图数据库的算法支持，正确实现

Daan de Graaf 埃因霍温理工大学 荷兰埃因霍温
d.j.a.d.graaf@tue.nl

Robert Brijder 埃因霍温理工大学 荷兰埃因霍温
r.brijder@tue.nl

Soham Chakraborty 代尔夫特理工大学 荷兰代尔夫特
s.s.chakraborty@tudelft.nl

George Fletcher 埃因霍温理工大学 荷兰埃因霍温
g.h.l.fletcher@tue.nl

Bram van de Wall 埃因霍温理工大学 荷兰埃因霍温
a.a.g.v.d.wall@tue.nl

Nikolay Yakovets 埃因霍温理工大学 荷兰埃因霍温
hush@tue.nl

arXiv:2601.06705v1 [cs.DB] 10 Jan 2026

摘要
图数据库查询语言无法表达 PageRank 等算法，导致数据整理代价高昂；而现有解决方案，如算法库、以顶点为中心的 API 和递归 CTE，缺少表达能力、性能与易用性三者的必要结合。我们提出 GraphAlg：一种用于图算法的领域专用语言，可编译为关系代数，从而与查询处理流水线无缝集成。GraphAlg 基于线性代数，为矩阵运算提供直观的表达方式，并支持包括稀疏性分析、循环不变代码外提和原地聚合在内的激进优化。我们在 AvantGraph 中的实现表明，与 SQL/Python 和 Pregel 相比，GraphAlg 能显著降低代码复杂度，同时在 LDBC Graphalytics 基准测试中实现出色的性能。GraphAlg 证明，图数据库可以成为同时支持查询与分析的统一平台。

制品
GraphAlg 编译器的源代码可从 github.com/wildarch/graphalg 免费获取。

更多文档请访问 wildarch.dev/graphalg。内容包括一份关于如何使用 GraphAlg 编写程序的交互式教程，以及编译器内部机制的文档。

1 引言

设想一位数据科学家使用 PageRank 分析引文网络，以识别具有影响力的论文；或者一位欺诈分析师在交易图上运行社区检测算法。尽管图数据库是存储此类数据的天然场所，但这些用户却无法使用 Cypher [16] 或 GQL [23],表达这些基础算法——这两种语言本来正是为图数据设计的！相反，他们必须导出数 GB 的数据，费力处理格式转换，并维护很快就会过时的重复副本。图存储与图分析之间的脱节，代表了当前数据库系统中的一个重大缺口。

尽管一些数据库尝试提供了解决方案，例如算法库（ArangoDB）、Pregel API（Neo4J）或递归 CTE（PostgreSQL），但每种方法都存在明显局限（见表 1）。算法库提供了一组固定的算法，很少能完全满足具体需求。以顶点为中心的 API 由于消息传递过于频繁而性能不佳。递归 CTE 以难以编写和优化而闻名。因此，用户仍不得不依赖外部工具，从而抵消了集成数据管理本应带来的优势。

本文介绍 GraphAlg，这是一种领域专用语言，能将图算法作为一等公民引入数据库。我们的关键洞见是，线性代数提供了理想的抽象层：矩阵运算能够自然地表达图算法、编译为关系代数（含聚合），并支持激进的优化。与以往的尝试不同，GraphAlg 满足四项关键要求：**(1) 表达力强**：用户可以通过组合矩阵运算实现任意算法；**(2) 易于使用**：与 SQL 或 Pregel 等替代方案相比，实现算法所需的代码显著更少；**(3)完全集成**：算法可直接嵌入查询，无需移动数据；**(4) 可优化**：程序可编译为查询计划，并利用现有的数据库优化。

我们的主要贡献有三项：

语言设计。我们开发了 GraphAlg，这是一种基于线性代数、具有 MATLANG [6] 形式化基础的语言。GraphAlg 提供高级矩阵运算，这些运算可以自然组合，并可编译为高效的（扩展）关系代数（第 3 节）。

数据库集成。我们在 Avant- Graph [36] 查询引擎中实现了 GraphAlg，使算法和查询共享统一表示。这使得跨边界优化成为可能，而在采用独立算法流水线的系统中则无法实现（第 4 节）。

新颖的优化。我们为 GraphAlg 程序的执行引入了关键优化：稀疏性分析维护紧凑表示，循环不变代码外提消除冗余计算，原地聚合则避免物化中间结果（第 5 节）。

我们的评估表明，与现有方法相比，GraphAlg 在标准基准测试中实现卓越性能的同时，显著降低了代码复杂度。在 LDBC Graphalytics [21],上，GraphAlg/AvantGraph 在 PageRank、单源最短路径和弱连通分量算法上的表现优于 DuckDB 和 Neo4j（第 6 节）。统一的查询-算法表示还支持强大的跨边界优化，使预处理步骤能够

表 1：现有图分析支持数据库系统方法概览。算法库、Pregel API 和算法 DSL 是针对图分析问题设计的解决方案。其他选项则是关系型数据库系统中可用的更通用工具。所列系统代表了部分流行的数据库系统，而非完整清单。

方法	关键问题	适用于
算法库	固定的算法集合	Neo4j、ArangoDB
Pregel API	性能问题，未进行优化	Neo4j
用户定义算子	不安全，未进行优化	Umbra
递归 CTE	难以编写，性能问题	DuckDB、Umbra、PostgreSQL、Oracle、SQL Server
过程式 SQL	开销大，优化有限	PostgreSQL、Oracle、SQL Server
算法 DSL	仅有专有实现	TigerGraph、Oracle PGX

被纳入算法执行，而无需修改算法本身。

借助 GraphAlg，图数据库终于获得了恰到好处的算法支持：兼具用户所需的表达能力、他们期待的性能，以及他们要求的无缝集成。图算法不再是被迫置身数据库之外的二等公民，而是回归其应有的位置，成为作用于图数据的一等操作。

2 审视设计空间

在详细介绍 GraphAlg 的设计之前，我们先探讨现有方法为何会失效，以及哪些设计选择使我们的解决方案成为可能。我们将回顾当前的数据库解决方案（第 2.1 节），分析图算法的计算模型（第 2.2 节），并说明 GraphAlg 如何在这一设计空间中进行探索（第 2.3 节）。

2.1 当前数据管理系统的局限

现有数据库中图分析的可选方案取决于所使用的具体系统。表 1 概述了现有方法及其局限。下面将详细介绍每种方法。

算法库。Neo4j [32]和 ArangoDB [2] 将常用图算法库集成到各自的系统中。这类算法库无疑最易于使用，通常也能提供良好的性能。其主要缺点是，只有在库中包含所需的特定算法时才能使用。以 Neo4J 图数据科学库 [32] 为例：其 PageRank 实现提供了十多个参数，可用于选择算法的不同变体，却不允许从汇点重新分配得分，而这一操作在自定义实现中轻而易举。

Pregel API。为了满足算法库无法满足的用户需求，Neo4j [32] 还提供了 Java API，以便编写受 Pregel [29]启发、采用顶点中心式的算法。Pregel API 灵活得多，但正如我们在第 6 节所展示的，这也带来了性能问题。看来这是以顶点为中心的模型 [26, 31] 本身存在的根本问题，而不只是我们进行基准测试的 Neo4j 实现所特有的问题。此外，由于 Pregel 实现的主体可以包含任意 Java 代码（包括外部库），要对其实现应用高级优化极具挑战。据我们所知，Neo4j 并未尝试这样做。

用户定义算子。 Umbra 数据库系统支持以 C++ [35] 编写的用户定义算子。它的性能出色，但由于缺乏隔离，存在缺陷的实现可能会导致系统崩溃，或在不知不觉中破坏任意内存位置的数据。由于用户可以提供任意 C++ 代码，因此无法进行高层优化。

递归 CTE。 递归公用表表达式（CTE）为 SQL 引入了递归，从而极大地增强了其表达能力。由于关系型数据库系统广泛支持递归 CTE，因此它们显然是表达图算法的理想选择。遗憾的是，递归 CTE 众所周知很难编写 [13]。在实践中，人们通常会避开递归 CTE，转而使用更易于编程的外部工具 [27]。在搭建基准测试环境时（见第 6 节），我们尝试在 DuckDB 中使用递归 CTE。我们发现，用这种形式实现算法颇具挑战性，而且性能不佳。在所有情况下，分别发出多个独立查询，并将其结果缓冲到临时表中，性能都更好。

近期的工作为 SQL 扩展了额外语法，以便在递归 CTE 中表达复杂控制流 [27] 和 upset 语义 [19] 。在未来的工作中，我们希望探索带有这些扩展语法的递归 CTE 是否适合作为 GraphAlg 的编译目标。这样一来，只要数据库实现了必要的 SQL 扩展，即使没有明确支持，也能在其上运行 GraphAlg 程序。

SQL 的过程式扩展。包括 PostgreSQL 在内的多个关系型数据库系统都提供了支持嵌入式 SQL 查询的过程式语言。这带来了很大的灵活性，理论上也能支持高级分析和优化。遗憾的是，实现似乎往往更注重功能而非性能。pl/pgSQL 之类的语言通常由简单解释器执行，而嵌入式 SQL 查询则单独进行优化和执行 [18]。甚至有研究建议将 pl/pgSQL 编译为纯 SQL，以避免解释器和上下文切换带来的开销 [14]。

