algorithms

MDPI

Systematic Review

Domain-Specific Languages for Algorithmic Graph Processing: A Systematic Literature Review

Houda Boukham *, Kawtar Younsi Dahbi and Dalila Chiadmi

SIP Resarch Team, Ecole Mohammadia d'Ingénieurs, Mohammed V University in Rabat, Rabat 10102, Morocco; younsidahbi@emi.ac.ma (K.Y.D.); chiadmi@emi.ac.ma (D.C.)
* Correspondence: houda_boukham@um5.ac.ma

Abstract

Graph analytics has grown increasingly popular as a model for data analytics across a variety of domains. This has prompted an emergence of solutions for large-scale graph analytics, many of which integrate user-facing domain-specific languages (DSLs) to support graph processing operations. These DSLs fall into two categories: query-based DSLs for graph-pattern matching and graph algorithm DSLs. While graph query DSLs are now standardized, research on DSLs for algorithmic graph processing remains fragmented and lacks a cohesive framework. To address this gap, we conduct a systematic literature review of algorithmic graph processing DSLs aimed at large-scale graph analytics. Our findings reveal the prevalence of property graphs (with 60% of surveyed DSLs explicitly adopting this model), as well as notable similarities in syntax and features. This allows us to identify a common template that can serve as the foundation for a standardized graph algorithm model, improving portability and unifying design between different DSLs and graph analytics toolkits. We additionally find that, despite achieving remarkable performance and scalability, only 20% of surveyed DSLs see real-life adoption. Incidentally, all DSLs for which user documentation is available are developed as part of academia–industry collaborations or in fully industrial contexts. Based on these results, we provide a comprehensive overview of the current research landscape, along with a roadmap of recommendations and future directions to enhance reusability and interoperability in large-scale graph analytics across industry and academia.

Keywords: graph algorithms; domain-specific languages; graph analytics; high-performance computing

check for updates

Academic Editor: Mauro Mezzini

Received: 5 June 2025
Revised: 1 July 2025
Accepted: 4 July 2025
Published: 19 July 2025

Citation: Boukham, H.; Younsi Dahbi, K.; Chiadmi, D. Domain-Specific Languages for Algorithmic Graph Processing: A Systematic Literature Review. *Algorithms* **2025**, *18*, 445. https://doi.org/10.3390/a18070445

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Graphs have grown increasingly prominent as a data model that enables the study of relationships and structures in interconnected data. As a result, graph analytics have become ubiquitous across a variety of domains, including biology, security, logistics and planning, social sciences, finance, and linguistics [1]. As the graph modeling of data becomes more widespread, a multitude of solutions have emerged from both academia and industry for large-scale graph analytics. Several of these solutions provide their users with domain-specific languages (DSLs) for intuitively and concisely expressing their graph processing operations while abstracting over underlying implementation concerns. We can distinguish two types of DSLs with regard to graph analytics: graph query DSLs and DSLs for algorithmic graph processing.

Graph query DSLs (such as Neo4j's Cypher [2], Oracle's PGQL [3] or TigerGraph's GSQL [4]) provide SQL-like abstractions for querying and matching patterns against graph data. These DSLs are now formalized in GQL, a standard graph query language published as of April 2024 as part of ISO/IEC 39075:2024 [5]. This standardization effort builds upon existing industry practices in order to provide a technical foundation that increases confidence in adhering solutions, promotes cross-platform portability and interoperability, and closes the functionality gap between different graph query DSLs.

In contrast, graph algorithms iterate over a graph through complex traversals, for example, to explore paths between nodes, the importance of nodes, or the clustering of nodes. This generally requires more sophisticated algorithmic logic than can easily be expressed in a query. Real-life application examples of graph algorithms include finding the shortest path in maps [6], detecting communities in social networks [7], ranking web pages in search engines [8], or detecting fraud in fiscal transactions [9]. While several DSLs for algorithmic graph processing have emerged, the domain still lacks a standard that would unify design, implementation, and testing practices.

In this paper, we review existing DSLs for algorithmic graph processing in order to identify a common template for graph algorithms. Several surveys and literature reviews have been conducted around graph processing [10–15] or graph querying [16–19]. However, our review is—to the best of our knowledge—the first to specifically focus on DSLs for algorithmic graph processing. To address the gap in literature left unaddressed in previous work, we pose our main research question: What unified template standardizes DSLs for algorithmic graph processing? In answering this question, our paper makes the following contributions:

- A systematic review of DSLs for algorithmic graph processing and their domain-specific abstractions;
- An exploration of cost-effective approaches to DSL implementation,
- An overview of the scalability and performance of state-of-the-art DSLs for algorithmic graph processing;
- A reflection on the opportunities and challenges of using DSLs in industrial large-scale graph analytics.

The remainder of this paper is structured as follows. Section 2 introduces the methodology used for our systematic literature review, and formulates our objectives as research questions. Section 3 presents our findings, synthesizing key insights with relevant figures and tables. Section 4 discusses the implications of our findings and highlights future research directions. Finally, Section 5 offers a conclusion.

2. Methodology

We conducted and reported on our literature review as per the guidelines of the PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) statement [20]. This section elaborates on the methodology and describes the setup for our systematic review.

2.1. Research Questions

Our literature review aims to answer the following research questions:

- RQ1. Which graph models are commonly employed by DSLs for algorithmic graph processing, and which model is predominant?
- RQ2. What common syntactic and semantic domain-specific abstractions do DSLs provide for this model?
- RQ3. What are the possible approaches for building a DSL for algorithmic graph processing?

- RQ4. How do existing DSLs for algorithmic graph processing compare in terms of supported platforms, scalability, and computational performance?
- RQ5. Can these DSLs integrate into third-party software or with other DSLs?
- RQ6. To what extent are these DSLs used in practice?

2.2. Eligibility Criteria

To be eligible for review, a paper needed to describe a DSL for algorithmic graph processing. Papers that do not introduce a DSL, but that implement a compiler for an existing DSL for algorithmic graph processing or otherwise introduce a solution, case study, or experience report that relates to an existing graph DSL, were also screened. They were included in our report if they provided relevant information and implementation details on the DSL in question. Query-based graph DSLs (in particular, DSLs for graph pattern-matching queries) generally do not allow the expression of complex graph algorithms intuitively and concisely. We chose to leave them out of the scope of this paper.

Therefore, our inclusion criteria are as follows:

- IC1. The paper describes a DSL for algorithmic graph processing.
- IC2. The paper implements a compiler for an algorithmic graph processing DSL.
- IC3. The paper describes research, case study, or experience report involving an existing DSL for algorithmic graph processing.

Similarly, our exclusion criteria are as follows:

- EC1. The paper does not relate to graph analytics.
- EC2. The paper does not describe a DSL for graph algorithms.
- EC3. The paper describes a query-based DSL for graph algorithms.

In the rest of the paper, we use graph DSLs for concision, but it should be understood that we refer to DSLs for algorithmic graph processing.

2.3. Search Strategy

We conducted electronic searches for eligible studies within three databases: Scopus, the ACM digital library, and IEEE Xplore. The aforementioned databases were searched within article metadata (title, abstract, and keywords) using the following search prompt: “graph algorithm” AND “DSL”. The date coverage was 2000 to the present. A computer science filter was applied to the search results. Conference reviews were excluded from the document type to avoid duplicates with corresponding conference papers. Full-text documents of eligible studies were retrieved from the ACM digital library, IEEE Xplore, Elsevier’s ScienceDirect, and Springer Nature link. We collected additional information from online documentation of products whose eligibility is known and from cross-reference search of publications with a scope similar to ours.

2.4. Selection Process

The first, second, and third authors (hereinafter referred to as H.B., K.Y.D., and D.C., respectively) independently reviewed the database search results for eligibility by reading abstracts and verifying whether they fell into our scope and corresponded to our inclusion criteria. For example, searching for graph algorithm DSLs yields a few results describing a DSL that ultimately relates to specific types of graphs (e.g., control-flow graphs, or scope graphs), but which is not directly used to express graph algorithms. When it was unclear whether a study was eligible, the three reviewers screened the full text and discussed its contributions until a consensus was reached. After this initial screening, H.B. further screened the full texts of all retained studies according to the eligibility criteria we set in Section 2.2. In cases of uncertain eligibility, D.C. was consulted. The final list of included studies was reviewed by D.C.

2.5. Data Collection Process

Data collection was performed manually and synthesized in relevant tables. Specifically, for articles describing a DSL, we retrieved the following information when available: the property graph model used, syntax characteristics, type system, domain-specific abstractions, platforms targeted by the DSL, benchmark results, tools used to implement the DSL, whether the DSL is used commercially, and any relevant distinctive features. When up-to-date online documentation with a detailed language specification was found for a DSL, it was favored over the article for information retrieval.

3. Results

In this section, we report our study selection process as per the PRISMA guidelines. Based on the data collected from selected articles, we answer the research questions laid out in Section 2.1. We use tables and charts to synthesize retrieved data where applicable.

