arXiv:2603.03705v1 [cs.DB] 4 Mar 2026

GraphLake：专为湖仓打造的图计算引擎

Shige Liu
普渡大学
USA
liu3529@purdue.edu

Songting Chen
TigerGraph
USA
songting.chen@tigergraph.com

Chengjie Qin
TigerGraph
USA
chengjie.qin@tigergraph.com

Mingxi Wu
TigerGraph
USA
mingxi.wu@tigergraph.com

Jianguo Wang
普渡大学
USA
csjgwang@purdue.edu

摘要

本文介绍了 GraphLake，这是一款专为湖仓打造的图计算引擎。GraphLake 构建于商业图数据库 TigerGraph 之上。它将湖仓表映射为带标签属性图中的顶点类型和边类型，并支持使用 GSQL 对湖仓表进行图分析。为缩短启动时间，它仅加载图拓扑。此外，GraphLake 还引入了一系列技术，以确保对湖仓表的查询效率，包括一种图感知缓存机制和两种针对湖仓优化的并行原语。大量实验表明，GraphLake 的性能显著优于目前最先进的湖仓图计算引擎 PuppyGraph，在启动时间和查询时间方面均更短。

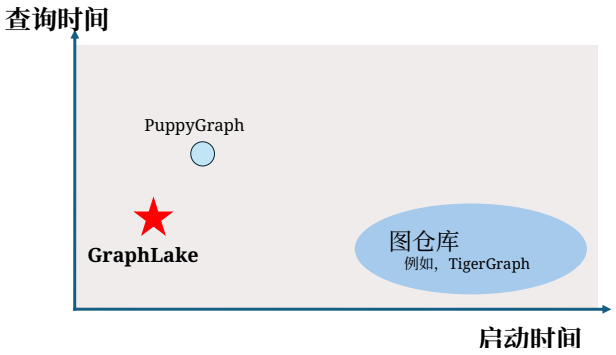


图 1：启动时间与查询时间的平衡

1 引言

过去十年间，随着云存储的发展以及对大规模分析需求的增长，数据管理架构经历了重大范式转变。传统数据仓库具有高性能和可靠的事务保证；然而，它们通常依赖专有格式，存在供应商锁定的风险，并采用紧密耦合的架构，扩展成本高昂。相比之下，数据湖 [28, 42] 应运而生，通过使用开放文件格式（例如 Apache Parquet [6] 和 Apache ORC [5]）将数据存储在云对象存储中，成为一种低成本替代方案，且不受特定系统限制。数据湖通过计算与存储分离支持弹性扩展，但对事务、模式管理和性能优化的支持有限，这可能会使可靠、高效的分析流水线开发变得复杂。

作为弥合这一差距的方案，数据湖仓架构 [21, 22, 36, 53]近来应运而生，它将数据湖的开放性和可扩展性与数据仓库的可靠性和性能优势相结合。开放表格式是这一架构的核心。Apache Iceberg [4, 43], 、Delta Lake [8, 21],和 Apache Hudi [3]都是其中的代表。湖仓将数据存储在云对象存储中——
利用这些开放表格式（底层采用 Apache Parquet 等开放文件格式 [6]），同时添加类似数据仓库的功能，包括 ACID 事务、模式演进、时间旅行和元数据管理。这种设计在业界迅速普及，因为它使多个处理引擎（例如 SQL 分析、机器学习和流处理系统）能够在同一数据之上运行，无需冗余 ETL 或重复存储。随着各组织对云规模的人工智能与分析统一平台的需求日益增长，湖仓通过提供表抽象，已成为现代数据生态系统中的基础架构。

因此，越来越多的数据系统（主要是 OLAP 引擎）开始针对湖仓构建，例如 Databricks Photon [22],Google BigLake [36], Microsoft Fabric [18], 和 Apache Presto [45]。此外，Amazon Redshift [2],Snowflake [33], SingleStore [40],和 DuckDB [41] 也已开始支持 Lakehouse 架构。

在这项工作中，我们专注于为湖仓构建图分析（*OLAP*）引擎。具体而言，我们关注带标签属性图（LPG）[20, 34, 48]，因为它们已广泛应用于欺诈检测、实体解析、网络安全和客户 360 等现实场景。在这些属性图中，数据被建模为由带标签顶点和边构成、具有属性的网络。属性图系统的示例包括 TigerGraph [25], Neo4j [13], Ama- zon Neptune [14], NebulaGraph [12], KÙZU [31], Google Spanner Graph [37], 和 IBM Db2 Graph [49]。

目标。 构建湖仓 [21, 22, 36, 53] 数据系统有两个主要目标：(1) 缩短启动时间 和 (2) 缩短查询时间。缩短启动时间可确保系统能够快速配置计算资源，并开始处理查询，而不会因冷启动而长时间延迟。这对于云环境尤为重要，因为计算资源可能是无服务器的，或具有弹性，以应对动态变化的查询工作负载。相比之下，缩短查询时间侧重于最小化

运行时执行延迟，这对性能敏感型应用至关重要。

现有方法及其局限。在湖仓表上执行图分析的现有方法（图1）要么启动时间长，要么查询时间长。一种直接的解决方案是从湖仓中提取数据，并将其加载到原生图数据管理系统中，例如 TigerGraph [25],Neo4j [13], 或 Amazon Neptune [14], , 然后在该系统中执行图分析。尽管使用外部图数据库可能带来较高性能，但这样做会背离湖仓设计理念，即计算与共享存储解耦。将数据迁移到独立的数据孤岛会因冗余 ETL 流水线和不断攀升的存储成本而造成显著开销。这种对数据重复的依赖不仅带来同步风险，还从根本上削弱了湖仓价值主张：让各种分析引擎能够基于单一、权威的数据源运行，而无需维护多个物理副本。

第二种方法是原位图分析，它将湖仓表视为分层或远程存储层，并按需将数据加载到无状态计算引擎中。从概念上讲，这类似于基于磁盘的系统，只不过底层存储位于湖仓中，而不是本地磁盘上。据我们所知，只有 PuppyGraph [15] 属于这一类，支持湖仓表。由于减少了预先加载和转换数据的工作，这种方法的启动时间较短。然而，在查询执行期间，存储与计算之间需要大量数据移动，因此查询性能较差。此外，PuppyGraph 并非开源，其系统设计和优化技术的许多方面都未公开。

挑战。构建一个同时实现低启动时间和低查询时间的湖仓图分析系统并非易事。尽管先前关于为湖仓构建关系型 OLAP 系统的工作（例如 Photon [22], BigLake [36], 和 Presto [45]）提供了宝贵见解，但由于关系型系统与图数据系统之间存在根本差异，这些技术无法直接应用于图系统。

此外，分布式图处理引入了复杂的分片与重分布策略（例如基于顶点的分片或基于边的分片），这比关系型分析中的策略更具挑战性。

最后，湖仓采用基于表的数据模型，这与图数据模型存在根本差异。直接在表上执行图计算会产生昂贵的连接操作，凸显了对新系统设计的需求。

GraphLake 概述。在本文中，我们介绍 GraphLake，一个专为湖仓表分析而构建的图计算引擎。它构建于 TigerGraph [25], 这一原生商业图数据库之上。它将湖仓表映射为图中的顶点类型和边类型，并通过 GSQL [26], TigerGraph 的图查询语言无缝支持图分析。

为缩短启动时间，我们在第4 节提出了一种新方法，将图拓扑与图元素属性分离，并在启动期间仅加载拓扑。我们还介绍一种新的数据结构——边列表，用于表示图拓扑。该数据结构支持快速构建，并为高效图分析奠定基础。我们还引入了转换后的顶点 ID，以支持高效的属性检索。

为确保查询效率，我们提出一种图感知缓存机制（第5节），以高效访问湖仓表中的图元素属性。我们还引入了两种湖仓优化原语（第6节），它们利用边列表和图感知缓存单元，在继承自 TigerGraph 的计算框架下，高效支持对湖仓表的图分析。

实验概述。 针对 LDBC 基准测试 [30,46] 的实验表明，GraphLake 的表现优于 PuppyGraph，启动时间和查询时间均显著更低。具体而言，与 PuppyGraph 相比，GraphLake 的启动速度最高可快 **26.3**× 倍，查询时间最高可快 **60.3**× 倍。

贡献。 本文是首项系统探讨如何为湖仓构建轻量且高效的图计算引擎的工作。我们识别了该场景下的关键挑战，并提出一套新技术加以应对。我们设计并实现了 GraphLake，这是一种与 TigerGraph 集成的新型图计算引擎，能够以 Apache Iceberg 作为代表性湖仓表格式，直接在湖仓表上高效开展图分析。我们相信，本文提出的技术在其他图数据管理系统基于湖仓表运行时也具有广泛适用性。

2 背景

2.1 湖仓架构

湖仓系统是构建在低成本云对象存储（又称数据湖）之上的数据管理平台。它们通过结合数据湖的成本效益与数据仓库强大的管理功能，实现数据湖与数据仓库的统一 [53]。现代湖仓依赖开放表格式（例如 Apache Iceberg [4], 、Delta Lake [21], 和 Apache Hudi [3]），这些格式将数据以不可变文件的形式存储在基于云的数据湖中（例如 Amazon S3、Google Cloud Storage），采用 Parquet、ORC 或 Avro 等格式。这些系统采用元数据驱动架构：更新作为元数据层中的逻辑变更执行，而底层数据文件保持不变。该层由目录（例如 Hive Metastore、AWS Glue）管理的一系列分层元数据文件构成，可确保原子更新，从而支持 ACID 事务和模式演进等管理功能。

湖仓主要使用列式开放文件格式（如 Parquet 和 ORC）来进行表格数据存储。我们以 Parquet 为例进行说明：Parquet 文件包含一个用于定义数据类型的架构，并在水平方向上划分为多个行组。在每个行组内，特定列的值会聚合到一个列块中，列块是扫描的基本单位。每个列块都会单独进行编码（例如使用游程编码或字典编码），并以连续页序列的形式存储。虽然各页支持独立解压缩和解码，但它们通常依赖列块中的辅助数据（例如用于字典解码的字典页）。最后，文件级元数据（包括行数和字节偏移量）存储在文件尾中。

