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

MACIEJ BESTA, ROBERT GERSTENBERGER, and EMANUEL PETER, Department of Computer Science, ETH Zurich, Switzerland
MARC FISCHER, PRODYNA (Schweiz) AG, Switzerland
MICHAŁ PODSTAWSKI, Future Processing, Poland
CLAUDE BARTHELS, GUSTAVO ALONSO, and TORSTEN HOEFLER, Department of Computer Science, ETH Zurich, Switzerland

Numerous irregular graph datasets, for example social networks or web graphs, may contain even trillions of edges. Often, their structure changes over time and they 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 and irregularity of such datasets, 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., document stores, tuple stores, native graph database systems, or object-oriented systems), the associated graph models (e.g., Resource Description Framework 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 Atomicity, Consistency, Isolation, Durability). Fifty-one graph database systems are presented and compared, including Neo4j, OrientDB, and Virtuoso. We outline graph database queries and relationships with associated domains (NoSQL stores, graph streaming, and dynamic graph algorithms). Finally, we outline future research and engineering challenges related to 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

Authors' addresses: M. Besta, R. Gerstenberger, E. Peter, C. Barthels, and T. Hoefler, ETH Zürich, Institut für Computing Platforms, OAT S 17, Andreasstrasse 5, 8092 Zürich, Switzerland; emails: maciej.besta@inf.ethz.ch, gerstenberger.robert@gmail.com, peterem@student.ethz.ch, claudeb@inf.ethz.ch, torsten.hoefler@inf.ethz.ch; G. Alonso, ETH Zürich, Dep. of Computer Science, Institut für Computing Platforms, STF J 420, Stampfenbachstrasse 114, 8092 Zürich, Switzerland; email: alonso@inf.ethz.ch; M. Fischer, PRODYNA (Schweiz) AG, Grosspeteranlage 29, 4052 Basel; email: marc.fischer@prodyna.com; M. Podstawski, Future Processing, Bojkowska 37a, 44-100 Gliwice; email: mpodstawski@gmail.com.

0360-0300/2023/09-ART31 $15.00
https://doi.org/10.1145/3604932

ACM Reference format:
Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2023. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *ACM Comput. Surv.* 56, 2, Article 31 (September 2023), 40 pages.
https://doi.org/10.1145/3604932

1 INTRODUCTION

Graph processing is behind numerous problems in computing, for example in medicine, machine learning, computational sciences, and others [113, 135]. Graph algorithms are inherently difficult to design because of challenges such as large sizes of processed graphs, little locality, or irregular communication [135]. 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 databases (GDBs) such as Neo4j [175] 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.

In general, we provide the following contributions:

- We provide the first taxonomy of graph databases,[1] identifying and analyzing key dimensions in the design of graph database systems.
- 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 Related Surveys

There exist several surveys dedicated to the theory of graph databases. In 2008, Angles et al. [9] 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. [8] 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, Angles and Gutierrez provided an overview [10] of basic notions in the graph database landscape. While being related to our work, it is largely orthogonal, focusing on historical developments

[1]Lists of graph databases can be found at
http://nosql-database.org
https://database.guide
https://www.g2.com/categories/graph-databases
https://www.predictiveanalyticstoday.com/top-graph-databases
https://db-engines.com/en/ranking/graph+dbms.

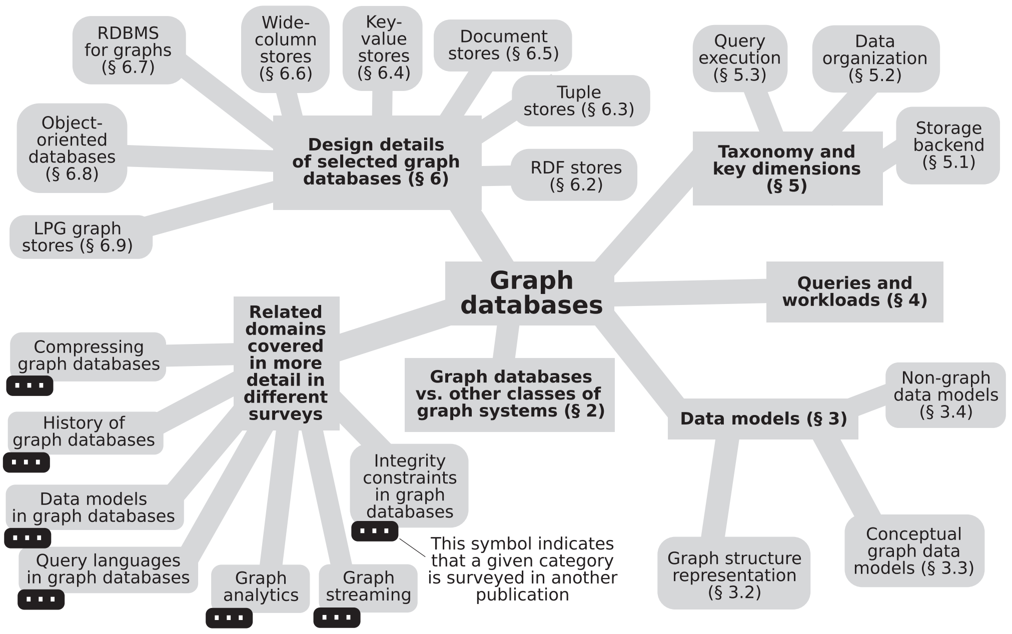
Graph databases
Design details of selected graph databases (§ 6)
RDBMS for graphs (§ 6.7)
Wide-column stores (§ 6.6)
Key-value stores (§ 6.4)
Document stores (§ 6.5)
Tuple stores (§ 6.3)
RDF stores (§ 6.2)
Object-oriented databases (§ 6.8)
LPG graph stores (§ 6.9)
Taxonomy and key dimensions (§ 5)
Query execution (§ 5.3)
Data organization (§ 5.2)
Storage backend (§ 5.1)
Queries and workloads (§ 4)
Data models (§ 3)
Non-graph data models (§ 3.4)
Graph structure representation (§ 3.2)
Conceptual graph data models (§ 3.3)
Graph databases vs. other classes of graph systems (§ 2)
Related domains covered in more detail in different surveys
Compressing graph databases
History of graph databases
Data models in graph databases
Query languages in graph databases
Graph analytics
Graph streaming
Integrity constraints in graph databases
This symbol indicates that a given category is surveyed in another publication

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

(which we exclude), details of many graph database models, and query languages (we only focus on the ones routinely supported by existing graph database systems), and it only sketches at a very high level a few selected graph database systems. Our work, instead, focuses primarily on graph database systems and the details of their design and analyzes in depth all other aspects (graph data models, query languages, queries) *through the perspective of being supported in these systems.* Also in 2018, Bonifati et al. [41] 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 [60, 83, 96] and **Resource Description Framework (RDF)** [161]. 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 [116, 118, 126, 164, 167], a survey of graph processing ubiquity [180], and performance evaluations of a few systems [125, 142, 203]).

2 GRAPH DATABASES AND OTHER CLASSES OF GRAPH SYSTEMS

Graph database systems are described in the literature as *"systems specifically designed for managing graph-like data following the basic principles of database systems, i.e., persistent data storage, physical/logical data independence, data integrity, and consistency"* [10]. However, other systems can also store and process dynamic graphs. We now briefly discuss relations to three such classes: other classes of databases, streaming graph frameworks, and general static graph processing systems.

2.1 Graph Databases vs. NoSQL Stores and Other Databases

NoSQL stores address various deficiencies of relational database systems, such as little support for flexible data models [60]. 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* [175]. Other

Table 1. The Most Relevant Symbols and Abbreviations Used in This Work

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, $\mathbf{M}$	The Adjacency Matrix representation. $\mathbf{M} \in \{0,1\}^{n,n}$, $\mathbf{M}_{u,v} = 1 \Leftrightarrow (u, v) \in E$.
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 (Section 3.3.2) and Resource Description Framework (Section 3.3.4).
KV, RDBMS	Key–Value store (Section 6.4) and Relational Database Management Systems (Section 6.7).
OODBMS	Object-Oriented Database Management Systems (Section 6.8).
OLTP, OLAP	Online Transaction Processing (Section 4.1) and Online Analytics Processing (Section 4.1).
ACID	Transaction guarantees (Atomicity, Consistency, Isolation, Durability).

types of NoSQL systems include *wide-column stores*, *document stores*, and general **key–value (KV)** stores [60]. We focus on both "native" graph databases such as Neo4j [175] and on other systems used specifically for maintaining graphs (relational databases, object-oriented databases, NoSQL, and others).

2.2 Graph Databases vs. Graph Streaming Frameworks

In *graph streaming* [30], 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 (LPG)** [8] or Resource Description Framework [57]) where both vertices and edges may be of different types and may be associated with arbitrary properties. Contrarily, graph streaming frameworks focus on simple graph models where edges or vertices may have weights and, in some cases, simple additional properties such as timestamps. Moreover, challenges in the design of graph databases include transactional support, persistence, physical/logical data independence, data integrity, or consistency; these topics are little related to graph streaming frameworks.

2.3 Graph Databases vs. Static Graph Processing Systems

A lot of effort has been dedicated to static graph analytics [27, 65, 97, 143, 186, 209]. 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 Section 3.3). Moreover, static graph processing systems do not focus on topics such as transactions, persistence, physical/logical data independence, data integrity, or consistency.
Graph streaming frameworks and static graph processing systems are *not* covered in this work.

3 GRAPH 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.

3.1 Simple Graph Model

We start with a simple graph model that is a basis for more complex and richer conceptual graph models used in graph databases. 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, then 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); $w : E \to \mathbb{R}$ maps edges to weights.

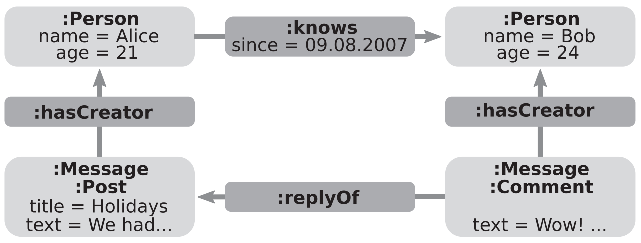

Fig. 2. The illustration of an example LPG. Vertices and edges can have labels (bold, prefixed with colon) and properties (key = value). We present a subgraph of a social network, where a person can know other persons, post messages, and comment on others' messages.

3.2 Fundamental Representations of Graph Structure

We also summarize two fundamental ways to represent the structure of connections between vertices. Two common such graph representations of vertex neighborhoods are the **adjacency matrix format (AM)** and the **adjacency list format (AL)**.

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 $v \in A_u \Leftrightarrow (u,v) \in E$.

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

3.3 Conceptual Graph Data Models Used in Graph Databases

We now introduce the conceptual graph models used by the surveyed systems; these models extend the simple graph model from Section 3.1. A simple graph model is often used in graph frameworks such as Pregel [136] or STINGER [70]. However, it is not commonly used with graph databases.

3.3.1 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 (Section 6.4.2) that focuses on storing and querying hypergraphs.

3.3.2 Labeled Property Graph Model. The classical graph model, a tuple $G = (V,E)$, is adequate for many problems such as computing vertex centralities [42]. However, it is not rich enough to model various real-world problems. This is why graph databases often use the LPG, sometimes simply called a property graph [8, 41]. 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* [41] (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 [41]. 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, and $p_E(e)$ denotes the set of property key–value pairs of the edge e. An example LPG is in Figure 2. Note that, in LPGs, E may be a multi-set (i.e., there may be more than a single edge between vertices, even having identical labels and/or key–value sets). All systems considered in this work use some variant of the LPG, with the exception of RDF systems or when explicitly discussed.

3.3.3 Variants of Labeled Property Graph Model. Several databases support variants of LPG. First, Neo4j [175] (a graph database described in detail in Section 6.9.1) supports an arbitrary number of labels for vertices. However, it only allows for one label (called the *edge-type*) per edge. Next, ArangoDB [16] (a graph database described in detail in Section 6.5.2) 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 [8] 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 [50].

3.3.4 Resource Description Framework. The *RDF* [57] is a collection of specifications for representing information. It was introduced by the World Wide Web Consortium 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 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

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

where 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 [103, 161].

3.3.5 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 [112] 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 3 shows an example of transforming an LPG

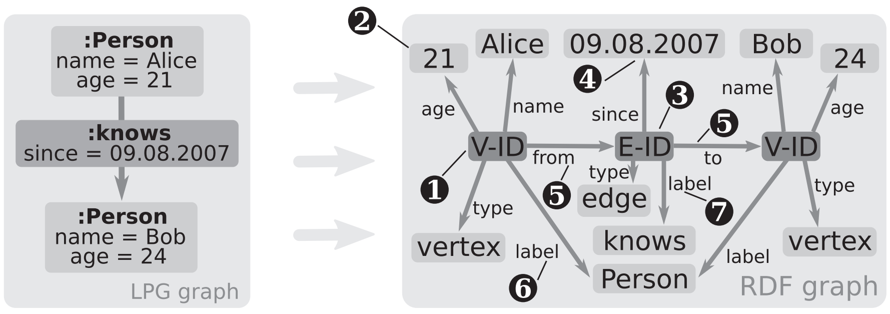

Fig. 3. Comparison of an LPG and an RDF graph: a transformation from LPG to RDF. “V-ID,” “E-ID,” “age,” “name,” “type,” “from,” “to,” “since,” and “label” are RDF URIs. Numbers in black circles refer to transformation steps in Section 3.3.5.

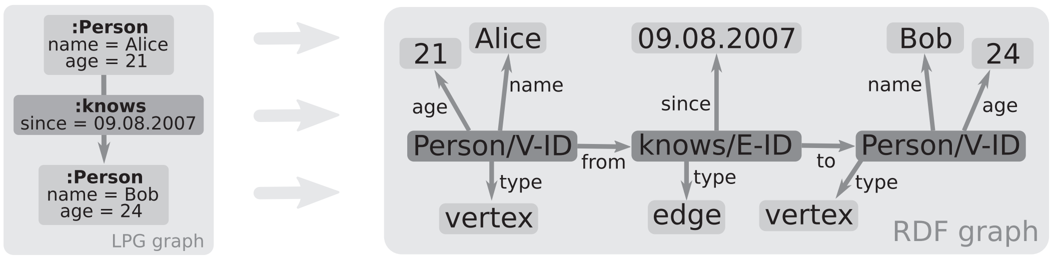

Fig. 4. Comparison of an LPG and an RDF graph: a transformation from LPG to RDF, given vertices and edges have only one label. “Person/V-ID,” “knows/E-ID,” “age,” “name,” “type,” “from,” “to,” and “since” are RDF URIs.

graph into RDF triples. More details on transformations between LPG and RDF are provided by Hartig [101].