3.1. Study Selection

We count 30 reports in our selected studies, between published articles, online documentation, and language specification documents. These reports either offer a graph DSL as one of the main contributions [21–37], describe research work that relates explicitly or implicitly to a given graph DSL [38–48], or otherwise fall under related work [49]. Figure 1 reports on our study selection process in a PRISMA diagram flow. The template used was retrieved from the 2020 PRISMA statement website, as is credited in the figure. From the reports retained for this review, we count 10 graph DSLs, chronologically ordered in Figure 2. Years are given based on the earliest publication date and might not reflect the actual development date. It is worth noting that a substantial number of studies were excluded after the initial screening of titles, keywords, and abstracts, bringing down the number of reports to assess from 110 to 28. This is because graphs (and consequently graph algorithms) are used in fields other than graph analytics, namely programming languages and compilers. Generally, studies that pertain to the domain we are interested in will explicitly list "graph processing", "graph analytics", "graph analysis", or "graph algorithms" among their keywords, with DSL (or domain-specific language) additionally appearing in the title and abstract fields. We did not make our search prompt more restrictive lest it filter out eligible studies; instead, we opted to exclude irrelevant papers manually at the screening stage.

3.2. Result Synthesis

3.2.1. RQ1. Graph Model Taxonomy

Table 1 categorizes the different DSLs we reviewed, depending on the graph model they adopt. We note three reoccurring models:

- The property graph model [50], which associates nodes and edges of the graph with arbitrary properties as key-value pairs. This model was the subject of the ISO standards for SQL/PGQ, which defines a way of building property graphs over relational data, and for GQL, a property graph query language [5].
- The GraphBLAS specification [51], which defines building blocks for graph algorithms in linear algebra.
- The vertex-centric model [52], which considers a local, vertex-oriented view of the graph, as opposed to a global view, and is thus more suitable for writing distributed implementations of graph algorithms. The vertex-centric model was first introduced in a paper on Pregel [53].

Table 1. Graph model taxonomy.

Property Graph Model	GraphBLAS Model	Vertex-Centric Model
GreenMarl [21]	PyGB [32]	Palgol [28]
Elixir [22]	Peng et al. [33]	Fregel [36]
Falcon [23]		
GraphIt [29]		
PGX Algorithm [27]		
StarPlat [37]		

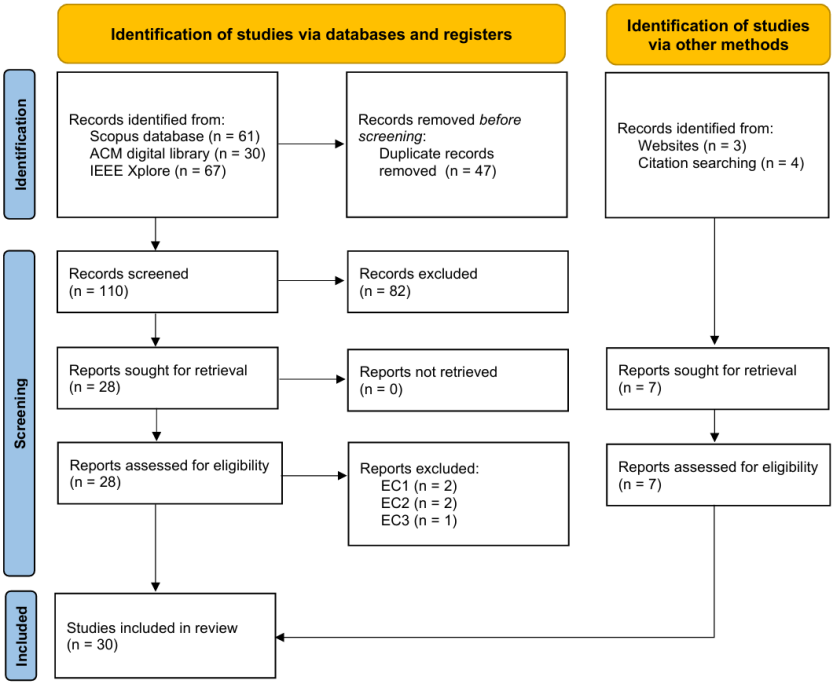


Figure 1. PRISMA flow diagram for our systematic review. Template source: https://www.prisma-statement.org/prisma-2020-flow-diagram (accessed on 3 July 2025).

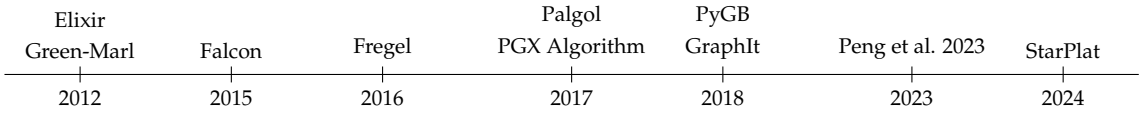


Figure 2. Timeline of surveyed graph DSLs. The year is given based on the earliest publication [33].

Sixty percent of the reviewed DSLs explicitly adopt the property graph model. It is worth noting that these models are not necessarily mutually exclusive: a property graph DSL can also be vertex-centric or target a GraphBLAS-based platform. With that in mind, we can safely say that the property graph model is indeed prevalent. The remainder of our results focus on this model. A popular counterpart to property graphs, which we do not encounter in our systematic review, is the Resource Description Framework (RDF) standardized by W3C [54], which models data as labeled edge-directed graphs. The standard query language for RDF is SPARQL. Several related works consider property graphs and RDF to be the most widely used graph models [5,15,17,40].

3.2.2. RQ2. Domain-Specific Abstractions for Property Graphs

Table 2 synthesizes the domain-specific abstractions introduced via DSLs for property graphs. These include semantic abstractions (domain-specific types) or syntactic abstractions, which we list below.

Table 2. Domain-specific abstractions provided by reviewed DSLs for property graphs.

		Elixir	Green-Marl	Falcon	GraphIt	PGX Algorithm	StarPlat
domain-specific types	graph	Graph	graph	Graph		PgxGraph	Graph
	node	node	vertex	Point	Element	PgxVertex	node
	edge	edge	edge	Edge	Element	PgxEdge	edge
	property	implicit	vertexProperty edgeProperty	implicit	Vector	VertexProperty EdgeProperty	propNode propEdge
	collection			Set Collection	VertexSet EdgeSet	VertexSequence EdgeSequence VertexSet EdgeSet	
graph iteration	sequential	for	for	for	for	for	for
	parallel	foreach	foreach	foreach	in scheduling language	foreach	forall
	fixed point	iterate					fixedPoint
	BFS		inBFS			Traversal.inBFS	
	DFS		inDFS			Traversal.inDFS	
aggregation operations	numeric		sum product max min avg	ReduxSum ReduxMul	Sum Max	sum max min avg	
	boolean		any all			anyMatch allMatch	
property operations	property read		n.p e.p	n.p e.p	p[n] p[src, dst]	p.get(n) p.get(e)	n.p e.p
	single-point update	a rewrite rule, [nodes(node n, prop p)] → [p = 0]	n.p = v; e.p = v;	n.p = v; e.p = v;	p[n] = v; p[src, dst] = v;	p.set(n, v); p.set(e, v);	n.p = v; e.p = v;
	batch update	iteration matching the update operator against the graph	iteration updating every element	iteration updating every element	function applied to set	iteration updating every element	iteration updating every element

- Domain-specific types embody graph-related notions, such as graphs, nodes (also called vertices), edges, graph properties, or collections of graph elements.
- Graph iterations are constructs for iterating over a set of graph elements (i.e., set of nodes or set of edges). In other words, they are loops specified for graph elements. These loops can be sequential, parallel, or a specific graph traversal (breadth-first search or depth-first search).
- Aggregations are constructs that perform a reduction operation (such as sum, average, max, or min) over a set of graph elements. For example, a sum aggregation over a set N of graph nodes equates to performing a mathematical sum over elements of N.
- And finally, property operations allow users to read the value of a property or update the value of a property. A property update can be performed for a single node or edge (single-point property update) or for a set of nodes or edges (batch property update).

We observe that Green-Marl, Falcon, GraphIt, and StarPlat adopt a very similar, C-like syntax, whereas PGX Algorithm's syntax is a subset of Java, and Elixir adopts a more declarative approach. Some DSLs provide built-in abstractions for concepts that need to be manually implemented in other DSLs, such as BFS and DFS traversals and fixpoint iterations. The graph type, which is explicitly provided in all but one DSL, is implicitly constructed in GraphIt from EdgeSets. Based on the language specifications available to us, four of the six property graph DSLs offer built-in aggregation operators. Five implement batch property updates as a loop iterating over a set and, for each element, updating the property value associated with it. GraphIt instead implements this as a built-in function that applies the update for a set of vertices or edges.

Despite their notable closeness in syntax, some DSLs still have distinctive features that others do not. For example, Falcon and GraphIt allow the morphing of the graph, whereas the remaining DSLs can only alter the values of its output properties. Another particularity is providing a separate scheduling language for specifying performance optimizations, as is the case for Elixir and GraphIt.

