different from, the BSP model. In the GAS model, a vertex accumulates information about its neighbourhood in the Gather phase, applies the accumulated value in the Apply phase, and updates its adjacent vertices and edges and activates its neighbouring vertices in the Scatter phase. Another key difference is that GraphLab partitions graphs using vertex cuts rather than edge cuts. Consequently, each edge is assigned to a unique machine, while vertices are replicated in the caches of remote machines. Besides graph processing, it also supports various machine learning algorithms.

There are several works comparing graph processing frameworks. For instance, the first evaluation study of modern big data frameworks, including Map-Reduce, Stratosphere, Hama, Giraph and Graphlab, is presented in [93]. In [105], a benchmarking suite for graph-processing platforms is presented. The suite was used to evaluate the performance of Hadoop, YARN, Stratosphere, Giraph, GraphLab, and Neo4j. In [184], the authors present a comparison study on parallel processing systems, including Giraph, GPS and GraphLab. Finally, an Experimental Comparison of Pregel-like Graph Processing Systems is presented in [109].

References

1. 3store. http://sourceforge.net/projects/threestore/
2. 4store. http://www.4store.org
3. AllegroGraph. http://www.franz.com/agraph/allegrograph/
4. Apache Giraph. http://giraph.apache.org
5. Apache Hadoop. https://hadoop.apache.org
6. Apache Hadoop NextGen MapReduce (YARN). http://hadoop.apache.org/docs/current/hadoop-yarn/
7. Apache TinkerPop - An open source graph computing framework. http://tinkerpop.incubator.apache.org

8. ArangoDB. http://www.arangodb.org
9. Bitsy. https://bitbucket.org/lambdazen/bitsy/wiki/Home
10. Blazegraph. http://www.blazegraph.com/bigdata
11. Blueprints. https://github.com/tinkerpop/blueprints/wiki
12. BrightstarDB - A native RDF database for the .NET platform. http://brightstardb.com
13. Cayley graph database. https://github.com/google/cayley
14. Cloudgraph. http://www.cloudgraph.com/
15. Cypher - Graph Query Language. http://neo4j.com/developer/cypher-query-language/
16. DEX. http://www.sparsity-technologies.com/dex
17. Filament - Graph Management Toolkits. http://filament.sourceforge.net/
18. FlockDB. https://github.com/twitter/flockdb/
19. G-Store. http://g-store.sourceforge.net/
20. Giraph. https://github.com/aching/Giraph
21. GPS: A Graph Processing System. http://infolab.stanford.edu/gps/
22. Graphbase. http://graphbase.net/
23. Graphd. http://wiki.freebase.com/wiki/Graphd
24. GraphX - Apache API for graphs and graph-parallel computation. https://spark.apache.org/graphx/
25. HyperGraphDB - A Graph Database. http://www.hypergraphdb.org/
26. IBM System G. http://systemg.research.ibm.com
27. InfiniteGraph. http://infinitegraph.com/
28. InfoGrid - The Internet Graph Database. http://infogrid.org/
29. jCoreDB Graph. https://sites.google.com/site/jcoredb/
30. Neo4j. http://neo4j.org/
31. Ontotext GraphDB. http://www.ontotext.com/products/ontotext-graphdb/
32. OQGraph. https://mariadb.com/kb/en/mariadb/oqgraph-storage-engine/
33. Oracle spatial and graph. http://www.oracle.com/technetwork/database/options/spatialandgraph/
34. OrientDB - Multi-Model NoSQL Database. http://orientdb.com
35. Orly. https://github.com/cmaloney/orly
36. PQL - Path Query Language. http://www.eecs.harvard.edu/syrah/pql/
37. Redis graph: Graph database for Python. https://github.com/amix/redis_graph
38. Sesame. http://www.openrdf.org/
