软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(8):3587−3604 [doi: 10.13328/j.cnki.jos.006552]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于核外计算的 Datalog 引擎设计与实现*

张奕裕[1,2], 王归航[1,2], 左志强[1,2], 李宣东[1,2]

[1](南京大学 计算机科学与技术系, 江苏 南京 210023)

[2](计算机软件新技术国家重点实验室 (南京大学), 江苏 南京 210023)

通信作者: 左志强, E-mail: zqzuo@nju.edu.cn

摘　要: 随着新兴技术的迅速发展, 领域软件对开发效率提出了新的要求. Datalog 语言作为一门具有简洁语法和良好语义的声明式编程语言, 能帮助开发人员快速开发和解决问题, 近年来越来越受到重视与欢迎. 但解决真实场景问题时, 现有的单机 Datalog 引擎计算规模往往受限于内存容量大小, 不具有可扩展性. 为解决上述问题, 设计并实现基于核外计算的 Datalog 引擎. 方法首先设计一系列计算 Datalog 程序所需的支持核外计算的操作算子, 然后将 Datalog 程序转换合成带核外计算算子的 C++程序, 接着方法设计基于 Hash 的分区策略和基于搜索树剪枝的最少置换调度策略, 将相应的分区文件调度执行计算并得到最终结果. 基于该方法, 实现原型工具 DDL (disk-based Datalog engine), 并选取广泛应用的真实 Datalog 程序, 在合成数据集以及真实数据集上进行实验, 实验结果体现了 DDL 良好性能以及高可扩展性.

关键词: Datalog 引擎; 核外计算; 操作算子; 分区策略; 调度策略

中图法分类号: TP311

中文引用格式: 张奕裕, 王归航, 左志强, 李宣东. 基于核外计算的Datalog引擎设计与实现. 软件学报, 2023, 34(8): 3587–3604. http://www.jos.org.cn/1000-9825/6552.htm

英文引用格式: Zhang YY, Wang GH, Zuo ZQ, Li XD. Design and Implementation of Datalog Engine Based on Out-of-core Computing. Ruan Jian Xue Bao/Journal of Software, 2023, 34(8): 3587–3604 (in Chinese). http://www.jos.org.cn/1000-9825/6552.htm

Design and Implementation of Datalog Engine Based on Out-of-core Computing

ZHANG Yi-Yu[1,2], WANG Gui-Hang[1,2], ZUO Zhi-Qiang[1,2], LI Xuan-Dong[1,2]

[1](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

[2](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: As emerging technologies develop rapidly, domain software puts forward new requirements for development efficiency. In addition, as a declarative programming language with concise syntax and well-defined semantics, Datalog can help developers solve complex problems rapidly and achieve smooth development and thus has attracted wide attention in recent years. However, when solving real-world problems, the existing single-machine Datalog engines are often limited by the size of memory capacity and possess no scalability. To solve these problems, this study designs and implements a Datalog engine based on out-of-core computing. Firstly, a series of operators supporting out-of-core computing are designed to compute the Datalog program, and then the program is converted into a C++ program with the operators. Next, the study designs a partition strategy based on Hash and a minimum replacement scheduling strategy based on search tree pruning. After that, the corresponding partition files are scheduled and computed to generate the final results. Based on this method, the study establishes the prototype tool DDL (disk-based Datalog engine) and selects widely used real-world Datalog programs to conduct experiments on both synthetic and real-world datasets. The experimental results show that DDL has positive performance and high scalability.

Key words: Datalog engine; out-of-core computation; operators; partition strategy; scheduling strategy

---

* 基金项目: 国家自然科学基金 (61802168); 江苏省自然科学基金 (BK20191247)
本文由“领域软件工程”专题特约编辑汤恩义副教授、江贺教授、陈俊洁副教授、李必信教授以及唐滨副教授推荐.
收稿时间: 2021-08-09; 修改时间: 2021-10-09; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28
CNKI 网络首发时间: 2023-01-19

近年来, 软件的快速发展对国民经济和社会发展起着积极的促进作用. 随着区块链, 云计算, 人工智能等新兴技术迅速发展, 这些特定领域对软件应用的需求显得尤为迫切, 而这也对领域软件开发的效率和执行性能提出了新的要求. 随着技术间的进一步融合, 领域软件面临着开发周期长, 设计复杂和开发效率低等挑战. 领域软件开发人员越来越需要编程语言提供高级抽象来帮助逻辑推理和应用的快速开发, 以此来满足日益增长的软件开发需求.

Datalog 作为一门声明式编程语言, 近年来在智能合约分析[1,2], 程序分析[3–5], 网络验证[6,7], 大数据分析[8–10], 反汇编[11]等领域得到广泛的应用, 越来越受到重视与欢迎. 归因于 Datalog 语言其声明式的特性, Datalog 语言天生具备简洁的语法和良好的语义等优势, 从而使用 Datalog 语言编写程序能够带来项目开发效率的提升, 同时软件开发人员只需要关注任务是什么以及任务处理的逻辑, 而无需考虑底层具体实现的细节. 与命令式编程语言 (例如 C/C++, Java) 命令式地一步步计算步骤不同, Datalog 程序以一种声明式的方式预先指定期望的计算结果, 所需的计算由 Datalog 引擎自动地完成. 而 Datalog 引擎设计实现的优劣直接决定解决问题规模的大小以及执行性能的快慢.

现有 Datalog 引擎被广泛应用于解决真实场景问题, 常常需要面对解决计算性能差和可扩展性低的问题. 在解决实际真实的问题时, Datalog 引擎往往需要处理上百条的 Datalog 程序, 以及包含上百万甚至千万条元组数据的输入文件. 特别地, 对于一些递归的程序 (如 Transitive Closure), 即使只有少量的输入数据, 但产生的中间结果或者最终结果却是扩大了上百倍[8]. 这些真实场景下需要面临的问题对 Datalog 引擎的可扩展性提出了很高的要求, 需要 Datalog 引擎能够处理大量数据, 同时执行的性能在可以接受范围之内.

现有的前沿工作在不同领域都设计提出了各自的 Datalog 引擎来解决实际的领域问题. 他们有基于单机实现的 BDDBDDB[12], μZ[13], LogicBlox[14]和 Souffle[3]等, 还有基于分布式系统实现的 Distributed socialite[15]和 BigDatalog[8]等. 基于分布式系统实现的 Datalog 引擎具有显著的可扩展性, Datalog 引擎处理的问题规模可以随着集群的扩大而增大, 且通过对算法的分布式并行化实现使得 Datalog 引擎执行性能表现良好. 但基于分布式系统实现的 Datalog 引擎问题在于, 一是在集群上的代码调试将更加困难, 二是扩大集群以处理更大规模问题的成本将显著增加, 三是通常开发人员的开发环境并不普遍具备分布式集群的条件. 基于单机实现的 Datalog 引擎对使用人员更加友好, 开发人员能够简单方便易操作地通过在单机开发环境下对 Datalog 程序进行调试或执行计算, 而无需分布式环境下繁琐的调试部署, 且现有工作通过对 Datalog 引擎中数据结构的优化以及执行策略的改进, 使得其执行性能表现在可接受范围之内. 但基于单机实现的 Datalog 引擎问题在于, 一是处理问题的规模受限于单机内存容量的大小, Datalog 引擎常因为执行过程中的内存溢出而直接终止计算, 例如 BDDBDDB, μZ 和 SQLite 引擎在面对真实的 OpenJDK7 数据集计算上下文敏感的指向分析时因内存溢出而无法完成计算[3]; 二是为了处理更大规模问题而扩大内存容量的成本也相对较为昂贵.

因而基于上述的讨论, 为了能够享受单机引擎的简便以省去分布式引擎的繁琐同时使得单机引擎具有高可扩展性, 本文设计提出了基于核外计算的 Datalog 引擎 DDL (disk-based Datalog engine). 在 Datalog 程序运行时, 并不是全部数据存放在内存中计算, 而只有部分数据被调入内存并执行计算, 同时在适当时候写回外存, 通过内外存的数据交换来完成整个计算[16]. 由于 Datalog 引擎的实现可以视作一系列关系代数操作算子的具体实现 (第 1.3 节), DDL 通过设计实现了一系列完成 Datalog 程序计算所需的支持核外计算的操作算子. 同时 DDL 在 Souffle 基础上将 Datalog 程序转换合成为由这些核外计算算子表示的 C++程序, 实现了完整的支持核外计算的 Datalog 引擎. DDL 还通过设计实现基于谓词属性的 Hash 分区策略以及基于搜索树剪枝的最少置换分区调度策略等优化手段降低 I/O 开销以提升引擎执行性能. DDL 引擎工作在单机环境下, 具备基于单机实现的 Datalog 引擎的优点, 同时使得问题处理的规模不再受限于内存的大小, 相反可以通过扩增相对便宜的硬盘容量来处理更大规模的问题.

总而言之, 本文做出了如下创新贡献.

- 设计了基于核外计算的 Datalog 引擎并实现了相应的原型工具 DDL, 解决了单机环境下 Datalog 引擎面对真实场景时受限于内存大小的局限性.
- 设计实现了完成 Datalog 程序计算所需的核外计算的操作算子, 并提出了基于谓词属性的 Hash 分区策略和基于搜索树剪枝的最少置换调度策略等优化手段, 来实现 DDL 引擎高效的核外计算.

● 在合成的和真实的数据集上与现有先进单机 Datalog 引擎进行了对比实验, 表明了 DDL 良好的性能以及高可扩展性. 特别地, 对于小规模数据能够在内存中完成计算的, DDL 引擎并没有引入过多的额外开销, 甚至在某些例子上性能更优; 而对于大规模数据发生内存溢出情况的, DDL 引擎能够完成计算, 具有高可扩展性, 且性能在可接受范围内.

本文第 1 节介绍本文工作的相关背景知识. 第 2 节介绍所提出的基于核外计算的算子. 第 3 节介绍为提升 DDL 引擎性能提出的优化策略. 第 4 节介绍 DDL 引擎的设计实现. 第 5 节通过实验展示 DDL 引擎的可扩展性和性能. 第 6 节回顾相关工作. 第 7 节对本文工作进行总结, 并提出未来工作展望.