算法 DSL。 领域专用语言（DSL）为数据库中的算法支持提供了一种极具吸引力的方法。领域专用语言为常见的图操作提供专用语法，使算法易于表达。用户可以编写任意算法，但这可能会受到特定领域专用语言的表达能力限制

领域专用语言。同样，性能和优化机会取决于语言的设计以及执行策略。据我们所知，有两个系统采用了这种方法：TigerGraph [11] 和 Oracle PGX [5]。遗憾的是，这两个系统都是专有系统：TigerGraph 最近才推出其系统的免费社区版，而 Oracle 则完全不提供免费版本。这些系统的源代码不可用，并且两者的许可协议都明确禁止运行基准测试。因此，关于这些系统内部机制的公开文档非常有限。TigerGraph 支持 GSQL 语言，这是一种支持迭代和累加器的图查询语言 [12]。由于算法可以用 GSQL 表达，我们假设它们与常规查询一样由同一个查询引擎执行。不过，根据文档[1], , 其查询优化似乎受到限制。至于 Oracle PGX，据悉其算法流水线与查询引擎是分离的，因此二者采用了不同于查询的优化方式。

2.2 图算法语言

文献中提出了多种用于编写图算法的语言。根据其计算模型，这些语言大致可分为以下四类，我们将在下文中逐一讨论。

通用 语言（如 Python 或 C++ ）可以说是当今最流行的方法。这类语言的明显优势是拥有庞大的用户群体，用户熟悉其语法。它们通常还提供许多库，以简化常见的图分析任务。但由于通用性极强，这类语言的表达能力很强，也极难分析和优化。我们的目标是将 GraphAlg 编译为高效的查询计划，而这种转换仅适用于极少数有效程序。

以顶点为中心。 Pregel [29], 开创了以顶点为中心的模型。该模型的限制更多，尤其体现在内存访问方式上。Fregel [24] 是基于 TLAV（“像顶点一样思考”）范式的语言之一。计算以单个顶点为单位进行，顶点之间的所有通信都通过消息传递完成。这使其非常适合在分布式环境中使用，但在单节点系统上，这种限制可能会对性能产生负面影响。我们还观察到，以顶点为中心的处理与

关系代数：在以顶点为中心的模型中，每个顶点独立行动；而关系代数则对整个关系应用相同的操作。因此，将以顶点为中心的程序转换为关系代数程序更加困难，也阻碍了数据并行处理的普遍应用。

线性代数是图分析领域一个成熟的模型，拥有丰富的相关文献 [25]。线性代数运算具有许多理想的特性。首先，线性代数运算具有高层次和可组合的特点。由于这些运算定义在整个矩阵之上，因此可以发掘查询优化机会；尤其是，执行过程很容易实现并行化。各种线性代数运算都可以直接转换为关系代数 [7]。最后，许多图分析任务都可以表述为线性代数计算。例如，图1说明了矩阵乘法与图遍历密切相关。基于这些原因，我们采用

[1] https://docs.tigergraph.com/gsql-ref/4.2/querying/query-optimizer/

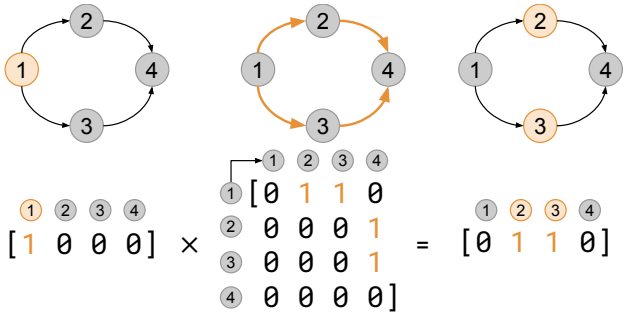

图1：矩阵乘法与图遍历之间的联系。

线性代数模型来实现 GraphAlg。MATLANG [6]这一用于矩阵操作的形式化语言此前研究过线性代数模型，但它纯粹是一种没有实现的形式语言。我们将在第3.3节讨论 GraphAlg 与 MATLANG 之间的关系。在实现方面，SuiteSparse:GraphBLAS [9],是一套稀疏矩阵算法，这些算法可以组合起来实现图算法 [30]。然而，SuiteS-parse:GraphBLAS 围绕自身的内部数据表示和运行时构建，因此与其他外部工具一样，需要经过数据整理步骤。它还要求将所有数据加载到主存中，因此无法扩展到大于主存容量的图。

顶点集和边集。 该模型与标准命令式模型相似，但要求所有图操作都以批量方式执行。这在一定程度上限制了表达能力，却使高级分析和优化成为可能。例如，Greenmarl [20] 和 GraphIt [38]。我们认为，基于顶点集和边集模型的语言颇具价值，因为它们在广泛的算法和工作负载上都表现出色 [3]。它们也能够得到有效的分析和优化。在许多方面，顶点集和边集模型类似于线性代数模型：顶点集类似于（稀疏）向量，边集则类似于矩阵。遵循顶点集和边集模型的语言可以实现线性代数运算。然而，反向关系通常并不成立。以 GraphIt [38] 为例：它支持创建数组，并在并行循环中向数组的各个位置写入数据。虽然可变数组在共享内存系统中效率很高，但它们并不适用于关系代数。

2.3 我们的方法

我们认为，为数据库提供算法支持的正确方法是采用集成的领域专用语言（DSL）。GraphAlg 基于线性代数计算模型，体现了我们关于如何正确实现算法支持的四项原则：

表达能力强且可优化。 GraphAlg 在表达能力与可优化性之间取得了审慎的平衡。用户可以通过组合高级线性代数原语来编码任意算法。同时，GraphAlg 避免使用难以优化或不具备数据并行性的操作，例如显式修改。线性代数运算处理完整矩阵，因此很容易

```
func 可达性(G: 矩阵 <, s, 布尔值 >,
    -> 向量 <源: 向量 <, 布尔值 >)
    , 布尔值> {
  v = 源;
  对于 i在 G.行数 {
    v += v * G;
  }
  返回 v;
}
```

图2：在 GraphAlg 中实现的可达性算法。

```
1  func 单源最短路径(G: 矩阵 <s, s, 热带_实数 >,
2      -> 向量 <源: 向量 <s, 热带_实数 >)
3      , 热带_实数> {
4    v = 源;
5    对于 i在 G.行数 {
6      v += v * G;
7    }
8    返回v;
9  }
```

图3：使用 GraphAlg 实现的单源最短路径算法。

进行并行化。此外，简化核心语言（第 3.2 节）和坚实的理论基础（第 3.3 节）使 GraphAlg 非常适合进行分析和优化。

易于使用。 GraphAlg 专为编写图算法而设计。线性代数运算具有广为人知的语义，能够简洁地表示图算法。

完全集成。 GraphAlg 表达式可以直接翻译为（关系代数的扩展），因此很容易集成到现有的查询处理流水线中。

3 THEGRAPHALG 语言

下一节中，我们将通过示例程序介绍 GraphAlg 语言的关键概念。在介绍完语言的“用户”侧之后，我们将讨论核心语言及其与 MATLANG [6] 的联系，这构成了 GraphAlg 的形式化基础。

3.1 GraphAlg 导览

第一个示例程序如图2所示。该程序从一组源顶点出发，遍历图，找出经过任意多跳后可从某个源顶点到达的所有顶点。如图1所示，矩阵乘法与图遍历密切相关。若取向量 v，其中先前迭代中已到达的所有顶点对应的值均为‘1’，并将其与表示该图的邻接矩阵相乘，结果便是一个向量，其中每个经过一跳可到达的顶点对应的值均为‘1’。可达性算法通过循环执行矩阵乘法，并累积每一步的结果。循环次数与图中的顶点数相同，因此代表对图进行一次完整遍历。GraphAlg 中的循环始终有界，其上限要么是图中的顶点数，要么是一个常数。为确保所有 GraphAlg 程序都能终止，禁止使用递归。

```
1  函数 withDamping(度数: 整数, 阻尼: 实数) -> real {
2    返回 类型转换 <实数 >(度数) / 阻尼;
3  }
4  func PageRank(G: 矩阵 <s, s, 布尔值 >,
5                 阻尼: 实数,
6                 迭代次数: int)
7      -> 向量 <s, 实数> {
8    n = G.行数;
9    传送 = (实数 (1.0) -阻尼) / 类型转换 <实数 >(n);
10   // 带阻尼的出度
11   d_出度 = reduceRows(类型转换 <整数 >(G));
12   d = 应用(withDamping, d_出度, 阻尼);
13   // 没有出边的顶点
14   连通 = reduceRows(G);
15   汇点 = 向量 <布尔值 >(n);
16   汇点 <! 连通 >[:] = 布尔值(true);
17
18   PageRank = 向量 <实数 >(n);
19   pr[:] = 实数 (1.0) / 类型转换 <实数 >(n);
20   对于 i在 整数(0):迭代次数 {
21     // 从悬挂节点重新分配的得分
22     汇点_pr = 向量 <实数 >(n);
23     汇点_pr <汇点> = pr;
24     重新分配 =(阻尼 / 类型转换 <实数 >(n))
25       * 减少(汇点_pr);
26     // 新的 PageRank 分数
27     w = pr (./) d;
28     pr[:] = 传送 + 重新分配;
29     pr += 类型转换 <实数 >(G).T * w;
30   }
31   返回 pr;
32 }
```

