# 图数据库基准测试深度报告

## 基于互联数据基准测试委员会 (LDBC) 的社交网络基准测试 (SNB) 标准

Florin Rusu 和 Zhiyi Huang {ffrusu, zhuang29g} @ucmerced.edu 加州大学默塞德分校

2019 年 7 月

摘要

在本研究中，我们提供了对两个原生图数据库系统 Neo4j 和 TigerGraph 完整实施 LDBC SNB 简单查询、深度查询和商业智能 (BI) 测试后的初步结果。我们不但周密评估了在本地和云端的三种计算架构和四种数据集（SF-1、SF-10、SF-100 和 SF-1000）下执行的基准测试中的全部 46 个查询的性能，还测量了批量加载时间和存储容量。我们的结果表明，TigerGraph 在大多数查询中的表现都优于 Neo4j，在特定的深度查询和商业智能查询中性能比后者高出两个（100 倍）甚至更多数量级。因为只有 TigerGraph 能够在数据集扩展到 SF-1000 时执行查询，所以随着数据量的增加，这种差距也在扩大。在 25 个商业智能查询中，Neo4j 在合理时间内仅完成了 12 个。然而，Neo4j 在批量加载 SF-100 或更小的数据集时速度一般较快。在研究中，供应商积极参与了其平台的优化，这对我们很重要。为了鼓励结果重现，我们在网上公布了所有代码、脚本和配置参数。

## 1 简介

过去十年来，在线社交网络迅猛发展，这大大促进了图结构数据处理需求的增长 [18]。这些社交网络具有高度连通的结构，这使得图建模成为了显而易见的理想之选，因为它们提供了直观的抽象结构来表示实体和关系。因此，工业界和学术界开发了许多图分析系统和图数据库 [9, 19]。图分析系统 [8]（例如 Pregel [7]、Giraph [14] 和 GraphLab [6]）专门用于在大规模计算集群上进行全图计算的批处理。而图数据库 [11]（例如 Neo4j [23]、TigerGraph [26] 和 Titan/JanusGraph [17]）侧重于快速查询实体之间的关系和图结构。图数据库将关系视为“头等公民”，而且得益于原生图存储以及对顶点和边建立索引，它的遍历效率很高。遍历很复杂，所以一般用命令式 API 表示，但也有系统定义和实施了面向图的声明式查询语言，例如 Neo4j 的 Cypher [24] 和 TigerGraph 的 GSQL [27]。但是，这些图查询语言还没有标准化 [2]，所以这些系统使用起来很麻烦。尽管如此，相比于用 C++ 或 Java 等底层的命令式语言去实现，这已经是向前迈出的重要一步。鉴于声明式查询语言的高度抽象能力，图数据库代表了迄今为止已开发的最先进的图处理系统。

图处理引擎数量繁多而且具有多样性，所以需要标准的基准测试来帮助用户找到最适合自身应用的工具。此外，无论是在学术界还是工业界，基准测试都可促进竞争，从而带动相应领域的发展，例如 TPC（事务处理性能委员会）基准测试套件之于关系型数据库。互联数据基准测试委员会 (LDBC) [18] 联合了多方力量，试图建立评估图数据管理系统的基准测试实践。LDBC 的主要目标是，设计基准测试规范和程序，并发布基准测试结果 [1]。LDBC 社交网络基准测试 (SNB) [19] 就是这种努力结出的第一个硕果。它为常见的社交网络图建模，并对其执行两大类的性能测试。第一类为交互查询性能测试 [4]，它只从图中读取少量数据进行遍历，这类测试又可进一步分为简单查询 (IS) 和深度查询 (IC)。商业智能 (BI) [12] 性能测试从图中读取大量数据进行探索，查找不同复杂度（包括查询结构的复杂度和属性过滤条件的复杂度）的模式的出现。这与图分析系统的性能测试 [5] 不同，因为它识别出的模式往往要进行分组、聚合和排序以汇总结果。鉴于 LDBC SNB 基于的图结构的通用性，以及其测验的图算法的广泛性，LDBC SNB 可谓是当今世界上最完善的图数据库基准测试标准。

**贡献** 我们提供了对两个支持声明式语言的原生图数据库系统 Neo4j 和 TigerGraph 完整实施 LDBC SNB 基准测试后的初步详尽结果。我们用 Neo4j 的 Cypher 和 TigerGraph 的 GSQL 查询语言实施了基准测试中的全部 46 个查询，并采用系统开发者的直接意见优化了查询。我们使用这些查询语句成功地执行了两种查询语言的交叉验证，在未来实现时，这些语句可作为参考。为此，我们在网上公布了所有代码、脚本和配置参数，以鼓励结果重现。我们在本地和云端的三种计算架构和 SF-1 (1 GB) 到 SF-1000 (1 TB) 四种数据集下评估查询性能，这些计算架构的 CPU 数量与内存容量存在极大的差异。此外，我们还测量了批量加载时间和存储容量。我们的结果表明，TigerGraph 在大多数查询中的表现都优于 Neo4j，在特定的深度查询和商业智能查询中性能比后者高出两个或更多数量级。随着数据量的增加，这种差距也在扩大。而且，只有 TigerGraph 能够扩展到 SF-1000，因为在 25 个商业智能查询中，Neo4j 在合理时间内仅完成了 12 个。但是, Neo4j 在批量加载 SF-100 或更小的数据集时速度一般较快, 只是它需要额外的操作, 就是建索引。

## 2 LDBC SNB 基准测试

在本部分中，我们简要介绍 LDBC SNB 基准测试。我们展示了它的数据结构 (schema)、数据生成过程和查询性能测试。要全面了解这一基准测试，请参考原始出版物 [4, 12] 和正式规范 [19, 21, 20]。

**Schema** LDBC SNB 基准测试的 schema 如图 1 中的 UML 图所示。该 schema 代表了一段时间里的真实社交网络，包括人和人的活动。它从实体及其关系的角度定义了 schema 。主要实体是人 (Person)。每个人在 schema 中都有一系列属性（例如姓名、性别和生日）以及与其他实体的一系列关系，例如某人在某一年从某大学毕业。一个人的活动有与他人的朋友关系，还有内容共享，比如发布消息、回复消息和给消息点赞。一批人组建论坛 (Forum)，可以讨论某些话题——以标签 (tag) 表示。从该 schema 生成的数据集形成一个全联通图，因为任何两个人都可通过多层朋友关系联通。每个人都有几个论坛，这些论坛中的消息形成巨大的讨论树。这些消息与人通过作者 (hasCreator) 和点赞 (likes) 关系相连。组织和地点信息类似于维度，不会随着人数增长而扩大。时间是一个隐含的维度，以 DateTime 类型的属性附属于实体和关系上。虽然 LDBC SNB 的 schema 结构为图，但不强行用图数据库中的图来存储，用关系型数据库中的表来存储也是可以的。