1 背景知识

1.1 Datalog 程序

一个 Datalog 程序是由有限条数量的 Datalog 规则以及数据事实组成. 每一条 Datalog 规则由 3 个部分构成, 分别是规则头, 连接符 (:–) 和规则体, 形如公式 (1) 所示.

$$H :- B_0,\ldots,B_i,\ldots,B_n. \tag{1}$$

其中, H 和 B_i 代表 Datalog 程序中的谓词, $B_0,\ldots,B_i,\ldots,B_n$ 的合取构成规则体, 其结果构成规则头 H. 一个谓词 $B(A_1,A_2,\ldots,A_k)$ 可以实现成一个有 k 列的二维表格, 其中 A_i 被叫作属性. 在表格中的每行元素 $t=<e_1,e_2,\ldots,e_k>$ 被视作一个元组. 本文把输入的元组叫作数据事实.

下列公式 (2)–公式 (5) 展示了一个 Datalog 示例程序. 该程序用来计算图中所有存在可达路径的两点.

$$edge(a,b). \tag{2}$$

$$edge(b,c). \tag{3}$$

$$path(X,Y) :- edge(X,Y). \tag{4}$$

$$path(X,Y) :- path(X,Z), edge(Z,Y). \tag{5}$$

该程序中的公式 (2) 和公式 (3) 表示的是给定数据事实, 即图中存在 a 点到 b 点的一条边和 b 点到 c 点的一条边. 程序中的公式 (4) 和公式 (5) 表示的是 Datalog 规则, 存在着两个谓词关系, 即 *edge* 谓词和 *path* 谓词. 其中公式 (4) 规则表示的是如果 X 点到 Y 点之间存在一条边, 那么就意味着 X 点到 Y 点之间就存在着一条路径, 即该条规则通过 *edge* 谓词派生出新的 *path* 谓词; 公式 (5) 规则表示如果 X 点到 Z 点之间存在一条路径, 并且 Z 点到 Y 点之间存在一条边, 那么就意味着 X 点到 Y 点之间存在着一条路径, 从而该条规则就派生出新的 *path* 谓词元组 *path*(X, Y).

1.2 Datalog 执行

Datalog 引擎根据给定的数据事实和 Datalog 规则, 计算得到目的谓词结果. 现有的 Datalog 执行方法按照搜索策略可以分为两类, 一类是自底向上执行, 另一类是自顶向下执行[17].

1.2.1 自底向上执行

给定输入的数据事实和 Datalog 规则, Datalog 引擎递归应用 Datalog 规则以及数据事实以派生出更多的元组, 直到达到计算的不动点 (即没有更多新的元组派生出来) 停止. 自底向上执行的方法通过将 Datalog 程序中所有规则应用于给定的数据事实进行计算, 把满足规则的元组派生为规则头的谓词. 具体来说, 结合 Datalog 不动点语义进行求值, 即从只包含有给定数据事实谓词的 Datalog 规则开始应用, 进行迭代求值; 在每次迭代中, 所有 Datalog 规则都被求值, 并计算派生得到满足规则的头部谓词; 当没有更多新的头部谓词派生时, 计算达到不动点, 执行停止. 该种方法被称作为朴素法. 但显然的是, 朴素法在执行过程当中许多谓词元组进行了多次派生, 例如公式 (2)–公式 (5) 所示的 Datalog 程序, 其中 *path*(a, b), *path*(b, c) 谓词元组在执行过程中就派生了两次, 造成了冗余的结果, 导致执行性能的低下. 产生这种冗余结果的原因在于, 朴素法在计算过程中每次迭代时都应用的是当前已知的所有事实 (包括输入数据事实和派生元组事实), 不管这些谓词是否会再额外产生新的谓词.

为了在计算的时候尽量避免重复之前迭代中已经完成的结果, 一种更优的自底向上执行方法被提出, 叫作半朴素法[18,19]. 半朴素法基于这样的观察, 只有在前一次迭代计算中产生的新元组才能在本次迭代计算中产生更多

的新元组. 因而半朴素法与朴素法不同的地方在于半朴素法在每次迭代计算都只应用在前一次迭代计算中新产生的元组, 从而减少冗余的计算.

1.2.2 自顶向下执行

虽然自底向上执行的半朴素法在程序的迭代计算中最小化了冗余计算的产生, 但是它并没有最小化在产生目标谓词元组结果时不相关的谓词元组派生. 例如在公式 (2)–公式 (5) 的 Datalog 程序中, 需要产生以 b 为起始节点所能到达的所有节点集合, 而显然 $path(a, b)$ 这个谓词不会也不需要参与到计算中, 但自底向上执行方法依然会在迭代计算过程产生该谓词. 而自顶向下执行的方法[17]则不会造成这种计算的冗余, 它会将条件从想要的规则向下推入可能回答该查询的规则中, 从而这些规则会创建更多子规则, 子规则依次类似的方法向下推入, 直到下推到输入的数据事实中判断是否存在满足条件的某个元组. 其中最具代表性的自顶向下执行方法是 QSQ 方法[20].

在过去, 不少研究工作设计实现基于自顶向下计算方法的 Datalog 引擎[21,22]. 虽然自顶向下的方法更加直接产生目标谓词元组结果, 且计算过程中不会产生跟目标结果无关的冗余中间结果, 但在实际实现中较繁琐, 无法直接投入真实问题场景中开展计算, 且可优化空间小, 执行性能并不比自底向上执行更优. 相反, 自底向上的方法实现简单, 能适用真实场景问题, 存在优化空间. 现在业界主流 Datalog 引擎[3,8,14]执行方法选取大都选择了自底向上执行, 且提供丰富的优化手段, 提高执行性能. 类似地, 本文方法也基于自底向上执行的半朴素方法设计基于核外计算的 Datalog 引擎.

1.3 Datalog 引擎实现

Datalog 语言可以看作是对关系代数的一种递归式扩展[23]. Datalog 语言中多种语法表示能够与关系代数进行相互转换[24], 例如表 1 所示. 在表 1 中, 展示了 7 种操作算子对应的 Datalog 规则与能进行相应转换的关系代数表示. 例如对 Diff 操作算子而言, $S(X,Y):-R(X,Y),!T(X,Y)$. Datalog 规则就可以由 $R(X,Y)-T(X,Y)$ 关系代数进行表示计算.

表 1 操作算子对应的 Datalog 规则表示与关系代数表示

操作算子	Datalog规则表示	关系代数表示
JOIN	$S(X,Y,Z):-R(X,Y),T(Y,Z).$	$R(X,Y)\bowtie T(X,Y)$
Diff	$S(X,Y):-R(X,Y),!T(X,Y).$	$R(X,Y)-T(X,Y)$
Union	$S(X,Y):-R(X,Y).$ $S(X,Y):-T(X,Y).$	$R(X,Y)\cup T(X,Y)$
Intersection	$S(X,Y):-R(X,Y),T(X,Y).$	$R(X,Y)\cap T(X,Y)$
Projection	$S(X):-R(X,Y).$	$\prod_X(R)$
Selection	$S(X,Y):-R(X,Y),X>c.$	$\sigma_{X>c}(R)$
Product	$S(X,Y,Z,W):-R(X,Y),T(Z,W).$	$R\times T$

类似地, 对于完整的 Datalog 程序而言, 它就可以被转换为由一系列操作算子表示的关系代数. 因而一般来说, 对于 Datalog 引擎的实现, 只需要支持对一系列操作算子的关系代数运算就可以完成 Datalog 程序的计算. 即 Datalog 引擎先通过对 Datalog 程序的解析, 将其转化为由一系列操作算子表示的关系代数, 接着通过实现一系列算子操作, 完成对相关谓词的代数运算, 最后实现对 Datalog 程序的计算. 在 DDL 引擎中, 也采用了上述 Datalog 引擎实现的思想. 不同的是, DDL 进一步通过实现了一系列支持核外计算的操作算子来实现 Datalog 程序的核外计算. 同时核外计算的操作算子并没有改变原先算子的计算语义, 只进行了支持核外计算的实现, 所以 Datalog 程序与转换后的一系列核外计算算子表示两者逻辑上是等价的, 因而对于 DDL 计算结果的正确性是可以得到保证的.

2 基于核外计算的算子设计

观察发现, 在现有的 Datalog 引擎计算过程中, 发生内存溢出的地方在于两两谓词之间做 JOIN 计算的时候产生大量的中间结果数据, 这些数据占用了大量的内存资源而导致内存溢出. 因而方法重点通过实现支持核外计算

的 JOIN 操作来支持引擎的整体核外计算. 同时对于 Datalog 程序中的规则而言, 一般可以分成两类[13], 一类是递归规则, 即规则递归地依赖于它自身或者依赖于它的另一条规则; 另一类是非递归规则, 即规则之间没有递归依赖关系. 递归规则的执行通常采用半朴素法进行高效计算, 其中关键的操作除了谓词之间的 JOIN 操作还有 SetDiff 操作. SetDiff 操作的目的是将本轮次计算出的结果与上轮次计算出的结果求差, 用来获得每一轮次计算中新派生出的谓词元组. 而非递归规则的执行, 通常视作是谓词之间的 JOIN 操作. 为了支持 Datalog 程序规则递归和非递归的特性, 方法设计并实现了 JOIN 和 SetDiff 操作的核外计算的算子. 同时当 Datalog 被应用于数据分析或图计算分析等领域时, 其程序常会涉及聚合操作 (例如 MIN, MAX, SUM 等). 为了 Datalog 引擎适用范围的通用性, 方法也为聚合操作设计了相应的支持核外计算的操作算子. 下面将主要描述 JOIN, SetDiff 以及聚合操作算子的核外计算设计. 其余操作算子或者不会进一步消耗内存空间 (如 Intersection, Selection 等), 或者在前述算子基础上进行相同的核外计算操作 (如 Union, Product), 因而不再展开阐述.