3.2.3. RQ3. Approaches for DSL Implementation

Implementing a DSL requires implementing a processor (generally a compiler or an interpreter) to execute programs in this language. All of the DSLs we review are compiled DSLs. Manually building a DSL and its compiler gives the developer full control over the implementation. Then, a secure and efficient implementation depends only on the developer's skill. However, this approach can be costly and involves a lot of boilerplate work. DSLs are often developed in small teams with limited resources, which makes it all the more important to have cost-effective tooling that supports both the developers (parser, type checker, and code generator) and the users of a DSL (IDE and debugger). The term language workbench (popularized by Martin Fowler [55]) refers to a category of such tools, which typically provide an IDE environment for the meta-development of a DSL and its compiler, and for the DSL itself. Language workbenches promise an accessible, efficient, and affordable approach to DSL implementation. Several case studies in the literature attest to a positive return on investment and an increase in productivity with the language workbench approach [42,56–62]. Notable language workbench examples include MPS [63], Xtext [64], Rascal [65], and Spoofax [66].

An alternative to developing DSL processors from scratch using language workbenches is to hook a DSL's frontend into an existing compiler architecture and reuse the compiler components it provides. One widely used example is LLVM [67], which offers a toolchain of modular and reusable components and technologies for building compilers. LLVM defines at its center a low-level, assembly-like IR in Static Single Assignment (SSA) form [68], allowing for a unified code representation of high-level languages. DSLs can

map their frontends to LLVM IR to benefit from its source- and target-independent optimizer, as well as its code generation support for many popular CPUs. Many programming languages rely on LLVM for various compilation tasks, namely Ruby, Python, Haskell, Rust, D, PHP, Pure, Lua, Julia, and Swift. However, the low-level abstractions of LLVM IR make it potentially unsuitable for high-level compilers, whose target platforms typically require higher-level paradigms than what LLVM offers. MLIR [69] (multi-level intermediate representation) is another project under the LLVM umbrella. Recognizing the restrictivity of LLVM IR's low-level abstractions, MLIR provides a customizable IR that allows its users to define the right level of abstraction for their programs. By allowing for multiple levels of abstraction to freely coexist, MLIR makes it possible to lower part of the representation and maintain another in a higher-level abstraction more suitable for specific transformations.

The DSL compiler presented by Peng et al. [33] is based on MLIR. Of the graph DSLs we review, two are implemented using a language workbench: namely Green-Marl and the PGX Algorithm, which rely on Spoofax. GraphIt, Falcon, and an early version of the Green-Marl compiler are implemented in C++. Brahmakshatriya et al. introduce their own solution, BuilDSL [48], an end-to-end framework for building DSLs using multistage programming in C++, which they use to implement a GraphIt to CUDA compiler. PyGB is implemented in Python, and it integrates with the NumPy, SciPy, and NetworkX libraries, for arrays, sparse matrices, and graphs, respectively. PyGB is dynamically compiled to C++.

3.2.4. RQ4. Target Platforms, Scalability, and Computational Performance

All surveyed studies that describe a DSL for algorithmic graph processing (IC1) include empirical evaluation to validate their results. This generally involves running DSL implementations of a number of commonly used graph algorithms against large-scale graphs. Graph algorithms that come up most often in evaluations are SSSP (Single-Source Shortest Path), PageRank, BFS, Triangle Counting, Connected Components, and Betweenness Centrality. Performance is validated in one of two ways: either by comparing DSL running times against those of state-of-the-art solutions or by comparing the performance of compiler-generated code against hand-tuned implementations and showing the performance speedup of applying certain compiler optimizations. DSLs that target more than one platform will generally run different benchmarks for each target. This heterogeneity of target platforms may be at the level of the memory architecture (shared-memory/distributed-memory), target language (Java, C, C++, Python, PL/SQL), or hardware architecture. Table 3 recapitulates the programming languages generated via the compilers of surveyed DSLs, as well as the memory architectures they target. While 40% of surveyed DSLs target both a shared-memory and a distributed-memory runtime, 40% exclusively target a single-machine environment. Among these, GraphIt targets different hardware architectures (CPU, GPU, Swarm, and HammerBlade), albeit all in a parallel, shared-memory context. The remaining 20% are vertex-centric DSLs intended for a distributed environment. Hardware architectures targeted by each DSL can be found in Table 4.

Table 3. Overview of the target languages and memory architectures supported with surveyed DSLs.

DSL	Target Language				Target Memory Architecture		
	C/C++	CUDA	Java	PL/SQL	Shared-Memory	Distributed	RDBMS
Elixir	✓				✓		
Green-Marl	✓		✓	✓	✓	✓	✓
Falcon	✓	✓			✓	✓	
GraphIt	✓	✓			✓		
StarPlat	✓	✓			✓	✓	
PyGB	✓				✓		
[33]	✓				✓		
Palgol	✓					✓	
Fregel	✓		✓			✓	

Table 4. Scalability overview of surveyed DSLs.

DSL	Target Platform	Graph Types Used					Largest Graph Used in Evaluation		
		Social Network	Web Graph	R-MAT	Road Network	Random Graph	Graph Type	#N	#E
Elixir	single-machine			✓	✓	✓	random graph	8.3 M	33.5 M
Green-Marl	single-machine	✓	✓				social network	41.6 M	1.4 B
	distributed	✓	✓			✓	web graph	77.7 M	2.9 B
Falcon	single-machine			✓	✓	✓	R-MAT	80 M	256 M
	distributed	✓	✓	✓			R-MAT	1.2 B	12 B
GraphIt	single-machine	✓	✓		✓		social network	65.6 M	3.6 B
StarPlat	single-machine	✓	✓	✓	✓	✓	social network	21.2 M	262 M
PyGB	linear algebra					✓		2048	92682
[33]	linear algebra	✓		✓	✓		social network	3.9 M	69.3 M
Palgol	distributed	✓	✓		✓	✓	social network	59.2 M	185 M
Fregel	distributed					✓	random graph	10 M	80 M

Existing surveys identify scalability as a key factor in graph analytics [11,13,14]. As such, DSL benchmarks validate performance through large input graphs, with a size scale between 10^7 and 10^{10} (the size of a graph being measured by the number of its edges). Table 4 provides an overview of graphs used for benchmarking. For each DSL and each target platform, it lists the graph types used and provides the size of the largest one. While the performance of PGX Algorithm is not directly tackled in the literature, the PGX Algorithm frontend maps to the Green-Marl compiler and, therefore, has the same implementation. The following types of graphs are commonly encountered: social networks, web graphs, road networks, R-MAT graphs, and random graphs. Social networks [70] (such as the Twitter, LiveJournal, or Friendster graphs) represent relationships between entities. Social networks typically exhibit small-world [71] and scale-free [72,73] properties, namely high local clustering, a small diameter, and power-law degree distributions [74]. Another scale-free network example is web graphs [75], whose vertices correspond to pages and whose edges correspond to hyperlinks on the World Wide Web. In contrast, road networks (such as the US road network) have large diameters and bounded degrees. R-MAT graphs are scale-free networks that are synthetically generated, based on the Recursive Matrix (R-MAT) method [76]. Finally, random graphs are created by randomly connecting a set of isolated vertices following a given model (e.g., the Erdős-Réyni model [77]). These graphs usually have a Poisson-like degree distribution and lack the clustering or community structures seen in real-world networks.

Typically, an experimental evaluation of a given DSL's performance will use more than one input graph, varying in size, to assess how scaling up affects performance. Graph topology and the nature of the graph algorithm are also factors with an impact on running times. For example, Zhang et al. [29] provide performance numbers for GraphIt on up to seven graphs and seven graph algorithms. The input graphs include social networks, road networks, and the Netflix rating dataset. The GraphIt implementation of PageRank outperforms state-of-the-art solutions on four out of five graphs, but it is outperformed by Gazelle on the road network, where the latter benefits from Vector-Sparse vectorization, which works well on graphs with low degrees. Because of this, it may be challenging to assess which solution performs best in the absolute: competitive analyses must consider the same algorithms and the same input graphs. Moreover, surveyed DSLs target different execution platforms with different data representations (graph, adjacency matrix, relational data), target languages, computing paradigms, and hardware architectures. Even for two DSLs targeting the same platform, running times must be obtained on the same machine to allow a meaningful and reliable comparison. These benchmarking difficulties have

been raised in existing surveys and related work [78,79]. That being said, some articles provide their own performance comparison against state-of-the-art DSLs. GraphIt reports superior performance to Green-Marl's shared-memory back end on different input graphs (between a web graph, a road network, and social networks) for five graph algorithms, namely PageRank, connected components, Bellmann–Ford, SSSP, and betweenness centrality. Falcon's collection-based Δ-stepping optimization [80] allows it to outperform Green-Marl on road networks. StarPlat also compares its performance against Green-Marl and other non-DSL-based solutions, and it finds that Green-Marl achieves the fastest times on most inputs.