2.2 TigerGraph 中的图分析与 GSQL

图分析，也称为图OLAP，是指涉及以下内容的图分析型工作负载：(1) 计算顶点和边属性的聚合度量 [24], ，以及 (2) 分析图的结构和连通性，通常通过迭代式图算法实现 [23, 48]。图分析广泛应用于社交网络、推荐、安全和欺诈检测等领域；在这些场景中，提取洞察需要同时理解全局结构和聚合属性。例如，PageRank 等图算法可以识别具有影响力的用户，而聚合查询（例如计算每个人创建的评论平均数量）则能揭示大型子图中的高层次趋势。与通常面向交互式图查询的图联机事务处理相比，图分析的特点是运行时间长且计算密集，通常需要遍历图的大部分区域。

TigerGraph [25] 是一种分布式原生图数据库，针对大规模图分析进行了优化。它支持基于集群的部署，数据会自动划分为分片，每个分片包含固定数量的顶点，并作为并行和分布式计算的基本单元。TigerGraph 使用压缩稀疏行（CSR） [7], 表示图拓扑，其中源顶点的出边与其所在的分片共置。TigerGraph 采用大规模并行处理架构，实现高性能图分析。两个关键的并行原语 *VertexMap* 和*EdgeMap*支持用户定义函数（用户定义函数）在顶点和边上执行。

TigerGraph [25] 采用 GSQL [26], ，这是一种为图分析设计的图灵完备语言。GSQL 查询由一系列查询块（SELECT-FROM-WHERE）组成，每个查询块都会生成一个顶点集变量。这些查询块基于以顶点为中心的计算模型运行：每个查询块处理一个输入顶点集，并遍历关联的边，生成供下一查询块使用的后续顶点集。FROM 子句允许这些查询块引用前序查询块中的变量，从而便于组合复杂查询。此外，GSQL 支持 UNION、交集和差集等集合运算，以操作这些顶点集变量。

累加器 [26]是处理复杂图的重要工具
GSQL 中的分析。这些运行时变量在整个查询生命周期内都可变，因此能够在图遍历期间并行更新复杂指标（例如 PageRank 分数或社区 ID）。通过将累加器与 WHILE 和 IF-ELSE 等控制流原语结合使用，GSQL 能够在遍历期间执行复杂的属性聚合和结构分析。

3 系统设计概述

在本节中，我们将介绍 GraphLake 的系统架构。如图2所示，GraphLake 是基于 TigerGraph 构建的分布式计算引擎，可在湖仓表（例如 Apache Iceberg）上执行原生图分析。该系统运行在计算节点上，并与湖仓的底层存储层交互。它在湖仓表之上提供图抽象，使用户能够构建图拓扑索引并管理表缓存，从而执行高性能分析型查询。

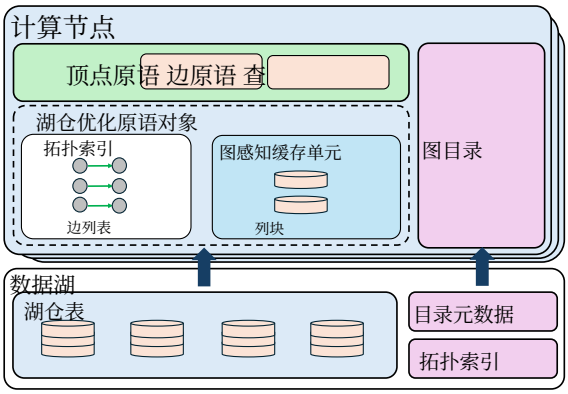


图 2：系统概述

数据模型。GraphLake 是构建于 TigerGraph 之上的图计算引擎，但其 图表示源自湖仓风格的关系表。继承 GQ-Fast 等先前研究的做法， [38], 图可以自然地使用关系表表示。具体而言，表分为两种类型：实体（顶点）表和关系（边）表。每个实体表都包含一个主键列，而每个关系表都包含两个外键列，分别引用实体表的主键。这些表共同定义了图结构。实体表中的元组对应顶点，关系表中的元组对应边；两类表中的列分别表示顶点和边的属性。不同类型的顶点和边可以存储在不同的表中。由于 TigerGraph 强制采用严格的类型系统，其中每个顶点或边只能具有一种类型，因此这种设计极大地促进了从湖仓表到图的准确、高效映射。

图分析计算框架。GraphLake 构建于 TigerGraph 之上，采用基于累加器的聚合范式 [26],，利用多态数据容器（累加器）直接在顶点上存储、更新和持久化计算状态。该框架支持高级图分析，可实现迭代控制流和状态组合，使用户能够通过声明式状态转换表达图聚合查询和复杂算法。GraphLake 使用整体同步并行 (BSP) [50] 模型评估查询，执行过程由活动顶点集（或顶点前沿）驱动，并分多个同步的“超步”进行。为实现弹性横向扩展，GraphLake 采用基于文件的分区方式，将文件分散到不同的计算节点，同时将顶点累加器与湖仓中对应的顶点文件共同存放。

图目录。 为了管理定义在湖仓表之上的图，我们实现了一个图目录模块。该模块维护映射元数据，将元素类型（顶点类型和边类型）与相应的湖仓表关联起来。它还会监控底层湖仓表的变化，例如表文件的添加或删除，以自动更新目录的映射元数据及相关拓扑索引。

仅拓扑启动加载。 为了加快湖仓表的启动速度，我们提出了一种新方法，将图拓扑 与图元素属性（或图内容）解耦

并在启动时仅加载拓扑。我们引入了一种新的数据结构 边列表 来表示图拓扑。这种数据结构能够快速构建，并为高效的图分析奠定基础。我们还引入了变换后的顶点 ID，以支持高效的属性检索。这种解耦的图拓扑设计带来了一个挑战：如何在查询执行期间访问属性数据。我们将在第 5 节和第 6节中解决这一问题。

图感知列式缓存。为提升查询性能，我们实现了一种缓存机制，以列块作为基本缓存单元。我们进一步将其细化为**图感知缓存单元**，通过使数据读取与特定图访问模式相匹配，最大限度地减少解码开销。我们还提出了一种图感知替换算法，用于在内存和磁盘约束下管理缓存分配。为了在图遍历过程中提前预测并加载数据，我们开发了一种由顶点前沿和边列表统计信息驱动的主动预取机制。

面向湖仓优化的并行原语。为在存储解耦的环境中实现计算框架，GraphLake 实现了两个面向湖仓优化的并行原语：*VertexMap* 和*EdgeScan*。这些原语由活跃顶点集驱动，通过 BSP 模型迭代执行。具体而言，*VertexMap* 对活跃集中的顶点应用用户定义函数，而*EdgeScan* 则通过对边列表结构进行并行扫描，对相邻边应用用户定义函数（第4节）。两种原语都使用图感知缓存单元汇总图内容（第5节）。在分布式集群上部署时，*EdgeScan* 采用两遍法解决边与顶点不共置的问题：第一遍识别远程顶点，第二遍则在获取所需顶点数据后执行用户定义函数。

4 仅拓扑启动加载

本节将介绍 GraphLake 如何在降低启动开销的同时实现快速数据加载，并支持对湖仓表进行高效图分析。

总体思路。 我们提出了一种新方法，将图拓扑与图内容解耦，并在启动时仅加载拓扑。图拓扑由顶点表中的主键列和边表中的外键列组成，不包括所有顶点和边的属性（或属性字段）。例如，在图 3中，我们加载

ID 列，来自*Person* 表，以及*Knows* 表中引用*Person*的两个外键列，然后构建图拓扑结构。

这种设计在较低的启动开销与较高的查询性能之间取得了平衡。图拓扑通常只占表总大小的一小部分；例如，LDBC_SNB SF100 数据集 [46]中的顶点表主键列仅占数据总量的 3%，如图4所示。然而，这些拓扑信息已足以通过遍历执行图分析查询。系统会根据具体的查询需求按需加载顶点和边的属性，详情及相关优化将在第5 节和第6节讨论。因此，即使湖仓表包含大量属性集，GraphLake 也能仅通过加载关键列实现高效启动。

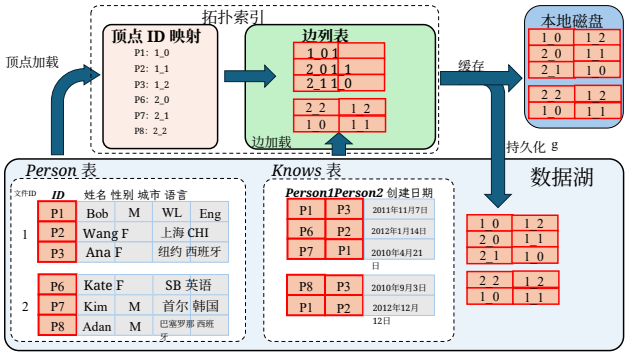


图 3：仅拓扑加载

仅加载拓扑的启动加载设计遵循在 Lakehouse 架构之上构建数据系统的一般理念 [53]，其中图拓扑可视为一种辅助数据结构，用于实现高效的查询处理。

然而，挑战在于如何在湖仓环境中表示图拓扑，尤其是在分布式环境中；我们将在第 4.1 节讨论这一挑战。我们还将在第 4.2 节介绍提升加载过程的优化，并在第 4.3 节介绍完整的加载工作流。

4.1 面向湖仓优化的图拓扑

接下来，我们描述如何在湖仓环境中表示图拓扑。

与其他现代图数据库一样，TigerGraph 在内部使用 CSR 结构来表示图。这是一种高度紧凑、以顶点为中心的布局，其中顶点的出边被连续存储。然而，我们观察到，这种表示并不适合在湖仓环境中实现快速启动，主要有两个原因：（1）构建此类数据结构的代价很高，因为它需要按源顶点对所有边进行分组。在分布式集群中部署时，边必须通过网络发送到多个计算节点，从而显著拖慢启动速度；（2）支持更新的代价很高。例如，当删除某个边文件时，CSR 必须识别被删除边的所有源顶点并更新索引结构，从而移动邻接数组的很大一部分。

