# GSQL:

# 一种类SQL的图查询语言

Mingxi Wu，TigerGraph研发部门副总裁
Alin Deutsch，加州大学圣地亚哥分校教授

1

# 简介

由TigerGraph开发的GSQL是一种高阶且图灵完备的查询语言，可用于编写图分析表达式。GSQL具有类似SQL的语法，这使得对于SQL程序员来说，学习曲线相当平缓。但同时它又支持NoSQL开发人员所偏爱的Map-Reduce编程模型，并且兼容可扩展的大规模并行求值。

2

# GSQL概述

GSQL通过图形遍历完成查询：它沿着遍历路径（以数据集或数据聚合的形式）收集数据，并将它们或是集中存储，或是分散存储在其访问过的顶点上。

GSQL查询的主要构成块被叫做顶点块（vertex blocks）和边块（edge blocks）。基于输入的顶点集合以及布尔条件（我们称之为控制条件），这种指定的计算可以在图上的顶点和相应的边上独立运行。这种计算的独立性为并行化的实现创造了很大潜力。

为了维持这种并行处理的潜力，数据经“顶点/边”运算生成后，被存储在累加器中。累加器是一种数据类型，它能够将并行写入的数值集合进行聚合。这种聚合通过反复调用二元运算符⊕来进行，并将每个新得到的值写入累加器的当前值中。累加器分为两种：全局累加器将运算结果集中存放；而顶点累加器则支持将结果分散存储在块运算相关的顶点，以及任何已知ID的目标顶点之中。

输出顶点集可作为后续块的输入顶点集。这使得GSQL查询成为块链，更普遍的说，是块的有向无环图。

GSQL的表达能力通过标准的控制流原语来增强（例如If-Then-Else分支和while循环），并且由一系列控制条件进行控制。这些条件包含了全局累加器中的值，也就是说，它们取决于在查询求值中动态运算后得到的数值。

GSQL的一个非常强大的特性是它能够定义“命名参数化查询”。它允许从其它查询中调用该查询，并传递参数（支持递归调用）。GSQL的这种“查询-调用-查询”的能力类似于甲骨文公司的 PL/SQL模块，以及微软SQL Server的Stored Procedure模块。

拥有多功能累加器库、基于累加器的控制流条件以及“查询-调用-查询”的功能，使得 GSQL已经达到了“SQL-完备”的程度（在第四节中定义），并实际上也达到了“图灵-完备”的程度。

1 GSQL附带一组预定义的累加器类型，各自对应SQL的标准聚合运算符（例如sum、count、min、max、avg等）。此外，它也支持用户定义累加器类型，用户需要为其提供一个⊕的实现。

3

# GSQL示例

假设我们建立一个图数据库，其顶点类型包括产品（Product）和客户（Customer）。产品顶点的属性包括每个产品的品名、类别和单价。客户顶点的属性则包括客户SSN、名称、地址。在这个示例中，假设客户c购买了产品p，则该行为被建模为一条从c到p的边，类型为购买（Bought）。
除此之外，“购买”类型的边还包含了其它属性——即价格、折扣和购买的数量。

**示例3.1** 假设我们想要找出的是：针对每一个客户c，从 “玩具（toy）”类产品到c的总销售额。为了达成这个目的，我们需要在“玩具”类别中找到所有相关的“出边b”，并且其类型为购买（Bought），它的源顶点为客户顶点c，目标顶点为产品顶点p。对于每个这样的边b，我们将其包含的单位产品折后价（〔1–b.discount/100.0〕*p. price〕乘以该产品购买数量（b.quantity）得到最终售价。需要注意的是，每条边上的运算都是独立且并行运行的。每条边b的运算结果被写入到顶点c的累加器中，然后将所有的运算结果汇总即可得到总销售额。下面的查询示例输出的是每个客户的姓名和其对应的销售额。

```
SumAccum<float> @revenue;                                                    (1)
Start = {Customer.*};                                                        (2)
Cust =   select   c                                                          (3)
         from     Start:c -(Bought:b)-> Product:p                            (4)
         where    p.category == "toy"                                        (5)
         accum    c.@revenue += p.price*(1-b.discount/100.0)*b.quantity;     (6)
print Cust.name, Cust.@revenue;                                              (7)
```