39. Sparksee - Scalable high-performance graph database. http://www.sparsity-technologies.com/
40. Stardog. http://stardog.com
41. StigDB. http://www.stigdb.org
42. Stratosphere - next generation big data analytics platform. http://stratosphere.eu
43. Titan - Distributed Graph Database. http://thinkaurelius.github.io/titan/
44. Trinity. http://research.microsoft.com/en-us/projects/trinity/
45. TurboGraph. http://wshan.net/turbograph/
46. VelocityGraph. http://www.velocitygraph.com
47. vertexdb. http://www.dekorte.com/projects/opensource/vertexdb/
48. Virtuoso Universal Server. http://virtuoso.openlinksw.com/
49. Weaver - New transactional graph database. http://www.ioremap.net/2014/12/23/weaver-new-transactional-graph-database/
50. Horton: Online Query Execution Engine for Large Distributed Graphs (Demo Track) (April 2012), `http://research.microsoft.com/apps/pubs/default.aspx?id=162643`

51. A Distributed Graph Engine for Web Scale RDF Data (August 2013), `http://research.microsoft.com/apps/pubs/default.aspx?id=183717`
52. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: Proceedings of the 33rd international conference on Very Large Data Bases (VLDB). pp. 411–422 (2007)
53. Abiteboul, S.: Querying Semi-Structured Data. In: Proceedings of the 6th International Conference on Database Theory (ICDT). LNCS, vol. 1186, pp. 1–18. Springer (Jan 1997)
54. Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J.L.: The Lorel query language for semistructured data. International Journal on Digital Libraries (JODL) 1(1), 68–88 (1997)
55. Abiteboul, S., Vianu, V.: Regular Path Queries with Constraints. In: Proceedings of the 16th ACM Symposium on Principles of Database Systems (PODS). pp. 122–133. ACM Press (1997)
56. Aggarwal, C.C., Wang, H. (eds.): Managing and Mining Graph Data (Advances in Database Systems). Springer Science – Business Media (2010)
57. Amann, B., Scholl, M.: Gram: A Graph Data Model and Query Language. In: European Conference on Hypertext Technology (ECHT). pp. 201–211. ACM (Nov - Dec 1992)
58. Andries, M., Gemis, M., Paredaens, J., Thyssens, I., den Bussche, J.V.: Concepts for Graph-Oriented Object Manipulation. In: Proceedings of the 3rd International Conference on Extending Database Technology (EDBT). LNCS, vol. 580, pp. 21–38. Springer (March 1992)
59. Angles, R.: A Comparison of Current Graph Database Models. In: Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops. pp. 171–177. IEEE Computer Society (2012)
60. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Computing Surveys (CSUR) 40(1), 1–39 (2008)
61. Anyanwu, K., Sheth, A.: P-Queries: enabling querying for semantic associations on the semantic web. In: Proceedings of the 12th International Conference on World Wide Web (WWW). pp. 690–699. ACM Press (2003)
62. Bakker, J.A., ter Bekke, J.H.: A query language solution for shortest path problems. In: Proc. of the IASTED International Conference Databases and Applications (2004)
63. Barceló, P., Hurtado, C., Libkin, L., Wood, P.: Expressive Languages for Path Queries over Graph-structured Data. In: Proc. of the 29th ACM Symposium on Principles of Database Systems (PODS). pp. 3–14 (2010)
64. Barcelo, P., Libkin, L., Reutter, J.: Querying graph patterns. In: Proc. of the 30th ACM Symposium on Principles of Database Systems (PODS). pp. 199–210 (2011)
65. Barceló Baeza, P.: Querying graph databases. In: Proceedings of the 32nd symposium on Principles of database systems. pp. 175–188. PODS '13, ACM, New York, NY, USA (2013)