If all vertices and edges only have one label, then 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 4.

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 that is bipartite at the cost of an increased graph size [103]. Details and examples are provided in a report by Hayes [103].

3.4 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. [60]. Thus, we omit extensive discussions and instead offer brief summaries, *focusing on how they are used to model or represent graphs.*

3.4.1 Collection of Key–Value Pairs. Key–value stores are the simplest NoSQL stores [60]. Here the data are 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 schemaless). However, a key–value store can also impose some additional data layout, structuring the schemaless values [60].

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 [62, 110, 172, 184] in Section 6.4. For example, one can use vertex labels as keys and encode the neighborhoods of vertices as values.

3.4.2 Collection of Documents. A document is a fundamental storage unit in a class of NoSQL databases called document stores [60]. 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 [43] or XML [44]. 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 [16, 46, 78, 131, 148] (Section 6.5), 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, are 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.

3.4.3 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 that 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 are specific to different databases; we describe an example [207] in Section 6.3.

3.4.4 Collection of Tables. Tables are the basis of **Relational Database Management Systems (RDBMS)** [20, 55, 104]. 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 [208, 211]. We present and analyze different graph database systems [21, 159] based on tables in Section 6.6 and Section 6.7.

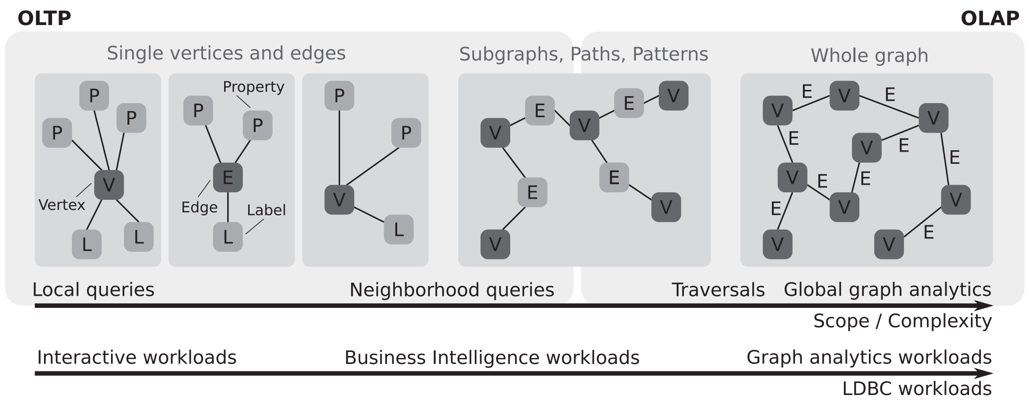

Fig. 5. Illustration of *different query scopes* and their relation to other graph query taxonomy aspects, in the context of accessing a Labeled Property Graph.

3.4.5 Collection of Objects. One can also use collections of objects in **Object-Oriented Database Management Systems (OODBMS)** [19] 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 [206] in Section 6.8.

4 QUERIES AND WORKLOADS IN THE LANDSCAPE OF GRAPH DATABASES

We describe graph database workloads.

4.1 OLAP and OLTP

First, one distinguishes between **Online Transactional Processing (OLTP)** and **Online Analytical Processing (OLAP)**. OLTP queries are small, interactive, transactional, and local in scope (i.e., they process only a small part of the graph). Examples are neighborhood queries, lookups, inserts, deletes, and updates of single (or a few) vertices and edges. OLAP queries are usually not processed at interactive speeds, as they are inherently complex and global in scope (i.e., they span the whole graphs). Examples are PageRank [163] or **Breadth-First Search (BFS)**.

Now, static graph processing systems (Section 2.3) focus on OLAP. However, many graph databases also support a rich set of OLAP workloads. This includes Neo4j [175], Cray Graph Engine [173], Amazon Neptune [5], TigerGraph [200], and many others (see Tables 2–3). For example, Neo4j provides algorithms for vertex centrality (e.g., PageRank, Betweenness Centrality, Eigenvector Centrality), community detection (e.g., Louvain, Triangle Counting, Weakly Connected Components, and Label Propagation), graph traversals (BFS and DFS), shortest paths (e.g., Delta Stepping, A*), and many others. *Thus, we focus on both OLTP and OLAP in the context of how they are supported by graph databases.*

4.2 Graph Queries beyond OLAP vs. OLTP

We also offer an analysis of graph queries beyond the simple distinction into OLAP and OLTP classes. Figure 5 illustrates the 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 [7, 8], analytics workloads [13], benchmarks related to certain aspects [53, 133, 134], and whole systems [6, 18, 25, 49, 75, 111, 115, 193] and surveys on system performance [66, 142]).

4.2.1 Scopes of Graph Queries. We describe queries in the increasing order of their scope. We focus on the LPG model, see Section 3.3.2. Figure 5 depicts the scope of graph queries.

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 verifying whether a given vertex or a given edge have a given label (given the label name) or whether they have a property of a specified type. These queries are used in social network workloads [18, 25] (e.g., to fetch the profile information of a user) and in benchmarks [115] (e.g., to measure the vertex look-up time).

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 [18, 25].

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 [53, 66, 115].

Global Graph Analytics. Finally, we identify global graph analytics queries, which by definition consider the whole graph (not necessarily every property but all vertices and edges). Different benchmarks [28, 49, 66, 142] take these large-scale queries into account, since they are used in different fields such as threat detection [69] or computational chemistry [24]. As indicated in Tables 2 and 3, many graph databases support such queries.

4.2.2 Classes of Graph Workloads. We also outline an existing taxonomy of graph database workloads that is provided as a part of the LDBC benchmarks [6]. 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*.

Interactive Workloads. A part of LDBC called the Social Network Benchmark [75] 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, modify, or remove 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, or friendship removal.

Business Intelligence Workloads. Next, LDBC identifies **Business Intelligence (BI)** workloads [193], 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.

Graph Analytics Workloads. Finally, the LDBC effort comes with a graph analytics benchmark [111], 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, weakly connected components [89], community detection using label propagation [40], deriving the local clustering coefficient [183], and computing single-source shortest paths [64].*

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*.

4.2.3 Graph Patterns and Navigational Expressions. Angles et al. [8] 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*. Next, *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 TAXONOMY OF GRAPH DATABASE SYSTEMS

We now describe how we categorize *graph database systems* considered in this survey. This taxonomy incorporates existing concepts related to graph data models (cf. Section 3) and to graph queries (cf. Section 4). Then, other aspects of the proposed taxonomy are novel. In this section, we describe the taxonomy in a general way. In Section 7, we analyze the taxonomy in the context of specific graph database systems. The main dimensions of the taxonomy are (1) the general backend type, (2) data organization, and (3) query execution. Figure 6 illustrates the general types of considered databases together with certain aspects of data models and organization. Figure 7 summarizes all elements of the proposed taxonomy.

5.1 Types of Graph Database Storage Backends

We first identify *general types of graph databases* that primarily differ in their *storage backends* (e.g., a triple store or a document store). This facilitates further taxonomization and analysis, 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.

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, RDBMS, or 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 [60]. Figure 6 illustrates these systems; they are discussed in more detail in Section 6.

5.2 Data Organization

In data organization, we distinguish the following taxonomy aspects: (1) graph structure representation, (2) conceptual data models, (3) indexes, (4) data distribution, and (5) common optimizations.

First, in *graph structure representation*, we analyze whether the graph structure is stored using the AL or the AM representation (see Section 3.2). The graph structure representation directly impacts the performance of queries.

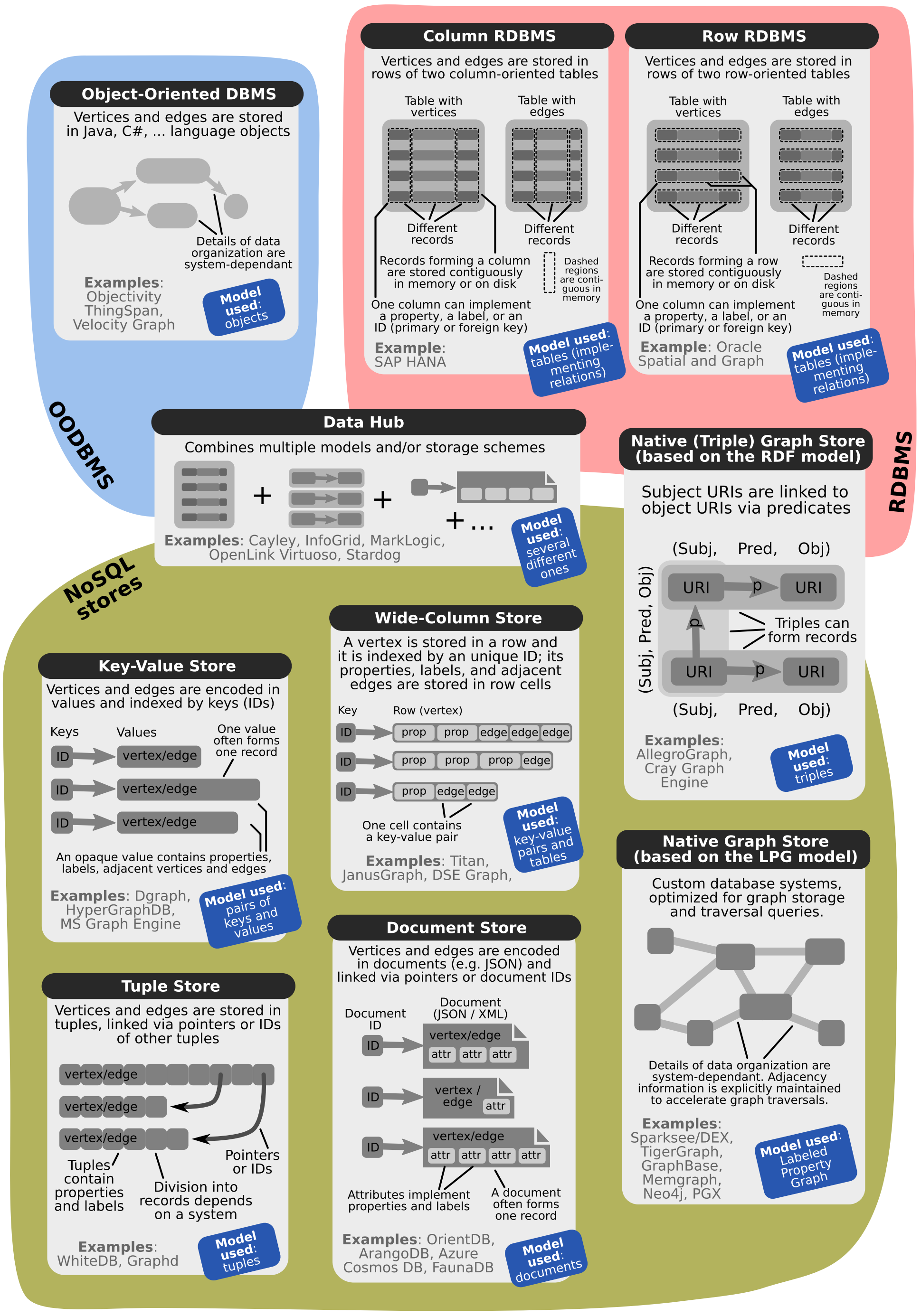
Object-Oriented DBMS
Vertices and edges are stored in Java, C#, ... language objects
Details of data organization are system-dependant
Examples: Objectivity ThingSpan, Velocity Graph
Model used: objects
OODBMS
Column RDBMS
Vertices and edges are stored in rows of two column-oriented tables
Table with vertices
Table with edges
Different records
Records forming a column are stored contiguously in memory or on disk
Dashed regions are contiguous in memory
One column can implement a property, a label, or an ID (primary or foreign key)
Example: SAP HANA
Model used: tables (implementing relations)
Row RDBMS
Vertices and edges are stored in rows of two row-oriented tables
Records forming a row are stored contiguously in memory or on disk
Example: Oracle Spatial and Graph
RDBMS
Data Hub
Combines multiple models and/or storage schemes
Examples: Cayley, InfoGrid, MarkLogic, OpenLink Virtuoso, Stardog
Model used: several different ones
Native (Triple) Graph Store (based on the RDF model)
Subject URIs are linked to object URIs via predicates
(Subj, Pred, Obj)
URI p URI
Triples can form records
Examples: AllegroGraph, Cray Graph Engine
Model used: triples
NoSQL stores
Key-Value Store
Vertices and edges are encoded in values and indexed by keys (IDs)
Keys Values
One value often forms one record
ID vertex/edge
An opaque value contains properties, labels, adjacent vertices and edges
Examples: Dgraph, HyperGraphDB, MS Graph Engine
Model used: pairs of keys and values
Wide-Column Store
A vertex is stored in a row and it is indexed by an unique ID; its properties, labels, and adjacent edges are stored in row cells
Key Row (vertex)
ID prop prop edge edge edge
One cell contains a key-value pair
Examples: Titan, JanusGraph, DSE Graph,
Model used: key-value pairs and tables
Native Graph Store (based on the LPG model)
Custom database systems, optimized for graph storage and traversal queries.
Details of data organization are system-dependant. Adjacency information is explicitly maintained to accelerate graph traversals.
Examples: Sparksee/DEX, TigerGraph, GraphBase, Memgraph, Neo4j, PGX
Model used: Labeled Property Graph
Tuple Store
Vertices and edges are stored in tuples, linked via pointers or IDs of other tuples
vertex/edge
Tuples contain properties and labels
Division into records depends on a system
Pointers or IDs
Examples: WhiteDB, Graphd
Model used: tuples
Document Store
Vertices and edges are encoded in documents (e.g. JSON) and linked via pointers or document IDs
Document ID
Document (JSON / XML)
ID vertex/edge attr attr attr
Attributes implement properties and labels
A document often forms one record
Examples: OrientDB, ArangoDB, Azure Cosmos DB, FaunaDB
Model used: documents

Fig. 6. Overview of different *graph storage backends*, with examples.