在 GraphLake 中，我们使用边列表来表示图拓扑，它只是从 Lakehouse 边表派生的边集合。具体而言，对于给定的边表，我们加载外键列，并构建一组（源，目标）顶点 ID 对作为边列表。因此，边列表中的条目保留了 Lakehouse 边表的原始顺序。这通过保持与湖仓中所存储边属性的行级对齐，便于进行 OLAP 风格扫描，从而实现对边的顺序访问。例如，在计算 *Knows* 边（见图 3）时，边列表条目与相应的 创建日期 列会被同步扫描，从而确保高效的数据处理。

由于湖仓边表在物理上被划分为多个边文件，以便在分布式环境中实现可扩展性和弹性，因此我们为每个边文件构建一个边列表。例如，*Knows*表在图 3中有两个边文件，每个文件对应一个唯一的边列表。

这种设计具有多项优势。首先，构建速度快，因此启动时间短。我们只需从边表下载外键列，并顺序扫描这些列来构建边列表。按文件设计还支持高效的并行构建，因为不同文件的边列表可以并行构建。

其次，它易于更新。由于边列表是按边文件构建的，这种设计支持对边列表进行增量更新。例如，将新的边文件添加到边表时，我们只需为新文件构建边列表，而无需改动其他边列表。

第三，它更适合图 OLAP 查询。这类查询是本文的重点，通常会评估大量边。边列表设计天然适用于以边为中心的处理方式，即遍历每个边列表项以评估相应的边。与基于压缩稀疏行的以顶点为中心的方法相比，它具有更高的 CPU 缓存效率。我们在第 7.6 节的实验表明，当边选择率超过 10% 时，边列表设计的性能优于压缩稀疏行。

最后，它易于分布式部署。我们采用前文提到的基于文件的分区方法。每个计算节点仅为自己的边文件构建边列表，以支持分布式加载。边列表与底层边文件一同分区，从而支持分布式查询，相关内容将在第 6.2 节讨论。

下一个挑战是如何高效地检索顶点和边的属性。由于边列表与底层边表保持行级对齐，因此检索边属性相对直接。然而，检索顶点属性更具挑战性，因为湖仓顶点表会被随机划分到多个顶点文件中。给定一个顶点 ID，需要扫描所有顶点文件才能找到对应记录，因为湖仓顶点表之上没有构建任何索引结构（例如 B 树）。

为应对这一挑战，我们将边列表中的原始顶点 ID 替换为**变换后的顶点 ID**，以优化属性查找。每个变换后的 ID 都是全局唯一的，并编码了文件 ID 和顶点的本地行索引。具体而言，我们将每个变换后的 ID 表示为一个 64 位整数：高 32 位存储唯一的文件 ID，低 32 位存储行索引。通过直接定位文件 ID 和行索引，这种变换后的 ID 可以高效地查找顶点属性。

我们使用一种哈希映射来存储原始顶点 ID 与转换后的 ID 之间的映射，并将其称为顶点 ID 映射（顶点 IDM）。在边列表构建过程中，我们利用顶点 IDM 将原始顶点 ID 转换为转换后的 ID。例如，在图 3 中，*Knows* 表中的第一条边引用了原始 ID P1 和 P3；根据顶点 IDM，它们在边列表中分别转换为 `1_0`（表示文件 ID 1、行索引 0）和 `1_2`（文件 ID 1、行索引 2）。

在分布式环境中，我们会将顶点 IDM 复制到所有计算节点上，因为将原始顶点 ID 转换为转换后的 ID 的延迟，是边列表构建中的一个关键瓶颈。此外，由于顶点数量通常比边的数量少一个数量级，顶点 IDM 的大小通常较小，可以轻松存入每个计算节点的内存中。

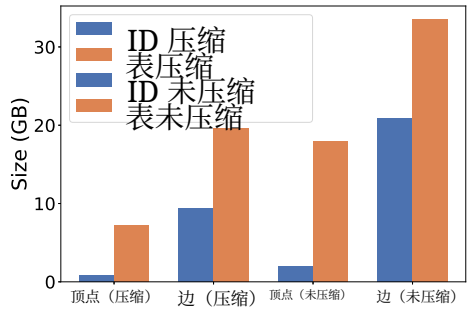


图 4：LDBC_SNB SF100 中 ID 列大小与表大小

4.2 优化

我们还实施了优化，以加快启动加载。

拓扑物化。构建图拓扑后，会将其作为二进制文件持久化存储在数据湖中。如果计算节点重启，GraphLake 会直接加载预存的图拓扑，而无需从头重新构建，从而显著加快启动加载过程。

流水线处理。构建顶点 IDM 和边列表需要从数据湖下载关键列，这一过程涉及多个步骤。我们首先发送 HTTP 请求以获取页脚元数据的长度，随后发送第二个请求以获取页脚元数据本身；最后发送第三个请求，获取构建 IDM 和边列表所需的特定列块。尽管我们使用多个线程并发处理多个文件，但 HTTP 请求的高延迟可能会阻塞后续计算，导致 CPU 空闲。

为了解决这一问题，我们维护了一个 I/O 线程池，负责与本地磁盘和数据湖进行异步 I/O，从而实现流水线工作流：I/O 线程获取列块或写入边列表时，计算线程可并行构建顶点 IDM 以及后续的边列表。这种任务重叠能够有效隐藏 I/O 延迟，并确保整个加载过程中 CPU 利用率保持稳定。

4.3 整体启动加载流程

接下来，我们介绍整体启动加载流程，其中会利用多个线程。

在启动加载期间，GraphLake 首先建立与湖仓和底层数据湖的连接器。它会识别每个顶点表和边表中的所有数据文件，记录它们在数据湖中的文件位置，然后直接与数据湖交互。

顶点 IDM 构建。 在识别所有顶点文件并分配唯一文件 ID 后，GraphLake 启动顶点 IDM（ID 映射）构建。计算线程并行扫描已下载的数据文件主键列。对于每个原始顶点 ID，计算线程根据其行索引和数据文件的文件ID计算其转换后的顶点 ID，然后准备一条由原始 ID 和转换后的 ID 组成的映射条目。这些条目被分组并以批次插入哈希映射，以尽量减少计算线程之间的锁竞争。

边列表构建。 在顶点 IDM 构建之后，GraphLake 继续构建边列表。这些边列表并发构建：每个计算线程处理一个边文件，扫描其外键列，并根据顶点 IDM 将原始顶点 ID 转换为转换后的 ID。由于每个线程操作独立的边文件，主要构建路径是无锁且高效的。

外键列中的原始顶点 ID 通常对应 Vertex IDM 中的条目。不过，在某些边界情况下，我们可能会遇到不属于任何顶点的悬空原始 ID。为处理这些情况，GraphLake 会预留一个由全局原子计数器管理的特殊文件 ID（例如，文件 ID 0）。计算节点遇到悬空原始 ID 时，会向计数器请求一个行索引，以为其生成转换后的 ID。即使顶点是在边表中隐式定义的，这种机制也能确保 GraphLake 实现完整的拓扑覆盖。

边列表构建完成后，Vertex IDM 会被释放以腾出内存。在运行时，引擎会解码转换后的顶点 ID，并通过直接使用文件路径和记录偏移量进行寻址，以高速查找顶点属性。

5 图感知列式缓存

如第4节所述，启动过程中仅加载图拓扑，而图元素属性（包括顶点属性和边属性）则在查询处理期间按需加载。因此，实现高效的缓存机制以降低 I/O 延迟并维持高查询性能至关重要。

尽管缓存策略已在现有基于磁盘的图数据库中得到广泛研究，包括 [35, 44, 51, 54], 其缓存技术仍无法应用于湖仓上的图分析。这些系统通常采用基于页的缓存单元，而 GraphLake 依赖开放格式文件中的列式分块（如下文所述）。此外，现有基于磁盘的图数据库基于以顶点为中心的存储引擎，将同一顶点的所有边归为一组，而 GraphLake 则在边列表及底层列式文件上运行（即以边为中心的设计）。

在 GraphLake 中，我们采用列块作为缓存单元，遵循先前关于湖仓架构上关系数据库的研究 [2, 40, 41]。如第 2.1 节所述，湖仓表中的每个文件被水平划分为多个行组，其中同一列的值被分组为列
列块。因此，基于列块检索属性可以剪除无关列。此外，列块的大小经过优化，适合高效网络传输。

然而，现有的湖仓关系数据库缓存技术 [2, 40, 41] 无法应对图工作负载的独特挑战：（1）如何优化属性访问以适应图访问模式，详见第5.1节；（2）如何在内存或磁盘预算限制下管理缓存淘汰，详见第5.2节；（3）如何预取预计将在未来图遍历中使用的缓存单元，详见第5.3节。

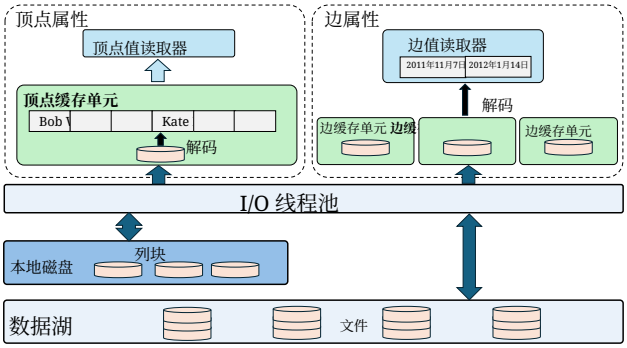


图5：缓存管理

5.1 图感知缓存单元

接下来，我们将介绍如何优化属性访问以适应图访问模式。

湖仓列块针对扫描进行了优化。然而，图分析依赖于利用图拓扑进行邻居遍历，因此顶点属性呈现出不规则的访问模式。在标准 Lakehouse 引擎中，这会导致频繁重新解码列块。此外，图分析通常涉及复杂的过滤谓词，这些谓词依赖于边和相邻顶点的属性（即跨实体谓词）。这种依赖使得必须进行行级求值。尽管 GraphLake 中的边属性采用面向扫描的访问模式，逐一评估这些值仍会产生巨大的解码开销。