66. Batra, S., Tyagi, C.: Comparative analysis of relational and graph databases. International Journal of Soft Computing and Engineering (IJSCE) 2(2) (2012)
67. Berge, C.: Graphs and Hypergraphs. North-Holland, Amsterdam (1973)
68. Bornea, M.A., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O., Bhattacharjee, B.: Building an efficient RDF store over a relational database. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD). pp. 121–132. ACM, New York, NY, USA (2013)

69. Bray, T., Paoli, J., Sperberg-McQueen, C.M.: Extensible Markup Language (XML) 1.0, W3C Recommendation. http://www.w3.org/TR/1998/REC-177-19980210 (10 February 1998)
70. Buchsbaum, A.L., Kanellakis, P.C., Vitter, J.S.: A data dtructure for arc insertion and regular path finding. Ann. Math. Artif. Intell. 3(2-4), 187–210 (1991)
71. Buerli, M.: The current state of graph databases. Tech. rep. (December 2012)
72. Buneman, P.: Semistructured Data. In: Proceedings of the 16th Symposium on Principles of Database Systems (PODS). pp. 117–121. ACM Press (May 1997)
73. Buneman, P., Davidson, S., Hillebrand, G., Suciu, D.: A Query Language and Optimization Techniques for Unstructured Data. SIGMOD Record 25(2), 505–516 (1996)
74. Calvanese, D., Giacomo, G.D., Lenzerini, M., , Vardi, M.Y.: Containment of conjunctive regular path queries with inverse. In: Proc. of the 7th Int. Conf. on the Principles of Knowledge Representation and Reasoning (2000)
75. Canim, M., Chang, Y.C.: System G Data Store: Big, Rich Graph Data Analytics in the Cloud. In: IEEE International Conference on Cloud Engineering (IC2E). pp. 328–337 (March 2013)
76. Carroll, J.: Matching RDF Graphs. In: Proceedings of the International Semantic Web Conference (ISWC) (2002)
77. Chang, C.S., Chen, A.L.P.: Supporting conceptual and neighborhood queries on the world wide web. IEEE Transactions on Systems, Man, and Cybernetics (TSMC) 28(2), 300–308 (1998)
78. Chen, P.P.S.: The Entity-Relationship Model - Toward a Unified View of Data. ACM Transactions on Database Systems (TODS) 1(1), 9–36 (1976)
79. Cheng, J., Yu, J.X., Ding, B., Yu, P.S., Wang, H.: Fast graph pattern matching. In: Proc. of the 2008 IEEE 24th International Conference on Data Engineering (ICDE). pp. 913–922. IEEE Computer Society (2008)
80. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An Efficient SQL-based RDF Querying Scheme. In: Proc. VLDB Endow. pp. 1216–1227 (2005)
81. Ciglan, M., Averbuch, A., Hluchy, L.: Benchmarking Traversal Operations over Graph Databases. In: Proceedings of the IEEE 28th International Conference on Data Engineering Workshops. pp. 186–189. IEEE Computer Society (2012)
82. Ciglan, M., Nørvåg, K.: SGDB: Simple Graph Database Optimized for Activation Spreading Computation. In: Proceedings of the 15th International Conference on Database Systems for Advanced Applications (DASFAA). pp. 45–56. Springer-Verlag (2010)
83. Codd, E.F.: A Relational Model of Data for Large Shared Data Banks. Communications of the ACM 13(6), 377–387 (1970)
84. Codd, E.F.: A Relational Model of Data for Large Shared Data Banks. Communications of the ACM 26(1), 64–69 (1983)
85. Company, H.P.D.: Jena - A Semantic Web Framework for Java. http://jena.sourceforge.net/ (2000)
86. Consens, M., Mendelzon, A.: Hy+: a Hygraph-based Query and Visualization System. SIGMOD Record 22(2), 511–516 (1993)
87. Consens, M.P., Mendelzon, A.O.: GraphLog: a Visual Formalism for Real Life Recursion. In: Proceedings of the 9th ACM Symposium on Principles of Database Systems. pp. 404–416. PODS, ACM Press (April 2-4 1990)