图4：在 GraphAlg 中实现的 PageRank。

Vector<s 中的 s、bool> 是一个<স্টyle id='3'>维度符号。它表示向量中的元素个数。使用抽象符号可以在无需为所有函数参数提供具体输入的情况下，对 GraphAlg 程序进行类型检查。例如，矩阵乘法要求左侧矩阵的列数与右侧矩阵的行数相同。GraphAlg 编译器可以根据维度符号在编译时检查这些要求，从而避免在运行时检测错误。第二个类型参数（此处为 bool）表示一个半环 [25]。半环定义矩阵元素的加法和乘法运算。对于布尔矩阵，这两种运算分别是 $\vee$ 和 $\wedge$。GraphAlg 还支持表示整数的 int 和表示实数/浮点数的 real，并采用常规运算符。半环是定义图算法的强大工具。例如，热带半环使用运算符 $\min(a, b)$ 执行加法，使用 + 执行乘法。若将图2中的可达性算法改为使用这一半环，就会得到图3所示的单源最短路径算法。请注意，由于热带半环将加法运算定义为 $\min(a, b)$，语句 v += v * G 应理解为 $v = min(v, v \cdot G)$。

图4中给出了 PageRank 的一个实现 [33]。我们重点介绍该算法中使用的几个其他语言构造：

- 类型转换<T>（第 2、9 行等）用于不同半环之间的转换，例如整数到实数的转换。
- reduceRows（第 11、14 行）将矩阵压缩为列向量，并使用加法运算符对元素求和。

• apply(f, M, c)（第 12 行）将标量函数 f 应用到 M 的每个元素，其中 c 是传递给 f 作为第二个参数的附加标量值。输出矩阵定义为 $C_{ij} = f(M_{ij}, c)$。• M[:] = c（第 16、19、28 行）将 M 的每个条目替换为标量 c。• A<M> = B（第 16、23 行）仅当 M 在相应位置的值非零时，才将 B 中的元素赋给 A 中的同一位置。• reduce（第 25 行）将矩阵的所有元素求和，得到一个标量。• (.f)（第 27 行）表示逐点函数应用。• M.T（第 29 行）表示 M 的转置。

PageRank 的实现表明，GraphAlg 具备通过组合线性代数运算来表达复杂算法的能力。命令式编程范式结合掩码、矩阵到向量的归约等高级运算，以及高级数据模型，使算法的定义清晰简洁。

3.2 核心语言

GraphAlg 编译器会将 GraphAlg 程序简化为 GraphAlg 的一个子集，我们将其称为 GraphAlg 核心。GraphAlg Core 用于分析、优化并将程序翻译为关系代数。核心语言由以下语法定义：

$$
\begin{aligned}
E &::= M && \text{(矩阵变量)}\\
&\mid E^{*} && \text{(转置)}\\
&\mid \mathrm{diag}(E) && \text{(向量对角化)}\\
&\mid \text{应用}[f](E_1, \cdots E_n) && \text{(逐点应用)}\\
&\mid E_1 \cdot E_2 && \text{(矩阵乘法)}\\
&\mid \mathbf{1}(r, s) && \text{(全一向量)}\\
&\mid \text{为}[b](s, E_1, \cdots E_n) && \text{（遍历维度）}\\
&\mid \text{循环}[b](l, E_1, \cdots E_n) && \text{（遍历范围）}\\
&\mid \mathrm{pickAny}(E) && \text{（保留行中第一个非零元素）}\\
b &::= \{M_1 = E_1, \cdots M_n = E_n\} && \text{(循环体)}\\
f &::= (C_1, \cdots C_n)\ e && \text{(逐点函数)}\\
e &::= C && \text{(标量变量)}\\
&\mid r(l) && \text{（标量字面量）}\\
&\mid e_1 \{+, \cdot -, / =\}\ e_2 && \text{（标量算术）}\\
&\mid \text{类型转换}(r, e) && \text{（标量类型转换）}
\end{aligned}
$$

大多数操作都是标准的线性代数概念，例如矩阵乘法或转置，遵循通常的语义 [28]。下面将更详细地介绍非标准操作的行为。

$\mathbf{1}(r, s)$ 构造一个维度为 s、半环为 r 的向量。向量的元素均初始化为乘法单位元，在大多数半环中，它是「1」。

'for' 操作定义了一个包含一个或多个状态变量 $M_1, \cdots M_n$ 的循环。循环体由每个状态变量对应的一个表达式组成，用于定义如何根据循环的前一状态生成下一次迭代的值。表达式可以通过引用某个状态变量来访问当前循环状态。状态变量的初始值由循环定义中的 $E_1, \cdots E_n$ 给出。循环的迭代次数由矩阵维度 s 或整数字面量 l 定义。循环表达式的结果是第一个状态变量的最终值 M_1。

'pickAny' 操作接收一个输入矩阵，并将每行除一个元素外的所有元素置零（前提是该行至少有一个非零元素）。这种原语适用于领导者选举问题，例如连通分量算法需要为每个分量选出一个代表顶点。图 5 第 8 行展示了 'pickAny' 操作的一个用法示例。

有关 GraphAlg Core 的更多细节，例如如何从完整的 GraphAlg 语言转换而来，或如何转换为关系代数，可参见语言规范 [10]。

3.3 理论基础

GraphAlg Core 基于 MATLANG [6] 及其扩展形式 MATLANG [17], , 它们是用于矩阵运算的形式语言。它们被视为关系代数在矩阵领域的对应物。MATLANG 之于 GraphAlg，正如关系代数之于 SQL：MATLANG 为 GraphAlg 提供了严谨的理论基础。该理论提供了一种将 GraphAlg 程序翻译为（扩展的）关系代数的方法 [7]。反过来，GraphAlg 可视为 MATLANG 的首个实现；此前，MATLANG 仅作为理论分析语言存在。大多数操作遵循 MATLANG [6] 中的定义，仅有少量改动。

GraphAlg 做出了三项关键设计决策，使其有别于 MATLANG，并支持高效的数据库集成。

可优化的循环分解。 for-MATLANG [17] 提供了表达能力极强的循环结构。GraphAlg 进一步将其分解为两个正交的基本操作：（1）与 [8], 中相同的有界循环；（2）专用于选择领导者的"pickAny"操作。这一设计旨在便于编译为关系代数并进行查询优化："pickAny"可以轻松地跨所有行并行执行，并编译为标准聚合，因此能够利用现有的数据库优化。将两者分离后，查询优化器便可分别分析并优化每个基本操作；若采用 for-MATLANG 更复杂的循环语义，则无法做到这一点。这种分解保留了与 for-MATLANG [10]相同的表达能力。

提升算法效率的同步归纳。 与 for-MATLANG 不同，GraphAlg 的循环支持同步归纳 [15], , 可在迭代过程中携带多个状态变量。这一设计选择借鉴自 [8], , 对于自然且高效地表达图算法至关重要。以广度优先搜索为例：它必须同时跟踪已访问节点，以及下一次迭代要探索的前沿。若没有同步归纳，算法就必须通过复杂的变通方法将多个值编码到单个矩阵中，大大增加实现和优化的难度。

实用的半环类型转换支持。 现实世界的图分析需要处理异构数据类型，但 MATLANG 假定所有矩阵共享同一个半环（通常为 $\mathbb{C}$）。GraphAlg 引入了"类型转换"操作，以支持实际分析工作负载，其中整数、浮点数和布尔值通常会混合使用。例如，计算无权最短路径

```
使用算法"
函数 WCC(G: 矩阵 <s、s、布尔值>)
    -> 矩阵 <s , s, 布尔值> {
  标识  = 向量 <布尔值 >(G.行数);
  标识符[: ] = 布尔值（真）;
  标签  =diag(id);
  对于 i在 G.行数 {
    标签  = pickAny(标签  +G * 标签);
  }
  返回 标签;
}" 调用 WCC((:人物)-[:发送_消息 ]->(:人物))
返回 row.name, col.name 作为 标签
```

图 5：嵌入 Cypher 中的弱连通分量算法。

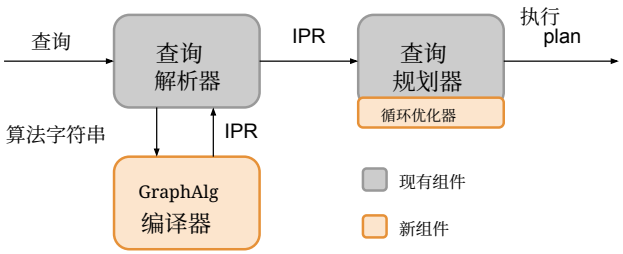

图 6：GraphAlg 编译器与 AvantGraph 中循环优化的集成。查询规划器之后的流水线阶段不受影响，因此未在图中显示。

需要将布尔邻接矩阵类型转换为热带半环。这项功能对于数据库集成至关重要，因为类型化数据是其基础。

4 AvantGraph 集成

我们将 GraphAlg 支持集成到 AvantGraph [36] 图查询引擎中，使用户能够直接通过 Cypher 在存储图上执行用户定义的图算法。

4.1 在查询中嵌入

算法定义可以直接嵌入 Cypher 查询中。我们引入 WITH ALGORITHM 子句，作为 Cypher 的扩展，使用户能够在查询内部定义 GraphAlg 程序。随后，可以使用通常用于内置过程的 CALL 语法，从 Cypher 中调用程序内定义的函数。图 5 展示了一个嵌入 GraphAlg 程序的查询示例。图作为参数传递给算法，其中可以包含标签谓词等附加过滤。虽然 Cypher 仅允许将节点/关系模式作为 CALL 参数，但在内部，AvantGraph 支持将任意查询作为 GraphAlg 程序的参数。

