

Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries

MACIEJ BESTA, Department of Computer Science, ETH Zurich

ROBERT GERSTENBERGER, Department of Computer Science, ETH Zurich

EMANUEL PETER, Department of Computer Science, ETH Zurich

MARC FISCHER, PRODYNA (Schweiz) AG

MICHAŁ PODSTAWSKI, Future Processing

CLAUDE BARTHEL, Department of Computer Science, ETH Zurich

GUSTAVO ALONSO, Department of Computer Science, ETH Zurich

TORSTEN HOEFLER, Department of Computer Science, ETH Zurich

Graph processing has become an important part of multiple areas of computer science, such as machine learning, computational sciences, medical applications, social network analysis, and many others. Numerous graphs such as web or social networks may contain up to trillions of edges. Often, these graphs are also dynamic (their structure changes over time) and have domain-specific rich data associated with vertices and edges. Graph database systems such as Neo4j enable storing, processing, and analyzing such large, evolving, and rich datasets. Due to the sheer size of such datasets, combined with the irregular nature of graph processing, these systems face unique design challenges. To facilitate the understanding of this emerging domain, we present the first survey and taxonomy of graph database systems. We focus on identifying and analyzing fundamental categories of these systems (e.g., triple stores, tuple stores, native graph database systems, or object-oriented systems), the associated graph models (e.g., RDF or Labeled Property Graph), data organization techniques (e.g., storing graph data in indexing structures or dividing data into records), and different aspects of data distribution and query execution (e.g., support for sharding and ACID). 45 graph database systems are presented and compared, including Neo4j, OrientDB, or Virtuoso. We outline graph database queries and relationships with associated domains (NoSQL stores, graph streaming, and dynamic graph algorithms). Finally, we describe research and engineering challenges to outline the future of graph databases.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Information systems** → **Data management systems**; **Graph-based database models**; **Data structures**; **DBMS engine architectures**; **Database query processing**; **Parallel and distributed DBMSs**; *Database design and models*; Distributed database transactions; • **Theory of computation** → *Data modeling*; *Data structures and algorithms for data management*; Distributed algorithms; • **Computer systems organization** → Distributed architectures;

Additional Key Words and Phrases: Graphs, Graph Databases, NoSQL Stores, Graph Database Management Systems, Graph Models, Data Layout, Graph Queries, Graph Transactions, Graph Representations, RDF, Labeled Property Graph, Triple Stores, Key-Value Stores, RDBMS, Wide-Column Stores, Document Stores

ACM Reference format:

Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoeffler. 2021. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. 41 pages.

1 INTRODUCTION

Graph processing is behind numerous problems in computing, for example in medicine, machine learning, computational sciences, and others [111, 130]. Graph algorithms are inherently difficult to design because of challenges such as large sizes of processed graphs, little locality, or irregular communication [37, 54, 126, 130, 171, 189]. The difficulties are increased by the fact that many such graphs are also dynamic (their structure changes over time) and have rich data, for example arbitrary properties or labels, associated with vertices and edges.

Graph databases¹ such as Neo4j [168] emerged to enable storing, processing, and analyzing large, evolving, and rich graph datasets. Graph databases face unique challenges due to overall properties of irregular graph computations combined with the demand for low latency and high throughput of graph queries that can be both *local* (i.e., accessing or modifying a small part of the graph, for example a single edge) and *global* (i.e., accessing or modifying a large part of the graph, for example all the edges). Many of these challenges belong to the following areas: “general design” (i.e., what is the most advantageous general structure of a graph database engine), “data models and organization” (i.e., how to model and store the underlying graph dataset), “data distribution” (i.e., whether and how to distribute the data across multiple servers), and “transactions and queries” (i.e., how to query the underlying graph dataset to extract useful information). This distinction is illustrated in Figure 1. In this work, we present the first survey and taxonomy on these system aspects of graph databases.

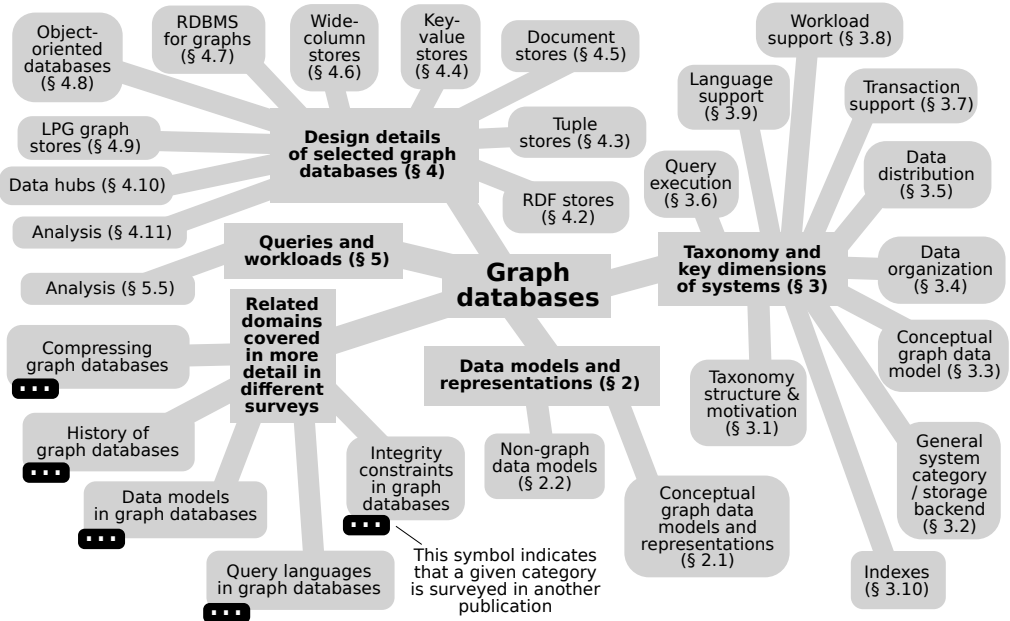


Fig. 1. The illustration of the considered areas of graph databases.

¹Lists of graph databases can be found at <http://nosql-database.org>, <https://database.guide>, <https://www.g2crowd.com/categories/graph-databases>, <https://www.predictiveanalyticstoday.com/top-graph-databases>, and <https://db-engines.com/en/ranking/graph+dbms>.

In general, we provide the following contributions:

- We provide the first taxonomy of graph databases, identifying and analyzing key dimensions in the design of graph databases: (1) general database engine, (2) data model, (3) data organization, (4) data distribution, (5) query execution, and (6) type of transactions.
- We use our taxonomy to survey, categorize, and compare 51 graph database systems.
- We discuss in detail the design of selected graph databases.
- We outline related domains, such as queries and workloads in graph databases.
- We discuss future challenges in the design of graph databases.

1.1 Discussion on Other Classes of Systems

In addition to graph databases, other systems can also store and process dynamic graphs. We now briefly relations to two such classes: NoSQL stores and streaming graph frameworks.

Graph Databases vs. NoSQL Stores and Other Database Systems NoSQL stores address various deficiencies of relational database systems, such as little support for flexible data models [63]. Graph databases such as Neo4j can be seen as one particular type of NoSQL stores; these systems are sometimes referred to as “*native*” graph databases [168]. Other types of NoSQL systems include *wide-column stores*, *document stores*, and general *key-value stores* [63]. Here, we focus on *any database system that enables storing and processing graphs*, including native graph databases and other types of NoSQL stores, relational databases, object-oriented databases, and others. Figure 2 shows the types of considered systems.

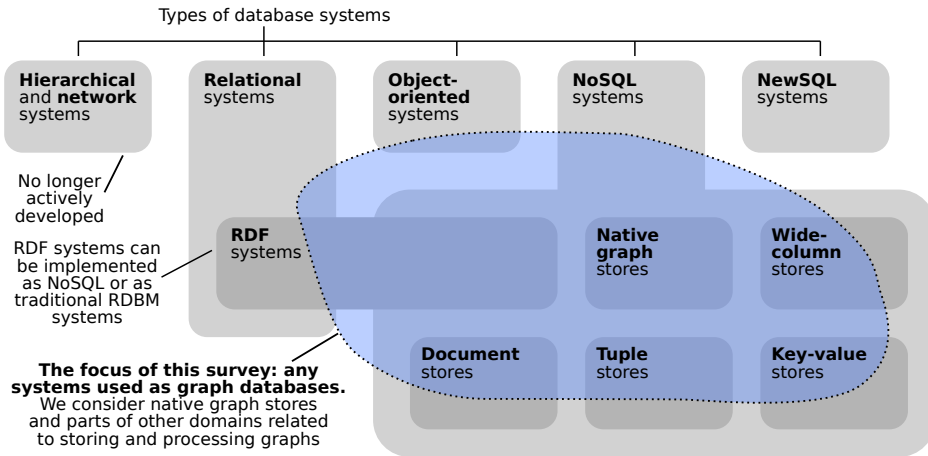


Fig. 2. The illustration of the considered types of databases.

Graph Databases vs. Graph Streaming Frameworks In *graph streaming* [23], the input graph is passed as a stream of updates, allowing to add and remove edges in a simple way. Graph databases are related to graph streaming in that they face graph updates of various types. Still, they usually deal with complex graph models (such as the Labeled Property Graph [4] or Resource Description Framework [59]) where both vertices and edges may be of different types and may be associated with arbitrary properties. Contrarily, graph streaming frameworks such as STINGER [73] focus on simple graph models where edges or vertices may have weights and, in some cases, simple additional properties such as time stamps. Moreover, challenges in the design of graph databases include transactional support, a topic little related to graph streaming frameworks.

Graph Databases vs. Graph Processing Systems A lot of effort has been dedicated to general graph processing, and several associated surveys and analyses exist [20, 68, 96, 138, 179, 201]. Many of these works focus on the vertex-centric paradigms [1, 34, 114, 179]. Some works also focus on edge-centric or linear algebra paradigms [119, 183, 187]. The key differences to graph databases are that graph processing systems usually focus on graphs that are static and simple, i.e., do not have rich attached data such as labels or key-value pairs (details in § 2.1). Moreover, the associated workloads focus on “global” graph analytics such as PageRank (details in Section 5).

1.2 Discussion on Related Surveys

There exist several surveys dedicated to the theory of graph databases. In 2008, Angles et al. [6] described the history of graph databases, and, in particular, the used data models, data structures, query languages, and integrity constraints. In 2017, Angles et al. [4] analyzed in more detail query languages for graph databases, taking both an edge-labeled and a property graph model into account and studying queries such as graph pattern matching and navigational expressions. In 2018, Bonifati et al. [42] provided an in-depth investigation into querying graphs, focusing on numerous aspects of query specification and execution. Moreover, there are surveys that focus on NoSQL stores [63, 83, 94] and RDF [154]. There is no survey dedicated to the systems aspects of graph databases, except for several brief papers that cover small parts of the domain (brief descriptions of a few systems, concepts, or techniques [113, 115, 123, 156, 160], a survey of graph processing ubiquity [173], and performance evaluations of a few systems [121, 137, 195]).

2 GRAPHS AND DATA MODELS IN THE LANDSCAPE OF GRAPH DATABASES

We start with data models. This includes conceptual graph models and representations, and non-graph models used in graph databases. Key symbols and abbreviations are shown in Table 1.

G	A graph $G = (V, E)$ where V is a set of vertices and E is a set of edges.
n, m	The count of vertices and edges in a graph G ; $ V = n, E = m$.
d, \hat{d}	The average degree and the maximum degree in a given graph, respectively.
$\mathcal{P}(S) = 2^S$	The power set of S : a set that contains all possible subsets of S .
AM, M	The Adjacency Matrix representation. $M \in \{0, 1\}^{n,n}$, $M_{u,v} = 1 \Leftrightarrow (u, v) \in E$.
AL, A_u	The Adjacency List representation and the adjacency list of a vertex u ; $v \in A_u \Leftrightarrow (u, v) \in E$.
LPG, RDF	Labeled Property Graph (§ 2.1.3) and Resource Description Framework (§ 2.1.5).
KV, RDBMS	Key-Value store (§ 4.4) and Relational Database Management Systems (§ 4.7).
OODBMS	Object-Oriented Database Management Systems (§ 4.8).
OLTP, OLAP	Online Transaction Processing (§ 3.7) and Online Analytics Processing (§ 3.7).
ACID	Transaction guarantees (Atomicity, Consistency, Isolation, Durability).

Table 1. The most relevant symbols and abbreviations used in this work.

2.1 Conceptual Graph Models

First, we introduce the graph models used by the surveyed systems.

2.1.1 Simple Graph Model. A graph G can be modeled as a tuple (V, E) where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. $G = (V, E)$ can also be denoted as $G(V, E)$. We have $|V| = n$ and $|E| = m$. For a directed G , an edge $e = (u, v) \in E$ is a tuple of two vertices, where u is the out-vertex (also called “source”) and v is the in-vertex (also called “destination”). If G is undirected, an edge $e = \{u, v\} \in E$ is a set of two vertices. Finally, a weighted graph G is modeled with a triple (V, E, w) ; $w : E \rightarrow \mathbb{R}$ maps edges to weights.

In the AM format, a matrix $\mathbf{M} \in \{0, 1\}^{n,n}$ determines the connectivity of vertices: $\mathbf{M}_{u,v} = 1 \Leftrightarrow (u, v) \in E$. In the AL format, each vertex u has an associated adjacency list A_u . This adjacency list maintains the IDs of all vertices adjacent to u . Each adjacency list is often stored as a contiguous array of vertex IDs. We have $v \in A_u \Leftrightarrow (u, v) \in E$.

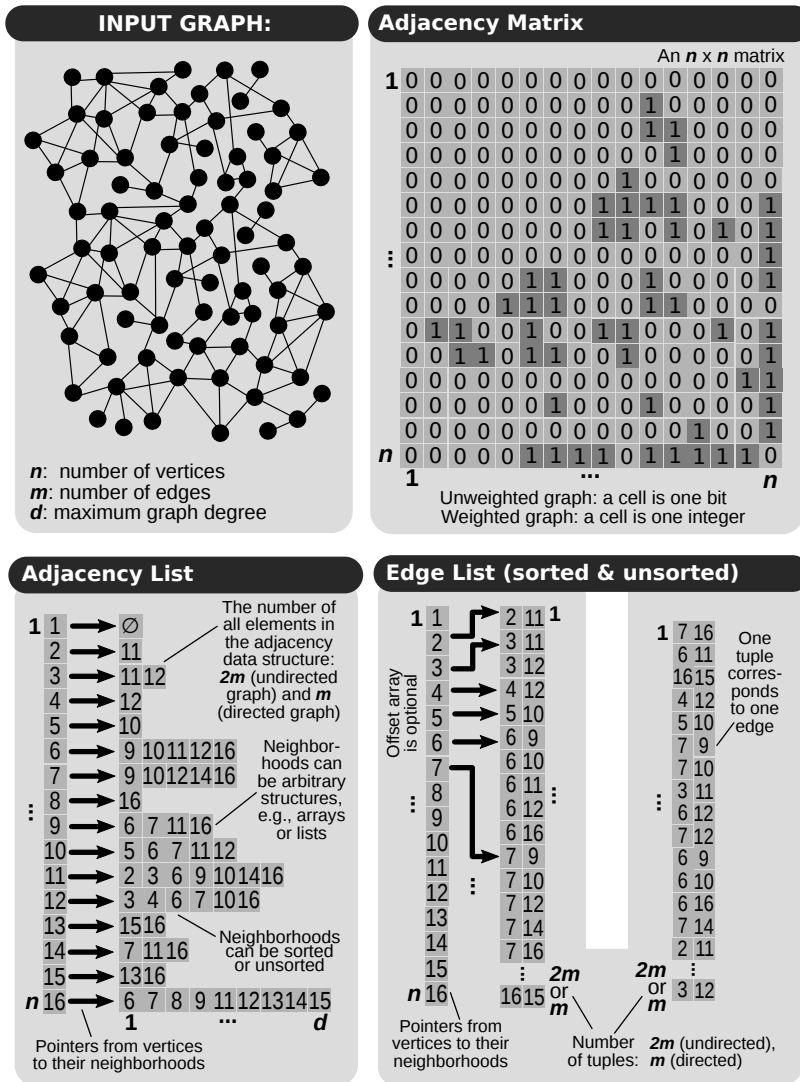


Fig. 3. Illustration of fundamental graph representations: Adjacency Matrix, Adjacency List, and Edge List.

AM uses $O(n^2)$ space and can check connectivity of two vertices in $O(1)$ time. AL requires $O(n + m)$ space and it can check connectivity in $O(|A_u|) \subseteq O(\bar{d})$ time. The AL or AM representations are used to maintain the graph structure (i.e., neighborhoods of vertices).

A simple graph model is often used in graph processing frameworks such as Pregel [131] or STINGER [73]. It is not commonly used with graph databases. Instead, it is a basis for more complex models, such as the Labeled Property Graph or Resource Description Framework.

2.1.2 Hypergraph Model. A hypergraph H generalizes a simple graph: any of its edges can join *any number of vertices*. Formally, a hypergraph is also modeled as a tuple (V, E) with V being a set of vertices. E is defined as $E \subseteq (\mathcal{P}(V) \setminus \emptyset)$ and it contains *hyperedges*: non-empty subsets of V .

Hypergraphs are rarely used in graph databases and graph processing systems. In this survey, we describe a system called HyperGraphDB (§ 4.4.2) that focuses on storing and querying hypergraphs.

2.1.3 Labeled Property Graph Model. The classical graph model, a tuple $G = (V, E)$, is adequate for many problems such as computing vertex centralities [43]. However, it is not rich enough to model various real-world problems. This is why graph databases often use the *Labeled Property Graph Model (LPG)*, sometimes simply called a property graph [4, 42]. In LPG, one augments the simple graph model (V, E) with *labels* that define different subsets (or classes) of vertices and edges. Furthermore, every vertex and edge can have any number of *properties* [42] (often also called *attributes*). A property is a pair $(key, value)$, where *key* identifies a property and *value* is the corresponding value of this property [42]. Formally, an LPG is defined as a tuple $(V, E, L, l_V, l_E, K, W, p_V, p_E)$ where L is the set of labels. $l_V : V \mapsto \mathcal{P}(L)$ and $l_E : E \mapsto \mathcal{P}(L)$ are labeling functions. Note that $\mathcal{P}(L)$ is the power set of L , denoting all the possible subsets of L . Thus, each vertex and edge is mapped to a subset of labels. Next, a vertex as well as an edge can be associated with any number of properties. We model a property as a key-value pair $p = (key, value)$, where $key \in K$ and $value \in W$. K and W are sets of all possible keys and values. Finally, $p_V(u)$ denotes the set of property key-value pairs of the vertex u , $p_E(e)$ denotes the set of property key-value pairs of the edge e . An example LPG is in Figure 4. All systems considered in this work use some variant of the LPG, with the exception of RDF systems or when explicitly discussed.

