Algorithm Support in a Graph Database, Done Right

Daan de Graaf
Supervised by dr. N. Yakovets
Eindhoven University of Technology
Eindhoven, The Netherlands
d.j.a.d.graaf@tue.nl

ABSTRACT

Graph databases are increasingly used for analytics tasks that cannot be served with traditional query languages. Existing solutions leave much to be desired in terms of expressivity, performance, and ease of use. During my PhD, I am creating GraphAlg: A language for graph algorithms that can be integrated into databases. Thanks to a strong theoretical foundation, GraphAlg is highly amenable to analysis and optimization. Based on the linear algebra model of computation, it has a straightforward transformation into relational algebra. So far I have designed the syntax of the language and developed an operational semantics based on MATLANG, a formal language for matrix manipulation. I have also developed a compiler for GraphAlg, and integrated it into AvantGraph, a state-of-the-art graph query engine. In the future, I intend to research advanced optimization techniques, bring GraphAlg support to other database systems, and formally prove the expressivity of the language.

VLDB Workshop Reference Format:
Daan de Graaf. Algorithm Support in a Graph Database, Done Right. VLDB 2025 Workshop: PhD Workshop.

1 INTRODUCTION

Graph databases have garnered a large user base in recent years, owing in part to their convenient interface for graph pattern matching through query languages such as Cypher [7] and, more recent ISO standard, GQL [11]. Some users also develop a need for more advanced graph analyses. For example, to find important nodes in the graph, the PageRank algorithm is often used [13]. Similarly, to identify communities or clusters of closely related entities within a social network, algorithms like the Louvain method are frequently employed [2]. Responding to this demand, some database vendors now provide functionality for running graph algorithms, ranging from built-in library functions for common algorithms to plugins and additional query syntax. Integrating algorithm support into databases avoids the *data wrangling* problem: Exporting data, converting it to a common format, and then importing it into another system. With integrated algorithm support, there is no need to duplicate potentially massive graphs, nor is there any risk of those copies becoming outdated. Furthermore, databases are well-suited for optimizing and efficiently executing these algorithms, thanks to their existing query processing infrastructure.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

I find that of the few systems with dedicated algorithm support, their approaches have major shortcomings. For users, this means they still need to rely on external tools in many cases. An overview of common approaches is given in Table 1. I briefly discuss the key problems below.

Key Problem: Fixed set of algorithms. Systems with a built-in algorithms library are only useful as long as the library provides the specific algorithm that the user needs. If the vendor does not provide the necessary algorithm or provides a different variant of it, the user will need to resort to external tools. To take the Neo4J Graph Data Science Library [17] as an example: Its PageRank implementation has more than 10 parameters to select different variations of the algorithm, yet it does not allow redistribution of scores from sinks, which is trivial to do in a custom implementation.

Key Problem: Performance issues. The Pregel [15] and recursive CTE approaches were slow to execute and had high memory consumption in my experiments. Individual SQL statements in procedural SQL generally show good performance, but the procedural interpreter adds significant overhead. Given a sufficiently large dataset, exporting and processing it externally is faster.

Key Problem: Lack of optimization. The optimal execution strategy for algorithms depends on the state of the graph, as is the case for queries. However, most systems treat algorithms as black boxes, especially when it comes to query optimization. Even in procedural SQL implementations, only individual SQL statements are optimized, without regard for the larger algorithm.

Key Problem: Difficult to write. Recursive common table expressions (CTEs) add recursion to SQL, making it powerful enough to express graph algorithms. Unfortunately, the unusual semantics of recursive CTEs makes them notoriously difficult to write [6]. In practice, they are typically avoided in favor of external tools that are easier to program [14].

My Proposal: Opening up the DSL approach

I argue that the best approach to algorithm support in databases is an integrated domain-specific language (DSL). DSLs offer solutions to all the key problems mentioned before.

- The user can create arbitrary algorithms by composing high-level primitives.
- The language can be designed to include efficient data-parallel operations.
- If care is taken to constrain the expressive power of the language, algorithms can be optimized similar to queries.
- By providing operations tailored to graph algorithms, a DSL facilitates intuitive and concise implementations.

Table 1: An overview of existing approaches to graph analytics support in database systems. Algorithms library, Pregel API and algorithm DSL are solutions tailored to the problem of graph analytics. The other options are more generic tools available in relational database systems. The listed systems are a representative subset of popular database systems rather than an exhaustive list.

Approach	Key Problems	Available in
Algorithms Library	Fixed set of Algorithms	Neo4J, ArangoDB
Pregel API	Performance issues, no optimization	Neo4J, ArangoDB
User-defined operators [18]	No optimization	UMBRA
Recursive CTE	Difficult to write, Performance issues	DuckDB, PostgreSQL
Procedural SQL	Performance issues, limited optimization	PostgreSQL, Oracle, SQL Server
Algorithm DSL	Only proprietary implementations	TigerGraph, Oracle PGX

The DSL approach has received some interest from database vendors, most notably TigerGraph [5] and Oracle PGX [3]. Both systems are unfortunately unavailable for evaluation, and publicly available documentation on their implementation is scarce.