2.1 JOIN

核外计算的 JOIN 算子, 其基础的设计想法是将过大的输入数据事实文件, 或者过大的临时中间文件分区, 分成多块小分区文件, 然后将其读入内存中进行 JOIN 计算, 并将 JOIN 结果写回到硬盘上存储. 以第 5.3 节中的 Program1 为例展示核外计算的 JOIN 操作流程图, 如图 1 所示 (假设 R, S 和 T 谓词的输入数据事实文件都被分成了两个分区文件).

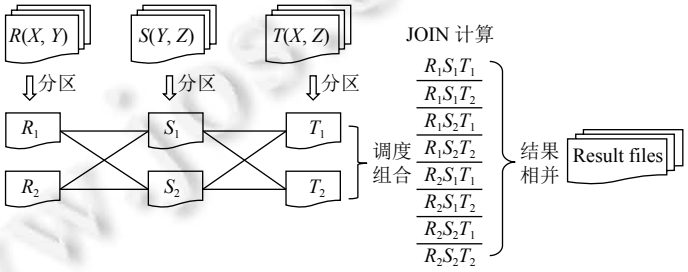


图 1 以 Program1 为例, 核外计算的 JOIN 算子操作流程图

Program1 在实现上可以看作是 R, S, T 表项之间 JOIN 操作, 找到存在三角关系的集合. 核外计算的 JOIN 算子首先将 R, S 和 T 谓词的输入数据事实文件按照分区策略各分成两个分区文件, 接着通过调度策略选择相应的分区文件从硬盘上读取入内存中进行 JOIN 计算. 调度策略显然需要依次遍历所有可能的 8 种组合才能不遗漏计算. 最终将每种组合的 JOIN 结果相并得到完整结果, 并将其写回硬盘.

2.2 SetDiff

核外计算的 SetDiff 算子, 其基础的设计想法与 JOIN 算子是类似的, 通过将大文件分区成小文件, 然后逐一遍历组合进行 SetDiff 处理. 但与 JOIN 算子不同的是, SetDiff 算子只能同时处理两个谓词关系的求差计算, 且对于同一被减分区谓词文件, 需要求其结果的交集, 同时最终结果需要将多个被减分区谓词文件的结果进行相并. 以 $S(X,Y):-R(X,Y),!T(X,Y).$ 为例展示核外计算的 SetDiff 操作流程图, 如图 2 所示 (假设 R 和 T 谓词的输入数据事实文件都被分成了两个分区文件).

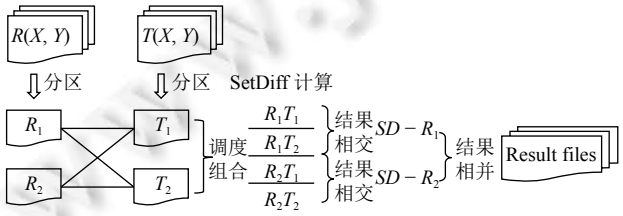


图 2 以 $S(X,Y):-R(X,Y),!T(X,Y).$ 为例, 核外计算的 SetDiff 算子操作流程图

核外计算的 SetDiff 算子首先将 R 和 T 谓词的输入数据事实文件按照分区策略各分成两个分区文件, 接着通过调度策略选择相应的分区文件从硬盘上读入内存中进行 SetDiff 计算, 显然需要依次遍历所有可能的 4 种组合才能完成 R 和 T 谓词整体的 SetDiff 计算. 因为 R 谓词文件可以看成是 R_1 和 R_2 分区文件的相并 ($R = R_1 \cup R_2$), 而 T 谓词文件可以看成是 T_1 和 T_2 分区文件的相并 ($T = T_1 \cup T_2$), 从而就可以推导出如下等式:

$$(R_1 \cup R_2) - (T_1 \cup T_2) = [(R_1 - T_1) \cap (R_1 - T_2)] \cup [(R_2 - T_1) \cap (R_2 - T_2)].$$

因而 SetDiff 算子在计算出每种组合求差集结果后, 需要将作为被减分区谓词文件 R_1 和 R_2 各自的差集相交得到结果 $SD - R_1$ 和 $SD - R_2$, 接着将 $SD - R_1$ 和 $SD - R_2$ 相并得到最终计算结果, 并写回硬盘.

2.3 Aggregation

核外计算的 Aggregation 算子, 其基础的设计想法启发于 RStream[10]与 XStream[25]的研究工作, 通过将分区后的文件逐一采用流数据处理的方式去记录所需的聚合操作结果. 方法在内存中维护一个固定大小的流数据缓冲区, 从硬盘上读取相应的谓词分区文件到流数据缓冲区内, 并根据具体的 Datalog 规则记录和维护该缓冲区内相应谓词分区文件的 Aggregation 算子数据. 通过流数据处理方式连续读取硬盘上的所有分区文件进入缓冲区内, 从而得到完整谓词文件的 Aggregation 算子数据. 目前 DDL 引擎主要考虑的 Aggregation 算子包括 MIN, MAX 和 SUM. 这 3 种算子在设计上是一致的, 在具体实现时根据各自算子语义的不同进行相应的计算.

3 优化策略

为了能够进一步提升 DDL 引擎执行核外计算的性能, 本文还提出了 3 点优化策略.

3.1 分区策略

将大文件分区成多个小文件, 是核外计算中的关键一步. 因为无论是计算开始执行前数据事实文件过大导致无法读入内存中计算, 还是在计算执行过程中临时中间结果文件过大导致内存溢出, 都是文件过大而导致的内存溢出异常. 而如果将输入数据事实文件或者中间结果文件分成多个独立的小文件, 能够读入内存中, 那么后续的计算则可以继续进行下去. 方法支持用户自定义分区的个数大小. 在执行过程中, 方法将按照自定义的分区数量对输入文件进行均分. 而分区的个数大小影响着 Datalog 引擎执行的性能. 若分区的文件粒度太粗, 导致分成的文件数目虽然少, 但是每个文件依然较大, 即使在初始轮次能够参与计算, 但很快就在后续轮次中导致内存溢出, 使得引擎需要花费更多额外的开销对文件进行再分区计算; 若分区的粒度太细, 导致虽然每个文件较小, 能够轻易读入内存中并完成后续计算, 但是分区的文件数目多, 引擎需要频繁读写这些分区文件, 导致 IO 开销增大. 且每个分区文件太小不能充分利用内存资源. 然而在未执行计算前, 有效准确衡量分区粒度是否合适较为困难, 因为无法准确知道输入数据事实文件中数据的分布情况也不知道计算过程中临时中间结果的产生情况, 而这些只有具体执行过后, 才能得知其中所花费 IO 的开销. 因而不方便直接从分区粒度角度出发设计合理的分区优化策略.

分区优化策略的目的是尽可能降低因分区文件而带来的 IO 开销, 同时较充分利用内存资源. 方法从另一个角度进行考虑, 提出基于谓词属性的 Hash 分区优化策略来达到上述目的. 方法基于 JOIN 操作的特性, 每次选取在规则体中被谓词包含数目最多的属性值作为被 Hash 分区的键值 (若有多个候选属性值则随机选取其中一个), 然后将其中包含该属性值的谓词数据文件进行 Hash 分区的优化操作. 对于每一轮参与 JOIN 计算的表项, 采用除留余数法, 将谓词中的元组 Hash 到对应的分区中, 使得后续计算只用考虑相同 Hash 分区中的情况, 而无需计算所有的分区文件, 降低了 IO 开销. 同时根据输入数据事实文件之间大小的最简比值, 作为各自期望分区的个数大小. 方法同时兼顾内存资源利用大小的影响. 但显然对于真实情况中数据分布情况, 可能会出现一次分区完成后, 导致该轮次读入内存计算的分区文件依然太大的情况. 方法会对分区文件进行再分区操作, 再分区操作与上述分区操作的区别在于, 再分区操作中将上一轮分区个数的下一个相邻的递增质数作为本轮次的分区个数, 如果依然无法读入内存, 将继续重复上述操作, 直到能够读入内存中进行计算为止. 而对于其他未涉及属性值的谓词元组, 仍然根据用户自定义的分区数量进行均分的操作.

3.2 调度策略

在 JOIN 以及 SetDiff 操作前都需要一个合理的调度策略, 选择将合适的分区文件读入内存进行计算. 如果随意选择某一分区组合进入内存中进行计算, 会使得下一次分区调度需要调换的分区数量随机, 导致 IO 的开销额外增加, 而调度策略的目的就是在分区组合数固定的情况下尽可能减少因分区调换而导致 IO 次数的增加.

为了实现上述目的, 方法提出了基于搜索树剪枝的最少置换策略. 所有的分区文件按照 Datalog 规则中声明的谓词顺序排序, 可以形成分区组合的搜索树. 树的根节点是规则头部谓词, 树的子节点是规则体谓词. 每个谓词子节点的数目是子节点分区文件的数目. 并定义规则顺序在前谓词是在后谓词的父节点, 相反地, 在后谓词是在前谓词的子节点. 如图 3, 展示了第 5.3 节 Program1 中 R, S 和 T 谓词一种可能的分区组合搜索树 (假设其中每个谓词都产生了两个分区文件, 并对 R 和 S 谓词进行了 Hash 分区操作).

由于每次计算都需要加载 3 个谓词文件的分区文件, 例如 $R_0S_0T_0$, 而基于深度优先的遍历会导致下一次调度的时候需要调度两个分区才能完成计算, 例如完成 $R_0S_0T_1$ 后需要计算 $R_0S_1T_0$, 这就导致需要置换 S 谓词和 T 谓词两个分区的文件, 而导致了 IO 的开销增大. 而完成所有的分区计算则需要在计算中一共调度的分区次数 (不包括起始 3 个分区文件的调度) C_1=1+2+1+3+1+2+1=11. 而使用最少置换策略, 即在每次计算时只调换其中一个分区文件即可, 完成所有分区计算所需的调度分区次数为 C_2= 1+1+1+1+1+1+1=7. 而基于搜索树剪枝的最少置换策略只用考虑与 Hash 地址相匹配的分区文件, 因而可以对搜索树进行剪枝. 在剪完枝的搜索树上, 方法选择一个最小的分区组合数作为起始点, 然后依照该起始点从叶子节点开始进行置换不同分区, 并从下往上选择节点对应的分区文件进行置换, 每次置换都保证只选择最少次数的分区进行, 如图 4 所示.