为解决这些问题，我们引入了图感知缓存单元。我们缓存列块中的已解码值，并实现专用值读取器，在查询处理过程中通过行索引检索属性值。如图 5 所示，这些单元可分为顶点缓存单元（例如，*Name* 列块，来自 *Person* 顶点表）和边缓存单元（例如，*createDate* 列块，来自*Knows* 边表），各自采用下文详述的不同缓存策略。

顶点缓存单元。 在图遍历过程中，顶点的访问是不规则的，从而对顶点列块产生随机访问请求。为避免重新解码列块，我们为每个顶点列块预先分配一个已解码值数组。该数组提供足够的容量来缓存块内的所有值，从而避免动态调整大小和数据复制带来的性能开销。

已解码值数组初始为空。执行期间，内部解码器按顺序从列块填充该数组，确保数组保持已解码条目的连续前缀。例如，如果值数组当前缓存了前 100 个值，而值读取器请求索引 300 处的条目，则顶点缓存单元会解码从索引 100 到 300 的值并填充中间条目，从而保持前缀的连续性。

我们解码列块，并确保值数组中的数据连续，主要有两个原因：第一，这可以避免重复解码。第二，通过维护递增的索引边界，简化状态管理。

这个数组能够高效地查找顶点属性。例如，如果值读取器请求索引 50 处的值，就可以直接通过数组索引获取。对于不规则顶点访问模式，这项功能十分重要。

对于字符串等可变长度数据类型，我们为内容分配独立的内存块，并仅在值数组中存储相应的偏移量和长度。

边缓存单元。 图分析查询中的跨实体谓词需要对边属性进行行级求值。为降低摊销解码开销，我们按批次解码列块中的值，以改善 CPU缓存局部性。

值读取器会维护一个临时缓冲区，在边属性扫描期间缓存值。例如，如果读取器需要索引 50 处的值，它会解码索引 40 到 60 处的值，将其缓存到临时缓冲区中，然后读取索引 50 处的值。后续请求中，如果读取器需要索引 55 处的值，它会直接从缓冲区中读取，从而降低摊销解码开销。如果请求的索引超出缓冲范围（例如索引 100），读取器会前进到相应页面，按批次解码，并更新缓冲区以纳入所需条目。

在边缓存单元中采用解码值数组设计并不值得。第 7.6.2 节中的实验表明，对于大多数 OLAP 查询，这种方法带来的性能提升有限，却会因边的数量庞大而产生显著的内存开销。因此，在边扫描期间采用滑动窗口设计最能提升 OLAP工作负载效率。

5.2 基于优先级的缓存替换

如图 5 所示，我们采用双层缓存架构。我们将内存层作为更大的基于磁盘的缓存层中的高性能子集，以平衡访问延迟与存储容量。如果边缓存单元从内存中被逐出，则将其丢弃，因为底层列块仍保留在本地磁盘上。相比之下，如果顶点缓存单元从内存中被逐出，则会将其已解码值刷新到磁盘，以避免后续检索时重复解码。从本地存储中逐出的单元会被直接删除。我们不会将解码后的顶点属性持久化回数据湖，因为这会产生不必要的网络开销，并造成数据湖上的存储冗余。

接下来，我们详细介绍在内存和磁盘容量约束下的缓存驱逐策略。SQL 引擎通常会一视同仁地处理列块，但它们不会区分 GraphLake 中的顶点缓存单元和边缓存单元。首先，顶点属性经常被访问，因为顶点是主要的访问锚点；而在 GraphLake 中，边属性在每次遍历中仅访问一次。其次，淘汰边缓存单元的开销较小，因为可以直接将其丢弃；而淘汰顶点缓存单元的开销更高，因为必须先将已解码值刷新到磁盘。

为实现这种差异化驱逐策略，我们采用了一种优先级感知扫描时钟算法（受 PostgreSQL 启发），以较低开销近似实现最近最少使用。缓存单元通过循环缓冲区组织，"时钟指针"在其中迭代。访问某个单元时，其使用计数会重置为特定的优先级值：例如，顶点单元被赋值为 3，而边单元被赋值为 1。在每次扫描期间，时钟指针都会递减所引用单元的使用计数。如果计数降至零，且该单元未被活动进程固定，就会将其驱逐，以容纳新数据。这种加权机制会自然地优先保留顶点单元，因为对其重新解码的成本更高，也更有可能被重复使用。

5.3 预取

接下来，我们介绍图遍历期间缓存单元的预取策略。我们利用一个异步 I/O 线程池，从本地磁盘或远程数据湖中获取缓存单元。然而，要确定最适合预取的单元仍然很有挑战性。

传统 SQL 引擎通常依赖顺序预取，即提前将后续列块加载到内存中。然而，这种方法不适用于顶点缓存单元，因为它们的访问模式不规则，无法通过简单的“下一个块”启发式来预测。虽然边缓存单元按顺序扫描，符合“下一个块”启发式，但预取所有边缓存单元仍会产生高昂的 I/O 成本。

因此，我们提出了一种预取机制，在图遍历过程中利用顶点前沿和边列表统计信息来预测数据需求。

基于顶点前沿的预取。GraphLake 通过迭代扩展顶点前沿来执行图查询。顶点前沿是其边将被遍历的顶点集（见第 6 节）。我们将顶点前沿用作预取顶点缓存单元的第一个运行时信号。

对于每个顶点文件，我们确定当前顶点前沿所涵盖的最小—最大 ID 范围。随后，我们遍历该顶点文件中的行组，并将每个行组的 ID 范围与顶点前沿的最小—最大值进行比较。一旦检测到重叠，我们就预取查询所需的所有列的缓存单元。

基于边列表统计信息的预取。无选择性地预取边缓存单元会产生高昂的 I/O 成本。然而，由于每条边的端点都可能落在顶点前沿内，因此很难剪枝这些边缓存单元。

我们利用边列表的统计信息来剪枝边缓存单元。边列表在逻辑上按边文件的行组索引范围划分为多个分区。在构建边列表时，我们会计算每个边列表分区的最小和最大源 ID。随后，我们将这些 ID 范围与顶点前沿的最小—最大 ID 范围进行比较。如果未检测到重叠，就剪枝相应的边列表分区及其对应的列块。当底层边表按指向源顶点的外键排序时，这种最小—最大值剪枝最为有效。

6 湖仓优化图分析

在本节中，我们将介绍 GraphLake 如何利用图拓扑（见第 4 节）和缓存的图属性（见第 5 节）对湖仓表执行图分析。

与 TigerGraph 的计算框架相一致， [25],GraphLake 采用 基于累加器的聚合范式，在 BSP 模型中通过声明式顶点状态转换执行图分析。在该模型中，执行严格由活动顶点集驱动。在每个超步中，引擎执行以下二者之一

一个 *VertexMap* 或一个 *EdgeMap* 原语。*VertexMap* 将用户定义函数应用于当前活跃集，并可访问静态顶点属性和运行时累加器；*EdgeMap* 将用户定义函数应用于入射于活跃顶点的边。在边处理期间，引擎读取一条边及其相邻顶点（或端点）的属性和累加器，然后更新两个端点的累加器。该模型在各步骤之间强制执行严格同步，其中后续迭代的活跃集会被动态确定：它可以由 *VertexMap* 之后经过筛选的顶点子集组成，也可以由在 *EdgeMap* 期间处理的经过筛选的边集的源（或目标）端点组成。

然而，在 TigerGraph 中执行这些原语依赖于专有存储层。它使用专有的 CSR 结构存储图拓扑和图内容。相比之下，湖仓表采用开放格式，使存储层无需由查询引擎直接管理。

为应对这一挑战，GraphLake 针对湖仓架构调整了 *VertexMap* 和 *EdgeMap* 原语。我们将 *EdgeMap* 更名为 *EdgeScan*，以强调其基于扫描的执行方式。这两种湖仓优化原语与 GSQL 查询无缝集成，并提供与原始 *VertexMap* 和 *EdgeMap* 相同的语义。

接下来，我们将在第 6.1 节说明如何高效执行这两种原语。随后，我们将在第 6.2 节说明它们在分布式环境中的执行方式。

我们以 LDBC_SNB BI [46], 中的一个代表性图OLAP查询为例，介绍 GraphLake 原语。LDBC_SNB BI 是一个知名的图分析基准测试。该查询查找在 2010-01-01 之后创建过带有“Music”标签评论的女性，并汇总每位此类用户创建的评论数。下面展示的 GSQL 查询在 FROM 子句中指定了模式：括号内是顶点类型（*Person* 和 *Comment*），方括号内是边类型（*HasTag* 和 *HasCreator*）。每个人创建的评论数存储在累加器 *@sum* 中。

```
选择 p 来自 (t:标签) <- [e1:HasTag] -(c:Comment)
```

6.1 面向湖仓优化的并行原语

VertexMap. 此原语接收一个活动顶点集以及一个用于顶点输入的用户定义函数。活动顶点集是高度压缩的位图，按顶点文件进行分段。该原语的输出是通过过滤的顶点子集。例如，示例查询中的一个 *VertexMap* 会根据标签 顶点的 *Name* 属性进行过滤，并输出满足谓词 *t.name ==“Music”* 的顶点。

VertexMap 会跨多个顶点文件并发评估顶点。它会为所有涉及的顶点列创建值读取器，并为包含活跃顶点的顶点文件分配线程任务。对于每个顶点文件，它会遍历该顶点文件的位图，并使用变换后的顶点 ID 为值读取器提供数据。如第 5.1 节所述，这些值读取器可以从

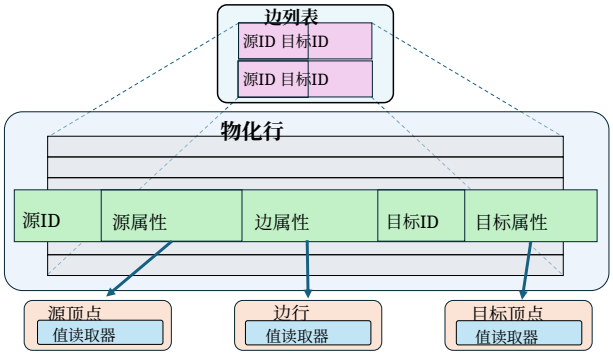


图 6：顶点与边行物化

缓存单元中获取属性值，文件路径和行索引可从我们变换后的顶点 ID 中解码得到。当值读取器成功从缓存单元中获取顶点属性后，一个顶点行便完成了实例化。然后 *VertexMap* 会对该顶点应用用户定义函数。