2 GRAPHALG

My vision for algorithm support in databases is based on the following principles:

- **Expressive:** The database should allow users to define custom algorithms in a language that can express a wide variety of graph algorithms.
- **User-friendly:** That language should offer an intuitive interface for defining these algorithms, with clear operations whose semantics are easily understood.
- **Fully Integrated:** Algorithms should be first-class citizens and be deeply integrated into the database.
- **Optimizable:** Algorithms should be aggressively optimized, leveraging the existing query optimization infrastructure where possible.

GraphAlg is the realization of this vision. It is a domain-specific language designed for writing graph algorithms that satisfies all the key principles above. The language has a strong theoretical foundation, building upon the formal MATLANG [4] language for matrix manipulation. By defining GraphAlg in terms of MATLANG, I obtain a small core language that is highly amenable to analysis and optimization. I also adopt the linear algebra programming model from MATLANG. Linear algebra operations such as matrix multiplication are widely taught and have well-understood semantics, making many of the constructs in GraphAlg immediately familiar to new users.

GraphAlg is fully integrated into the AvantGraph [19] query engine developed by our team at TU Eindhoven. GraphAlg programs can be embedded directly into regular Cypher queries. In AvantGraph, I use a unified intermediate representation (IR) that encodes both queries and algorithms. After an initial conversion, there is no distinction between query and algorithm. This has a number of important benefits.

- All optimization rules for queries are automatically applied to algorithms, too.
- There is no communication overhead between queries and algorithms. Passing large intermediates as algorithm parameters does not add an additional cost.
- No need for a separate algorithm execution backend. Instead, the query execution pipeline also handles algorithm execution.

3 CONTRIBUTIONS SO FAR

My contributions so far can be split into three parts:

(1) A practical version of the MATLANG language that can be implemented in a real database system.
(2) A compiler to optimize and transform this language into an extended relation algebra supported by AvantGraph with minor extensions to the query processing pipeline.
(3) Query optimization strategies for iterative algorithms.

In the following, I describe each contribution in more detail.

3.1 Practical MATLANG

Extensions to MATLANG. Based on my examination of common graph algorithms from the GAP [1] and LDBC Graphalytics [9] benchmark suites, I have identified two necessary language primitives that are not included in the original definition of MATLANG. The first is a loop construct with support for multiple iteration variables. Although loop constructs for MATLANG have previously been proposed [8], they are not a good fit for the iterative algorithms I have studied, such as Single-Source Shortest Paths, PageRank, Weakly Connected Components, Breadth-First Search and Label Propagation. Existing loop definitions are, for the purpose of encoding graph algorithms, too limited in one way and too powerful in another. My novel loop construct addresses this by allowing multiple iteration variables to be carried over between iterations but otherwise being very minimal and restrictive. For example, it is explicitly *not* Turing-complete and does not guarantee an observable order between iterations. The design strikes a careful balance between expressivity and simplicity: Powerful enough to encode a wide variety of algorithms, yet restrictive enough to allow for aggressive optimization.

Secondly, certain algorithms require a *leader election* mechanism to select a single representative from a set of vertices. This operation is common in clustering algorithms like *Weakly Connected Components*, but it is not representable in standard MATLANG.

The solution is simple: extend MATLANG with a leader-election operation.

GraphAlg Core. Based on MATLANG with the proposed extensions, I define a minimal language *GraphAlg Core* that is powerful enough to express all algorithms in the GAP and LDBC graphalytics benchmark suites. I provide a complete operational semantics of the operations. Where MATLANG leaves open certain details of the language that are not relevant for theoretical analysis, in GraphAlg Core, these are fully defined so that it is also executable.

GraphAlg. I define the syntax of the *GraphAlg* language for graph algorithms. It supports all operations in GraphAlg Core, plus higher-level operations that allow users to define custom algorithms in an intuitive and concise way. Unlike MATLANG, programs are written in an imperative style similar to Python. GraphAlg retains the linear algebra model of computation. An example program is given in Figure 1.

```
func WCC(graph: Matrix<s,s,bool>) {
    id = Vector<bool>(graph.nrows);
    id[:] = bool(true);
    label = diag(id);
    for i in graph.nrows {
        alternatives = label;
        alternatives += graph * label;
        label = pickAny(alternatives);
    }
    return label;
}
```

Figure 1: Weakly Connected Components Algorithm written in GraphAlg.