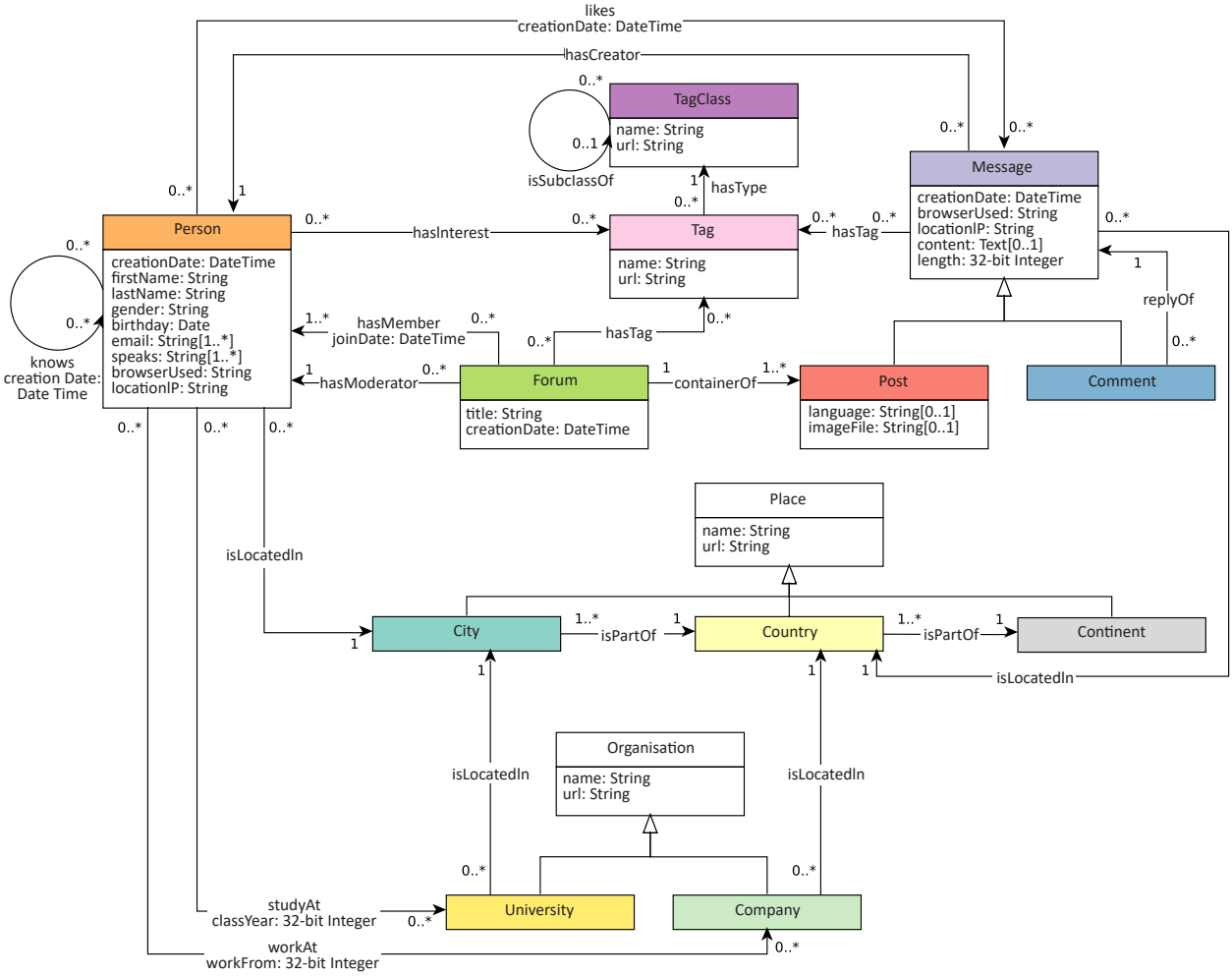



图 1: LDBC SNB 的 schema（完全依据 [21] 重现）。

**数据** LDBC SNB 数据生成器按照与真实社交网络相似的分布和相关性，以不同的数据规模合成不同的数据集。实现的方法是，整合属性值的相关性、基于时间的活动和图结构 [4, 20]。属性值提取自 DBpedia 字典。它们之间是相互关联的，也会影响社交图中的连接模式。例如，一个人居住的地方会影响他 / 她的名字、大学、公司、语言和兴趣（即论坛和标签），进而影响他 / 她的发帖主题，也会影响消息文本。一个人的活动量（即消息的数量）会受到现实世界事件的影响。每当发生重要的事件时，谈论这个话题的人和消息的数量就会激增，尤其是会吸引对这个话题感兴趣的人发出消息。图结构依赖于互连实体实例的属性值。例如，对某个话题感兴趣而且同一年在同一所大学里学习过的人更有可能成为朋友。就像影响者和社区一样，朋友的数量也因人而异。此外，朋友图中的相关性也会传播到消息和评论。基准测试的规模是由社交网络中的人数决定的，后者直接影响到所有其他基数。然而，因为生成过程非常复杂，所以不同大小的数据集之间没有明显的对应关系。具体而言，较大的数据集并不包括较小的数据集。这给查询参数的选择带来了困难，因为结果基数不是随数据集大小线性增长的。在较大的数据集上执行相同的查询时，可能得到基数较小的结果，甚至结果为空。

**查询** 在常见 SNB schema 和对应数据之上定义了三种查询性能测试。它们分别是简单查询 (IS)、深度查询 (IC) 和商业智能 (BI)。每个性能测试都包含许多查询模板（7 个 IS 模板，14 个 IC 模板和 25 个 BI 模板），可测试实施该基准测试的系统的不同特征。为了尽可能接近现实世界的场景，我们从真实系统中总结出一些瓶颈点，也就是在执行或优化查询时容易出问题的方面，然后依据这些瓶颈点挑选查询。例如，使用数据倾斜 (data skew) 和相关性 (correlation) 估计图遍历的基数 (cardinality)；处理分散的索引访问模式 (scattered index access pattern)；重复使用子查询 (sub-query)、查询内 (intra-query) 和查询间 (inter-query) 结果；top-k 下推；后期推测 (late projection)；稀疏外键连接 (sparse foreign key joins)；维度群集 (dimensional clustering) 等。完整列表参见 [12]。每个查询模板都带有一组预定义的模板参数代替值，即参数绑定。鉴于图的倾斜结构，选择参数绑定时需要特别小心，因为这会导致执行时间出现很大的差异。数据生成期间选择代替值的参数策划过程 [4] 旨在确保在所有参数绑定中的执行时间保持稳定。由于结构图的变化，每种数据集将生成一系列不同的绑定。

**简单查询 (IS) 性能测试** IS 性能测试中的查询是相对简单的路径遍历，作为一种参数绑定，可从原点访问最多经过 2 次跳跃的顶点。它们起始于一个人或一条消息，然后访问相邻的朋友和关联的消息。假设可以通过索引查找检索到原点，那么必须访问的数据量要远小于数据集的大小。此外，因为访问的数据量相同，所以执行时间不会受到数据集的直接影响。

我们以查询 IS_3 为例：*给定用 ID 标识的人，查找他们的所有朋友和成为朋友的日期；按日期由近及远的顺序返回朋友*。此查询需要根据个人 ID 查找朋友，然后再查找每个朋友的信息。最后，在朋友列表上执行排序，该列表的基数应该相对较小。访问的数据量与输入的个人的朋友数成正比，检索到的这些朋友通过友谊路径与原点个人相连，且路径上只有 1 次跳跃。其他 IS 查询需要 0 到 2 次跳跃。具体规范可参见 [21]。

**深度查询 (IC) 性能测试** IC 性能测试中的查询超出了 2 次跳跃的范围，并计算简单的聚合，而非仅返回元组。另外，其中两个查询必须计算作为参数的两个顶点之间的最短路径。为了支持这种查询，执行引擎中必须包含递归，因为遍历的深度取决于参数绑定。这样便不再需要进行预处理和实体化，仍能使得方案可行。虽然遍历的原点仍然是固定的，但是查询必须检查的数据量大于 IS 查询。此外，聚合计算需要在全局或组级别进行状态处理。这些通常会导致执行时间随数据集的增大而延长。

我们以查询 IC_9 作为这种性能测试的代表性示例：*给定用 ID 标识的人，查找他们的朋友或朋友的朋友在给定日期前发布的最近 20 条消息*。此查询查找 2 跳或 3 跳长的路径，从给定的人开始，前进到他 / 她的朋友和朋友的朋友，最后是他们发布的消息。虽然朋友通常是实体化的，但是要找到朋友的朋友则需要再一次遍历图。此遍历的结果与直接朋友合并，并用于消息的遍历。如果允许查找友谊关系更远的朋友，则需要遍历更多次。日期参数可用于减少查找的消息数量。朋友和消息遍历的执行顺序会对执行时间造成显著影响。其他 IC 查询 [4] 也是如此。

**商业智能 (BI) 性能测试** BI 性能测试中的查询将访问更大部分的图。它的实现方式是，将交互式查询中的原点替换为较为一般化的选择，如消息创建日期或人员位置。因此，遍历并非起源于单个源，而是源于图中的多个点。需要计算的聚合也更加复杂。它们包含基于多个属性（有些是合成的）和非平凡 (non-trivial) 函数的复杂分组条件。top-k 排名和排序将应用于这些复杂的聚合。除了单一源（单目的地最短路径）查询，较一般化的加权路径查询和固定大小的团（例如三角形）查询也是 BI 性能测试的一部分。要高效执行 BI 查询，除了图遍历，还需要在系统的所有层上进行大量优化。

我们以 BI 5 作为这种性能测试的代表性查询：*查找给定国家 / 地区里最受欢迎的 100 个论坛；对于这些论坛里的每个人，清点他们在所有流行论坛里的发帖数量。*此查询很好地结合了图遍历和复杂的 top-k 聚合。这种最优图遍历需要在论坛和人之间找到正确的方向。top-k 被用作计数聚合的条件，其结果将进一步运用到另一个分组聚合中。这需要查询语言和执行引擎的高级支持。BI 性能测试中的其他查询有着相同的模式和复杂度 [12, 21]。

# 3 声明式图数据库系统

在本部分中，我们将介绍本研究中考量的两个图数据库系统 Neo4j [23, 11] 和 TigerGraph [26, 3]，重点在于它们各自的查询语言 Cypher [24, 8] 和 GSQL [27, 13]，而非它们的架构或执行引擎。我们的目标是，用声明式图查询语言完整地实现 LDBC 基准测试，并分析它们的特点。我们选择 Neo4j 和 TigerGraph 的原因是，二者是仅有的提供类 SQL 声明式查询语言的免费原生图数据库。我们不考虑 Oracle PGX [10] 和 Amazon Neptune [25] 等系统，因为它们不是免费的或者只能在云端运行。我们也不考虑支持 Gremlin [15] 这种函数式 / 过程化查询语言的系统（例如 JanusGraph [17]），或者通过在关系型数据库中用 SQL、在 RDF 数据库中用 SPARQL 语句来实现图的方案。