EdgeScan。该原语接收一个源顶点集，以及分别用于源顶点、相邻边和目标顶点的用户定义函数。执行后，它会输出经过过滤的边所对应的源顶点集或目标顶点集。*EdgeScan* 利用**以边为中心的处理方法**，借助边列表和湖仓缓存单元，而不是采用 TigerGraph 使用的以顶点为中心的方法。

具体而言，*EdgeScan* 会并发扫描边列表，并为活动顶点集及其端点所关联的每条边应用用户定义函数。它首先针对查询中涉及的顶点/边属性（例如示例中的 *date* 和 *HasCreator*，包括投影或过滤的属性）创建值读取器。然后 *EdgeScan* 使用多个线程扫描边列表，并为每个边列表创建线程任务。

系统会为一个线程分配边列表，并由该线程对其进行迭代扫描。对于每个条目，系统都会验证源顶点 ID 是否存在于输入集中。匹配成功后，系统会从边条目中解码两个端点的文件路径和行索引，同时根据偏移量计算边的位置。随后，系统将这些坐标传递给值读取器，以检索两个顶点及边本身的属性值。因此，对于这些边，我们会将邻接顶点行和边行完全物化，因为所有值读取器都会从缓存单元中读取属性，如图 6所示。然后，我们对这些边行和顶点行应用用户定义函数，计算过滤谓词并更新累加器。例如，示例查询包含两个用于两跳图遍历的 *EdgeScan* 原语。第二个 *EdgeScan* 会计算 *e2.date > date (’ 2010-01-01’)* 这一过滤谓词 *WHERE* 子句中的内容，并针对每条边及其端点更新 *p.@sum* 这一 ACCUM 子句中的累加器。引擎会将这些累加器更新发送到各端点，并在下一次迭代之前合并这些更新。

我们还会在扫描边列表时尽可能进行剪枝。例如，根据指定的边类型剪除无关的边列表，这些边类型决定了需要考虑的边表和底层边文件。此外，正如第 5.3 节所述，我们还会将边列表的顶点 ID 最小值和最大值范围与顶点前沿进行比较，以此剪枝。

对于需要遍历入边而非出边的查询，*EdgeScan* 支持双向遍历，且无需

额外的存储成本。遍历出边时，我们将边列表项中的第一个顶点 ID 视为源顶点，第二个视为目标顶点。遍历入边时，则将第一个 ID 视为目标顶点，第二个视为源顶点。相比之下，基于压缩稀疏行（CSR）的引擎（如 TigerGraph）通常需要存储两份边才能支持双向遍历。

6.2 分布式查询处理

接下来，我们介绍如何高效执行这两个新原语，以评估分布式图查询。我们主要关注*EdgeScan*，因为*VertexMap*的分布式执行很简单：每个计算节点处理其所属的顶点。

如第4.1节所述，我们采用基于文件的分片方法，将顶点文件和边文件随机分配到不同的计算节点。然而，这种方法会导致EdgeScan的分布式执行出现一个问题：边可能不会与其端点位于同一节点。

有两种简单的解决方案。第一种是直接访问远程数据。每当*EdgeScan*处理一条边并遇到远程端点时，就直接向远程计算节点请求顶点属性。然而，这种方案会触发大量请求，并因网络延迟而成为*EdgeScan*的瓶颈。第二种方案是允许所有计算节点访问顶点文件，以此复制顶点。然而，这种方案会增加内存负担，并导致顶点缓存单元重复解码。

请注意，我们在第 4.1 节复制了VertexIDM，因为它相对较小，且能显著改善边列表的构建。这里我们不复制顶点属性，因为批量请求后，网络延迟是可以接受的。

因此，我们提出一种两遍扫描方法：计算节点仍会请求远程顶点数据，以减轻内存和解码负担，但会将这些请求批量处理，从而尽可能降低总体延迟。在第一遍中，每个计算节点扫描本地边列表，以确定哪些顶点行需要物化但位于远程节点上，例如图 7 的示例中的 P1 和 P6。随后，它收集所有远程数据请求，并将它们作为一个批次发送。远程节点会先过滤顶点数据，然后按照经典过滤器下推技术的方式返回数据（例如，图 7 中的 P7 会被谓词 *p.gender == "Female"*过滤掉）。活动顶点集会刷新，只记录来自远程节点且已通过过滤的顶点。在第二遍中，每个计算节点对

将顶点和边行物化，如第 6.1 节所述。最后，顶点累加器的部分更新会被推送回主机，以合并为最终值。

7 实验

本节将介绍实验结果，以评估 GraphLake。

7.1 实验设置

实验平台。 我们在实验中使用 Apache Iceberg [4],，这是最受欢迎的湖仓系统之一。我们将 Iceberg 表存储在 AWS S3 的 us-east-2 区域。默认情况下，我们在一台 Linux 服务器上进行实验，该服务器配备 40 核英特尔至强处理器（2.30 GHz）、64 GB 动态随机存取存储器和 1.8 TB NVMe 固态硬盘，网络至 AWS S3 的吞吐量为 1.1 GB/s。分布式实验则

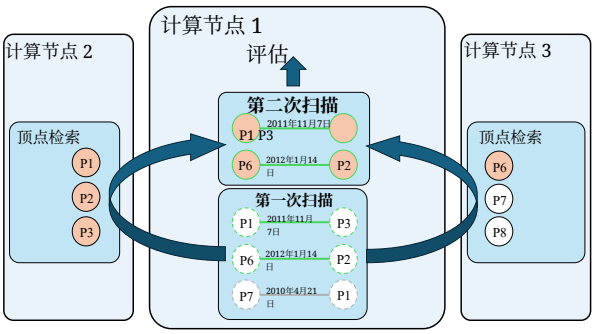


图7：分布式*EdgeScan*表1：LDBC SNB 统计数据_

规模因子	# 顶点数	# 边数	大小（GB）
SF1	2,997,352	17,196,776	0.28
SF3	8,513,157	51,035,227	0.80
SF10	27,231,349	170,343,945	2.62
SF30	78,244,709	505,722,361	7.79
SF100	254,634,696	1 703 042 944	26.76
SF300	738,162,716	5 078 844 191	79.94
SF1000	2,433,117,531	17,203,259,133	278.46

使用同一区域中的 AWS EC2 实例进行（Intel Xeon Platinum 8175M、32 个 vCPU、128 GB 内存和 64 MB L3 缓存）。

数据集。 我们主要使用 LDBC_SNB BI [46], 这一图分析标准工作负载进行实验。它是一个社交网络图，由顶点和边组成，例如 *Comment*、*Person*和*Comment_HasCreator_Person*。该工作负载包含不同规模的数据集，其大小随规模因子（SF）成比例增长，如表 1所示。例如，SF1 包含 300 万个顶点和 1700 万条边，而 SF30 总计包含 7800 万个顶点和 5.05 亿条边。在实验中，单机测试使用 SF1 至 SF300，可扩展性测试使用 SF30 至 SF1000。我们对这些数据集进行预处理，将其加载到湖仓表中，其中每种顶点或边类型对应一个表。为尽可能准确地评估 GraphLake 的并发能力，我们将每个表拆分为 32 个数据文件，因为 32 个 vCPU 是许多计算节点的常见配置。当 SF 较大时，某些表的文件可能过大。在这种情况下，我们会将表拆分为更多文件，以确保每个文件大小在 128 MB 至 256 MB 之间，这是 AWS 指南推荐的数据大小范围 [16]。

由于图算法也是图分析的重要组成部分，我们使用 LDBCGraph-Analytics [30], ——LDBC 推出的另一个热门图算法基准测试——开展图算法实验。具体而言，我们使用 Graph500-22，这是该基准测试中的一个代表性数据集，包含 240 万个顶点和 6420 万条边。我们评估了 Graph500-22 中的全部五种图算法，包括 PageRank (PR)、弱连通分量 (WCC)、基于标签传播的社区检测 (CDLP)、局部聚类系数 (LCC) 和广度优先搜索 (BFS)。

竞品。我们主要将 GraphLake 与 PuppyGraph [15],进行比较，后者是支持湖仓的最先进图计算引擎。由于 GraphLake 基于 TigerGraph 构建，我们还

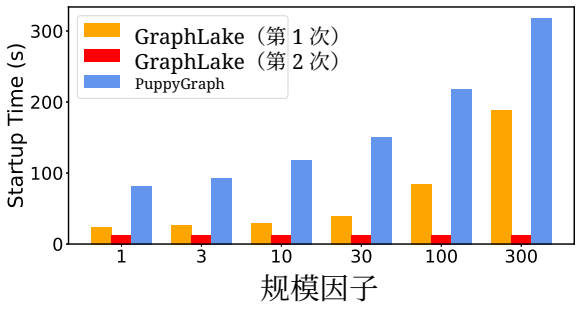


图 8：启动时间评估

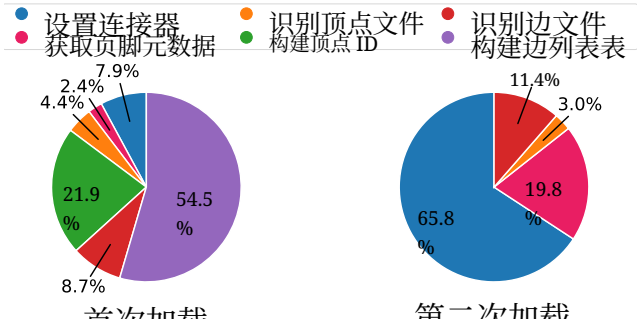


图9：SF100 上的启动时间分解

将其与 TigerGraph 进行比较，以评估支持湖仓时造成的性能损失（第7.6.1节）。

7.2 启动加载时间结果

在本实验中，我们评估 GraphLake 相较于 PuppyGraph 的启动时间。我们区分首次连接的启动时间（此时图拓扑尚未构建）与第二次及后续连接的启动时间（为简洁起见称为“第二次连接”，此时图拓扑已持久化到数据湖中）。