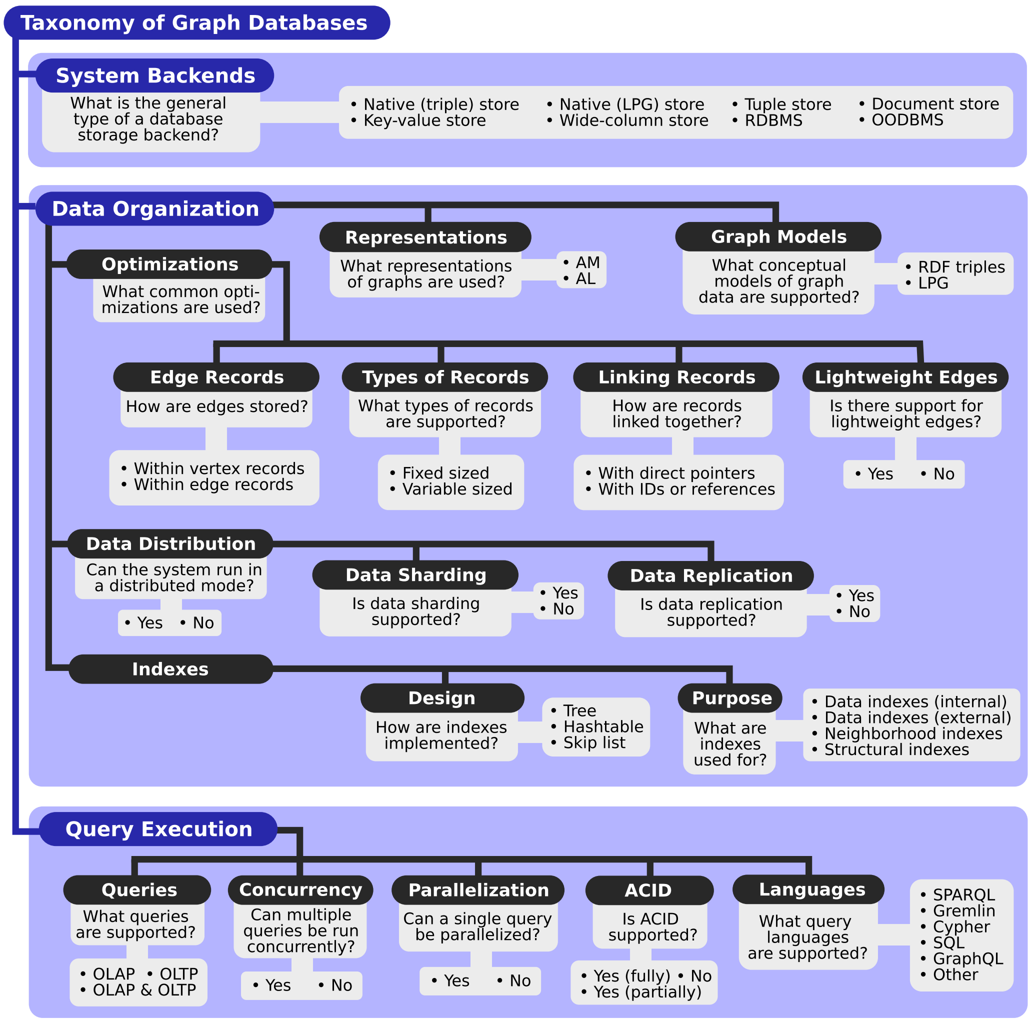
Taxonomy of Graph Databases
System Backends
What is the general type of a database storage backend?
• Native (triple) store • Key-value store
• Native (LPG) store • Wide-column store
• Tuple store • RDBMS
• Document store • OODBMS
Data Organization
Representations
What representations of graphs are used?
• AM • AL
Graph Models
What conceptual models of graph data are supported?
• RDF triples • LPG
Optimizations
What common optimizations are used?
Edge Records
How are edges stored?
• Within vertex records • Within edge records
Types of Records
What types of records are supported?
• Fixed sized • Variable sized
Linking Records
How are records linked together?
• With direct pointers • With IDs or references
Lightweight Edges
Is there support for lightweight edges?
• Yes • No
Data Distribution
Can the system run in a distributed mode?
• Yes • No
Data Sharding
Is data sharding supported?
• Yes • No
Data Replication
Is data replication supported?
• Yes • No
Indexes
Design
How are indexes implemented?
• Tree • Hashtable • Skip list
Purpose
What are indexes used for?
• Data indexes (internal) • Data indexes (external) • Neighborhood indexes • Structural indexes
Query Execution
Queries
What queries are supported?
• OLAP • OLTP • OLAP & OLTP
Concurrency
Can multiple queries be run concurrently?
• Yes • No
Parallelization
Can a single query be parallelized?
• Yes • No
ACID
Is ACID supported?
• Yes (fully) • No • Yes (partially)
Languages
What query languages are supported?
• SPARQL • Gremlin • Cypher • SQL • GraphQL • Other

Fig. 7. Overview of the *identified taxonomy of graph databases*.

Second, we investigate what *conceptual data models* are supported by different graph databases. Here we focus on the RDF and LPG models, as well as on their variants, described in Section 3.3. 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.

We also analyze *how graph databases use indexes* to accelerate accessing data. Indexes can significantly improve the performance of GDBs, and they are widely used, for example, in RDF systems [2, 123, 169]. Here we consider the functionality (i.e., the use case) of a given index and how a given index is implemented.[2] As for the former, we identify four different index use cases: storing the locations of vertex neighborhoods (referred to as "neighborhood indexes"); indexing graph elements, such as vertices, that satisfy pre-specified conditions related to rich graph data

[2]We do not include the index information in Tables 2 and 3 because of lack of space and instead provide a detailed separate analysis in Section 7.2.7.

(referred to as "data indexes"); storing the actual graph data; and maintaining non-graph related data (referred to as "structural indexes"). As for the latter, we identify three fundamental data structures used to implemented indexes: trees, skip lists, and hashtables.

We also identify whether a database can run in a *distributed mode*. 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 they may allow for *sharding* [74] (data fragmentation, i.e., storing only a part of the given dataset on one server). Replication often allows for more fault tolerance [73], and sharding reduces the amount of used memory per node and can improve performance [73]. Gathering 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 scale badly for particularly large graphs that would require disk spilling.

Finally, we identify *common optimizations*: (1) dividing data into records, (2) lightweight edges, and (3) linking records with direct pointers.

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 (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.

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 [47].

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 the 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.

Note that specific systems can employ other diverse optimizations. For example, 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 hashtables.

5.3 Query Execution

In query execution, we identify the following aspects: (1) concurrent execution of different queries, (2) parallelization of single queries, (3) **Atomicity, Consistency, Isolation, Durability (ACID)** transactions, (4) support for classes of queries, and (5) support for query languages.

Note that almost all of the studied graph databases are closed source or do not come with any associated discussions of the details of the design of query execution (except for general descriptions). Thus, we do not offer a detailed associated taxonomy for algorithmic aspects of query 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. We also analyze relationships among the backend type, the data organization, and the query execution. 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.

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.

Many graph databases support *transactions*; we analyze them in Section 7.3.2. ACID [105] 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.

We also analyze *supported queries* in Section 7.3.3. Some databases (e.g., ArangoDB [16]) are oriented towards OLTP, where focus is on executing many smaller, interactive, transactional queries. Other systems (e.g., Cray Graph Engine [173]) focus more on OLAP: They execute analytics queries that span the whole graphs, usually taking more time than OLTP. Finally, different databases (e.g., Neo4j [175]) offer extensive support for both.

Although we do not focus on graph database languages, we also report on *supported query languages* in Section 7.3.4). We consider the leading languages such as SPARQL [165], Gremlin [176], Cypher [81, 93, 106], and SQL [59]. We also mention other system-specific languages such as GraphQL [102] and support for APIs from languages such as C++ or Java.[3] Note that mapping graph queries to SQL was also addressed in past work [191].

6 ANALYSIS OF 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 section.[4] "?" 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 Section 7.

6.1 Discussion on Selection Criteria

When selecting systems for consideration in the survey, we use two criteria. First, we use the DB-Engines Ranking[5] to select the most popular systems in each considered backend category. We also pick interesting research systems (e.g, SQLGraph [192], LiveGraph [212], or Weaver [67]) that 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).

6.2 RDF Stores (Triple Stores)

RDF stores [1, 45, 98, 153], also called triple stores, implement the RDF model (Section 3.3.4). These systems organize data into triples. We now describe in more detail a selected recent RDF store, Cray Graph Engine (Section 6.2.1). We also provide more details on two other systems, AllegroGraph and BlazeGraph, focusing on *variants of the RDF model* used in these systems (Section 6.2.2).

[3]We bring to the reader's attention a manifesto on creating GQL, a standardized graph query language (https://gql.today).
[4]We 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.
[5]https://db-engines.com/en/ranking/graph+dbms.

Table 2. Comparison of Graph Databases (Native Graph Databases based on RDF and LPG, Key–Value Stores, and Document Stores)

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) (Section 6.2). The main data model used: **RDF triples** (Section 3.3.4).																				
AllegroGraph [82]	✗	✗	■	✗	✗	■*	✗	✗	✗	✗	✗	■	■	■	■	✗	■	■	?	*Triples are stored as integers (RDF strings map to integers).
BlazeGraph [37]	✗	■*	■*	✗	✗	?	?	✗	✗	✗	✗	■	■	■	?	?	■	?	?	*BlazeGraph uses RDF*, an extension of RDF (details in Section 6.2.2).
Cray Graph Engine [173]	✗	✗	■	✗	✗	✗*	✗*	✗	✗	✗	✗	■	✗	■	✗	■	✗	✗	■	*RDF triples are stored in hashtables.
Amazon Neptune [5]	✗	■	■	✗	✗	?	?	✗	✗	✗	✗	■	■	✗	■	✗	■	■	■	—
AnzoGraph [48]	✗	■	■	✗	✗	?	?	✗	✗	✗	✗	■	✗	■	■	■	■	■	■	—
Apache Jena TBD [197]	✗	✗	■	?	✗	?	?	?	?	?	?	✗	?	✗	■	✗	■	■	?	—
Apache Marmotta [14]	✗	✗	■	✗	✗	■*	✗	✗	✗	✗	✗	?	?	?	■	■	■	■	■	*The structure of data records is based on that of different RDBMS systems (H2 [151], PostgreSQL [150], MySQL [68]).
BrightstarDB [152]	✗	✗	■	✗	✗	?	?	✗	✗	✗	✗	?	?	?	■	?	■	■	?	—
gStore [213]	✗	✗	■	■	✗	✗	■	✗	✗	■	✗	?	?	✗	?	✗	?	?	?	—
Ontotext GraphDB [157]	✗	✗	■	✗	✗	?	?	✗	✗	✗	✗	■	■	✗	■	?	■	■	?	—
Profium Sense [168]	✗	✗	■*	✗	✗	?	?	✗	✗	✗	✗	■	■	?	■	?	■	■	?	*The format used is called JSON-LD: JSON for vertices and RDF for edges.
TripleBit [210]	✗	✗	■	✗	✗	✗	■*	✗	✗	✗	✗	✗‡	✗	✗	✗	✗	?	?	■	The data organization uses compression. *Strings map to variable size integers. ‡Described as future work.
NATIVE GRAPH DATABASES (LPG model based) (Section 6.9). The main data model used: **LPG** (Sections 3.3.2, 3.3.3).																				
Neo4j [175]	✗	■	✗	■	✗	■	✗	■	■	✗	✗	■	■	✗	■	✗	■	■	■	Neo4j is provided as a cloud service by a system called Graph Story [92].
Sparksee/DEX [139]	✗	■	✗	✗*	✗	✗‡	✗‡	✗	✗	✗	✗	■	■	✗	■	■	■	■	■	*Bitmaps are used for connectivity. ‡The system uses maps only.
GBase [119]	✗	✗*	✗	✗‡	■	✗	✗	✗	✗	✗	✗	■	?	?	?	?	?	✗	■	*GBase supports simple graphs only (Section 3.1). ‡GBase stores the AM sparsely.
GraphBase [76]	✗	✗*	✗	?	✗	✗	■	?	?	?	?	■	?	■	■	?	■	■	?	*No support for edge properties, only two types of edges available.
Graphflow [120]	✗	■	✗	■	✗	?	?	?	?	?	?	✗	?	?	?	?	?	?	■	—
LiveGraph [212]	✗	■	✗	■	✗	✗	■	✗	■	✗	✗	✗	?	✗	■	?	■	■	■	—
Memgraph [144]	✗	■	✗	■	✗	?	?	?	?	?	?	■	■	■	◧*	◧‡	■	■	■	*This feature is under development. ‡Available only for some algorithms.
TigerGraph [200]	✗	■	✗	?	✗	?	?	?	?	?	?	■	■	■	■	■	■	■	■	—
Weaver [67]	✗	■	✗	?	✗	?	?	?	?	?	?	■	■	■	■	■	■	■	■	—
KEY-VALUE STORES (Section 6.4). The main data model used: **key–value pairs** (Section 3.4.1).																				
HyperGraphDB [110]	✗	✗*	✗	✗‡	✗	✗	■	✗	■	✗	✗	■	■	■	■	■	◧†	■	■	*A Hypergraph model. ‡The system uses an incidence index to retrieve edges of a vertex. †Support for ACI only.
MS Graph Engine [184]	■	■	✗	■*	✗	✗	■‡	✗	■	✗	■	■	✗	■	■	■	✗	■	■	*AL contains IDs of edges and/or vertices. ‡Schema is defined by Trinity Specification Language (TSL).
Dgraph [62]	✗	■	✗	■	✗	✗	■	✗	■	✗	✗	■	■	■	■	■	■	■	■	Dgraph is based on Badger [61].
RedisGraph [170, 172, 187]	✗	■	✗	✗	✗	✗	■	✗	✗	✗	✗	■	■	✗	■	✗	✗	✗	■*	RedisGraph is based on Redis [171]. *The OLAP part uses GraphBLAS [122].
DOCUMENT STORES (Section 6.5). The main data model used: **documents** (Section 3.4.2).																				
ArangoDB [16]	■	■	✗	✗*	✗	✗	■	✗	■	✗	✗	■	■	■	■	✗	■	■	◧	*Uses a hybrid index for retrieving edges.
OrientDB [46]	■	■	✗	■*	✗	✗	■	■	■	✗	■	■	■	■‡	■	■	■	■	✗	*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 [148]	■	■	✗	✗	✗	✗	■	✗	■	✗	✗	■	■	■	■	✗	■	■	?	—
Bitsy [131]	✗	■	✗	✗	✗	✗	■	✗	■	✗	✗	✗	✗	✗	■	✗	■	■	✗	The system is disk based and uses JSON files. The storage only allows for appending data.
FaunaDB [78]	■*	■	✗	■‡	✗	✗	■	■	■	✗	✗	■	■	■	■	✗	■	■	✗	*Document, RDBMS, graph, "time series". ‡Adjacency lists are separately precomputed.