2.1.4 Variants of Labeled Property Graph Model. Several databases support variants of LPG. First, Neo4j [168] (a graph database described in detail in § 4.9.1) supports an arbitrary number of labels for vertices. However it only allows for one label, (called *edge-type*), per edge. Next, ArangoDB [11] (a graph database described in detail in § 4.5.2) only allows for one label per vertex (*vertex-type*) and one label per edge (*edge-type*). This facilitates the separation of vertices and edges into different document collections. Moreover, edge-labeled graphs [4] do not allow for any properties and use labels in a restricted way. Specifically, only edges have labels and each edge has exactly one label. Formally, $G = (V, E, L)$, where V is the set of vertices and $E \subseteq V \times L \times V$ is the set of edges. Note that this definition enables two vertices to be connected by multiple edges with different labels. Finally, some effort was dedicated to LPG variants that facilitate storing historical graph data [51].

2.1.5 Resource Description Framework (RDF). The *Resource Description Framework (RDF)* [59] is a collection of specifications for representing information. It was introduced by the World Wide Web Consortium (W3C) in 1999 and the latest version (1.1) of the RDF specification was published in 2014. Its goal is to enable a simple format that allows for easy data exchange between different formats of data. It is especially useful as a description of irregularly connected data. The core part of the RDF model is a collection of *triples*. Each triple consists of a *subject*, a *predicate*, and an *object*. Thus, RDF databases are also often called *triple stores* (or *triplestores*). Subjects can either be identifiers (called Uniform Resource Identifiers (URIs)) or blank nodes (which are dummy identifiers

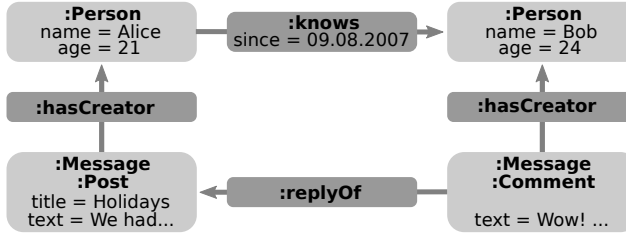


Fig. 4. **The illustration of an example Labeled Property Graph (LPG).** Vertices and edges can have labels (bold, prefixed with colon) and properties (key = value). We present a subgraph of a social network, where a person can know other persons, post messages, and comment on others' messages.

for internal use). Objects can be URIs, blank nodes, or literals (which are simple values). With triples, one can connect identifiers with identifiers or identifiers with literals. The connections are named with another URI (the predicate). RDF triples can be formally described as

$$(s, p, o) \in (URI \cup blank) \times (URI) \times (URI \cup blank \cup literal)$$

s represents a subject, p models a predicate, and o represents an object. *URI* is a set of Uniform Resource Identifiers; *blank* is a set of blank node identifiers, that substitute internally URIs to allow for more complex data structures; *literal* is a set of literal values [101, 154].

2.1.6 Transformations between LPG and RDF. To represent a Labeled Property Graph in the RDF model, LPG vertices are mapped to URIs (❶) and then RDF triples are used to link those vertices with their LPG properties by representing a property key and a property value with, respectively, an RDF predicate and an RDF object (❷). For example, for a vertex with an ID *vertex-id* and a corresponding property with a key *property-key* and a value *property-value*, one creates an RDF triple (*vertex-id*, *property-key*, *property-value*). Similarly, one can represent edges from the LPG graph model in the RDF model by giving each edge the URI status (❸), and by linking edge properties with specific edges analogously to vertices: (*edge-id*, *property-key*, *property-value*) (❹). Then, one has to use two triples to connect each edge to any of its adjacent vertices (❺). Finally, LPG labels can also be transformed into RDF triples in a way similar to that of properties [110], by creating RDF triples for vertices (❻) and edges (❼) such that the predicate becomes a “label” URI and contains the string name of this label. Figure 5 shows an example of transforming an LPG graph into RDF triples. More details on transformations between LPG and RDF are provided by Hartig [99].

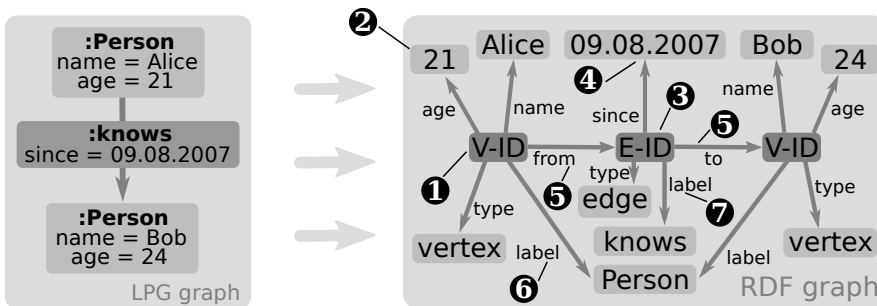


Fig. 5. **Comparison of an LPG and an RDF graph:** a transformation from LPG to RDF. “V-ID”, “E-ID”, “age”, “name”, “type”, “from”, “to”, “since” and “label” are RDF URIs. Numbers in black circles refer to transformation steps in § 2.1.6.

If all vertices and edges only have one label, one can omit the triples for labels and store the label (e.g., “Person”) *together with* the vertex or the edge name (“V-ID” and “E-ID”) in the identifier. We illustrate a corresponding example in Figure 6.

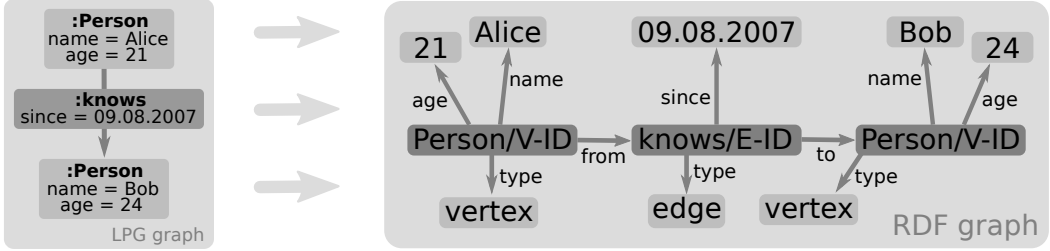


Fig. 6. **Comparison of an LPG and an RDF graph:** a transformation from LPG to RDF, given vertices and edges have only one label. “Person/V-ID”, “knows/E-ID”, “age”, “name”, “type”, “from”, “to” and “since” are RDF URIs.

Transforming RDF data into the LPG model is more complex, since RDF predicates, which would normally be translated into edges, are URIs. Thus, while deriving an LPG graph from an RDF graph, one must map edges to vertices and link such vertices, otherwise the resulting LPG graph may be disconnected. There are several schemes for such an RDF to LPG transformation, for example deriving an LPG graph which is bipartite, at the cost of an increased graph size [101]. Details and examples are provided in a report by Hayes [101].

2.2 Non-Graph Data Models and Storage Schemes Used in Graph Databases

In addition to the conceptual graph models, graph databases also often incorporate different storage schemes and data models that do not target specifically graphs but are used in various systems to model and store graphs. These models include *collections of key-value pairs*, *documents*, and *tuples* (used in different types of NoSQL stores), *relations and tables* (used in traditional relational databases), and *objects* (used in object-oriented databases). Different details of these models and the database systems based on them are described in other surveys, for example in a recent publication on NoSQL stores by Davoudian et al. [63]. Thus, we omit extensive discussions and instead offer brief summaries, *focusing on how they are used to model or represent graphs*.

2.2.1 Collection of Key-Value Pairs. Key-value stores are the simplest NoSQL stores [63]. Here, the data is stored as a collection of *key-value* pairs, with the focus on high-performance and highly-scalable lookups based on keys. The exact form of both keys and values depends on a specific system or an application. Keys can be simple (e.g., an URI or a hash) or structured. Values are often encoded as byte arrays (i.e., the structure of values is usually schema-less). However, a key-value store can also impose some additional data layout, structuring the schema-less values [63].

Due to the general nature of key-value stores, there can be many ways of representing a graph as a collection of KV values. We describe several concrete example systems [65, 108, 165, 177] in § 4.4. For example, one can use vertex labels as keys and encode the neighborhoods of vertices as values.

2.2.2 Collection of Documents. A document is a fundamental storage unit in a class of NoSQL databases called document stores [63]. These documents are stored in collections. Multiple collections of documents constitute a database. A document is encoded using a selected standard semi-structured format, e.g., JSON [44] or XML [45]. Document stores extend key-value stores in that a document can be seen as a value that has a certain flexible *schema*. This schema consists of *attributes*, where each attribute has a *name* along with one or more *values*. Such a structure based

on documents with attributes allows for various value types, key-value pair storage, and recursive data storage (attribute values can be lists or key-value dictionaries).

In all surveyed document stores [11, 47, 80, 125, 142] (§ 4.5), each vertex is stored in a vertex document. The capability of documents to store key-value pairs is used to store vertex labels and properties within the corresponding vertex document. The details of edge storage, however, is system-dependent: edges can be stored in the document corresponding to the source vertex of each edge, or in the documents of the destination vertices. As documents do not impose any restriction on what key-value pairs can be stored, vertices and edges may have different sets of properties.

2.2.3 Collection of Tuples. Tuples are a basis of NoSQL stores called tuple stores. A tuple store generalizes an RDF store: RDF stores are restricted to triples (or – in some cases – 4-tuples, also referred to as *quads*) whereas tuple stores can contain *tuples of an arbitrary size*. Thus, the number of elements in a tuple is not fixed and can vary, even within a single database. Each tuple has an ID which may also be a direct memory pointer.

A collection of tuples can model a graph in different ways. For example, one tuple of size n can store pointers to other tuples that contain neighborhoods of vertices. The exact mapping between such tuples and graph data is specific to different databases; we describe an example [199] in § 4.3.

2.2.4 Collection of Tables. Tables are the basis of Relational Database Management Systems (RDBMS) [15, 57, 102]. Tables consist of rows and columns. Each row represents a single data element, for example a car. A single column usually defines a certain data attribute, for example the color of a car. Some columns can define unique IDs of data elements, called *primary keys*. Primary keys can be used to implement relations between data elements. A one-to-one or a one-to-many relation can be implemented with a single additional column that contains the copy of a primary key of the related data element (such primary key copy is called the *foreign key*). A many-to-many relation can be implemented with a dedicated table containing foreign keys of related data elements.

To model a graph as a collection of tables, one can implement vertices and edges as rows in two separate tables. Each vertex has a unique primary key that constitutes its ID. Edges can relate to their source or destination vertices by referring to their primary keys (as foreign keys). LPG labels and properties, as well as RDF predicates, can be modeled with additional columns [200, 203]. We present and analyze different graph database systems [16, 152] based on tables in § 4.6 and § 4.7.

2.2.5 Collection of Objects. One can also use collections of objects in Object-Oriented Database Management Systems (OODBMS) [14] to model graphs. Here, data elements and their relations are implemented as objects linked with some form of pointers. The details of modeling graphs as objects heavily depend on specific designs. We provide details for an example system [198] in § 4.8.

3 TAXONOMY OF GRAPH DATABASE SYSTEMS

We now describe how we categorize graph database systems considered in this survey [2, 9, 11, 12, 16, 39, 47, 49, 61, 65, 80, 82, 89, 108, 116, 125, 133, 134, 142, 146, 150, 151, 161, 165–168, 177, 191, 198–200, 202].

3.1 Taxonomy Structure

We first outline and motivate the proposed taxonomy. A primary way to group systems is by their **general backend** type (e.g., a triple store or a document store). This facilitates further taxonomization and analysis of graph databases because (1) the backend design has a profound impact on almost all other aspects of a graph database such as data organization, and because (2) it straightforwardly enables categorizing all considered graph databases into a few clearly defined groups.

After identifying the general types of backends, we further consider:

- Supported **conceptual graph data models and representations** (§ 3.3). Here, we identify fundamental approaches towards modeling the maintained graph dataset, and towards representing the structure of this graph (i.e., neighborhoods of each vertex). The used graph model strongly influences what graph query languages can be used together with a given system, and it also has impact on the associated data layout. Moreover, the used graph representation directly impacts the performance of different graph queries.
- Details and optimizations of **data organization** (§ 3.4). Here, we identify different optimizations in the data organization. These optimizations provide more insights into the details of how a given graph database maintains its graph dataset.
- Supported modes for **data distribution** (§ 3.5). We identify whether a database can run in a distributed mode, and if yes, if it supports data replication or sharding. This information facilitates selecting a system with the most appropriate performance properties in a given context. For example, systems that replicate but not shard the data, may offer more performance for read only workloads, but may fail to scale well for particularly large graphs that would require disk spilling.
- Finally, we also offer insights into the support for concurrent and/or parallel **query execution** (§ 3.6), **transaction types** (§ 3.7), and **supported query languages** (§ 3.9). This enables deriving certain insights on the performance of the studied systems, e.g., parallelization of queries suggests that queries may scale well in a given database. Unfortunately, almost all of the studied graph databases are closed source or do not come with any associated discussions on the details of their query and transaction execution (except for general descriptions). Thus, we do not offer a detailed associated taxonomy for algorithmic aspects of query and transaction execution, beyond the above criteria. However, we provide a detailed associated discussion on a few systems that do come with more details on their query execution. Moreover, we analyze the correlations between the backend type and data model vs. the support for transactions, query parallelization, and supported query languages. This enables deriving certain insights about the design of different backends. For example, the query language support is primarily affected by the supported conceptual graph model; if it is RDF, then the system usually supports SPARQL while systems focusing on LPG usually support Cypher or Gremlin.

Figure 7 illustrates the general types of considered databases together with certain aspects of data models and organization. Figure 8 summarizes all elements of the proposed taxonomy.

3.2 Types of Graph Database Storage Backends

We first identify **general types of graph databases** that primarily differ in their **storage backends**. First, some classes of systems use a certain *specific backend* technology, adapting this backend to storing graph data, and adding a *frontend* to query the graph data. Examples of such systems are **tuple stores**, **document stores**, **key-value stores**, **wide-column stores**, **Relational Database Management Systems (RDBMS)**, or **Object-Oriented Database Management Systems (OODBMS)**. Other graph databases are designed *specifically* for maintaining and querying graphs; we call such systems **native graph databases** (or **native graph stores**), they are based on either the **LPG** or the **RDF** graph data model. Finally, we consider designs called the **data hubs**; they enable using *many different* storage backends, facilitating storing data in different formats and models.

Some of the above categories of systems fall into the domain of NoSQL stores. For example, this includes document stores, key-value stores, or some triple stores. However, there is no strict assignment of specific storage backends as NoSQL. For example, triple stores can also be implemented as, e.g., RDBMS [63]. Figure 7 illustrates these systems, they are discussed in more detail in Section 4.

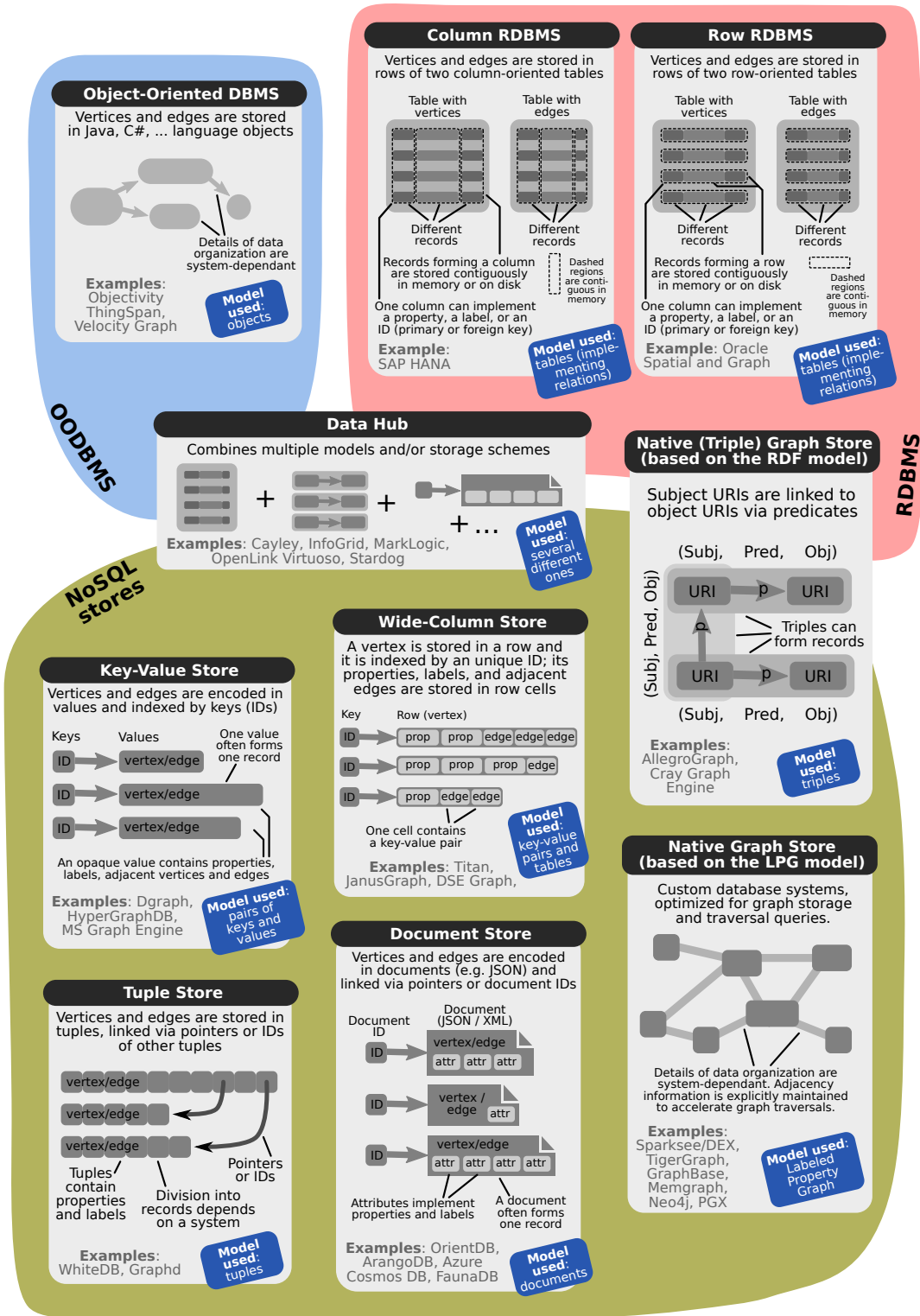


Fig. 7. Overview of different categories of graph database systems, with examples.