4.2 集成到查询流水线中

图 6 展示了 GraphAlg 编译器与 AvantGraph 查询流水线在高层次上的连接方式。IPR 是“内部计划表示”（Internal Plan Representation）的缩写，是 AvantGraph 内部使用的逻辑查询计划表示。GraphAlg 编译器最终输出 IPR，但在内部，它会基于 GraphAlg Core 表示执行分析和优化。这种设计确保 GraphAlg 编译器与 AvantGraph 查询流水线松耦合。第 5.2 节和第 5.3 节详细介绍的某些循环相关优化，必须应用于物理计划，而非逻辑计划。因此，这些优化与 GraphAlg 编译器分开，并作为查询规划器的扩展来实现。请注意，GraphAlg 编译器本身已经能够将 GraphAlg 程序转换为有效的 IPR：循环优化器只会提升所生成计划的质量。

4.3 统一表示

图 6 还揭示了另一个重要结论：GraphAlg 编译器完成工作后，查询和算法都会以 IPR 表示。此时，一个单独的 IPR 查询同时表示查询和嵌入的算法。查询与算法彼此变得无法区分。这对算法的执行具有一系列重要影响。首先，现有的查询规划器和运行时可以复用于算法执行。与为查询处理和算法处理分别设置流水线的其他系统相比，实现和维护算法支持所需的工程工作大幅减少。此外，在查询与算法之间交换数据不会产生代价。如图 5 所示，大型中间结果（甚至完整的图）可以从查询传递给算法，而无需先将其物化。由于查询规划器基于统一表示运行，它甚至可以跨越查询/算法边界应用优化，例如将输入参数上的查询侧筛选器下推到算法中。

4.4 对 AvantGraph 的更改

由于 GraphAlg 中的大多数操作都会编译为标准数据库操作（例如连接和聚合），因此只需进行少量更改，就能让 AvantGraph 成为 GraphAlg 编译器的功能完备目标。GraphAlg 中的循环结构是迭代算法所必需的，也是唯一无法模拟的操作。为支持这一结构，我们扩展了 AvantGraph 的内部查询表示，加入了一个「循环」算子，其设计紧密参照 GraphAlg Core 的定义。该算子的输入包括循环变量的初始状态，以及循环将执行的迭代次数。循环算子为每个循环变量嵌入一个子查询，用于计算该变量的下一个值。这些子查询共同编码了原始 GraphAlg 程序的循环体。图7展示了一个示例：在逻辑查询计划中使用循环算子，通过其他标准查询算子来表达单源最短路径算法。循环可以包含多个状态变量，但与 GraphAlg Core 一样，其输出始终恰好只有一个。最终输出是其中一个状态变量的值（示例中的状态 A）。这一点很重要，因为 AvantGraph 查询优化器要求查询计划必须是树形结构。

5 查询优化

我们介绍了对 GraphAlg 程序执行的关键优化，以便在 AvantGraph 中实现具有竞争力的性能。

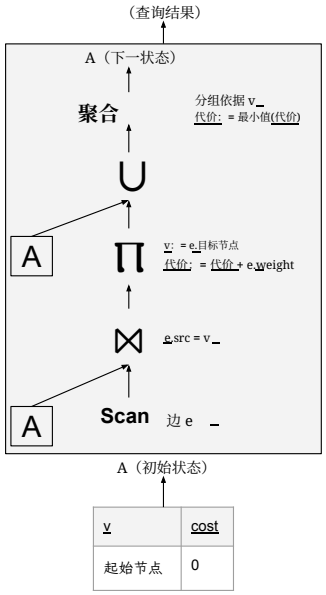

图7：单源最短路径算法的逻辑查询计划。

如图 6 所示，GraphAlg 编译器负责执行稀疏性分析，而循环优化则在 AvantGraph 查询规划器中实现。除了通过统一中间表示（第 4.3 节）对算法应用的标准查询优化技术（如谓词下推和连接顺序优化）之外，我们还执行了这些优化。

5.1 稀疏性

GraphAlg 编译器将矩阵表示为一个包含三个属性（行、列、值）的表。最朴素的关系代数转换方式，是让表中包含每一种有效行列组合对应的元组。在 AvantGraph 中，矩阵的维度对应于图中的顶点数量 V。这意味着矩阵表有 V^2 个条目，对于大型图而言，其开销高得令人难以承受。幸运的是，大多数真实世界图的邻接矩阵都非常稀疏，因此 GraphAlg 程序通常操作的是大部分元素为零的矩阵。为了节省存储空间和计算时间，通常只存储非零元素，我们将这种方式称为稀疏矩阵表示。GraphAlg 的设计使得大多数操作都能直接在稀疏矩阵表示上执行，而不必先转换为开销巨大的稠密表示。

举一个简单的例子，考虑 GraphAlg 表达式 A (.*) B。无论底层表是否省略了零元素，该表达式都可以转换为以下关系代数表达式：$\pi_{A_v \cdot B_v}(A \bowtie_{r,c} B)$。如果 A 和 B 是稀疏的，那么输入中的隐式零会使输出中相同位置也出现隐式零。对于任意半环上的矩阵乘法，情况也相同。不过，并非所有运算都能直接处理稀疏表示。例如，即使 $A_{rc} = B_{rc} = 0$，表达式 A (.==) B; 在位置 (r, c) 也有一个非零值。

GraphAlg 编译器会执行一项分析，跟踪程序中的每个矩阵是否可能是稀疏的（省略一些值为零的元组）。默认情况下，所有矩阵都以稀疏形式存储。如果遇到需要稠密矩阵的运算，就会显式地将相应值转换为稠密形式（添加值为零的元组）。这种方法会尽可能让数据保持稀疏表示，同时仍能为需要稠密输入的运算保留正确的语义。

5.2 循环不变量代码移动

循环不变代码外提 [1]是一种经典的程序优化技术。该优化会找出循环内部计算且每次迭代都产生相同值的表达式，并将其提前到循环之前计算一次，以避免重复计算。我们将这种程序优化技术应用于查询计划。在这种情况下，难点在于中间结果可能是庞大的元组流，必须将其缓存在缓冲区中，以便反复读取。在循环内部查找循环不变查询片段很容易，但要判断将其外提是否能提升查询性能，则没那么简单。请看图7中的查询计划。循环体包含一个会被反复执行且循环不变的边扫描节点。不过，将边扫描移出循环不太可能带来好处：扫描算子会生成元组流，必须将其缓存在主存中。对于足够大的图，存储所有元组的代价高得难以承受。

我们修改了循环不变代码外提的实现，使其只提取能生成哈希表的循环不变查询片段。这样一来，我们无需额外付出代价即可实现所需的缓存，因为哈希表可以反复探测。这也避免了在查询计划中引入额外的流水线阻断器。由于查询片段本身可能包含哈希表构建算子，我们总是优先提取尽可能大的查询片段。由 GraphAlg 生成的查询通常会大量使用（哈希）聚合，因此我们的方法非常有效。不过，这种方法并非最优，因为有些代价高昂的查询片段只会生成很小的中间结果，因此值得缓存。

5.3 跨循环迭代聚合

许多图算法中常见的一种模式，是聚合多次循环迭代产生的值。单源最短路径算法就是一个例子，如图7所示，其循环体的根节点是一个“聚合”算子。该聚合有两个输入，通过并集算子合并为一个：（1）左侧是当前迭代计算出的顶点新代价；（2）另一个是此前迭代收集到的顶点现有代价。每次迭代都会读取状态 A，将其与额外结果合并，然后再次进行聚合。假设状态 A 已存储在哈希表中，就没有必要读取该表并将结果聚合到另一个哈希表中。我们实现了一种循环优化遍，将额外结果直接添加到现有哈希表中。图8展示了应用原地聚合优化后，单源最短路径算法的循环体。请注意，循环开始前，哈希表中已填入初始状态。循环终止后，也会读取该表以获取最终循环结果。

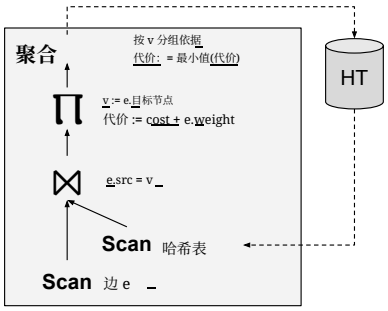

图8：采用原地聚合的单源最短路径算法的循环体。

采用原地聚合的循环也可以进行不动点分析。每次循环迭代结束时，AvantGraph 查询运行时都会检查任意循环变量的状态是否自上一次迭代以来发生了变化。如果某次迭代期间循环状态没有变化，那么循环就已达到不动点，不会再有进一步的状态变化，因此运行时会终止循环。正因如此，GraphAlg 程序通常可以安全地将循环定义为运行与图中顶点数量相同的迭代次数（例如参见图3），并依靠运行时提前退出循环。