该查询从定义顶点累加器开始。累加器用于保存销售额运算结果。第（1）行中，每个顶点都配置了一个叫做revenue的累加器。该累加器将浮点数值聚合成一个sum值（⊕ 以算术加法方式实现）。

第（3）行到第（6）行定义了一个边块（edge block）。该边块的From语句〔第（4）行〕定义了这些边的起点为Start顶点集（由第（2）行初始化，包含了所有客户顶点），终点为产品顶点，边类型为“Bought”。该From语句同时也声明了几个变量：变量b定义为边，变量c定义为b的源顶点，变量p为b的目标顶点。

Where子句（第（5）行）指定布尔类型的控制条件，该条件确定哪些边参与边块的计算（即目标顶点的类别属性值为“toy”的边）。我们将这些符合条件的边称为“被允许的边”（allowed edges）。

Accum子句〔第（6）行〕在每个“被允许的边”上执行。它将表达式右侧算式的运算结果〔即该条边上的总销售额〕写入累加器中。

Select子句（第（3）行）也在每个“被允许的边”上执行。它指定了该查询的输出顶点集。示例中的块返回的是客户顶点的集合，该集合里是所有“被允许的边”的目标顶点。输出的顶点集被赋值给变量Cust。

最后，针对Cust变量表示的顶点集中的每个顶点，PRINT子句打印出其对应的客户名称属性和累加器里的营业额数值。

**复值累加器** 到目前为止，我们已经展示了累加器如何将数字表示的信息聚合为单一的标量值。GSQL的表达能力之所以十分强大，便是得益于其对于复值累加器（complex-valued accumulator）的支持。复值累加器包含了多元组集合，它们可以被划分成组，并且可以选择是否按组别进行聚合。

**示例3.2** 我们修改了销售额聚合部分的代码，使之可以显示对于每个客户c，按产品门类划分的销售额以及按折扣划分的销售额。

```
GroupByAccum<string categ, SumAccum<float> revenue> @byCategory;                    (1)
GroupByAccum<int disc, SumAccum<float> revenue> @byDiscount;                        (2)
Start = {Customer.*};                                                               (3)
Cust =  select   c                                                                  (4)
        from     Start:c -(Bought:b)-> Product:p                                    (5)
        accum    c.@byCategory += (p.category -> p.price*(1-b.discount/100.0)*b.quantity),   (6)
                 c.@byDiscount += (b.discount -> p.price*(1-b.discount/100.0)*b.quantity);    (7)
```

在第（1）行中声明了顶点累加器“byCategory”。这些按组划分（group-by）的累加器将组键（group key）定义为字符串“categ”。同时，将每个组关联到一个求和累加器“revenue”，以负责聚合组内数值。第（6）行演示了如何把值（p.price*（1-b.discount/100.0）*b.quantity）写入对应的“revenue”求和累加器之中。“revenue”求和累加器对应的组键为p.category，该组键位于顶点c上的byCategory累加器中。

类似地，第（2）行定义了按照折扣分组的累加器，让销售额数字按折扣值分组，并将每组结果加总求和。
为了操作集合类型的累加器的值，GSQL还提供了一个foreach的原语，它可以遍历集合内的所有元素，并将该元素的值绑定到一个迭代变量。具体用法请参看教程。

**包含中间结果流（Intermediate Result Flow）的多步遍历** GSQL支持在多步遍历某个图的过程中，携带该路径上计算出的中间结果。下面的例子说明了如何在GSQL中编写一个简洁的推荐系统。

**示例3.3** 以下GSQL查询最终会生成客户1的玩具推荐列表。计算结果按照推荐系统的经典方式排序：即每个被推荐的玩具的排名基于其它顾客对其喜欢程度的加权求和。每个客户c喜欢某个玩具的程度的加权值由客户c与客户1的相似度决定。相似度由标准的对数-余弦值（log-cosine）表示，它反映了客户1与客户c共同喜欢的玩具有多少。