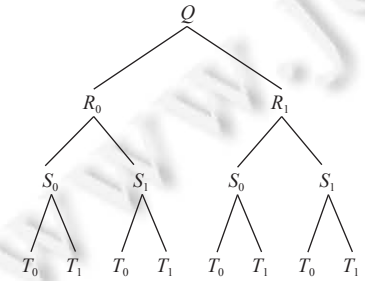


图 3 R, S 和 T 谓词一种可能的分区组合搜索树

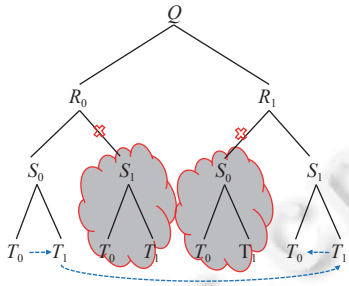


图 4 优化后的分区组合搜索树

因为对 R 谓词和 S 谓词进行 Hash 分区操作, 因而对于 R 和 S 谓词的分区文件计算只需要计算 R_0 与 S_0 分区和 R_1 和 S_1 分区, 而 R_0 和 S_1 以及 R_1 和 S_0 分支将不会产生 JOIN 结果, 因而可以对其剪枝. 同时在计算 $R_0S_0T_0$ 后, 基于最少置换的原则, 下一步选择 T_1 进行置换, 而后选择 R 和 S 谓词进行置换, 以此类推. 最终基于搜索树剪枝的最少置换策略完成所有分区计算所需的调度分区次数为 C_3 =1+2+1=4. 通过调度分区次数就可以得知 ($C_3 < C_2 < C_1$), 在 IO 开销上, 基于搜索树剪枝的最少置换策略交换分区数最小, 所需 IO 开销最少.

观察发现要实现上述最小置换策略, 只需在常规的 DFS 遍历逻辑上引入一个 isIncreased 数组 (bool 类型) 记录遍历的方向信息即可实现. 并在此基础上进一步考虑基于 Hash 的分区优化后, 引入 hashDepth 参数, 用来记录当前数据集的前 hashDepth 个数据集是基于 Hash 分区得到的, 即实现了基于搜索树剪枝的最少置换调度策略. 具体的基于搜索树剪枝的最少置换调度策略, 如算法 1 所示.

算法 1. 基于搜索树剪枝的最少置换调度算法.

```
1. Schedule(depth, maxDepth, hashDepth, partitionIndex, partitionSize, isIncreased){
2. if depth ≤ hashDepth then //只组合数据集相同的 index
3.     if !check(partitionIndex, depth) then //保证待计算的分区序列相同
```

```
4.              return;
5. if depth==maxDepth then //得到从根到叶子的路径, 执行计算
6.      // do the computation
7.      return;
8. if isIncreased[depth] then
9.      for i = 0 to partitionSize[depth]–1 do //增序遍历
10.             partitionIndex[depth] = i; //对第 depth 个数据集, 选择第 i 个分区
11.             Schedule(depth+1, maxDepth, hashDepth, partitionIndex, partitionSize, isIncreased);
12.             if i==partitionSize[depth]–1 then //到最大下标处, 下一次做降序遍历
13.                 isIncreased[depth] = false;
14. else
15.     for i = partitionSize[depth]–1 to 0 do //降序遍历
16.             partitionIndex[depth] = i;
17.             Schedule(depth+1, maxDepth, hashDepth, partitionIndex, partitionSize, isIncreased);
18.             if i==0 then //到最小下标处, 下一次做增序遍历
19.                 isIncreased[depth] = true;
20. }
```

算法从第 2 行开始的 if 语句块中首先通过遍历搜索树的前 hashDepth 层, 即那些经过 Hash 分区的分区文件, 通过在前 hashDepth 层的过程中保证只组合 Hash 地址值相同的分区文件, 而不考虑其他不相同 Hash 地址值的分区组合, 从而达到剪枝的目的. 算法第 3 行的 if 语句块对已经遍历到的各层进行校验, 需要保证 partitionIndex[0, depth) 范围内的序列都相等. 因为基于 Hash 分区的性质, 如果范围序列不完全相同, 说明不存在 JOIN 的结果值, check 函数返回 false, 然后直接 return. 当 depth 与 maxDepth 相等时 (算法第 5 行), 说明已经得到了一个从根到叶子节点的完整路径, 就可以处理 depth 个数据分区的相关计算, 计算结束后返回. 若 depth 与 maxDepth 不相等, 则需要继续遍历. 如果 isIncreased[depth] 为 true (算法第 8 行), 则对于第 depth 个数据集, 对其 partition 做增序遍历 (算法第 9 行), 对第 depth 层的谓词, 选取其第 i 个 partition (算法第 10 行), 然后继续调用 Schedule 函数进行递归遍历处理第 depth+1 层谓词 (算法第 11 行). 当已经遍历到最大下标时, 将该数据集所对应的 isIncreased 置为 false, 在下一次处理该数据集时, 会对 index 做降序遍历 (算法第 12 和 13 行). 如果 isIncreased[depth] 为 false (算法第 14 行), 则对于第 depth 个数据集对其 partition 做降序遍历 (算法第 15 行), 同样地, 对第 depth 层的谓词, 选取其第 i 个 partition (算法第 16 行), 然后继续调用 Schedule 函数进行递归遍历处理第 depth+1 层谓词 (算法第 17 行). 当已经遍历到最小下标时, 将该数据集所对应的 isIncreased 置为 true, 在下一次处理该数据集时, 会对 index 做增序遍历 (算法第 18 和 19 行).

3.3 二进制格式文件读写

核外计算需要涉及大量的文件读写操作, 而文件格式上的不同表示会影响文件读写操作的性能[26]. 文本格式表示便于编辑, 而二进制表示存储的是值的内部表示. 以二进制格式保存数据的速度更快, 因为不需要转换, 并且可以大块地存储数据. 且二进制格式通常占用的空间较小. 在实际的 Datalog 程序中谓词的属性通常由数字进行表示且数据事实文件通常较大, 因而为了能够进一步提升文件读写的性能, 方法采用二进制的格式来表示文件.

4 DDL 引擎实现

4.1 DDL 引擎框架

基于核外计算的 Datalog 引擎 DDL 主要分为前端解析合成模块和后端核外计算模块. DDL 引擎的框架流程图如图 5 所示.

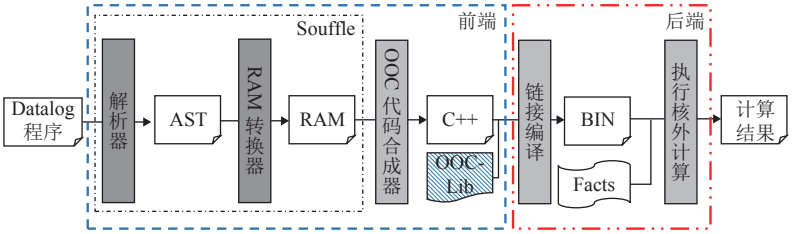


图 5 DDL 引擎框架流程图

在前端解析合成模块, 方法集成 Souffle 工具[27], 用于前端 Datalog 程序的解析和优化, 以及用于产生优化后的 RAM (relational algebra machine, 关系代数机) 结果. 然后方法基于 RAM 利用程序合成的方法实现合成自定义的带有支持核外计算的算子操作的 C++代码. 在后端核外计算模块, 方法将上述 C++代码结合支持核外计算的依赖库函数 OOC-Lib, 编译链接得到二进制可执行文件 BIN, 接着通过给定输入数据事实文件 Facts, 执行上述二进制文件 BIN, 运行后端核外计算并得到最终的计算结果.

4.2 前端合成模块

对于前端 Datalog 程序的解析与优化模块, 方法使用现成工具 Souffle 生成前端分析结果. Souffle 工具中对前端解析进行了大量的优化, 例如对 Datalog 程序规则间进行常量传播, 对最终输出结果没有贡献的谓词或者规则进行消除等, 使得后端执行的时候可以省去冗余的计算. 同时 Souffle 还考虑到计算硬件资源上 Cache 缓存和多核并行化等优势, 生成了高度优化后的 RAM 用来进一步合成最终的 C++程序.

在前端模块, 通过设计一系列的核外计算的操作算子, 只需使得最终待编译的 C++程序中是由核外计算的操作算子所表示的 Datalog 程序计算逻辑即可, 后续就可以交给后端部分通过编译执行实现对 Datalog 程序的核外计算. 而同时为了能够充分利用已有的前端分析优化结果, 且使得最终 C++程序符合 Datalog 程序计算逻辑, 方法在利用 Souffle 工具生成的 RAM 阶段抽取满足核外计算算子操作所需的操作数, 接着采用程序合成的方法从 RAM 阶段合成带有核外计算算子的 C++程序. 方法先通过人工观察 JOIN 操作, SetDiff 操作以及 Aggregation 等操作在 Souffle 的 RAM 阶段的表示形式, 接着手动抽取出其中每种操作 RAM 表示形式的特点 (例如, RAM 中两个 for 语句构成一个 JOIN 操作等), 然后在 Souffle 的 RAM 合成 C++程序阶段时, 方法根据上述提取的特点识别每种操作对应的语句模式, 利用 Souffle 提供的函数接口获取语句的操作数, 最后按照既定的模板生成核外计算算子的表示形式. 如图 6 所示是 Datalog 规则公式 (5) 经过 Souffle 转换后得到的部分 RAM 表示, 其中 delta_path 和 edge 已由 path 赋值得到. RAM 中用 QUERY 关键字表示 Datalog 程序逻辑执行的开始, 首先判断 delta_path 和 edge 谓词是否为空, 在存在 delta_path 和 edge 谓词情况下, 通过两个 for 循环依次分别循环遍历其中 delta_path 和 edge 谓词中的每个元组 t0 和 t1, 然后判断 t0 元组中的第 2 个属性值是否和 t1 元组中的第 1 个属性值相同, 若相等, 则将 t0 元组的第 1 个属性值和 t1 元组的第 2 个属性值组成一个新的元组, 并判断该元组是否已经存在于 path 谓词中, 若不存在则将其追加到 new_path 元组中.

```
QUERY
  IF ((NOT (@delta_path = Ø)) AND (NOT (edge = Ø)))
   FOR t0 IN @delta_path
    FOR t1 IN edge ON INDEX t1.0 = t0.1
     IF (NOT (t0.0, t1.1) ∈  path)
      PROJECT (t0.0, t1.1) INTO @new_path
```