对于 PuppyGraph，由于其并非开源，我们无法像对 GraphLake 那样直接检查引擎状态，因此通过定期发送一个轻量级图查询来测量其启动延迟，直到该查询成功返回结果。该查询是针对标签 和标签类 顶点类型的简单一跳查询，这两种顶点类型的顶点数量都很少。

图 8 展示了启动时间评估的结果。这些结果表明，GraphLake 的启动时间远低于 PuppyGraph。它实现了 **1.7**× 至 **4.0**× 倍更快的首次连接启动，因为 GraphLake 仅加载图拓扑。在将已构建的拓扑持久化到数据湖后，它实现了 **6.9**× 至 **26.3**× 倍更快的第二次及后续连接启动。这些结果表明，GraphLake 的启动速度显著快于 PuppyGraph。

图 9 展示了 GraphLake 在 LDBC SNB SF100 上的启动时间分解。首次连接的分解结果显示_，启动时间主要耗费在图拓扑构建上，其中顶点 IDM 构建占 21.9%，边列表构建占 54.5%。第二次连接（以及后续连接）的分解结果显示，启动时间主要耗费在与湖仓和数据湖建立连接上，所需时间几乎恒定。这是因为在数据湖中持久化边列表后，我们可以跳过图拓扑构建。

7.3 图聚合查询结果

在本实验中，我们评估了 GraphLake 的图聚合查询性能，并将其与 PuppyGraph 进行了比较。我们从标准 LDBC 基准测试中选取了五个具有代表性的查询 [11],该基准测试为 Neo4j 提供 Cypher 查询，为 TigerGraph 提供 GSQL 查询。我们采用了这些查询，但由于 PuppyGraph 不支持部分语法，因此对 Cypher 查询进行了少量修改。我们使用相同的查询参数，以确保比较公平。

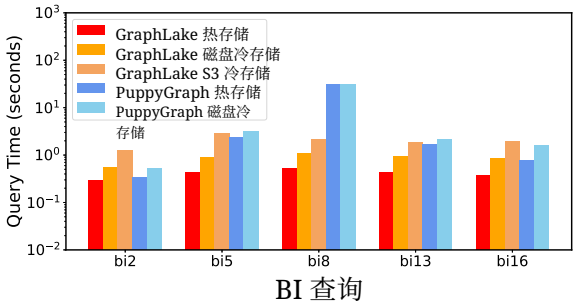


图10：SF30 的查询时间评估

我们测量了热运行时的查询时间，此时所有数据都已预热至内存中。我们还测量了 S3 和磁盘冷运行时的查询时间，这意味着查询开始时，缓存单元和边列表会从 S3 或本地磁盘加载。由于 PuppyGraph 有一个后台进程，会自动将数据从湖仓提取到本地磁盘，因此我们无法提供其 S3 冷运行结果。

图10 和图11 展示了热运行和冷运行的查询时间。对于 SF30 上的实验，我们观察到 GraphLake 在热运行中最多比 PuppyGraph 快**60.3**× 倍，并且在磁盘冷运行中最多快**29.8**× 倍。对于 SF300 上的实验，GraphLake 在热运行中最多比 PuppyGraph 快**12.8**× 倍，并且在磁盘冷运行中最多快**4.4**× 倍。PuppyGraph 因内存溢出（OOM）在 BI8 上失败，但 GraphLake 仍能在合理时间内执行该查询。这是因为我们的引擎直接利用边列表和湖仓原生缓存单元来高效遍历图。

上述结果表明，在不同的图聚合查询中，GraphLake 的表现始终与 PuppyGraph 相当或更优。在多种工作负载下，无论是热运行还是磁盘冷运行，GraphLake 的查询时间都能缩短一个数量级。

7.4 图算法的结果

在本实验中，我们比较了 GraphLake 与 PuppyGraph 执行图算法的运行时间。我们利用两者内置的过程，在 Graph500-22 数据集上以相同的算法参数执行图算法。

如表 2 所示，对于 PR 和 WCC，GraphLake 的速度最高可达 **9.5**× 倍，快于 PuppyGraph。对于 CDLP、BFS 和 LCC，GraphLake 能够在合理时间内执行这些算法，而 PuppyGraph 要么因内存溢出而失败，要么未提供支持该算法的过程。这是因为 GSQL 中的顶点累加器

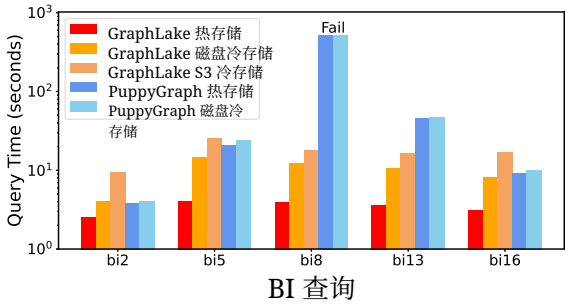


图11：SF300 查询时间评估 表2：图算法运行时间

	PR	WCC	CDLP	LCC	BFS
GraphLake	**28.0秒**	**9.3s**	**117.0秒**	**640.8秒**	**3.5s**
PuppyGraph	59.6秒	88.1秒	OOM	不支持	OOM

能够高效地减少中间结果，而 PuppyGraph 可能会因中间工作表过大而遭遇内存溢出故障。

7.5 可扩展性评估

在本实验中，我们从节点数量和数据集大小（即规模因子）两个方面评估 GraphLake 的可扩展性。由于湖仓文件分布在各个节点上，我们使用吞吐量来评估查询性能，以便每个节点都能充分利用其计算资源。

我们使用一台额外的发送端机器，将请求均匀分发到所有机器，以尽量降低网络端口的影响。发送端机器使用流行的 HTTP 基准测试工具 wrk2 [19]。在该实验中，由于 OLAP 查询运行时间较长，工具保持 16 个连接，并总共启动 2 个线程。每个线程都会准备带有随机查询参数的负载，以尽量降低相同负载对 CPU 缓存造成的影响。每秒发送的请求数足够高，可以使系统吞吐量达到饱和。

图 12 展示了单节点上吞吐量随数据集增大而变化的情况。我们展示了 GraphLake 对两个查询 BI2 和 BI16 的吞吐量。规模因子从 SF1 增至 SF300 时，BI2 的吞吐量从每秒 113.6 个查询降至 0.6 个，数据量增加 **186**× 时，吞吐量下降约 **300**× 。对于 BI16，在相同范围内，吞吐量从每秒 128.8 个查询降至 0.6 个，约下降 **222**× 。两项结果均表明系统具有良好的可扩展性。这是因为扫描边列表这一核心评估步骤，在较大的数据集上能够实现更高的 CPU 缓存效率。

图 13 展示了随着机器节点数量增加、数据集规模扩大，启动时间的变化情况。对于首次启动，我们可以观察到，当节点数较少时，GraphLake 的可扩展性接近线性。例如，从 1 个节点扩展到 2 个节点时，它实现了**1.8**× 到 **1.9**× 针对不同大小的数据集，启动速度均有所提升。这是因为构建边列表可以通过增加节点高效地横向扩展，但仍占据启动时间的大部分。随着节点数量增加，构建

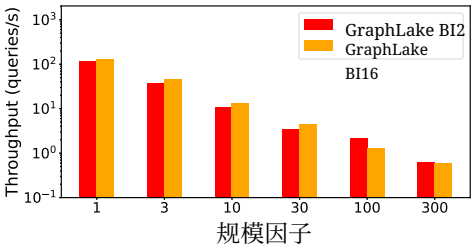


图 12：BI2 和 BI16 的吞吐量评估

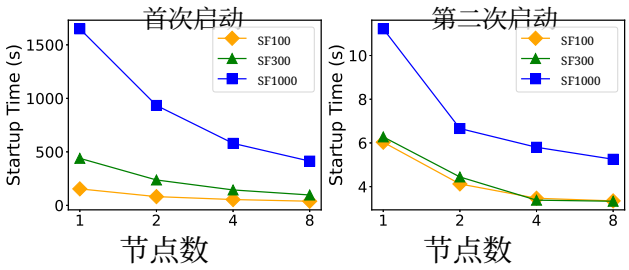


图 13：启动时间可扩展性评估

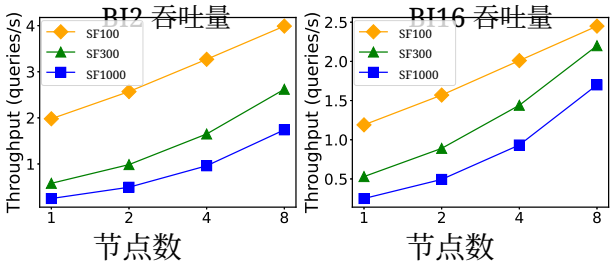


图 14：BI2 和 BI16 的吞吐量可扩展性评估

边列表所需时间相对减少，GraphLake 的可扩展性也随之降低，因为顶点 IDM 构建并未进行分区。第二次启动的可扩展性始终较差，因为大部分时间都花在建立连接上，而这项工作耗时基本恒定。

图 14 展示了两个查询 BI2 和 BI16 的吞吐量如何随着机器节点增多和数据集变大而扩展。我们可以观察到，对于较大的数据集，GraphLake 具有更好的可扩展性。例如，在 SF1000 上，节点数翻倍时，GraphLake 的吞吐量可提升 **1.8**× 至 **1.9**× 倍。然而，在 SF100 上，GraphLake 的吞吐量仅提升 **1.2**× 至 **1.3**× 倍。这是因为较大的数据集会带来相对更多的本地计算。尽管分布式查询评估需要从远程节点获取顶点数据，但与本地边数据的大小相比，这类数据量仍然很小。我们还可以观察到，对于特定的数据集大小，随着节点数增加，可扩展性会变差，因为网络通信变得更加复杂，并产生相对更多的开销。