更正式的，对于每一个客户c，我们定义一个比特向量likesc，当且仅当客户c喜欢玩具i时，其对应的bit i才设置为1。则两个客户x和y之间的对数-余弦值相似度被定义为：

$$lc(x, y) = \log(1 + likes_x \odot likes_y)$$

其中的 • 表示两个向量的点积。请注意，$Likes_x \bullet likes_y$简明地计算出共有多少玩具同时被客户x和y喜欢。

下文中，我们假设客户c喜欢了产品p的事实，被建模为一条连接c到p的唯一的边，边的类型为“likes”。

```
SumAccum<float> @rank, @lc;                                    (1)
SumAccum<int> @inCommon;                                       (2)
Start = {Customer.1};                                          (3)
ToysILike =                                                    (4)
        select      p                                          (5)
        from        Start:c -(Likes)-> Product:p               (6)
        where       p.category == "toy";                       (7)
OthersWhoLikeThem =                                            (8)
        select      o                                          (9)
        from        ToysILike:t <-(Likes)- Customer:o          (10)
        where       o.id != 1                                  (11)
        accum       o.@inCommon += 1                           (12)
        post-accum  o.@lc = log (1 + o.@inCommon);             (13)
ToysTheyLike =                                                 (14)
        select      t                                          (15)
        from        OthersWhoLikeThem:o -(Likes)-> Product:t   (16)
        where       t.category == "toy"                        (17)
        accum       t.@rank += o.@lc;                          (18)
RecommendedToys = ToysTheyLike MINUS ToysILike;                (19)
print RecommendedToys.name, RecommendedToys.@rank;             (20)
```

该查询从客户1开始遍历，然后在第一个分支语句中沿着边“Likes”找到她喜欢的玩具（第6行），并将结果存储在ToysILike顶点集（第4行）中。在第二个分支语句中，遍历继续从这些玩具开始，沿着边“Likes”的反向前进，找到其他也喜欢这些玩具的客户（第10行）。这些被找到的客户被存储在顶点集“OthersWhoLikeThem”中。接着，在第三个分支语句中，遍历继续从这些顾客出发，沿着边“Likes”找到他们喜欢的玩具（第16行），结果存储在顶点集合“ToysTheyLike”中。请注意一下我们是如何使用某个块的输出顶点集作为后继块的输入顶点集，从而将块与块链接在一起的。

最终推荐玩具列表是由顶点集“ToysTheyLike”和“ToysILike”的差值得到，这样我们就避免向顾客1推荐她已经喜欢的玩具。

伴随着这个遍历过程，数据聚合运算过程如下：第二个语句分支计算客户o与客户1共同喜爱的玩具数量。它的实现方式为，对于每个客户o喜欢的玩具，将数值1写入o的顶点累加器“o.@inCommon”之中（第12行）。这些值由顶点累加器求和，最终的结果表示期望计数。在post-accumulation阶段，累加器o.@inCommon的值为最终值，此时可以计算客户o与客户1的对数-余弦相似度（第13行）。然后，第三个语句分支将每个喜欢玩具t的客户的相似度值“o.@lc”写入到每个玩具的t之中（第18行）。最终，“t.@rank”记录了上述所有结果的总和，非常精确地表示了最终玩具推荐的排名。

请注意，一个分支语句中的中间计算结果可通过累加器被后续块访问。（例如，客户o对于客户1的对数-余弦相似度被第二个分支语句存储到顶点累加器@lc中，随后被第三个分支语句读取用于计算玩具排名）。