图 6 Datalog 规则公式 (5) 经过 Souffle 转换得到的 RAM 表示

如图 7 所示是在图 6 RAM 表示上期望抽取得到的含有 JOIN 和 SetDiff 算子的表示. 首先, 将 edge 谓词和 delta_path 谓词基于 edge 谓词的第 2 个属性值和 delta_path 谓词的第 1 个属性值进行 JOIN 操作, 并将 JOIN 的结果 join_res 写回到 new_path 中, 然后将 new_path 谓词和 path 谓词进行 SetDiff 的操作, 并将 SetDiff 操作的结果 setdiff_res 写回到 delta_path 中.

```
edge JOIN @delta_pathon edge.0 and @delta_path.1
    write join_resto @new_pathon edge.1 and @delta_path.0
@new_path SetDiff path
    write setdiff_resto @delta_path
```

图 7　Datalog 规则公式 (5) 用核外计算算子的表示

4.3　后端核外计算模块

在编译链接得到二进制可执行文件后, 结合给定的输入数据事实文件, 即可执行后端核外计算. 考虑到方法对 Datalog 程序处理的通用性, 以递归的 Datalog 规则进行举例, 展示后端核外计算的执行流程. 如图 8 所示, 展示了递归的 Datalog 公式 (5) 后端核外计算执行的流程图.

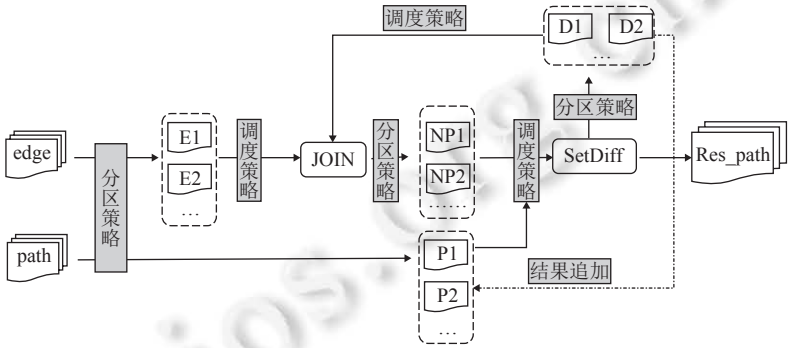


图 8　Datalog 程序公式 (5) 后端核外计算流程图

方法首先根据分区策略, 将参与计算的谓词文件分区, 例如将 edge 文件分成 E1, E2 等多个分区, 类似地, 对 path 文件 (根据 Datalog 公式 (4) 知, 初始 path 文件由 edge 文件派生得到) 分成 P1, P2 等多个分区. 因为分区后的每个分区文件大小已经能够保证被一轮执行所读入内存中进行计算, 然后通过调度策略, 从中选取合适的分区组合读入内存中, 与 Delta 文件 (初始 Delta 文件由 path 文件派生得到) 执行谓词之间的 JOIN 计算, 并将产生的中间结果同样按照分区策略写回到硬盘中, 即 NP1, NP2 (NP 表示 new_path 的缩写) 等, 接着将 new_path 分区文件和 path 分区文件根据调度策略, 选择合适的分区组合读入内存中进行 SetDiff 的计算, 如果本轮计算中有 Delta 文件产生, 则将产生的 Delta 文件根据分区策略写回到硬盘上, 即图中的 D1, D2 等多个分区, 同时将其追加到 path 的分区文件中. 再循环到 JOIN 操作计算中, 调度合适的 Delta 分区文件和 edge 分区文件进入内存中进行下一轮的计算, 以此往复; 如果本轮计算中没有 Delta 文件产生, 则认为执行达到不动点, 计算停止, 同时输出结果文件 Res_path. 值得一提的是, 由于基于 Datalog 程序计算逻辑, 在 SetDiff 操作计算后, Delta 文件需要将结果追加到 path 文件中, 而这会导致 path 文件中某个分区增大导致下一轮计算中无法调入内存中进行计算, 因而方法在 SetDiff 操作前会对该种情况进行判断, 若存在过大的 path 分区文件, 则会将该 path 分区文件根据再分区策略进行再分区以使得其能够被加载入内存进行计算.

5　实验评估

在前端模块, 方法复用了 Souffle 解析优化技术的同时, 修改了 Souffle 中 RAM 到 C++的合成代码模块, 使之能够生成符合后端计算的带有核外计算算子的程序. 在后端模块, 基于上述的方法实现了核外计算算子的操作以及分区和调度的策略, 使之能够高效完成核外计算.

5.1　实验设置

对 Datalog 引擎 DDL 的实验评估主要试图回答以下两个问题.

RQ1: DDL 引擎可扩展性怎么样?

RQ2: DDL 引擎计算性能怎么样?

所有实验都是在一台配备 Intel i7-8700 3.20 GHz CPU, 16 GB 内存, 1 TB SSD 硬盘的计算机上进行的. 并且将 DDL 与目前性能先进的单机 Datalog 引擎 Souffle[3], μZ (4.8.13 版本)[13]和 BDDBDDB (1.2 版本)[12]进行了对比实验.

5.2 基准输入选取

为了更全面评估 Datalog 引擎的性能和可扩展性, 方法基准输入的选择借鉴了 Shkapsky 等人[8]的研究工作, 类似地采用了合成数据集和真实数据集的组合. 合成数据集和真实数据集信息汇总在表 2 中, 表中 Vertices 列表示了顶点的数量, Edges 列表示了边的数量. 其中合成数据集是选取自 Shkapsky 等人[8]的研究工作, 真实数据集是选取自 Leskovec 等人[28]的研究工作. 特别地, 对于 3 个真实数据集, 分别将其按照 1–6 的基数均分了每个数据集, 并各取其中一份用来后续引擎可扩展性部分的实验评估.

表 2 基准输入数据集信息

数据集类型	数据集名称	Vertices	Edges
合成数据集	Tree11[8]	71 391	71 390
	Grid150[8]	22 801	45 300
	Gnp10K[8]	10 000	1 000 185
真实数据集	Wiki-Talk[28]	2 394 385	5 021 410
	Cit-Patents[28]	3 774 768	16 518 948
	LiveJournal[28]	3 997 962	34 681 189

● 合成数据集部分: Tree11 数据集是高度为 11 的树, 非叶子节点的度数是 2 到 6 之间的随机数. Grid150 数据集是 151×151 的网格状数据. Gnp10K 数据集是由采用 ER 模型[29]随机连接 10 000 个顶点生成的图, 其中每两个顶点连接存在边的概率为 0.001.

● 真实数据集部分: Wiki-Talk 数据集使用了维基百科用户 Talk 界面的编辑历史记录, 提取了从 2001 年 1 月维基百科创建到 2008 年 1 月的所有用户及其编辑讨论的数据, 创建了一个网络. 网络中的节点表示维基百科用户, 节点 i 到节点 j 的有向边表示用户 i 至少编辑过用户 j 的 Talk 界面. Cit-Patents 数据集是 1975–1999 年之间美国专利局授予所有实用专利的引用网络. 网络中的节点表示专利, 节点 i 到节点 j 的有向边表示专利 i 引用了专利 j 的内容. LiveJournal 数据集是从在线博客社区 LiveJournal 中的用户好友关系构建而来. 节点表示用户, 节点 i 和节点 j 存在边意味着用户 i 和用户 j 之间是好友关系.

5.3 基准 Datalog 规则选取

为了展示引擎计算 Datalog 程序的通用性, 实验选取了非递归的 Datalog 规则 1 个, 递归的 Datalog 规则 1 个和包含有 MIN 聚合操作的 Datalog 规则 1 个. 其中非递归的 Datalog 规则, 选取的是在学术界被广泛讨论应用且在领域软件例如大数据分析中被作为基础算法[10,29–31]的 Triangle Counting 例子, 如图 9 的 Program1 所示; 递归的 Datalog 规则, 选取的是具有代表性的例子 Transitive Closure[8], 该程序也是区块链智能合约分析和程序分析可达性算法的基础算法, 如图 10 的 Program2 所示; 含 MIN 聚合操作的 Datalog 规则, 选取的是在图分析计算领域中被广泛使用[8,23]的 Connected Components 例子, 如图 11 的 Program3 所示. Triangle Counting 是用来计算图上存在三角形关系的元组, Transitive Closure 是用来计算图上的传递闭包关系, Connected Components 是用来计算图中存在的连通分量.

Program1: Triangle Counting
r_1: Triangle (X, Y, Z) :−R (X, Y), S (Y, Z), T (X, Z).

图 9 Triangle Counting Datalog 规则

Program2: Transitive Closure
r_1: tc (X, Y) :−arc (X, Y).
r_2: tc (X, Y) :−tc (X, Z), arc (Z, Y).

图 10 Transitive Closure Datalog 规则

```
Program3: Connected Components
r1: cc2 (X, MIN (X)) : −arc (X, _).
r2: cc2 (Y, MIN(Z)) : −cc2 (X, Z), arc (X, Y).
r3: cc (X, MIN(Y)) : −cc2 (X, Z).
```

图 11　Connected Components Datalog 规则

5.4　可扩展性

为了评估 DDL 引擎可扩展的能力, 实验选择了 Program2 这个 Datalog 递归规则在 3 个真实数据集上进行了对比实验. 相比于非递归规则, 递归规则在计算过程中会产生更多的中间数据以及计算结果, 更能反映引擎可扩展的能力. 实验分别统计了引擎在 1/6, 1/5, 1/4, 1/3, 1/2 和 1 的输入数据集时执行的总时间, 其中 OOD (out of disk) 表示的是 DDL 引擎计算过程中耗光了所有硬盘资源仍未完成计算的情况 (例如某一次计算产生的中间结果过大占用了所有硬盘资源), OOM (out of memory) 表示引擎计算过程中内存发生了溢出的情况.