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.

6.2.1 *Cray Graph Engine.* **Cray Graph Engine (CGE)** [173] 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 of which they are a part. 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.

Table 3. Comparison of Graph Databases (RDBMS, Wide-Column Stores, Tuple Stores, OODBMS, and Data Hubs)

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) (Section 6.7). The main data model used: **tables (implementing relations)** (Section 3.4.4).																				
Oracle Spatial and Graph [159]	■*	■	■	✗	✗	?*	?*	✗	■*	✗	✗	■	■	■	■	■	■	■	■	*LPG and RDF use row-oriented storage. The system can also run on top of PGX [107] (effectively as a **native graph database**).
AgensGraph [36]	■	■	✗	✗	✗	?	?	✗	?	✗	✗	■	■	■	■	■	■	■	■	AgensGraph is based on PostgreSQL.
FlockDB [202]	✗	✗	✗	✗	✗	?	?	✗	?	✗	✗	■	■	■	■	✗	✗	■	✗	The system focuses on "shallow" graph queries, such as finding mutual friends.
IBM Db2 Graph [199]	✗	■	✗	✗	✗	?	?	✗	■	■*	■*	■‡	■‡	■‡	■‡	■‡	■‡	■	?	*can store vertices/edges in the same table. ‡inherited from the underlying IBM Db2™.
MS SQL Server 2017 [149]	■	■	✗	✗	✗	?	?	✗	?	✗	✗	■	■	■	■	■	■	■	■	The system uses an SQL graph extension.
OQGRAPH [137]	✗	■	✗	✗	✗	?*	?*	✗	■*	✗	✗	■	■	■	■	✗	■	■	?	OQGRAPH uses MariaDB [26]. *OQGRAPH uses row-oriented storage.
SAP HANA [181]	■	■	✗	✗	✗	✗*	✗*	✗	✗*	✗	✗	■	■	■	■	■	■	■	■	*SAP HANA is column-oriented, edges and vertices are stored in rows. SAP HANA can be used with a dedicated graph engine [179]; it offers some capabilities of a JSON document store [181]
SQLGraph [192]	✗	■	✗	✗	✗	?	■*	✗	■‡	■	✗	■†	■†	■†	■†	■†	■†	■	?	*SQLGraph uses JSON for property storage. ‡SQLGraph uses row-oriented storage. †depends on the used SQL engine.
WIDE-COLUMN STORES (Section 6.6). The main data model used: **key–value pairs** and **tables** (Sections 3.4.1, 3.4.4).																				
JanusGraph [21]	✗	■	✗	■	✗	✗	■	✗	✗	■	✗	■	■	■	■	■	■	■	■	JanusGraph is the continuation of Titan.
Titan [21]	■	■	✗	■	✗	✗	■	✗	✗	■	✗	■	■	■	■	■	■	■	■	Enables various backends (e.g., Cassandra [130]).
DSE Graph (DataStax) [58]	✗	■	✗	■	✗	✗	■	✗	✗	■	✗	■	■	■	■	?	◧*	■	■	DSE Graph is based on Cassandra [130]. *Support for AID, Consistency is configurable.
HGraphDB [174]	✗	■	✗	✗	✗	✗	■	✗	■	✗	✗	■	■	■	?	?	✗*	■	■	HGraphDB uses TinkerPop3 with HBase [87]. *ACID is supported only within a row.
TUPLE STORES (Section 6.3). The main data model used: **tuples** (Section 3.4.3).																				
WhiteDB [207]	✗	✗	◧*	◧‡	✗	✗	■*	■	◧‡	◧‡	◧‡	✗	✗	✗	■	■	◧†	■	?	*Implicit support for triples of integers. ‡Implementable by the user. †Transactions use a global shared/exclusive lock.
Graphd [91]	✗	✗	◧*	✗	✗	?	?	?	■	✗	✗	■	■	?	■	?	◧‡	?	?	Backend of Google Freebase. *Implicit support for triples. ‡Subset of ACID.
OBJECT-ORIENTED DATABASES (OODBMS) (Section 6.8). The main data model used: **objects** (Section 3.4.5).																				
Velocity-Graph [206]	■	■	✗	■	✗	✗	■	■	■	✗	✗	■	■	■	■	■	■	■	■	The system is based on VelocityDB [205]
Objectivity ThingSpan [155]	✗	■	✗	?	?	?	?	?	■	?	?	■	■	■	■	■	■	■	■	The system is based on ObjectivityDB [95].
DATA HUBS. The main data model used: **several different ones**.																				
MarkLogic [138]	■	✗*	■	✗	✗	?	■	✗	■*	✗	✗	■	■	■	■	?	■	■	■	Supported storage/models: relational tables, RDF, various documents. *Vertices are stored as documents, edges are stored as RDF triples.
OpenLink Virtuoso [158]	■	✗	■	✗	✗	?	?	✗	✗	✗	✗	■	■	■	■	◧*	■	■	■	Supported storage/models: relational tables and RDF triples. *This feature can be used relational data only.
Cayley [51]	■	■	■	?	?	?	■	?	?	?	?	■	■	✗	■	?	■*	?	?	Supported storage/models: relational tables, RDF, document, key–value. *This feature depends on the backend.
InfoGrid [109]	■	■	✗	?	?	?	■	?	■	✗	✗	■	■	■	■	?	◧*	?	?	Supported storage/models: relational tables, Hadoop's filesystem, grid storage. *A weaker consistency model is used instead of ACID.
Stardog [190]	◧	■*	◧*	✗	✗	?	■	✗	■*	✗	✗	■	■	✗	■	?	■	■	■	Supported storage/models: relational tables, documents. *RDF is simulated on relational tables. Both LPG and RDF are enabled through virtual quints.

Bolded systems are described in more detail in the corresponding sections. All columns and symbols are explained in the caption of Table 2.

6.2.2 *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 8 presents an example RDF graph with such attributes. This figure uses the same LPG graph as in previous examples provided in Figures 3 and 4, which contain example transformations from the LPG into the original RDF model.

Second, BlazeGraph [37] implements *RDF** [99, 100], an augmentation of RDF that allows for attaching triples to triple predicates (see Figure 9). Vertices can use triples for storing labels and

Table 4. Support for Different Graph Database Query Languages in Different Graph Database Systems

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) (Section 6.2).							
AllegroGraph	●	●	✖	✖	✖	✖	✖
Amazon Neptune	●	●	✖	✖	✖	✖	✖
AnzoGraph	●	✖	●	✖	✖	✖	✖
Apache Jena TDB	●	✖	✖	✖	✖	● (Java)	✖
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	●	✖	✖	✖	✖	✖	✖
Profium Sense	●	✖	✖	✖	✖	✖	✖
TripleBit	●	✖	✖	✖	✖	✖	✖
NATIVE GRAPH DATABASES (LPG model based) (Section 6.9).							
Gbase	✖	✖	✖	●	✖	✖	✖
GraphBase	✖	✖	✖	✖	✖	✖	GraphBase uses its native query language.
Graphflow	✖	✖	◐*‡	✖	✖	✖	*Graphflow supports a subset of Cypher [147]. ‡Graphflow supports Cypher++ extension with subgraph-condition-action triggers [120].
LiveGraph	✖	✖	✖	✖	✖	✖	No focus on languages and queries.
Memgraph	✖	✖	●*	✖	✖	✖	*openCypher.
Neo4j	✖	◐*	●	✖	●‡	◐†	*Gremlin is supported as a part of TinkerPop integration. ‡GraphQL supported with the GRANDstack layer. †Neo4j can be embedded in Java applications.
Sparksee/DEX	✖	●	✖	✖	✖	● (.NET)*	*Sparksee/DEX also supports C++, Python, Objective-C, and Java APIs.
TigerGraph	✖	✖	✖	✖	✖	✖	TigerGraph uses GSQL [200].
Weaver	✖	✖	✖	✖	✖	● (C)*	*Weaver also supports C++, Python.
TUPLE STORES (Section 6.3).							
Graphd	✖	✖	✖	✖	✖	✖	Graphd uses MQL [91].
WhiteDB	✖	✖	✖	✖	✖	● (C)*	*WhiteDB also supports Python.
DOCUMENT STORES (Section 6.5).							
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	●	●	●	●*	✖	● (Java)‡	*An SQL extension for graph queries. ‡OrientDB offers bindings to C, JavaScript, PHP, .NET, Python, and others.
KEY-VALUE STORES (Section 6.4).							
Dgraph	✖	✖	✖	✖	●*	✖	*A variant of GraphQL.
HyperGraphDB	✖	✖	✖	✖	✖	● (Java)	✖
MS Graph Engine	✖	✖	✖	✖	✖	✖	MS Graph Engine uses LINQ [184].
RedisGraph	✖	✖	●	✖	✖	✖	✖
WIDE-COLUMN STORES (Section 6.6).							
DSE Graph (DataStax)	✖	●	✖	✖	✖	✖	DSE Graph also supports CQL [58].
HGraphDB	✖	●	✖	✖	✖	✖	✖
JanusGraph	✖	●	✖	✖	✖	✖	✖
Titan	✖	●	✖	✖	✖	✖	✖
RELATIONAL DBMS (RDBMS) (Section 6.7).							
AgensGraph	✖	✖	●*	●‡	✖	✖	*A variant called openCypher [94, 140]. ‡ANSI-SQL.
FlockDB	✖	✖	✖	●	✖	✖	FlockDB uses the Gizzard framework and MySQL.
IBM Db2 Graph	✖	◐*	✖	●	✖	● (Java)‡	*IBM Db2 Graph supports only graph queries whose 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 [204], an SQL-like graph query language.
SAP HANA	✖	✖	✖	●*	✖	●‡	*SAP HANA offers bindings to Rust, ODBC, and others. ‡GraphScript, a domain-specific graph query language.
SQLGraph	✖	◐*	✖	◐‡	✖	✖	*SQLGraph doesn't support Gremlin side effect pipes. ‡Graph is encoded in a way specific to SQLGraph.
OBJECT-ORIENTED DATABASES (OODBMS) (Section 6.8).							
Objectivity ThingSpan	✖	✖	✖	✖	✖	✖	Objectivity ThingSpan uses a native DO query language [155].
VelocityGraph	✖	✖	✖	✖	✖	● (.NET)	✖
DATA HUBS.							
Cayley	✖	●*	✖	✖	●	✖	*Cayley supports Gizmo, a Gremlin dialect [51]. Cayley also uses MQL [51].
InfoGrid	✖	✖	✖	✖	✖	● (REST)	✖
MarkLogic	✖	✖	✖	✖	✖	✖	MarkLogic uses XQuery [39].
OpenLink Virtuoso	●	✖	✖	●	✖	✖	OpenLink Virtuoso also supports XQuery [39], XPath v1.0 [54], and XSLT v1.0 [121].
Stardog	●*	●	✖	✖	●	✖	*Stardog supports the Path Query extension [190].

“**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.

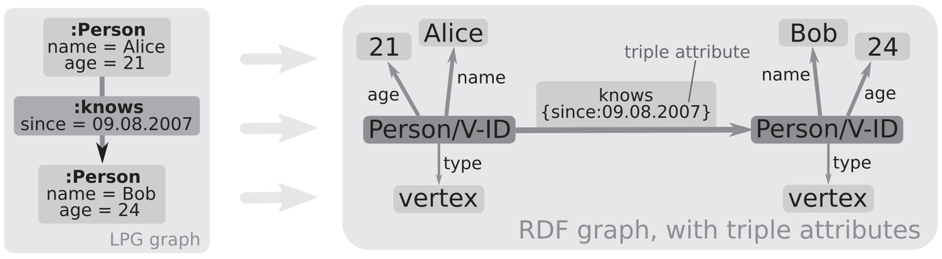

Fig. 8. Comparison of an LPG graph and an RDF graph: a transformation from LPG to RDF *with triple attributes*. We represent the triple attributes as a set of key–value pairs. "Person/V-ID," "age," "name," "type," and "knows" are RDF URIs. The transformation uses the assumption that there is one label per vertex and edge.

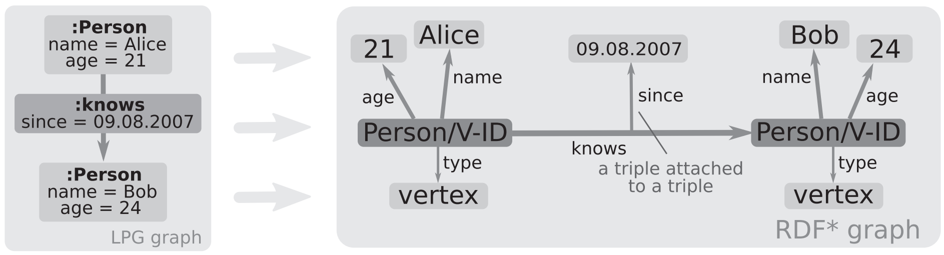

Fig. 9. Comparison of an LPG graph and an RDF* graph: a transformation from LPG to RDF* that enables *attaching triples to triple predicates*. "Person/V-ID," "age," "name," "type," "since," and "knows" are RDF URIs. The transformation uses the assumption that there is one label per vertex and edge.

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.

6.3 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 Section 3.4.3.

6.3.1 WhiteDB. WhiteDB [207] 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 [145, 207].

6.4 Key–Value Stores

One can also explicitly use KV stores for maintaining a graph (cf. Section 3.4.1).

