Where the Path Leads: State of the Art and Challenges of Graph Database Systems

Tim Träris
Faculty of Computer Science
Furtwangen University
Furtwangen, Germany
tim.jannes.traeris@hs-furtwangen.de

Maxim Balsacq
Faculty of Computer Science
Furtwangen University
Furtwangen, Germany
maxim.balsacq@hs-furtwangen.de

***Abstract*—Compared to relational databases, graph database systems provide a novel way of processing and analyzing highly interconnected data. Due to their unique properties, graph databases embody an interesting area of research in academic circles. For this reason, this work is fundamentally concerned with examining the state of the industry and current challenges. In this regard, we revisit the basic concepts and highlight the tremendous heterogeneity of available systems using the example of differing path semantics. Based on this insight, we explore algorithmic advancements for graph query processing regarding path finding and worst-case optimal joins. Subsequently, we discuss issues regarding performance and support for graph analytics. Finally, we provide an overview of GQL, a joint standardization effort towards unification of property graph databases.**

***Index Terms*—graph database, path semantics, worst-case optimal join, graph analytics, GQL**

I. Introduction

Graph databases store and represent data as graphs, simplifying the exploration and analysis of highly interconnected information. In the following, we introduce the basic concepts and graph models.

A. The Limits of Relational Relationships

Soon after Codd had proposed his original version of the relational data model [1], relational database systems became the go-to solution for a broad number of use cases and applications with the need to represent, store, retrieve and interact with structured data. In this model, databases use relations (tables) and tuples (rows) to manage data with each row representing a data set. Attribute values are stored as columns, forcing the same structure for all data sets in the table. In this regard, the relation schema strictly defines the number and type of attributes. Data is commonly inserted and retrieved using the standardized SQL query language. Relationships between different tables are established by foreign keys referencing the primary key of another table. Naturally, when querying for data sets and their relationships in multiple tables, these implicit interconnections must be mapped at query runtime. In relational algebra, this process is called a join operation.

When dealing with highly interconnected information, queries often require multiple sub-query joins. With this in mind, the implicit relationship mapping of relational databases becomes increasingly costly to compute with large data sets.

B. Introducing Graphs

Graph databases can be categorized as NoSQL databases and differ fundamentally in structure, design and operation compared to relational databases. In contrast to relational database systems, graph databases prioritize the relationships between the data, effectively treating them as first-class citizens. As a result, the mapping of hierarchical and networked structures is simplified.

Graphs consist of uniquely identifiable data entities called vertices and interconnections between them referred to as edges. These two elements are commonly visualized as points and lines as shown in figure 1. Each vertex may have an arbitrary number of incoming, outgoing or unidirected edges. Two vertices can also be connected by several edges, sometimes referred to as multigraph in graph theory. Further, graphs may assign weights to edges indicating contextual properties like cost, length or capacity.

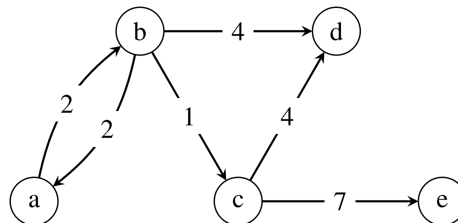

Fig. 1. A weighted graph with five vertices and six directed edges.

C. Graph Models

Graph models define the general behavior of graphs by specifying how data can be explored and graphically displayed. Graph databases mainly implement two different graph models: the labeled-property graph and the resource description framework.

1) Labeled-property Graph (LPG): In a LPG, vertices are commonly referred to as nodes and edges between them are called relationships. Both of them are referenced using a unique ID within the graph. Besides this identifier, properties in the form of key-value pairs can be attributed to both node and edge elements [2]. This results in graph elements labeled with arbitrary additional properties, hence the name.

Since all elements can be uniquely identified using their ID, duplicate nodes as well as duplicate edges are possible. Exemplary, multiple nodes could have the same labels and multiple relationships with the same properties could exist between two nodes.