3.2 GraphAlg Compiler

I have developed a compiler for GraphAlg programs to parse, optimize, and transform GraphAlg programs into an extended relational algebra supported by AvantGraph. An overview of the compiler pipeline is given in Figure 2. The compiler leverages GraphAlg Core by first simplifying programs to the core language, and only then performing optimizations or transformations.

I have integrated the GraphAlg compiler into AvantGraph. Users can embed GraphAlg programs in Cypher queries, and AvantGraph executes them as part of the query. I use a unified representation for both queries and algorithms in AvantGraph, so the query optimizer has full visibility into algorithms and can optimize them just like regular queries.

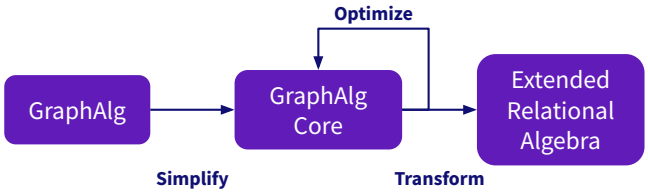

Figure 2: High-level overview of the GraphAlg compiler pipeline.

3.3 Query Optimizations for Algorithms

Apart from adding loop support in query plans, few changes to AvantGraph were required to facilitate algorithm execution. However, to obtain the best possible performance, I have found that it is beneficial to also include additional query optimization rules. In particular, I find that aggregation is common in graph algorithms and, therefore, worth paying special attention to.

Loop-Invariant Code Motion. Moves computation outside the loop if it is independent of all loop iteration variables. My optimization moves query plan subtrees at pipeline breakers (often aggregations). This avoids additional caching of intermediate results (the pipeline breaker is doing the caching anyway) and prevents pulling out very cheap pipelines.

Loop Accumulation. It is common for algorithms to populate a result matrix over multiple iterations, writing additional entries each run. Rather than building a new intermediate result from scratch at every iteration, it is more efficient to preserve the aggregation state across iterations and accumulate the new entries into it. I have developed an optimization rule to detect this case automatically, without requiring annotations in the source program.

Bounded Key Range Aggregation. In query plans generated from GraphAlg programs, aggregations by vertex are often among the most expensive operations. Vertex identifiers are integers assigned and managed internally in AvantGraph, so the range of possible identifiers is known at query optimization time. I define a specialized table data structure to exploit these properties, outperforming standard hash tables, particularly in scenarios where the majority of vertices have an associated aggregate value.

4 RESEARCH PLAN

I have identified three opportunities for future research building upon my original concept of GraphAlg.

4.1 Cross-Optimization

The unified representation for queries and algorithms allows query optimization across the boundary between query and algorithm. My current implementation already does this for simple cases such as predicate pushdown or join ordering. There are more complex cases, particularly those that involve loops, that the current system does not optimize yet. An example is the imposing of a filter on the output of a shortest path algorithm, as shown in Figure 3.

The naive way of executing this query would be to first compute all shortest paths and then apply the filter to remove paths with a high cost. A more efficient strategy is to instead push the filter into the algorithm, and skip exploring paths with a high cost entirely. If many shortest paths exceed the maximum cost, this optimization will be highly beneficial.

```
WITH ALGORITHM SSSP (...)
CALL SSSP(..)
YIELD source, target, cost
WHERE cost < 100
RETURN source, target, cost;
```

Figure 3: Single-Source Shortest Path algorithm with a maximum distance filter.

4.2 GraphAlg support in other DBMS

While the current version of GraphAlg is tightly integrated with AvantGraph, I expect many potential users prefer to stick with an existing DBMS they know well. With graph query support included in the latest SQL standard [10], this system may even be relational rather than graph-native. Since GraphAlg compiles to relational algebra, I believe it is feasible to port GraphAlg to many other DBMS, and bring algorithm support to them this way.

4.3 Graph algorithm primitives

While I have empirically verified that GraphAlg can express a variety of algorithms, there is no formal proof of any particular class of algorithms that it can encode. It is possible that there exist less common but important algorithms that cannot be written in GraphAlg, or that they can only be expressed in a way that is inefficient to compute.

Thus, the challenge remains to formalize a *minimal* set of graph operations that are *provably* sufficient to express a meaningful class of algorithms. This class should be a set of algorithms that can be seen as 'realistic' in terms of time and space complexity for running on large-scale graphs. If such a set of operations can indeed be formalized, I envision it would form an ideal basis for user-defined algorithm support in many systems, even beyond databases.

5 APPLICATION

GraphAlg is developed in the context of the SciLake[1] project. As part of this project, AvantGraph will host the OpenAIRE graph [16], a large scientific knowledge graph containing hundreds of millions of publications. The OpenAIRE graph currently relies on the BIP! Ranker [12] tool to enrich the publication data with research impact indicators based on the citation graph. Impact indicators are computed based on different algorithms, typically derivatives of PageRank or simple citation counts.