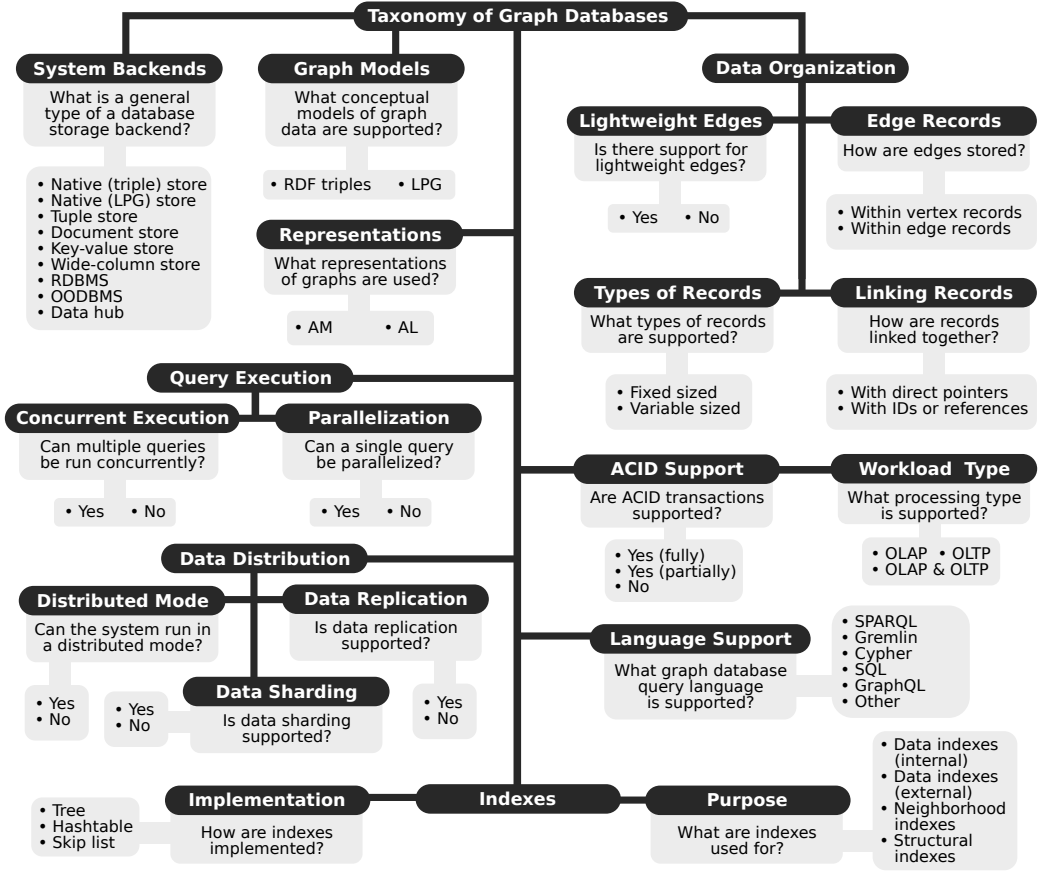


Fig. 8. Overview of the identified taxonomy of graph databases.

3.3 Conceptual Graph Models

We also investigate what conceptual data models are supported by different graph databases. Here, we focus on the RDF and LPG models as well as their variants, described in § 2.1. In addition, we call a system *Multi Model* if it allows for more than one data model, for example when it directly supports both LPG and RDF. Finally, we also indicate whether the graph structure is stored using the AL or the AM representation of a simple graph model.

3.4 Details and Optimizations of Data Organization

Next, while surveying databases, we consider different aspects of data organization. This part of the taxonomy provides more insights into the fundamental graph database backend types. We provide an analysis of this part in § 4.11.

3.4.1 Dividing Data into Records. Graph databases usually organize data into small units called *records*. One record contains information about a certain single entity (e.g., a person), this information is organized into specified logical fields (e.g., a name, a surname, etc.). A certain number of records is often kept together in one contiguous block in memory or disk to enhance data access locality.

The details of record-based data organization heavily depend on a specific system. For example, a relational database could treat a table row as a record, key-value stores often maintain a single value in a single record, while in document stores, a single document could be a record. Importantly, some systems allow *variable sized* records (e.g., ArangoDB), others only enable *fixed sized* records (e.g., Neo4j). Finally, we observe that while some systems (e.g., some triple stores such as Cray Graph Engine) do not explicitly mention records, the data could still be implicitly organized in a record-based way. In triple stores, one would naturally associate a triple with a record.

Graph databases often use one or more records per vertex (these records are sometimes referred to as *vertex records*). Neo4j uses multiple fixed-size records for vertices, while document databases use one document per vertex (e.g., ArangoDB). Edges are sometimes stored in the same record together with the associated (source or destination) vertices (e.g., Titan or JanusGraph). Otherwise, edges are stored in separate *edge records* (e.g., ArangoDB).

3.4.2 Storing Data in Index Structures. Graph databases commonly use indexes to speed up queries. Now, systems based on non-graph backends, for example RDBMS or document stores, usually rely on existing indexing infrastructure present in such systems. Native graph databases employ index structures for the neighborhoods of each vertex, often in the form of direct pointers [168].

In addition to using index structures to maintain the locations of data, **some databases also store the graph data in the indexes themselves.** In such cases, the index does not point to a certain data record but the index itself contains the desired data. Example systems with such functionality are Sparksee/DEX and Cray Graph Engine. **To maintain indices, the former uses bitmaps and B+ trees while the latter uses hash tables.**

3.4.3 Enabling Lightweight Edges. Some systems (e.g., OrientDB) allow edges without labels or properties to be stored as *lightweight edges*. Such edges are stored in the records of the corresponding source and/or destination vertices. These lightweight edges are represented by the ID of their destination vertex, or by a pointer to this vertex. This can save storage space and accelerate resolving different graph queries such as verifying connectivity of two vertices [48].

3.4.4 Linking Records with Direct Pointers. In record based systems, vertices and edges are stored in records. To enable efficient resolution of connectivity queries (i.e., verifying whether two vertices are connected), these records have to point to other records. One option is to store *direct pointers* (i.e., memory addresses) to the respective connected records. For example, an edge record can store direct pointers to vertex records with adjacent vertices. Another option is to assign each record a unique ID and use these IDs instead of direct pointers to refer to other records. On one hand, this requires an additional indexing structure to find the physical location of a record based on its ID. On the other hand, if the physical location changes, it is usually easier to update the indexing structure instead of changing all associated direct pointers.

A given system can also use *direct pointers* to avoid maintaining an additional dedicated indexing structure to traverse the graph. Note that an index may still be used to *find* a vertex; using direct pointers in this context means that only the structure of the adjacency data has no additional index. Using direct pointers can accelerate graph traversals [168], as additional index traversals are avoided. However, when the adjacency data needs to be updated, usually a large number of pointers need to be updated as well, generating additional overhead [12].

3.5 Data Distribution

A system is *distributed* or *multi-server* if it can run on multiple servers (also called compute nodes) connected with a network. In such systems, data may be *replicated* [84] (maintaining copies of the dataset at each server), or it may allow for *sharding* [77] (data fragmentation, i.e., storing only a part

of the given dataset on one server). Replication often allows for more fault tolerance [76], sharding reduces the amount of used memory per node and can improve performance [76]. In § 4.11.3, we correlate the support for data distribution with different fundamental backend types.

3.6 Query Execution

We define *concurrent execution* as the execution of separate queries at the same time. Concurrent execution of queries can lead to higher throughput. We also define *parallel execution* as the parallelized execution of a single query, possibly on more than one server or compute node. Parallel execution can lead to lower latencies for queries that can be parallelized. In § 4.11.5, we correlate the support for concurrent and parallel queries with different fundamental backend types, and we describe the details of query execution in graph databases that disclose this information.

3.7 Support for Transactions

Many graph databases support *transactions*; we analyze them in § 4.11.6. *ACID* [103] (Atomicity, Consistency, Isolation, Durability) is a well-known set of properties that database transactions uphold in many database systems. Different graph databases explicitly ensure some or all of ACID.

3.8 Support for OLTP vs. OLAP

Some databases (e.g., ArangoDB [11]) are oriented towards the *Online Transaction Processing* (OLTP), where focus is on executing many smaller, interactive, transactional queries. Other systems (e.g., Cray Graph Engine [166]) focus more on the *Online Analytics Processing* (OLAP): they execute analytics queries that span the whole graphs, usually taking more time than OLTP operations. Analytics queries are often parallelized to minimize their latency. Finally, different databases (e.g., Neo4j [168]) offer extensive support for both. We analyze this in § 5.6.




3.9 Query Language Support

Although we do not focus on graph database languages, we report which query languages are supported by each considered graph database system (details are in § 5.6). We consider the leading languages such as SPARQL [157], Gremlin [169], Cypher [81, 91, 104], and SQL [62]. We also mention other system-specific languages such as GraphQL [100] and support for APIs from languages such as C++ or Java². Note that mapping graph queries to SQL was also addressed in past work [185].

3.10 Harnessing Index Structures

We also analyze how graph databases use indexes to accelerate accessing data. Here, we consider (1) the functionality (i.e., the use case) of a given index, and (2) how a given index is implemented. We do not include the index information in Tables table 2–table 3 because of lack of space, and instead provide a detailed separate analysis in § 4.11.7.

4 DATABASE SYSTEMS

We survey and describe selected graph database systems with respect to the proposed taxonomy. In each system category, we describe selected representative systems, focusing on the associated graph model, as well as data and storage organization. Tables 2 and 3 illustrate the details of different graph database systems, including the ones described in this section³. The tables indicate which features are supported by which systems. We use symbols “”, “”, and “” to indicate that a

²We bring to the reader’s attention a manifesto on creating GQL, a standardized graph query language (<https://gql.today>).




³We encourage participation in this survey. In case the reader is in possession of additional information relevant for the tables, the authors would welcome the input.

Graph Database System	oB	Model		Repr.	Data Organization						Data Distribution & Query Execution							Additional remarks			
		lpg	rdf	al	am	fs	vs	dp	se	sv	lw	ms	rp	sh	ce	pe	tr		oltp	olap	
NATIVE GRAPH DATABASES (RDF model based, triple stores) (§ 4.2). The main data model used: RDF triples (§ 2.1.5).																					
AllegroGraph [82]	×	×	☐	×	×	×	☐	×	×	×	×	×	☐	☐	☐	×	×	☐	☐	?	*Triples are stored as integers (RDF strings map to integers).
BlazeGraph [39]	×	☐	☐	×	×	?	?	×	×	×	×	×	☐	☐	☐	?	?	☐	?	?	*BlazeGraph uses RDF*, an extension of RDF (details in § 4.2).
Cray Graph Engine [166]	×	×	☐	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	*RDF triples are stored in hashtables.
Amazon Neptune [2]	×	☐	☐	×	×	?	?	×	×	×	×	×	×	×	×	×	×	☐	☐	☐	—
AnzoGraph [49]	×	☐	☐	×	×	?	?	×	×	×	×	×	×	×	×	×	×	☐	☐	☐	—
Apache Jena TBD [190]	×	×	☐	?	×	?	?	?	?	?	?	?	×	?	×	×	×	×	?	?	—
Apache Marmotta [9]	×	×	☐	×	×	☐	×	×	×	×	×	×	?	?	?	×	×	☐	☐	☐	*The structure of data records is based on that of different RDBMS systems (H2 [145], PostgreSQL [144], MySQL [71]).
BrightstarDB [146]	×	×	☐	×	×	?	?	×	×	×	×	×	?	?	?	×	?	☐	☐	?	—
gStore [205]	×	×	☐	×	×	×	☐	×	×	×	×	×	?	?	×	×	?	?	?	?	—
Ontotext GraphDB [150]	×	×	☐	×	×	?	?	×	×	×	×	×	×	×	×	×	?	☐	☐	?	—
Profium Sense [161]	×	×	☐	×	×	?	?	×	×	×	×	×	×	?	×	×	?	☐	☐	?	*The format used is called JSON-LD: JSON for vertices and RDF for edges.
TripleBit [202]	×	×	☐	×	×	×	☐	×	×	×	×	×	×	×	×	×	×	?	?	☐	The data organization uses compression.
																					*Strings map to variable size integers.
																					‡Described as future work.
NATIVE GRAPH DATABASES (LPG model based) (§ 4.9). The main data model used: LPG (§ 2.1.3, § 2.1.4).																					
Neo4j [168]	×	☐	×	×	×	×	☐	×	×	×	×	×	×	×	×	×	×	×	×	×	Neo4j is provided as a cloud service by a system called Graph Story [90].
Sparksee/DEX [134]	×	☐	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	*Bitmaps are used for connectivity.
																					‡The system uses maps only.
GBase [116]	×	×	×	×	×	×	×	×	×	×	×	×	×	?	?	?	?	?	?	×	*GBase supports simple graphs only (§ 2.1.1).
																					‡GBase stores the AM sparsely.
GraphBase [79]	×	×	×	?	×	×	☐	?	?	?	?	?	?	?	☐	×	?	☐	☐	?	*No support for edge properties, only two types of edges available.
Graphflow [117]	×	☐	×	×	×	?	?	?	?	?	?	×	?	?	?	?	?	?	?	?	—
LiveGraph [204]	×	☐	×	×	×	×	☐	×	×	×	×	×	?	×	×	×	?	?	?	?	—
Memgraph [139]	×	☐	×	×	×	?	?	?	?	?	?	?	?	?	×	×	×	×	×	×	*This feature is under development.
																					‡Available only for some algorithms.
TigerGraph [193]	×	☐	×	?	×	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	—
Weaver [70]	×	☐	×	?	×	?	?	?	?	?	?	?	☐	☐	☐	×	×	×	×	×	—
KEY-VALUE STORES (§ 4.4). The main data model used: key-value pairs (§ 2.2.1).																					
HyperGraphDB [108]	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	☐	†	☐	*A Hypergraph model. ‡The system uses an incidence index to retrieve edges of a vertex. †Support for ACI only.
MS Graph Engine [177]	☐	☐	×	☐	×	×	☐	†	×	×	×	×	×	×	×	×	×	×	×	×	*AL contains IDs of edges and/or vertices.
																					‡Schema is defined by Trinity Specification Language (TSL).
Dgraph [65]	×	☐	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	Dgraph is based on Badger [64].
RedisGraph [162, 165, 181]	×	☐	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	RedisGraph is based on Redis [164].
																					*The OLAP part uses GraphBLAS [119].
DOCUMENT STORES (§ 4.5). The main data model used: documents (§ 2.2.2).																					
ArangoDB [11]	☐	☐	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	☐	*Uses a hybrid index for retrieving edges.
OrientDB [47]	☐	☐	×	☐	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	*AL contains RIDs (i.e., physical locations) of edge and vertex records. ‡Sharding is user defined. OrientDB supports JSON and it offers certain object-oriented capabilities.
Azure Cosmos DB [142]	☐	☐	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	?	—
Bitsy [125]	×	☐	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	The system is disk based and uses JSON files. The storage only allows for appending data.
FaunaDB [80]	☐	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	*Document, RDBMS, graph, “time series”.
																					‡Adjacency lists are separately precomputed.

Table 2. **Comparison of graph databases.** **Bolded systems** are described in more detail in the corresponding sections.

oB: A system supports secondary data models / backend types (in addition to its primary one). **lpg, rdf:** A system supports, respectively, the **Labeled Property Graph** and **RDF** without prior data transformation. **am, al:** The structure is represented as the **adjacency matrix** or the **adjacency list**. **fs, vs:** Data records are **fixed size** and **variable size**, respectively. **dp:** A system can use **direct pointers** to link records. This enables storing and traversing adjacency data without maintaining indices. **se:** Edges can be **stored in a separate edge record**. **sv:** Edges can be **stored in a vertex record**. **lw:** Edges can be **lightweight** (containing just a vertex ID or a pointer, both stored in a vertex record). **ms:** A system can operate in a **Multi Server** (distributed) mode. **rp:** Given a distributed mode, a system enables **Replication** of datasets. **sh:** Given a distributed mode, a system enables **Sharding** of datasets. **ce:** Given a distributed mode, a system enables **Concurrent Execution** of multiple queries. **pe:** Given a distributed mode, a system enables **Parallel Execution** of single queries on multiple nodes/CPUs. **tr:** Support for **ACID Transactions**. **oltp:** Support for **Online Transaction Processing**. **olap:** Support for **Online Analytical Processing**. **☐:** A system offers a given feature. **☐:** A system offers a given feature in a limited way. **×**: A system does not offer a given feature. **?**: Unknown.

Graph Database System	oB	Model		Repr.	Data Organization						Data Distribution & Query Execution								Additional remarks		
		lpg	rdf		al	am	fs	vs	dp	se	sv	lw	ms	rp	sh	ce	pe	tr		oltp	olap
RELATIONAL DBMS (RDBMS) (§ 4.7). The main data model used: tables (implementing relations) (§ 2.2.4).																					
Oracle Spatial and Graph [152]																					* LPG and RDF use row-oriented storage. The system can also run on top of PGX [105] (effectively as a native graph database). AgensGraph is based on PostgreSQL. The system focuses on “shallow” graph queries, such as finding mutual friends.
AgensGraph [38]																					The system focuses on “shallow” graph queries, such as finding mutual friends.
FlockDB [194]																					* can store vertices/edges in the same table. ‡ inherited from the underlying IBM Db2™. The system uses an SQL graph extension.
IBM Db2 Graph [192]																					
MS SQL Server 2017 [143]																					
OQGRAPH [132]																					OQGRAPH uses MariaDB [19]. * OQGRAPH uses row-oriented storage.
SAP HANA [174]																					SAP HANA is column-oriented, edges and vertices are stored in rows. SAP HANA can be used with a dedicated graph engine [172]; it offers some capabilities of a JSON document store [174]
SQLGraph [186]																					* SQLGraph uses JSON for property storage. ‡ SQLGraph uses row-oriented storage. † depends on the used SQL engine.
WIDE-COLUMN STORES (§ 4.6). The main data model used: key-value pairs and tables (§ 2.2.1, § 2.2.4).																					
JanusGraph [16]																					JanusGraph is the continuation of Titan. Enables various backends (e.g., Cassandra [124]).
Titan [16]																					DSE Graph is based on Cassandra [124]. * Support for AID, Consistency is configurable.
DSE Graph (DataStax) [61]																					HGraphDB uses TinkerPop3 with HBase [85]. * ACID is supported only within a row.
HGraphDB [167]																					
TUPLE STORES (§ 4.3). The main data model used: tuples (§ 2.2.3).																					
WhiteDB [199]																					* Implicit support for triples of integers. ‡ Implementable by the user. † Transactions use a global shared/exclusive lock.
Graphd [89]																					Backend of Google Freebase. * Implicit support for triples. ‡ Subset of ACID.
OBJECT-ORIENTED DATABASES (OODBMS) (§ 4.8). The main data model used: objects (§ 2.2.5).																					
Velocity-Graph [198]																					The system is based on VelocityDB [197]
Objectivity ThingSpan [148]																					The system is based on ObjectivityDB [93].
DATA HUBS (§ 4.10). The main data model used: several different ones .																					
MarkLogic [133]																					Supported storage/models: relational tables, RDF, various documents. * Vertices are stored as documents, edges are stored as RDF triples.
OpenLink Virtuoso [151]																					Supported storage/models: relational tables and RDF triples. * This feature can be used relational data only.
Cayley [52]																					Supported storage/models: relational tables, RDF, document, key-value. * This feature depends on the backend.
InfoGrid [107]																					Supported storage/models: relational tables, Hadoop's filesystem, grid storage. * A weaker consistency model is used instead of ACID.
Stardog [184]																					Supported storage/models: relational tables, documents. * RDF is simulated on relational tables. Both LPG and RDF are enabled through virtual quints.