2) Resource Description Framework (RDF): The RDF is a World Wide Web Consortium (W3C) standard [3] originally designed for descriptive modeling of metadata. In this regard, it is a fundamental component for the Semantic Web. The RDF model defines all data entities as instances of subject-predicate-object triples [4]. In these triple statements, a resource (subject) is described in more detail (predicate) with another resource or literal value (object). In contrast to LPG, entities do not contain any internal data structure and additional information is solely attributed by adding subject-predicate-object statements with object literals. Resources are globally identified by a Uniform Resource Identifier (URI). Naturally, all subjects and objects result in graph vertices connected by directed edges. Since all resources are identified by their URI, multiple triples can form an interconnected graph. Duplicate graph elements with the same URI are not possible. Exemplary, table I lists the subject-predicate-object triples to form the graph displayed in figure 1. Since the graph does not contain any additional key-value pairs, the translation is straightforward and does not require further triples to map data.

TABLE I
SUBJECT-PREDICATE-OBJECT TRIPLES OF FIGURE 1

Subject	Predicate	Object
a	2	b
b	2	a
b	4	d
b	1	c
c	4	d
c	7	e

Further, triples can be extended by a graph reference, therefore resulting in quads. This graph identifier allows storing multiple graphs in one database as all quads can be conjugated to a specific graph.

II. Related Work

In ”The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey” (2020) [5], Sahu et al. conduct an extensive study to investigate the use of graph databases and related technologies in practice. In this regard, the authors interview various different stakeholders in the industry and work out use cases, used software stacks, and encountered challenges. Their results indicate a tremendous variety in graph applications and workloads. Additionally, the authors observe that many use cases involve very large graphs with more than a billion edges. Consequently, scalability becomes a pressing issue as available technologies struggle with emerging data volumes. Sahu et al. conclude among other things that, given the broad variety of use cases, available software, and query languages, collaborative standardization efforts are needed to unify existing technologies. We highlight this issue further in section III and subsequently investigate standardization efforts around GQL in section VII.

In ”RDF and Property Graphs Interoperability: Status and Issues” (2019) [6], Angles et al. explore the interoperability of RDF and LPG graph models. The authors argue that syntactic, semantic, and query interoperability is vital in order to create common understanding and improve accessibility, exchangeability, shareability, and reusability of data and deployed software stacks. Subsequently, Angles et al. present a review of interoperability efforts and discuss current challenges. In this regard, they highlight the lack of a standard data format for LPG resulting in challenges for syntactic interoperability. Similarly, the lack of a standardized query language results in query incompatibility. Finally, semantic interoperability often requires manual translation and human intervention as RDF semantics cannot easily be transferred to LPG. Consequently, the authors conclude that a standardized query language is needed to facilitate graph workloads and transformations between RDF and LPG.

III. State of the Industry

Various applications for graph databases have been found in fields of work including but not limited to finance, healthcare and transportation [5]. As such it is no surprise that many different graph databases have been developed, each with their own strengths and weaknesses. In this section we will present a short overview of two general issues affecting the industry.

A. Heterogeneity of Graph Databases

While many different graph databases systems have been developed, no single unified standard for their capabilities or query language exists. Graph database developers often resort to developing their own querying language along with the database. The closest thing to agreed-upon standards seem to be SPARQL and openCypher, developed by the W3C and the openCypher Implementers Group, respectively. Unsurprisingly, available systems implement similar behavior in some aspects, while fundamentally differing in others. As we will discuss in later sections, this heterogeneity of graph databases is not limited to the query language. The differences also extend to basic concepts such as the graph model, path semantics as well as practical issues e.g. the ability to use path finding algorithms.

B. Lack of Scalability

By distributing data across multiple machines (known as partitioning or sharding), it is possible to scale up the size of the data to process, as queries can be performed on all machines in parallel. However, implementing partitioning for graph databases is easier said than done, as the relationships between nodes make it difficult to determine on which machine data should be stored. In case the wrong location is chosen, an increased number of round trips between machines may

be necessary to process queries. This problem is known as "minimum point-cut problem" and difficult to solve due to the fact that most graph partitioning problems are known to be NP-complete, meaning that the solution cannot be verified in polynomial time [7]. Additionally, optimal data distribution is subject to change as new data is inserted into the database. Exemplary, new vertices introducing numerous relationships could require a redistribution of storage units. Further, many current graph database systems have not been designed with today's amount of data in mind. Consequently, support for horizontal scalability is a major concern.