88. Cruz, I.F., Mendelzon, A.O., Wood, P.T.: A Graphical Query Language Supporting Recursion. In: Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data. pp. 323–330. ACM Press (May 1987)

89. Cudré-Mauroux, P., Enchev, I., Fundatureanu, S., Groth, P., Haque, A., Harth, A., Keppmann, F., Miranker, D., Sequeda, J., Wylot, M.: Nosql databases for rdf: An empirical evaluation. In: Alani, H., Kagal, L., Fokoue, A., Groth, P., Biemann, C., Parreira, J., Aroyo, L., Noy, N., Welty, C., Janowicz, K. (eds.) Proc. of the International Semantic Web Conference (ISWC), LNCS, vol. 8219, pp. 310–325. Springer Berlin Heidelberg (2013)
90. De Moor, O., Lacey, D., Van Wyk, E.: Universal Regular Path Queries. Higher Order Symbol. Comput. 16(1-2), 15–35 (Mar 2003)
91. Dominguez-Sal, D., Martinez-Bazan, N., Muntes-Mulero, V., Baleta, P., Larriba-Pey, J.L.: A discussion on the design of graph database benchmarks. In: TPC Technology Conference on Performance Evaluation and Benchmarking (2010)
92. Dominguez-Sal, D., Urbón-Bayes, P., Giménez-Vañó, A., Gómez-Villamor, S., Martínez-Bazán, N., Larriba-Pey, J.L.: Survey of graph database performance on the hpc scalable graph analysis benchmark. In: Proc. of the 2010 international conference on Web-age information management (WAIM). pp. 37–48. Springer-Verlag (2010)
93. Elser, B., Montresor, A.: An evaluation study of BigData frameworks for graph processing. In: Proceedings of the IEEE International Conference on Big Data. pp. 60–67. IEEE (2013)
94. Fan, W., Li, J., Luo, J., Tan, Z., Wang, X., Wu, Y.: Incremental graph pattern matching. In: Proc. of the International Conference on Management of data (SIGMOD). ACM Press (2011)
95. Fan, W., Li, J., Ma, S., Tang, N., Wu, Y.: Adding regular expressions to graph reachability and pattern queries. In: Proc. of the IEEE 27th International Conference on Data Engineering (ICDE). pp. 39–50 (2011)
96. Fan, W., Li, J., Ma, S., Tang, N., Wu, Y., Wu, Y.: Graph pattern matching: from intractable to polynomial time. Proc. VLDB Endowment 3, 264–275 (September 2010)
97. Faye, D.C., Cure, O., Blin, G.: A survey of RDF storage approaches. In: ARIMA Journal. vol. 15 (2012)
98. Feigenbaum, L., Williams, G.T., Clark, K.G., Torres, E.: Sparql 1.1 protocol. w3c recommendation. http://www.w3.org/TR/2013/REC-sparql11-protocol-20130321/ (March 21 2013)
99. Fernández, M., Florescu, D., Kang, J., Levy, A., Suciu, D.: Catching the boat with Strudel: Experiences with a Web-site Management System. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD). pp. 414–425. ACM Press (June 1998)
100. Gemis, M., Paredaens, J.: An Object-Oriented Pattern Matching Language. In: Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software. pp. 339–355. Springer-Verlag (1993)
101. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI). pp. 17–30. USENIX Association, Berkeley, CA, USA (2012)
102. Graves, M., Bergeman, E.R., Lawrence, C.B.: A Graph-Theoretic Data Model for Genome Mapping Databases. In: Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS). p. 32. IEEE Computer Society (1995)
103. Gubichev, A., Neumann, T.: Path query processing on very large rdf graphs. In: Proc. of the 14th International Workshop on the Web and Databases (WebDB) (2011)

104. Guha, R., McCool, R., Miller, E.: Semantic Search. In: Proceedings of the 12th International Conference on World Wide Web (WWW). pp. 700–709. ACM Press (2003)
105. Guo, Y., Biczak, M., Varbanescu, A.L., Iosup, A., Martella, C., Willke, T.L.: How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis. In: Proceedings of the IEEE 28th International Parallel and Distributed Processing Symposium. pp. 395–404. IEEE Computer Society, Washington, DC, USA (2014)