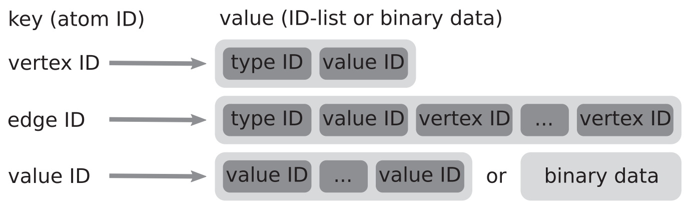

Fig. 10. An example utilization of key–value stores for maintaining hypergraphs in HyperGraphDB (a type is a term used in HyperGraphDB to refer to a label).

6.4.1 Microsoft's Graph Engine (Trinity). Microsoft's Graph Engine [184] 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, then such an edge (together with the associated data) can also be stored in a separate dedicated cell.

6.4.2 HyperGraphDB. HyperGraphDB [110] stores hypergraphs (definition in Section 3.3.1). 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 10 shows an example of how a KV store is used to represent a hypergraph in HyperGraphDB.

6.5 Document Stores

In document stores, a fundamental storage unit is a document, described in Section 3.4.2. We select two document stores for a more detailed discussion, OrientDB and ArangoDB.

6.5.1 OrientDB. In OrientDB [46], 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 [46] 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, however, 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 stores not only 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.

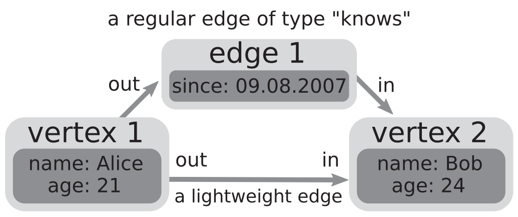

Fig. 11. Two vertex documents connected with *a lightweight edge and a regular edge (knows) in OrientDB.*

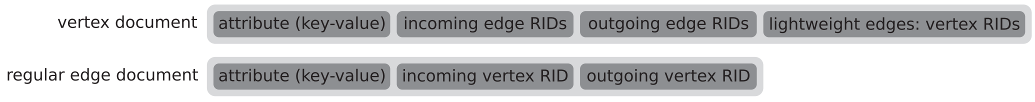

Fig. 12. Example OrientDB *vertex and edge documents (complex JSON documents are also supported).*

Figure 11 contains an example of using documents for representing vertices, regular edges, and lightweight edges in OrientDB. Figure 12 shows example vertex and edge documents.

6.5.2 ArangoDB. ArangoDB [16, 17] keeps its documents in a *binary format* called *VelocyPack*, which is a compacted implementation of JSON documents. Documents can be stored in different collections and have a *_key* attribute that is a unique ID within a given collection. Unlike OrientDB, these IDs are no direct memory pointers. For maintaining graphs, ArangoDB uses *vertex 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 [17].

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").

6.6 Wide-Column Stores

Wide-column stores combine different features of key–value stores and relational tables. On the 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 13.

6.6.1 Titan and JanusGraph. Titan [21] and its continuation JanusGraph [198] are built on top of wide-column stores. They can use different wide-column stores as backends, for example, Apache Cassandra [12]. In both systems, when storing a graph, each row represents a vertex. Each vertex

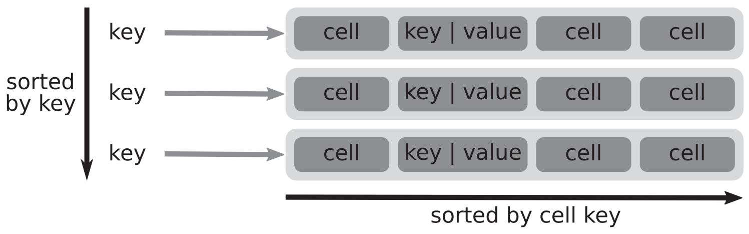

Fig. 13. An illustration of wide-column stores: mapping keys to rows and column-keys to cells within the rows.

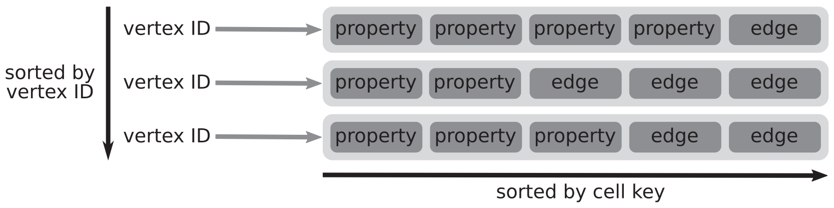

Fig. 14. An illustration of Titan and JanusGraph: using wide-column stores for storing graphs. The illustration is inspired by and adapted from the work by Sharma [185].

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 14. Since rows are ordered by the key, both systems straightforwardly partition tables into so-called *tablets*, which can then be distributed over multiple servers.

6.7 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 Section 3.4.4.

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. In many row RDBMS, data items in memory are kept in contiguous rows [22, 177]. Column RDBMS, however, 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 [159], OQGRAPH built on MariaDB [137]) and column RDBMS (e.g., SAP HANA [181]).

6.7.1 Oracle Spatial and Graph. Oracle Spatial and Graph [159] 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.

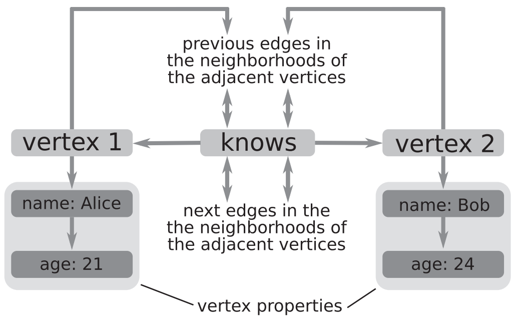
previous edges in the neighborhoods of the adjacent vertices
vertex 1
knows
vertex 2
name: Alice
age: 21
next edges in the the neighborhoods of the adjacent vertices
name: Bob
age: 24
vertex properties

Fig. 15. 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.

6.8 Object-Oriented Databases

OODBMS [19] enable modeling, storing, and managing data in the form of *language objects* used in object-oriented programming languages. We summarize such objects in Section 3.4.5.

6.8.1 VelocityGraph. VelocityGraph [206] is a graph database relying on the VelocityDB [205] 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, 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.

6.9 LPG-based Native Graph Databases

Graph database systems described in the previous sections with the exception of triple stores 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.

6.9.1 Neo4j: Direct Pointers. Neo4j [175] is the most popular graph database system, according to different database rankings (see the links provided in the introduction). 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, then 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 15 outlines the Neo4j design; Figure 16 shows the details of vertex and edge records.

A core concept in Neo4j is using *direct pointers* [175]: 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

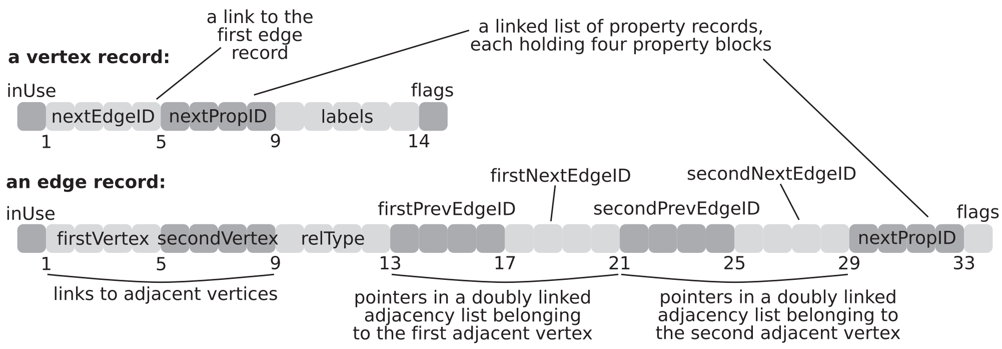
a vertex record:
a link to the first edge record
a linked list of property records, each holding four property blocks
inUse
flags
nextEdgeID
nextPropID
labels
1
5
9
14
an edge record:
firstNextEdgeID
secondNextEdgeID
inUse
firstPrevEdgeID
secondPrevEdgeID
flags
firstVertex
secondVertex
relType
nextPropID
1
5
9
13
17
21
25
29
33
links to adjacent vertices
pointers in a doubly linked adjacency list belonging to the first adjacent vertex
pointers in a doubly linked adjacency list belonging to the second adjacent vertex

Fig. 16. An overview of the Neo4j vertex and edge records.

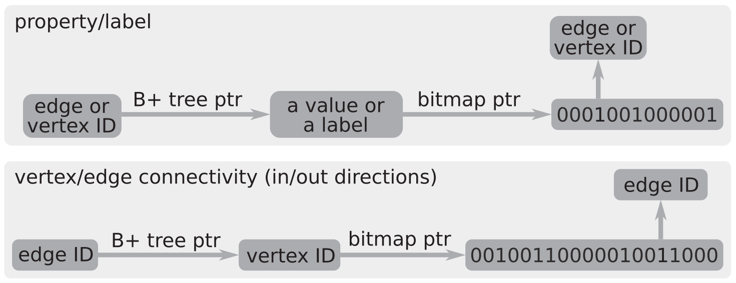

Fig. 17. Sparksee maps for properties, labels, and vertex/edge connectivity. All mappings are bidirectional.

query complexity does not dependent on the graph size. Instead, it only depends on how large the visited subgraph is.[6]

6.9.2 Sparksee/DEX: B+ Trees and Bitmaps. Sparksee is a graph database system that was formerly known as DEX [139]. 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 17 illustrates example mappings.

Sparksee is one of the few systems that are *not* record based. Instead, Sparksee uses *maps* implemented as B+ trees [56] 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," then 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,

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

they are cut into 32-bit *clusters*. If a cluster contains a non-zero bit, then 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.

7 TAKEAWAYS, INSIGHTS, FUTURE DIRECTIONS

We now offer different insights about the described systems, considering both practitioners and researchers. We interleave these insights with suggestions for future developments and research.

7.1 Discussion, Takeaways, and Insights on Data Organization

We discuss the data organization aspects of our taxonomy with respect to specific graph databases.

7.1.1 Conceptual Graph Models. 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 in different scenarios [175]. Still, it is not standardized, and there are many variants (cf. Section 3.3.3); However, it is becoming standardized in the upcoming SQL/PGQ and GQL standards from ISO [80]. Some systems limit the number of labels to just 1. 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 Section 3.3.5. 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 Section 6.2.2).

Among native graph databases, while no LPG focused system simultaneously supports RDF, some RDF systems (e.g., Amazon Neptune) also support LPG. Further, there has been recent work on unifying RDF and LPG [11, 85, 132]. 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 datasets than the RDF datasets with the respective non-graph data models such as a collection of documents.

Another interesting challenge is to understand better the design tradeoffs between LPG and RDF. For example, it is unclear which one is more advantageous for different workload classes under different design constraints (disk vs. in-memory representation, distributed vs. shared-memory, replicated vs. sharded, etc.). This could be achieved by developing formal runtime and storage models followed by an extensive evaluation.

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. Thus, these models are of less relevance to practitioners, but they offer a potentially interesting direction for researchers. Especially in the context of hypergraphs, there has been a recent proliferation of novel schemes in other domains of graph processing, such as graph learning [23], suggesting that exploring hypergraphs for graph databases may be a timely direction.

7.1.2 Graph Structure Representations. Many graph databases use variants of AL, since it makes traversing neighborhoods efficient and straightforward [175]. 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 $\mathcal{O}(n^2)$ space, especially for sparse graphs. Systems using AM focus on compression of the adjacency matrix, trying to mitigate storage and query overheads (e.g., GBase [119]). Thus, as with hypergraphs, the AM representation is of less relevance to practitioners. However, the recent explosion of the popularity of graph analytics based on linear algebra (e.g., GraphBLAS [122] and CAGNET [201]) suggests that exploring AM in the context of graph databases may be a useful direction, especially with respect to high-performance OLAP analytics that very often could directly use the graph AM representation.

7.2 Discussion, Takeaways, and Insights on Data Optimizations

We discuss separately common optimizations in data organization.

7.2.1 Using Existing Storage Designs. Most graph database systems are built 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. Overall, there is a lot of research potential in more efficient and more effective use of existing storage designs for graph databases. For example, a promising direction would be to investigate how recent RDBMS development, such as worst-case optimal joins [154], could be used for graph related workloads.

7.2.2 Data Layout of Records. The details of record-based data organization heavily depend on a specific system. For example, an RDBMS 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), and 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 as records. 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).

The records used by the studied graph databases may be unstructured (i.e., not having a prespecified format such as JSON), as is the case with KV 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.

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. 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).

Overall, most systems use records to store vertices, most often one vertex per one record. Some systems store edges in separate records, and others store them together with the adjacent vertices. 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. We observe that these design choices are made arbitrarily. We conclude that an interesting research direction would be developing formal performance models and using them to guide the most advantageous design choice for each type of storage backend for graph databases.

7.2.3 Adjacencies between Records. 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).

Using direct pointers can accelerate graph traversals [175], as additional index traversals are avoided. Another option is to assign each record a unique ID and use these IDs instead of direct pointers to refer to other records. On the 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, then it is usually easier to update the indexing structure instead of changing all associated direct pointers, which may come with significant overhead [17]. Here one could also make these design choices and tradeoffs more accurate by designing performance models to guide them.

7.2.4 Storing Data Directly in Indexes. Sometimes graph data are 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 [197]. HyperGraphDB uses a key–value index, namely Berkeley DB [156], to access its physical storage. This approach also enables sharing primitive data values with a reference count so that multiple identical values are stored only once [110].

7.2.5 Storing Strings. Many systems, for example RDFs, *heavily use strings*, for example for keeping IDs of different parts of graph datasets. Thus, there are different schemes or maintaining strings. First, many systems support both fixed-size and variable-size strings. This enables more performance: The former (that encode, e.g., IDs) can be kept and accessed rapidly in dedicated structures that assume certain specific sizes, while the latter (that encode, e.g., arbitrary text items) often use separate dynamic structures that are slower to access but offer more flexibility [175, 207]. Such dynamic stores can be both in-memory structures, but they could even be separate files [175].

The considered systems offer other string related optimizations. For example, CGE optimizes how 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.

RDF systems also harness external dedicated libraries for more effective string management. For example, the RDF String library [166] facilitates constructing RDF strings. Specifically, it automatically generates string representations of specified values, together with appropriate URI prefixes. This library can be used to even encode multi-line strings or conduct string interpolation with embedded expressions. Another example is RDF String Turtle [195], a package that enables conversions between the string-based and RDF JSON representations of RDF.