**Cypher** Neo4j 的图查询语言基于标签化属性图（labeled property graph, 简称 LPG）数据模型 [11, 2], 这是表示图的最流行模型。标签化属性图是一个有向图, 其顶点和边上都有标签, 并且会有一些 < 属性，值 > 对。一个顶点 / 边通常关联有多个属性，而只有一个标签。映射到关系型数据库语境，标签对应于表名，而属性对应于表的列名。但是，每个顶点 / 边可以有自己独有的标签和属性，即它们不是预定义表中的元组。因此，标签化属性图模型是无 schema 的。这一方面提供了极大的灵活性, 而另一方面也扩大了存储, 降低了计算效率。Cypher 查询是指定图中路径的子句。子句通过标签识别顶点，并通过顶点的属性筛选出结果集（可能很大）。路径上的边也是根据它们的标签选择的。Cypher 沿用了声明式风格，定义了有限数量的关键字，其中许多关键字借鉴自 SQL。子句是根据 Datalog 语法编写的。

为了更具体地说明，我们提供了 LDBC 查询 IS_3 和 IC_9 的 Cypher 语句，所有 LDBC 查询语句都在网上公布 [22, 16]。IS_3 是一段简单的 `MATCH-RETURN` Cypher 子句，始于一个给定 `ID` 的 `Person` 顶点，经过一条类型为 `Knows` 的边 `r`，指向另一个 `Person` 顶点 `friend`，从而获取所有符合这一模式的路径。该查询按照排序的顺序返回 `friend` 和 `r` 的属性。输出可视为有固定 schema 的关系型数据库中的表。查询 IC9 指定的路径更复杂。首先，它包含朋友及朋友的朋友，即一条或连续两条 `Knows` 边连接的 `Person` 顶点，给予别名 friend。其次，匹配到始于 `friend`、指向 `Message` 的有向边 `Has_Creator`。仅考虑在 `maxDate` 之前发布的那些 `Message` 顶点。

```
MATCH (:Person id:$personId)-[r:Knows]-(friend:Person)
RETURN
   friend.id AS personId,
   friend.firstName AS firstName,
   friend.lastName AS lastName,
   r.creationDate AS friendshipCreationDate
ORDER BY friendshipCreationDate DESC, personId ASC
MATCH (:Person id:$personId)-[:Knows*1..2]-(friend:Person)
      <-[:HasCreator]-(message:Message)
WHERE message.creationDate < $maxDate
RETURN DISTINCT
   friend.id AS personId,
   friend.firstName AS personFirstName,
   friend.lastName AS personLastName,
   message.id AS messageId,
   CASE exists(message.content)
     WHEN true THEN message.content
     ELSE message.imageFile
   END AS messageContent,
   message.creationDate AS messageCreationDate
ORDER BY messageCreationDate DESC, messageId ASC
LIMIT 20
```

这些查询显示了 Cypher 与 SQL 之间的密切关系。事实上，Cypher 继承了 SQL 的表达能力，是 SQL 完备的语言。这是通过标签化属性图模型和表函数的组合而实现的。因此，Cypher 对控制流的支持非常有限，也很难通过子查询实现嵌套查询。这就限制了可以用 Cypher 表达的图计算，特别是递归和迭代算法。尽管如此，正因为 Cypher 与 SQL 很相似，新手过渡到 Neo4j 相对就容易很多。

**GSQL** GSQL 是 TigerGraph 的查询语言 [3]。顾名思义，GSQL [27] 是 SQL 向图数据库的直接延伸。它在查询之前采用一个严格的 schema 声明。其 schema 采用标签化属性图数据模型，并包含顶点、边、图和标签四种类型 [13]。顶点类型对应于 SQL 表。它有一个名称和多个属性。定义边类型时，要确定其连接的两顶点的类型。它可以是无向边或者有向边。如果是有向边，可额外定义一个反向边类型。图类型定义了创建该图的顶点和边类型。加入标签类型只是为了与标签化图数据模型兼容。因为一开始便指定了一切，所以 TigerGraph 可采用最优存储格式和查询执行策略。GSQL 中的查询不是单个 SQL `SELECT` 语句，而是包含多个 `SELECT` 子句和命令式指令（如分支与循环）的存储过程。GSQL 查询本质上就是 SQL 存储过程。促使采用这种方法的原因是某些图计算的高度复杂性。与 Cypher 中的 `MATCH` 相似，GSQL 中的 `SELECT` 语句始于一些顶点，紧随着边，匹配到图中符合一定模式的路径。路径是在 `FROM` 子句中指定的。GSQL 引入了与路径相关的累加器 `ACCUM` 的概念。沿路径找到的数据可根据不同的分组标准收集并聚合到累加器中。这个过程是并行的，程序为 `FROM` 子句中每条匹配的边都会生成一个线程。聚合的结果可以临时存储到各顶点上，以支持多次遍历和迭代计算。

我们选择了 LDBC 查询 IS_3 和 IC_9 的 GSQL 语句，与对应的 Cypher 查询进行对比。用 GSQL 实现的所有其他 LDBC 查询均在网上公布 [28]。IS_3 中的 `SELECT` 语句与 Cypher 中的 `MATCH` 语句非常相似。主要的不同点在于，累加器 `@creationDate` 附加到通过 `SELECT` 返回的 `Person` 顶点。这是必要的，在 GSQL 中 `SELECT` 语句返回的是一个明确定义的类型。IC_9 展现了 GSQL 的多种特性，包括若干类型的累加器和循环。参数 `vPerson` 的朋友和朋友的朋友是在一个 WHILE 循环中计算的，该循环会从起点出发，通过 `Person_Knows_Person` 边连续跳两次进行遍历。这些朋友全都存储在集合累加器 (SetAccum) `@@friendAll` 中。朋友发布的 `Messages` 存储在固定大小的 heap 累加器中，后者在在定义时声明了一个比较函数以供排序之用。这样做的好处是只会存储需要返回的 `Messages`。GSQL 并不像 Cypher 代码那么简洁，因为它包含命令式指令。但是，它遵循完善的存储过程 SQL 范式，后者将所有逻辑编译为对象嵌入到数据库内。应用只需要通过函数调用来调用该过程。尽管在这些示例查询中没有尽显 GSQL 的强大，但我们确知，用 GSQL 可以实现许多 Cypher 不能实现的图计算功能。Cypher 是 SQL 完备的，而 GSQL 是图灵完备的。

```
CREATE QUERY IS3(VERTEX<Person> personId) FOR GRAPH ldbc_snb {
  SumAccum<INT> @creationDate;
  vPerson = {$personId};
  vFriend = SELECT t
            FROM vPerson:s-(PersonKnowsPerson:e)->Person:t
            ACCUM t.@creationDate+=e.creationDate
            ORDER BY t.@creationDate DESC, t.id ASC;
  PRINT vFriend [
      vFriend.id AS personId,
      vFriend.firstName AS firstName,
      vFriend.lastName AS lastName,
      vFriend.@creationDate AS friendshipCreationDate
  ];
}

CREATE QUERY IC9(VERTEX<Person> personId, DATETIME maxDate)
FOR GRAPH ldbcsnb {
  TYPEDEF tuple < INT personId,
     STRING personFirstName,
     STRING personLastName,
     INT messageId,
     STRING messageContent,
     DATETIME messageCreationDate
   > msgInfo;
  OrAccum @visited;
  SetAccum<VERTEX<Person>> @@friendAll;
  HeapAccum<msgInfo>(
    20, messageCreationDate DESC, messageId ASC
  ) @@msgInfoTop;
  vPerson = {$personId};
  INT i = 0;
  WHILE i < 2 DO
```

```
    vPerson = SELECT t
              FROM vPerson:s-(PersonKnowsPerson:e)->Person:t
              WHERE t.@visited==False
              ACCUM s.@visited+=True, t.@visited+=True, @@friendAll+=t;
    i = i + 1;
  END;
  vFriend = {@@friendAll};
  vMessage =
   SELECT t
   FROM vFriend:s-(CommentHasCreatorPersonREVERSE:e)->Comment:t
   WHERE t.creationDate < $maxDate
   ACCUM @@msgInfoTop += msgInfo(s.id, s.firstName, s.lastName,
              t.id, t.content, t.creationDate);
  PRINT @@msgInfoTop;
}
```

# 4 基准测试实验

在本部分中，我们呈现了在 Neo4j 和 TigerGraph 中执行全部 LDBC SNB 基准测试的结果。虽然以前发布过少部分性能测试的结果（[4] 中的 IS 和 IC，[12] 中的 BI），但本论文首次在同一研究中考虑了所有性能测试。此外，我们还首次提供了基于 1 TB 数据级的测试集 SF-1000 的测试结果。我们的主要焦点是报告两个系统的查询执行时间。此外，我们还通过加载时间和存储容量两个指标来评估数据加载性能。在提供结果之前，我们首先介绍实验设置。

