图数据库及其查询编译优化的研究

摘要：图数据与图数据库在未来大数据发展中是逃不掉的必然发展方向，发展和优化图数据库中的查询操作也是一个具有基础性和艰巨性的任务与目标。在本论文中，我们首先对图数据库、图查询语言以及图数据库的应用等开展了基础的认知与探究；然后我们基于图的综述论文[1]来深入理解图、图数据库以及图数据库上的操作在四种维度上的前景与困境；接下来我们针对图数据库的查询编译优化，研究了一种集成图查询解释和编译的自适应方法[2]，深挖其背后的优化细节与手段；最后我们针对该方法提出一系列进一步优化的想法与思考。

关键字：图数据库，图查询语言，编译优化

1.引入

图数据在当今社会与实际应用中发挥了越来越重要的作用，相对于传统关系数据的组织形式，图数据将关系直接存储在边上，这导致它们在处理和分析数据之间的关系上有着天然的优势。研究和发展由大量图数据组成的图数据库及其应用是必然的方向。针对图数据库的发展和应用也必然不是一帆风顺的，它在四个维度上都有着挑战，也有着更进一步的希望。与此同时，针对图数据库的应用产生的各种数据操作也应运而生，图查询语言也随之发展出了各种不同版本。然而针对不同的图查询语言，我们最后依旧需要通过编译来执行，这个时候，图查询语言的编译优化来提高查询效率就显得格外重要。

2.图查询语言及其语义

2.1 图数据库

在计算机科学中，图数据库（英语：graph database，GDB）是一个使用图结构进行语义查询的数据库，它使用节点、边和属性来表示和存储数据。该系统的关键概念是图，它直接将存储中的数据项，与数据节点和节点间表示关系的边的集合相关联。这些关系允许直接将存储区中的数据链接在一起，并且在许多情况下，可以通过一个操作进行检索。图数据库将数据之间的关系作为优先级。查询图数据库中的关系很快，因为它们永久存储在数据库本身中。可以使用图数据库直观地显示关系，使其对于高度互连的数据非常有用。

2.2 图查询语言

从图数据库中检索数据需要 SQL 之外的查询语言，SQL 是为了处理关系系统中的数据而设计的，因此无法“优雅地”处理遍历图。截至 2017 年，没有一个像 SQL 那样通用的图查询语言，通常都是仅限与一个产品的。不过，已经有一些标准化的工作，使得 Gremlin、SPARQL 和 Cypher 成为了多供应商查询语言。除了具有查询语言接口外，还可以通过应用程序接口（API）访问一些图数据库。

2.3 图查询语言应用

在社会网络中，可以把用户看作为图的顶点， 用户之间的关系（如朋友关系）看作图的边。图的相关理论和技术在社会网络中有着重要的应用，是目前学术界和业界关注的热点之一。

2.3.1 社会关系查找

例如一个社交网络记录了华人之间的社会关系，其中网络的顶点是人，且顶点上带有属性值，用来记录人的姓名；边是人与人之间的各种人际关系，且边上带有属性值，用来记录所连接的两人之间的相应关系。有几类常见的社会关系查询：（1）查找给定两人是否是远房亲戚；（2）查找给定两人是不是三代以内的近亲；（3）查找给定人所有三代以内具有血缘关系的亲属；（4）查找并输出所有与张三有三代以内血缘关系的亲戚且是张三某个兄弟的老板。以上 4 种查询应用都可以用带边属性约束的可达性查询完成。

2.3.2 角色分析

为了分析员工对公司的重要性，公司需要了解哪些人的工作是可以相互替换的，即分析角色的等效性，而这是图匹配查询的一 个经典应用。公司的所有职员组成一个社会网络，他们之间的工作关系用有向边来表示。 我们可以通过基于图模拟的图匹配查询来找出任意两个员工之间的是否可被“模拟” 的关系。如果角色 X 能被 Y 所模拟，则证明 X 可以被 Y 替代。

2.3.3 推荐系统

在推荐系统中也常常会用到图匹配查询，如对新兴特定应用的高级推荐功能。例如构造一类面向领域专家的求职社会网络图 G2，其中顶点表示专家，顶点标签是专业领域，边表示良好的领导合作关系。 我们需要所选择的软件开发团队成员满足模式图 Q2 所示的合作关系 。至此，招聘合适专家构建软件开发团队就