Table 3. **Comparison of graph databases.** **Bolded systems** are described in more detail in the corresponding sections. **oB**: A system supports secondary data models / backend types (in addition to its primary one). **lpg, rdf**: A system supports, respectively, the **Labeled Property Graph** and **RDF** without prior data transformation. **am, al**: The structure is represented as the **adjacency matrix** or the **adjacency list**. **fs, vs**: Data records are **fixed size** and **variable size**, respectively. **dp**: A system can use **direct pointers** to link records. This enables storing and traversing adjacency data without maintaining indices. **se**: Edges can be **stored in a separate edge record**. **sv**: Edges can be **stored in a vertex record**. **lw**: Edges can be **lightweight** (containing just a vertex ID or a pointer, both stored in a vertex record). **ms**: A system can operate in a **Multi Server** (distributed) mode. **rp**: Given a distributed mode, a system enables **Replication** of datasets. **sh**: Given a distributed mode, a system enables **Sharding** of datasets. **ce**: Given a distributed mode, a system enables **Concurrent Execution** of multiple queries. **pe**: Given a distributed mode, a system enables **Parallel Execution** of single queries on multiple nodes/CPUs. **tr**: Support for **ACID Transactions**. **oltp**: Support for **Online Transaction Processing**. **olap**: Support for **Online Analytical Processing**. : A system offers a given feature. : A system offers a given feature in a limited way. **x**: A system does not offer a given feature. : Unknown.

Graph Database System	Graph database query language						Other languages and additional remarks
	SPARQL	Gremlin	Cypher	SQL	GraphQL	Progr. API	
NATIVE GRAPH DATABASES (RDF model based, triple stores) (§ 4.2).							
AllegroGraph							
Amazon Neptune							
AnzoGraph							
Apache Jena TDB							
Apache Marmotta							Apache Marmotta also supports its native LDP and LDPATH languages.
BlazeGraph							*BlazeGraph offers SPARQL* to query RDF*.
BrightstarDB							
Cray Graph Engine							
gStore							
Ontotext GraphDB							
Proform Sense							
TripleBit							
NATIVE GRAPH DATABASES (LPG model based) (§ 4.9).							
Gbase							
GraphBase							GraphBase uses its native query language.
Graphflow							*Graphflow supports a subset of Cypher [141]. ‡Graphflow supports Cypher++ extension with subgraph-condition-action triggers [117].
LiveGraph							No focus on languages and queries.
Mengraph							*openCypher.
Neo4j							*Gremlin is supported as a part of TinkerPop integration.
Sparksee/DEX							‡GraphQL supported with the GRANDstack layer.
TigerGraph							†Neo4j can be embedded in Java applications.
Weaver							*Sparksee/DEX also supports C++, Python, Objective-C, and Java APIs.
							TigerGraph uses GSQL [193].
							*Weaver also supports C++, Python.
TUPLE STORES (§ 4.3).							
Graphd							Graphd uses MQL [89].
WhiteDB							*WhiteDB also supports Python.
DOCUMENT STORES (§ 4.5).							
ArangoDB							ArangoDB uses AQL (ArangoDB Query Language).
Azure Cosmos DB							
Bitsy							Bitsy also supports other Tinkerpop-compatible languages such as SQL2Gremlin and Pixy.
FaunaDB							
OrientDB							*An SQL extension for graph queries. ‡OrientDB offers bindings to C, JavaScript, PHP, .NET, Python, and others.
KEY-VALUE STORES (§ 4.4).							
Dgraph							*A variant of GraphQL.
HyperGraphDB							
MS Graph Engine							MS Graph Engine uses LINQ [177].
RedisGraph							
WIDE-COLUMN STORES (§ 4.6).							
DSE Graph (DataStax)							DSE Graph also supports CQL [61].
HGraphDB							
JanusGraph							
Titan							
RELATIONAL DBMS (RDBMS) (§ 4.7).							
AgensGraph							*A variant called openCypher [92, 135]. ‡ANSI-SQL.
FlockDB							FlockDB uses the Gizzard framework and MySQL.
IBM Db2 Graph							*IBM Db2 Graph supports only graph queries which results can be returned to rows. ‡IBM Db2 Graph also supports Scala, Python and Groovy.
MS SQL Server 2017							*Transact-SQL.
OQGRAPH							
Oracle Spatial and Graph							*PGQL [196], an SQL-like graph query language.
SAP HANA							*SAP HANA offers bindings to Rust, ODBC, and others.
SQLGraph							‡GraphScript, a domain-specific graph query language.
							*SQLGraph doesn't support Gremlin side effect pipes.
							‡Graph is encoded in a way specific to SQLGraph.
OBJECT-ORIENTED DATABASES (OODBMS) (§ 4.8).							
Objectivity ThingSpan							Objectivity ThingSpan uses a native DO query language [148].
VelocityGraph							
DATA HUBS (§ 4.10).							
Cayley							*Cayley supports Gizmo, a Gremlin dialect [52].
InfoGrid							Cayley also uses MQL [52].
MarkLogic							MarkLogic uses XQuery [40].
OpenLink Virtuoso							OpenLink Virtuoso also supports XQuery [40], XPath v1.0 [56], and XSLT v1.0 [118].
Stardog							*Stardog supports the Path Query extension [184].

Table 4. Support for different graph database query languages in different graph database systems. “Progr. API” determines whether a given system supports formulating queries using some native programming language such as C++.

“”: A system supports a given language. “”: A system supports a given language in a limited way. “”: A system does not support a given language.

given system offers a given feature, offers a given feature in a limited way, and does not offer a given feature, respectively. “?” indicates we were unable to infer this information based on the available documentation. We report the support for different graph query languages in Table 4. Finally, we analyze different taxonomy aspects in § 4.11 and § 5.6.

4.1 Discussion on Selection Criteria

When selecting systems for consideration in the survey, we use two criteria. First, we use the DB-Engines Ranking⁴ to select the most popular systems in each considered backend category. We also pick interesting research systems (e.g., SQLGraph [186], LiveGraph [204], or Weaver [70]) which are not included in this ranking. For detailed discussions, we also consider the availability of technical details (i.e., most systems are closed source or do not offer any design details).

4.2 RDF Stores (Triple Stores)

RDF stores, also called triple stores, implement the Resource Description Framework (RDF) model (§ 2.1.5). These systems organize data into triples. We now describe in more detail a selected recent RDF store, Cray Graph Engine (§ 4.2.1). We also provide more details on two other systems, AllegroGraph and BlazeGraph, focusing on *variants of the RDF model* used in these systems (§ 4.2.2).

4.2.1 Cray Graph Engine. Cray Graph Engine (CGE) [166] is a triple store that can scale to a trillion RDF triples. CGE does not store triples but *quads* (4-tuples), where the fourth element is a graph ID. Thus, one can store multiple graphs in one CGE database. Quads in CGE are grouped by their predicate and the identifier of the graph that they are a part of. Thus, only a pair with a subject and an object needs to be stored for one such group of quads. These subject/object pairs are stored in hashtables (one hashtable per group). Since each subject and object is represented as a unique 48-bit integer identifier (HURI), the subject/object pairs can be packed into 12 bytes and stored in a 32-bit unsigned integer array, ultimately reducing the amount of needed storage.

4.2.2 AllegroGraph and BlazeGraph. There exist many other RDF graph databases. We briefly describe two systems that extend the original RDF model: AllegroGraph and BlazeGraph.

First, some RDF stores allow for attaching *attributes* to a triple explicitly. AllegroGraph [82] allows an arbitrary set of attributes to be defined per triple when the triple is created. However, these attributes are immutable. Figure 9 presents an example RDF graph with such attributes. This figure uses the same LPG graph as in previous examples provided in Figure 5 and Figure 6, which contain example transformations from the LPG into the original RDF model.

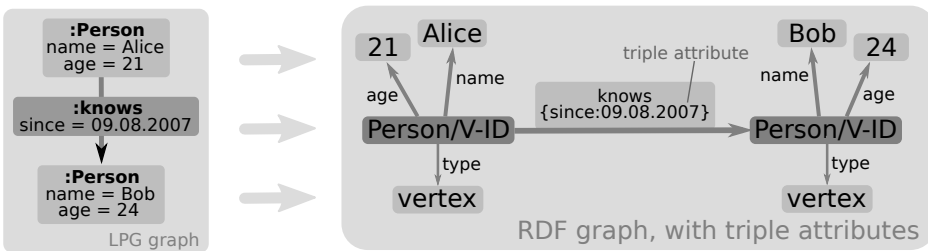


Fig. 9. **Comparison of an LPG graph and an RDF graph: a transformation from LPG to RDF with triple attributes.** We represent the triple attributes as a set of key-value pairs. “Person/V-ID”, “age”, “name”, “type” and “knows” are RDF URIs. The transformation uses the assumption that there is one label per vertex and edge.

⁴<https://db-engines.com/en/ranking/graph+dbms>

Second, BlazeGraph [39] implements RDF^* [97, 98], an augmentation of RDF that allows for attaching triples to triple predicates (see Figure 10). Vertices can use triples for storing labels and properties, analogously as with the plain RDF. However, with RDF^* , one can represent LPG edges more naturally than in the plain RDF. Specifically, edges can be stored as triples, and edge properties can be linked to the edge triple via other triples.

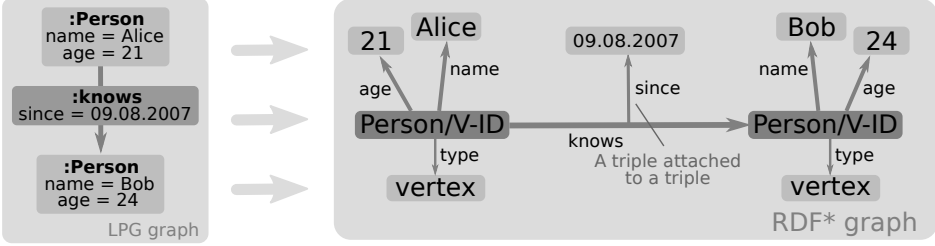


Fig. 10. **Comparison of an LPG graph and an RDF^* graph:** a transformation from LPG to RDF^* , that enables **attaching triples to triple predicates**. “Person/V-ID”, “age”, “name”, “type”, “since” and “knows” are RDF URIs. The transformation uses the assumption that there is one label per vertex and edge.

4.3 Tuple Stores

A tuple store is a generalization of an RDF store. RDF stores are restricted to triples (or quads, as in CGE) whereas tuple stores can maintain tuples of arbitrary sizes, as detailed in § 2.2.3.

4.3.1 WhiteDB. WhiteDB [199] is a tuple store that enables allocating new records (tuples) with an arbitrary tuple length (number of tuple elements). Small values and pointers to other tuples are stored directly in a given field. Large strings are kept in a separate store. Each large value is only stored once, and a reference counter keeps track of how many tuples refer to it at any time. WhiteDB only enables accessing single tuple records, there is no higher level query engine or graph API that would allow to, for example, execute a query that fetches all neighbors of a given vertex. However, one can use tuples as vertex and edge storage, linking them to one another via memory pointers. This facilitates fast resolution of various queries about the structure of an arbitrary irregular graph structure in WhiteDB. For example, one can store a vertex v with its properties as consecutive fields in a tuple associated with v , and maintain pointers to selected neighborhoods of v in v ’s tuple. More examples on using WhiteDB (and other tuple stores such as Graphd) for maintaining graph data can be found online [140, 199].

4.4 Key-Value Stores

One can also explicitly use key-value (KV) stores for maintaining a graph (cf. § 2.2.1). We provide details of using a collection of key-value pairs to model a graph in § 2.2.1. Here, we describe selected KV stores used as graph databases: MS Graph Engine (also called Trinity) and HyperGraphDB.

4.4.1 Microsoft’s Graph Engine (Trinity). Microsoft’s Graph Engine [177] is based on a distributed KV store called Trinity. Trinity implements a globally addressable distributed RAM storage. In Trinity, keys are called *cell IDs* and values are called *cells*. A cell can hold data items of different data types, including IDs of other cells. MS Graph Engine introduces a graph storage layer on top of the Trinity KV storage layer. Vertices are stored in cells, where a dedicated field contains a vertex ID or a hash of this ID. Edges adjacent to a given vertex v are stored as a list of IDs of v ’s neighboring

vertices, directly in v 's cell. However, if an edge holds rich data, such an edge (together with the associated data) can also be stored in a separate dedicated cell.

4.4.2 HyperGraphDB. HyperGraphDB [108] stores hypergraphs (definition in § 2.1.2). The basic building blocks of HyperGraphDB are *atoms*, the values of the KV store. Every *atom* has a cryptographically strong ID. This reduces a chance of collisions (i.e., creating identical IDs for different graph elements by different peers in a distributed environment). Both hypergraph vertices and hyperedges are atoms. Thus, they have their own unique IDs. An atom of a hyperedge stores a list of IDs corresponding to the vertices connected by this hyperedge. Vertices and hyperedges also have a *type ID* (i.e., a label ID) and they can store additional data (such as properties) in a recursive structure (referenced by a *value ID*). This recursive structure contains value IDs identifying other atoms (with other recursive structures) or binary data. Figure 11 shows an example of how a KV store is used to represent a hypergraph in HyperGraphDB.

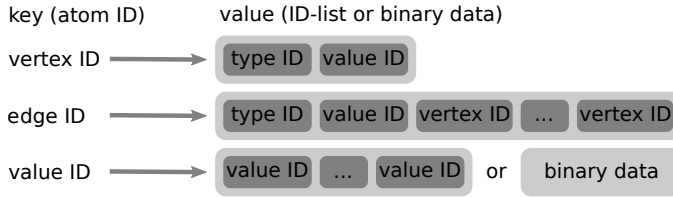


Fig. 11. An example utilization of key-value stores for maintaining hypergraphs in HyperGraphDB (a *type* is a term used in HyperGraphDB to refer to a label).

4.5 Document Stores

In document stores, a fundamental storage unit is a document, described in § 2.2.2. We select two document stores for a more detailed discussion, OrientDB and ArangoDB.

4.5.1 OrientDB. In OrientDB [47], every document d has a *Record ID (RID)*, consisting of the ID of the *collection of documents* where d is stored, and the *position* (also referred to as the *offset*) within this collection. Pointers (called *links*) between documents are represented using these unique RIDs.

OrientDB [47] introduces *regular edges* and *lightweight edges*. Regular edges are stored in an *edge document* and can have their own associated key/value pairs (e.g., to encode edge properties or labels). Lightweight edges, on the other hand, are stored directly in the document of the adjacent (source or destination) vertex. Such edges do not have any associated key/value pairs. They constitute simple pointers to other vertices, and they are implemented as document RIDs. Thus, a vertex document not only stores the labels and properties of the vertex, but also a list of lightweight edges (as a list of RIDs of the documents associated with neighboring vertices), and a list of pointers to the adjacent regular edges (as a list of RIDs of the documents associated with these regular edges). Each regular edge has pointers (RIDs) to the documents storing the source and the destination vertex. Each vertex stores a list of links (RIDs) to its incoming and the outgoing edges.

Figure 12 contains an example of using documents for representing vertices, regular edges, and lightweight edges in OrientDB. Figure 13 shows example vertex and edge documents.

4.5.2 ArangoDB. ArangoDB [11, 12] keeps its documents in a *binary format* called *VelocityPack*, which is a compacted implementation of JSON documents. Documents can be stored in different collections and have a `_key` attribute which is a unique ID within a given collection. Unlike OrientDB, these IDs are no direct memory pointers. For maintaining graphs, ArangoDB uses *vertex*

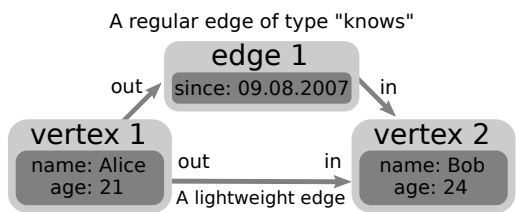


Fig. 12. Two vertex documents connected with a **lightweight edge** and a **regular edge (knows)** in OrientDB.

collections and *edge collections*. The former are regular document collections with vertex documents. Vertex documents store no information about adjacent edges. This has the advantage that a vertex document does not have to be modified when one adds or removes edges. Second, edge collections store edge documents. Edge documents have two particular properties: *_from* and *_to*, which are the IDs of the documents associated with two vertices connected by a given edge. An optimization in ArangoDB’s design prevents reading vertex documents and enables directly accessing one edge document based on the vertex ID *within another edge document*. This may improve cache efficiency and thus reduce query execution time [12].

One can use different collections of documents to store different edge types (e.g., “friend_of” or “likes”). When retrieving edges conditioned on some edge type (e.g., “friend_of”), one does not have to traverse the whole adjacency list (all “friend_of” and “likes” edges). Instead, one can target the collection with the edges of the specific edge type (“friend_of”).

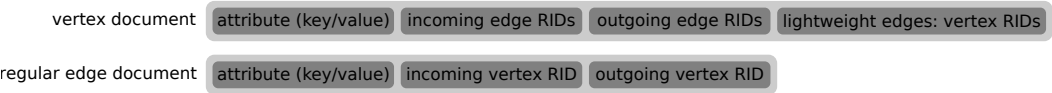


Fig. 13. Example OrientDB **vertex and edge documents (complex JSON documents are also supported)**.