IV. Path Semantics

The discussed heterogeneity of graph databases extends to fundamental concepts such as path semantics. Path semantics specify which paths are valid while querying a pattern over multiple nodes. The applied semantics vary between query languages. This leads to portability issues, as a syntactically equivalent query may yield different results when the query is translated to a different query language. Commonly used path semantics include no-repeated-node, no-repeated-edge and arbitrary path semantics. In the following, we will illustrate the different semantics using the graph in figure 2. When applying the query below on the example graph, each path semantic yields a different set of result paths. The query finds all paths starting at a node `a`, traversing at least one `X` relation and reaching a node `c` via a `Y` relation.

```
(a)->[:X*1..]->()->[:Y]->(c)
```

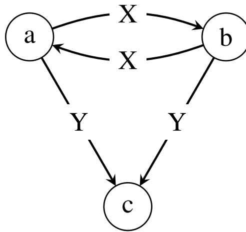

Fig. 2. Path semantics example graph

A. Arbitrary Path Semantics

Arbitrary path semantics allow any amount of repeated edges and vertices. Thus, the result set of a query using arbitrary path semantics is unbounded. This introduces difficulties in handling query results and regular path traversals, as the query may not terminate without manually specifying upper bounds. Notably, SPARQL uses arbitrary path semantics [8]. Using the query on the example graph in figure 2 with arbitrary semantics, we would obtain an infinite number of results, as a cycle exists between vertices `a` and `b`.

B. No-Repeated-Edge Path Semantics

No-repeated-edge path semantics prohibit that a path passes through the same edge twice. However, two nodes in a graph may be connected by multiple edges, so it is possible that a node pair is visited twice through different edges. No-repeated-edge semantics are used by the Cypher query language [8]. Using no-repeated-edge path semantics and the query on the example graph in figure 2, we would obtain two results: `a->b->c` and `a->b->a->c`. Note that node `a` appears twice in the second case, however no edges are passed through twice.

C. No-Repeated-Node Path Semantics

No-repeated node semantics are useful when no-repeated-edge path are considered too permissive. By prohibiting that the same node occurs more than once in a path it is possible to completely remove any looping paths from the set of results. Using no-repeated node semantics and the query on the example graph in figure 2 we would obtain the result `a->b->c`.

V. Challenges in Graph Query Processing

Given that many use cases involve large graphs with billions of nodes [5], the algorithms used to process graphs at scale need to be chosen with care. Choosing the wrong algorithms can severely impact query performance. In this section, we present two algorithmic challenges graph databases face when planning and executing queries.

A. Worst-Case Optimal Joins

As previously shown in figure 1, a graph can be represented by a relational table. Naturally, graph pattern queries can also be translated into a series of joins on such a table. Since matching multiple nodes at once is a common task in graph databases, so are these join-like queries. To improve the performance of join queries, fast algorithms are needed. So-called Worst-Case Optimal Join (WCOJ) algorithms provide the best possible runtime in cases where the number of results of a join query is equal to the maximum possible value.

A well-known benchmark for WCOJ consists of the so-called triangle query. Such a query is illustrated in Figure 3. The nodes `x`, `y` and `z` represent wildcards which can be matched onto any distinct set of nodes. However, the directed connections (named `R`, `S` and `T`) between the matched node must exist, e.g. `x -> y`, `y -> z` and `x -> z`. When processing this query, a naive approach might first find the set of matching node pairs for each of the three edges `R`, `S` and `T`, then use two joins to combine the results into the final pattern. While processing the first of these two joins, a temporary projection with $\Omega(N^2)$ entries would be required, where `N` is the number of nodes in the graph database. However, it can be proven that there are at most $\sqrt{|R| \times |S| \times |T|} = O(N^{3/2})$ results using the triangle query, where `R`, `S` and `T` represent the upper bound of possible matches for each relation. Using a Worst-Case Optimal Join algorithm, it is possible to eliminate the temporary projection and directly generate the query results.

As a result, WCOJs contribute to improving the performance of graph database systems by optimizing join queries. However, there is no single best WCOJ algorithm: Various algorithms with differing performance have been found in

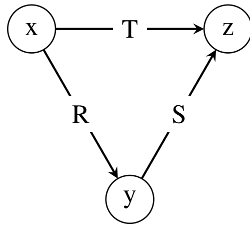

Fig. 3. A Triangle Query

different branches of mathematics, such as information theory and geometric inequalities [9]. Further, while the algorithms may perform optimally in the worst case, this may not be true in other cases. An algorithm which guarantees optimal performance in the worst case (a WCOJ algorithm) may be outperformed by a different, non-WCOJ algorithm which prioritizes the performance of common cases.