3.2.5. RQ5. Third-Party Integration

DSLs for algorithmic graph processing are not standalone: they are typically building blocks of larger data analytics pipelines, and they are integrated into larger toolkits for large-scale graph analytics. Furthermore, some use cases require DSLs or their processors to integrate with other DSLs: for example, the same graph analytics product may combine two DSLs, or a DSL front end can map to another DSL in order to reuse its compiler. We survey the different levels of this third-party integration below.

Integration into Larger Toolkits for Graph Analytics

When used industrially, DSLs for algorithmic graph processing—such as Green-Marl, PGX Algorithm, or Gremlin—are integrated into larger software for graph analytics as part of a client-server architecture. Users can express their graph analytics logic in a dedicated DSL through interactions with a client, while the actual graph data is processed on a server. This has implications on security: the DSL compiler running on the server constitutes a potential target for attacks (e.g., denial-of-service), especially if it introduces vulnerabilities through transitive dependencies [81,82]. The academic DSLs we review are generally standalone, though this is a recognized limitation [30].

Integration into Larger Data Processing Pipelines

Graph analytics application scenarios typically combine graph querying and algorithmic graph processing. Generally, the different graph data processing operations can be categorized according to the pipeline shown in Figure 3. The first step is OLTP (online transactional processing), which refers to the real-time processing of graph data using a graph query DSL (such as PGQL or Cypher). After OLTP, data can undergo more complex analyses through OLAP (Online Analytical Processing) operations. This is the step where DSLs for algorithmic graph processing can be utilized either to implement graph algorithms available to users off-the-shelf or to allow users to express their own custom graph algorithm logic. Finally, graph data feed into various applications, including RAG (retrieval-augmented generation) for large language models (LLMs), or ML-based tasks, for example, to train or further tune a model. Such a model can then be used to provide more sophisticated insights or predictions on analyzed data [83]. Several graph analytics solutions that use a DSL approach provide utilities for both OLTP and OLAP and, therefore, integrate graph query DSLs, as well as DSLs for algorithmic graph processing. Oracle's PGX integrates PGQL, PGX Algorithm, and Green-Marl. Neo4j allows querying graph data using Cypher, and it provides a library of prebuilt graph algorithms that can be invoked via Cypher statements.

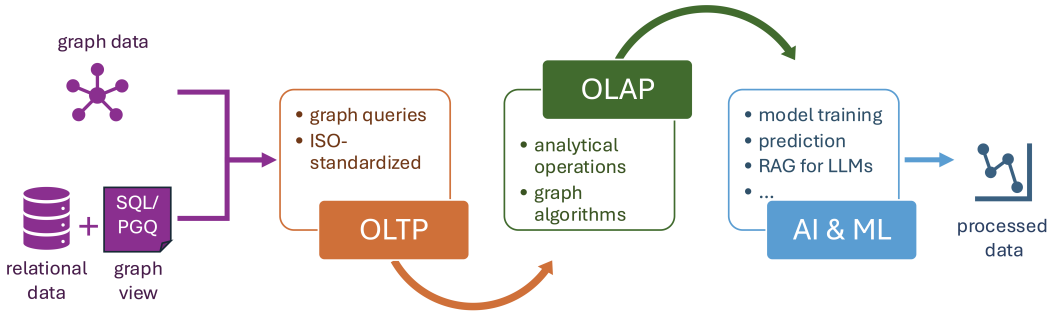


Figure 3. A typical graph data processing pipeline.

Integration into General-Purpose Languages

As explained in Section 3.2.3 (RQ3), some DSLs for algorithmic graph processing are syntactically a subset of a general-purpose language and/or are implemented in a general-purpose language. This serves two purposes. First, users are exposed to a general-purpose syntax they are familiar with (though some DSL developers prefer to design their own domain-specific abstractions). Second, DSLs embedded in a general-purpose language may reuse the tooling provided for this language, which aids both the DSL developer and the DSL users. PGX Algorithm is a subset of Java: a PGX Algorithm program is a syntactically valid Java program. Fregel is a subset of Haskell, and both the Haskell interpreter and compiler are implemented in Haskell. PyGB uses Python syntax and is itself implemented in the Python language. GraphIt opts against an embedded DSL approach, citing as motivations a customized domain-specific syntax and optimizations involving global transformations across different data structures and functions [30]. However, the author recognizes that DSLs for algorithmic graph processing are only part of larger data analytics and ML pipelines and lists the embedding of GraphIt into Python or C++ as future work that can ease this integration.

DSL Interoperability and Reuse

All DSLs we review map to a general-purpose language compiler at one level or another of the compilation process. DSL compilers are typically high-level source-to-source compilers: i.e., they generate code in a high-level general-purpose language, which is then further compiled. Some DSLs map to another DSL to partially or globally reuse its compiler or to provide users with a more accessible syntax than the DSL it maps to. For example, PGX Algorithm maps to the Green-Marl compiler in order to provide a Java-based syntax for graph algorithms. Emoto et al. [33] map a "global view style" DSL to the vertex-centric Fregel in order to provide users with a more natural syntax for writing graph algorithms. DH-Falcon [24] maps to the Falcon compiler in order to extend it with support for distributed graph processing.

3.2.6. RQ6. Usability and Real-Life Adoption

DSLs for algorithmic graph processing are developed in an academic context, an industrial context, or as part of an industrial-academic collaboration. Figure 4 gives the development context distribution of surveyed DSLs and shows the availability of online documentation for each context. Interestingly, all the DSLs for which we were able to find documentation with a language specification (Green-Marl, GraphIt, and PGX Algorithm) are developed in a fully or partially industrial context. Green-Marl was developed at Standford University in collaboration with Oracle, and it is used internally by the latter to implement over 60 graph algorithms available off the shelf in Oracle products. PGX Algorithm is developed at Oracle Labs as a DSL for writing graph algorithms using Java syntax. PGX Algorithm maps to the Green-Marl compiler and is used in several Oracle

products for large-scale graph analytics. The authorship affiliations for papers on GraphIt include both MIT (Massachusetts Institute of Technology) and Adobe Research, which hints at an industrial-academic collaboration. The vertex-centric DSLs we survey, as well as early work on Green-Marl, target execution platforms based on Pregel, a distributed graph processing framework developed at Google. None of the publications we review explicitly study the usability of their DSL solutions, though the intuitive and concise syntax is often mentioned as a benefit. Hong et al. [38] report on their early experiences using Green-Marl for industrial graph analytics, citing an intuitive programming model as a key benefit and identifying the learning curve and DSL documentation as areas for improvement.

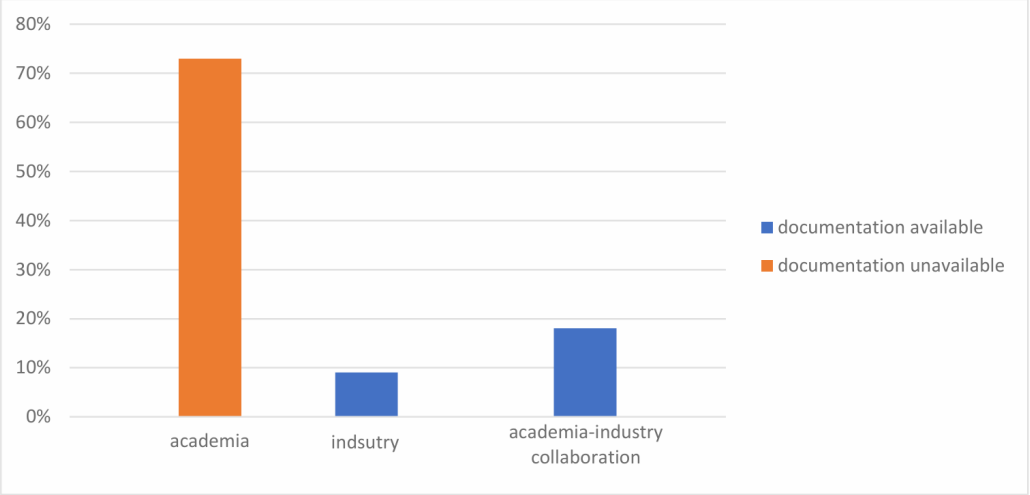


Figure 4. Availability of DSL documentation per development context.

3.3. Reporting Biases

We now discuss the potential bias risks in our study findings. The first risk is selection bias, as the databases we search archive research publications and thus may overlook industry sources. We counter this risk by expanding our search coverage to include a cross-reference checking of relevant publications, PhD theses, white papers, industry reports, and websites of industrial graph analytics vendors. The second risk concerns the sources we rely on in data collection (specifically with regard to DSL syntax, features, supported platforms, and performance), which may not be up to date. We limit this risk by referring to the most recent publications and online documentation when available.

4. Discussion

In this section, we recapitulate our main review findings, and we discuss the opportunities and challenges of adopting a DSL approach in large-scale graph analytics. Based on this, we offer our insights and recommendations for future research.

4.1. Our Findings' Highlights

4.1.1. Prevalence of the Property Graph Model

Our review of existing DSLs for algorithmic graph processing revealed a predominant reliance on the property graph model. This model is also notably preferred in industry (Amazon, DataStax, IBM, Microsoft, Neo4j, Oracle, and TigerGraph) [84], and it is the subject of recent ISO standards [85].