4.6 Wide-Column Stores

Wide-column stores combine different features of key-value stores and relational tables. On one hand, a wide-column store maps keys to *rows* (a KV store that maps keys to values). Every row can have an arbitrary number of *cells* and every cell constitutes a key-value pair. Thus, a row contains a mapping of cell keys to cell values, effectively making a wide-column store a *two-dimensional KV store* (a row key and a cell key both identify a specific value). On the other hand, a wide-column store is a *table*, where cell keys constitute column names. However, unlike in a relational database, the names and the format of columns may differ between rows within the same table. We illustrate an example subset of rows and cells in a wide-column store in Figure 14.

4.6.1 *Titan and JanusGraph*. Titan [16] and its continuation JanusGraph [191] are built on top of wide-column stores. They can use different wide-column stores as backends, for example Apache Cassandra [7]. In both systems, when storing a graph, each row represents a vertex. Each vertex property and adjacent edge is stored in a separate cell. One edge is thus encoded in a single cell, including all the properties of this edge. Since cells in each row are sorted by the cell key, this sorting order can be used to find cells efficiently. For graphs, cell keys for properties and edges are chosen such that after sorting the cells, the cells storing properties come first, followed by the cells containing edges, see Figure 15. Since rows are ordered by the key, both systems straightforwardly partition tables into so called *tablets*, which can then be distributed over multiple data servers.

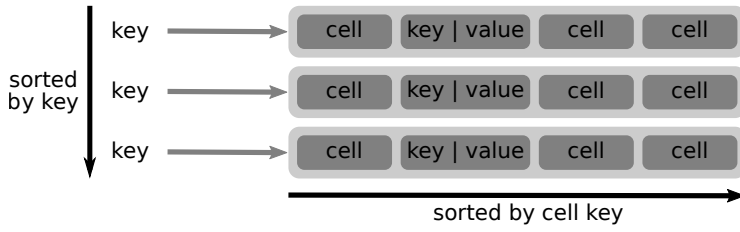


Fig. 14. **An illustration of wide-column stores:** mapping keys to rows and column-keys to cells within the rows.

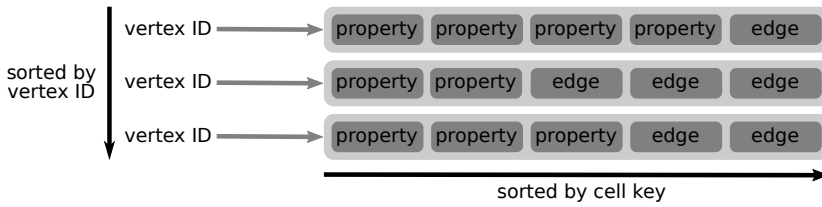


Fig. 15. **An illustration of Titan and JanusGraph:** using wide-column stores for storing graphs. The illustration is inspired by and adapted from [178].

4.7 Relational Database Management Systems

Relational Database Management Systems (RDBMS) store data in two dimensional *tables* with *rows* and *columns*, described in more detail in the corresponding data model section in § 2.2.4.

There are two types of RDBMS: *column* RDBMS (not to be confused with *wide-column* stores) and *row* RDBMS (also referred to as *column-oriented* or *columnar* and *row-oriented*). They differ in physical data persistence. Row RDBMS store table rows in consecutive memory blocks. Column RDBMS, on the other hand, store table columns contiguously. Row RDBMS are more efficient when only a few rows need to be retrieved, but with all their columns. Conversely, column RDBMS are more efficient when many rows need to be retrieved, but only with a few columns. Graph database solutions that use RDBMS as their backends use both row RDBMS (e.g., Oracle Spatial and Graph [152], OQGRAPH built on MariaDB [132]) and column RDBMS (e.g., SAP HANA [174]).

4.7.1 Oracle Spatial and Graph. Oracle Spatial and Graph [152] is built on top of Oracle Database. It provides a rich set of tools for administration and analysis of graph data. Oracle Spatial and Graph comes with a range of built-in parallel graph algorithms (e.g., for community detection, path finding, traversals, link prediction, PageRank, etc.). Both LPG and RDF models are supported. Rows of RDBMS tables constitute vertices and relationships between these rows form edges. Associated properties and attributes are stored as key-value pairs in separate structures.

4.8 Object-Oriented Databases

Object-oriented database management systems (OODBMS) [14] enable modeling, storing, and managing data in the form of *language objects* used in object-oriented programming languages. We summarize such objects in § 2.2.5.

4.8.1 VelocityGraph. VelocityGraph [198] is a graph database relying on the VelocityDB [197] distributed object database. VelocityGraph edges, vertices, as well as edge or vertex properties are stored in C# objects that contain references to other objects. To handle this structure, VelocityGraph introduces abstractions such as *VertexType*, *EdgeType*, and *PropertyType*. Each object has a unique

object identifier (Oid), pointing to its location in physical storage. Each vertex and edge has one type (label). Properties are stored in dictionaries. Vertices keep the adjacent edges in collections.

4.9 LPG-Based Native Graph Databases

Graph database systems described in the previous sections are all based on some database backend that was not originally built just for managing graphs. In what follows, we describe *LPG-based native graph databases*: systems that were specifically build to maintain and process graphs.

4.9.1 Neo4j: Direct Pointers. Neo4j [168] is the most popular graph database system, according to different database rankings (see the links on page 2). Neo4j implements the LPG model using a storage design based on fixed-size records. A vertex v is represented with a *vertex record*, which stores (1) v 's labels, (2) a pointer to a linked list of v 's properties, (3) a pointer to the first edge adjacent to v , and (4) some flags. An edge e is represented with an *edge record*, which stores (1) e 's edge type (a label), (2) a pointer to a linked list of e 's properties, (3) a pointer to two vertex records that represent vertices adjacent to e , (4) pointers to the ALs of both adjacent vertices, and (5) some flags. Each property record can store up to four properties, depending on the size of the property value. Large values (e.g., long strings) are stored in a separate *dynamic store*. Storing properties outside vertex and edge records allows those records to be small. Moreover, if no properties are accessed in a query, they are not loaded at all. The AL of a vertex is implemented as a doubly linked list. An edge is stored once, but is part of two such linked lists (one list for each adjacent vertex). Thus, an edge has two pointers to the previous edges and two pointers to the next edges. Figure 16 outlines the Neo4j design; Figure 17 shows the details of vertex and edge records.

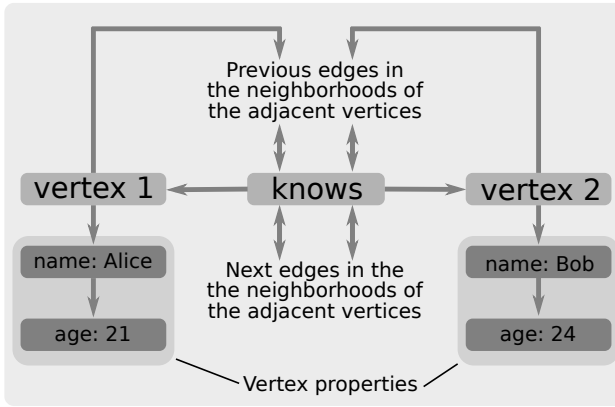


Fig. 16. **Summary of the Neo4j structure:** two vertices linked by a “knows” edge. Both vertices maintain linked lists of properties. The edges are part of two doubly linked lists, one linked list per adjacent vertex.

A core concept in Neo4j is using *direct pointers* [168]: a vertex stores pointers to the physical locations of its neighbors. Thus, for neighborhood queries or traversals, one needs no index and can instead follow direct pointers (except for the root vertices in traversals). Consequently, the query complexity does not dependent on the graph size. Instead, it only depends on how large the visited subgraph is⁵.

⁵That said, if the graph does not fit into the main memory, the execution speed heavily depends on caching and cache pre-warming, i.e., the running time may significantly increase

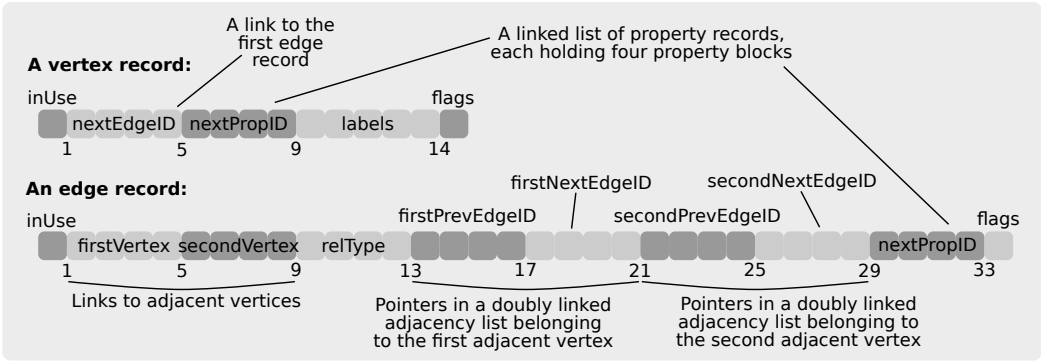


Fig. 17. An overview of the Neo4j vertex and edge records.

4.9.2 Sparksee/DEX: B+ Trees and Bitmaps. Sparksee is a graph database system that was formerly known as DEX [134]. Sparksee implements the LPG model in the following way. Vertices and edges (both are called objects) are identified by unique IDs. For each property name, there is an associated B+ tree that maps vertex and edge IDs to the respective property values. The reverse mapping from a property value to vertex and edge IDs is maintained by a bitmap, where a bit set to one indicates that the corresponding ID has some property value. Labels and vertices and edges are mapped to each other in a similar way. Moreover, for each vertex, two bitmaps are stored: One bitmap indicates the incoming edges, and another one the outgoing edges. Furthermore, two B+ trees maintain the information about what vertices an edge is connected to (one tree for each edge direction). Figure 18 illustrates example mappings.

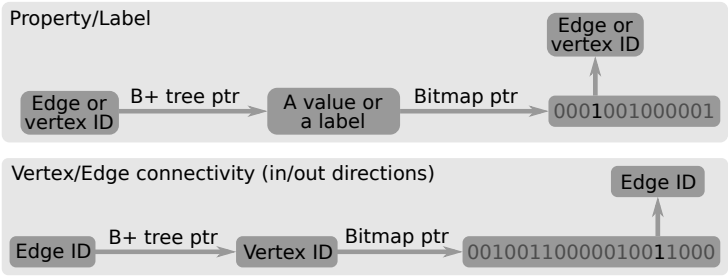


Fig. 18. Sparksee maps for properties, labels, and vertex/edge connectivity. All mappings are bidirectional.

Sparksee is one of the few systems that are *not* record based. Instead, Sparksee uses *maps* implemented as B+ trees [58] and bitmaps. The use of bitmaps allows for some operations to be performed as bit-level operations. For example, if one wants to find all vertices with certain values of properties such as “age” and “first name”, one can simply find two bitmaps associated with the “age” and the “first name” properties, and then derive a third bitmap that is a result of applying a bitwise AND operation to the two input bitmaps.

Uncompressed bitmaps could grow unmanageably in size. As most graphs are sparse, bitmaps indexed by vertices or edges mostly contain zeros. To alleviate large sizes of such sparse bitmaps, they are cut into 32-bit *clusters*. If a cluster contains a non-zero bit, it is stored explicitly. The bitmap

is then represented by a collection of (*cluster-id*, *bit-data*) pairs. These pairs are stored in a sorted tree structure to allow for efficient lookup, insertion, and deletion.

4.9.3 GBase: Sparse Adjacency Matrix Format. GBase [116] is a system that can only represent the *structure* of a directed graph; it stores neither properties nor labels. The goal of GBase is to maintain a compression of the adjacency matrix of a graph such that one can efficiently retrieval all incoming and outgoing edges of a selected vertex without the prohibitive $O(n^2)$ matrix storage overheads. Simultaneously, using the adjacency matrix enables verifying in $O(1)$ time whether two arbitrary vertices are connected. To compress the adjacency matrix, GBase cuts it into K^2 quadratic blocks (there are K blocks along each row and column). Thus, queries that fetch in- and out-neighbors of each vertex require only to fetch K blocks. The parameter K can be optimized for specific databases. When K becomes smaller, one has to retrieve more small files (assuming one block is stored in one file). If K grows larger, there are fewer files but they become larger, generating overheads. Further optimizations can be made when blocks contain either only zeroes or only ones; this enables higher compression rates.

4.10 Data Hubs

Data hubs are systems that enable using multiple data models and corresponding storage designs. They often combine relational databases with RDF, document, and key-value stores. This can be beneficial for applications that require a variety of data models, because it provides a variety of storage options in a single unified database management system. One can keep using RDBMS features, upon which many companies heavily rely, while also storing graph data.

4.10.1 OpenLink Virtuoso. OpenLink Virtuoso [151] provides RDBMS, RDF, and document capabilities by connecting to a variety of storage systems. Graphs are stored in the RDF format only, thus the whole discussion from § 2.1.5 also applies to Virtuoso RDF.

4.10.2 MarkLogic. MarkLogic [133] models graphs with documents for vertices, therefore allowing an arbitrary number of properties for vertices. However, it uses RDF triples for edges.

4.11 Discussion and Takeaways

In this section, we summarize all aspects of our taxonomy, and analyze the trade-offs between these aspects and the general system architecture. For a detailed description and analysis of all the considered aspects, see Section 3 and Tables 2 and 3.

4.11.1 Conceptual Graph Models and Graph Representations. There is no one standard conceptual graph model, but two models have proven to be popular: RDF and LPG. RDF is a well-defined standard. However, it only supports simple triples (subject, predicate, object) representing edges from subject identifiers via predicates to objects. LPG allows vertices and edges to have labels and properties, thus enabling more natural data modeling. Still, it is not standardized, and there are many variants (cf. § 2.1.4). Some systems limit the number of labels to just one. For example, MarkLogic allows properties for vertices but none for edges, and thus can be viewed as a combination of LPG (vertices) and RDF (edges). Data stored in the LPG model can be converted to RDF, as described in § 2.1.6. To benefit from different LPG features while keeping RDF advantages such as simplicity, some researchers proposed and implemented modifications to RDF. Examples are triple attributes or attaching triples to other triples (described in § 4.2.2).

Among native graph databases, while no LPG focused system simultaneously supports RDF, some RDF systems (e.g., Amazon Neptune) also support LPG. Many other classes (KV stores, document stores, RDBMS, wide-column stores, OODBMS) offer only LPG (with some exceptions, e.g., Oracle

Spatial and Graph). The latter suggests that it may be easier to express the LPG model (than the RDF model) with the respective non-graph data models such as a collection of documents.

There are very few systems that use neither RDF nor LPG. HyperGraphDB uses the hypergraph model and GBase uses a simple directed graph model without any labels or properties.

When representing graph structure, many graph databases use variants of AL since it makes traversing neighborhoods efficient and straightforward [168]. This includes several (but not all) systems in the classes of LPG based native graph databases, KV stores, document stores, wide-column stores, tuple stores, and OODBMS. Contrarily, none of the considered RDF, RDBMS, and data hub systems explicitly use AL. This is because the default design of the underlying data model, e.g., tables in RDBMS or documents in document stores, do not often use AL.

Moreover, none of the systems that we analyzed use an *uncompressed* AM as it is inefficient with $O(n^2)$ space, especially for sparse graphs. Systems using AM focus on compression of the adjacency matrix [30], trying to mitigate storage and query overheads (e.g., GBase [116]).

In AL, a potential cause for inefficiency is scanning all edges to find neighbors of a given vertex. To alleviate this, index structures are employed [35]. For a graph with n vertices, such an index is an array of pointers to respective neighborhoods, taking only $O(n)$ space.

4.11.2 Details and Optimizations of Data Organization. Most graph database systems are build upon existing storage designs, including key-value stores, wide-column stores, RDBMS, and others. The advantage of using existing storage designs is that these systems are usually mature and well-tested. The disadvantage is that they may not be perfectly optimized for graph data and graph queries. This is what native graph databases attempt to address.

The records used by the studied graph databases may be unstructured (i.e., not having a pre-specified format such as JSON), as is the case with key-value stores. They can also be structured: document databases often use the JSON format, wide-column stores have a key-value mapping inside each row, row-oriented RDBMS divide each row into columns, OODBMS impose some class definition, and tuple stores as well as some RDF stores use tuples. The details of data layout (i.e., how vertices and edges are exactly represented and encoded in records) may still vary across different system classes. Some structured systems still enable *highly flexible* structure inside their records. For example, document databases that use JSON or wide-columns stores such as Titan and JanusGraph allow for different key-value mappings for each vertex and edge. Other record based systems are more *fixed* in their structure. For example, in OODBMS, one has to define a class for each configuration of vertex and edge properties. In RDBMS, one has to define tables for each vertex or edge type. Overall, most of these systems use records to store vertices, most often one vertex per one record. Some systems store edges in separate records, others store them together with the adjacent vertices. If one wants to find a property of a particular vertex, one has to find a record containing the vertex. The searched property is either stored directly in that record, or its location is accessible via a pointer.

Some systems (e.g., Sparksee, some triple stores, or column-oriented RDBMS) do not store information about vertices and edges *contiguously* in dedicated records. Instead, they maintain separate data structures for each property or label. The information about a given vertex is thus distributed over different structures. If one wants to find a property of a particular vertex, one has to query the associated data structure (index) for that property and find the value for the given vertex. Examples of such used index structures are B+ trees (in Sparksee) or hashtables (in some RDF systems).

Another aspect of a graph data layout is the *design of the adjacency between records*. One can either assign each record an ID and then *link records to one another via IDs*, or one can *use direct memory pointers*. Using IDs requires an indexing structure to find the physical storage address of

a record associated with a particular ID. Direct memory pointers do not require an index for a traversal from one record to its adjacent records. Note that an index might still be used, for example to retrieve a vertex with a particular property value (in this context, direct pointers only facilitate resolving adjacency queries between vertices).

Sometimes graph data is stored directly in an index. Triple stores use indexes for various permutations of subject, predicate and object to answer queries efficiently. Jena TBD stores its triple data inside of these indexes, but has no triple table itself, since the indexes already store all necessary data[190]. HyperGraphDB uses a key-value index, namely Berkeley DB [149], to access its physical storage. Additionally this approach enables the sharing of primitive data values with a reference count, so that multiple identical values are stored only once [108].

The considered systems offer other data layout optimizations. For example, CGE optimizes the way in which it stores strings from its triples/quads. Storing multiple long strings per triple/quad is inefficient, considering the fact that many triples/quads may share strings. Therefore, CGE – similarly to many other RDF systems – maintains a dictionary that maps strings to unique 48-bit integer identifiers (HURIs). For this, two distributed hashtables are used (one for mapping strings to HURIs and one for mapping HURIs to strings). When loading, the strings are sorted and then assigned to HURIs. This allows integer comparisons (equal, greater, smaller, etc.) to be used instead of more expensive string comparisons. This approach is shared by, e.g., tuple stores such as WhiteDB.