7.2.6 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. 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 and thus more attractive for industry purposes and for practitioners, simultaneously implying research potential in the native graph database 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, then it is the responsibility of the user to partition the collection to avoid any additional overheads. Thus, it would be interesting to research how to automatize sharding of single document collections, or even single (large) documents. 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.

Here another interesting research opportunity would be to accelerate graph database workloads such as OLAP by harnessing *partial* data replication. Specifically, many OLAP graph database workloads such as BFS can be expressed with linear algebra operations [122], and these workloads could benefit from the partial replication of the adjacency matrix, for example as done in two-and-a-half-dimensional (2.5D) and 3D matrix multiplications [188, 189].

7.2.7 Indexes. Most graph database systems use indexes. 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 [175].

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 [21]. While JanusGraph allows multiple vertex-centric indexes per vertex, each optimized for different conditions, which are then chosen by the query optimizer, simpler solutions 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 [212]. Graphflow indexes the neighbors of a vertex into forward and backward adjacency lists, where each list is first partitioned by the edge label and, second, by the label of the neighbor vertex [120]. Another example is Sparksee, which uses various different index structures to find the adjacent vertices and properties of a vertex [139].

An interesting research opportunity is to design indexes for richer "higher-order" structural information beyond plain neighborhoods. Specifically, a recent wave of pattern matching schemes [35, 146] indicates the importance of higher-order graph structure, such as triangles to which each vertex belongs. Indexing such information would significantly speed up queries related to clique mining, dense subgraph discovery, clustering, and many others. While some work has been done in this respect [182], many designs could be proposed.

Data indexes concern data beyond the neighborhood information, and they can be used to accelerate query plan generation and query execution. It is possible, for example, to index all

Table 5. Support for Different Index Implementations in Different Graph Database Systems

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 hashtable index
VelocityGraph	🔋*	✖	✖	*B-tree
Virtuoso	🔋*	✖	✖	*2D R-tree
WhiteDB	🔋*	✖	✖	*T-tree

"🔋": A system supports a given index implementation. "✖": A system does not support a given index implementation.

vertices that have a specific property (value). They are usually employed to speed up Business Intelligence workloads (details on workloads are in Section 4). 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 [210]. 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 [213]. Some database systems like Amazon Neptune [5] or AnzoGraph [48] 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 data indexes. Some of them, like Azure Cosmos DB [148], 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 [21] use internal indexing for label- and value-based lookups but rely on external indexing backends (e.g., Elasticsearch [72] or Apache Solr [15]) for non-trivial lookups involving multiple properties, ranges, or full-text search.

Structural indexes are used for various internal data. Here LiveGraph uses a vertex index to map its vertex IDs to a physical storage location [212]. ArangoDB uses a hybrid index, a hashtable, to find the documents of incident edges and adjacent vertices of a vertex [16].

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. A research opportunity, useful especially for practitioners, would be to conduct a detailed and broad performance comparison of different index implementations for different workloads.

7.2.8 Data Organization vs. Database 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 are 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. However, if one only wants to retrieve single properties about vertices or edges, then 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 [6, 125, 141, 142, 203] 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 [141]. It underlies the potential and research opportunities for developing hybrid database systems for graph workloads, that could combine the advantages of databases using non-graph data models and of native graph stores.

7.3 Discussion and Takeaways on Query Execution

We discuss the query execution aspects of our taxonomy with respect to the specific graph databases. Our discussion is by necessity brief, as most systems do not disclose this information.[7]

7.3.1 Concurrency and Parallelization. We start with concurrency and parallelization.

Support for Concurrency and Parallelism in OLTP Queries. We conclude that (1) almost all systems support concurrent OLTP queries, and (2) in almost all classes of systems, fewer systems support parallel OLTP 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 the 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. With the growing importance of more complex OLTP workloads and the ongoing process of blurring the difference between complex OLTP and Business Intelligence workloads, putting more focus on parallel OLAP designs is becoming a more attractive and urgent research direction [194].

Support for Concurrency and Parallelism in OLAP Queries. OLAP queries are usually parallelized. This is because such queries often involve all the vertices and edges, making the sequential execution prohibitively long. Moreover, parallelization of such queries is facilitated by the fact that there exists a plethora of related work, due to the prevalence of certain OLAP queries (e.g., traversals, centralities) in static graph analytics [27, 34, 117, 122, 143]. At the same time, concurrent execution of different OLAP queries is supported only to some degree. For example, Weaver supports concurrent large-scale OLAP queries such as BFS, but each such query processes an immutable separate snapshot of the graph dataset. One could attempt to investigate how to effectively run concurrent OLAP queries (as well as large-scale read-only Business Intelligence workloads that have similar characteristics), which potentially only needs certain amount of basic state bookkeeping.

Support for Concurrent OLAP and OLTP Queries. Concurrent execution of OLAP and OLTP queries is not widely supported. Systems that support multi-versioning, such as LiveGraph or Weaver, run an OLAP query on a consistent graph snapshot, and current OLTP queries modify different vertex and/or edge versions or introduce newer versions. Other systems such as Neo4j, while allowing concurrent OLTP and OLAP queries, leave dealing with inconsistencies of OLAP queries to the client as the default isolation level (read-committed) does not protect such queries from modification by other queries or even offers repeatable reads. This aspect is potentially rich

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

in research and new design opportunities, as it requires fundamental understanding of different effects that could occur in native graph stores when executing concurrent OLAP and OLTP, analogously to the effects happening at different isolation levels in RDBMS systems.

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 [207] that 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 Section 7.3.2. Here, an interesting research opportunity would be to harness lock-free synchronization protocols known from parallel computing [38] when implementing different fine-gained OLTP queries.

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 [88]. For example, Trinity can be deployed on InfiniBand [108] to leverage Remote Direct Memory Access [88]. 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 [88].

Here a promising research opportunity is to harness established optimized communication routines such as collectives [52] for large-scale OLAP and BI.

Other Execution Optimizations. The considered databases come with numerous other system-specific design optimizations. For example, an optimization in ArangoDB's design allows us to skip accessing the vertex document 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 [17]. 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 the possibility to switch the underlying relational storage with the native graph storage provided by the PGX processing engine [71, 107, 178]. 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 [107]. Furthermore, some systems use the LPG specifics to implement OLAP queries more effectively. For example, to implement BFS, Weaver uses special designated properties associated with vertices to indicate, whether a vertex has already been already visited. Such special properties are not visible to an external user of the graph database and are usually deallocated after a given query is finalized.

A lot of work has been done into optimizing distributed-memory algebraic operations such as matrix products using optimal communication routines [86, 90, 127–129, 188, 189]. It would be interesting to investigate how such routines can be used to speed up OLAP graph queries.

There is a large body of existing work in the design of dynamic graph processing frameworks [30]. These systems differ from graph databases in several aspects, for example they often employ simple graph models (and not LPG or RDF). 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, especially in the context of raw graph updates per second [142, 203]. This suggest that harnessing mechanisms used in such frameworks in the context of graph databases could significantly enhance the performance of the latter and is a potentially fruitful research direction.

Furthermore, while there exists past research into the impact of the underlying network on the performance of a distributed graph analytics framework [160], 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 [33, 124] and routing architectures [32].

Finally, 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 [29], designs related to network interface cards such as SmartNICs [63], or processing in memory [4].

7.3.2 ACID. We also discuss various aspects of ACID transactions. ACID transactions are usually used to implement OLTP queries. OLAP queries are read-only analytics and thus are less relevant for the notion of ACID. For example, in Weaver, OLAP workloads are referred to as "node programs" and are treated differently from transactions, which are used to implement OLTP queries.

Support. 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.

Implementation. Two important ways to implement transactions are through locking or timestamps. Neo4j uses write locks to protect modifications until they are committed. Weaver uses timestamps to reorder transactions into an serializable order if necessary. Other systems such as LiveGraph combine both ways and use timestamps to select the correct version of a vertex/edge (solving simple read/write conflicts) and vertex locks to deal with concurrent writes.

7.3.3 Support for OLAP and OLTP Queries. 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. [211] 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. [208] 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.

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, and Graphflow.

There also exist many OLAP graph workloads that have been largely unaddressed by the design and performance analyses of existing graph database systems. This includes vertex reordering problems (e.g., listing vertices by their degeneracy) or optimization (e.g., graph coloring) [31]. There problems were considered in the context of graph algorithms processing simple graphs, and incorporating rich models such as RDF would further increase complexity and offer many associated research challenges, for example designing indexes, data layouts, or distribution strategies.

7.3.4 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)** [184]. 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 [204], 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, 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.

7.4 Insights for Practitioners

An important question on whether RDBMS or non-relational native graph backends are more suitable for graph workloads is far from being fully answered. Several analyses [3, 77, 114, 162, 199]—including very recent ones [196]—indicate better performance of RDBMS over native graph database designs. However, these analyses focus on systems designed with a single class of workloads in mind (e.g., OLAP analytics) and on homogeneous graphs without rich additional label and property data. Thus, they are not conclusive for more realistic scenarios where a mix of OLAP, OLTP, and BI workloads runs over rich LPG datasets. This is supported by another recent study, which states that *"the workloads (...) require several storage and processing features that existing RDBMSs are generally not optimized for"* [79].

In general, whenever the data schema is known in advance, RDBMS—with its long-standing history of optimizations for such cases—would be a preferable choice. Contrarily, when the data schema is not known, graph databases would most probably offer more performance.

Overall, systems based on non-graph data models, such as RDBMS or—to a certain degree—others (e.g., document stores) offer most mature designs and well-understood behavior related to different isolation levels. As such, these systems are the best option when one needs a system that offers predictable behavior in the first place, and reasonable performance for standard workloads. However, when one aims at highest performance of purely graph workloads, it is worth considering native graph stores. This is especially the case for the most recent graph workload classes that are only now being introduced in the GDB landscape, such as subgraph queries [146].

8 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.

REFERENCES

[1] Daniel J. Abadi et al. 2007. Scalable semantic web data management using vertical partitioning. In *VLDB*. 411–422.
[2] Sunitha Abburu and Suresh Babu Golla. 2015. Effective partitioning and multiple RDF indexing for database triple store. *Eng. J.* 19, 5 (2015), 139–154.
[3] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, et al. 2017. EmptyHeaded: A relational engine for graph processing. *ACM Trans. Database Syst.* 42, 4, Article 20 (2017), 44 pages. https://doi.org/10.1145/3129246
[4] Junwhan Ahn et al. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *ACM ISCA*. 105–117. https://doi.org/10.1145/2749469.2750386
[5] Amazon. Amazon Neptune. Retrieved from https://aws.amazon.com/neptune/.
[6] Renzo Angles et al. 2014. The Linked Data Benchmark Council: A graph and RDF industry benchmarking effort. *ACM SIGMOD Rec.* 43, 1 (2014), 27–31.
[7] Renzo Angles, Marcelo Arenas, Pablo Barcelo, et al. 2018. G-CORE: A core for future graph query languages. In *ACM SIGMOD*. 1421–1432.
[8] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, et al. 2017. Foundations of modern query languages for graph databases. *ACM Comput. Surv.* 50, 5, Article 68 (2017), 40 pages. https://doi.org/10.1145/3104031
[9] Renzo Angles and Claudio Gutierrez. 2008. Survey of graph database models. *ACM Comput. Surv.* 40, 1, Article 1 (2008), 39 pages. https://doi.org/10.1145/1322432.1322433
[10] Renzo Angles and Claudio Gutierrez. 2018. An introduction to graph data management. In *Graph Data Management, Fundamental Issues and Recent Developments*. 1–32.
[11] Renzo Angles, Aidan Hogan, Ora Lassila, et al. 2022. Multilayer graphs: A unified data model for graph databases. In *ACM GRADES-NDA*. Article 11, 6 pages.
[12] Apache. Apache Cassandra. Retrieved from https://cassandra.apache.org/.
[13] Apache. Apache Giraph. Retrieved from https://giraph.apache.org/.
[14] Apache. Apache Mormotta. Retrieved from http://marmotta.apache.org/.
[15] Apache. Apache Solr. Retrieved from https://solr.apache.org/.
[16] ArangoDB Inc. ArangoDB. Retrieved from https://www.arangodb.com/docs/stable/data-models.html.
[17] ArangoDB Inc. ArangoDB: Index free adjacency or hybrid indexes for graph databases. Retrieved from https://www.arangodb.com/2016/04/index-free-adjacency-hybrid-indexes-graph-databases/.

[18] Timothy G. Armstrong et al. 2013. LinkBench: A database benchmark based on the Facebook social graph. In *ACM SIGMOD*. 1185–1196.

[19] Malcolm Atkinson, David DeWitt, David Maier, François Bancilhon, et al. 1990. The object-oriented database system manifesto. In *DOOD*. 223–240. https://doi.org/10.1016/B978-0-444-88433-6.50020-4

[20] Paolo Atzeni and Valeria De Antonellis. 1993. *Relational Database Theory*.

[21] Aurelius. Titan data model. Retrieved from http://s3.thinkaurelius.com/docs/titan/1.0.0/data-model.html.

[22] AWS. 2022. Columnar storage developer guide. Retrieved from https://docs.aws.amazon.com/redshift/latest/dg/c_columnar_storage_disk_mem_mgmnt.html.

[23] Song Bai, Feihu Zhang, and Philip H. S. Torr. 2021. Hypergraph convolution and hypergraph attention. *Pattern Recogn.* 110 (2021), 107637. https://doi.org/10.1016/j.patcog.2020.107637

[24] Alexandru T. Balaban. 1985. Applications of graph theory in chemistry. *J. Chem. Inf. Comput. Sci.* 25, 3 (1985), 334–343.

[25] Sumita Barahmand and Shahram Ghandeharizadeh. 2013. BG: A benchmark to evaluate interactive social networking actions. In *CIDR*.

[26] Daniel Bartholomew. 2012. Mariadb vs. MySQL. *Dostopano* 7, 10 (2012), 2014.

[27] Omar Batarfi et al. 2015. Large scale graph processing systems: Survey and an experimental evaluation. *Cluster Computing* 18, 3 (2015), 1189–1213.

[28] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. arXiv:1508.03619. Retrieved from https://arxiv.org/abs/1508.03619.