Atserias et al. first proved the size maximum bound for joins in "Size Bounds and Query Plans for Relational Joins" [10] based on Shearer's lemma. This maximum bound of these joins is since known as the AGM bound. Ngo et al. built on this and developed one of the first WCOJ algorithms [9]. Nguyen et al. argued in "Join Processing for Graph Patterns: An Old Dog with New Tricks" [11], that applying the algorithms for WCOJs to relational databases let them compete with, and possible outperform, graph databases for pattern matching. They demonstrated this by using WCOJs to improve their commercial database called "LogicBlox". However, Aberger et al. showed that the performance of graph databases can be improved further by using WCOJs, changing data layout and using Single Instruction, Multiple Data (SIMD). Their resulting graph database "EmptyHeaded" was able to outperform "LogicBlox" and comparable databases by multiple magnitudes [12]. Efforts have been made to adapt the approach of using WCOJs to graph database models such as SPARQL [13]. Further work is needed to use Worst-Case Optimal Joins in graph databases at their full potential.

B. Path Finding Algorithms

While path semantics determine which paths are valid, path finding algorithms determine how these paths can be found in the first place. Naturally, different path algorithms exist, with different goals and characteristics. Simple algorithms such as Breadth-First-Search or Depth-First-Search are used to inspect adjacent nodes. Breadth-First-Search first visits all adjacent nodes and only then continues traversing paths further away, while Depth-First-Search traverses through the graph as far as possible before trying to find the next adjacent nodes. Other algorithms such as Dijkstra's algorithm [14] are used to find the shortest path between two nodes with an unknown distance. Positive weights can be assigned to edges, making it possible to simulate distance between nodes. An example use case for Dijkstra's algorithm would be to compute the shortest real-world route to travel between two locations. Further common path finding algorithms include A* and Yen's k-shortest paths. A* can be seen as an extension of Dijkstra's algorithm, as it improves the performance by using heuristics. In contrast, Yen's k-shortest paths allows finding multiple top-k shortest paths between the same vertices.

However, applying path finding algorithms to graph databases is difficult in practice as constraints on node and edges must be considered. Additionally, graph queries are likely required to find not one, but many paths between different nodes. In combination with the different path semantics, this leads to a high computational complexity. As an example, the SPARQL language (see section III-A) uses arbitrary path semantics and simple walks. A simple walk is a path which may end at the start node, but does not visit other nodes twice [15]. The chosen combination leads to the fact that querying to check if some path (which matches a regular expression) exists is NP-complete [15].

Although there is a broad variety of theoretical path algorithms available, current graph database systems often implement only selected ones. For example, while Neo4j offers a `shortestPath` function (see listing 1), it is limited to node distance and does not accept weights for the edges. To use Dijkstra's algorithm, the Graph Data Science (GDS) Library [16] needs to be installed. While allowing extensions to provide algorithms solves the problem of missing algorithms, this leads to a new problem: Lack of standardization. While an extension may exist for one of the Cypher implementations (Neo4j), this is likely not the case for other implementations such as RedisGraph or Cypher for Apache Spark.

VI. Challenges in Graph Analytics

Graph analytics is a popular use case for graph databases and one of their major selling points. Graph structures are generally well-suited for mapping relationships and discovering patterns and coherences. However, given the heterogeneity, the lack of standards and scalability issues, providing analytical functionality and optimizing performance for analytical workloads becomes challenging. In this section, we discuss current limitations and key properties of graph analytics to understand how this situation can be improved. Subsequently, we examine TigerGraph as a promising distributed graph database solution.

A. Functionality and Performance Limitations

Although native graph structures are suited for analytical queries, graph databases available today often lack extensive analytical functionality. In fact, the exemplary Cypher query in listing 1 presents a rare occasion where analytical functionality is built directly into the database system. The `shortestPath()` function in Cypher returns the shortest path between nodes in a graph. Unfortunately, few graph database systems and graph query languages support such built-in functions for computing analytical aspects of a graph. Even with Cypher, analytic capabilities are very limited. In fact, the Neo4j Graph Data Science (GDS) Library [16] is the recommended extension framework for graph data science in Cypher. The GDS Library supports various algorithms e.g. for determining the community grouping, centrality, and similarity of nodes. Additionally, it offers more complex methods for

path finding as well as machine learning capabilities like heuristic relationship prediction and node embedding. However, such complex graph algorithms are not supported directly in current graph databases.