4.11.3 Data Distribution. Almost all considered systems support a multi server mode and data replication. Data sharding is also widely supported, but there are some systems that do not offer this feature, most notably, Neo4j. We expect that, with growing dataset sizes, data sharding will ultimately become as common as data replication. Still, it is more complex to provide. We observe that, while sharding is as widely supported on graph databases based on non-graph data models (e.g., document stores) as data replication, there is a significant fraction of native graph databases (both RDF and LPG based) that offer replication but *not* sharding. This indicates that non-graph backends are usually more mature in designs. We also observe that certain systems offer some form of tradeoff between replication and sharding. Specifically, OrientDB offers a form of sharding, in which not all collections of documents have to be copied on each server. However, OrientDB does *not* enable sharding of the collections *themselves* (i.e., distributing one collection across many servers). If an individual collection grows large, it is the responsibility of the user to partition the collection to avoid any additional overheads. Another such example is Neo4j which supports replication and provides *certain level* of support for sharding. Specifically, the user can partition the graph and store each partition in a separate database, limiting data redundancy.

4.11.4 Data Organization vs. Query Performance. Record based systems usually deliver more performance for queries that need to retrieve all or most information about a vertex or an edge. They are more efficient because the required data is stored in consecutive memory blocks. In systems that store data in indexes, one queries a data structure per property, which results in a more random access pattern. On the other hand, if one only wants to retrieve single properties about vertices or edges, such systems may only have to retrieve a single value. Contrarily, many record based systems cannot retrieve only parts of records, fetching more data than necessary.

Furthermore, a decision on whether to use IDs versus direct memory pointers to link records depends on the read/write ratio of the workload for the given system. In the former case, one has to use an indexing structure to find the address of the record. This slows down read queries compared to following direct pointers. However, write queries can be more efficient with the use of IDs instead of pointers. For example, when a record has to be moved to a new address, all pointers

to this record need to be updated to reflect this new address. IDs could remain the same, only the indexing structure needs to modify the address of the given record.

The available performance studies [5, 121, 136, 137, 195] indicate that systems based on non-graph data models, for example document stores or wide-column stores, usually achieve more performance for transactional workloads that update the graph. Contrarily, read-only workloads (both simple and global analytics) often achieve more performance on native graph stores. Global analytics particularly benefit from native graph stores that ensure parallelization of single queries [136].

4.11.5 Query Execution. We now summarize different aspects of query execution. We first analyze how different graph database backends support concurrent and parallel queries, and then we discuss how certain specific systems enhance their execution schemes. Our discussion is by necessity brief, because most systems do not disclose this information⁶.

Support for Concurrency and Parallelism We conclude that (1) almost all systems support concurrent queries, and (2) in almost all classes of systems, fewer systems support parallel query execution (with the exception of OODBMS based graph databases). This indicates that more databases put more stress on high throughput of queries executed per time unit rather than on lowering the latency of a single query. A notable exception is Cray Graph Engine, which does *not* support concurrent queries, but it *does* offer parallelization of single queries. In general, we expect most systems to ultimately support both features.

Implementing Concurrent Execution One of the methods for query concurrency are different types of locks. For example, WhiteDB provides database wide locking with a reader-writer lock [163, 199] which enables concurrent readers but only one writer at a time. As an alternative to locking the whole database, one can also update fields of tuples atomically (set, compare and set, add). WhiteDB itself does not enforce consistency, it is up to the user to use locks and atomics correctly. Another method is based on transactions, used for example by OrientDB that provides distributed transactions with ACID semantics. We discuss transactions separately in § 4.11.6.

Optimizing Parallel Execution Some of the systems that support parallel query execution explicitly optimize the amount of data communicated when executing such parallelized queries. For example, the computation in CGE is distributed over the participating processes. To minimize the amount of all-to-all communication, query results are aggregated locally and – whenever possible – each process only communicates with a few peers to avoid network congestion. Another way to minimize communication, used by MS Graph Engine and the underlying Trinity database, is to reduce the sizes of data chunks exchanged by processes. For this, Trinity maintains special *accessors* that allow for accessing single attributes within a cell without needing to load the complete cell. This lowers the I/O cost for many operations that do not need the whole cells. Several systems harness *one-sided communication*, enabling processes to access one another’s data directly [86]. For example, Trinity can be deployed on InfiniBand [106] to leverage Remote Direct Memory Access (RDMA) [86]. Similarly, Cray’s infrastructure makes memory resources of multiple compute nodes available as a single global address space, also enabling one-sided communication in CGE. This facilitates parallel programming in a distributed environment [27, 86, 176].

Other Execution Optimizations The considered databases come with numerous other system-specific design optimizations. For example, an optimization in ArangoDB’s design prevents reading vertex documents and enables directly accessing one edge document based on the vertex ID *within another edge document*. This may improve cache efficiency and thus reduce query execution time [12]. Another example is Oracle Spatial and Graph that offers an interesting option of *switching its data backend based on the query being executed*. Specifically, its in-memory analysis is boosted by

⁶There is usually much more information available on the data layout of a graph database, and *not* its execution engine.

the possibility to switch the underlying relational storage with the native graph storage provided by the PGX processing engine [74, 105, 170]. In such a configuration, Oracle Spatial and Graph effectively becomes a native graph database. PGX comes with two variants, PGX.D and PGX.SM, that – respectively – offer distributed and shared-memory processing capabilities [105].

4.11.6 Types of Transactions. Overall, support for ACID transactions is widespread in graph databases. However, there are some differences between respective system classes. For example, *all* considered document and RDBMS graph databases offer full ACID support. Contrarily, only around half of all considered key-value and wide-column based systems support ACID transactions. This could be caused by the fact that some backends have more mature transaction related designs.

4.11.7 Indexes. Most graph database systems use indexes. However, their exact purpose varies widely between different systems. We identify four different index use cases: storing the locations of vertex neighborhoods (referred to as “neighborhood indexes”), storing the locations of rich graph data (referred to as “data indexes”), storing the actual graph data, and maintaining non-graph related data (referred to as “structural indexes”).

Neighborhood indexes are used mostly to speed up the access of adjacency lists to accelerate traversal queries. JanusGraph calls these indexes vertex-centric. They are constructed specifically for vertices, so that incident edges can be filtered efficiently to match the traversal conditions [16]. While JanusGraph allows multiple vertex-centric indexes per vertex, each optimized for different conditions, which are then chosen by the query optimizer, simpler solution exist as well. LiveGraph uses a two level hierarchy, where the first level distinguishes edges by their label, before pointing to the actual physical storage [204]. Graphflow indexes the neighbors of a vertex into forward and backward adjacency lists, where each list is first partitioned by the edge label, and secondly by the label of the neighbor vertex [117]. Another example is Sparksee, which uses various different index structures to find the adjacent vertices and properties of a vertex [134].

Data indexes concern data beyond the neighborhood information. It is possible for example to index all vertices that have a specific property (value). They are usually employed to speed up business intelligence workloads (details on workloads are in Section § 5). Many triple stores, for example AllegroGraph [82], provide all six permutations of subject (S), predicate (P), and object (O) as well as additional aggregated indexes. However, to reduce associated costs, other approaches exist as well: TripleBit uses just two permutations (PSO, POS) with two aggregated indexes (SP, SO) and two auxiliary index structures [202]. gStore implements pattern matching queries with the help of two index structures: a VS*-tree, which is a specialized B+-tree, and a trie-based T-index [205]. Some database systems like Amazon Neptune [2] or AnzoGraph [49] only provide implicit indexes, while still being confident to answer all kinds of queries efficiently. However, most graph database systems allow the user to explicitly define indexes. Some of them, like Azure Cosmos DB [142], support composite indexes (a combination of different labels/properties) for more specific use cases. In addition to internal indexes, some systems employ external indexing tools. For example, Titan and JanusGraph [16] use internal indexing for label- and value-based lookups, but rely on external indexing backends (e.g., Elasticsearch [75] or Apache Solr [10]) for non-trivial lookups involving multiple properties, ranges, or full-text search.

We further categorize data indexes based on how they are implemented. Here, we identify three fundamental data structures used to implemented these indexes: trees, skip lists, and hashtables. We categorize systems (for which we were able to find this information) according to this criteria in Table 5. We find no clear connection between the index type and the backend of a graph database, but most systems use tree based indexes.













Graph Database System	Tree	Hashtable	Skip list	Additional remarks
Apache Jena TBD		✗	✗	* B+-tree
ArangoDB	✗			* depends on the used index engine
Blazegraph		✗	✗	* B+-tree
Dgraph	✗		✗	
Memgraph	✗	✗		
OrientDB			✗	* SB-tree with base 500 ‡ also supports a distributed hash table index
VelocityGraph		✗	✗	* B-tree
Virtuoso		✗	✗	* 2D R-tree
WhiteDB		✗	✗	* T-tree

Table 5. **Support for different index implementations in different graph database systems.** “”: A system supports a given index implementation. “✗”: A system does not support a given index implementation.

Data is usually stored in data structures. When these data structures become more complex, some graph database choose to enhance their design with **structural indexes**. LiveGraph among other systems uses a vertex index to map its vertex IDs to a physical storage location [204]. Similarly ArangoDB uses a hybrid index, a hashtable, multiple times to find the documents of incident edges and adjacent vertices of a vertex [11].

Finally, we discuss the use of indexes for **data storage** in more detail in § 4.11.2.

5 GRAPH DATABASE QUERIES AND WORKLOADS

We provide a taxonomy of graph database queries and workloads. First, we categorize them using the *scope of the accessed graph* and thus, implicitly, the amount of accessed data (§ 5.1). We then outline the classification from the *LDBC Benchmark* [5] (§ 5.2). Next, a categorization of graph queries based on the *matched patterns* (§ 5.3) is discussed. Finally, we illustrate the most general distinction into *OLTP* and *OLAP* (§ 5.4). We also briefly mention *loading input datasets* into the database (§ 5.5). Figure 19 summaries all elements of the proposed taxonomy. Figure 20 illustrates the taxonomy of queries in the context of accessing the LPG graph. We omit detailed discussions and examples as they are provided in different associated papers (query languages [3, 4], OLAP workloads [8], benchmarks related to certain aspects [55, 127, 128] and whole systems [5, 13, 18, 50, 78, 109, 112, 188] and surveys on system performance [69, 137]). Instead, our goal is to deliver a broad overview and taxonomy, and point the reader to the detailed material available elsewhere.

5.1 Scopes of Graph Queries

We describe queries in the increasing order of their scope. We focus on the LPG model, see § 2.1.3. Figure 20 depicts the scope of graph queries.

5.1.1 Local Queries. Local queries involve single vertices or edges. For example, given a vertex or an edge ID, one may want to retrieve the labels and properties of this vertex or edge. Other examples include fetching the value of a given property (given the property key), deriving the set of all labels, or verifying whether a given vertex or an edge has a given label (given the label name). These queries are used in social network workloads [13, 18] (e.g., to fetch the profile information of a user) and in benchmarks [112] (e.g., to measure the vertex look-up time).

5.1.2 Neighborhood Queries. Neighborhood queries retrieve all edges adjacent to a given vertex, or the vertices adjacent to a given edge. This query can be further restricted by, for example, retrieving only the edges with a specific label. Similarly to local queries, social networks often require a retrieval of the friends of a given person, which results in querying the local neighborhood [13, 18].

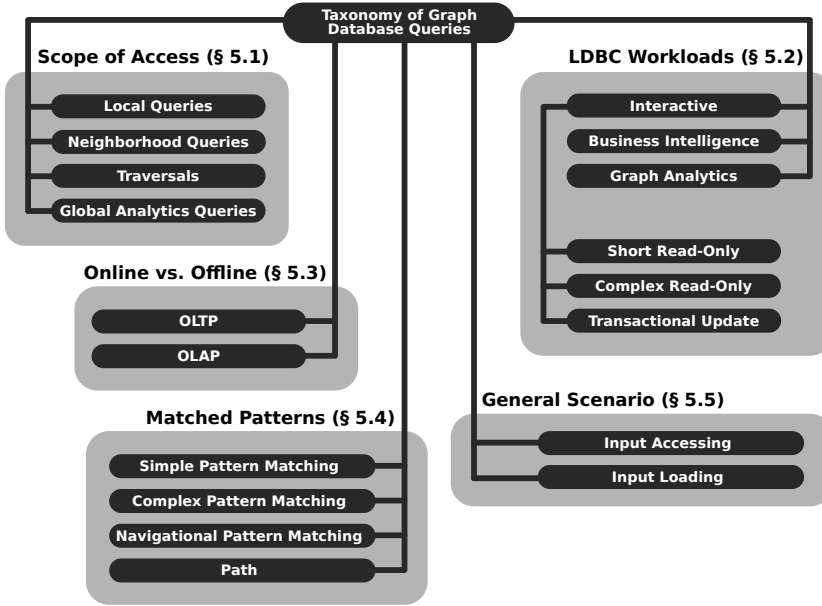
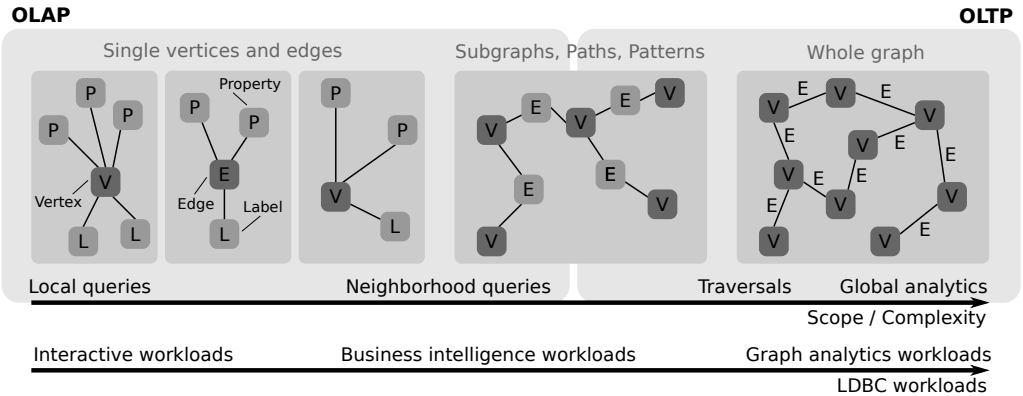


Fig. 19. Taxonomy of different graph database queries and workloads.

Fig. 20. Illustration of **different query scopes** and their relation to other graph query taxonomy aspects, in the context of accessing a Labeled Property Graph.

5.1.3 Traversals. In a traversal query, one explores a part of the graph beyond a single neighborhood. These queries usually start at a single vertex (or a small set of vertices) and traverse some graph part. We call the initial vertex or the set of vertices the *anchor* or *root* of the traversal. Queries can restrict what edges or vertices can be retrieved or traversed. As this is a common graph database task, this query is also used in different performance benchmarks [55, 69, 112].

5.1.4 Global Graph Analytics. Finally, we identify graph analytics queries, often referred to as OLAP, which by definition consider the whole graph (not necessarily every property but all vertices and edges). Different benchmarks [21, 50, 69, 137] take these large-scale queries into account since

they are used in different fields such as threat detection [72] or computational chemistry [17]. As indicated in Tables 2 and 3, many graph databases support such queries. Graph processing systems such as Pregel [131] or Giraph [8] focus specifically on resolving OLAP [96]. Example queries include resolving global pattern matching [53, 180], shortest paths [67], max-flow or min-cut [60], minimum spanning trees [122], diameter, eccentricity, connected components, PageRank [155], and many others. Some traversals can also be global (e.g., finding all shortest paths of unrestricted length), thus falling into the category of global analytics queries.

5.2 Classes of Graph Workloads

We also outline an existing taxonomy of graph database workloads that is provided as a part of the LDBC benchmarks [5]. LDBC is an effort by academia and industry to establish a set of standard benchmarks for measuring the performance of graph databases. The effort currently specifies *interactive workloads*, *business intelligence workloads*, and *graph analytics workloads*.

5.2.1 Interactive Workloads. A part of LDBC called the Social Network Benchmark (SNB) [78] identifies and analyzes *interactive workloads* that can collectively be described as either read-only queries or simple transactional updates. They are divided into three further categories. First, *short read-only queries* start with a single graph element (e.g., a vertex) and lookup its neighbors or conduct small traversals. Second, *complex read-only queries* traverse larger parts of the graph; they are used in the LDBC benchmark to not just assess the efficiency of the data retrieval process but also the quality of query optimizers. Finally, *transactional update queries* insert either a single element (e.g., a vertex), possibly together with its adjacent edges, or a single edge. This workload tests common graph database operations such as the lookup of a friend profile in a social network.

5.2.2 Business Intelligence Workloads. Next, LDBC identifies *business intelligence (BI) workloads* [188], which fetch large data volumes, spanning large parts of a graph. Contrarily to the interactive workloads, the BI workloads heavily use summarization and aggregation operations such as sorting, counting, or deriving minimum, maximum, and average values. They are read-only. The LDBC specification provides an extensive list of BI workloads that were selected so that different performance aspects of a database are properly stressed when benchmarking.

5.2.3 Graph Analytics Workloads. Finally, the LDBC effort comes with a graph analytics benchmark [109], where six graph algorithms are proposed as a standard benchmark for a graph analytics part of a graph database. These algorithms are “*Breadth-First Search*, *PageRank* [155], *weakly connected components* [88], *community detection using label propagation* [41], *deriving the local clustering coefficient* [175], and *computing single-source shortest paths* [67]”.

5.2.4 Scope of LDBC Workloads. The LDBC *interactive workloads* correspond to *local*, *neighborhood*, and *traversals*. The LDBC *business intelligence workloads* range from *traversals* to *global graph analytics queries*. The LDBC graph analytics benchmark corresponds to *global graph analytics*.

5.3 Graph Patterns and Navigational Expressions

Angles et al. [4] inspected in detail the theory of graph queries. In one identified family of graph queries, called *simple graph pattern matching*, one prescribes a graph pattern (e.g., a specification of a class of subgraphs) that is then matched to the graph maintained by the database, searching for the occurrences of this pattern. This query can be extended with aggregation and a projection function to so called *complex graph pattern matching*. Furthermore, *path queries* allow to search for paths of arbitrary distances in the graph. One can also combine complex graph pattern matching

and path queries, resulting in *navigational graph pattern matching*, in which a graph pattern can be applied recursively on the parts of the path.