[29] Maciej Besta, Dimitri Stanojevic, Johannes De Fine Licht, Tal Ben-Nun, and Torsten Hoefler. Graph processing on FPGAs: Taxonomy, survey, challenges. arXiv:1903.06697. Retrieved from https://arxiv.org/abs/1903.06697.

[30] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2019. Practice of streaming processing of dynamic graphs: Concepts, models, and systems. arXiv:1912.12740. Retrieved from https://arxiv.org/abs/1912.12740.

[31] Maciej Besta et al. 2020. High-performance parallel graph coloring with strong guarantees on work, depth, and quality. In *ACM/IEEE SC*.

[32] Maciej Besta et al. 2021. High-performance routing with multipathing and path diversity in ethernet and HPC networks. *IEEE Trans. Parallel Distrib. Syst.* 32, 4 (2021), 943–959. https://doi.org/10.1109/TPDS.2020.3035761

[33] Maciej Besta and Torsten Hoefler. 2014. Slim Fly: A cost effective low-diameter network topology. In *ACM/IEEE SC*. 348–359. https://doi.org/10.1109/SC.2014.34

[34] Maciej Besta, Michał Podstawski, Linus Groner, Edgar Solomonik, and Torsten Hoefler. 2017. To push or to pull: On reducing communication and synchronization in graph computations. In *ACM HPDC*. 93–104. https://doi.org/10.1145/3078597.3078616

[35] Maciej Besta, Zur Vonarburg-Shmaria, Yannick Schaffner, Leonardo Schwarz, Grzegorz Kwasniewski, Lukas Gianinazzi, Jakub Beranek, Kacper Janda, Tobias Holenstein, Sebastian Leisinger, et al. 2021. GraphMineSuite: Enabling high-performance and programmable graph mining algorithms with set algebra. *Proc. VLDB Endow.* 14, 11 (2021), 1922–1935. https://doi.org/10.14778/3476249.3476252

[36] Bitnine Global Inc. AgensGraph. Retrieved from https://bitnine.net/agensgraph/.

[37] Blazegraph. BlazeGraph DB. Retrieved from https://www.blazegraph.com/.

[38] Guy E. Blelloch and Bruce M. Maggs. 2010. Parallel algorithms. In *Algorithms and Theory of Computation Handbook: Special Topics and Techniques*.

[39] Scott Boag, Don Chamberlin, Mary F Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. 2007. XQuery 1.0: An XML Query Language. World Wide Web Consortium.

[40] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *ACM WWW*. 587–596. https://doi.org/10.1145/1963405.1963488

[41] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. Querying graphs. *Synth. Lect. Data Manage.* 10, 3 (2018), 1–184.

[42] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *J. Math. Sociol.* 25, 2 (2001), 163–177.

[43] Tim Bray. 2014. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159.

[44] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and Franois Yergeau. 2008. Extensible markup language (XML) 1.0. https://www.w3.org/TR/2008/REC-xml-20081126/.

[45] Jeen Broekstra et al. 2002. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *ISWC*. 54–68.

[46] Callidus Software Inc. OrientDB. Retrieved from https://orientdb.com.

[47] Callidus Software Inc. OrientDB: Lightweight edges. Retrieved from https://orientdb.com/docs/3.0.x/java/Lightweight-Edges.html.

[48] Cambridge Semantics. AnzoGraph. Retrieved from https://www.cambridgesemantics.com/product/anzograph/.
[49] Mihai Capotă, Tim Hegeman, Alexandru Iosup, et al. 2015. Graphalytics: A big data benchmark for graph-processing platforms. In *ACM GRADES*. https://doi.org/10.1145/2764947.2764954
[50] Arnaud Castelltort et al. 2013. Representing history in graph-oriented NoSQL databases: A versioning system. In *IEEE ICDIM*. 228–234.
[51] Cayley. CayleyGraph. Retrieved from https://cayley.io/ and https://github.com/cayleygraph/cayley.
[52] Ernie Chan et al. 2007. Collective communication: Theory, practice, and experience. *Concurr. Comput.: Pract. Exper.* 19, 13 (2007), 1749–1783.
[53] Marek Ciglan, Alex Averbuch, et al. 2012. Benchmarking traversal operations over graph databases. In *IEEE ICDE Workshops*. 186–189.
[54] James Clark and Steve DeRose. 1999. *XML Path Language (XPath) Version 1.0*. World Wide Web Consortium.
[55] Edgar F. Codd. 1989. Relational database: A practical foundation for productivity. In *Readings in Artificial Intelligence and Databases*. 60–68.
[56] Douglas Comer. 1979. The ubiquitous B-tree. *ACM Comput. Surv.* 11, 2 (1979), 17. https://doi.org/10.1145/356770.356776
[57] Richard Cyganiak, David Wood, and Markus Lanthaler. 2014. RDF 1.1 Concepts and abstract syntax.
[58] DataStax, Inc. DSE graph (DataStax). Retrieved from https://www.datastax.com/.
[59] Chris J. Date and Hugh Darwen. 1987. *A Guide to the SQL Standard*. Vol. 3.
[60] Ali Davoudian, Liu Chen, and Mengchi Liu. 2018. A survey on NoSQL stores. *ACM Comput. Surv.* 51, 2, Article 40 (2018), 43 pages. https://doi.org/10.1145/3158661
[61] Dgraph Labs Inc. BadgerDB. Retrieved from https://dbdb.io/db/badgerdb.
[62] Dgraph Labs Inc. Dgraph. Retrieved from https://dgraph.io/ and https://dgraph.io/docs/.
[63] Salvatore Di Girolamo et al. 2019. Network-accelerated non-contiguous memory transfers. In *ACM/IEEE SC*. https://doi.org/10.1145/3295500.3356189
[64] E. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1, 1 (1959), 269–271. https://doi.org/10.1007/BF01386390
[65] Niels Doekemeijer and Ana Lucia Varbanescu. 2014. *A Survey of Parallel Graph Processing Frameworks*. Technical Report. Delft University of Technology.
[66] David Dominguez-Sal et al. 2010. Survey of graph database performance on the HPC scalable graph analysis benchmark. In *WAIM*. 37–48.
[67] Ayush Dubey et al. 2016. Weaver: A high-performance, transactional graph database based on refinable timestamps. *Proc. VLDB Endow.* 9, 11 (2016), 852–863. https://doi.org/10.14778/2983200.2983202
[68] Paul DuBois. 1999. *MySQL*. New Riders Publishing.
[69] William Eberle, Jeffrey Graves, et al. 2010. Insider threat detection using a graph-based approach. *J. Appl. Secur. Res.* 6, 1 (2010), 32–81.
[70] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *IEEE HPEC*. 1–5.
[71] Hamid El Maazouz, Guido Wachsmuth, Martin Sevenich, et al. 2019. A DSL-based framework for performance assessment. In *EMENA-ISTL*. 260–270.
[72] Elastic. Elasticsearch. Retrieved from https://www.elastic.co/elasticsearch/.
[73] Ramez Elmasri et al. 2011. Advantages of distributed databases. In *Fundamentals of Database Systems, 6th Edition*. Chapter 25.1.5, 882.
[74] Ramez Elmasri and Shamkant B. Navathe. 2011. Data fragmentation. In *Fundamentals of Database Systems, 6th Edition*. Chapter 25.4.1, 894–897.
[75] Orri Erling, Alex Averbuch, Josep Larriba-Pey, et al. 2015. The LDBC social network benchmark: Interactive workload. In *ACM SIGMOD*. 619–630.
[76] FactNexus. GraphBase. Retrieved from https://graphbase.ai/.
[77] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. 2015. The case against specialized graph analytics engines. In *CIDR*.
[78] Fauna. FaunaDB. Retrieved from https://fauna.com/.
[79] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2023. KÙZU graph database management system. In *CIDR*.
[80] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, et al. 2023. A researcher's digest of GQL. In *ICDT*. 1:1–1:22. https://doi.org/10.4230/LIPIcs.ICDT.2023.1
[81] Nadime Francis, Alastair Green, Paolo Guagliardo, et al. 2018. Cypher: An evolving query language for property graphs. In *ACM SIGMOD*. 1433–1445.
[82] Franz Inc. AllegroGraph. Retrieved from https://allegrograph.com/products/allegrograph/.

[83] Santhosh Kumar Gajendran. 2012. *A Survey on NoSQL Databases.* Technical Report. University of Illinois.

[84] Hector Garcia-Molina, Jeffrey D. Ullman, et al. 2002. Data replication. In *Database Systems: The Complete Book, 1st Edition.* Chapter 19.4.3, 1021.

[85] Ewout Gelling, George Fletcher, and Michael Schmidt. 2023. Bridging graph data models: RDF, RDF-star, and property graphs as directed acyclic graphs. Retrieved from https://arxiv.org/abs/2304.13097.

[86] Evangelos Georganas et al. 2012. Communication avoiding and overlapping for numerical linear algebra. In *ACM/IEEE SC.*

[87] Lars George. 2011. *HBase: The Definitive Guide.*

[88] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. 2014. Enabling highly-scalable remote memory access programming with MPI-3 one sided. *Sci. Program.* 22, 2 (2014), 75–91.

[89] Lukas Gianinazzi et al. 2018. Communication-avoiding parallel minimum cuts and connected components. *ACM SIGPLAN Not.* 53, 1 (2018), 219–232. https://doi.org/10.1145/3200691.3178504

[90] Niels Gleinig, Maciej Besta, and Torsten Hoefler. 2022. I/O-optimal cache-oblivious sparse matrix-sparse matrix multiplication. In *IEEE IPDPS.* 36–46.

[91] Google. Graphd. Retrieved from https://github.com/google/graphd.

[92] Graph Story Inc. Graph story. Retrieved from https://github.com/graphstory.

[93] Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, et al. 2019. Updating graph databases with Cypher. *Proc. VLDB Endow.* 12, 12 (2019), 2242–2254. https://doi.org/10.14778/3352063.3352139

[94] Alastair Green, Martin Junghanns, Max Kießling, et al. 2018. openCypher: New directions in property graph querying. In *EDBT.* 520–523.

[95] L. Guzenda. 2000. Objectivity/DB – A high performance object database architecture. In *HIPOD.*

[96] Jing Han, E. Haihong, Guan Le, and Jian Du. 2011. Survey on NoSQL database. In *IEEE ICPCA.* 363–366.

[97] Minyang Han, Khuzaima Daudjee, Khaled Ammar, et al. 2014. An experimental comparison of Pregel-like graph processing systems. *Proc. VLDB Endow.* 7, 12 (2014), 1047–1058. https://doi.org/10.14778/2732977.2732980

[98] Andreas Harth et al. 2007. YARS2: A federated repository for querying graph structured data from the web. In *ISWC/ASWC.*

[99] Olaf Hartig. 2014. Reconciliation of RDF* and property graphs. arXiv:1409.3288. Retrieved from https://arxiv.org/abs/1409.3288.

[100] Olaf Hartig. 2017. RDF* and SPARQL*: An alternative approach to annotate statements in RDF. In *ISWC (Poster).*

[101] Olaf Hartig. 2019. Foundations to query labeled property graphs using SPARQL*. In *SEM4TRA-AMAR.*

[102] Olaf Hartig and Jorge Pérez. 2018. Semantics and complexity of GraphQL. In *WWW.* 1155–1164. https://doi.org/10.1145/3178876.3186014

[103] Jonathan Hayes. 2004. *A Graph Model for RDF.* Diploma Thesis. Technische Universität Darmstadt, Universidad de Chile.

[104] Joseph M. Hellerstein and Michael Stonebraker. 2005. *Readings in Database Systems.*

[105] Jeffrey A. Hoffer, Venkataraman Ramesh, and Heikki Topi. 2011. *Modern Database Management.*

[106] Florian Holzschuher and René Peinl. 2013. Performance of graph query languages: Comparison of Cypher, Gremlin and native access in Neo4j. In *EDBT.* 195–204. https://doi.org/10.1145/2457317.2457351

[107] Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, et al. 2015. PGX.D: A fast distributed graph processing engine. In *ACM/IEEE SC.* https://doi.org/10.1145/2807591.2807620

[108] InfiniBand Trade Association. 2015. InfiniBand: Architecture specification 1.3.

[109] InfoGrid. The InfoGrid graph database. Retrieved from http://infogrid.org.

[110] Borislav Iordanov. 2010. HyperGraphDB: A generalized graph database. In *WAIM.* 25–36.

[111] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, et al. 2016. LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. *Proc. VLDB Endow.* 9, 13 (2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[112] Jesús Barrasa. 2017. RDF triple stores vs. labeled property graphs: What's the difference? Retrieved from https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/.

[113] Bin Jiang. 2011. A short note on data-intensive geospatial computing. In *Information Fusion and Geographic Information Systems.* 13–17.

[114] Alekh Jindal, Samuel Madden, Malú Castellanos, and Meichun Hsu. 2015. Graph analytics using Vertica relational database. In *IEEE Big Data.* 1191–1200. https://doi.org/10.1109/BigData.2015.7363873

[115] Salim Jouili and Valentin Vansteenberghe. 2013. An empirical comparison of graph databases. In *IEEE SocialCom.* 708–715.

[116] Martin Junghanns, André Petermann, Martin Neumann, and Erhard Rahm. 2017. Management and analysis of big graph data: Current systems and open challenges. In *Handbook of Big Data Technologies.* 457–505.