6 实验

我们将嵌入 AvantGraph 的 GraphAlg 解决方案与其他支持用户定义算法的数据库系统进行比较。这些系统既包括原生图数据库（Neo4j），也包括关系型系统（PostgreSQL、DuckDB）。评估所用的算法和数据集来自 LDBC Graphalytics [21],，这是一个图分析基准测试。Graphalytics 面向图分析平台设计，并非专门针对数据库，但它为兼容 PostgreSQL 的数据库服务器提供了参考实现。我们的评估基于该基准测试规定的五种图算法：

- **广度优先搜索 (BFS)**：从源顶点出发到各顶点的最小跳数。
- **PageRank (PR)**：使用 PageRank [33], 近似计算顶点重要性，并从汇点重新分配权重。
- **弱连通分量 (WCC)**：找出图中的所有弱连通分量，并为每个分量中的顶点赋予相同的标签。
- **基于标签传播的社区检测 (CDLP)**：找出图中的弱连通社区 [34]。Graphalytics 规范 [22] 对算法做了少许修改，使其能够并行执行且结果具有确定性。
- **单源最短路径 (SSSP)**：从源顶点到所有顶点的加权最短路径总代价。

我们省略了 Graphalytics 中的第六种算法——局部聚类系数 (LCC)。虽然 GraphAlg 能够表达 LCC，但使用 Cypher 或非递归 SQL 也很容易表达，因此在此不值得作为一种有趣的算法加以探讨。

除了 LDBC Graphalytics，我们还在用于能源规划的 OpenAIRE 数据集上运行 PageRank， [4] 并采用不同的预处理步骤。这展示了系统在真实图上的表现；运行算法前，需要对图进行过滤或数据清洗等转换。

我们根据两项指标评估系统：**(1) 代码复杂度**：实现 LDBC Graphalytics 基准测试中的算法有多容易？**(2) 执行时间**：我们记录这些算法在 Graphalytics 的多个数据集以及用于能源规划的 OpenAIRE 数据集上的查询执行时间。

所有经过测试的实现都基于通用 API，可用于实现任意算法。我们明确不考虑 Neo4j 的图数据科学库 [32]或 DuckPGQ 的算法 [37], , 因为这些都是针对特定算法的解决方案：它们不能用于实现用户定义算法，只有在用户需要的正是该算法时才适用。

6.1 测试的系统

PostgreSQL。 我们纳入 PostgreSQL 15.2，因为已有 LDBC Graphalytics 的参考实现[2]。该实现使用 Java 编写循环控制流程，并向 PostgreSQL 服务器发送查询，以遍历图并执行计算。循环迭代之间保留的状态存储在临时表中。Java 组件必须在客户端执行，因此这并非完全在 PostgreSQL 中运行的自包含解决方案。另一种实现方式是使用 PL/pgSQL 实现算法，从而避免这一问题。这样做还有两个优点：(1) 减少客户端与服务器之间的往返次数；(2) 避免在每次循环迭代时解析查询。不过，对于我们在此测试的图而言，与查询执行时间相比，这些代价可以忽略不计。

DuckDB。 DuckDB 并非专门面向图的图数据库，也不支持过程控制流。不过，它针对分析处理进行了优化，并提供了易于使用的 Python API。尽管 Python 与 SQL 之间的数据交换不够便捷，但两种语言的结合能够提供稳健的查询性能和出色的表达能力。由于当时没有适用于 DuckDB 的现成 Graphalytics 配置，我们通过改造参考 PostgreSQL 实现，开发了自己的配置。我们使用 DuckDB v1.4，这是实验进行时的最新版本。与 PostgreSQL 相比，我们在 DuckDB 版本中进行了两项非平凡的修改。首先，PostgreSQL 实现的弱连通分量使用递归 CTE。在 DuckDB 中，这会导致我们测试过的所有图都出现内存不足错误。我们将其替换为使用临时表的迭代方法，该方法能够处理我们测试过的所有数据集。其次，所有临时表都保留在主存中，以避免磁盘 I/O 导致查询速度变慢。

Neo4j。 Neo4j 是一款广受欢迎的图数据库，支持用户定义算法。我们的实现基于 Pregel API [32], , 可使用 Java 定义自定义算法。我们使用 Neo4j 社区版 2025.08 和图数据科学库 2.21，这是实验时的最新版本。我们将自定义算法编译为 Neo4j 插件，然后使用 Cypher CALL 语法调用。我们指出 Neo4j 存在两个问题，它们不仅影响我们的实验，也会影响实际应用。首先，Neo4j 目前不支持按需加载

[2]https://github.com/ldbc/ldbc_graphalytics_platforms_umbra

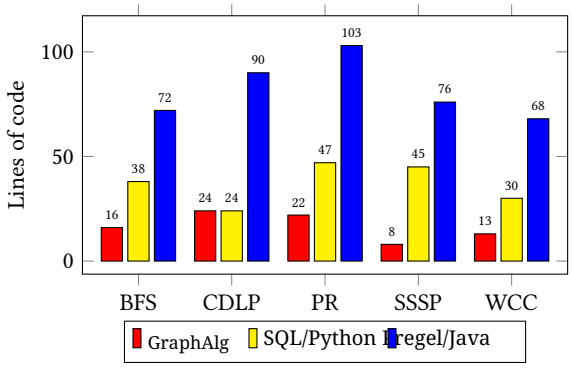

图9：各算法的代码行数，不包括空白和注释。GraphAlg 在所有算法中的代码行数均为最少，最多比其他实现少 9.5 倍。

插件。要加载自定义算法，必须关闭数据库服务器，然后重新启动。这使得算法设计的反复迭代变得繁琐，在多租户实例上几乎不可能实现。或许正因如此，Pregel API 并未在 Neo4j 的托管云服务中提供。其次，算法并不运行在常规查询所用的图存储上。用户需要创建一个单独的内存中 图对象，然后在该图对象上运行算法。因此，Neo4j 要求图必须完全装入主存，严重影响了可扩展性。我们预计，在实际场景中，用户通常需要在执行算法后立即卸载图，以便为其他任务释放内存。

6.2 实验1：GraphAlg 降低代码复杂度

为展示不同系统在代码复杂度方面的差异，我们在图10中并列比较了 PageRank 算法的循环体。三种实现都以相同的方式计算更新后的 PageRank 分数。首先，计算需要从汇点重新分配的得分。然后，将传送因子和重新分配因子相加，并加上相邻顶点的得分，从而计算出新的得分。

GraphAlg 在代码量方面表现突出，只需 7 行代码即可完成这两项操作。SQL 版本则因负责协调执行的 Python 代码而变得复杂。此外，还需要复杂的多路连接，才能汇集计算更新后的 PageRank 分数所需的所有数据。Pregel 实现由命令式 Java 代码构成，并行处理多个顶点。其复杂性主要源于顶点之间显式的消息交换，以及对汇点和消息的显式迭代。

各种算法中都反复呈现出类似的模式。图 9 展示了在不同系统中表达各个算法所需的代码行数。由于 PostgreSQL 实现的 SQL/Java 版本与 DuckDB 版本非常相似，我们未将其纳入统计。统计时不计空白字符和注释，也不计构建 Neo4j 插件所需的 Gradle 构建文件。我们发现，GraphAlg 程序在所有情况下代码量都最少。

Pregel/Java 实现最为冗长，这主要是因为 Java 语言本身通常较为冗长。对于 CDLP 算法，GraphAlg 和 SQL/Python 的代码行数竟然完全相同，但两者的实现却大不相同。

上述结果支持了我们的假设：与 SQL/Python 或 Pregel/Java 相比，在 GraphAlg 中实现算法更加容易。通过针对 GraphAlg 编程开展用户研究，还可以获得更深入的见解。这将留待未来工作。

6.3 实验 2：GraphAlg/AvantGraph 在 LDBC Graphalytics 上具有竞争力

我们测量了 LDBC Graphalytics 中所有不超过“S”规模的数据集上广度优先搜索、CDLP、PageRank、单源最短路径和弱连通分量算法的执行时间。结果通过 LDBC Graphalytics 基准测试驱动程序收集。我们记录的是处理时间：执行实际算法所需的时间，不包括将图导入系统的时间，或（对于 Neo4j）将其加载到主存中的时间。实验在一台笔记本电脑上进行，配备 Intel i7-11800H CPU（8 核、16 个超线程）、32GB 主存和一块 NVMe 固态硬盘。结果见表 2。

PostgreSQL。 PostgreSQL 的速度始终显著慢于其他系统。由于篇幅限制，表中省略了相关结果。有兴趣的读者可参照我们的基准测试设置复现这些结果。

Neo4j 中的内存不足错误。 在包含大量顶点（13M）的数据集上，Neo4j 中的 CDLP、单源最短路径和弱连通分量算法因“内存不足”错误而失败。这表明 Pregel 接口的每顶点开销较高。此外，Neo4j 要求将整个图加载到主存中，因此留给算法中间状态的空间更少。