转化成了图模式匹配问题了，即如何从图 G2 所示的领域专家社交网络中找出子图与模式图 Q2 匹配，以满足软件开发团队的各种要求。

2.3.4 交通路线

如果需要自己开车以最短的时间到达某市，则该问题可以用最短路径查询来表达。即模式图表达的约束是两点之间的最短路径。此外如果要考虑公共健康和安全，许多桥梁和铁路交汇处是不允许车辆通行的。这时，可通过含有特定路线约束（如正则表达式）的模式图来查找最优交通路线。

2.4 大图的前景与困境

大数据通常具有四个 V 特征：容量（Volume）、速度（Velocity）、多样性（Variety）和真实性（ Veracity）。当涉及到大规模图时，这些挑战将变得更加艰巨。每个 V 都引发了新的问题，从理论到系统和实践。

2.4.1 体积：并行计算问题

考虑图的 Q 类模式查询。给定一个查询 Q ∈ Q 和一个图形 G，我们希望计算图 G 中模式 Q 的所有匹配项的集合 Q（G）。模式匹配是根据子图同构来定义的，即使判断 Q（G）是否为空也很难。在现实世界中，图形很容易拥有数十亿个顶点和数万亿个边，这一工作量是巨大的。为解决这一问题，一个想法是使用基于图的模型，将现有的顺序算法在多台机器的集群上并行化。在满足一定条件的情况下，它保证并行化计算会正确地收敛到答案。对于图模拟等计算问题，与基于顶点的模型相比，基于图的模型在效率和编程的简便性方面表现更好。

2.4.2 速度：图算法的增量化

实际生活中的图经常被小规模的更新修改。假设我们已经计算出图 G 中模式 Q 的匹配 Q(G)。当 G 通过ΔG 进行更新时，我们需要计算更新后图 G ⊕ ΔG 中匹配 Q 的 Q(G ⊕ ΔG)，例如用于欺诈检测。一种批处理的方法是从头开始重新计算 Q(G ⊕ ΔG)，但对于大型图 G 而言很昂贵。另一种方法是使用增量算法 A Δ，它以 Q、G、Q(G)和ΔG 作为输入，计算出旧输出 Q(G)的变化ΔO，使得 Q(G ⊕ ΔG) = Q(G) ⊕ ΔO，并最小化不必要的重新计算。当ΔG 较小时，对 Q(G)的更新ΔO 也往往很小。增量方法通常比批处理方法更高效。但增量算法很难编写和分析。需要系统的方法来开发有效的增量算法。

2.4.3 多样性：跨关系和图的查询

即如何在关系数据库 D 和无模式图 G 之间编写 SQL 查询。研究这个问题的需求是显而易见的。尽管商业数据通常存储在关系数据库中，但图结构化的数据越来越常见。这就需要在 D 和 G 之间合成数据，并将它们有关同一实体的信息相关联。毕竟，大数据的增值来自于各种数据源。为此需要 SQL 连接的语义扩展。如果确定关系数据库 D 中的元组 t 和图中的 v 由 HER（异构实体对齐）引用同一个现实世界实体，则可以自然地“连接”两者，提取顶点 v 的相关属性，并丰富元组 t 的额外“属性”。

2.4.4 真实性：大图的质量和价值

现实生活中的数据常常是脏的。即使在广泛使用的知识图中，也常常发现重复和语义不一致。事实上，在生物医学知识图中的噪声被认为是药物发现的重大挑战。脏数据带来的代价是昂贵的。据估计，差数据质量每年给组织带来平均 1500 万美元的损失，并且仅在 2016 年就使美国损失了 3.1 万亿美元。基于脏数据的数据驱动决策可能比没有数据的决策还要糟糕。因此，需要进行数据清洗，以准确地检测和修复数据中的错误。

3.图数据库编译优化

图数据库图查询语言最后终归是需要被编译成机器代码才可供计算机执行的，那么这背后的图数据库查询的编译及其优化就值得我们研究与思考。我们研究的这种方法是一种自适应的图查询编译方法。我们深挖了该方法背后的优化机制与手段。本质上，该技术宏观的想法是将查询解释和编译集成到处理中。最初使用解释器，而编译在后台进行。编译完成后，执行切换到编译后的代码。