106. Gutiérrez, A., Pucheral, P., Steffen, H., Thévenin, J.M.: Database Graph Views: A Practical Model to Manage Persistent Graphs. In: Proceedings of the 20th International Conference on Very Large Data Bases (VLDB). pp. 391–402. Morgan Kaufmann (September 1994)
107. Gutierrez, C., Hurtado, C., O.Mendelzon, A.: Foundations of Semantic Web Databases. In: Proceedings of the 23rd Symposium on Principles of Database Systems (PODS). pp. 95–106 (June 14-16 2004)
108. Gyssens, M., Paredaens, J., den Bussche, J.V., Gucht, D.V.: A Graph-Oriented Object Database Model. In: Proceedings of the 9th Symposium on Principles of Database Systems (PODS). pp. 417–424. ACM Press (1990)
109. Han, M., Daudjee, K., Ammar, K., Özsu, M.T., Wang, X., Jin, T.: An Experimental Comparison of Pregel-like Graph Processing Systems. Proc. VLDB Endow. 7(12), 1047–1058 (Aug 2014)
110. Han, W.S., Lee, S., Park, K., Lee, J.H., Kim, M.S., Kim, J., Yu, H.: TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC. In: Proceedings of the 19th ACM International Conference on Knowledge Discovery and Data Mining. pp. 77–85. ACM, New York, NY, USA (2013)
111. Hayes, J., Gutierrez, C.: Bipartite Graphs as Intermediate Model for RDF. In: Proceedings of the 3th International Semantic Web Conference (ISWC). pp. 47–61. No. 3298 in LNCS, Springer-Verlag (Nov 2004)
112. Hidders, J.: Typing Graph-Manipulation Operations. In: Proceedings of the 9th International Conference on Database Theory (ICDT). pp. 394–409. Springer-Verlag (2002)
113. Hidders, J., Paredaens, J.: GOAL, A Graph-Based Object and Association Language. Advances in Database Systems: Implementations and Applications, CISM pp. 247–265 (Sept 1993)
114. Hull, R., King, R.: Semantic Database Modeling: Survey, Applications, and Research Issues. ACM Computing Surveys 19(3), 201–260 (1987)
115. Iordanov, B.: HyperGraphDB: a generalized graph database. In: Proceedings of the International Conference on Web-age Information Management (WAIM). pp. 25–36. Springer-Verlag (2010)
116. Jouili, S., Reynaga, A.: imGraph: A Distributed In-Memory Graph Database. In: Proc. of the International Conference on Social Computing (SocialCom). pp. 732–737 (2013)
117. Jouili, S., Vansteenberghe, V.: An empirical comparison of graph databases. In: Social Computing (SocialCom), 2013 International Conference on. pp. 708–715 (Sept 2013)
118. Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In: Proceedings of the 9th IEEE International Conference on Data Mining. pp. 229–238. IEEE Computer Society, Washington, DC, USA (2009)
119. Khan, A., Elnikety, S.: Systems for Big-Graphs. In: Proc. of the 40th International Conference on Very Large Data Bases (VLDB) (2014)

120. Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., Kalnis, P.: Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In: Proceedings of the 8th ACM European Conference on Computer Systems. pp. 169–182. ACM, New York, NY, USA (2013)
121. Kiesel, N., Schurr, A., Westfechtel, B.: GRAS: A Graph-Oriented Software Engineering Database System. In: IPSEN Book. pp. 397–425 (1996)
122. Kim, W.: Object-Oriented Databases: Definition and Research Directions. IEEE Transactions on Knowledge and Data Engineering (TKDE) 2(3), 327–341 (1990)
123. Klyne, G., Carroll, J.: Resource Description Framework (RDF) Concepts and Abstract Syntax. http://www.w3.org/TR/2004/REC-115-concepts-20040210/ (February 2004)