图 12 展示了在 Program2 程序下, DDL, Souffle, μZ 和 BDDBDDB 在不同规模的数据集下总执行计算时间.

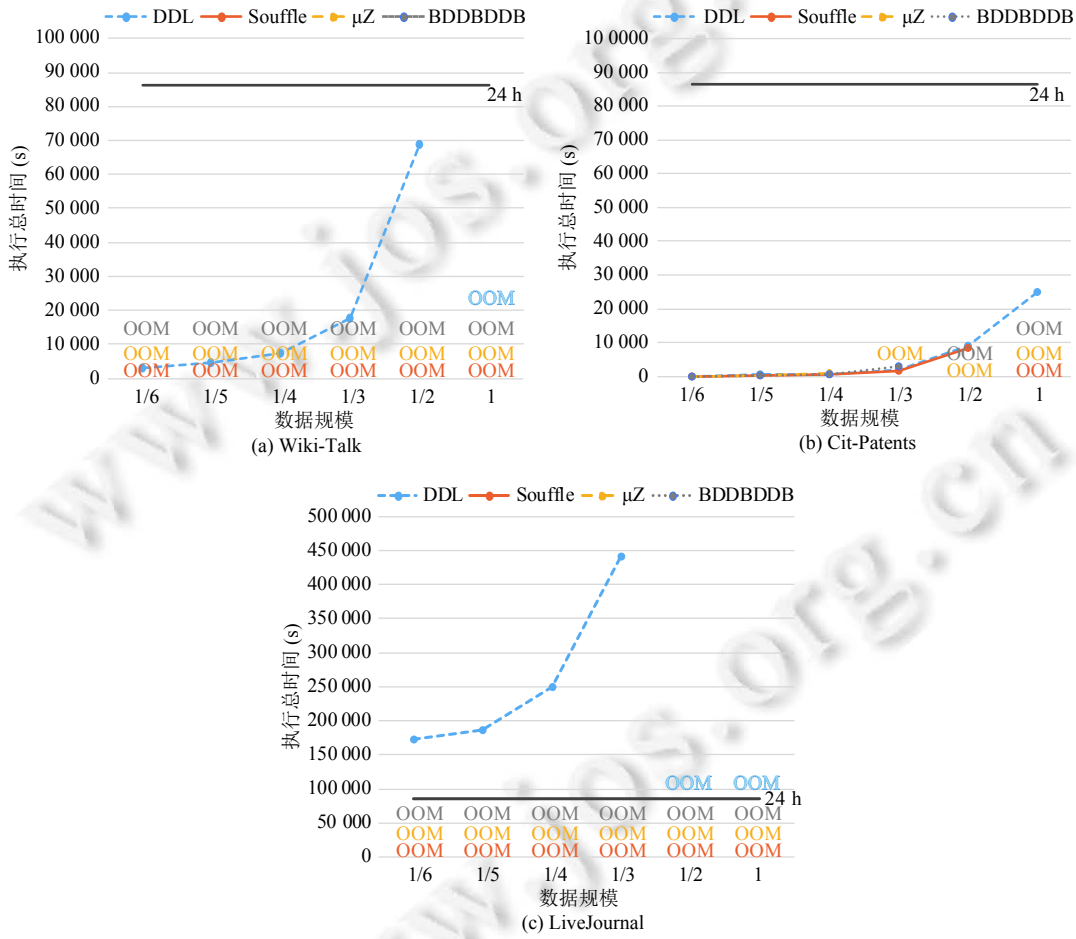


图 12　DDL, Souffle, μZ 和 BDDBDDB 在不同规模的数据集下执行 Program2 程序的总时间

由图 12 可见, 通过在不同规模数据集下的执行情况, 来反映出 DDL, Souffle, μZ 和 BDDBDDB 之间计算的可扩展能力差异. 在 Wiki-Talk 数据集下, Souffle, μZ 和 BDDBDDB 在 1/6 处的计算就已经 OOM, 而 DDL 在 1/2 处仍然能够完成计算, 但在计算完整数据集时发生了 OOD, 且执行时间也都在 24 小时的基准线以下. 在 Cit-Patents 数据集下, Souffle 在 1/2, BDDBDDB 在 1/3, μZ 在 1/4 处数据集能够完成计算, 而 DDL 能够完成完整数据集的计

算, 并且完成完整数据集的计算只需要不到 7 个小时的时间. 在更大的 LiveJournal 数据集下, Souffle, μZ 和 BDDBDDB 与在 Wiki-Talk 数据集下表现相同, 在 1/6 处的计算就已经 OOM, 而 DDL 在直到 1/3 处才发生了 OOD. 但比较遗憾的是, 即使在 1/6 处 DDL 所需计算时间已经远超过了 24 小时. 同时为了能够量化评估 DDL 的可扩展能力, 实验对 Wiki-Talk 和 LiveJournal 数据集进行了更细粒度的划分, 最终实验得到在 Wiki-Talk 数据集下 Souffle 在 1/7, μZ 在 1/15, BDDBDDB 在 1/12 处能够完成计算, 在 LiveJournal 数据集下 Souffle 在 1/36, μZ 在 1/60, BDDBDDB 在 1/48 处能够完成计算, 从而在本文的实验环境下 DDL 相较于 Souffle, μZ 和 BDDBDDB 能够处理其 2–20 倍规模的问题. 值得注意的是, 对于上述 DDL 发生 OOD 的情况, 方法可以通过简单地扩增硬盘的容量来帮助完成后续的计算. 相比较扩增内存容量来说, DDL 方法更加方便和实惠.

对于 RQ1, DDL 能够处理 1/2 的 Wiki-Talk 数据集, 完整的 Cit-Patents 数据集和 1/3 的 LiveJournal 数据集. 且对于更大规模的问题, DDL 可以通过简单扩增硬盘容量的方式加以解决. 实验结果表明, DDL 比现有先进的单机 Datalog 引擎具有更高的可扩展能力, 至少能够解决 2–20 倍于现有先进的单机 Datalog 引擎处理的问题规模, 且性能在可接受范围内.

5.5 性　能

为了评估 DDL 引擎的计算性能, 实验分别在 3 个 Datalog 规则上应用了上述 6 个输入数据集进行了对比. 对于 Program2 和 Program3 只需要一个输入数据事实文件即可, 而对于 Program1 需要 R, S, T 这 3 个输入数据事实文件. 为了能够在每个数据集上进行实验评估, 实验将每个数据集复制了 3 份, 然后将其分别命名为 R, S 和 T 用于计算. 同时为了能较全面评估对比性能, 对于真实数据集下的 Program2 和规则 Program3, 选取了 Wiki-Talk 数据集的 1/7, Cit-Patents 数据集的 1/3 (对 Program3 而言为 Cit-Patents 数据集的 1/2) 和 LiveJournal 数据集的 1/36 进行了实验, 而对于 Program1 则使用了完整的数据集进行实验. 需要注意的是, μZ 和 BDDBDDB 引擎并不支持 Aggregation 语义 (do not support, DNS), 因而无法完成 Program3 的计算. 实验记录了 Datalog 引擎执行计算的总时间, 和执行 Datalog 程序逻辑的时间 (ExeTime) 以及引擎执行计算前预处理的时间 (PreTime). 同时还记录了递归程序在不同数据集上计算的迭代轮数.

图 13 展示了 DDL, Souffle, μZ 和 BDDBDDB 在 Program1, Program2 和 Program3 Datalog 程序上的执行总时间, 执行 Datalog 程序逻辑时间和预处理时间, 从而表现出 DDL, Souffle, μZ 和 BDDBDDB 四者性能上的差异. 从中可以看出, 对于非递归程序 Program1 的计算性能, 在 3 个合成的小数据集上, DDL, Souffle, μZ 和 BDDBDDB 几乎没有差别, 但随着数据集的增大, 性能差异开始显现. 在 3 个真实数据集上, DDL 都要快于 Souffle 的执行, 而 μZ 和 BDDBDDB 都发生了 OOM 导致计算未完成. 从图 13(a) 可以看出在执行 Datalog 规则逻辑上所花费的时间上较接近, 但 Souffle 在预处理阶段花费了较多的时间去读取加载文件以及构建相应的数据结构, 相比之下 DDL 采用较简单的数据结构并基于 Hash 分区策略在预处理时花费时间较少, 同时得益于基于搜索树剪枝的最少置换调度策略, 使得 DDL 在执行 JOIN 和 SetDiff 运算时也能省掉较多不必要的计算.

而对于递归程序 Program2 的计算性能, 由于递归计算的特性使得其与非递归计算的结果大相径庭, 即使对于 3 个小的合成数据集上, DDL, Souffle, μZ 和 BDDBDDB 也出现了较大的性能差异. 特别是在 Grid150 数据集上 DDL 性能要远差于其他引擎, 而在 Gnp10K 数据集上 DDL 却要更快于 Souffle 和 BDDBDDB, 与 μZ 性能接近. DDL, Souffle, μZ 和 BDDBDDB 在 Tree11 上执行的时间都不到 3 s, 因而在讨论中忽略不计. 在 3 个真实数据集上, DDL 在 Wiki-Talk 数据集上表现要快于 Souffle 和 BDDBDDB, 而在 Cit-Patents 数据集上 DDL 则慢于 Souffle 但快于 BDDBDDB, 在 LiveJournal 数据集上 DDL 也要慢于 Souffle. μZ 引擎在 3 个真实数据集下均未完成计算. 同时从图 13(b) 可以看出 Datalog 程序执行逻辑的时间直接影响了最终执行性能.