广度优先搜索。 GraphAlg/AvantGraph 与 DuckDB 的性能相当，Neo4j 则远远落后。在较大的 datagen-7_* 数据集上，GraphAlg/AvantGraph 的表现不如 DuckDB。边查找是 GraphAlg/AvantGraph 的主要瓶颈，这很可能是因为其磁盘格式不如 DuckDB 高效。对于 datagen-7_8-zf 数据集，加载到 AvantGraph 后，图的大小是原来的 5 倍以上。

CDLP。 在此算法中，DuckDB 和 Neo4j 的表现都远胜于 GraphAlg/AvantGraph。DuckDB 根据 SQL 生成的查询计划与 GraphAlg 生成的 AvantGraph 查询计划非常相似，因此性能差异不能归因于翻译不佳。相反，我们发现，在执行大型哈希聚合时，GraphAlg/AvantGraph 的速度明显慢于 DuckDB。该操作会在每次循环迭代中重复执行，因此对处理时间有重大影响。这凸显了 AvantGraph 中一个亟待解决的重要性能问题。

PageRank。 GraphAlg/AvantGraph 在此基准测试中表现最佳，但 DuckDB 也不遑多让。dota-league 数据集是个例外，DuckDB 的速度快于 GraphAlg/AvantGraph。我们认为，这是因为该数据集的顶点数（61K）相较于边数（50M）极低。在 AvantGraph 中，这会导致执行哈希聚合的不同线程之间发生争用。DuckDB 似乎能更好地处理这种情况。

清单 1：GraphAlg（AvantGraph）

```
汇点_PageRank = 向量<实数>(n);
汇点_PageRank<汇点> = pr;
重新分配 = 阻尼 / 类型转换<实数>(n)
重新分配 = 重新分配 * 归约(汇点_PageRank);

w = PageRank(,) d;
PageRank[:] = 传送 + 重新分配;
PageRank += 类型转换<实数>(图)。T * w;
```

清单 2：SQL/Python（DuckDB）

```
(重新分配,) = 连接.sql(f"""
  SELECT 合并(
    ?*求和(pr{i-1}.value), 0)
  FROM 悬挂节点
  连接 PageRank{i-1} 使用 (id)
""",
  [重新分配因子]).fetchone()

连接.execute(f"""
  创建表 PageRank{i}作为
  选择
    g.v.id 作为 标识符,
    ($teleport +
      ($damping * 求和(
        合并(pr{i-1}.value
          / out_degree.degree, 0)))
      + $重新分配)作为值
  FROM g.v
  左连接 g.e ON(
    g.e.target =g.v.id)
  左连接 pr{i-1} ON (
    pr{i-1}.id = g.e.source)
  左连接 出_度数 于(
    pr{i-1}.id =出_度数.id)
  分组依据g.v.id
""", {
  "传送": 传送,
  "阻尼": 阻尼,
  "重新分配": 重新分配,
})
conn.execute(f"删除表 pr{i-1}")
```

清单 3：Pregel/Java（Neo4J）

```
变量 求和 = new 可变双精度数();
汇点.forEach(汇点 ->{
  double pr =
    context.doubleNodeValue(
      汇点, PAGE_RANK);
  sum.add(pr);
});
重新分配 =(
  context.config().阻尼()
    / context.节点数())
      * 求和.doubleValue();

double imp = 0;
遍历(变量 消息 : messages) {
  imp += 消息;
}
变量 阻尼 = context.config().
    阻尼因子();
var 传送 = (1 - 阻尼)
  / context.节点数();
newRank = 传送 +(阻尼 * 重要性)
  + 重新分配;
context.设置节点值(PAGE_RANK,
    newRank);
if (!汇点) {
  context.sendToNeighbors(newRank /
      context.degree());
}
```

图10：PageRank 循环体不同实现的代码复杂度。GraphAlg 版本的代码复杂度低得多，篇幅也短得多。

数据集	BFS			CDLP			PR			SSSP			WCC		
	GAA	DDB	N4J	GAA	DDB	N4J	GAA	DDB	N4J	GAA	DDB	N4J	GAA	DDB	N4J
专利引用网络	0.69	**0.43**	4.67	62.18	12.41	**0.96**	**4.49**	5.73	7.59	*N/A*	*N/A*	*N/A*	**5.84**	14.31	17.96
Dota 联赛	**0.33**	0.43	3.03	35.11	**9.26**	22.15	8.66	**7.61**	15.37	6.64	**5.40**	20.75	**1.27**	2.34	7.56
Graph500-22	**1.42**	1.44	11.78	97.24	**18.66**	54.07	**15.37**	22.53	18.71	*N/A*	*N/A*	*N/A*	**4.35**	11.15	23.22
kgs	0.78	**0.43**	2.13	27.42	**5.52**	17.55	**4.09**	5.00	5.88	**3.96**	4.62	11.04	**1.22**	3.90	6.14
维基百科讨论网络	0.63	**0.33**	1.97	17.41	3.08	**0.41**	**2.30**	2.41	5.73	*N/A*	*N/A*	*N/A*	**1.45**	1.90	5.51
数据生成-7_5-fb	**0.52**	0.52	3.56	85.86	**17.88**	25.61	**7.00**	9.58	11.49	**1.57**	1.78	9.05	**1.41**	4.24	12.04
数据生成-7_6-fb	**0.60**	0.72	4.23	106.69	**22.76**	34.21	**8.78**	12.34	9.95	**1.86**	2.15	11.93	**1.68**	5.33	14.40
数据生成-7_7-zf	14.84	**3.35**	13.81	130.33	**28.40**	OOM	**16.59**	19.75	34.41	**23.40**	32.25	OOM	**16.41**	41.30	OOM
数据生成-7_8-zf	22.64	**4.75**	23.23	174.63	**36.93**	OOM	**20.44**	25.67	35.79	**34.97**	51.12	OOM	**25.65**	62.07	OOM
数据生成-7_9-fb	**1.20**	1.38	10.09	229.05	**48.81**	80.59	**19.30**	26.26	20.27	**3.15**	3.90	25.80	**3.96**	11.61	33.85

表 2：GraphAlg/AvantGraph（GAA）、DuckDB（DDB）和 Neo4j（N4J）在 LDBC Graphalytics 数据集（最大规模 S）上的处理时间。每个数据集/算法的最短执行时间以粗体标出，次优结果加下划线。GraphAlg/AvantGraph 在 PageRank、单源最短路径和弱连通分量算法上占据优势，在广度优先搜索算法上也具有竞争力。

单源最短路径。 GraphAlg/AvantGraph 的速度快于 DuckDB 和 Neo4j，与 PageRank 的情况相似。与 DuckDB 相比，GraphAlg/AvantGraph 的一项主要优势是，它会在多次循环迭代中维护同一个结果表；而 DuckDB 版本则必须在每次迭代时重新创建一张表。

弱连通分量。 GraphAlg/AvantGraph 在基准测试中占据明显优势。GraphAlg/AvantGraph 与 DuckDB 之间的差距更大因为该算法不过是反复进行聚合，直到达到不动点。这比单源最短路径更清楚地表明，原地聚合对于高效执行有多么重要。

总结。 GraphAlg/AvantGraph 在 PageRank、单源最短路径和弱连通分量算法方面优于其他系统，在广度优先搜索方面也表现出有竞争力的性能。

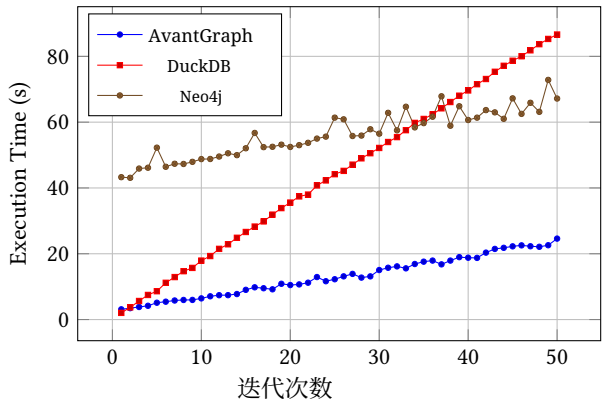

图 11：在移除了自环边的 Ope-nAIRE 能源规划数据集中，对引文运行 PageRank 的执行时间。GraphAlg/AvantGraph 的扩展性优于 DuckDB，同时避免了 Neo4j 的预处理开销。

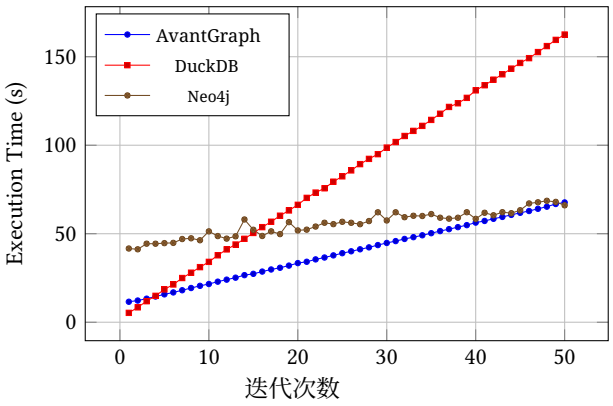

图 12：GraphAlg/AvantGraph 在 Ope-nAIRE 能源规划数据集的引文数据上运行 PageRank 的执行时间。 [4] 移除重复边后，GraphAlg/AvantGraph 的扩展性优于 DuckDB，并且在迭代次数较多时仍比 Neo4j 更快。