**控制流** GSQL的全局控制流原语包括if-then-else形式的分支（也包括从SQL借用的标准case语句）和while循环。两者都可由涉及全局累加器、全局变量和查询参数的布尔条件控制。这尤其适用于编写迭代计算，其中每个迭代都会更新全局的中间计算结果。循环控制根据该中间计算结果决定是否退出循环。一个典型的例子是PageRank算法变体。该算法持续迭代直到达到最大迭代数，或者直到所有顶点上的得分计算最大误差（在全局Max累加器中聚合）低于一个给定阈值（例如作为参数提供给查询的值）。有关GSQL PageRank的实现，请参阅在线教程。

循环可自由嵌套，最内层循环的代码是块的有向无环图（DAG）（循环过程允许中间数据在图上传递）。

**查询-调用-查询** GSQL支持某一查询调用其他具有名称和参数的查询。详情请见在线教程。

**图更新** GSQL支持类似SQL语法的顶点级、边级和属性级修改（插入、删除和更新）。详细请见在线教程。

**GSQL 教程** 关于GSQL的详细教程，请查阅在线教程。

**未来会发布的"语法糖"（Syntactic Sugar）** GSQL在未来会有的改进之一，是计划让其可以在同一个FROM子句中可定义多个多步路径。但这只是个简单的"语法糖"（即不会带来功能方面的提升），因为该遍历过程目前通过多个语句块也可以实现。

**示例3.4** 例如对于示例3.3中来说，其第（4）行到第（13）行可被更简洁地重写为：

```
ToysILike, OthersWhoLikeThem =
        select      p, o
        from        Start:c -(Likes)-> Product:p <-(Likes)- Customer:o
        where       p.category == "toy" and o != c
        accum       o.@inCommon += 1
        post-accum  o.@lc = log (1 + o.@inCommon);
```

4

## 与其他语言比较

我们将GSQL与当今主流的其它图查询语言进行比较，忽略掉特定的句法和定义语义的方式，聚焦于以下几个关键维度分析语言的表达力：

**1、聚合：** 该语言如何存储查询运算得到的数据（包括数据集合和数据聚合）？

**2、多步路径遍历：** 该语言是否支持多步路径遍历，并能够在遍历过程中收集数据吗？

**3、运算中间结果数据流：** 该语言是否支持遍历过程中的中间计算结果在路径上传递？

**4、控制流：** 该语言支持哪些控制流原语？

**5、查询-调用-查询：** 该语言支持一个查询调用另一个查询吗？

**6、SQL完整性：** 该语言是SQL-完备的吗？即对于一个图数据库G来说，任何一个关系型数据库D上的SQL查询都能够在G上用GSQL表示出来吗？

**7、图灵-完备性：** 该语言是图灵完备吗？

4.1

## Gremlin

Gremlin是一种图查询语言，它将图分析设定为作业的管道（pipelines of operations），管道内流动的内容是对象（叫做遍历者“traversers”）流，它们沿着遍历途径演进并收集信息。例如，某次作业可以（在“遍历者”之中）将一个顶点替换成它的内或外邻居、内外边或其属性；同时，由“遍历者”内容计算出来的值，也会被赋值到变量中，帮助“遍历者”延展遍历范围。Gremlin的这种语义以作业为核心，并具有功能性的编程风格。

Gremlin是图灵-完备的语言，这点上它的表达能力和GSQL一样棒。参考Apache tinkerpop3 v3.2.7关于它在表达力方面的优势，请阅读Apache TinkerPop3 v3.2.7 documentation。

1、**数据累加** Gremlin支持数据集（列表或表格的形式）或将数据聚合；聚合可采用用户自定义的聚合函数，亦或是直接采用SQL内置的通用聚合函数。Gremlin同样支持按照组别聚合，只需要通过指定分组属性和聚合列定义一个关系型表即可实现。

   - Gremlin从SQL中继承了一个局限性：像SQL查询一样，Gremlin查询可以通过分组属性列表（group-by attribute list）来对表分组，但同一时间只能使用一个分组属性列表。 如果想在同一组数据上，同时按照两个分组属性列表分组，则需要运行详细查询（verbose query）；该详细查询将同一张变量表绑定两次，然后分别对每张表进行分组。与此相比，在GSQL中实现上述功能则更加简单，因为同步的运算过程会将两组同样的数据分别写入两个不同的组累加器中。