表 3 展示了 Program2 在不同数据集上计算的迭代轮数. 结合表 3 的结果, 分析发现, 在 DDL 所需计算迭代轮数多的数据集上性能都较差 (例如 Grid150, Cit-Patents, LiveJournal), 而在所需计算迭代轮数少的数据集上性能较优 (例如 Gnp10K, Wiki-Talk). 原因在于 DDL 需要在每一轮迭代计算中需要进行固定次数的分区调度来完成计算 (若过程中需要再分区则会进一步增多分区的调度次数), 对于迭代轮数多的数据集会导致整体执行过程需要总分

区调度次数增加, 进而在 IO 上花费更多的时间. 而 IO 开销过多就会导致将优化策略所带来的性能收益抵消甚至拖累整体性能, 从而导致 DDL 计算性能下降.

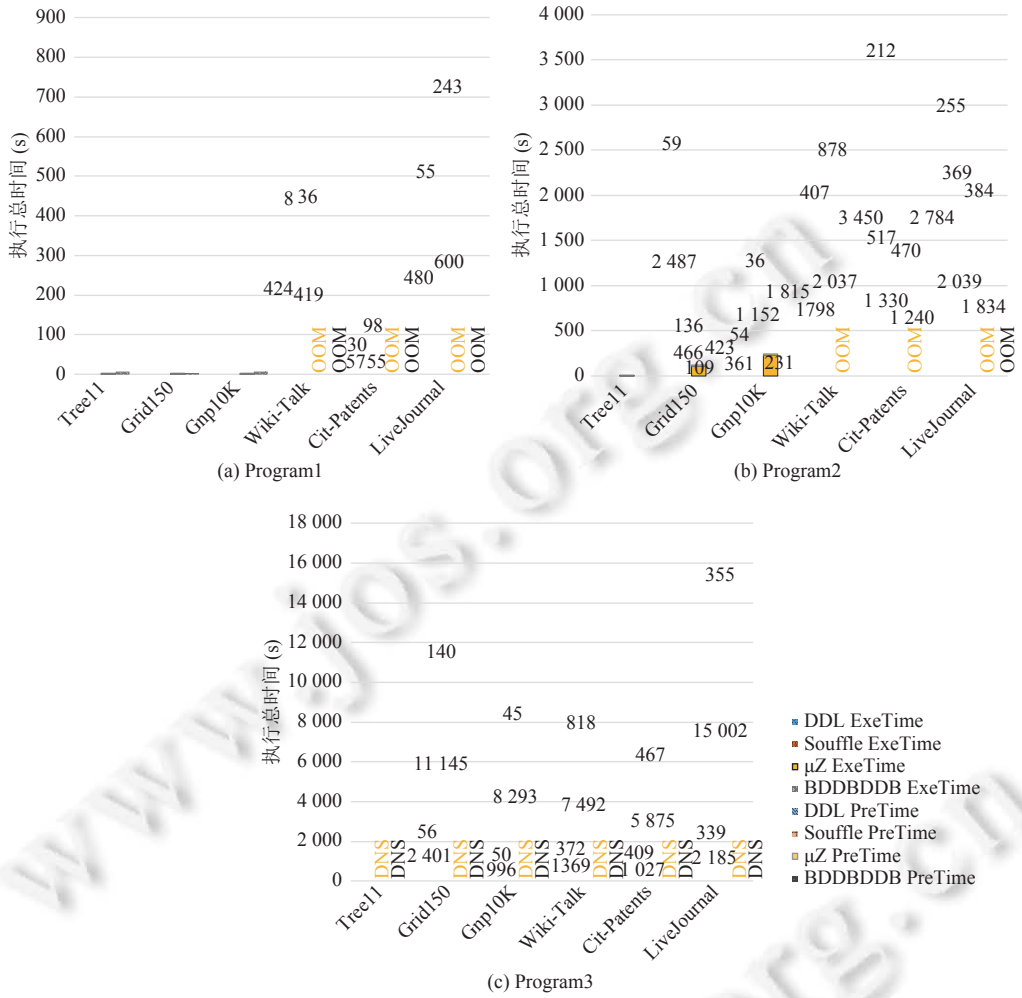


图 13 DDL, Souffle, μZ 和 BDDBDDB 在 Program1, Program2 和 Program3 上的性能表现

表 3 Program2 在不同数据集上计算的迭代轮数

数据集名称	迭代轮数
Tree11	12
Grid150	300
Gnp10K	7
Wiki-Talk	8
Cit-Patents	15
LiveJournal	23

对于含有聚合操作的 Datalog 程序 Program3 的计算性能, 可以从图 13(c) 明显看出无论是在小规模合成数据集上还是在大规模真实数据集上, DDL 的执行时间要远小于 Souffle (在 Tree11 数据集上执行时间相近且都小于 3 s, 因而在讨论中忽略不计), 反映出 DDL 对于含有聚合操作的 Datalog 程序计算的性能表现要更优于 Souffle. 得益于 DDL 采用流数据处理的方式计算聚合操作算子, 对于所有分区文件都只需遍历一遍即可求得聚合操作的结

果, 是 O(*N*) 的时间复杂度. 而对于 Souffle 则需要在内存中更新维护每一轮计算谓词的数据结构 (在 Souffle 中用 B 树表示谓词数据), 同时在其中进行聚合操作的计算得到结果, 是 O(*N*log*N*) 的时间复杂度.

对于 RQ2, DDL 在非递归 Datalog 程序小规模数据集上计算性能与 Souffle, μZ 和 BDDBDDB 相近, 在大规模数据集上更要优于其他引擎. 对于递归程序无论在什么规模数据集上, DDL 性能更受限于数据集的迭代计算轮数, 对于迭代轮数少的数据集, DDL 性能更优, 而对于迭代轮数多的数据集, DDL 受限于 IO 开销导致计算性能下降. 而对于含有聚合操作的 Datalog 程序, DDL 得益于流数据处理的核外计算方式, 使得无论在小规模合成数据集还是大规模真实数据集上性能表现都要优于 Souffle.

6 相关工作

6.1 前沿的 Datalog 引擎研究

随着 Datalog 在不同领域中得到广泛的应用, 研究人员在不同领域设计了特定的 Datalog 引擎帮助完成计算. 在单机 Datalog 引擎研究中, Whaley 等人提出了 BDDBDDB 引擎[12], 将 Datalog 应用于程序分析领域, 并提出使用 BDD 的数据结构来表示 Datalog 中的谓词关系. 通过为设计巧妙的 BDD 操作以及有效的优化手段, 引擎性能比手工调优的代码还要快上两倍, 并且能够对大型程序进行上下文敏感的指针分析. Hoder 等人提出了 μZ 引擎[13], 将 Datalog 程序转换为一阶逻辑谓词命题, 并利用 SMT 求解器 Z3 进行求解, 实现了引擎的高可扩展性和灵活性以及高效的性能. Aref 等人提出了 LogicBlox 系统[14], 第 1 个商业化的 Datalog 引擎. 他们旨在利用 Datalog 语言简洁的优势, 为现代应用程序减少软件开发的复杂性. 他们创新性地在 Datalog 语言上扩展了 Datalog 语法, 提出了 LogiQL 语言. 不同于传统的基于 pair 的表项 JOIN 进行计算, LogicBlox 使用 Leapfrog Triejoin 作为底层进行 JOIN 计算的核心算法, 并设计相应的数据结构, 进行了特定的优化. Scholz 等人提出了 Souffle[3], 用于大规模程序分析的 Datalog 引擎. 得益于 Datalog 语法的简洁性与引擎的优越性能, 使得普通开发人员无需专业的程序分析领域知识, 也能方便快速且高效地完成程序分析任务. Souffle 将 Datalog 程序经过多层次优化转换合成为 C++程序, 并最终得到对应的可执行文件. 除了通过前端分析优化以及选择合理的执行策略, Souffle 选择使用 Trie 和 B 树等数据结构表示谓词, 以及充分利用硬件并行化以及缓存等特性, 使得执行性能表现优异.

在分布式 Datalog 引擎研究中, Seo 等人提出了 Distributed Socialite[15], 基于 Datalog 的大规模图分析引擎. Distributed Socialite 基于分布式并行环境实现, 程序员只需简单标注数据是如何分布的, 引擎就会自动化产生适用于集群上并行执行的程序. Distributed Socialite 提出增量步进技术, 以优化递归单调聚合函数的并行执行, 并且它支持近似计算, 允许程序员用更少的时间和空间来权衡数值计算结果的准确性. Shkapsky 等人提出了 BigDatalog[8], 基于 Spark 分布式系统的 Datalog 引擎. Spark 分布式系统被广泛用于大规模的机器学习任务以及图分析. BigDatalog 利用 Datalog 的语法简洁以及表达能力强的优势, 同时结合 Spark 优势高效处理大规模数据任务. BigDatalog 设计 SetRDD 数据结构优化并行化的半朴素法. BigDatalog 提出了限制 Shuffle 操作的分区策略, 提高执行效率, 同时也利用缓存输入和 Spark 广播变量的结构来优化线性递归的 JOIN 操作.

单机环境下的 Datalog 引擎, 受限于内存, 并不具备很强的可扩展性, 在面对真实场景下的问题时往往因内存溢出而无法解决. 而分布式环境下的 Datalog 引擎, 集群环境维护困难, 代码调试也是个很大的挑战, 同时需要花费大量的时间设计合适的分布式并行算法, 并且执行也难以找到性能瓶颈. 而本文提出的基于核外计算的 Datalog 引擎 DDL, 享受单机环境的优点, 同时提供很高的可扩展性, 并且性能依然表现良好.