未来还需改进 AvantGraph 中哈希聚合的性能，以支持 CDLP 的高效执行。

6.4 实验3：GraphAlg/AvantGraph 能稳健地处理预处理

真实图分析需要灵活的预处理，但传统系统迫使用户做出尴尬的选择：要么物化清洗后的数据（这对探索性工作流而言代价高昂），要么将预处理硬编码到算法中（缺乏灵活性）。此类预处理在实践中无处不在——例如按时间窗口筛选边、移除机器人账户或测试账户、处理重复项和自环、规范化权重，或提取领域特定子图。GraphAlg/AvantGraph 的统一 IR 提供了第三种选择：查询端预处理，优化器可以将其与算法执行融合。本实验评估这一集成能否带来实际效益。

我们在 SciLake[3] 项目中观察到了灵活预处理的重要性。在该项目中，研究人员会在引文图上运行 PageRank，以找出高影响力出版物。图存在需要清理的数据质量问题，但合适的清洗流程是通过反复试验发现的。这种迭代探索要求支持查询端预处理，同时不修改算法定义。

我们假设，GraphAlg/AvantGraph 的交叉优化（第 4.3 节、第 5.2 节）能够以极低的开销实现查询端预处理，相较之下，其他系统要么在每次迭代中重新执行预处理（DuckDB），要么需要代价高昂的物化操作（Neo4j）。

实验设置。 我们选择 SciLake 项目中使用的最大引文图，即用于能源规划的 OpenAIRE 数据集 [4]，这是一个涵盖能源规划领域科学出版物的知识图谱。我们考虑两种需要对图进行预处理的用例：

无自引。 我们通过移除引用论文与被引论文的第一作者相同的引文，来降低引用自己论文的作者所产生的影响。

[3]https://scilake.eu

这通过每条边上的一个布尔标志来表示。绝大多数引文并非自引的（99%），因此这是一个简单、选择性较低的谓词。

无重复边。 两篇论文之间存在多条引文边，说明数据清洗存在问题。解决方法是从图中移除重复边。这是一种成本显著更高的筛选操作，因为它需要对图中的所有边进行聚合。

这两种情况用于测试不同的优化能力：（1）针对廉价谓词进行筛选下推；（2）针对开销较大的聚合执行循环不变代码外提。

使用 GraphAlg/AvantGraph，我们可以在查询端将所需的预处理作为参数传递给算法，而无需修改底层算法。我们依靠查询优化器选择合适的查询计划，将预处理步骤高效地整合到算法执行中。对于 DuckDB，我们修改上一项实验中使用的 PageRank 实现，使其额外接收一个边视图 作为参数，并使用该视图，而不是直接扫描边表。这样，查询优化器也能将预处理纳入考虑。对于 Neo4j，我们可以直接使用现有的 PageRank 实现，而在将节点和关系从磁盘加载到主存时完成所有预处理。因此，预处理是一个独立步骤，必须在算法执行前完成。预处理时间计入测得的执行时间。

迭代次数是用户为算法设定的参数。迭代次数越多，得分就越准确，但最终得分会趋于稳定，继续增加迭代次数带来的改善也很有限。我们最多收集 50 次迭代的结果。PageRank [33] 论文指出，对于包含 1.61 亿条边的图，45 次迭代已足够，而本文使用的图仅包含 4700 万条边。

结果。 结果（图 11和图 12）验证了我们的两项设计原则：集成（通过查询组合表达预处理）和可优化性（查询与算法之间的交叉优化）。GraphAlg/AvantGraph 的表现优于

其他系统，且迭代次数设定在合理范围内（4–50 次）。

对于“无自引”的情况，得益于统一中间表示（第 4.3 节），GraphAlg/AvantGraph 可以将查询端的筛选器下推到算法内部的边表扫描中。这使其速度远快于 Neo4j；Neo4j 必须先完成耗时的加载步骤，才能开始执行算法。即使迭代 50 次，GraphAlg/AvantGraph 也能在 Neo4j 完成图预处理之前执行完毕。查询计划分析表明，DuckDB 也能够将筛选器下推到表扫描中，但如表 2 所示，其执行 PageRank 算法的效率不如 GraphAlg/AvantGraph。

对于“无重复边”的情况，GraphAlg/AvantGraph 应用循环不变代码外提优化（第 5.2 节），只计算一次聚合。这使其比 DuckDB 更具可扩展性，因为 DuckDB 会在每次循环迭代中重新计算聚合。Neo4j 的可扩展性优于 GraphAlg/AvantGraph，但只有在迭代 50 次后，这一优势才能抵消其高昂的预处理开销；对于这张图而言，50 次迭代并不现实。

这些预处理示例只是跨优化机会的冰山一角。统一中间表示支持在查询与算法边界上进行许多其他优化：将算法结果与外部表连接，在流水线中组合多个算法，对中间结果应用复杂转换，或根据算法输出动态选择子图。这些场景都受益于同一原则：在单一优化框架中将查询和算法视为一等公民，展现了我们方法的广泛适用性，远不止于本文研究的具体案例。

7 结论与未来工作

我们介绍了 GraphAlg，这是一种领域专用语言，旨在作为图数据库系统中的一等公民来设计图算法。GraphAlg以 MATLANG 和 for-MATLANG 为基础，扎根于线性代数，提供可组合操作、清晰的语义，并可直接翻译为关系代数的扩展。

GraphAlg 与 AvantGraph 的集成表明，图算法和图查询能够共存于统一的高性能框架中。通过将 GraphAlg 编译为与 Cypher 相同的内部表示，我们实现了查询与算法之间的交叉优化，消除了数据整理开销。我们的实验评估验证了全部四项设计原则：通过多样化的 LDBC Graphalytics 实现体现表达能力；通过代码量减少 2–10 倍体现易用性；通过统一的 IPR 表示实现集成；以及通过稀疏性分析、循环不变代码外提和原地聚合实现可优化性。值得注意的是，GraphAlg/AvantGraph 在 PageRank、单源最短路径和弱连通分量等关键算法上实现了卓越性能，其表现可与高度优化的分析型数据库相媲美，甚至更胜一筹，同时直接在持久化图结构上运行。