4.1.2. Common Domain-Abstractions for Graph Algorithms

DSLs adhering to the property graph model exhibit a notable overlap in features and close similarities in the domain-specific abstractions they provide. They all allow users

to express graph algorithms through statements that interact with graph data by reading or writing to properties. There is also a noted overlap in terms of targeted platforms (single-memory, distributed), with the PGX Algorithm compiler additionally targeting relational databases. These shared features make it possible to have a unified template for algorithmic graph processing.

4.1.3. Performance and Scalability as Key Factors for DSL Validation

DSLs for algorithmic graph processing are intended for large-scale graph analytics. It is, therefore, vital that they are able to efficiently process large real-world graphs going up to 10^{10} in scale (i.e., tens of billions). DSLs for algorithmic graph processing typically validate their performance in experimental evaluations run on real-life graphs (such as social networks, web graphs, or road networks) or synthetic graphs.

4.1.4. Between Research Prototypes and Real-Life Adoption

The majority of the DSLs we review are developed in an academic context. These DSLs generally lack dedicated websites, documentation, and user communities, which points to unclear or inexistent real-life adoption. They also tend to be presented as standalone solutions that generate high-performance programs, and it is not always clear to what extent these solutions can be integrated into existing third-party software or larger data analytics pipelines. This is problematic, as DSLs for graph analytics are not standalone, and their real-life adoption likely requires them to be integrated into larger software. That being said, some DSLs (either originating from or later picked up by industry) are integrated into commercial solutions for large-scale analytics and, therefore, see real-life use.

4.2. DSLs for Industrial Large-Scale Graph Analytics: Opportunities and Open Challenges

Our review findings highlighted several opportunities presented via the DSL approach, specifically in the algorithmic graph processing domain. However, despite this potential, the majority of DSLs developed in academic contexts do not see real-life adoption. In this section, we reflect on the potential of using DSLs for large-scale graph analytics, and we highlight the challenges they face as areas for improvement.

4.2.1. Potential of the DSL Approach

Productivity

Concisely expressing a graph algorithm in a non-graph-specific language can be a tedious task, a fortiori for novice or unaccustomed programmers. DSLs for algorithmic graph processing help manage this complexity through tailored semantics that allow programmers to encapsulate their graph algorithm logic while abstracting over implementation concerns. DSL implementations of graph algorithms are free of the boilerplate code that would otherwise be required for an API-based approach [21]. The reluctance to learn yet another language can be countered with fewer language constructs and a generally smaller grammar than what one might expect from a general-purpose language. Moreover, many DSL developers take this factor into account and strive for a familiar syntax that either mirrors an existing, commonly familiar language (e.g., Green-Marl, Falcon, GraphIt. . .) or is a subset of one (e.g., PGX Algorithm, PyGB).

Proven Benefit Gains

The DSL approach enables substantial performance gains, as DSL compilers leverage their precise knowledge of domain semantics to encode specific optimizations and produce high-performance programs. Several DSLs for algorithmic graph processing report superior performance to non-DSL-based state-of-the-art solutions.

Tools for Cost-Efficient DSL Implementation

Emerging DSLs for algorithmic graph processing can reuse existing DSL processors to save up on implementation efforts or extend existing DSL compilers with support for new frontends or new target platforms. Recent advancements in language workbenches and DSL tooling have lowered development costs and improved the DSL experience for users, as well as developers of the language. This addresses the issue of a lack of the DSL tool and support raised in existing literature [86]. Multiple case studies in industry and beyond report on a positive experience using these tools for domain-specific software language engineering [87]. DSL users can also benefit from state-of-the-art IDE tooling at minimal cost [88], which somewhat helps close the gap between DSLs and more widely used general-purpose languages.

4.2.2. Usability and Adoption Challenges

Uncertain Futures

Compared to general-purpose languages, DSLs have a short lifespan, and it is not uncommon that a DSL project gets abandoned or deprecated despite a non-negligible investment in the development and maintenance efforts. This notably (but not exclusively) happens in academic settings, where frequent changes of personnel (for example, due to PhD students graduating) or limited funding can cut short the future of a given project. This same risk extends to the tools that these DSLs are developed with and rely on.

Lack of Documentation and User Support

Limited or uncertain resources also impact the level of support that a DSL may be backed with, such as documentation, user support, and maintenance. Granted, not all DSLs are intended for wide adoption: some only serve as an exploratory prototype to answer some research question or lay the foundation for a more extensive and mature solution. Additionally, 73% of the DSLs we reviewed do not have, to the best of our knowledge, online documentation available. However, some DSLs do see real-life adoption, either by the domain community or specifically by a commercial graph analytics solution. There, having proper documentation and user guides is crucial.

Redundant Effort

A large portion of the DSLs we review are implemented manually, which may work for research-level prototypes. However, a fully manual language implementation involves redundant effort that could be avoided by utilizing existing tools for software language engineering. Moreover, the reviewed DSLs show an obvious overlap both in language features and in target platforms, with none acquiring a significant user base. This issue is not specific to the graph algorithms domain and can extend to DSLs in general. On average, DSLs have a shorter lifespan than their general-purpose counterparts and are more likely to be abandoned or deprecated, which, on top of being a waste of investment, is potentially problematic to parties that depend on these DSLs and might discourage adoption. Developing new DSLs for one's specific needs instead of picking up an existing one results in what Voelter calls "DSL Hell", in which we end up with a myriad of half-baked DSLs that never fully emerge.

4.3. Insights and Recommendations for Future Research

Based on patterns observed throughout our literature review, we identify several areas where further research and community effort could significantly benefit the development and adoption of DSLs for algorithmic graph processing. Below, we offer our key recommendations for future research.

4.3.1. Academia/Industry Collaboration

Collaboration with industry provides academically developed DSLs with ground for empirical case studies. This allows DSLs for algorithmic graph processing to progress past the prototype stage into active, real-life use, potentially acquiring a user community. Such a collaboration also increases development resources that can be invested in documentation or more future-proof implementations, thus solving at least a subset of challenges faced by DSLs.

4.3.2. A Standard for Algorithmic Graph Processing

Standardization is another effort where collaboration between academia and industry benefits both communities [89]. In the last two years, the International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) have published two standards relating to property graphs. As of June 2023, the standard for SQL (ISO/IEC 9075-16:2023) includes SQL/PGQ [90], which standardizes ways to define property graph views over relational data. In April 2024, GQL (ISO/IEC 39075:2024) was published as a standard for graph query languages [91]. These standards have been developed with input from various national bodies, with participating organizations across academia, industry, and beyond. GQL itself builds on two industrial graph query DSLs: Neo4j's Cypher and Oracle's PGQL. However, a similar standardization effort has yet to be undertaken for graph algorithms. Standards increase confidence in a given solution, normalize implementation best practices, and encourage real-life adoption. Moreover, having a lingua franca for graph algorithms (similar to SQL for relational databases or the now emerging GQL for graph queries) enables the transportability of knowledge across different platforms and helps build a large user community for the language. The recent literature has already identified the standardization of graph algorithms as a worthwhile future direction in the roadmap of graph analytics [92–94].

4.3.3. Graphs for Machine Learning

The intersection of algorithmic graph processing and machine learning presents a lot of potential for future research and frontier innovation [95]. One direction is to explore how DSLs can enable algorithmic graph processing for ML and LLM tasks [96]. The seamless integration of DSLs with ML and GenAI stacks paves the way for end-to-end graph analytics. A different but thematically related research direction is to utilize ML for compiler construction [97], particularly in compiler optimizations [98,99], in order to unlock the full potential of high-performance computing in graph analytics.

5. Conclusions

In this paper, we conducted a systematic review of DSLs for algorithmic graph processing. We found that the predominant graph model across graph algorithm DSLs in the literature is the property graph model, which associates the vertices and edges of a graph with arbitrary values, said properties. Our observation is that DSLs for the algorithmic processing of property graphs follow a common template: they all interact with graph data through the property read and property write operations. In principle, this makes it possible to work towards a common standard for graph algorithms that different DSLs can adhere to. While DSLs for algorithmic graph processing live up to their promise of efficient and scalable implementations of graph algorithms, some never move past the research prototype stage into active real-life use, nor do they provide suitable documentation and support to promote adoption. Industry collaboration could help bridge this gap and drive concrete efforts to achieve a standard for graph algorithms.

Author Contributions: Conceptualization, H.B.; methodology, H.B. and D.C.; validation, D.C. and K.Y.D.; formal analysis, H.B.; investigation, H.B. and K.Y.D.; resources, H.B.; data curation, H.B. and K.Y.D.; writing—original draft preparation, H.B.; writing—review and editing, D.C. and K.Y.D.; visualization, H.B.; supervision, D.C.; project administration, D.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data that support the findings of this study are available upon request from the corresponding author (H.B.).