## 4.1 设置

**实现** 实验中使用的两个图数据库是 Neo4j 3.5.0 社区版和 TigerGraph 2.3.1 开发者版本。这些版本是免费的，可能不包括商业支持版本中提供的所有优化。虽然这两个系统都可以在分布式模式下运行，但我们对其进行了最优单节点执行配置，即我们允许充分利用内存和线程。Cypher 查询是用 Python 实现的，并通过标准的 ODBC/JDBC 接口传递给 Neo4j 服务器执行。我们使用了与 LDBC 代码库 [22] 中一样的代码。执行结果和执行时间返回给 Python 应用以便记录 / 显示。在 TigerGraph 中执行查询的过程遵循关系数据库的存储过程工作流。首先，查询必须在服务器中注册和编译。这将在数据库目录中注册一个对象，并为查询生成相应的可执行代码。在第二阶段，类似于 Neo4j 驱动程序的 Python 应用通过函数来调用该查询 / 存储过程。GSQL 查询语句 [28] 相对新颖，所以根据 TigerGraph 工程师直接提供的意见进行了优化，我们感谢他们的帮助。我们为 Neo4j 的工作人员提供了同样的优化机会，但他们在发布本论文之前的两个多月里拒绝向我们提供任何意见。两个系统的查询结果已用于成功地执行 Cypher 和 GSQL 查询语言的交叉验证，并能作为未来执行 LDBC 基准测试的参考。

**系统** 我们使用三种不同的计算机进行实验。它们的配置可参见表 1。最小的计算机是我们在加州大学默塞德分校的服务器，专用于运行两个图数据库。另两个计算机是亚马逊 AWS 实例。AWS r4.8xlarge 是可由多个用户共享的虚拟实例，AWS x1e.16xlarge 是专门独享的物理实例。所有计算机上的操作系统都是 Ubuntu 18.04.2 LTS。根据可用的内存容量，我们在内存足以支持所需数据大小的最小计算机上执行给定数据集的实验。因此，SF-1 (1 GB) 和 SF-10 (10 GB) 的实验在加州大学默塞德分校服务器上执行，SF-100 (100 GB) 的实验在 AWS r4.8xlarge 上执行，SF-1000 (1 TB) 的实验则在 AWS x1e.16xlarge 上执行。除了内存大小的差异，这三个计算机的 CPU 内核 / 线程数量也有很大的差异。由此会导致明显不同的并行度，这一点在执行相近数据量的查询时特别明显（无论数据量有多大），例如做简单查询。总之，数据大小和硬件特征的组合清楚显示了两个图数据库在 LDBC 基准测试中所表现的性能。我们未发现有任何其他研究采用了如此详尽的方法。

| 数据集 | 计算机 | （虚拟）CPU 内核 | RAM | OS | Java | Python |
|---|---|---|---|---|---|---|
| SF-1 和 SF-10 | 加州大学默塞德分校 | 16 | 28 GB | Ubuntu 18.04.2 LTS | 内部版本号 1.8.0 191 | 2.7.15 |
| SF-100 | AWS r4.8xlarge | 32 | 240 GB | Ubuntu 18.04.2 LTS | 内部版本号 1.8.0 191 | 2.7.15 |
| SF-1000 | AWS x1e.16xlarge | 64 | 2 TB | Ubuntu 18.04.2 LTS | 内部版本号 1.8.0 201-b09 | 3.6.5 |

表 1：基准测试所使用的硬件和系统软件的规格。

**方法** 默认情况下，我们将所有查询执行 10 次并将最后 9 次运行的中值报告为结果，以图表和表格展示。我们使用中值而非平均值，是因为在存在偶发的异常波动时仍有稳定的结果。忽略第一次运行，是因为它所用的时间通常比后续运行长得多。这是冷缓存启动的缘故。为了限制专用于特定查询的时间量，我们将超时设置为 18,000 秒（5 小时）。超时后，查询将终止。数据加载实验只执行一次，因为它们花费的时间长很多，而且运行时更稳定。存储实验中报告的数据大小是使用 ls 和 du 命令根据占用的文件系统空间测量的。

**数据集** 实验中使用的 LDBC SNB 图的特征参见表 2。我们可以观察到，随着数据集的增大，顶点和边的数量几乎呈线性增长。这种关系并不完全是线性的，因为数据生成过程考虑了图的稀疏性，节点的出入度在数据规模变大时保持相对的稳定。尽管如此，在 SF-1000 数据集下，图中有近 27 亿个顶点和 180 亿条边，原始数据需要 900GB 的存储空间。总而言之，即使对于现有最大的社交网络，这也是一个极大的图。

| 数据集 | 顶点（百万） | 边（百万） | 原始大小 (GB) |
|---|---|---|---|
| SF-1 | 3.18 | 17.26 | 0.813 |
| SF-10 | 30.00 | 176.62 | 8.400 |
| SF-100 | 282.64 | 1,780.00 | 87.300 |
| SF-1000 | 2,690.00 | 17,790.00 | 900.000 |

表 2：LDBC SNB 图特征。

## 4.2 结果

我们将结果分为加载和查询两组。对于加载实验，我们测量两个系统中图所需的存储空间，以及查询数据前所需的数据加载时间，即查询就绪时间。对于查询实验，我们报告 Neo4j 和 TigerGraph 分别基于四种不同大小的数据集执行全部 46 个 SNB 查询的运行时间，总共有 2×4×46=368 种配置。

**加载的数据大小** LDBC SNB 数据生成器生成原始图数据后，两个图数据库将使用各自的查询语言所支持的构造方式，将原始图数据批量加载到图数据库中。Neo4j 提供导入 API，从不同格式生成标签化属性图。我们使用导入 API 从分隔符分隔的文本文件导入数据。该 API 使用一个命令加载具有相同标签的顶点和边（例如同时加载所有 `Person` 顶点），并加载每个顶点 / 边实例的所有属性。为了高效地识别具有指定属性的顶点，必须在加载的数据上建立索引。这是一个独立的加载后过程。我们基于 SNB 性能测试在 Neo4j 中创建以下索引：`Person(id)`、`Message(id)`、`Post(id)`、`Comment(id)`、`Forum(id)`、`Organisation(id)`、`Place(name)`、`Tag(name)` 和 `TagClass(name)`。TigerGraph 了解图数据加载的复杂性，于是特地提供了一种用 GSQL 实现的数据加载语言 (DDL)。因此，TigerGraph 中的加载采用衍生自图定义 DDL 的声明式 GSQL 语句执行。这些语句创建并行作业来提取顶点 / 边属性，并转化为 TigerGraph 内部存储格式。既没有预处理（例如提取唯一的顶点 ID），也没有后处理（例如显式建立索引）。因为整个过程是离线执行的，所以在 TigerGraph 中叫做离线加载。离线加载分为加载和生成两个阶段。加载阶段解析并准备二进制数据，而生成阶段则将二进制数据合并和打包成实际的存储格式。

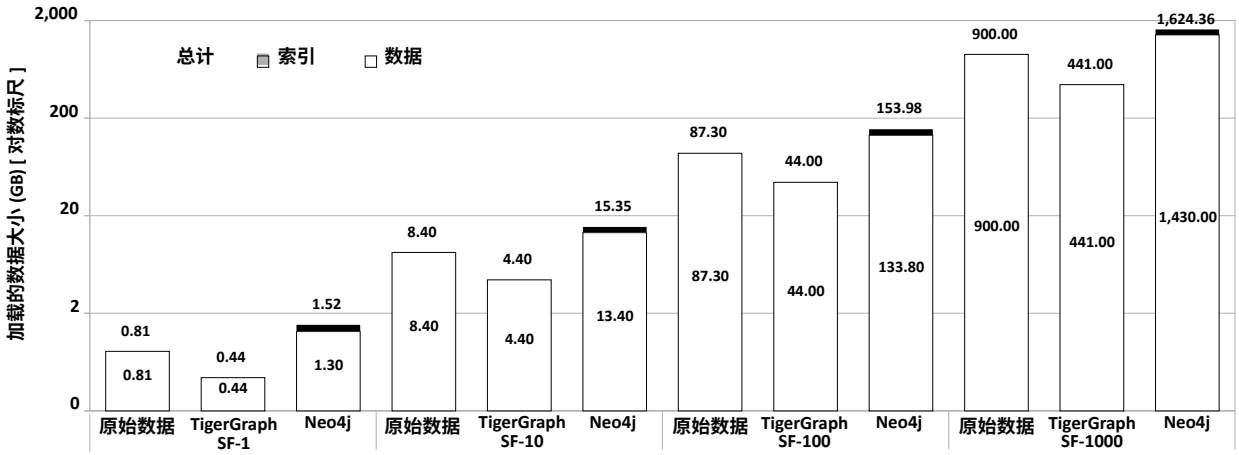



图 2：加载数据的大小，分为实际数据大小和索引大小。原始数据量对应于 LDBC 基准测试生成器生成的数据大小。TigerGraph 不创建显式索引。条形内的数字表示实际数据的大小，而条形上方的数字对应总数据大小。两者之间的差表示索引的大小。