Further, analytical use cases may require queries to be executed on a certain part of the graph or the results of a previous query. However, using query results for further processing with subqueries is limited and current graph query languages lack the expressiveness required. This is particularly obstructive for graph analytics as analyses may need the execution context of another analytic query.

Besides insufficient support for complex graph algorithms, performance of analytical workloads is a general concern. In the context of graphs spanning billions of nodes and edges [5], real-time online analytical processing (OLAP) poses tremendous requirements in terms of scalability and parallelization.

With this in mind, it is not surprising that a third-party framework is commonly introduced to help mitigating some of these issues. Strong contenders in this field include GraphLab [17], Galois [18], Pregel [19], Apache Giraph [20], and Gunrock [21]. These frameworks offer complex graph computing functionality and facilitate analytical processing of graph data. In this case, the graph database merely acts as a backend providing data for computation in the frameworks. Metadata and parts of the graph are loaded into memory in order to work around the performance limitations of disk-based databases and scalability issues. However, this seems redundant to a certain degree. Since data is already stored as a native graph structure in most graph databases, integrating complex analytical functionality directly into the database will further improve performance while simplifying the overall workflow. Instead of loading data into a third-party framework, graph databases could directly process analytical queries.

B. Key Properties of Graph Analytics

Listing 1 depicts a basic analytical query found in various use cases. For a pair of nodes `(n1, n2)`, this query returns the shortest possible path between them. Since no further filters are applied to the graph elements, the query processes all node pair combinations. While naively iterating over all node combinations would generate the correct set of results, iterating in this fashion would be rather slow. Luckily, it is possible to exclude combinations and obtain the same set of results through the use of graph algorithms.

Listing 1
A SHORTEST PATH QUERY IN CYPHER

```
MATCH (n1), (n2),
    path = shortestPath((n1)-[*..]->(n2))
RETURN path
```

Based on this example, we can identify two key properties of analytical queries in a graph:

- Node neighbors: Analytical queries often involve neighbors of nodes. Proceeding from a certain node, the query is commonly concerned with analyzing direct and transitive neighbors and their relationships.
- Iterative queries: Most analytical graph workloads are not only executed repetitively depending on the use case, but also traverse the graph iteratively. This results in challenging situations with large graphs as iterations and related operations get increasingly costly to compute. Some analyses may even require exponentially more query execution time with increasing node count.

With this in mind, the core solution to the performance and scalability issues of graph analytics can be based on a MapReduce-like [22] approach. Since an analytical query consists of a multitude of iterative operations concerning the respective nodes in proximity, the analysis can be split into many smaller parts. These subproblems can be distributed to available processing nodes in order to execute them in parallel (map phase). Subsequently, intermediate results of these subproblems are combined to form the overall analytical outcome of the query (reduce phase).

Unsurprisingly, this solution is commonly applied by third-party graph computing frameworks to achieve performance optimization and graph processing at scale. However, as discussed in section VI-A, employing third-party frameworks to compute analyses on data already available as native graph structures in a database seems redundant.

C. Graph Analytics in TigerGraph

TigerGraph is a distributed graph database system built with big data requirements and scalability in mind [23]. It is specifically optimized for massively parallel processing (MPP) of queries and graph analytics. Built as a native distributed graph store, TigerGraph stores data on disk and uses compression and in-memory caching to optimize data access speeds. Data is intelligently partitioned and sliced into parts by the internal storage engine. Subsequently, these storage units are distributed across a cluster of available nodes, which improves the performance of analytical queries in particular. Each server in the cluster contributes to the outcome of the query by processing local data and communicating the results with other servers as needed. As a result, TigerGraph can exploit the natural parallelizability of graph path traversal operations.

Like many graph database systems, TigerGraph introduces its own query language "GSQL" to support expressive query statements. GSQL is turing complete and offers transparent parallelization. The Data Definition Language (DDL) of GSQL implements a high-level SQL-like interface to insert, read, update and delete data (known as "CRUD") in the graph. Besides its influence from SQL, GSQL also supports language features that resemble NoSQL concepts. Loops and accumulators facilitate the iterative nature of graph queries and enable graph interpretation using MapReduce and bulk synchronous parallel (BSP) [24] processing. In particular, GSQL supports SQL-like `GROUP BY` aggregations as well as aggregations based on an `ACCUM` clause [25]. The latter facilitates aggregating inputs in parallel by specifying compute functions and accumulation variables. Similar to a MapReduce

concept, the `ACCUM` clause is executed once for every yield by the previous `WHERE` clause. Subsequently, generated inputs are reduced by the defined accumulators to form aggregation results.