5.4 Interactive Transactional Querying (OLTP) and Offline Graph Processing (OLAP)

One can also distinguish between *Online Transactional Processing (OLTP)* and *Offline Analytical Processing (OLAP)*. Typically, OLTP workloads consist of many queries local in scope, such as neighborhood queries, certain restricted traversals, or lookups, inserts, deletes, and updates of single vertices and edges. They are usually executed with some transactional guarantees. The goal is to achieve high throughput and answer the queries at interactive speed (low latency). OLAP workloads have been a subject of numerous research efforts in the last decade [20, 31, 32, 114, 138, 159, 182]. They are usually not processed at interactive speeds, as the queries are inherently complex and global in scope. OLAP and OLTP are the most general categories, with OLTP largely covering local queries, simple neighborhood queries, simple subgraph and pattern queries, and LDBC’s interactive and simple BI workloads. OLAP correspond to complex subgraph and pattern queries, traversals, and LDBC’s global graph analytics and complex BI workloads. Thus, in the following, we will focus on analyzing graph databases in the context of their support for OLAP and OLTP.

5.5 Input Loading

Finally, certain benchmarks also analyze bulk input loading [55, 69, 112]. Specifically, given an input dataset, they measure the time to load this dataset into a database. This scenario is common when data is migrated between systems.

5.6 Discussion and Takeaways

We now analyze different aspects related to supported workloads and languages.

5.6.1 Supported Workloads. We analyze support for OLTP and OLAP. Both categories are widely supported, but with certain differences across specific backend classes, specifically, (1) all considered document stores focus solely on OLTP, (2) some RDBMS graph databases do not support or focus on OLAP, and (3) some native graph databases do not support OLTP. We conjecture that this is caused by the prevalent historic use cases of these systems, and the associated features of the backend design. For example, document stores have traditionally mostly focused on maintaining document related data and to answer simple queries, instead of running complicated global graph analytics. Thus, it may be very challenging to ensure high performance of such global workloads on this backend class. Instead, native graph databases work directly with the graph data model, making it simpler to develop fast traversals and other OLAP workloads. As for RDBMS, they were traditionally not associated with graph global workloads. However, graph analytics based on RDBMS has become a separate and growing area of research. Zhao et al. [203] study the general use of RDBMS for graphs. They define four new relational algebra operations for modeling graph operations. They show how to define these four operations with six smaller building blocks: basic relational algebra operations, such as group-by and aggregation. Xirogiannopoulos et al. [200] describe GraphGen, an end-to-end graph analysis framework that is built on top of an RDBMS. GraphGen supports graph queries through so called Graph-Views that define graphs as transformations over underlying relational datasets. This provides a graph modeling abstraction, and the underlying representation can be optimized independently.

Some document stores still provide at least partial support for traversal-like workloads. For example, in ArangoDB, documents are indexed using a hashtable, where the `_key` attribute serves as the hashtable key. A traversal over the neighbors of a given vertex works as follows. First, given the `_key` of a vertex v , ArangoDB finds all v ’s adjacent edges using the hybrid index. Next, the

system retrieves the corresponding edge documents and fetches all the associated *_to* properties. Finally, the *_to* properties serve as the new *_key* properties when searching for the neighboring vertices. An optimization in ArangoDB's design prevents reading vertex documents and enables directly accessing one edge document based on the vertex ID *within another edge document*. This may improve cache efficiency and thus reduce query execution time [12].

There are other correlations between supported workloads and system design features. For instance, we observe that systems that do not target OLTP, also often do not provide, or focus on, ACID transactions. This is because ACID is not commonly used with OLAP. Examples include Cray Graph Engine, RedisGraph, or Graphflow.

5.6.2 Supported Languages. We also analyze support for graph query languages. Some types of backends focus on one specific language: triple stores and SPARQL, document stores and Gremlin, wide-column stores and Gremlin, RDBMS and SQL. Other classes are not distinctively correlated with some specific language, although Cypher seems most popular among LPG based native graph stores. Usually, the query language support is primarily affected by the supported conceptual graph model; if it is RDF, then the system usually supports SPARQL while systems focusing on LPG often support Cypher or Gremlin.

Several systems come with their own languages, or variants of the established ones. For example, in MS Graph Engine, cells are associated with a schema that is defined using the Trinity Specification Language (TSL) [177]. TSL enables defining the structure of cells similarly to C-structs. For example, a cell can hold data items of different data types, including IDs of other cells. Moreover, querying graphs in Oracle Spatial and Graph is possible using PGQL [196], a declarative, SQL-like, graph pattern matching query language. PGQL is designed to match the hybrid structure of Oracle Spatial and Graph, and it allows for querying both data stored on disk in Oracle Database as well as in in-memory parts of graph datasets.

Besides their primary language, different systems also offer support for additional language functionalities. For example, Oracle Spatial and Graph also supports SQL and SPARQL (for RDF graphs). Moreover, the offered Java API implements Apache Tinkerpop interfaces, including the Gremlin API.

6 CHALLENGES

There are numerous research challenges related to the design of graph database systems.

First, establishing a single graph model for these systems is far from being complete. While LPG is used most often, (1) its definition is very broad and it is rarely fully supported, and (2) RDF is also often used in the context of storing and managing graphs. Moreover, it is unclear what are precise relationships between a selected graph model and the corresponding consequences for storage and performance tradeoffs when executing different types of workloads.

Second, a clear identification of the most advantageous design choices for different existing graph database workloads and use cases is yet to be determined. As illustrated in this survey, existing systems support a plethora of forms of data organization, and it is not clear which ones are best for many scenarios, such as OLAP vs. OLTP. A strongly related challenge is the best design for a high throughput and low latency system that supports *both* OLAP and OLTP workloads.

There also exist many graph workloads that have been largely unaddressed by the design and performance analyses of existing graph database systems. First, there are numerous graph pattern matching problems such as listing maximal cliques, listing k -cliques, subgraph isomorphism, and many others [36]. These problems are usually computationally challenging (e.g., listing maximal cliques is NP-hard) and the associated algorithms come with complex control flow and load balancing [36]. Other areas include vertex reordering problems (e.g., listing vertices by their

degeneracy), or optimization (e.g., graph coloring) [24]. These problems were considered in the context of graph algorithms processing simple graphs, and incorporating rich models such as LPG or RDF would further increase complexity, and offer many associated research challenges for future work, for example designing specific indexes, data layouts, or distribution strategies.

Another interesting avenue of research is to enhance graph databases with the capabilities of deep learning. For example, one could train a neural network using the incoming workload requests and the associated performance patterns, and then use the outcomes of that training for better load balancing of the future workload demands. This approach could be applied to other aspects of a graph database, such as data partitioning, index placement, or even to selecting the most beneficial data model (i.e., one could attempt to learn the best model for a given class of workloads).

There is a large body of existing work in the design of dynamic graph processing frameworks [23]. These systems differ from graph databases in several aspects, for example they often employ simple graph models (and not LPG or RDF) or do not often target business intelligence workloads, instead focusing on maximizing the rate of simple graph updates (e.g., inserting an edge) and the performance of global graph analytics. Simultaneously, they share the fundamental property of graph databases: dealing with a dynamic graph with evolving structure. Moreover, different performance analyses indicate that streaming frameworks are much faster (up to orders of magnitude) than graph databases [137, 195]. This suggests that harnessing mechanisms used in such frameworks in the context of graph databases could significantly enhance the performance of the latter.

Furthermore, while there exists past research into the impact of the underlying network on the performance of a distributed graph analytics framework [153], little was done into investigating this performance relationship in the context of graph database workloads. To the best of our knowledge, there are no efforts into developing a topology-aware or routing-aware data distribution scheme for graph databases, especially in the context of recently proposed data center and high-performance computing network topologies [28, 120] and routing architectures [33, 87, 129].

Moreover, contrarily to the general static graph processing and graph streaming, little research exists into accelerating graph databases using different types of hardware architectures, accelerators, and hardware-related designs, for example FPGAs [25, 34], designs related to network interface cards such as SmartNICs [22, 66], hardware transactions [29], processing in memory [1], and others [1, 26]. In addition, a related research direction focuses on re-using different concepts from general distributed graph processing in the domain of graph databases, and vice versa [95].

Finally, many research challenges in the design of graph databases are related specifically to the design of NoSQL stores. These challenges are discussed in more detail in past recent work [63] and include efficient data partitioning [46, 147, 158], user-friendly query formulation, high-performance transaction processing, and ensuring security in the form of authentication and encryption.

7 CONCLUSION

Graph databases constitute an important area of academic research and different industry efforts. They are used to maintain, query, and analyze numerous datasets in different domains in industry and academia. Many graph databases of different types have been developed. They use many data models and representations, they are constructed using miscellaneous design choices, and they enable a large number of queries and workloads. In this work, we provide the first survey and taxonomy of this rich graph database landscape. Our work can be used not only by researchers willing to learn more about this fascinating subject, but also by architects, developers, and project managers who want to select the most advantageous graph database system or design.

ACKNOWLEDGEMENTS We thank Gabor Szarnyas for extensive feedback, and Hannes Voigt, Matteo Lissandrini, Daniel Ritter, Lukas Braun, Janez Ales, Nikolay Yakovets, and Khuzaima Daudjee for insightful remarks.

REFERENCES

- [1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. *ACM SIGARCH Comput. Archit. News*, 43(3S):105–117, 2015.
- [2] Amazon. Amazon Neptune. Available at <https://aws.amazon.com/neptune/>.
- [3] R. Angles, M. Arenas, P. Barcelo, et al. G-CORE: A Core for Future Graph Query Languages. In *ACM SIGMOD*, pages 1421–1432, 2018.
- [4] R. Angles, M. Arenas, P. Barceló, A. Hogan, et al. Foundations of Modern Query Languages for Graph Databases. *ACM CSUR*, 50(5), 2017.
- [5] R. Angles et al. The Linked Data Benchmark Council: A Graph and RDF Industry Benchmarking Effort. *ACM SIGMOD Rec.*, 43(1):27–31, 2014.
- [6] R. Angles and C. Gutierrez. Survey of Graph Database Models. *ACM CSUR*, 40(1), 2008.
- [7] Apache. Apache Cassandra. Available at <https://cassandra.apache.org/>.
- [8] Apache. Apache Giraph. Available at <https://giraph.apache.org/>.
- [9] Apache. Apache Marmotta. Available at <http://marmotta.apache.org/>.
- [10] Apache. Apache Solr. Available at <https://lucene.apache.org/solr/>.
- [11] ArangoDB Inc. ArangoDB. Available at <https://docs.arangodb.com/3.3/Manual/DataModeling/Concepts.html>.
- [12] ArangoDB Inc. ArangoDB: Index Free Adjacency or Hybrid Indexes for Graph Databases. Available at <https://www.arangodb.com/2016/04/index-free-adjacency-hybrid-indexes-graph-databases/>.
- [13] T. G. Armstrong et al. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *ACM SIGMOD*, pages 1185–1196, 2013.
- [14] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, et al. The Object-Oriented Database System Manifesto. In *DOOD*, pages 223–240, 1990.
- [15] P. Atzeni and V. De Antonellis. *Relational Database Theory*. 1993.
- [16] Aurelius. Titan Data Model. Available at <http://s3.thinkaurelius.com/docs/titan/1.0.0/data-model.html>.
- [17] A. T. Balaban. Applications of Graph Theory in Chemistry. *J. Chem. Inf. Comput. Sci.*, 25(3):334–343, 1985.
- [18] S. Barahmand and S. Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. In *CIDR*, 2013.
- [19] D. Bartholomew. Mariadb vs. MySQL. *Dostopano*, 7(10):2014, 2012.
- [20] O. Batarfi et al. Large scale graph processing systems: survey and an experimental evaluation. *Cluster Computing*, 18(3):1189–1213, 2015.
- [21] S. Beamer, K. Asanović, and D. Patterson. The GAP Benchmark Suite. *arXiv preprint arXiv:1508.03619*, 2015.
- [22] M. Besta et al. Active Access: A Mechanism for High-Performance Distributed Data-Centric Computations. In *ACM ICS*, pages 155–164, 2015.
- [23] M. Besta et al. Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems. *arXiv preprint arXiv:1912.12740*, 2019.
- [24] M. Besta et al. High-Performance Parallel Graph Coloring with Strong Guarantees on Work, Depth, and Quality. In *ACM/IEEE SC*, 2020.
- [25] M. Besta, M. Fischer, T. Ben-Nun, et al. Substream-Centric Maximum Matchings on FPGA. In *ACM FPGA*, page 152–161, 2019.
- [26] M. Besta, S. M. Hassan, S. Yalamanchili, R. Ausavarungrun, O. Mutlu, and T. Hoefler. Slim NoC: A Low-Diameter On-Chip Network Topology for High Energy Efficiency and Scalability. *ACM SIGPLAN Not.*, 53(2):43–55, 2018.
- [27] M. Besta and T. Hoefler. Fault Tolerance for Remote Memory Access Programming Models. In *ACM HPDC*, pages 37–48, 2014.
- [28] M. Besta and T. Hoefler. Slim Fly: A Cost Effective Low-Diameter Network Topology. In *ACM/IEEE SC*, pages 348–359, 2014.
- [29] M. Besta and T. Hoefler. Accelerating Irregular Computations with Hardware Transactional Memory and Active Messages. In *ACM HPDC*, pages 161–172, 2015.
- [30] M. Besta and T. Hoefler. Survey and Taxonomy of Lossless Graph Compression and Space-Efficient Graph Representations. *arXiv preprint arXiv:1806.01799*, 2018.
- [31] M. Besta, F. Marending, et al. SlimSell: A Vectorizable Graph Representation for Breadth-First Search. In *IEEE IPDPS*, pages 32–41, 2017.

- [32] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoeﬂer. To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations. In *ACM HPDC*, pages 93–104, 2017.
- [33] M. Besta, M. Schneider, K. Cynk, M. Konieczny, E. Henriksson, S. Di Girolamo, A. Singla, and T. Hoeﬂer. FatPaths: Routing in Supercomputers, Data Centers, and Clouds with Low-Diameter Networks when Shortest Paths Fall Short. *arXiv preprint arXiv:1906.10885*, 2019.
- [34] M. Besta, D. Stanojevic, J. De Fine Licht, et al. Graph Processing on FPGAs: Taxonomy, Survey, Challenges. *arXiv preprint arXiv:1903.06697*, 2019.
- [35] M. Besta, D. Stanojevic, T. Zivic, J. Singh, et al. Log(Graph): A near-Optimal High-Performance Graph Representation. In *ACM PACT*, 2018.
- [36] M. Besta, Z. Vonarburg-Shmaria, Y. Schaffner, L. Schwarz, G. Kwasniewski, L. Gianinazzi, J. Beranek, K. Janda, T. Holenstein, et al. GraphMineSuite: Enabling High-Performance and Programmable Graph Mining Algorithms with Set Algebra. *arXiv preprint arXiv:2103.03653*, 2021.
- [37] M. Besta, S. Weber, L. Gianinazzi, R. Gerstenberger, A. Ivanov, Y. Olthik, and T. Hoeﬂer. Slim Graph: Practical Lossy Graph Compression for Approximate Graph Processing, Storage, and Analytics. In *ACM/IEEE SC*, 2019.
- [38] Bitnine Global Inc. AgensGraph. Available at <https://bitnine.net/agensgraph-2/>.
- [39] Blazegraph. BlazeGraph DB. Available at <https://www.blazegraph.com/>.
- [40] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0: An XML Query Language. 2002.
- [41] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered Label Propagation: A Multiresolution Coordinate-Free Ordering for Compressing Social Networks. In *ACM WWW*, pages 587–596, 2011.
- [42] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets. Querying Graphs. *Synthesis Lectures on Data Management*, 10(3):1–184, 2018.
- [43] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.
- [44] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, 2014.
- [45] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0, 2000.
- [46] A. Buluç et al. Recent Advances in Graph Partitioning. In *Algorithm Engineering: Selected Results and Surveys*, pages 117–158. 2016.
- [47] Callidus Software Inc. OrientDB. Available at <https://orientdb.com>.
- [48] Callidus Software Inc. OrientDB: Lightweight Edges. Available at <https://orientdb.com/docs/3.0.x/java/Lightweight-Edges.html>.
- [49] Cambridge Semantics. AnzoGraph. Available at <https://www.cambridgesemantics.com/product/anzograph/>.
- [50] M. Capotà, T. Hegeman, A. Iosup, et al. Graphalytics: A Big Data Benchmark for Graph-Processing Platforms. In *ACM GRADES*, 2015.
- [51] A. Castelltort et al. Representing history in graph-oriented NoSQL databases: A versioning system. In *IEEE ICDIM*, pages 228–234, 2013.
- [52] Cayley. CayleyGraph. Available at <https://cayley.io/> and <https://github.com/cayleygraph/cayley>.
- [53] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast Graph Pattern Matching. In *IEEE ICDE*, pages 913–922, 2008.
- [54] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, et al. One Trillion Edges: Graph Processing at Facebook-Scale. *VLDB*, 8(12):1804–1815, 2015.
- [55] M. Ciglan, A. Averbuch, et al. Benchmarking Traversal Operations over Graph Databases. In *IEEE ICDE Workshops*, pages 186–189, 2012.
- [56] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0, 1999.
- [57] E. F. Codd. Relational Database: A Practical Foundation for Productivity. In *Readings in Artificial Intelligence and Databases*, pages 60–68. 1989.
- [58] D. Comer. The Ubiquitous B-Tree. *ACM CSUR*, 11(2), 1979.
- [59] R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 Concepts and Abstract Syntax, 2014.
- [60] G. B. Dantzig and D. R. Fulkerson. On the Max Flow Min Cut Theorem of Networks. In *RAND Corporation Paper Series*, 1955.
- [61] DataStax, Inc. DSE Graph (DataStax). Available at <https://www.datastax.com/>.
- [62] C. J. Date and H. Darwen. *A Guide to the SQL Standard*, volume 3. 1987.
- [63] A. Davoudian, L. Chen, and M. Liu. A Survey on NoSQL Stores. *ACM CSUR*, 51(2), 2018.
- [64] Dgraph Labs Inc. BadgerDB. <https://dbdb.io/db/badgerdb>.
- [65] Dgraph Labs, Inc. DGraph. Available at <https://dgraph.io/>, <https://docs.dgraph.io/design-concepts>.
- [66] S. Di Girolamo, K. Taranov, et al. Network-Accelerated Non-Contiguous Memory Transfers. *arXiv preprint arXiv:1908.08590*, 2019.
- [67] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