图 2 呈现了在考虑的所有数据集下加载数据和原始数据的大小。我们可以看到，TigerGraph 将原始数据大小减小了接近一半，而 Neo4j 却增加了一倍。TigerGraph 之所以压缩到如此程度，是因为图 schema 采用了固定格式，不再需要存储每个实例的属性名称。在 Neo4j 中，这导致加载数据大小比原始数据大了 50% 以上。在增加的大约一倍大小中，9 个索引占了 40%。因此，显然采用图 schema 对于存储有积极影响。Neo4j 中的数据大小是 TigerGraph 中的 3 倍，算上索引是 4 倍。

**加载时间** 图 3 显示了加载时间，后者分为数据摄取时间和索引创建时间。在 Neo4j 中测量此时间需要为两个独立的进程计时。尽管如此，Neo4j 中的总加载时间也比 TigerGraph 少，SF-1000 除外。两个系统的差异随着数据集的增大而减小。这完全归结于索引效率。Neo4j 中的索引生成是不可扩展的，随着数据集的增大而呈指数级增长，从 12 秒 (SF-1) 增加到 103 秒 (SF-10)，再到 961 秒 (SF-100)。数据集是 SF-1000 时，建立索引需要 34,424 秒，是摄取时间的两倍多。因为 TigerGraph 不建立索引，所以我们将离线加载的生成阶段所用的时间视为 Neo4j 索引建立的等价时间。生成阶段的可扩展性比建立索引高很多，在 SF-1000 数据集下只需要 3,551 秒。这种巨大的差异归因于 TigerGraph 较短的加载时间，尽管 Neo4j 的摄取时间仍然只有 TigerGraph 的三分之二。虽然 Neo4j 中的索引建立时间可通过少生成索引而缩短，但我们会看到即使索引数量如此之大，SF-1000 数据的查询仍然非常低效。索引减少后，可能几乎无法在可接受的时间里完成任何查询。

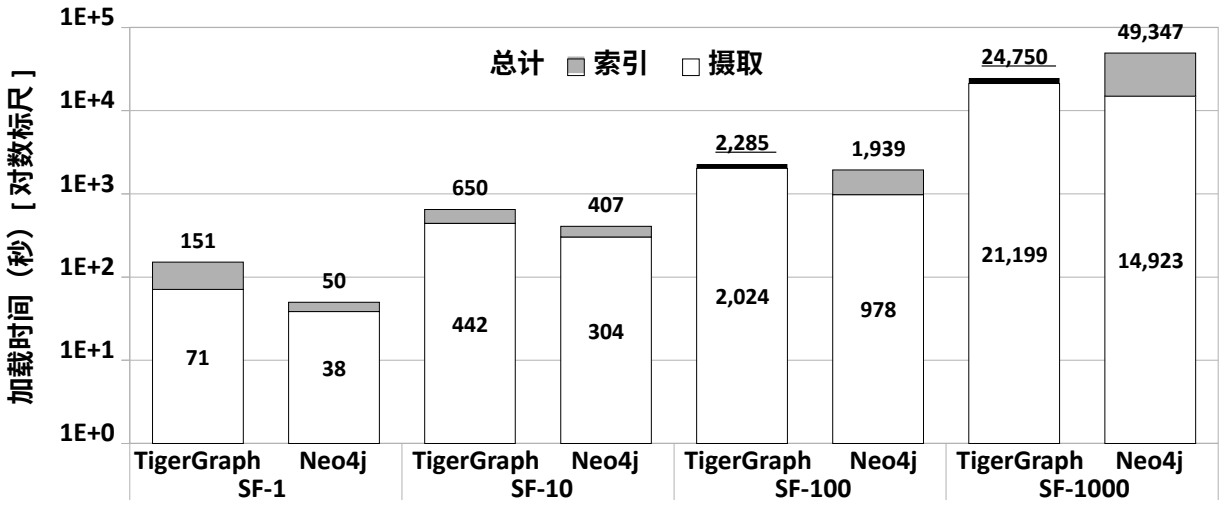
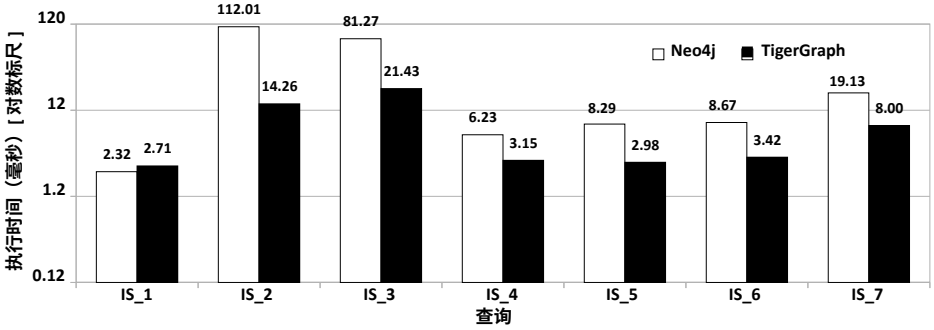
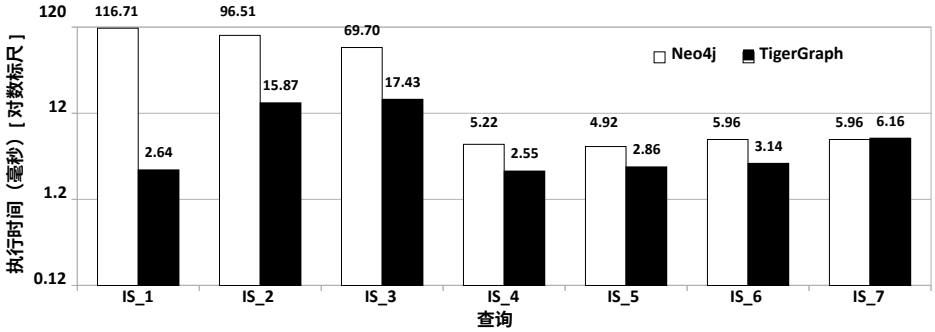


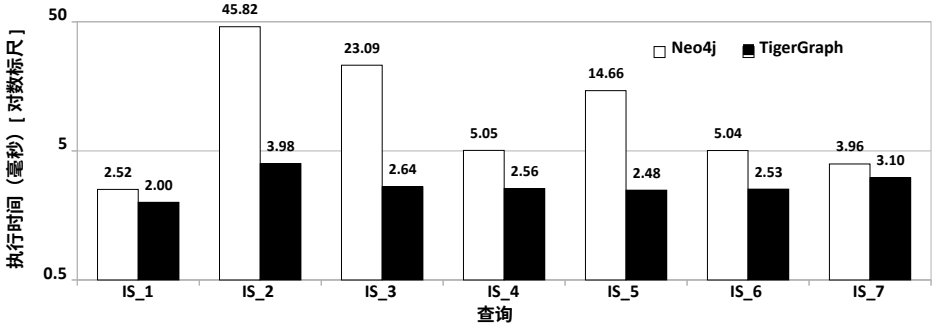

图 3：加载时间，分为摄取时间和索引时间。条形内的数字表示摄取时间，而条形上方的数字对应总加载时间。两者之间的差表示索引时间，对应 TigerGraph 中的生成阶段时间。



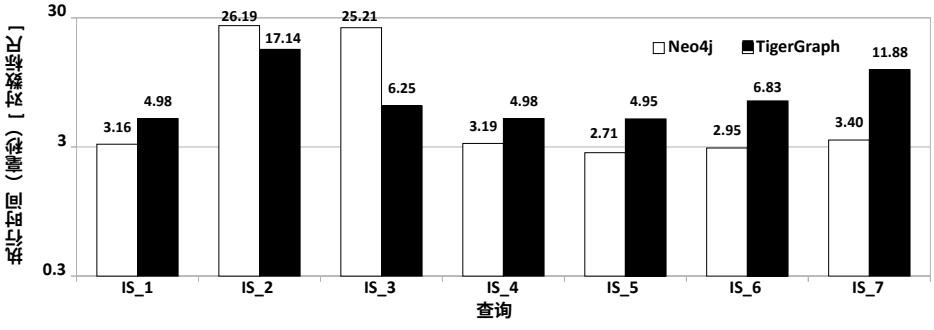

(a)




(b)




(c)




(d)