VII. GQL Standardization Efforts

As discussed previously and criticized in related work, advancements in the graph database industry are severely hampered by a general lack of standardization. In the following, we examine the current efforts around GQL, an upcoming proposal for a property graph query language standard.

A. History and Goals of GQL

GQL is a graph query language standard for property graphs originally described in the informal GQL Manifesto [26] and later approved as an official standard by the ISO/IEC Joint Technical Committee 1 (JTC1). Like SQL for relational databases, GQL aims to be a declarative query language unifying the diverse ecosystem of existing graph query languages. In this regard, GQL will adopt the fundamental concepts of SQL to ensure compatibility and interoperability between the two standards. Consequently, GQL builds on existing concepts to implement native graph structures with language support for complex pattern matching and composable views.

The standardization efforts are supported by various stakeholders in the graph database community including Neo4j, TigerGraph, Redis Labs, and Oracle. However, GQL is still in its conceptional phase and no real-world implementations for existing graph databases are available at this time.

B. The Best of All Worlds

The working group around GQL examines existing graph query languages to extract the most useful concepts and condense them into a unified standard [27]. Although not complete, the following list presents the key elements taken from their respective language:

1) Cypher: Neo4j's Cypher query language and its open counterpart "openCypher" [28] are among the most widely adopted graph query languages and lay an important foundation for many ideas in GQL. In particular, GQL will implement support for full create, read, update, delete (CRUD) operations. Additionally, GQL intends to operate on native and composable graph structures while also allowing the handling of subgraphs and views ("omnigraphs").

2) TigerGraph GSQL: Besides the analytical advancements discussed in section VI-C, GSQL also supports named graphs and schema specifications. A restricted graph schema allows for more specific optimization in contrast to schema-less databases. Schema specifications along with metadata are saved into a catalog for global referencing.

3) Oracle PGQL: PGQL is a graph query language based on an SQL-like syntax and developed by Oracle [29]. Particularly, it features native support for intrinsic graph element types and advanced graph pattern matching techniques using complex path expressions. Exemplary, PGQL queries can match results using regular expressions and allow combinations of different fine-granular matching patterns.

4) G-CORE: G-CORE is a research graph query language design that subscribes itself to two key characteristics: composable graphs for input and output of queries as well as treating paths as first-class citizens [30]. To a degree, G-CORE is a precursor of GQL, taking influences from other graph query languages to improve standardization in the industry.

5) Configurable Match Semantics: Motivated by the different interpretations of path semantics discussed in section IV, GQL aims to implement configurable semantics. Consequently, graph databases with differing semantics could still rely on GQL as a unified standard interface.

VIII. Conclusion

In this work, we revisited the core concepts of graph databases and investigated state-of-the-art challenges. Although graph databases are not a new technology, we observed a general lack of standards in the industry as well as severe performance limitations due to the constrained scalability of some systems. While similarities between different graph databases and graph query languages exist, we also found fundamental differences i.e. the interpretation of path semantics. Based on this insight, we examined current challenges regarding graph query processing and graph analytics.

For graph query processing, we found that various theoretical algorithms exist with the potential to improve query performance. However, current graph databases have rarely implemented these algorithms. Additionally, the heterogeneity of graph databases complicates the usage of algorithms, as different concepts such as path semantics must be taken into account.

Although many use cases require analytic query functionality, most graph databases lack analytical capabilities and the required performance. As a consequence, users often rely on third-party frameworks for graph analysis. In this regard, TigerGraph represents a promising approach to distributed graph processing at scale.

Finally, standardization efforts around GQL are aiming to unify the property graph database community. While its adoption and success remains to be seen, GQL lays an important foundation for graph database development.

For future work, we expect further advancements regarding distributed graph databases, especially in terms of scalability and query performance. Particularly, GPU acceleration of parallelizable query parts could take graph processing and analysis to new heights. Additionally, real-time stream processing for graph analytics could improve performance by solely processing newly added graph data.

References

[1] E. F. Codd, "A relational model of data for large shared data banks," in *Software pioneers*. Springer, 2002, pp. 263–294.