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DSL	Domain-Specific Language
ML	Machine Learning
GenAI	Generative Artificial Intelligence

References

1. Hegeman, T.; Iosup, A. Survey of Graph Analysis Applications. *arXiv* **2018**, arXiv:1807.00382v1.
2. Francis, N.; Green, A.; Guagliardo, P.; Libkin, L.; Lindaaker, T.; Marsault, V.; Plantikow, S.; Rydberg, M.; Selmer, P.; Taylor, A. Cypher: An Evolving Query Language for Property Graphs. In Proceedings of the SIGMOD '18: 2018 International Conference on Management of Data, Houston, TX, USA, 10–15 June 2018; pp. 1433–1445. [CrossRef]
3. van Rest, O.; Hong, S.; Kim, J.; Meng, X.; Chafi, H. PGQL: A Property Graph Query Language. In Proceedings of the GRADES '16: Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, 24 June 2016. [CrossRef]
4. Deutsch, A.; Xu, Y.; Wu, M.; Lee, V.E. Aggregation Support for Modern Graph Analytics in TigerGraph. In Proceedings of the SIGMOD '20: 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 14–19 June 2020; pp. 377–392. [CrossRef]
5. Deutsch, A.; Francis, N.; Green, A.; Hare, K.; Li, B.; Libkin, L.; Lindaaker, T.; Marsault, V.; Martens, W.; Michels, J.; et al. Graph Pattern Matching in GQL and SQL/PGQ. In Proceedings of the SIGMOD '22: 2022 International Conference on Management of Data, Philadelphia, PA, USA, 12–17 June 2022; pp. 2246–2258. [CrossRef]
6. Edelkamp, S.; Schrödl, S. Basic Search Algorithms. In *Heuristic Search*; Edelkamp, S., Schrödl, S., Eds.; Morgan Kaufmann: San Francisco, CA, USA, 2012; pp. 47–87. [CrossRef]
7. Browet, A. Algorithms for Community and Role Detection in Networks. Ph.D. Thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2014.
8. Brin, S.; Page, L. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. Isdn Syst.* **1998**, *30*, 107–117. [CrossRef]
9. Chung, J. *Graph Data Science Use Cases: Fraud and Anomaly Detection*; White Paper; Neo4j: San Mateo, CA, USA, 2022.
10. Yan, D.; Bu, Y.; Tian, Y.; Deshpande, A. Big Graph Analytics Platforms. *Found. Trends Databases* **2017**, *7*, 1–195. [CrossRef]
11. Sahu, S.; Mhedhbi, A.; Salihoglu, S.; Lin, J.; Özsu, M.T. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.* **2017**, *11*, 420–431. [CrossRef]
12. Heidari, S.; Simmhan, Y.; Calheiros, R.N.; Buyya, R. Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges. *ACM Comput. Surv.* **2018**, *51*, 60. [CrossRef]
13. Shi, X.; Zheng, Z.; Zhou, Y.; Jin, H.; He, L.; Liu, B.; Hua, Q.S. Graph Processing on GPUs: A Survey. *ACM Comput. Surv.* **2018**, *50*, 81. [CrossRef]
14. Sahu, S.; Mhedhbi, A.; Salihoglu, S.; Lin, J.; Özsu, M.T. The ubiquity of large graphs and surprising challenges of graph processing: Extended survey. *VLDB J.* **2020**, *29*, 595–618. [CrossRef]
15. Besta, M.; Gerstenberger, R.; Peter, E.; Fischer, M.; Podstawski, M.; Barthels, C.; Alonso, G.; Hoefler, T. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *ACM Comput. Surv.* **2023**, *56*, 31. [CrossRef]
16. Singh, K.; Singh, V. Graph pattern matching: A brief survey of challenges and research directions. In Proceedings of the 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi, India, 16–18 March 2016; pp. 199–204.

17. Angles, R.; Arenas, M.; Barceló, P.; Hogan, A.; Reutter, J.; Vrgoč, D. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* **2017**, *50*, 68. [CrossRef]
18. Bhowmick, S.S.; Choi, B.; Li, C. Graph Querying Meets HCI: State of the Art and Future Directions. In Proceedings of the SIGMOD '17: 2017 ACM International Conference on Management of Data, New York, NY, USA, 14–19 May 2017; pp. 1731–1736. [CrossRef]
19. Bouhenni, S.; Yahiaoui, S.; Nouali-Taboudjemat, N.; Kheddouci, H. A Survey on Distributed Graph Pattern Matching in Massive Graphs. *ACM Comput. Surv.* **2021**, *54*, 36. [CrossRef]
20. Page, M.J.; McKenzie, J.E.; Bossuyt, P.M.; Boutron, I.; Hoffmann, T.C.; Mulrow, C.D.; Shamseer, L.; Tetzlaff, J.M.; Akl, E.A.; Brennan, S.E.; et al. The PRISMA 2020 statement: An updated guideline for reporting systematic reviews. *BMJ* **2021**, *372*, n71. [CrossRef] [PubMed]
21. Hong, S.; Chafi, H.; Sedlar, E.; Olukotun, K. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In Proceedings of the ASPLOS XVII: Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, London, England, UK, 3–7 March 2012; pp. 349–362. [CrossRef]
22. Prountzos, D.; Manevich, R.; Pingali, K. Elixir: A system for synthesizing concurrent graph programs. In Proceedings of the OOPSLA '12: ACM International Conference on Object Oriented Programming Systems Languages and Applications, Tucson, AZ, USA, 19–26 October 2012; pp. 375–394. [CrossRef]
23. Cheramangalath, U.; Nasre, R.; Srikant, Y.N. Falcon: A Graph Manipulation Language for Heterogeneous Systems. *ACM Trans. Archit. Code Optim.* **2015**, *12*, 54. [CrossRef]
24. Cheramangalath, U.; Nasre, R.; Srikant, Y.N. DH-Falcon: A Language for Large-Scale Graph Processing on Distributed Heterogeneous Systems. In Proceedings of the 2017 IEEE International Conference on Cluster Computing (CLUSTER), Honolulu, HI, USA, 5–8 September 2017; pp. 439–450. [CrossRef]
25. Cheramangalath, U.; Nasre, R.; Srikant, Y.N. Falcon: A Domain Specific Language for Graph Analytics. In *Distributed Graph Analytics: Programming, Languages, and Their Compilation*; Springer International Publishing: Cham, Switzerland, 2020; pp. 153–179. [CrossRef]
26. Cheramangalath, U. Falcon: A Graph Manipulation Language for Distributed Heterogeneous Systems. Ph.D. Thesis, Indian Institute of Science, Bengaluru, India, 2017.
27. Oracle. *Graph Developer's Guide for Property Graph*; Oracle: Austin, TX, USA, 2024.
28. Zhang, Y.; Ko, H.S.; Hu, Z. Palgol: A High-Level DSL for Vertex-Centric Graph Processing with Remote Data Access. In *Programming Languages and Systems*; Chang, B.Y.E., Ed.; Springer: Cham, Switzerland, 2017; pp. 301–320.
29. Zhang, Y.; Yang, M.; Baghdadi, R.; Kamil, S.; Shun, J.; Amarasinghe, S. GraphIt: A High-Performance Graph DSL. *Proc. ACM Program. Lang.* **2018**, *2*, 121. [CrossRef]
30. Zhang, Y. GraphIt: Optimizing the Performance and Improving the Programmability of Graph Algorithms. Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2020.
31. Zhang, Y. GraphIt—A High-Performance Domain Specific Language for Graph Analytics, 2024. Available online: https://graphit-lang.org/ (accessed on 3 July 2025).
32. Chamberlin, J.; Zalewski, M.; McMillan, S.; Lumsdaine, A. PyGB: GraphBLAS DSL in Python with Dynamic Compilation Into Efficient C++. In Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Vancouver, BC, Canada, 21–25 May 2018; pp. 310–319. [CrossRef]
33. Peng, Z.; Ashraf, R.A.; Guo, L.; Tian, R.; Kestor, G. Automatic Code Generation for High-Performance Graph Algorithms. In Proceedings of the 2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT), Vienna, Austria, 21–25 October 2023; pp. 14–26. [CrossRef]
34. Emoto, K.; Matsuzaki, K.; Hu, Z.; Morihata, A.; Iwasaki, H. Think like a vertex, behave like a function! A functional DSL for vertex-centric big graph processing. *SIGPLAN Not.* **2016**, *51*, 200–213. [CrossRef]
35. Emoto, K.; Sadahira, F. A DSL for graph parallel programming with vertex subsets. *J. Supercomput.* **2020**, *76*, 4998–5015. [CrossRef]
36. Iwasaki, H.; Emoto, K.; Morihata, A.; Matsuzaki, K.; Hu, Z. Fregel: A functional domain-specific language for vertex-centric large-scale graph processing. *J. Funct. Program.* **2022**, *32*, e4. [CrossRef]
37. Behera, N.; Kumar, A.; Rajadurai T, E.; Nitish, S.; M, R.P.; Nasre, R. StarPlat: A versatile DSL for graph analytics. *J. Parallel Distrib. Comput.* **2024**, *194*, 104967. [CrossRef]
38. Hong, S.; Van Der Lugt, J.; Welc, A.; Raman, R.; Chafi, H. Early experiences in using a domain-specific language for large-scale graph analysis. In Proceedings of the GRADES '13: First International Workshop on Graph Data Management Experiences and Systems, New York, NY, USA, 23 June 2013. [CrossRef]
39. Sevenich, M.; Hong, S.; Welc, A.; Chafi, H. Fast In-Memory Triangle Listing for Large Real-World Graphs. In Proceedings of the SNAKDD'14: 8th Workshop on Social Network Mining and Analysis, New York, NY, USA, 24–27 August 2014. [CrossRef]
40. Sevenich, M.; Hong, S.; van Rest, O.; Wu, Z.; Banerjee, J.; Chafi, H. Using Domain-Specific Languages for Analytic Graph Databases. *Proc. VLDB Endow.* **2016**, *9*, 1257–1268. [CrossRef]