图 4：数据集 1 (a)、10 (b)、100 (c) 和 1000 (d) 下简单查询 (IS) 的执行时间（毫秒）

**IS 性能测试** 图 4 显示了考虑的所有四种数据集下 IS 性能测试的七个查询的运行时间。因为所有查询访问的数据量都非常有限（在搜索键上建立索引），所以在所有情况下运行时间都是亚秒级的。事实上，除了查询 IS_2 和 IS_3，其他所有运行时间几乎都少于 10 毫秒。这与以前发布的其他系统的结果 [4, 9] 一致。细心的读者会立刻察觉到，数据集和运行时间之间没有明显的关系，运行时间没有随着数据集的增大而增加。恰好相反，有些查询的运行时间反而减少了。这是因为在所有数据集下，索引都是在内存中生成和存储的。因此，随机内存访问的时间非常相似，与数据大小无关。此外，在较大数据集下生成的图不是较小数据集下生成的图的超集，SF-1 下显示的 ID 不一定显示在 SF-10 下。因此，我们不得不修改每个数据集的查询参数，从而产生所观察到的变化。通过比较两个系统，显然 TigerGraph 在 SF-1、SF-10 和 SF-100 下胜出，Neo4j 只在两个查询中速度较快。然而令人有些吃惊的是，Neo4j 在 SF-1000 下执行的五个查询中竟然比 TigerGraph 速度更快。我们唯一的解释就是，执行实验的计算机的属性造成了这种结果。尽管如此，考虑到执行时间都很短，所以这两个系统之间的差异对于这种性能测试来说并不重要。如果一个人只运行这种性能测试，那么两个系统都是不错的选择。

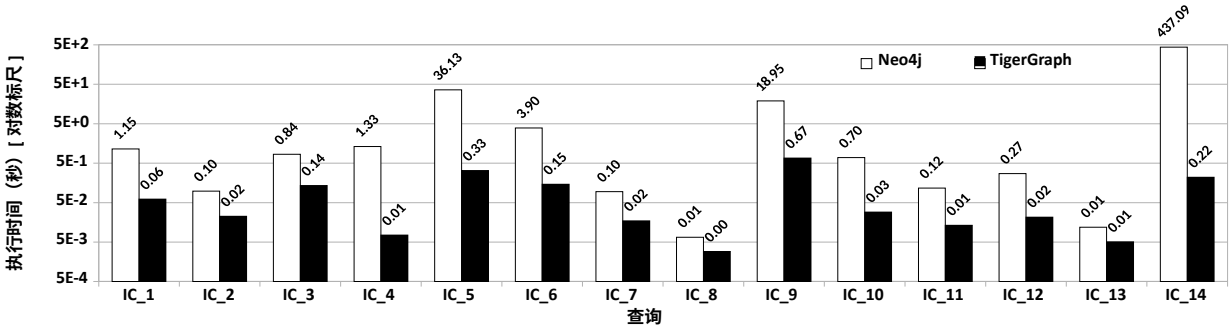
**IC 性能测试** 这种情形完全不同于图 5 中所显示的 IC 性能测试。在所有数据集下，Neo4j 仅在三个查询中略胜于 TigerGraph。在其他所有查询中，TigerGraph 都明显更快，有时高出四个数量级。TigerGraph 在几十毫秒内就能完成的查询在 Neo4j 中需要一千多秒。此外，当数据集巨大时，Neo4j 甚至不能在分配的 5 小时超时限制内完成某些查询的执行。因此，对于这种性能测试对应的工作，TigerGraph 显然是更优的选择。因为访问的数据量较大，并且与数据集成一定比例，所以运行时间通常也会随着数据集的增大而增加。至于索引时间，Neo4j 所用时间的增长幅度超过 Tigergraph。然而，因为图是不同的，所以我们必须更改查询参数，这有时会导致非线性行为。

**BI 性能测试** BI 性能测试的结果显示在图 6 中。同样，在所有数据集下及所有查询中，TigerGraph 的表现显然优于 Neo4j，只有两个 SF-1 查询和一个 SF-10 查询除外。在所有 SF-100 和 SF-1000 查询中，TigerGraph 平均比 Neo4j 快近一个数量级（即 10 倍）。此外，随着数据集的增大，Neo4j 在分配的时间内可执行完的查询越来越少。在 SF-1000 数据集下，25 个查询中只有 12 个在超时之前执行完毕。这表明在执行复杂 BI 查询时 Neo4j 无法查询大量数据。TigerGraph 的结果与文献 [12] 中发布的仅有的其他结果（最大规模 SF-10）相似。当然，因为这不是在相同硬件上的直接比较，所以只能作为一个基本指导，而不是权威证明。我们没有发现任何在更大数据集下得出的查询结果。

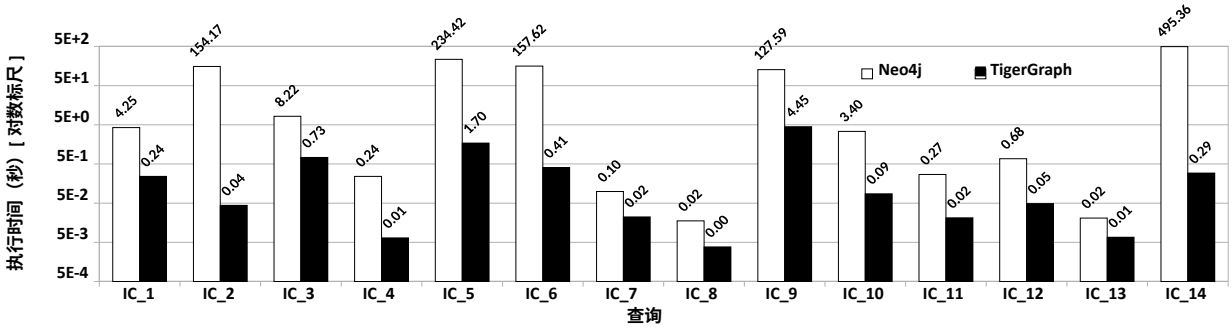
## 4.3 小结

我们可以将深入的实验研究结果归纳如下：

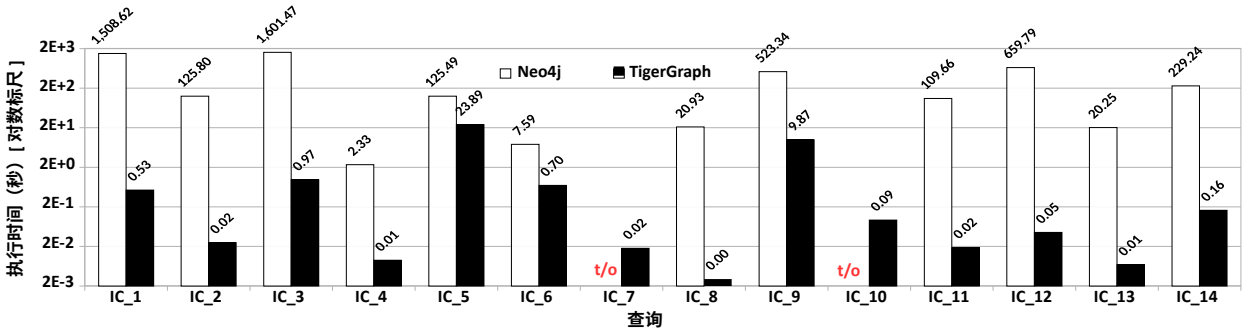
- TigerGraph 存储图数据的紧凑度比 Neo4j 高很多。它的原始数据占用的存储空间是后者的三分之一，如果算上索引，则只有四分之一。此外，TigerGraph 将 LDBC SNB 数据生成器生成的原始数据压缩到二分之一。
- Neo4j 摄入原始数据的速度比 TigerGraph 更快。在 SF-1 下两者的差距是 3 倍，但这个差距会随着数据集的增大而减小。然而，Neo4j 的索引生成算法可扩展性很差，索引时间呈指数级增长。因此，在 SF-1000 下，TigerGraph 的总加载时间是 Neo4j 的二分之一。
- 表 3 汇总了所有四种数据集下 SNB 性能测试中全部 46 个查询的运行时间。在 368 个配置中，Neo4j 仅在 13 个配置下速度超过 TigerGraph（在表中以粗体显示）。



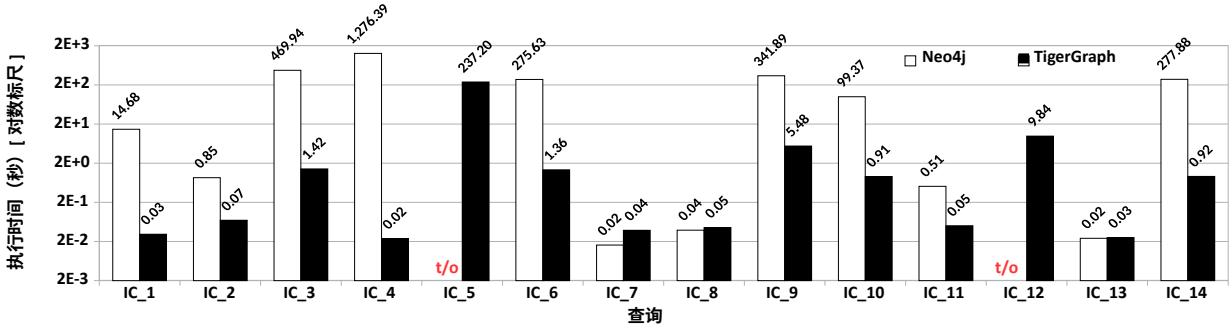

(a)