[2] D. Anikin, O. Borisenko, and Y. Nedumov, "Labeled property graphs: SQL or NoSQL?" in *2019 Ivannikov Memorial Workshop (IVMEM)*. IEEE, 2019, pp. 7–13.

[3] E. Miller, "An introduction to the resource description framework," *Bulletin of the American Society for Information Science and Technology*, vol. 25, no. 1, pp. 15–19, 1998.

[4] J. Z. Pan, "Resource description framework," in *Handbook on ontologies*. Springer, 2009, pp. 71–90.
[5] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing: extended survey," *The VLDB Journal*, vol. 29, no. 2, pp. 595–618, 2020.
[6] R. Angles, H. Thakkar, and D. Tomaszuk, "RDF and Property Graphs Interoperability: Status and Issues." in *AMW*, 2019.
[7] J. Pokornỳ, "Graph databases: their power and limitations," in *IFIP International Conference on Computer Information Systems and Industrial Management*. Springer, 2015, pp. 58–69.
[8] R. Angles, M. Arenas, P. Barcelo, A. Hogan, J. Reutter, and D. Vrgoc, "Foundations of Modern Query Languages for Graph Databases," 2017.
[9] H. Q. Ngo, "Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems," *CoRR*, vol. abs/1803.09930, 2018.
[10] A. Atserias, M. Grohe, and D. Marx, "Size Bounds and Query Plans for Relational Joins," in *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, 2008, pp. 739–748.
[11] D. Nguyen, M. Aref, M. Bravenboer, G. Kollias, H. Q. Ngo, C. Ré, and A. Rudra, "Join processing for graph patterns: An old dog with new tricks," in *Proceedings of the GRADES'15*, 2015, pp. 1–8.
[12] C. R. Aberger, A. Nötzli, K. Olukotun, and C. Ré, "EmptyHeaded: Boolean Algebra Based Graph Processing," *CoRR*, 2015.
[13] A. Hogan, C. Riveros, C. Rojas, and A. Soto, "A Worst-Case Optimal Join Algorithm for SPARQL," in *International Semantic Web Conference*. Springer, 2019, pp. 258–275.
[14] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *NUMERISCHE MATHEMATIK*, vol. 1, no. 1, pp. 269–271, 1959.
[15] K. Losemann and W. Martens, "The Complexity of Evaluating Path Expressions in SPARQL," in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2012, p. 101–112.
[16] Neo4j. Neo4j Graph Data Science Library. [Online]. Available: https://neo4j.com/product/graph-data-science-library/
[17] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein, "GraphLab: a new framework for parallel machine learning," in *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, 2010, pp. 340–349.
[18] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 456–471.
[19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 135–146.
[20] S. Sakr, F. M. Orakzai, I. Abdelaziz, and Z. Khayyat, *Large-scale graph processing using Apache Giraph*. Springer, 2016.
[21] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel *et al.*, "Gunrock: GPU graph analytics," *ACM Transactions on Parallel Computing (TOPC)*, vol. 4, no. 1, pp. 1–49, 2017.
[22] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[23] A. Deutsch, Y. Xu, M. Wu, and V. Lee, "Tigergraph: A native MPP graph database," *arXiv preprint arXiv:1901.08248*, 2019.
[24] L. G. Valiant, "A bridging model for multi-core computing," *Journal of Computer and System Sciences*, vol. 77, no. 1, pp. 154–166, 2011.
[25] A. Deutsch, Y. Xu, M. Wu, and V. E. Lee, "Aggregation support for modern graph analytics in TigerGraph," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 377–392.
[26] A. Green. The GQL Manifesto. [Online]. Available: https://gql.today/
[27] R. Angles, A. Deutsch, T. Frisendal, V. Lee, R. Lipman, J. Lovitz, P. Selmer, H. Thakkar, O. van Rest, M. Wu, and B. Iordanov. Existing Languages. [Online]. Available: https://www.gqlstandards.org/existing-languages
[28] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, M. Schuster, P. Selmer *et al.*, "Formal semantics of the language cypher," *arXiv preprint arXiv:1802.09984*, 2018.
[29] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi, "PGQL: a property graph query language," in *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, 2016, pp. 1–6.
[30] R. Angles, M. Arenas, P. Barceló, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda *et al.*, "G-CORE: A core for future graph query languages," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1421–1432.