3.1 基础优化

3.1.1 图数据库中数据的存储结构

我们对数据的存储结构主要可以做以下优化：

分块存储组成单向链表：节点或关系记录以单向链表方式组织，由定长数据块组成。这种结构有利于顺序访问，因为每个块包含多个记录，扫描整个向量只需要遍历块之间的链接，无需进行复杂的数据结构遍历。因为是单向链表，这也利于数据块的增删操作，毕竟指针变换相对简单。

位图信息：每个块的每条数据中都包含一个位图信息，该消息用来标记哪些是非空的，以便有效跳过空的记录，从而提高顺序访问的效率。

稀疏索引：我们是分块存储信息的，每块中又有块内偏移量，通过维护一个稀疏索引，可以根据标识符快速定位任意块的第一条记录，进而遍历整个向量。

按顺序存储：节点、关系和属性向量在存储时分别按顺序记录，有利于基于偏移量的顺序访问，充分发挥空间局部性。

可以看出，我们主要通过利用和发挥顺序读写性能，采用各种手段设计和优化数据的存储结构，以提高图查询的性能。

3.1.2 图数据库关系代数拓展

我们通过前人的工作[3]引入和发展了这样一个一般图代数及其相应的等价关系。主要内容是通过三个主要图算子来扩展关系代数 。

NODESCAN：扫描图的节点。

FOREAHRELATIONSHIP：遍历给定方向（即入边或出边）的节点的所有关系。

EXPAND ：可以获得边（即关系）的源或目的地。

我们的主要目的是基于这些运算符达到可以实现其他查询语言的效果，例如 Cypher 或 GQL。并且因为它们类似于众所周知的关系代数，我们可以通过这组运算符毫不费力地编写各种查询。

举例：

Expand(OUT, "Person", ForeachRelationship(FROM, ":knows", NodeScan("Person")))

针对上面举例的查询语言，主要有以下三个执行过程：

首先，NodeScan("Person") 表示在图中扫描节点类型为 "Person" 的所有节点。

然后，ForeachRelationship(FROM, ":knows", NodeScan("Person")) 表示在从扫描的 "Person" 节点中，遍历所有传出的关系，其中关系类型为 ":knows"。这个运算符返回与每个关系相关联的节点。

最后，
Expand(OUT, "Person", ForeachRelationship(FROM, ":knows", NodeScan("Person"))) 表示将之前的结果与图中的所有 "Person" 节点进行组合，以获取与每个节点相关联的人员节点。

值得一提的是，我们可以看到上述过程中，这种语言中的单个操作符按照它们正在处理的正向顺序进行表述，而各个运算符之间则以相反的顺序进行。这是为了以便与下文中基于推送的查询处理兼容。

3.1.3 基于推送的查询

基于推送的查询的主要思想是尽可能地将查询计算移动到数据源端，从而减少数据传输和处理的开销，提高查询处理效率。

具体来说，基于推送的查询处理方式通常是从查询语句的最后一个运算符开始向前推送结果。在推送过程中，每个运算符都会接收上一个运算符的输出结果，并对其进行进一步的处理，然后将处理后的结果推送给前面的运算符。这样，整个查询过程就可以在数据源端进行，并尽可能地减少数据的传输和处理量。

而在图关系代数中，我们的运算符就是前文中我们定义的关系代数运算符。每个运算符将其结果（即生成的元组元素）附加到现有元组，形成元组元素列表。运算符可以从前一个运算符访问元组中的任何元素从而实现推送。元组将一直被推送直到到达终止点。

该方法利用 C++ 成功编写和实现了所有图代数运算符及其对应的函数，达到了基于推送的查询效果。这些函数和代码将用于我们的 AOT 编译的查询引擎，并且这些 AOT 编译的代码已经过高度优化，这使得我们很方便的运行高质量的查询。这些代码构建了我们后面会提到的解释执行模式的基本块。

3.1.4 并行性

在基于推送的查询的基础上，我们可以看出查询语句的执行能够被分解为多个线程（比如一个运算符一个线程）。每个线程负责执行查询语句中的一部分，并将计算结果推送给下一个线程。这样可以充分利用多核处理器的并行运算能力，同时减少数据传输和处理的开销。