With the graph already available in AvantGraph, GraphAlg is ideally suited to compute these indicators, replacing a complex pipeline running on a large cluster with a simpler and more efficient query with an embedded algorithm. Project partners are also able to experiment with their own custom algorithms using GraphAlg.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon Europe framework programme under grant agreement No. 101058573 as part of the SciLake project.

[1] https://scilake.eu

REFERENCES

[1] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. https://doi.org/10.48550/arXiv.1508.03619 arXiv:1508.03619 [cs].

[2] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (Oct. 2008), P10008. https://doi.org/10.1088/1742-5468/2008/10/P10008

[3] Houda Boukham, Guido Wachsmuth, Toine Hartman, Hamza Boucherit, Oskar van Rest, Hassan Chafi, Sungpack Hong, Martijn Dwars, Arnaud Delamare, and Dalila Chiadmi. 2023. Spoofax at Oracle: Domain-Specific Language Engineering for Large-Scale Graph Analytics. In *Eelco Visser Commemorative Symposium (EVCS 2023) (Open Access Series in Informatics (OASIcs))*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.), Vol. 109. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:8. https://doi.org/10.4230/OASIcs.EVCS.2023.5 ISSN: 2190-6807.

[4] Robert Brijder, Floris Geerts, Jan Van Den Bussche, and Timmy Weerwag. 2019. On the Expressive Power of Query Languages for Matrices. *ACM Trans. Database Syst.* 44, 4 (Oct. 2019), 15:1–15:31. https://doi.org/10.1145/3331445

[5] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. 2019. TigerGraph: A Native MPP Graph Database. https://doi.org/10.48550/arXiv.1901.08248 arXiv:1901.08248 [cs].

[6] Christian Duta. 2022. Another way to implement complex computations: functional-style SQL UDF. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. ACM, Philadelphia Pennsylvania, 1–7. https://doi.org/10.1145/3546930.3547508

[7] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1433–1445. https://doi.org/10.1145/3183713.3190657

[8] Floris Geerts, Thomas Muñoz, Cristian Riveros, and Domagoj Vrgoč. 2021. Expressive Power of Linear Algebra Query Languages. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'21)*. Association for Computing Machinery, New York, NY, USA, 342–354. https://doi.org/10.1145/3452021.3458314

[9] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC graphalytics: a benchmark for large-scale graph analysis on parallel and distributed platforms. *Proceedings of the VLDB Endowment* 9, 13 (Sept. 2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[10] ISO. 2023. Information technology — Database languages SQL — Part 16: Property Graph Queries (SQL/PGQ). https://www.iso.org/standard/79473.html

[11] ISO. 2024. Information technology — Database languages — GQL. https://www.iso.org/standard/76120.html

[12] Ilias Kanellos, Thanasis Vergoulis, Claudio Atzori, Andrea Mannocci, Serafeim Chatzopoulos, Sandro La Bruzzo, Natalia Manola, and Paolo Manghi. 2024. BIP! Ranker. https://doi.org/10.5281/zenodo.10564110

[13] Ilias Kanellos, Thanasis Vergoulis, Dimitris Sacharidis, Theodore Dalamagas, and Yannis Vassiliou. 2021. Impact-Based Ranking of Scientific Publications: A Survey and Experimental Evaluation. *IEEE Transactions on Knowledge and Data Engineering* 33, 4 (April 2021), 1567–1584. https://doi.org/10.1109/TKDE.2019.2941206

[14] Louisa Lambrecht, Torsten Grust, Altan Birler, and Thomas Neumann. 2025. Trampoline-Style Queries for SQL. *Proceedings of CIDR* (2025).

[15] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[16] Paolo Manghi, Alessia Bardi, Claudio Atzori, Miriam Baglioni, Natalia Manola, Jochen Schirrwagen, and Pedro Principe. 2019. *The OpenAIRE Research Graph Data Model*. Technical Report. Zenodo. https://doi.org/10.5281/zenodo.2643199

[17] Neo4J inc. [n.d.]. Graph algorithms - Neo4j Graph Data Science. https://neo4j.com/docs/graph-data-science/2.17/algorithms/

[18] Moritz Sichert and Thomas Neumann. 2022. User-defined operators: efficiently integrating custom algorithms into modern databases. *Proceedings of the VLDB Endowment* 15, 5 (Jan. 2022), 1119–1131. https://doi.org/10.14778/3510397.3510408

[19] Wilco van Leeuwen, Thomas Mulder, Bram van de Wall, George Fletcher, and Nikolay Yakovets. 2022. AvantGraph query processing engine. *Proceedings of the VLDB Endowment* 15, 12 (Aug. 2022), 3698–3701. https://doi.org/10.14778/3554821.3554878