上述结果表明，GraphLake 在启动加载和查询处理方面均展现出强大的可扩展性。

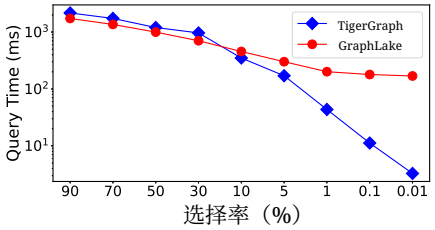


图 15：使用边列表进行以边为中心的处理，与使用 CSR 结构进行以顶点为中心的处理

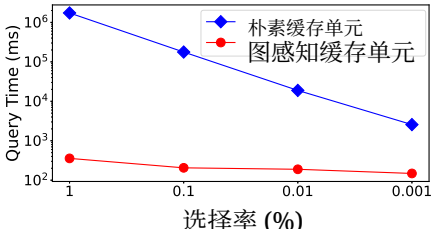


图 16：图感知缓存单元与朴素列块

我们的图感知缓存单元对于不规则的图访问模式非常有效。

7.6 其他实验

接下来，我们开展实验，进一步分析 GraphLake 的性能。

7.6.1 与 *TigerGraph* 的比较。由于 GraphLake 构建于 TigerGraph 之上，因此将 GraphLake 的查询性能与 TigerGraph 进行比较颇有意义；不过，GraphLake 从开放格式的湖仓表中读取属性，而 TigerGraph 使用专有存储，因此预计其性能会较低。

由于这两个系统的主要差异在于原语 *EdgeMap* 和*EdgeS-can*的实现方式，因此我们直接比较不同选择率下 *EdgeMap* 和 *EdgeScan* 的执行情况。具体而言，我们在 LDBC_SNB SF100 数据集上运行一个具有代表性的单跳模式匹配查询；该查询旨在查找所有创建给定评论的 Person。这种模式匹配在 LDBC_SNB BI 查询中很常见。我们测量两个系统中 *EdgeMap* 的执行时间。通过调整输入顶点集的大小来改变选择率。为避免 TigerGraph 的行式存储和 GraphLake 的列式缓存单元所带来的影响，该查询不涉及属性，而是在边上应用计算函数。

从图15的结果中，我们观察到，当选择率大于 10% 时，GraphLake 甚至比 TigerGraph 更快，这是因为顺序扫描边列表能够提供更好的 CPU缓存局部性。然而，当选择率较低时，例如 0.01%，TigerGraph 可以在几毫秒内完成执行，因为它能够按顶点剪枝大量边；而 GraphLake 则需要花费数百毫秒扫描边列表。因此，我们得出结论：GraphLake 最适合边选择率较高的繁重 OLAP工作负载。

7.6.2 评估图感知缓存单元。图感知缓存单元设计会解码顶点列块，并将属性缓存在已解码值数组中；而朴素缓存单元设计则以向量化方式反复扫描湖仓列块。我们希望比较这两种设计之间的差距。

我们使用与第7.6.1节相同的查询，但增加了对顶点属性的过滤。我们仅测量热运行查询时间，以尽量减少 I/O 影响。

根据图16中的结果，我们观察到朴素设计的查询时间非常高，介于 1734.2 秒到 2.5 秒之间，因为顶点列块会被反复解码。相比之下，我们的
图感知设计的查询时间较短，在相同的选择率范围内，从 357.1 毫秒降至 148.2 毫秒。本研究表明，

8 相关工作

基于湖仓的关系分析。 随着 Lakehouse 架构的发展，先前的工作主要关注基于湖仓的关系分析。例如，Databricks 提供原生 SQL 分析，能够实现最先进的 SQL 性能 [9]。许多关系型引擎也将湖仓格式视为一等表数据源 [1, 10, 17, 45]。一些 SQL 数据仓库正在扩展其功能，以支持基于湖仓的关系分析 [2, 33, 36, 40, 41]。然而，这些工作关注的是基于湖仓的关系分析，而我们的工作关注的是基于湖仓的图分析，并优先考虑图拓扑以及边与顶点之间的结构依赖。

关系数据库中的图分析。 另有一类工作研究如何在关系数据库上执行图查询 [29, 32, 39, 47] 。这类系统通常会构建 CSR 风格的图结构，以加速图遍历；一旦 CSR 结构完成物化，这种方法便十分有效。然而，这些系统需要付出不小的构建成本，且采用的查询处理模型不同于 GraphLake；GraphLake 必须直接从湖仓表着手。此外， [39]提出了用于图查询的缓存技术，但假设可以访问元组可寻址关系表，并能通过行 ID 获取属性。然而，湖仓表中的属性经过压缩后以列块形式存储在对象存储中，因此其缓存方案无法直接应用于外部列式文件。Spanner Graph [37] 采用交错存储，将边行与其源顶点行放在一起；但由于我们无法控制湖仓表，这项技术不适用于我们的场景。

值得注意的是，GRFusion [29] 也将图拓扑作为索引，与图元素属性解耦。不过，它面向的是内存环境，而非湖仓环境。此外，与我们的工作不同，它缺乏针对快速启动加载的优化。

基于磁盘的图数据库。 从概念上讲，我们的场景与基于磁盘的图数据库相关 [35, 44, 51, 54], ，因为我们可以将对象存储（以及湖仓）视为磁盘层。不过，这些系统围绕固定大小块和基于页的缓冲区管理器设计，而 GraphLake 处理远程存储中的可变大小对象（例如列块和文件），因此必须针对快速启动加载和高效图分析进行优化。此外，以往的基于磁盘的图引擎通常采用压缩稀疏行（CSR）或类似 CSR 的拓扑表示，将每个顶点与其所有邻居关联起来。相比之下，GraphLake 使用边列表，这些边列表

经过专门优化，可从 Lakehouse 列式数据快速构建，并支持对现有表进行潜在更新。

其他系统。 Apache Spark [52] 是湖仓领域一种著名的分析引擎。尽管其 GraphX [27] 扩展通过基于 RDD 的抽象支持图算法，但它缺乏对带标签属性图模型的原生支持，而这正是我们系统的主要要求。

9 结论

在本文中，我们介绍了 GraphLake，这是一种专门构建的图计算引擎，构建于 TigerGraph 之上，用于支持在湖仓表上进行图分析。GraphLake 引入了一套新颖技术，在保证查询性能的同时缩短启动时间。大量实验表明，与 PuppyGraph 相比，GraphLake 的性能显著更优，启动时间和查询时间都大幅降低。