该方法利用 Morsel 驱动（一种并行编程模型)的并行性来实现这样的并行化。一个大任务被分解成多个小任务（任务包），这些任务包可以并行执行。由线程拉取任务包来执行工作，即实际查询。每个任务包将负责处理被分配的一系列数据块。查询的最后一步是合并来自各个工作线程的所有结果并将它们返回给调用者。

通过以上并行化的设计和任务分配方式，引擎能够利用多核系统的并行性，同时处理多个任务包和块，提高查询的效率和吞吐量。每个工作线程在自己的核心上独立执行任务，从而实现了并行处理的效果。

3.2 三种查询及其优化

3.2.1 查询解释

查询解释是执行查询的一种简单方法，对于给定查询的每个运算符，都会调用适当的 AOT 编译函数。我们其实可以理解成一个解释器，并可以通过访问者模式来实现。对于每个运算符，我们至少提供一个 AOT 编译函数，该函数使用给定参数执行运算符。

虽然查询解释启动很快，但这种方法的缺点是在实现每个查询运算符时需要额外的工作，因为没有根据实际情况进行优化，查询解释执行速度相对比较慢。而且生成的代码很大程度上基于模板，并且需要运行时提供类型信息，这增加了数据库代码的开发工作量和维护难度。

3.2.2 查询编译

查询编译是一种容易想到的加速查询处理的技术。我们选择 LLVM 作为编译器后端，用于生成即时（JIT）查询代码。它提供了用于代码生成的低级 IR 语言，有强大完善的代码优化能力，能在运行时生成和编译代码（这是查询编译器的主要要求），并且 LLVM 支持多种架构。对此我们有如下几点优化：

1. 在寄存器中处理元组

我们期望尽可能长时间地在寄存器中处理元组结果，因为寄存器处的操作更快。这要求我们需要将生成代码的所有指令都内联在单个函数中。这对于在寄存器中尽可能长时间地处理元组而不具体化至关重要。由于基于推送的方法一次处理一个元组结果，因此实际元组是可以直接存储到寄存器中，并一直在寄存器中操作的，直到在有需要和必要的情况下我们才将原组具体化，即存储到内存/外存等介质中去。

2.预处理

分配内存是一个成本高昂的过程，并且可能会影响代码的最终性能。类型检查等操作也同样如此。我们可以通过对查询进行静态分析，可以获取查询涉及的数据结构、数据类型、元组元素类型以及查询计划中的操作信息。

这些信息可以在代码生成的过程中被编译器使用，以便提前为查询预留和分配足够的内存空间。并且查询中每个元组元素的类型信息在代码生成之前是已知的，从而可以省略一些显式类型检查，比如用于仅生成元组处理所需的代码。

3.一些固定逻辑查询操作的直接实现

对于某些查询操作，其逻辑是固定的且无需额外优化的（比如聚合运算），可以通过用 C++编写相应的代码来直接实现。这样可以避免将整个查询代码都生成为 LLVM IR。而这种操作是与 AOT 编译的查询引擎完全可以做到兼容的，我们可以实现的。

这种方法的优点是，通过使用 C++来实现一些固定逻辑的查询操作，可以提高代码的可读性和维护性。而且，由于不需要进行额外的优化步骤，可以节省编译时间和资源。

4 脏数据传播

为了尽可能隐藏 I/O 延迟，我们不需要将每次节点或关系信息的修改及时的在外存等介质中存储起来。例如，我们可以将更改的节点或关系在脏列表中进行管理，脏列表在 DRAM 内存中进行管理，以实现尽可能低的延迟。每当事务完成时，脏列表的记录才会传播到适当的存储介质中保存起来。

结合第一点我们可以发现，首先我们尽可能在寄存器处理原组结果，然后我们尽可能在内存中保存当前结果，最后我们才适时将结果传播到外存等介质中去。我们通过这种层层递进的方式来优化查询性能。

3.2.3 IR 代码生成的实现与进一步优化

1.查询管道

我们利用访问者模式为每个操作符生成适当的 IR 代码，并将完整的查询管道内联到单个 IR 函数中。因此， 一个函数对应一个查询管道，而一个查询管道由一系列操作符组成，这些操作符在访问者模式下生成 IR 代码。