我们未来的工作包括进一步开展用户研究，以验证其可用性优势、解决 AvantGraph 的性能瓶颈，并探索更多编译目标。

致谢

这项工作获得了欧盟“地平线欧洲”框架计划的资助，资助协议编号为 101058573，属于 SciLake 项目的一部分。

参考文献

[1] Alfred V. Aho、Monica S. Lam、Ravi Sethi 和 Jeffrey D. Ullman（编）。2007。《编译原理、技术与工具》（第2版，Pearson 国际版）。Pearson Addison-Wesley，波士顿、慕尼黑。[2] ArangoDB Inc. [无日期]。《图分析》| ArangoDB 文档。https://docs.arangodb.com/stable/data-science/graph-analytics/ [3] Ariful Azad、Mohsen Mahmoudi Aznaveh、Scott Beamer、Maia P. Blanco、Jin- hao Chen、Luke D’ Alessandro、Roshan Dathathri、Tim Davis、Kevin Deweese、Jesun Firoz、Henry A Gabb、Gurbinder Gill、Balint Hegyi、Scott Kolodziej、Tze Meng Low、Andrew Lumsdaine、Tugsbayasgalan Manlaibaatar、Timothy G Mattson、Scott McMillan、Ramesh Peri、Keshav Pingali、Upasana Sridhar、Gabor Szarnyas、Yunming Zhang 和 Yongzhe Zhang。2020。使用 GAP 基准测试套件评估图分析框架。载于*2020 IEEE* 国际工作负载特征研讨会（*IISWC*）。216–227。https://doi.org/10.1109/IISWC50251.2020.00029。[5] Miriam Baglioni。无日期。符合 SKG-IF 标准的 SciLake 能源规划试点 OpenAIRE 数据集。https://doi.org/10.5281/zenodo.17084896[5] Houda Boukham、Guido Wachsmuth、Toine Hartman、Hamza Boucherit、Oskar van Rest、Hassan Chafi、Sungpack Hong、Martijn Dwars、Arnaud Delamare 和 Dalila Chiadmi。2023。Oracle 中的 Spoofax：面向大规模图分析的领域专用语言工程。载于*Eelco Visser* 纪念研讨会（*EVCS 2023*）（开放获取信息学丛书（*OASIcs*））。Ralf Lämmel、Peter D. Mosses 和 Friedrich Steimann（编），第109卷。Schloss Dagstuhl – Leibniz 信息学中心，德国 Dagstuhl，5:1–5:8。https://doi.org/10.4230/OASIcs.EVCS.2023.5 ISSN：2190-6807。[6] Robert Brijder、Floris Geerts、Jan Van Den Bussche 和 Timmy Weerwag。2019。矩阵查询语言的表达能力。*ACM* 数据库系统汇刊44, 4（2019年12月），15:1–15:31。https://doi.org/10.1145/3331445。[7] Robert Brijder、Marc Gyssens 和 Jan Van den Bussche。无日期。关于矩阵与 K-关系。《数学与人工智能年刊》90，2（2022年3月），181–210。https://doi.org/10.1007/s10472-021-09760-4。[8] Ashok K. Chandra。无日期。数据库语言的编程原语。载于第*8*届 *ACM SIGPLAN-SIGACT* 编程语言原理研讨会论文集（*POPL ’81*）。计算机协会，美国纽约州纽约市，50–62。https://doi.org/10.1145/567532.567537。[9] Timothy A. Davis。无日期。算法：SuiteSparse:GraphBLAS：稀疏线性代数语言中的图算法。*ACM* 数学软件汇刊45，4（2019年12月），44:1–44:25。https://doi.org/10.1145/3322125。[10] Daan de Graaf。无日期。GraphAlg：一种用于以线性代数编写图算法的可嵌入语言。http://wildarch.dev/graphalg/ [11] Alin Deutsch、Yu Xu、Mingxi Wu 和 Victor Lee。2019。TigerGraph：原生 MPP 图数据库。https://doi.org/10.48550/arXiv.1901.08248。[12] Alin Deutsch、Yu Xu、Mingxi Wu 和 Victor E. Lee。2020。TigerGraph 对现代图分析的聚合支持。载于*2020* 年 *ACM SIGMOD* 数据管理国际会议论文集（*SIGMOD ’20*）。计算机协会，美国纽约州纽约市，377–392。https://doi.org/10.1145/3318464.3386144[13] Christian Duta。2022。实现复杂计算的另一种方法：函数式风格 SQL 用户定义函数。载于人机协同数据分析研讨会论文集。ACM，美国宾夕法尼亚州费城，1–7。https://doi.org/10.1145/3546930.3547508[14] Christian Duta、Denis Hirn 和 Torsten Grust。2019。编译消除 PL/SQL。*CoRR* abs/1909.03291（2019）。http://arxiv.org/abs/1909.03291 arXiv：1909.03291。[15] Heinz-Dieter Ebbinghaus、Jörg Flum 和 Wolfgang Thomas。2021。数理逻辑。数学研究生教材，第291卷。施普林格国际出版公司，尚。https://doi.org/10.1007/978-3-030-73839-6。[16] Nadime Francis、Alastair Green、Paolo Guagliardo、Leonid Libkin、Tobias Lindaaker、Victor Marsault、Stefan Plantikow、Mats Rydberg、Petra Selmer 和 Andrés Taylor。2018。Cypher：一种不断演进的属性图查询语言。载于*2018* 年数据管理国际会议论文集（*SIGMOD ’18*）。计算机协会，美国纽约州纽约市，1433–1445。https://doi.org/10.1145/3183713.3190657。[17] Floris Geerts、Thomas Muñoz、Cristian Riveros 和 Domagoj Vrgoč。2021。线性代数查询语言的表达能力。载于第*40*届 *ACM SIGMOD-SIGACT-SIGAI* 数据库系统原理研讨会论文集（*PODS’21*）。计算机协会，美国纽约州纽约市，342–354。https://doi.org/10.1145/3452021.3458314。[18] Surabhi Gupta 和 Karthik Ramachandra。2021。SQL 过程化扩展：了解其在实际中的使用情况。*VLDB* 基金会会刊14，8

（4月 2021 1378 1391 10 14778 3457390 3457402），–。[19] https://doi.org/。/。Denis Hirn 和 Torsten Grust。无日期。对固守不动点的修正。载于第 *13*届创新数据系统研究会议论文集。[20] Sungpack Hong、Hassan Chafi、Edic Sedlar 和 Kunle Olukotun。2012。Green-Marl：一种便捷高效的图分析领域专用语言。*ACM SIGPLAN Notices* 47，4（2012 349 362 10 1145 2248487 2151013 年6月），–。https://doi.org/。[21] /。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 和 Peter Boncz。2016。LDBC Graphalytics：面向并行与分布式平台大规模图分析的基准测试。*VLDB* 基金会会刊9，13（2016年9月），1317–1328。https://doi.org/10.14778/3007263.3007270[22] Alexandru Iosup、Ahmed Musaafir、Alexandru Uta、Arnau Prat Pérez、Gábor Szárnyas、Hassan Chafi、Ilie Gabriel Tănase、Lifeng Nai、Michael Anderson、Mihai Capotă、Narayanan Sundaram、Peter Boncz、Siegfried Depner、Stijn Heldens、Thomas Manhardt、Tim Hegeman、Wing Lung Ngai 和 Yinglong Xia。2023。LDBC Graphalytics 基准测试。https://doi.org/10.48550/arXiv.2011.15028 arXiv：2011.15028 [cs]。[23] 国际标准化组织。2024。信息技术——数据库语言——GQL。https: //www.iso.org/standard/76120.html[24] Hideya Iwasaki、Kento Emoto、Akimasa Morihata、Kiminori Matsuzaki 和 Zhenjiang Hu。2022。Fregel：一种用于以顶点为中心的大规模图处理的函数式领域专用语言。函数式编程杂志32（2022 4 10 1017 0956796821000277 年1月），e。https://doi.org/。/S Jeremy Kepner 和 John Gilbert（编）。[25] 无日期。线性代数语言中的图算法。工业与应用数学学会。https: //doi.org/ 10 1137 1 9780898719918 。/。Arijit Khan。无日期。以顶点为中心的图处理：优点、缺点与弊端。[26] https://doi.org/ 10 5441 002 2017 39 。/ /EDBT。。Louisa Lambrecht、Torsten Grust、Altan Birler 和 Thomas Neumann。2025。SQL的蹦床式查询。载于第*15*届创新数据系统研究会议论文集。[27] [28] Serge Lang。1986。线性代数导论。施普林格，美国纽约州纽约市。https://doi.org/10.1007/978-1-4612-1070-2 2017

[29] Grzegorz Malewicz、Matthew H. Austern、Aart J.C Bik、James C. Dehnert、Ilan Horn、Naty Leiser 和 Grzegorz Czajkowski。2010。Pregel：用于大规模图处理的系统。收录于*2010* 年 *ACM SIGMOD* 数据管理国际会议论文集（*SIGMOD '10*）。美国计算机协会，纽约州纽约市，135 146 10 1145 1807167 1807184–。[30] https://doi.org/。/。Tim Mattson、Timothy A. Davis、Manoj Kumar、Aydin Buluc、Scott McMillan、Jose Moreira 和 Carl Yang。2019。LAGraph：一个致力于收集基于 GraphBLAS 构建的图算法的社区项目。收录于*2019* 年 *IEEE* 并行与分布式处理国际研讨会论文集（*IPDPSW*）。276–284。https://doi.org/ 10 1109 2019 00053 。/IPDPSW.。Frank McSherry、Michael Isard 和 Derek G. Murray。2015。可扩展性！但代价是什么？收录于第 *15* 届 *USENIX* 操作系统热点议题会议论文集（*HOTOS'15*）。USENIX 协会，美国，14。[31] [32] Neo4J 公司 [n.d.]。图算法 - Neo4j 图数据科学。https://neo4j.com/docs/graph-data-science/2.17/algorithms/ [33] Lawrence Page、Sergey Brin、Rajeev Motwani 和 Terry Winograd。1999。*PageRank* 引文排名：为网络带来秩序。技术报告 1999-66。斯坦福信息实验室。http://ilpubs.stanford.edu:8090/422/ [34] Usha Nandini Raghavan、Réka Albert 和 Soundar Kumara。2007。用于检测大规模网络中社区结构的近线性时间算法。物理评论 *E* 76, 3（2007 年 9 月），036106。https://doi.org/10.1103/PhysRevE.76.036106 出版方：美国物理学会。[35] Moritz Sichert 和 Thomas Neumann。2022。用户定义算子：将自定义算法高效集成到现代数据库中。*VLDB* 基金会会刊 15, 5（2022 年 1 月），1119–1131。https://doi.org/10.14778/3510397.3510408[36] Wilco van Leeuwen、Thomas Mulder、Bram van de Wall、George Fletcher 和 Nikolay Yakovets。2022。AvantGraph 查询处理引擎。*VLDB* 基金会会刊 15, 12（2022 年 8 月），3698–3701。https://doi.org/10.14778/ 3554821.3554878[37] Daniel Ten Wolde、Gábor Szárnyas 和 Peter Boncz。2023。DuckPGQ：将 SQL/PGQ 引入 DuckDB。*VLDB* 基金会会刊 16, 12（2023 年 8 月），4034–4037。https://doi.org/10.14778/3611540.3611614[38] Yunming Zhang、Mengjiao Yang、Riyadh Baghdadi、Shoaib Kamil、Julian Shun 和 Saman Amarasinghe。2018。GraphIt：高性能图领域专用语言。*ACM* 编程语言论文集 2，面向对象编程、系统、语言与应用（2018 年 10 月），121:1–121:30。https://doi.org/10.1145/3276491