124. Kowalik, L.: Adjacency queries in dynamic sparse graphs. Information Processing Letters 102, 191–195 (May 2007)
125. Kunii, H.S.: DBMS with Graph Data Model for Knowledge Handling. In: Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow. pp. 138–142. IEEE Computer Society Press (1987)
126. Kuper, G.M., Vardi, M.Y.: A New Approach to Database Logic. In: Proceedings of the 3th Symposium on Principles of Database Systems (PODS). pp. 86–96. ACM Press (April 1984)
127. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: Large-scale Graph Computation on Just a PC. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. pp. 31–46. USENIX Association, Berkeley, CA, USA (2012)
128. Lécluse, C., Richard, P., Vélez, F.: O2, an Object-Oriented Data Model. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD). pp. 424–433. ACM Press (June 1988)
129. Levene, M., Loizou, G.: A Graph-Based Data Model and its Ramifications. IEEE Transactions on Knowledge and Data Engineering (TKDE) 7(5), 809–823 (1995)
130. Levene, M., Poulovassilis, A.: The Hypernode Model and its Associated Query Language. In: Proceedings of the 5th Jerusalem Conference on Information technology. pp. 520–530. IEEE Computer Society Press (1990)
131. Levene, M., Poulovassilis, A.: An Object-Oriented Data Model Formalised Through Hypergraphs. Data & Knowledge Engineering (DKE) 6(3), 205–224 (1991)
132. Liu, Y.A., Stoller, S.D.: Querying complex graphs. In: Proc. of the 8th Int. Symposium on Practical Aspects of Declarative Languages. pp. 16–30. Springer-Verlag (2006)
133. Liu, Y.A., Yu, F.: Solving Regular Path Queries. In: Proceedings of the 6th International Conference on Mathematics of Program Construction. pp. 195–208. Springer-Verlag (2002)
134. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. Proc. VLDB Endow. 5(8), 716–727 (2012)
135. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: GraphLab: A New Parallel Framework for Machine Learning. In: Conference on Uncertainty in Artificial Intelligence (UAI) (July 2010)
136. Mainguenaud, M.: Simatic XT: A Data Model to Deal with Multi-scaled Networks. Computer, Environment and Urban Systems 16, 281–288 (1992)
137. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of

the International Conference on Management of data (SIGMOD). pp. 135–146. ACM, New York, NY, USA (2010)

138. Martínez-Bazan, N., Muntés-Mulero, V., Gómez-Villamor, S., Nin, J., Sánchez-Martínez, M.A., Larriba-Pey, J.L.: DEX: High-Performance Exploration on Large Graphs for Information Retrieval. In: Proceedings of the 16th Conference on Information and Knowledge Management (CIKM). pp. 573–582. ACM (2007)
139. Matono, A., Amagasa, T., Yoshikawa, M., Uemura, S.: An Eficient Pathway Search Using an Indexing Scheme for RDF. Genome Informatics (14), 374–375 (2003)
140. McColl, R., Ediger, D., Poovey, J., Campbell, D., Bader, D.A.: A Brief Study of Open Source Graph Databases. CoRR (2013)
141. McGuinness, D.L., van Harmelen, F.: OWL Web Ontology Language Overview, W3C Recommendation. http://www.w3.org/TR/2004/REC-133-features-20040210/ (10 February 2004)
142. Mendelzon, A.O., Wood, P.T.: Finding Regular Simple Paths in Graph Databases. In: Proceedings of the 15th International Conference on Very Large Data Bases (VLDB). pp. 185–193. Morgan Kaufmann Publishers Inc. (1989)
143. Meyer, S.M., Degener, J., Giannandrea, J., Michener, B.: Optimizing Schema-last Tuple-store Queries in Graphd. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD). pp. 1047–1056. ACM, New York, NY, USA (2010)