41. Rajendran, A.; Nandivada, V.K. DisGCo: A Compiler for Distributed Graph Analytics. *ACM Trans. Archit. Code Optim.* **2020**, *17*, 28. [CrossRef]
42. Boukham, H.; Wachsmuth, G.; Hartman, T.; Boucherit, H.; van Rest, O.; Chafi, H.; Hong, S.; Dwars, M.; Delamare, A.; Chiadmi, D. Spoofax at Oracle: Domain-Specific Language Engineering for Large-Scale Graph Analytics. In Proceedings of the Eelco Visser Commemorative Symposium (EVCS 2023), Delft, The Netherlands, 6 April 2022; Lämmel, R., Mosses, P.D., Steimann, F., Eds.; Open Access Series in Informatics (OASIcs); Dagstuhl: Wadern, Germany, 2023; Volume 109, pp. 5:1–5:8. [CrossRef]
43. Upadhyay, N.; Patel, P.; Cheramangalath, U.; Srikant, Y.N. Large Scale Graph Processing in a Distributed Environment. In *Euro-Par 2017: Parallel Processing Workshops*; Heras, D.B., Bougé, L., Mencagli, G., Jeannot, E., Sakellariou, R., Badia, R.M., Barbosa, J.G., Ricci, L., Scott, S.L., Lankes, S., et al., Eds.; Springer: Cham, Switzerland, 2018; pp. 465–477.
44. Boukham, H.; Wachsmuth, G.; Dwars, M.; Chiadmi, D. A Multi-target, Multi-paradigm DSL Compiler for Algorithmic Graph Processing. In Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), Auckland, New Zealand, 6–7 December 2022; Association for Computing Machinery: New York, NY, USA, 2022. [CrossRef]
45. Pai, S.; Pingali, K. A Compiler for Throughput Optimization of Graph Algorithms on GPUs. *SIGPLAN Not.* **2016**, *51*, 1–19. [CrossRef]
46. Brahmakshatriya, A.; Furst, E.; Ying, V.A.; Hsu, C.; Hong, C.; Ruttenberg, M.; Zhang, Y.; Jung, D.C.; Richmond, D.; Taylor, M.B.; et al. Taming the Zoo: The Unified GraphIt Compiler Framework for Novel Architectures. In Proceedings of the 48th Annual International Symposium in Computer Architecture (ISCA), Spain (virtual event), 14–19 June 2021.
47. Wendt, N.; Austin, T.; Bertacco, V. PriMax: Maximizing DSL application performance with selective primitive acceleration. In Proceedings of the DAC '22, 59th ACM/IEEE Design Automation Conference, San Francisco, CA, USA, 10–14 June 2022; pp. 139–144. [CrossRef]
48. Brahmakshatriya, A.; Amarasinghe, S. GraphIt to CUDA Compiler in 2021 LOC: A Case for High-Performance DSL Implementation via Staging with BuilDSL. In Proceedings of the 2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Republic of Korea (virtual event), 2–6 April 2022; pp. 53–65. [CrossRef]
49. Nguyen, D.; Lenharth, A.; Pingali, K. A lightweight infrastructure for graph analytics. In Proceedings of the SOSP '13: Twenty-Fourth ACM Symposium on Operating Systems Principles, Farminton, PA, USA, 3–6 November 2013; pp. 456–471. [CrossRef]
50. Green, A.; Guagliardo, P.; Libkin, L. *Property Graphs and Paths in GQL: Mathematical Definitions*; Technical Reports TR-2021-01; Linked Data Benchmark Council (LDBC): London, UK, 2021. [CrossRef]
51. Kepner, J.; Aaltonen, P.; Bader, D.; Buluç, A.; Franchetti, F.; Gilbert, J.; Hutchison, D.; Kumar, M.; Lumsdaine, A.; Meyerhenke, H.; et al. Mathematical foundations of the GraphBLAS. In Proceedings of the 2016 IEEE High Performance Extreme Computing Conference (HPEC), Waltham, MA, USA, 13–15 September 2016; pp. 1–9. [CrossRef]
52. Kalavri, V.; Vlassov, V.; Haridi, S. High-Level Programming Abstractions for Distributed Graph Processing. *IEEE Trans. Knowl. Data Eng.* **2018**, *30*, 305–324. [CrossRef]
53. Malewicz, G.; Austern, M.H.; Bik, A.J.; Dehnert, J.C.; Horn, I.; Leiser, N.; Czajkowski, G. Pregel: A system for large-scale graph processing. In Proceedings of the SIGMOD '10: 2010 ACM SIGMOD International Conference on Management of Data, Indianapolis, IN, USA, 6–10 June 2010; pp. 135–146. [CrossRef]
54. Arenas, M.; Gutierrez, C.; Pérez, J. Foundations of RDF Databases. In *Reasoning Web. Semantic Technologies for Information Systems, Proceedings of the 5th International Summer School 2009, Brixen-Bressanone, Italy, 30 August–4 September 2009*; Tessaris, S., Franconi, E., Eiter, T., Gutierrez, C., Handschuh, S., Rousset, M.C., Schmidt, R.A., Eds.; Tutorial Lectures; Springer: Berlin/Heidelberg, Germany, 2009; pp. 158–204. [CrossRef]
55. Fowler, M. Language Workbenches: The Killer-App for Domain Specific Languages, 2005. Available online: https://martinfowler.com/articles/languageWorkbench.html (accessed on 3 July 2025).
56. Schuts, M. Industrial Experiences in Applying Domain Specific Languages for System Evolution. Ph.D. Thesis, Radboud University, Nijmegen, The Netherlands, 2017.
57. Ratiu, D.; Nehls, H.; Joanni, A.; Rothbauer, S. Use MPS to Unleash the Creativity of Domain Experts: Language Engineering Is a Key Enabler for Bringing Innovation in Industry. In *Domain-Specific Languages in Practice: With JetBrains MPS*; Bucchiarone, A., Cicchetti, A., Ciccozzi, F., Pierantonio, A., Eds.; Springer International Publishing: Cham, Switzerland, 2021; pp. 25–52. [CrossRef]
58. Voelter, M.; Koščejev, S.; Riedel, M.; Deitsch, A.; Hinkelmann, A. A Domain-Specific Language for Payroll Calculations: An Experience Report from DATEV. In *Domain-Specific Languages in Practice: With JetBrains MPS*; Bucchiarone, A., Cicchetti, A., Ciccozzi, F., Pierantonio, A., Eds.; Springer International Publishing: Cham, Switzerland, 2021; pp. 93–130. [CrossRef]
59. Stotz, N.; Birken, K. Migrating Insurance Calculation Rule Descriptions from Word to MPS. In *Domain-Specific Languages in Practice: With JetBrains MPS*; Bucchiarone, A., Cicchetti, A., Ciccozzi, F., Pierantonio, A., Eds.; Springer International Publishing: Cham, Switzerland, 2021; pp. 165–194. [CrossRef]