[117] Vasiliki Kalavri, Vladimir Vlassov, and Seif Haridi. 2017. High-level programming abstractions for distributed graph processing. *IEEE Trans. Knowl. Data Eng.* 30, 2 (2017), 305–324.
[118] R. Kumar Kaliyar. 2015. Graph databases: A survey. In *IEEE ICCCA*. 785–790.
[119] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. 2012. GBase: An efficient analysis platform for large graphs. *VLDB J.* 21, 5 (2012), 637–650. https://doi.org/10.1007/s00778-012-0283-9
[120] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, et al. 2017. Graphflow: An active graph database. In *ACM SIGMOD*. 1695–1698. https://doi.org/10.1145/3035918.3056445
[121] Michael Kay. 2001. *XSLT Programmer's Reference.*
[122] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, et al. 2016. Mathematical foundations of the GraphBLAS. In *IEEE HPEC*. 1–9.
[123] Vaibhav Khadilkar et al. 2012. Jena-HBase: A distributed, scalable and efficient RDF triple store. In *ISWC (Poster)*. 85–88.
[124] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. 2008. Technology-driven, highly-scalable dragonfly topology. In *IEEE ISCA*. 77–88. https://doi.org/10.1109/ISCA.2008.19
[125] Vojtech Kolomicenko. 2013. *Analysis and Experimental Comparison of Graph Databases.* Master's Thesis. Charles University in Prague.
[126] Vijay Kumar and Anjan Babu. 2015. Domain suitable graph database selection: A preliminary report. In *ICAESAM*. 26–29.
[127] Grzegorz Kwasniewski et al. 2019. Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication. In *ACM/IEEE SC*. https://doi.org/10.1145/3295500.3356181
[128] Grzegorz Kwasniewski et al. 2021. On the parallel I/O optimality of linear algebra kernels: Near-optimal LU factorization. In *ACM PPoPP*. 463–464. https://doi.org/10.1145/3437801.3441590
[129] Grzegorz Kwasniewski, Tal Ben-Nun, Lukas Gianinazzi, Alexandru Calotoiu, Timo Schneider, Alexandros Nikolaos Ziogas, Maciej Besta, and Torsten Hoefler. 2021. Pebbles, graphs, and a pinch of combinatorics: Towards tight I/O lower bounds for statically analyzable programs. In *ACM SPAA*. 328–339. https://doi.org/10.1145/3409964.3461796
[130] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922
[131] LambdaZen LLC. Bitsy. Retrieved from https://github.com/lambdazen/bitsy and https://bitbucket.org/lambdazen/bitsy/wiki/Home.
[132] Ora Lassila et al. 2023. The OneGraph vision: Challenges of breaking the graph model lock-in. *Semant. Web* 14, 1 (2023), 125–134.
[133] Matteo Lissandrini et al. 2017. *An Evaluation Methodology and Experimental Comparison of Graph Databases.* Technical Report. University of Trento.
[134] Matteo Lissandrini, Martin Brugnara, et al. 2018. Beyond macrobenchmarks: Microbenchmark-based graph database evaluation. *Proc. VLDB Endow.* 12, 4 (2018), 390–403. https://doi.org/10.14778/3297753.3297759
[135] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan W. Berry. 2007. Challenges in parallel graph processing. *Par. Proc. Let.* 17, 1 (2007), 5–20.
[136] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, et al. 2010. Pregel: A system for large-scale graph processing. In *ACM SIGMOD*. 135–146. https://doi.org/10.1145/1807167.1807184
[137] MariaDB. OQGRAPH. Retrieved from https://mariadb.com/kb/en/oqgraph-storage-engine/.
[138] MarkLogic Corporation. MarkLogic. Retrieved from https://www.marklogic.com.
[139] Norbert Martínez-Bazan, M. Ángel Águila Lorente, et al. 2012. Efficient graph management based on bitmap indices. In *ACM IDEAS*. 110–119. https://doi.org/10.1145/2351476.2351489
[140] József Marton, Gábor Szárnyas, and Dániel Varró. 2017. Formalising openCypher graph queries in relational algebra. In *ADBIS*. 182–196.
[141] Kristyn J. Maschhoff, Robert Vesse, et al. 2017. Quantifying performance of CGE: A unified scalable pattern mining and search system. In *CUG*.
[142] Robert Campbell McColl, David Ediger, Jason Poovey, et al. 2014. A performance evaluation of open source graph databases. In *ACM PPAA*. 11–18. https://doi.org/10.1145/2567634.2567638
[143] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.* 48, 2, Article 25 (2015), 39 pages. https://doi.org/10.1145/2818185
[144] Memgraph Ltd. Memgraph. Retrieved from https://memgraph.com/.
[145] Scott M. Meyer, Jutta Degener, et al. 2010. Optimizing schema-last tuple-store queries in Graphd. In *ACM SIGMOD*. 1047–1056. https://doi.org/10.1145/1807167.1807283
[146] Amine Mhedbhi et al. 2021. LSQB: A large-scale subgraph query benchmark. In *ACM GRADES-NDA*. https://doi.org/10.1145/3461837.3464516

[147] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.* 12, 11 (2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

[148] Microsoft. Azure Cosmos DB. Retrieved from https://azure.microsoft.com/en-us/products/cosmos-db/.

[149] Microsoft. Microsoft SQL server 2017. Retrieved from https://www.microsoft.com/en-us/sql-server/sql-server-2017.

[150] Bruce Momjian. 2001. *PostgreSQL: Introduction and Concepts*. Vol. 192, (2001).

[151] Thomas Mueller. 2005. H2 database engine. (2005). Retrieved from http://www.h2database.com.

[152] Networked Planet Limited. BrightstarDB. Retrieved from http://brightstardb.com/.

[153] Thomas Neumann and Gerhard Weikum. 2010. X-RDF-3X: Fast querying, high update rates, and consistency for RDF databases. *Proc. VLDB Endow.* 3, 1–2 (2010), 256–263. https://doi.org/10.14778/1920841.1920877

[154] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *J. ACM* 65, 3, Article 16 (2018), 40 pages. https://doi.org/10.1145/3180143

[155] Objectivity Inc. ThingSpan. Retrieved from https://www.objectivity.com/products/thingspan/.

[156] Mike Olson, Keith Bostic, and Margo Seltzer. 1999. Berkeley DB. In *USENIX ATC*.

[157] Ontotext. GraphDB. Retrieved from https://www.ontotext.com/products/graphdb/.

[158] OpenLink. Virtuoso. Retrieved from https://virtuoso.openlinksw.com/.

[159] Oracle. Oracle Spatial and Graph. Retrieved from https://www.oracle.com/database/technologies/spatialandgraph.html.

[160] Kay Ousterhout et al. 2015. Making sense of performance in data analytics frameworks. In *NSDI*. 293–307.

[161] M. Tamer Özsu. 2016. A survey of RDF data management systems. *Front. Comput. Sci.* 10, 3 (2016), 418–432.

[162] Anil Pacaci et al. 2017. Do we need specialized graph databases? Benchmarking real-time social networking applications. In *ACM GRADES*. Article 12, 7 pages. https://doi.org/10.1145/3078447.3078459

[163] Lawrence Page, Sergey Brin, Rajeev Motwani, et al. 1999. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report. Stanford InfoLab.

[164] N. S. Patil, P Kiran, et al. 2018. A survey on graph database management techniques for huge unstructured data. *Int. J. Electr. Comput. Eng.* 8, 2 (2018), 1140–1149.

[165] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3, Article 16 (2009), 45 pages. https://doi.org/10.1145/1567274.1567278

[166] Tomasz Pluskiewicz. 2022. RDF String. Retrieved from https://github.com/tpluscode/rdf-string.

[167] Jaroslav Pokornỳ. 2015. Graph databases: Their power and limitations. In *CISIM*. 58–69.

[168] Profium. Profium sense. Retrieved from https://www.profium.com/en/products/graph-database/.

[169] Roshan Punnoose, Adina Crainiceanu, and David Rapp. 2012. Rya: A scalable RDF triple store for the clouds. In *Cloud-I*. Article 4, 8 pages. https://doi.org/10.1145/2347673.2347677

[170] Ashish Rana. 2019. Detailed Introduction: Redis Modules, from Graphs to Machine Learning (Part 1). Retrieved from https://medium.com/@ashishrana160796/15ce9ff1949f.

[171] Redis Labs. Redis. Retrieved from https://redis.io/.

[172] Redis Labs. RedisGraph. Retrieved from https://redis.io/docs/stack/graph/.

[173] Christopher D. Rickett, Utz-Uwe Haus, James Maltby, et al. 2018. Loading and querying a trillion RDF triples with Cray Graph Engine on the Cray XC. In *CUG*.

[174] Robert Yokota. HGraphDB. Retrieved from https://github.com/rayokota/hgraphdb.

[175] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. Graph database internals. In *Graph Databases, 2nd Edition*. 149–170.

[176] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language. In *DBPL*. 1–10. https://doi.org/10.1145/2815072.2815073

[177] Shahin Roozkhosh, Denis Hoornaert, Ju Hyoung Mun, Tarikul Islam Papon, Ahmed Sanaullah, Ulrich Drepper, Renato Mancuso, and Manos Athanassoulis. 2021. Relational memory: Native in-memory accesses on rows and columns. arXiv:2109.14349. Retrieved from https://arxiv.org/abs/2109.14349.

[178] Nicholas P. Roth, Vasileios Trigonakis, Sungpack Hong, et al. 2017. PGX.D/Async: A scalable distributed graph pattern matching engine. In *ACM GRADES*. https://doi.org/10.1145/3078447.3078454

[179] Michael Rudolf et al. 2013. The graph story of the SAP HANA database. In *Datenbanksysteme für Business, Technologie und Web*. 403–420.

[180] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, et al. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.* 11, 4 (2017), 420–431.

[181] SAP. SAP HANA. Retrieved from https://www.sap.com/products/technology-platform/hana.html.

[182] Y. Sasaki, G. Fletcher, and O. Makoto. 2022. Language-aware indexing for conjunctive path queries. In *IEEE ICDE*. 661–673.

[183] Satu Elisa Schaeffer. 2007. Survey: Graph clustering. *Comput. Sci. Rev.* 1, 1 (2007), 27–64. https://doi.org/10.1016/j.cosrev.2007.05.001

[184] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A distributed graph engine on a memory cloud. In *ACM SIGMOD*. 505–516.

[185] Sanjay Sharma. 2014. *Cassandra Design Patterns*.

[186] Xuanhua Shi, Zhigao Zheng, Yongluan Zhou, Hai Jin, Ligang He, Bo Liu, and Qiang-Sheng Hua. 2018. Graph processing on GPUs: A survey. *ACM Comput. Surv.* 50, 6, Article 81 (2018), 35 pages. https://doi.org/10.1145/3128571

[187] solid IT gmbh. System properties comparison: Neo4j vs. Redis. Retrieved from https://db-engines.com/en/system/Neo4j%3BRedis.

[188] Edgar Solomonik, Erin Carson, Nicholas Knight, and James Demmel. 2014. Tradeoffs between synchronization, communication, and computation in parallel linear algebra computations. In *ACM SPAA*. 307–318. https://doi.org/10.1145/2612669.2612671

[189] Edgar Solomonik and Torsten Hoefler. 2015. Sparse tensor algebra as a parallel programming model. arXiv:1512.00066. Retrieved from https://arxiv.org/abs/1512.00066.

[190] Stardog Union. 2018. Stardog. Retrieved from https://www.stardog.com/.

[191] Benjamin A. Steer, Alhamza Alnaimi, Marco A. B. F. G. Lotz, et al. 2017. Cytosm: Declarative property graph queries without data migration. In *ACM GRADES*. https://doi.org/10.1145/3078447.3078451

[192] Wen Sun, Achille Fokoue, Kavitha Srinivas, et al. 2015. SQLGraph: An efficient relational-based property graph store. In *ACM SIGMOD*. 1887–1901. https://doi.org/10.1145/2723372.2723732

[193] Gábor Szárnyas et al. 2018. An early look at the LDBC social network benchmark's business intelligence workload. In *ACM GRADES-NDA*. https://doi.org/10.1145/3210259.3210268

[194] Gábor Szárnyas et al. 2022. The LDBC social network benchmark: Business intelligence workload. *Proc. VLDB Endow.* 16, 4 (2022), 877–890. https://doi.org/10.14778/3574245.3574270

[195] Ruben Taelman. 2022. RDF String Turtle. Retrieved from https://github.com/rubensworks/rdf-string-ttl.js.

[196] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter Boncz. 2023. DuckPGQ: Efficient property graph queries in an analytical RDBMS. In *CIDR*.

[197] The Apache Software Foundation. 2021. Apache Jena TBD. Retrieved from https://jena.apache.org/documentation/tdb/index.html.

[198] The Linux Foundation. 2018. JanusGraph. Retrieved from http://janusgraph.org/.

[199] Yuanyuan Tian et al. 2020. IBM Db2 Graph: Supporting synergistic and retrofittable graph queries inside IBM Db2. In *ACM SIGMOD*. 345–359. https://doi.org/10.1145/3318464.3386138

[200] TigerGraph. 2018. TigerGraph. Retrieved from https://www.tigergraph.com/.

[201] Alok Tripathy, Katherine Yelick, and Aydın Buluç. 2020. Reducing communication in graph neural network training. In *ACM/IEEE SC*.

[202] Twitter. 2010. FlockDB. Retrieved from https://github.com/twitter-archive/flockdb.

[203] Aparna Vaikuntam and Vinodh Kumar Perumal. 2014. Evaluation of contemporary graph databases. In *ACM COMPUTE*. https://doi.org/10.1145/2675744.2675752

[204] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A property graph query language. In *ACM GRADES*. https://doi.org/10.1145/2960414.2960421

[205] VelocityDB Inc. 2019. VelocityDB. Retrieved from https://velocitydb.com/.

[206] VelocityDB Inc. 2019. VelocityGraph. Retrieved from https://velocitydb.com/QuickStartVelocityGraph.

[207] WhiteDB Team. 2013. WhiteDB. Retrieved from http://whitedb.org/: https://github.com/priitj/whitedb.

[208] Konstantinos Xirogiannopoulos, Virinchi Srinivas, and Amol Deshpande. 2017. GraphGen: Adaptive graph processing using relational databases. In *ACM GRADES*. https://doi.org/10.1145/3078447.3078456

[209] Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, and James Cheng. 2016. Big graph analytics systems. In *ACM SIGMOD*. 2241–2243. https://doi.org/10.1145/2882903.2912566

[210] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: A fast and compact system for large scale RDF data. *Proc. VLDB Endow.* 6, 7 (2013), 517–528. https://doi.org/10.14778/2536349.2536352

[211] Kangfei Zhao and Jeffrey Xu Yu. 2017. All-in-one: Graph processing in RDBMSs revisited. In *ACM SIGMOD*. 1165–1180.

[212] Xiaowei Zhu et al. 2020. LiveGraph: A transactional graph storage system with purely sequential adjacency list scans. *Proc. VLDB Endow.* 13, 7 (2020), 1020–1034. https://doi.org/10.14778/3384345.3384351

[213] Lei Zou, M. Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. 2014. gStore: A graph-based SPARQL query engine. *VLDB J.* 23, 4 (2014), 565–590. https://doi.org/10.1007/s00778-013-0337-7

Received 3 July 2022; revised 12 March 2023; accepted 31 May 2023