144. Michael Schmidt and Thomas Hornung and Norbert Küchlin and Georg Lausen and Christoph Pinkel: An Experimental Comparison of RDF Data Management Approaches in a SPARQL Benchmark Scenario. In: Proc. of the 7th International Semantic Web Conference (ISWC). pp. 82–97. Springer-Verlag (2008)
145. Morari, A., Castellana, V., Villa, O., Tumeo, A., Weaver, J., Haglin, D., Choudhury, S., Feo, J.: Scaling semantic graph databases in size and performance. IEEE Micro 34(4), 16–26 (July 2014)
146. Navathe, S.B.: Evolution of Data Modeling for Databases. Communications of the ACM 35(9), 112–123 (1992)
147. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. The VLDB Journal 19(1), 91–113 (2010)
148. Papadopoulos, A.N., Manolopoulos, Y.: Nearest Neighbor Search - A Database Perspective. Series in Computer Science, Springer (2005)
149. Papakonstantinou, Y., Garcia-Molina, H., Widom, J.: Object Exchange across Heterogeneous Information Sources. In: Proceedings of the 11th International Conference on Data Engineering (ICDE). pp. 251–260. IEEE Computer Society (1995)
150. Paredaens, J., Kuijpers, B.: Data Models and Query Languages for Spatial Databases. Data & Knowledge Engineering (DKE) 25(1-2), 29–53 (1998)
151. Paredaens, J., Peelman, P., Tanca, L.: G-Log: A Graph-Based Query Language. IEEE Transactions on Knowledge and Data Engineering (TKDE) 7(3), 436–453 (1995)
152. Peckham, J., Maryanski, F.J.: Semantic Data Models. ACM Computing Surveys 20(3), 153–189 (1988)
153. Poulovassilis, A., Levene, M.: A Nested-Graph Model for the Representation and Manipulation of Complex Objects. ACM Transactions on Information Systems (TOIS) 12(1), 35–68 (1994)
154. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation. http://www.w3.org/TR/2008/REC-115-sparql-query-20080115/ (January 15 2008)

155. Ralf Hartmut Güting: GraphDB: Modeling and Querying Graphs in Databases. In: Proceedings of 20th International Conference on Very Large Data Bases (VLDB). pp. 297–308. Morgan Kaufmann (1994)
156. Rodriguez, M.A., Neubauer, P.: Constructions from dots and lines. Bul. Am. Soc. Info. Sci. Tech. 36(6), 35–41 (2010)
157. Rodriguez, M.A.: Mapping Semantic Networks to Undirected Networks. International Journal of Applied Mathematics and Computer Sciences 5(1), 30–42 (2009)
158. Roussopoulos, N., Mylopoulos, J.: Using Semantic Networks for Database Management. In: Proceedings of the International Conference on Very Large Data Bases (VLDB). pp. 144–172. ACM (Sept 1975)
159. Salihoglu, S., Widom, J.: GPS: A Graph Processing System. In: Proceedings of the 25th International Conference on Scientific and Statistical Database Management. pp. 22:1–22:12. ACM, New York, NY, USA (2013)
160. Samet, H.: Modern Database Systems: The Object Model, Interoperability and Beyond, chap. Spatial data structures, pp. 361–385. Addison Wesley - ACM Press (1995)
161. Sarwat, M., Elnikety, S., He, Y., Mokbel, M.F.: Horton+: A Distributed System for Processing Declarative Reachability Queries over Partitioned Graphs. Proc. VLDB Endow. 6(14), 1918–1929 (Sep 2013)
162. Seidl, T., peter Kriegel, H.: A 3d molecular surface representation supporting neighborhood queries. In: Proc. of the 3rd Conference on Intelligent Systems for Molecular Biology (ISMB). pp. 240–258. Springer (1995)
163. Shang, Z., Yu, J.X.: Catch the wind: Graph workload balancing on cloud. In: 30th International Conference on Data Engineering. pp. 553–564. IEEE Computer Society (2013)
164. Shao, B., Wang, H., Li, Y.: Trinity: a distributed graph engine on a memory cloud. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD). pp. 505–516. ACM, New York, NY, USA (2013)