换句话说，这意味着每个操作符会生成相应的 IR 代码，并按照一定的顺序连接在一起，形成一个完整的查询管道。在一个管道中每个操作符 IR 代码的输出将作为下一个操作符 IR 代码的输入，以此类推，最终得到查询管道的结果，以实现整个查询管道/函数的功能。

2.生成高效的 IR 代码

为了生成高质量的 IR 代码，该方法实现了不同的抽象，以方便 LLVM IR 代码中各种运算符的实现。 下面是一些举例：

循环是查询运算符代码中常用的控制流模式。生成的查询代码中基本上有两种类型的循环：带有计数器的 for 循环和头控制的 while 循环。对于这两种类型，该方法提供了一个高级接口来生成适当的代码。循环体可以作为 C++ 函数进行传递，其中将生成进一步的 IR 代码。

指针算术是查询代码中出现的另一种常见结构。由于 GEP（ GetElementPtr ）指令是处理元组时经常使用的指令，因此我们为此实现了进一步的抽象以简化开发。对于每个出现的记录字段，我们提供一个执行 GEP 指令来检索指向字段值的指针。我们从而能够进一步通过使用以指针作为参数的加载指令，将字段的值直接加载到寄存器中。

3.查询的启动

为了启动每个查询，我们将先扫描相应的存储介质，从而提取出我们需要的数据。该存储可以位于各种可能的存储介质上。对各个存储的调用我们是通过外部函数调用来执行的，类似于事务管理的调用。这种外部调用减少了生成 IR 代码的工作量，减少了重复代码，简化了程序结构。

节点或关系数据信息通过外部调用被单独加载到寄存器中。随后的运算符将访问相应的寄存器。由于 CPU 的寄存器可以被非常快速地访问，这提高了查询处理的运行时间（与前文相符合）。

4.编译优化

大致的优化策略与传统 IR 优化是是一致的。

具体的优化包括：将内存提升为寄存器即寄存器分配；控制流简化即必要时合并条件分支；死代码消除即删除未使用的代码；指令组合即组合相似或共享相同条件的指令；死存储消除即删除未使用内存的未使用存储操作。

执行这些后，生成的查询代码仅包含处理查询所需和所必要的指令。

5.代码缓存

当每次编译类似的查询时，即具有相同运算符但具有不同参数（如标签）的查询时，重复编译会对性能产生负面影响，这是完全没有必要的。为了解决这个问题，我们可以建立一个缓存机制来存储已编译查询的代码。它可以存储在磁盘或其他存储介质上。

开始编译前，我们通过关键字进行查询。如果已经存在编译后的代码，则将其用于处理查询。否则，将对查询进行新的编译，并将代码与其关键字一起存储起来。

使用缓存代码的总体工作流程与代码编译相同，只是这种缓存代码立即可用，可以通过 LLVM 框架直接执行，而不消耗更多的编译时间。

3.2.4 自适应查询

尽管上文中的编译的查询代码速度很快，但在执行仅涉及几个元组的短期运行查询时会出现问题。编译过程将消耗比实际查询运行时更多的时间。

1.前提：并行性

在前文中我们提到，利用 Morsel 驱动（一种并行编程模型)的并行性来实现并行化。在这其中，每块任务块之间是并行执行的。任务被分配时，只以它们启动的模式完成。这意味着任务在执行过程中不需要额外的信息或状态来完成自己的工作。换句话说，当一个任务被执行时，它会按照预定的方式进行处理，并且不需要与其他任务或模块进行通信或交换信息。每个任务在自己的模式下独立地完成工作。这种任务块的独立性是该方法后续内容的前提。

2.自适应方法

接下来展示该自适应查询方法的具体过程，这种方法通过在后台线程中将查询编译成机器代码来提高查询的执行效率。具体过程如下：

初始阶段，查询以解释模式执行（使用预先 AOT 编译好的代码进行解释），以便快速获取结果。同时，后台线程会对查询进行编译，并生成更高效的机器代码，为提高后续查询的执行速度做准备。当编译完成后，任务函数切换到新编译的查询代码去。 通过这种方式，可以充分利用查询编译的优势，提高整体查询处理的性能，并且通过并行处理解释查询和编译查询过程，进一步提升效率。