参考文献

[1] Amazon Athena（https://docs.aws.amazon.com/athena/）。[2] [n。d.]。Amazon Redshift：借助 SQL 为您的数据湖仓提供大规模、无与伦比的性价比（https://aws.amazon.com/redshift/）。[3] [n。d.]。Apache Hudi（https://hudi.apache.org/）。[4] [n。d.]。Apache Iceberg（https://iceberg.apache.org/）。[5] [n。d.]。Apache ORC（https://orc.apache.org）。[6] [n。d.]。Apache Parquet（https://parquet.apache.org）。[7] [n。d.]。压缩稀疏行（https://en.wikipedia.org/wiki/Sparse_matrix）。[8] [n。d.]。Delta Lake（https://delta.io/）。[9] [n。d.]。DeltaEngine（https://databricks.com/blog/2020/06/24/introducing-delta- engine.html）。[10] [n。d.]。Dremio（https://www.dremio.com/）。[11] [n。d.]。LDBC_SNB_BI_Github（https://github.com/ldbc/ldbc_snb_bi）。[12] [n。d.]。NebulaGraph（https://www.nebula-graph.io/）。[13] [n。d.]。Neo4j（https://neo4j.com/）。[14] [n。d.]。Neptune（https://aws.amazon.com/neptune/）。[15] [n。d.]。PuppyGraph（https://www.puppygraph.com/）。[16] [n.d.]。S3 指南（https://docs.aws.amazon.com/prescriptive-guidance/latest/apache-iceberg-on-aws/best-practices-read.html）。[17] [n。d.]。Trino（https://trino.io/）。[18] [n。d.]。Microsoft Fabric 中的湖仓是什么？（https://learn.microsoft.com/en- us/fabric/data-engineering/lakehouse-overview）。[19] [n。d.]。wrk2（https://github.com/giltene/wrk2）。[20] Renzo Angles、Angela Bonifati、Stefania Dumbrava、George Fletcher、Alastair Green、Jan Hidders、Bei Li、Leonid Libkin、Victor Marsault、Wim Martens、Filip Murlak、Stefan Plantikow、Ognjen Savkovic、Michael Schmidt、Juan Sequeda、Slawek Staworko、Dominik Tomaszuk、Hannes Voigt、Domagoj Vrgoc、Mingxi Wu 和 Dusan Zivkovic。2023。PG-Schema：属性图的模式。Proc. ACM Manag. Data 1, 2（2023），198:1–198:25。[21] Michael Armbrust、Tathagata Das、Sameer Paranjpye、Reynold Xin、Shixiong Zhu、Ali Ghodsi、Burak Yavuz、Mukul Murthy、Joseph Torres、Liwen Sun、Peter Boncz、Mostafa Mokhtar、Herman Van Hovell、Adrian Ionescu、Alicja Luszczak、Michal Switakowski、Takuya Ueshin、Xiao Li、Michal Szafranski、Pieter Senster 和 Matei Zaharia。2020。Delta Lake：云对象存储上的高性能 ACID 表存储。VLDB基金会会刊 13, 12（2020），3411–3424。[22] Alexander Behm、Shoumik Palkar、Utkarsh Agarwal、Timothy Armstrong、David Cashman、Ankur Dave、Todd Greenstein、Shant Hovsepian、Ryan Johnson、Arvind Sai Krishnan、Paul Leventis、Ala Luszczak、Prashanth Menon、Mostafa Mokhtar、Gene Pang、Sameer Paranjpye、Greg Rahn、Bart Samwel、Tom van Bus-sel、Herman Van Hovell、Maryann Xue、Reynold Xin 和 Matei Zaharia。2022。Photon：湖仓系统的快速查询引擎。发表于 SIGMOD。2326–2339。[23] Angela Bonifati、M. Tamer Özsu、Yuanyuan Tian、Hannes Voigt、Wenyuan Yu 和 Wenjie Zhang。2024。图分析路线图。SIGMOD Rec. 53, 4（2024），43–51。[24] Chen Chen、Xifeng Yan、Feida Zhu、Jiawei Han 和 Philip S. Yu。2009。图 OLAP：用于图数据分析的多维框架。Knowl. Inf. Syst. 21, 1（2009），41–63。[25] Alin Deutsch、Yu Xu、Mingxi Wu 和 Victor E. Lee。2019。TigerGraph：原生 MPP 图数据库。CoRR abs/1901.08248（2019）。[26] Alin Deutsch、Yu Xu、Mingxi Wu 和 Victor E. Lee。2020。TigerGraph 对现代图分析的聚合支持。发表于 ACM 国际数据管理会议（SIGMOD）。377–392。[27] Joseph E. Gonzalez、Reynold S. Xin、Ankur Dave、Daniel Crankshaw、Michael J. Franklin 和 Ion Stoica。2014。GraphX：分布式环境中的图处理数据流框架。在第 11 届 USENIX 操作系统设计与实现研讨会（OSDI '14）上，美国科罗拉多州布鲁姆菲尔德，2014年10月6日至8日。599–613。[28] Rihan Hai、Christos Koutras、Christoph Quix 和 Matthias Jarke。2023。数据湖：功能与系统综述。IEEE 知识与数据工程汇刊 35，12（2023），12571–12590。[29] Mohamed S. Hassan、Tatiana Kuznetsova、Hyun Chai Jeong、Walid G. Aref 和 Mohammad Sadoghi。2018。GRFusion：将图作为一等公民纳入主内存关系数据库系统。在 2018 年国际数据管理会议论文集（2018年 SIGMOD 会议），美国得克萨斯州休斯敦，2018年6月10日至15日。1789–1792。[30] Alexandru Iosup、Tim Hegeman、Wing Lung Ngai、Stijn Heldens、Arnau Prat-Pérez、Thomas Manhardt、Hassan Chafi、Mihai Capota、Narayanan Sundaram、Michael J. Anderson、Ilie Gabriel Tanase、Yinglong Xia、Lifeng Nai 和 Peter Boncz。2016。LDBC Graphalytics：面向并行与分布式平台的大规模图分析基准测试。VLDB基金会会刊 9，13（2016），1317–1328。[31] Guodong Jin、Xiyang Feng、Ziyi Chen、Chang Liu 和 Semih Salihoglu。2023。KÙZU 图数据库管理系统。在 CIDR。[32] Guodong Jin 和 Semih Salihoglu。2022。通过预定义连接提升关系数据库管理系统处理图工作负载的效率。VLDB基金会会刊 15，5（2022），1011–1023。[33] Sandya Kola。2025。Snowflake 通过增强开放数据访问和灵活性，为智能体人工智能打造企业级湖仓（https://www.snowflake.com/en/news/press-releases/snowflake-delivers-the-enterprise-lakehouse-with-enhanced-open--data-access-and-flexibility-for-agentic-ai/）。[34] Haridimos Kondylakis、Stefania Dumbrava、Matteo Lissandrini、Nikolay Yakovets、Angela Bonifati、Vasilis Efthymiou、George Fletcher、Dimitris Plexousakis、Riccardo Tommasini、Georgia Troullinou 和 Elisjana Ymeralli。2025。属性图标准：现状与开放挑战。VLDB基金会会刊 18，12（2025），5477–5481。[35] Aapo Kyrola、Guy E. Blelloch 和 Carlos Guestrin。2012。GraphChi：仅用一台个人计算机进行大规模图计算。在第 10 届 USENIX 操作系统设计与实现研讨会（OSDI 2012），10，2012 8 31 46，美国加利福尼亚州好莱坞，10月 - 。– 。[36] Justin J. Levandoski、Garrett Casto、Mingge Deng、Rushabh Desai、Pavan Edara、Thibaud Hottelier、Amir Hormati、Anoop Johnson、Jeff Johnson、Dawid Kurzyniec、Sam McVeety、Prem Ramanathan、Gaurav Saxena、Vidya Shanmugan 和 Yuri Volobuev。2024。BigLake：BigQuery 向多云湖仓的演进。在 SIGMOD。334–346。[37] Bei Li 和 Christopher Taylor。2024。推出 Spanner Graph：重新构想图数据库（https://cloud.google.com/blog/products/databases/ announcing-spanner-graph/）。[38] Chunbin Lin、Benjamin Mandel、Yannis Papakonstantinou 和 Matthias Springer。2016。带类型的图上的快速内存 SQL 分析。VLDB基金会会刊 10，3（2016），265–276。[39] Mikael Morales、Vlad Ioan Haprian、Srinivas Karthik、Danica Porobic、Laurent Daynès 和 Anastasia Ailamaki。2022。关系数据上的图查询属性投影优化。在第 1 届可组合数据管理系统国际研讨会（CDMS@VLDB 2022），澳大利亚悉尼，2022年9月9日。[40] Hlib Pylypets、Manish Kumar、Abhinav Srivastava 和 Vijay Raja。2024。宣布：Apache Iceberg 双向集成（https://www.singlestore.com/ blog/bidirectional-integration-for-apache-iceberg/）。[41] Mark Raasveldt 和 Hannes Mühleisen。2025。DuckLake：作为湖仓格式的 SQL（https://ducklake.select/2025/05/27/ducklake-01/）。[42] Raghu Ramakrishnan、Baskar Sridharan、John R. Douceur、Pavan Kasturi、Balaji Krishnamachari-Sampath、Karthick Krishnamoorthy、Peng Li、Mitica Manu、Spiro Michaylov、Rogério Ramos、Neil Sharman、Zee Xu、Youssef Barakat、Chris Douglas、Richard Draves、Shrikant S. Naidu、Shankar Shastry、Atul Sikaria、Simon Sun 和 Ramarathnam Venkatesan。2017。Azure Data Lake Store：面向大数据分析的超大规模分布式文件服务。在 SIGMOD。51–63。[43] Tomer Shiran、Jason Hughes 和 Alex Merced。2024。Apache Iceberg 权威指南。O'Reilly。[44] Robert Soulé 和 Bugra Gedik。2014。用于交互图自适应存储的优化磁盘布局。CoRR abs/1410.5290（2014）。[45] Yutian Sun、Tim Meehan、Rebecca Schlussel、Wenlei Xie、Masha Basmanova、Orri Erling、Andrii Rosa、Shixuan Fan、Rongrong Zhong、Arun Thirupathi、Nikhil Collooru、Ke Wang、Sameer Agarwal、Arjun Gupta、Dionysios Logothetis、Kostas Xirogiannopoulos、Amit Dutta、Varun Gajjala、Rohit Jain、Ajay Palakuzhy、Prithvi Pandian、Sergey Pershin、Abhisek Saikia、Pranjal Shankhdhar、Neerad Somanchi、Swapnil Tailor、Jialiang Tan、Sreeni Viswanadha、Zac Wen、Biswapesh Chattopadhyay、Bin Fan、Deepak Majeti 和 Aditi Pandit。2023。Presto：Meta 十年来的 SQL 分析。Proc. ACM Manag. Data 1，2（2023），189:1–189:25。[46] Gábor Szárnyas、Jack Waudby、Benjamin A. Steer、Dávid Szakállas、Altan Birler、Mingxi Wu、Yuchen Zhang 和 Peter Boncz。2022。LDBC 社交网络基准测试：商业智能工作负载。VLDB 会刊（PVLDB）16，4（2022），877–890。[47] Daniel ten Wolde、Tavneet Singh、Gábor Szárnyas 和 Peter Boncz。2023。DuckPGQ：分析型关系数据库管理系统中的高效属性图查询。在第 13 届

创新数据系统研究会议，*CIDR 2023*，阿姆斯特丹，荷兰，*2023*年*1*月*8*日至*11*日。[48] Yuanyuan Tian。2022。行业视角下的图数据库世界。*SIGMOD Rec.* 51，4（2022），60–67。[49] Yuanyuan Tian、En Liang Xu、Wei Zhao、Mir Hamid Pirahesh、Suijun Tong、Wen Sun、Thomas Kolanko、Md. Shahidul Haque Apu 和 Huijuan Peng。2020。IBM Db2 Graph：在 IBM Db2 内支持协同且可改造的图查询。在 *SIGMOD*。345–359。[50] Leslie G. Valiant。1990。并行计算的桥接模型。*Commun. ACM* 33，8（1990），103–111。[51] Song Yu、Shufeng Gong、Qian Tao、Sijie Shen、Yanfeng Zhang、Wenyuan Yu、Pengxi Liu、Zhixin Zhang、Hongfu Li、Xiaojian Luo、Ge Yu 和 Jingren Zhou。2024。LSMGraph：具有多级CSR的高性能动态图存储系统。*Proc.ACMManag.Data* 2，6（2024），243:1–243:28。

[52] Matei Zaharia、Mosharaf Chowdhury、Tathagata Das、Ankur Dave、Justin Ma、Murphy McCauly、Michael J. Franklin、Scott Shenker 和 Ion Stoica。2012。《弹性分布式数据集：内存集群计算的容错抽象》。载于第九届 *USENIX* 网络系统设计与实现研讨会论文集，*NSDI 2012*，美国加利福尼亚州圣何塞，*2012*年*4*月*25*日至*27*日。15–28。[53] Matei Zaharia、Ali Ghodsi、Reynold Xin 和 Michael Armbrust。2021。《湖仓：新一代开放平台，统一数据仓库与高级分析》。载于 *CIDR*。[54] Da Zheng、Disa Mhembere、Randal C. Burns、Joshua T. Vogelstein、Carey E. Priebe 和 Alexander S. Szalay。2015。《FlashGraph：在商用固态硬盘数组上处理十亿节点图》。载于第十三届 *USENIX*文件与存储技术会议论文集，*FAST 2015*，*19*，*2015*年*16*45 58美国加利福尼亚州圣克拉拉，*2*月 -。– 。