165. Shasha, D., Wang, J.T.L., Giugno, R.: Algorithmics and Applications of Tree and Graph Searching. In: Proceedings of the 21th Symposium on Principles of Database Systems (PODS). pp. 39–52. ACM Press (2002)
166. Shipman, D.W.: The Functional Data Model and the Data Language DAPLEX. ACM Transactions on Database Systems (TODS) 6(1), 140–173 (1981)
167. Stegmaier, F., Grobner, U., Dolller, M., Kosch, H., Baese, G.: Evaluation of current rdf database solutions. In: 10th International Workshop of the Multimedia Metadata Community on Semantic Multimedia Database Technologies (SeMuDaTe). vol. 539. CEUR (2009)
168. Stein, L.D., Tierry-Mieg, J.: AceDB: A genome Database Management System. Computing in Science & Engineering (CiSE) 1(3), 44–52 (1999)
169. Tarjan, R.E.: Fast Algorithms for Solving Path Problems. Journal of the ACM 28(3), 594–614 (Jul 1981)
170. Thompson, B.: Literature survey of graph databases. Tech. rep., SYSTAP (January 23 2013)
171. Tompa, F.W.: A Data Model for Flexible Hypertext Database Systems. ACM Transactions on Information Systems (TOIS) 7(1), 85–100 (1989)
172. Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., Wilkins, D.: A comparison of a graph database and a relational database. In: ACM Southeast Regional Conference (2010)
173. Wang, X.: Finding patterns on protein surfaces: Algorithms and applications to protein classification. IEEE Transactions on Knowledge and Data Engineering 17, 1065–1078 (2005)

174. Washio, T., Motoda, H.: State of the Art of Graph-based Data Mining. SIGKDD Explorer Newsletter 5(1), 59–68 (2003)
175. Watters, C., Shepherd, M.A.: A Transient Hypergraph-Based Model for Data Access. ACM Transactions on Information Systems (TOIS) 8(2), 77–102 (1990)
176. Watts, D.J., Strogatz, S.H.: Collective dynamics of small-world networks. Nature 393(6684), 440–442 (1998)
177. Wood, P.T.: Query languages for graph databases. SIGMOD Record 41(1), 50–60 (Apr 2012)
178. Xia, Y., Tanase, I., Nai, L., Tan, W., Liu, Y., Crawford, J., Lin, C.: Explore efficient data organization for large scale graph analytics and storage. In: IEEE International Conference on Big Data (2014)
179. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: GraphX: A Resilient Distributed Graph System on Spark. In: First International Workshop on Graph Data Management Experiences and Systems (GRADES). pp. 2:1–2:6. ACM, New York, NY, USA (2013)
180. Xu, A., Lei, H.: LCGMiner: Levelwise Closed Graph Pattern Mining from Large Databases. In: Proceedings of the 16th International Conference on Scientific and Statistical Database Management. pp. 421–422. IEEE (June 2004)
181. Xu, A., Lei, H.: LCGMiner: Levelwise Closed Graph Pattern Mining from Large Databases. In: Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM'04). p. 421. IEEE (June 2004)
182. Yannakakis, M.: Graph-Theoretic Methods in Database Theory. In: Proceedings of the 9th Symposium on Principles of Database Systems (PODS). pp. 230–242. ACM Press (1990)
183. Yuan, P., Liu, P., Wu, B., Jin, H., Zhang, W., Liu, L.: TripleBit: A Fast and Compact System for Large Scale RDF Data. Proc. VLDB Endow. 6(7), 517–528 (May 2013)
184. Zhao, Y., Yoshigoe, K., Xie, M., Zhou, S., Seker, R., Bian, J.: Evaluation and analysis of distributed graph-parallel processing frameworks. Journal of Cyber Security and Mobility 3(3), 289–316 (July 2014)
185. Zou, L., Özsu, M., Chen, L., Shen, X., Huang, R., Zhao, D.: gStore: a graph-based SPARQL query engine. The VLDB Journal 23(4), 565–590 (2014)