同时，这种查询处理模式的切换（即上述过程中从解释模式切换到新编译代码）的实现也是充分利用 Morsel 驱动实现的并行性。

3.该方法在不同存储介质下的优势

该方法对于我们隐藏不同存储介质下的 I/O 延迟有巨大的作用。这是由于延迟较高的存储类型会导致额外的存储访问延迟，从而使得我们有更多的处理时间来实现后台线程的编译过程。因此，在高延迟介质作用下，虽然在处理的早期解释阶段，处理器可能会花费更多的时间等待存储数据的到达，但是这也给了更多时间来进行后台编译和优化，大幅提高后续查询的性能。一旦数据到达并开始处理管道后面的操作符，编译器可以选择更快的编译模式来进一步优化代码，以提高整体性能。

于是这种方法最终在整体上实现了有效隐藏磁盘等存储介质的延迟，使得查询操作达到在各种介质上都快速且稳定的效果。

4.自适应方法中的代码缓存

我们同样可以使用普通查询编译中的代码缓存的思想与手段。但是值得注意的是，我们检索代码的过程也会带来开销，且具有一定的失败率。对此，我们选择以与编译类似的方式处理此开销。即查询引擎从解释阶段开始，同时后台线程先从缓存中检索代码开始。这有效地隐藏了额外的代码缓存开销并进一步提高性能。

3.2.5 该方法最终效果

在单线程情况下，JIT 编译的方法优于 AOT 编译（单线程无法自适应）；

在多线程情况下，自适应的方法更优。首先大多数情况下自适应查询时间更短，即使在少量情况下也有同等耗时的优势性，其次在高延迟介质上自适应查询达到了更接近在 DRAM 内存上效果的效果，有效隐藏了不同介质的差异性；

4.从该方法展开的问题与想法

根据该方法的优化思路与实践，我们提出了一些更进一步优化的想法或问题：

1.在图的速度维度上我们有提到增量操作，这同样可能运用到图查询编译中。具体来讲，在实际的编译优化中，我们的代码缓存只是在有查找相同关键字（操作符）的代码，从而实现替换参数的快速编译效果，否则将从头编译该查询操作。我们可以考虑这样一种方法，即将查找范围拓展至关键字（操作符）相近的查询操作，对于相近的查询操作我们期望有某种增量操作可以通过在缓存中查找到的这个相近操作的优化代码快速得到我们所需的优化代码。

值得注意的是，第一，这种针对图数据库增量操作或许是不平凡的，也或许实际的消耗远大于我们直接从头编译消耗的时间，达到得不偿失的效果，所以这种想法需要更进一步的研究和实验来验证；第二，如果有这样的操作，我们能更进一步可以探究这样一个问题：一系列可增量的操作代码是否只需要缓存其“主代码”，其他相近代码根据缓存中的主代码进行增量操作即可，这与全部缓存相比哪种更有优势？

2.缓存编译是根据我们在实际运行中将生成的优化代码缓存起来的，然后在后续编译中进行查找。那么我们是否可以提前缓存一些常见的查询操作的优化代码在预缓存于其中？这能省略前期一些常见操作的编译时间以及初始阶段空查找产生的无用时间。

另外缓存代码的机制与我们 OS 中虚拟缓存机制相同，会有使用频率的高低之分和缓存空间的大小限制，那么仿照虚拟缓存的进出机制，及时清理不用或用的少的缓存代码等一系列手段值得我们深入研究。

3.虽然该方法的查询编译及其优化是在图数据库的情景下实现的，但是其中的思路和方法在其他的关系数据库等领域依然有着研究的前景与可能性。

附录

组员分工及贡献比

吴越：

对图数据库、图查询语言以及图数据库的应用等开展了基础的认知与探究；基于图的综述论文来深入理解图、图数据库以及图数据库上的操作在四种维度上的前景与困境；

刘开开：

针对图数据库的查询编译优化，研究了一种集成图查询解释和编译的自适应方法，深挖其背后的优化细节与手段；针对该方法提出一系列进一步优化的想法与思考。

共同编写报告与 PPT