60. Denkers, J. A longitudinal field study on creation and use of domain-specific languages in industry. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; pp. 1152–1155.
61. Voelter, M.; Kolb, B.; Szabó, T.; Ratiu, D.; Deursen, A. Lessons learned from developing mbeddr: A case study in language engineering with MPS. *Softw. Syst. Model.* **2019**, *18*, 585–630. [CrossRef]
62. Bunte, O.; Denkers, J.; van Gool, L.C.M.; Vinju, J.J.; Visser, E.; Willemse, T.A.C.; Zaidman, A. OIL: An industrial case study in language engineering with Spoofax: OIL: An industrial case study in language engineering with Spoofax. *Softw. Syst. Model.* **2024**, *24*, 139–182. [CrossRef]
63. Campagne, F. *The MPS Language Workbench Volume I*, 1st ed.; CreateSpace Independent Publishing Platform: North Charleston, SC, USA, 2014.
64. Eysholdt, M.; Behrens, H. Xtext: Implement Your Language Faster than the Quick and Dirty Way. In Proceedings of the OOPSLA '10: ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, Reno-Tahoe, NV, USA, 17–21 October 2010; pp. 307–309. [CrossRef]
65. Klint, P.; Van Der Storm, T.; Vinju, J. Rascal, 10 years later. In Proceedings of the 2019 IEEE 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), Cleveland, OH, USA, 30 September–1 October 2019; p. 139.
66. Kats, L.C.; Visser, E. The Spoofax Language Workbench. In Proceedings of the OOPSLA '10: ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, Reno-Tahoe, NV, USA, 17–21 October 2010; pp. 237–238. [CrossRef]
67. Lattner, C.; Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the International Symposium on Code Generation and Optimization, 2004, Palo Alto, CA, USA, 20–24 March 2004; pp. 75–86. [CrossRef]
68. Appel, A.W. SSA is functional programming. *ACM Sigplan Not.* **1998**, *33*, 17–20. [CrossRef]
69. Lattner, C.; Amini, M.; Bondhugula, U.; Cohen, A.; Davis, A.; Pienaar, J.; Riddle, R.; Shpeisman, T.; Vasilache, N.; Zinenko, O. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Seoul, Republic of Korea (virtual event), 27 February–3 March 2021; pp. 2–14. [CrossRef]
70. Wilson, C.; Sala, A.; Puttaswamy, K.P.N.; Zhao, B.Y. Beyond Social Graphs: User Interactions in Online Social Networks and their Implications. *ACM Trans. Web* **2012**, *6*, 17. [CrossRef]
71. Watts, D.J.; Strogatz, S.H. Collective dynamics of 'small-world' networks. *Nature* **1998**, *393*, 440–442. [CrossRef]
72. Bollobás, B.; Borgs, C.; Chayes, J.; Riordan, O. Directed scale-free graphs. In Proceedings of the SODA '03: Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, MD, USA, 12–14 January 2003; pp. 132–139.
73. Choromański, K.; Matuszak, M.; Miekisz, J. Scale-Free Graph with Preferential Attachment and Evolving Internal Vertex Structure. *J. Stat. Phys.* **2013**, *151*, 1175–1183. [CrossRef]
74. Nettleton, D.F. Data mining of social networks represented as graphs. *Comput. Sci. Rev.* **2013**, *7*, 1–34. [CrossRef]
75. Albert, R.; Jeong, H.; Barabási, A.L. Diameter of the World-Wide Web. *Nature* **1999**, *401*, 130–131. [CrossRef]
76. Chakrabarti, D.; Faloutsos, C. The RMat (Recursive MATrix) Graph Generator. In *Graph Mining: Laws, Tools, and Case Studies*; Springer International Publishing: Cham, Switzerland, 2012; pp. 81–86. [CrossRef]
77. Erdös, P.; Rényi, A. On the evolution of random graphs. In *The Structure and Dynamics of Networks*; Princeton University Press: Princeton, NJ, USA, 2006; pp. 38–82. [CrossRef]
78. Guo, Y.; Varbanescu, A.L.; Iosup, A.; Martella, C.; Willke, T.L. Benchmarking graph-processing platforms: A vision. In Proceedings of the ICPE '14: 5th ACM/SPEC International Conference on Performance Engineering, Dublin, Ireland, 22–26 March 2014; pp. 289–292. [CrossRef]
79. Mehrotra, P.; Anand, V.; Margo, D.; Hajidehi, M.R.; Seltzer, M. SoK: The Faults in our Graph Benchmarks. *arXiv* **2024**, arXiv:2404.00766v1.
80. Sridhar, U.; Blanco, M.P.; Mayuranath, R.; Spampinato, D.G.; Low, T.M.; McMillan, S. Delta-Stepping SSSP: From Vertices and Edges to GraphBLAS Implementations. In Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Rio de Janeiro, Brazil, 20–24 May 2019; pp. 241–250. [CrossRef]
81. Hohnka, M.J.; Miller, J.A.; Dacumos, K.M.; Fritton, T.J.; Erdley, J.D.; Long, L.N. Evaluation of Compiler-Induced Vulnerabilities. *J. Aerosp. Inf. Syst.* **2019**, *16*, 409–426. [CrossRef]
82. Li, C.; Wu, Y.; Shen, W.; Zhao, Z.; Chang, R.; Liu, C.; Liu, Y.; Ren, K. Demystifying Compiler Unstable Feature Usage and Impacts in the Rust Ecosystem. In Proceedings of the ICSE '24: IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024. [CrossRef]
83. Subagdja, B.; Shanthoshigaa, D.; Wang, Z.; Tan, A.H. Machine Learning for Refining Knowledge Graphs: A Survey. *ACM Comput. Surv.* **2024**, *56*, 156. [CrossRef]
84. Tian, Y. The World of Graph Databases from An Industry Perspective. *SIGMOD Rec.* **2023**, *51*, 60–67. . [CrossRef]

85. Francis, N.; Gheerbrant, A.; Guagliardo, P.; Libkin, L.; Marsault, V.; Martens, W.; Murlak, F.; Peterfreund, L.; Rogova, A.; Vrgoč, D. A Researcher's Digest of GQL. In Proceedings of the 26th International Conference on Database Theory (ICDT 2023), Ioannina, Greece, 28–31 March 2023; Geerts, F., Vandevoort, B., Eds.; Leibniz International Proceedings in Informatics (LIPIcs); Dagstuhl: Wadern, Germany, 2023; Volume 255, pp. 1:1–1:22. [CrossRef]
86. Gray, J.; Fisher, K.; Consel, C.; Karsai, G.; Mernik, M.; Tolvanen, J.P. DSLs: The good, the bad, and the ugly. In Proceedings of the OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, Nashville, TN, USA, 19–23 October 2008; pp. 791–794. [CrossRef]
87. Bucchiarone, A.; Cicchetti, A.; Ciccozzi, F.; Pierantonio, A. *Domain-Specific Languages in Practice: With JetBrains MPS*; Springer Nature: Berlin/Heidelberg, Germany, 2021.
88. Wachsmuth, G.H.; Konat, G.D.; Visser, E. Language Design with the Spoofax Language Workbench. *IEEE Softw.* **2014**, *31*, 35–43. [CrossRef]
89. Angles, R.; Bonifati, A.; Dumbrava, S.; Fletcher, G.; Green, A.; Hidders, J.; Li, B.; Libkin, L.; Marsault, V.; Martens, W.; et al. PG-Schema: Schemas for Property Graphs. *Proc. ACM Manag. Data* **2023**, *1*, 198. [CrossRef]
90. *ISO/IEC 39075:2024*; Information Technology—Database Languages SQL—Part 16: Property Graph Queries (SQL/PGQ). ISO: Geneva, Switzerland, 2024.
91. *ISO/IEC 9075-16*; Information Technology—Database Languages—GQL. ISO: Geneva, Switzerland, 2023.
92. Sakr, S.; Bonifati, A.; Voigt, H.; Iosup, A.; Ammar, K.; Angles, R.; Aref, W.; Arenas, M.; Besta, M.; Boncz, P.A.; et al. The future is big graphs: A community view on graph processing systems. *Commun. ACM* **2021**, *64*, 62–71. [CrossRef]
93. Bonifati, A.; Özsu, M.T.; Tian, Y.; Voigt, H.; Yu, W.; Zhang, W. The Future of Graph Analytics. In Proceedings of the SIGMOD/PODS '24: Companion of the 2024 International Conference on Management of Data, Santiago, Chile, 9–15 June 2024; pp. 544–545. [CrossRef]
94. Bonifati, A.; Ozsu, M.T.; Tian, Y.; Voigt, H.; Yu, W.; Zhang, E. A Roadmap to Graph Analytics. *SIGMOD Rec.* **2025**, *53*, 43–51. [CrossRef]
95. Ju, W.; Fang, Z.; Gu, Y.; Liu, Z.; Long, Q.; Qiao, Z.; Qin, Y.; Shen, J.; Sun, F.; Xiao, Z.; et al. A Comprehensive Survey on Deep Graph Representation Learning. *Neural Netw.* **2024**, *173*, 106207. [CrossRef]
96. Pan, S.; Luo, L.; Wang, Y.; Chen, C.; Wang, J.; Wu, X. Unifying Large Language Models and Knowledge Graphs: A Roadmap. *IEEE Trans. Knowl. Data Eng.* **2024**, *36*, 3580–3599. [CrossRef]
97. Leather, H.; Cummins, C. Machine Learning in Compilers: Past, Present and Future. In Proceedings of the 2020 Forum for Specification and Design Languages (FDL), Kiel, Germany, 15–17 September 2020; pp. 1–8. [CrossRef]
98. Seeker, V.; Cummins, C.; Cole, M.; Franke, B.; Hazelwood, K.; Leather, H. Revealing Compiler Heuristics Through Automated Discovery and Optimization. In Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Edinburgh, Scotland, UK, 2–6 March 2024; pp. 55–66. [CrossRef]
99. Cummins, C.; Seeker, V.; Grubisic, D.; Roziere, B.; Gehring, J.; Synnaeve, G.; Leather, H. LLM Compiler: Foundation Language Models for Compiler Optimization. In Proceedings of the CC '25: 34th ACM SIGPLAN International Conference on Compiler Construction, Las Vegas, NV, USA, 1–2 March 2025; pp. 141–153. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.