- [68] N. Doekemeijer and A. L. Varbanescu. A Survey of Parallel Graph Processing Frameworks. Technical report, Delft University of Technology, 2014.
- [69] D. Dominguez-Sal et al. Survey of Graph Database Performance on the HPC Scalable Graph Analysis Benchmark. In *WAIM*, pages 37–48, 2010.
- [70] A. Dubey et al. Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps. *VLDB*, 9(11):852–863, 2016.
- [71] P. DuBois. *MySQL*. 1999.
- [72] W. Eberle, J. Graves, et al. Insider Threat Detection Using a Graph-Based Approach. *Journal of Applied Security Research*, 6(1):32–81, 2010.
- [73] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. STINGER: High performance data structure for streaming graphs. In *IEEE HPEC*, pages 1–5, 2012.
- [74] H. El Maazouz, G. Wachsmuth, M. Sevenich, et al. A DSL-Based Framework for Performance Assessment. In *EMENA-ISTL*, pages 260–270, 2019.
- [75] Elastic. Elasticsearch. Available at <https://www.elastic.co/products/elasticsearch>.
- [76] R. Elmasri et al. Advantages of Distributed Databases. In *Fundamentals of Database Systems, 6th Edition*, chapter 25.1.5, page 882. 2011.
- [77] R. Elmasri and S. B. Navathe. Data Fragmentation. In *Fundamentals of Database Systems, 6th Edition*, chapter 25.4.1, pages 894–897. 2011.
- [78] O. Erling, A. Averbuch, J. Larriba-Pey, et al. The LDBC Social Network Benchmark: Interactive Workload. In *ACM SIGMOD*, pages 619–630, 2015.
- [79] FactNexus. GraphBase. Available at <https://graphbase.ai/>.
- [80] Fauna. FaunaDB. Available at <https://fauna.com/>.
- [81] N. Francis, A. Green, P. Guagliardo, et al. Cypher: An Evolving Query Language for Property Graphs. In *ACM SIGMOD*, pages 1433–1445, 2018.
- [82] Franz Inc. AllegroGraph. Available at <https://franz.com/agraph/allegrograph/>.
- [83] S. K. Gajendran. A Survey on NoSQL Databases. Technical report, University of Illinois, 2012.
- [84] H. Garcia-Molina, J. D. Ullman, et al. Data Replication. In *Database Systems: The Complete Book, 1st Edition*, chapter 19.4.3, page 1021. 2002.
- [85] L. George. *HBase: The Definitive Guide*. 2011.
- [86] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling highly-scalable remote memory access programming with MPI-3 One Sided. *Scientific Programming*, 22(2):75–91, 2014.
- [87] S. Ghorbani, Z. Yang, et al. DRILL: Micro Load Balancing for Low-Latency Data Center Networks. In *ACM SIGCOMM*, pages 225–238, 2017.
- [88] L. Gianinazzi et al. Communication-Avoiding Parallel Minimum Cuts and Connected Components. *ACM SIGPLAN Not.*, 53(1):219–232, 2018.
- [89] Google. Graphd. Available at <https://github.com/google/graphd>.
- [90] Graph Story Inc. Graph Story. Available at <https://www.graphstory.com/>.
- [91] A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, et al. Updating Graph Databases with Cypher. *VLDB*, 12(12):2242–2254, 2019.
- [92] A. Green, M. Junghanns, M. Kießling, et al. openCypher: New Directions in Property Graph Querying. In *EDBT*, pages 520–523, 2018.
- [93] L. Guzenda. Objectivity/DB – A high performance object database architecture. In *HIPOD*, 2000.
- [94] J. Han, E. Haihong, G. Le, and J. Du. Survey on NoSQL database. In *IEEE ICPA*, pages 363–366, 2011.
- [95] M. Han and K. Daudjee. Providing Serializability for Pregel-like Graph Processing Systems. In *EDBT*, pages 77–88, 2016.
- [96] M. Han, K. Daudjee, K. Ammar, et al. An Experimental Comparison of Pregel-like Graph Processing Systems. *VLDB*, 7(12):1047–1058, 2014.
- [97] O. Hartig. Reconciliation of RDF* and Property Graphs. *arXiv preprint arXiv:1409.3288*, 2014.
- [98] O. Hartig. RDF* and SPARQL*: An Alternative Approach to Annotate Statements in RDF. In *ISWC (Poster)*, 2017.
- [99] O. Hartig. Foundations to Query Labeled Property Graphs using SPARQL*. In *SEM4TRA-AMAR*, 2019.
- [100] O. Hartig and J. Pérez. Semantics and Complexity of GraphQL. In *WWW*, pages 1155–1164, 2018.
- [101] J. Hayes. A Graph Model for RDF. diploma thesis, Technische Universität Darmstadt, Universidad de Chile, 2004.
- [102] J. M. Hellerstein and M. Stonebraker. *Readings in Database Systems*. 2005.
- [103] J. A. Hoffer, V. Ramesh, and H. Topi. *Modern Database Management*. 2011.
- [104] F. Holzschuher and R. Peinl. Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4j. In *EDBT*, pages 195–204, 2013.

- [105] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, et al. PGX.D: A Fast Distributed Graph Processing Engine. In *ACM/IEEE SC*, 2015.
- [106] InfiniBand Trade Association. InfiniBand: Architecture Specification 1.3. 2015.
- [107] InfoGrid. The InfoGrid Graph Database. Available at <http://infogrid.org>.
- [108] B. Iordanov. HyperGraphDB: A Generalized Graph Database. In *WAIM*, pages 25–36, 2010.
- [109] A. Iosup, T. Hegeman, W. L. Ngai, S. Heldens, A. Prat-Pérez, T. Manhardt, H. Chafio, M. Capotă, N. Sundaram, M. Anderson, I. G. Tănase, et al. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *VLDB*, 9(13):1317–1328, 2016.
- [110] Jesús Barrasa. RDF Triple Stores vs. Labeled Property Graphs: What’s the Difference? Available at <https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/>.
- [111] B. Jiang. A Short Note on Data-Intensive Geospatial Computing. In *Information Fusion and Geographic Information Systems*, pages 13–17. 2011.
- [112] S. Jouili and V. Vansteenbergh. An Empirical Comparison of Graph Databases. In *IEEE SocialCom*, pages 708–715, 2013.
- [113] M. Junghanns, A. Petermann, M. Neumann, and E. Rahm. Management and Analysis of Big Graph Data: Current Systems and Open Challenges. In *Handbook of Big Data Technologies*, pages 457–505. 2017.
- [114] V. Kalavri, V. Vlassov, and S. Haridi. High-Level Programming Abstractions for Distributed Graph Processing. *IEEE TKDE*, 30(2):305–324, 2017.
- [115] R. K. Kaliyar. Graph databases: A survey. In *IEEE ICCCA*, pages 785–790, 2015.
- [116] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: An Efficient Analysis Platform for Large Graphs. *VLDB Journal*, 21(5):637–650, 2012.
- [117] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, et al. Graphflow: An Active Graph Database. In *ACM SIGMOD*, pages 1695–1698, 2017.
- [118] M. Kay. *XSLT Programmer’s Reference*. 2001.
- [119] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, et al. Mathematical foundations of the GraphBLAS. In *IEEE HPEC*, pages 1–9, 2016.
- [120] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. In *IEEE ISCA*, pages 77–88, 2008.
- [121] V. Kolomicenko. Analysis and Experimental Comparison of Graph Databases. master thesis, Charles University in Prague, 2013.
- [122] J. B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proc. Amer. Math. Soc.*, 7(1):48–50, 1956.
- [123] V. Kumar and A. Babu. Domain Suitable Graph Database Selection: A Preliminary Report. In *ICAESAM*, pages 26–29, 2015.
- [124] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.
- [125] LambdaZen LLC. Bitsy. Available at <https://github.com/lambdazen/bitsy> and <https://bitbucket.org/lambdazen/bitsy/wiki/Home>.
- [126] H. Lin, X. Zhu, B. Yu, X. Tang, et al. ShenTu: Processing Multi-Trillion Edge Graphs on Millions of Cores in Seconds. In *ACM/IEEE SC*, 2018.
- [127] M. Lissandrini, M. Brugnara, et al. Beyond Macrobenchmarks: Microbenchmark-Based Graph Database Evaluation. *VLDB*, 12(4):390–403, 2018.
- [128] M. Lissandrini et al. An Evaluation Methodology and Experimental Comparison of Graph Databases. Technical report, University of Trento, 2017.
- [129] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, et al. Multi-Path Transport for RDMA in Datacenters. In *NSDI*, pages 357–371, 2018.
- [130] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in Parallel Graph Processing. *Par. Proc. Let.*, 17(1):5–20, 2007.
- [131] G. Malewicz, M. H. Austern, A. J. Bik, et al. Pregel: A System for Large-Scale Graph Processing. In *ACM SIGMOD*, pages 135–146, 2010.
- [132] MariaDB. OQGRAPH. Available at <https://mariadb.com/kb/en/library/oqgraph-storage-engine/>.
- [133] MarkLogic Corporation. MarkLogic. Available at <https://www.marklogic.com>.
- [134] N. Martínez-Bazan, M. A. Águila Lorente, et al. Efficient Graph Management Based on Bitmap Indices. In *ACM IDEAS*, page 110–119, 2012.
- [135] J. Marton, G. Szárnyas, and D. Varró. Formalising openCypher Graph Queries in Relational Algebra. In *ADBIS*, pages 182–196, 2017.

- [136] K. J. Maschhoff, R. Vesse, et al. Quantifying Performance of CGE: A Unified Scalable Pattern Mining and Search System. In *CUG*, 2017.
- [137] R. C. McColl, D. Ediger, J. Poovey, et al. A Performance Evaluation of Open Source Graph Databases. In *ACM PPAA*, pages 11–18, 2014.
- [138] R. R. McCune, T. Weninger, and G. Madey. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *ACM CSUR*, 48(2), 2015.
- [139] Memgraph Ltd. Memgraph. Available at <https://memgraph.com/>.
- [140] S. M. Meyer, J. Degener, et al. Optimizing Schema-Last Tuple-Store Queries in Graphd. In *ACM SIGMOD*, pages 1047–1056, 2010.
- [141] A. Mhedhbi and S. Salihoglu. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *VLDB*, 12(11):1692–1704, 2019.
- [142] Microsoft. Azure Cosmos DB. Available at <https://azure.microsoft.com/en-us/services/cosmos-db/>.
- [143] Microsoft. Microsoft SQL Server 2017. Available at <https://www.microsoft.com/en-us/sql-server/sql-server-2017>.
- [144] B. Momjian. *PostgreSQL: Introduction and Concepts*. 2001.
- [145] T. Mueller. H2 Database Engine. Available at <http://www.h2database.com>, 2005.
- [146] Networked Planet Limited. BrightstarDB. Available at <http://brightstardb.com/>.
- [147] D. Nicoara, S. Kamali, et al. Hermes: Dynamic Partitioning for Distributed Social Network Graph Databases. In *EDBT*, pages 25–36, 2015.
- [148] Objectivity Inc. ThingSpan. Available at <https://www.objectivity.com/products/thingspan/>.
- [149] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *USENIX ATC*, 1999.
- [150] Ontotext. GraphDB. Available at <https://www.ontotext.com/products/graphdb/>.
- [151] OpenLink. Virtuoso. Available at <https://virtuoso.openlinksw.com/>.
- [152] Oracle. Oracle Spatial and Graph. Available at <https://www.oracle.com/database/technologies/spatialandgraph.html>.
- [153] K. Ousterhout, R. Rasti, S. Ratnasamy, et al. Making Sense of Performance in Data Analytics Frameworks. In *NSDI*, pages 293–307, 2015.
- [154] M. T. Özsu. A Survey of RDF Data Management Systems. *Front. Comput. Sci.*, 10(3):418–432, 2016.
- [155] L. Page, S. Brin, R. Motwani, et al. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford InfoLab, 1999.
- [156] N. Patil, P. Kiran, et al. A Survey on Graph Database Management Techniques for Huge Unstructured Data. *IJECE*, 8(2):1140–1149, 2018.
- [157] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.
- [158] F. Petroni, L. Querzoni, K. Daudjee, et al. HDRF: Stream-Based Partitioning for Power-Law Graphs. In *ACM CIKM*, pages 243–252, 2015.
- [159] H. Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In *ACM SIGMOD*, pages 1–2, 2009.
- [160] J. Pokorný. Graph Databases: Their Power and Limitations. In *CISIM*, pages 58–69, 2015.
- [161] Profium. Profium Sense. Available at <https://www.profium.com/en/>.
- [162] A. Rana. Detailed Introduction: Redis Modules, from Graphs to Machine Learning (Part 1). Available at <https://medium.com/@ashishrana160796/15ce9ff1949f>, 2019.
- [163] M. Raynal. Using Semaphores to Solve the Readers-Writers Problem. In *Concurrent Programming: Algorithms, Principles, and Foundations*, chapter 3.2.4, pages 74–78. 2013.
- [164] Redis Labs. Redis. <https://redis.io/>.
- [165] Redis Labs. RedisGraph. Available at <https://oss.redislabs.com/redisgraph/>.
- [166] C. D. Rickett, U.-U. Haus, J. Maltby, et al. Loading and Querying a Trillion RDF triples with Cray Graph Engine on the Cray XC. In *CUG*, 2018.
- [167] Robert Yokota. HGraphDB. Available at <https://github.com/rayokota/hgraphdb>.
- [168] I. Robinson, J. Webber, and E. Eifrem. Graph Database Internals. In *Graph Databases, 2nd Edition*, chapter 6, pages 149–170. 2015.
- [169] M. A. Rodriguez. The Gremlin Graph Traversal Machine and Language. In *DBPL*, page 1–10, 2015.
- [170] N. P. Roth, V. Trigonakis, S. Hong, et al. PGX.D/Async: A Scalable Distributed Graph Pattern Matching Engine. In *ACM GRADES*, 2017.
- [171] A. Roy, L. Bindschaedler, J. Malicevic, et al. Chaos: Scale-out Graph Processing from Secondary Storage. In *SOSP*, pages 410–424, 2015.
- [172] M. Rudolf et al. The Graph Story of the SAP HANA Database. In *Datenbanksysteme für Business, Technologie und Web*, pages 403–420, 2013.

- [173] S. Sahu, A. Mhedhbi, S. Salihoglu, et al. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *VLDB*, 11(4):420–431, 2017.
- [174] SAP. SAP HANA. Available at <https://www.sap.com/products/hana.html> and <https://help.sap.com/>.
- [175] S. E. Schaeffer. Survey: Graph Clustering. *Comput. Sci. Rev.*, 1(1):27–64, 2007.
- [176] P. Schmid, M. Besta, and T. Hoefler. High-Performance Distributed RMA Locks. In *ACM HPDC*, pages 19–30, 2016.
- [177] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *ACM SIGMOD*, pages 505–516, 2013.
- [178] S. Sharma. *Cassandra Design Patterns*. 2014.
- [179] X. Shi, Z. Zheng, Y. Zhou, H. Jin, L. He, B. Liu, and Q.-S. Hua. Graph Processing on GPUs: A Survey. *ACM CSUR*, 50(6), 2018.
- [180] K. Singh and V. Singh. Graph pattern matching: A brief survey of challenges and research directions. In *INDIACom*, pages 199–204, 2016.
- [181] solid IT gmbh. System Properties Comparison: Neo4j vs. Redis. Available at <https://db-engines.com/en/system/Neo4j%3BRedis>.
- [182] E. Solomonik et al. Scaling Betweenness Centrality Using Communication-Efficient Sparse Matrix Multiplication. In *ACM/IEEE SC*, 2017.
- [183] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen. GraphR: Accelerating Graph Processing Using ReRAM. In *IEEE HPCA*, pages 531–543, 2018.
- [184] Stardog Union. Stardog. Available at <https://www.stardog.com/>.
- [185] B. A. Steer, A. Alnaimi, M. A. B. F. G. Lotz, et al. Cytosm: Declarative Property Graph Queries Without Data Migration. In *ACM GRADES*, 2017.
- [186] W. Sun, A. Fokoue, K. Srinivas, et al. SQLGraph: An Efficient Relational-Based Property Graph Store. In *ACM SIGMOD*, pages 1887–1901, 2015.
- [187] N. Sundaram, N. Satish, M. M. A. Patwary, et al. GraphMat: High Performance Graph Analytics Made Productive. *VLDB*, 8(11):1214–1225, 2015.
- [188] G. Szárnyas et al. An Early Look at the LDSC Social Network Benchmark’s Business Intelligence Workload. In *ACM GRADES-NDA*, 2018.
- [189] A. Tate, A. Kamil, A. Dubey, A. Größlinger, B. Chamberlain, et al. Programming Abstractions for Data Locality. In *PADAL Workshop*, 2014.
- [190] The Apache Software Foundation. Apache Jena TBD. Available at <https://jena.apache.org/documentation/tdb/index.html>.
- [191] The Linux Foundation. JanusGraph. Available at <http://janusgraph.org/>.
- [192] Y. Tian et al. IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. In *ACM SIGMOD*, pages 345–359, 2020.
- [193] TigerGraph. TigerGraph. Available at <https://www.tigergraph.com/>.
- [194] Twitter. FlockDB. Available at <https://github.com/twitter-archive/flockdb>.
- [195] A. Vaikuntam and V. K. Perumal. Evaluation of Contemporary Graph Databases. In *ACM COMPUTE*, 2014.
- [196] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: A Property Graph Query Language. In *ACM GRADES*, 2016.
- [197] VelocityDB Inc. VelocityDB. Available at <https://velocitydb.com/>.
- [198] VelocityDB Inc. VelocityGraph. Available at <https://velocitydb.com/VelocityEngine.aspx>.
- [199] WhiteDB Team. WhiteDB. Available at <http://whitedb.org/> or <https://github.com/priitj/whitedb>.
- [200] K. Xirogiannopoulos, V. Srinivas, and A. Deshpande. GraphGen: Adaptive Graph Processing Using Relational Databases. In *ACM GRADES*, 2017.
- [201] D. Yan, Y. Bu, Y. Tian, A. Deshpande, and J. Cheng. Big Graph Analytics Systems. In *ACM SIGMOD*, pages 2241–2243, 2016.
- [202] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. TripleBit: A Fast and Compact System for Large Scale RDF Data. *VLDB*, 6(7):517–528, 2013.
- [203] K. Zhao and J. X. Yu. All-in-One: Graph Processing in RDBMSs Revisited. In *ACM SIGMOD*, pages 1165–1180, 2017.
- [204] X. Zhu et al. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *VLDB*, 13(7):1020–1034, 2020.
- [205] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao. GStore: A Graph-Based SPARQL Query Engine. *VLDB Journal*, 23(4):565–590, 2014.