   - 遍历过程中计算中间结果，可以通过“副作用”（“side-effect”）操作符（类似于GSQL的全局累加器）存储在全局变量中，亦或是存储在顶点/边的属性（又名properties）中。

2、**多步路径遍历** Gremlin的设计初衷，是为了简洁地把多步线性路径，定义为一个包含了许多单个操作步骤的作业管道。
3、**中间结果流** 中间结果可以通过被绑定在变量上的方式存储在“遍历者”对象上，然后沿着管道流转（GSQL也有类似的方式，只不过GSQL中数据通过累加器传送）。
4、**控制流** Gremlin同时支持if-then-else逻辑和loop循环，其控制条件不仅可以指定全局中间结果，也可以是“遍历者”中的原生数据。
5、**查询-调用-查询** Gremlin允许用户定义可互相调用的函数，包括递归。
6、**SQL和图灵完备性** Gremlin是图灵-完备的，因此也是SQL-完备的。

**关于编程风格的注意事项** Gremlin的一个特点为，顶点和边的属性（property）被建模为一张属性图（property graph），并且需要指定属性值的方向。若要访问多个属性，则需要在属性图中执行分支导航（branching navigation）（即从一个给定的顶点或边开始，我们需要沿着一个导航分支前往其每一个属性值）。

**另一个特点是，优化过的语法可以简洁地表达线性（非分支）路径导航（linear path navigation）。** 分支导航通过详细（verbose）的方式实现：首先将变量绑定在分支顶点处的顶点上，然后沿着辅分支作另一次线性导航，最后跳转回分支顶点并回到主分支上。

这两个特征综合后会产生两个效果：即当编写需要访问同一个边或顶点的数个属性的查询时，导航过程会变得蜿蜒曲折，查询表达式也会变得复杂冗长。 下面列出了如何将示例3.1重写成Gremlin的表达方式。

```
toys =
      V().hasLabel('Customer').as('c')                                    (1)
      .values('name').as('name')                                          (2)
      .outE('Bought').as('b')                                             (3)
      .values('discount').as('disc')                                      (4)
      .select('b')                                                        (5)
      .values('quantity').as('quant')                                     (6)
      .select('b')                                                        (7)
      .outV()                                                             (8)
      .has('Product', 'category', eq('toy'))                              (9)
      .as('p')                                                            (10)
      .values('price').as('price')                                        (11)
.select('price', 'disc', 'quant')                                         (12)
.map{it.get().price*(1-it.get().disc/100.0)*it.get().quant}               (13)
      .as('sale_price')                                                   (14)
.select('name','sale_price')                                              (15)
.group().by('name').by(sum())                                             (16)
```

上述程序执行了以下步骤：（1）检查顶点是否具有标签“Customer”，如果是，则将其绑定变量c；（2）前往它的“name”属性，将变量name绑定到该属性；（3）前往出发边“Bought”，并将变量b绑定到该边；（4）前往b的“discount”属性的值，将其绑定变量disc；（5）回到边b；（6）前往b的属性中的“quantity”的值，绑定到变量“quant”；（7）回到边b；（8）前往b的目标顶点；（9）检查它是否有标签“Product”以及属性“category”的值是“toy”；（10）将变量p绑定到这个“Product”顶点；（11）前往“price”属性的值，将其与变量price绑定；（12）输出三元组，组件分别为变量price， disc和quant；（13）对于每个元组，将其绑定到变量it，并按如下方法计算每个订单的金额：执行定义的方程，该方程作为参数传递给map；（14）将变量sale_price与步骤（13）中计算出的值绑定；（15）输出二元组，其组件分别为变量name和sale_price；（16）将这些元组按“name”进行分组，并在“sale_price”栏中计算总额。

请注意，“price”，“discount”和“quantity”是分支的属性，每次访问它们过后都要返回边b（例如第6行和第8行）。[2]