(b)

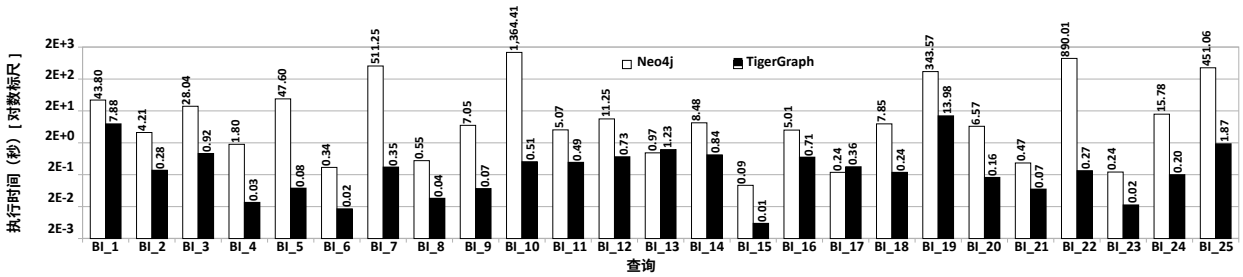



(c)

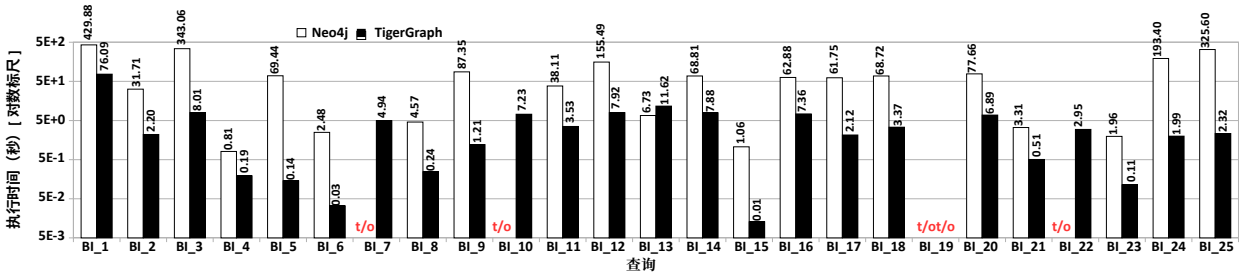



(d)

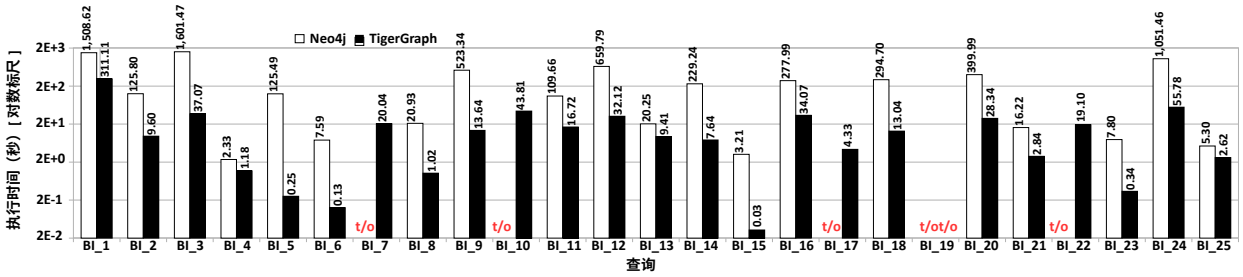
图 5：在 1 (a)、10 (b)、100 (c) 和 1000 (d) 的数据集下，深度查询 (IC) 查询的执行时间（秒）。t/o 代表超时，即在 18,000 秒内没有执行完毕。



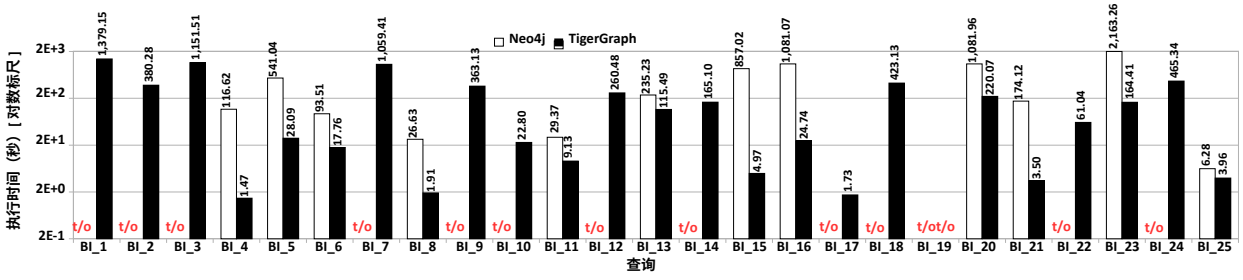

(a)




(b)




(c)




(d)

图 6：在 1 (a)、10 (b)、100 (c) 和 1000 (d) 的数据集下，商业智能 (BI) 查询的执行时间（秒）。t/o 代表超时，即在 18,000 秒内没有执行完毕。

| 查询 | 执行时间（秒） | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | SF-1 | | SF-10 | | SF-100 | | SF-1000 | |
| | TigerGraph | Neo4j | TigerGraph | Neo4j | TigerGraph | Neo4j | TigerGraph | Neo4j |
| IS_1 | 0.0027 | **0.0023** | 0.0026 | 0.1167 | 0.0020 | 0.0025 | 0.0050 | **0.0032** |
| IS_2 | 0.0143 | 0.1120 | 0.0159 | 0.0965 | 0.0040 | 0.0458 | 0.0171 | 0.0262 |
| IS_3 | 0.0214 | 0.0813 | 0.0174 | 0.0697 | 0.0026 | 0.0231 | 0.0062 | 0.0252 |
| IS_4 | 0.0032 | 0.0062 | 0.0026 | 0.0052 | 0.0026 | 0.0051 | 0.0050 | **0.0032** |
| IS_5 | 0.0030 | 0.0083 | 0.0029 | 0.0049 | 0.0025 | 0.0147 | 0.0049 | **0.0027** |
| IS_6 | 0.0034 | 0.0087 | 0.0031 | 0.0060 | 0.0025 | 0.0050 | 0.0068 | **0.0029** |
| IS_7 | 0.0080 | 0.0191 | 0.0062 | **0.0060** | 0.0031 | 0.0040 | 0.0119 | **0.0034** |
| IC_1 | 0.0613 | 1.1497 | 0.2412 | 4.2540 | 0.5317 | 1,508.6240 | 0.0307 | 14.6836 |
| IC_2 | 0.0226 | 0.0980 | 0.0439 | 154.1679 | 0.0249 | 125.7995 | 0.0696 | 0.8533 |
| IC_3 | 0.1357 | 0.8421 | 0.7330 | 8.2241 | 0.9703 | 1,601.4695 | 1.4237 | 469.9391 |
| IC_4 | 0.0075 | 1.3262 | 0.0065 | 0.2405 | 0.0090 | 2.3326 | 0.0237 | 1,276.3919 |
| IC_5 | 0.3257 | 36.1317 | 1.7019 | 234.4228 | 23.8868 | 125.4889 | 237.1967 | t/o |
| IC_6 | 0.1458 | 3.8971 | 0.4050 | 157.6176 | 0.6994 | 7.5908 | 1.3598 | 275.6304 |
| IC_7 | 0.0172 | 0.0950 | 0.0222 | 0.0987 | 0.0180 | t/o | 0.0387 | **0.0163** |
| IC_8 | 0.0029 | 0.0066 | 0.0038 | 0.0176 | 0.0030 | 20.9287 | 0.0454 | **0.0391** |
| IC_9 | 0.6717 | 18.9488 | 4.4511 | 127.5934 | 9.8687 | 523.3391 | 5.4814 | 341.8940 |
| IC_10 | 0.0288 | 0.6950 | 0.0866 | 3.4005 | 0.0928 | t/o | 0.9118 | 99.3705 |
| IC_11 | 0.0133 | 0.1167 | 0.0211 | 0.2693 | 0.0186 | 109.6632 | 0.0503 | 0.5141 |
| IC_12 | 0.0213 | 0.2718 | 0.0487 | 0.6813 | 0.0451 | 659.7865 | 9.8416 | t/o |
| IC_13 | 0.0051 | 0.0119 | 0.0067 | 0.0208 | 0.0069 | 20.2515 | 0.0251 | **0.0242** |
| IC_14 | 0.2188 | 437.0878 | 0.2927 | 495.3559 | 0.1640 | 229.2400 | 0.9184 | 277.8833 |
| BI_1 | 7.8821 | 43.7982 | 76.0949 | 429.8811 | 311.1128 | 1,508.6240 | 1,379.1499 | t/o |
| BI_2 | 0.2766 | 4.2070 | 2.2038 | 31.7144 | 9.5975 | 125.7995 | 380.2766 | t/o |
| BI_3 | 0.9235 | 28.0367 | 8.0141 | 343.0650 | 37.0746 | 1,601.4695 | 1,151.5091 | t/o |
| BI_4 | 0.0270 | 1.7952 | 0.1929 | 0.8106 | 1.1765 | 2.3326 | 1.4743 | 116.6172 |
| BI_5 | 0.0755 | 47.5976 | 0.1433 | 69.4361 | 0.2512 | 125.4889 | 28.0933 | 541.0417 |
| BI_6 | 0.0170 | 0.3387 | 0.0330 | 2.4841 | 0.1263 | 7.5908 | 17.7641 | 93.5088 |
| BI_7 | 0.3453 | 511.2457 | 4.9366 | t/o | 20.3964 | t/o | 1,059.4062 | t/o |
| BI_8 | 0.0365 | 0.5524 | 0.2437 | 4.5672 | 1.0219 | 20.9287 | 1.9114 | 26.6252 |
| BI_9 | 0.0735 | 7.0492 | 1.2107 | 87.3451 | 13.6357 | 523.3391 | 363.1347 | t/o |
| BI_10 | 0.5125 | 1,364.4088 | 7.2261 | t/o | 43.8146 | t/o | 22.7974 | t/o |
| BI_11 | 0.4874 | 5.0742 | 3.5283 | 38.1115 | 16.7220 | 109.6632 | 9.1296 | 29.3734 |
| BI_12 | 0.7324 | 11.2471 | 7.9241 | 155.4858 | 32.1192 | 659.7865 | 260.4788 | t/o |
| BI_13 | 1.2253 | **0.9713** | 11.6158 | **6.7296** | 9.4079 | 20.2515 | 115.4907 | 235.2280 |
| BI_14 | 0.8399 | 8.4844 | 7.8764 | 68.8087 | 7.6439 | 229.2400 | 165.1050 | t/o |
| BI_15 | 0.0060 | 0.0927 | 0.0128 | 1.0641 | 0.0326 | 3.2056 | 4.9700 | 857.0237 |
| BI_16 | 0.7085 | 5.0081 | 7.3629 | 62.8762 | 34.0740 | 277.9866 | 24.7435 | 1,081.0724 |
| BI_17 | 0.3550 | **0.2371** | 2.1243 | 61.7536 | 4.3261 | t/o | 1.7270 | t/o |
| BI_18 | 0.2373 | 7.8512 | 3.3700 | 68.7250 | 13.0443 | 294.7010 | 423.1270 | t/o |
| BI_19 | 13.9819 | 343.5706 | t/o | t/o | t/o | t/o | t/o | t/o |
| BI_20 | 0.1641 | 6.5713 | 6.8944 | 77.6586 | 28.3382 | 399.9949 | 220.0690 | 1,081.9585 |
| BI_21 | 0.0708 | 0.4676 | 0.5054 | 3.3095 | 2.8448 | 16.2169 | 3.5018 | 174.1151 |
| BI_22 | 0.2688 | 890.0080 | 2.9528 | t/o | 19.0996 | t/o | 61.0383 | t/o |
| BI_23 | 0.0226 | 0.2429 | 0.1133 | 1.9604 | 0.3400 | 7.8020 | 164.4104 | 2,163.2589 |
| BI_24 | 0.2008 | 15.7829 | 1.9883 | 193.4047 | 55.7750 | 1,051.4582 | 465.3403 | t/o |
| BI_25 | 1.8657 | 451.0648 | 2.3218 | 325.5987 | 2.6213 | 5.2952 | 3.9553 | 6.2800 |