还要注意，在Gremlin中，能够绑定变量的表达能力对于编写这个查询（以及其它我们要运行的查询）至关重要。目前的一些系统，例如亚马逊的Neptune，仅支持无变量的Gremlin表达式（参见用户指南）而并不包含上述查询的规范。

**最后请注意，若想将示例3.2也重写成Gremlin的格式会有一个额外的障碍：像SQL查询一样，Gremlin查询可以通过一个分组属性列表对一个表进行分组，但是不能同时针对两个分组属性列表分组**。因此，像示例3.2那样，既按照“类别”又按照“折扣”的分组查询，就需要用到详细查询。该详细查询定义同一个变量绑定表（table of variable binding）两次，每个变量绑定表都有它自己的分组，最后把两者拼接或者合并（合并的运算结果比较精简）。在GSQL中则更加简单，因为它可以将两组相同的数据写入两个独立的分组累加器中同步运算（示例3.2中的第6行和第7行）。

[2] 引自TinkerPop3 documentation（2018年3月）：“在早期版本的Gremlin-Groovy中，有许多语法糖，用户可以依靠它们使其遍历更加简洁。这些规则中，有很多都使用了Java反射，因此并不追求高性能。而在TinkerPop3中，这些语法糖都已经被移除，使其支持标准的Gremlin-Groovy语法，从而既符合Gremlin-Java8语法，又始终保持最高性能。 然而，对于那些愿意以性能损失为代价使用语法糖的用户，还有SugarGremlinPlugin（又名Gremlin-Groovy-Sugar）插件可以使用。”在TinkerPop3中取消的语法糖里，有一个是方便程序员更简单写入v.name的语法糖。该语法糖可以让所有需要写入顶点v的name属性值的地方使用v.values（“name”）。

## 4.2 Cypher

Cypher是一种基于变量模式的声明性图形查询语言，每种模式（pattern）匹配都会产生一个变量绑定的多元组。这些多元组的集合是一个可以像SQL一样操作的关系表。

1. **数据累加** Cypher对累加的支持是有限制的。它支持收集数据到列表（List）或表（table）中，并且与Gremlin一样，通过定义关系表并指定其分组属性和聚合列，便可以以SQL的方式分组聚合数据。**在遍历期间计算的中间结果可以存储到顶点或边的属性（又名properties）中，但它们只能是标量（scalar）或标量列表（list-of-scalar）的类型，因为它们是Cypher中唯一的属性类型**。相比之下，GSQL可以使用累加器在顶点存储各种类型的数值（例如元组集合，包括set，bag，list，heap，map）。

尤其是，示例3.2中的查询需要在客户顶点存储多个元组（每个类别一个，每个折扣值一个），但在Cypher中类似实现是没有的。 对于这两个分组中的任何一个，Cypher能够做到最接近的是得到一个查询结果，这个结果不是将每个客户顶点的值累加并存储起来，而是构建一个将客户顶点id与每个分组相关联的表。这个Cypher的近似算法值也继承了另一个来自SQL的限制：与SQL查询相似，Cypher查询同时只能按照一个分组属性列表进行分组，不能同时指定两个分组属性列表。 因此，按类别分组和按折扣分组需要将同一个变量绑定表定义两次，确保每个分组都是恰当的，最后组合在一起：

```
MATCH  (c:Customer) -[b:Bought]-> (p:Product)
WITH   c, p.category as cat,
       sum (p.price*(1-b.discount/100.0)*b.quantity) as rev
WITH   c, collect({category:cat, revenue:rev}) as byCateg
MATCH  (c) -[b:Bought]-> (p:Product)
WITH   c, byCateg, b.discount as disc,
       sum (p.price*(1-b.discount/100.0)*b.quantity) as rev
RETURN c, byCateg, collect({discount:disc, revenue:rev}) as byDisc
```

在GSQL中，上述表达式则更加简单。GSQL通过简单地将每个值写入两个不同的分组累加器，同时计算两组相同数据。 请注意，为了进行有效评估，Cypher的优化器必须识别出两个Match子句中匹配的数据且可以将其重复使用，而不是冗余地重新计算它们。

2. **多步的路径遍历** 本着例3.4中说明的即将发布的GSQL语法糖的精神，Cypher可以定义带有变量的多步模式。
3. **中间结果流** 在路径遍历过程中，中间结果可以通过引用绑定到它们的变量访问
4. **控制流** Cypher的控制流包括if-then-else样式分支。 Loop循环控制是受限的，仅支持在迭代开始之前已经计算完成的元素列表上的循环。为了模拟给定迭代次数的循环，可以先创建一个相应的整数列表：“unwind range(1,30）as iter”，指定了一个30次迭代的循环，变量iter为迭代变量；“unwind L as e”，指定了一个列表L的遍历，迭代遍历为e。

5. 上文已经提及，在PageRank中迭代持续运行直到最大误差值低于某个阈值，需注意的是，这个过程需要循环的控制条件不断更新迭代中产生的中间结果。 这种循环在Cypher中并不支持，并且相应的算法类也无法表达。（例如，PageRank是作为Java实现的内置原语提供的）。
6. **查询-调用-查询** Cypher允许用户定义函数，但必须参照Neo4j开发者手册在Java中实现。 这种做法有两个问题。首先，程序编写者必须同时熟练使用Cypher和Java。其次，用户定义的函数对于neo4j优化器来说是黑盒，因此无法在调用查询的过程前后进行优化。
7. **SQL和图灵完备** Cypher是SQL-完备，但不是图灵-完备。上面提到的循环控制限制是该语言不能达到图灵-完备的原因之一。这里忽略了Neo4j支持的用户自定义函数。由于自定义函数是Java实现的，而Java是图灵完备的，所以这个问题对于JAVA来说就不是问题。

5

## 讨论

我们在这里还想着重指出GSQL的另一个区别于其他图查询语言的功能：得益于对优化和安全性的支持，**GSQL的数据模型定义可假定预先定义的图模型/元数据（schema/metadata）信息，一次存储即服务全局，并将顶点/边/属性各自分解出来。**

这不仅可以节省空间，而且还可以更好地评估优化计划时间，从而提高性能。在优化时间方面运用预定义的**图模型**是一种久经考验的数据库技术，能够一次又一次地带来性能优势。

此外，为了安全和隐私的目的，这种预定义的**图模型**，仅允许用户查询他们有权访问的图（或是图的一部分，由视图定义）。此功能在实际应用中非常重要，因为它实现了多用户功能（例如公司中的多个部门），即针对同一个图数据库，不同用户拥有不同访问权限。这是迄今为止所有图数据库中第一也是唯一一个支持此类应用的产品。

6

## 总结

至此我们已经介绍了GSQL的所有主要功能。它这是一种全新高阶、图灵完备、支持并行计算的图查询语言。我们已经写明了有关表达式能力的几个关键维度，并将它们与其他图查询语言进行比较。 GSQL拥有达到甚至超越了其它图形查询语言的表达能力。虽然基本范式的模样只是一个偏好问题，但我们正在尽一切努力，使得GSQL能够兼容SQL和NoSQL的Map-Reduce编程范式，使其对于上述两个程序员社区都具备吸引力。

扫码关注微信公众号下载试用版

## 关于TigerGraph

TigerGraph是基于原生并行图（NPG）技术的全球首个实时图分析平台。TigerGraph通过为具有复杂和海量数据的企业提供实时深度链接分析支持，实现图平台的真正承诺和好处。TigerGraph的成熟技术已经被蚂蚁金服、VISA、软银、中国国家电网公司、Wish、Elementum等客户所采用。TigerGraph由许晏博士于2012年创立，并获得启明创投、百度、蚂蚁金融、AME云创投、莫拉多风险投资公司、佐德·纳齐姆、丹华资本和DCVC风投基金公司的投资。TigerGraph的总部位于加利福尼亚州红木市。如欲了解更多信息，请访问www.tigergraph.com.cn。