表 3：所有查询和所有数据集数据的执行时间（秒）。t/o 代表超时，即在 18,000 秒内没有执行完毕。粗体值表示 Neo4j 速度超过 TigerGraph。这在性能测试中所占的比例为 3.5%。因此，按照 LDBC SNB 基准测试，TigerGraph 显然比 Neo4j 更优秀。

## 5 总结和未来研究

在本研究中，我们提供了对两个原生图数据库系统 Neo4j 和 TigerGraph 完整实施 LDBC SNB 基准测试后的初步结果。我们不但周密评估了在三种计算架构和四种数据集下执行基准测试中的全部 46 个查询的性能，还测量了批量加载时间和存储容量。我们的结果表明，TigerGraph 在绝大多数查询中（超过 95% 的性能测试）的表现始终优于 Neo4j。因为只有 TigerGraph 能够在数据集扩展到 SF-1000 时执行查询，所以随着数据量的增加，两个系统间的差距也在扩大。在 25 个 BI 查询中，Neo4j 在合理时间内仅完成了 12 个。然而，Neo4j 在批量加载图数据时速度一般较快（如果忽略索引生成时间），而且声明式查询语言较为简洁。为了鼓励结果重现，我们在网上公布了所有代码、脚本和配置参数 [16, 28]。未来，我们计划将更多系统纳入研究范围，包括图数据库和关系数据库。

**鸣谢** 我们感谢 TigerGraph, Inc. 为此次研究提供的支持。这种支持有两种形式。首先，TigerGraph 工程师积极参与了 GSQL 语言的 SNB 性能测试的优化。其次，Zhiyi Huang 得到了 TigerGraph 基金的资金支持。尽管如此，本报告提出的研究成果完全归功于作者。

## 参考资料

[1] R. Angles, P. Boncz, J. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martinez-Bazan, V. Kotsev, and I. Toma. The Linked Data Benchmark Council: A Graph and RDF Industry Benchmarking Effort. *ACM SIGMOD Record,* 43(1), 2014.

[2] R. Angles, M. Arenas, P. Barcelo, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, and H. Voigt.G-CORE: A Core for Future Graph Query Languages. In *SIGMOD 2018*.

[3] A. Deutsch, Y. Xu, M. Wu, and V. Lee. TigerGraph: A Native MPP Graph Database. *arXiv: 1901.08248*, 2019.

[4] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat-Perez, M.-D. Pham, and P. Boncz. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD 2015*.

[5] A. Iosup, T. Hegeman, W.L.Ngai, S. Heldens, A. Prat-Perez, T. Manhardt, H. Chafi, M. Capota, N. Sundaram, M. Anderson, I.G. Tanase, Y. Xia, L. Nai, and P. Boncz. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *PVLDB, 9(13),* 2016.

[6] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. GraphLab: A New Framework for Parallel Machine Learning. In *UAI 2010*.

[7] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-Scale Graph Processing. In *SIGMOD 2010*.

[8] M. Needham and A.E. Hodler. Graph Algorithms—Practical Examples in Apache Spark and Neo4j. O'Reilly, 2019.

[9] A. Pacaci, A. Zhou, J. Lin, and M.T. Ozsu. Do We Need Specialized Graph Databases? Benchmarking Real-Time Social Networking Applications. In *GRADES@SIGMOD 2017*.

[10] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: A Property Graph Query Language. In *GRADES@SIGMOD 2016*.

[11] I. Robinson, J. Webber, and E. Eifrem. Graph Databases—New Opportunities for Connected Data, 2nd Edition. O'Reilly, 2015.

[12] G. Szarnyas, A. Prat-Perez, A. Averbuch, J. Marton, M. Paradies, M. Kaufmann, O. Erling, P. Boncz, V. Haprian, and J.B. Antal. An Early Look at the LDBC Social Network Benchmark's Business Intelligence Workload. In *GRADES-NDA@SIGMOD 2018*.

[13] M. Wu. A Property Graph Type System and Data Definition Language. *arXiv:1810.08755, 2018.*

[14] Apache Giraph. `https://giraph.apache.org/`.

[15] Apache TinkerPop: The Gremlin Graph Traversal Machine and Language. `https://tinkerpop.apache.org/gremlin.html`.

[16] Z. Huang.LDBC SNB Benchmark. `https://github.com/zhuang29/graph_database_benchmark`.

[17] JanusGraph. `https://janusgraph.org/`.

[18] Linked Data Benchmark Council (LDBC). `http://www.ldbcouncil.org/`.

[19] LDBC Social Network Benchmark (SNB). `http://ldbcouncil.org/benchmarks/snb`.

[20] LDBC SNB Data Generator. `https://github.com/ldbc/ldbc_snb_datagen`.

[21] LDBC SNB Documentation. `https://github.com/ldbc/ldbc_snb_docs`.

[22] LDBC SNB Implementations. `https://github.com/ldbc/ldbc_snb_implementations`.

[23] Neo4j. `https://neo4j.com/`.

[24] Neo4j Cypher Query Language. `https://neo4j.com/developer/ cypher-query-language/`.

[25] Amazon Neptune. `https://aws.amazon.com/neptune/`.

[26] TigerGraph. `https://www.tigergraph.com/`.

[27] TigerGraph GSQL Query Language. `https://www.tigergraph.com/gsql/`.

[28] TigerGraph GSQL Queries for LDBC SNB. `https://github.com/tigergraph/ecosys/tree/ldbc/ldbc_benchmark/tigergraph/queries`.