6.2 基于硬盘的计算系统

Kyrola 等人提出 GraphChi[32], 基于硬盘的单机图计算系统. 通过将大图分解成多个小图分区的方式, GraphChi 创新性地提出使用并行滑动窗口方法处理小图分区, 使得其能够在普通 PC 上对大规模的图实现数据挖掘, 图挖掘等功能. 同时进一步扩展 GraphChi, 能够支持随时间演化的图计算, 使得在普通 PC 上可以每秒同时执行超过 10 万个图计算任务. Wang 等人提出 Graspan[5], 基于硬盘并行计算的图计算系统. Graspan 可用于大规模程序的过程间程序分析. 将传统的程序分析算法, 从大数据处理角度进行看待, 基于硬盘实现, 将程序表示成边表的文件形

式存储在硬盘上, 采用以边对为中心的计算方法, 设计相应的并行化算法以及优化的调度策略和相应的数据结构, 使得在普通 PC 机上也能高效完成大规模程序的分析. Wang 等人提出 RStream[10], 基于硬盘的大规模图挖掘计算系统. RStream 基于硬盘顺序访问速率要比随机访问速率更快的思路, 对硬盘上完全无序的边列表形式表示的图文件采用流分区技术进行分区, 将硬盘上的大图文件分成多个流分区, 接着使用流处理的技术逐一处理每个流分区的计算, 以此减少分区 IO 开销. 同时 RStream 使用生产者-消费者的多线程模式, 通过创建线程本地缓冲区, 并行处理从硬盘读入的数据以及并行写回硬盘存储中间数据. 本文的方法受上述基于硬盘的图计算系统启发, 提出基于核外计算的 Datalog 引擎 DDL, 设计一系列核外计算的算子以及对应的分区策略和调度策略, 使得 Datalog 引擎在单机环境下高效执行计算同时不受内存容量制约.

7 总 结

本文设计并实现了基于核外计算的 Datalog 引擎 DDL. DDL 解决现有先进的单机 Datalog 引擎计算规模受限于内存大小的问题, 通过核外计算手段利用硬盘大幅提升单机 Datalog 引擎处理计算规模. DDL 将 Datalog 程序转换为一系列核外计算算子的表示来实现核外计算, 同时提出基于 Hash 的分区策略和基于搜索树剪枝的最少置换调度策略等优化手段提升引擎性能. 本文还实验评估了 DDL 良好的性能和高可扩展性.

本文方法和原型工具仍存在一些不足: 目前对于更复杂的 Datalog 程序, DDL 引擎前端解析模块并不能很好处理, 存在着无法从 RAM 中抽取得到带有核外计算算子的表示情况. 并且对于递归规则计算上较多的中间结果存储在硬盘上仍会导致 OOD 情况发生而终止计算. 在未来计划进一步完善 DDL 引擎前端解析模块以支持更复杂的 Datalog 程序, 并扩展 DDL 使得其能支持更丰富的 Datalog 语法 (如仿函数, 谓词组合等), 以及对暂存硬盘的中间结果作进一步的优化尽量避免 OOD 情况出现.

References:

[1] Brent L, Grech N, Lagouvardos S, Scholz B, Smaragdakis Y. Ethainter: A smart contract security analyzer for composite vulnerabilities. In: Proc. of the 41st ACM SIGPLAN Conf. on Programming Language Design and Implementation. London: ACM, 2020. 454–469. [doi: 10.1145/3385412.3385990]

[2] Grech N, Brent L, Scholz B, Smaragdakis Y. Gigahorse: Thorough, declarative decompilation of smart contracts. In: Proc. of the 41st Int'l Conf. on Software Engineering. Montreal: IEEE, 2019. 1176–1186. [doi: 10.1109/ICSE.2019.00120]

[3] Scholz B, Jordan H, Subotić P, Westmann T. On fast large-scale program analysis in Datalog. In: Proc. of the 25th Int'l Conf. on Compiler Construction. Barcelona: ACM, 2016. 196–206. [doi: 10.1145/2892208.2892226]

[4] Bravenboer M, Smaragdakis Y. Strictly declarative specification of sophisticated points-to analyses. In: Proc. of the 24th ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications. Auckland: ACM, 2009. 243–262. [doi: 10.1145/1640089.1640108]

[5] Wang K, Hussain A, Zuo ZQ, Xu GQ, Amiri Sani A. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. ACM SIGPLAN Notices, 2017, 52(4): 389–404. [doi: 10.1145/3093336.3037744]

[6] Fogel A, Fung S, Pedrosa L, Walraed-Sullivan M, Govindan R, Mahajan R, Millstein T. A general approach to network configuration analysis. In: Proc. of the 12th USENIX Conf. on Networked Systems Design and Implementation (NSDI). Oakland: USENIX Association, 2015. 469–483.

[7] Zhang P, Huang YH, Gember-Jacobson A, Shi WB, Liu X, Yang HK, Zuo ZQ. Incremental network configuration verification. In: Proc. of the 19th ACM Workshop on Hot Topics in Networks. ACM, 2020. 81–87. [doi: 10.1145/3422604.3425936]

[8] Shkapsky A, Yang MH, Interlandi M, Chiu H, Condie T, Zaniolo C. Big data analytics with Datalog queries on spark. In: Proc. of the 2016 Int'l Conf. on Management of Data. San Francisco: ACM, 2016. 1135–1149. [doi: 10.1145/2882903.2915229]

[9] Moustafa WE, Papavasileiou V, Yocum K, Deutsch A. Datalography: Scaling Datalog graph analytics on graph processing systems. In: Proc. of the 2016 IEEE Int'l Conf. on Big Data. Washington: IEEE, 2016. 56–65. [doi: 10.1109/BigData.2016.7840589]

[10] Wang K, Zuo ZQ, Thorpe J, Nguyen TQ, Xu GH. RStream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In: Proc. of the 13th USENIX Symp. on Operating Systems Design and Implementation (OSDI). Carlsbad: USENIX Association, 2018. 763–782.

[11] Flores-Montoya A, Schulte E. Datalog disassembly. In: Proc. of the 29th USENIX Conf. on Security Symp. (Security). Berkeley:

USENIX Association, 2020. 61.

[12] Whaley J, Avots D, Carbin M, Lam MS. Using Datalog with binary decision diagrams for program analysis. In: Proc. of the 3rd Asian Symp. on Programming Languages and Systems. Tsukuba: Springer, 2005. 97–118. [doi: 10.1007/11575467_8]

[13] Hoder K, Bjørner N, de Moura L. μZ—An efficient engine for fixed points with constraints. In: Proc. of the 23rd Int'l Conf. on Computer Aided Verification (CAV). Snowbird: Springer, 2011. 457–462. [doi: 10.1007/978-3-642-22110-1_36]

[14] Aref M, ten Cate B, Green TJ, Kimelfeld B, Olteanu D, Pasalic E, Veldhuizen TL, Washburn G. Design and implementation of the LogicBlox system. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. Melbourne: ACM, 2015. 1371–1382. [doi: 10.1145/2723372.2742796]

[15] Seo J, Park J, Shin J, Lam MS. Distributed socialite: A Datalog-based language for large-scale graph analysis. Proc. of the VLDB Endowment, 2013, 6(14): 1906–1917. [doi: 10.14778/2556549.2556572]

[16] Tang JQ, Fang BX, Hu MZ, Wang W. Research on I/O optimizations in out-of-core computation. Journal of Computer Research and Development, 2005, 42(10): 1820–1825 (in Chinese with English abstract). [doi: 10.1360/crad20051028]

[17] Green TJ, Huang SS, Loo BT, Zhou WC. Datalog and Recursive Query Processing. Boston: Now Publishers, 2013. [doi: 10.1561/1900000017]

[18] Balbin I, Ramamohanarao K. A generalization of the differential approach to recursive query evaluation. The Journal of Logic Programming, 1987, 4(3): 259–262. [doi: 10.1016/0743-1066(87)90004-5]

[19] Wang YM, Shi BL. The group of DATALOG programs and its applications. Ruan Jian Xue Bao/Journal of Software, 1997, 8(9): 641–646 (in Chinese with English abstract). http://www.jos.org.cn/jos/article/abstract/19970901?st=search [doi: 10.13328/j.cnki.jos.1997.09.001]

[20] Abiteboul S, Hull R, Vianu V. Foundations of Databases. Reading: Addison-Wesley, 1995. 8.

[21] Hulin G. Parallel processing of recursive queries in distributed architectures. In: Proc. of the 15th Int'l Conf. on Very Large Data Bases. Amsterdam: Morgan Kaufmann Publishers Inc., 1989. 87–96.

[22] Ganguly S, Silberschatz A, Tsur S. A framework for the parallel processing of Datalog queries. ACM SIGMOD Record, 1990, 19(2): 143–152. [doi: 10.1145/93605.98724]

[23] Fan ZW, Zhu JQ, Zhang ZY, Albarghouthi A, Koutris P, Patel JM. Scaling-up in-memory Datalog processing: Observations and techniques. Proc. of the VLDB Endowment, 2019, 12(6): 695–708. [doi: 10.14778/3311880.3311886]

[24] Houtsma MAW, Apers PMG. Algebraic optimization of recursive queries. Data & Knowledge Engineering, 1992, 7(4): 299–325. [doi: 10.1016/0169-023X(92)90029-B]

[25] Roy A, Mihailovic I, Zwaenepoel W. X-stream: Edge-centric graph processing using streaming partitions. In: Proc. of the 24th ACM Symp. on Operating Systems Principles. Pennsylvania: ACM, 2013. 472–488. [doi: 10.1145/2517349.2522740]

[26] Prata S. C++ Primer Plus. 6th ed., Upper Saddle River: Addison-Wesley Professional, 2011.

[27] The souffle datalog engine. 2020. https://souffle-lang.github.io/

[28] Leskovec J, Sosič R. SNAP: A general-purpose network analysis and graph-mining library. ACM Trans. on Intelligent Systems and Technology, 2017, 8(1): 1. [doi: 10.1145/2898361]

[29] Azad A, Buluç A, Gilbert J. Parallel triangle counting and enumeration using matrix algebra. In: Proc. of the 2015 IEEE Int'l Parallel and Distributed Processing Symp. Workshop. Hyderaba: IEEE, 2015. 804–811. [doi: 10.1109/IPDPSW.2015.75]

[30] Mawhirter D, Wu B. AutoMine: Harmonizing high-level abstraction and high performance for graph mining. In: Proc. of the 27th ACM Symp. on Operating Systems Principles. Huntsville: ACM, 2019. 509–523. [doi: 10.1145/3341301.3359633]

[31] Becchetti L, Boldi P, Castillo C, Gionis A. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In: Proc. of the 14th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. Las Vegas: ACM, 2008. 16–24. [doi: 10.1145/1401890.1401898]

[32] Kyrola A, Blelloch GE, Guestrin C. GraphChi: Large-scale graph computation on just a PC. In: Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI). Hollywood: USENIX Association, 2012. 31–46.

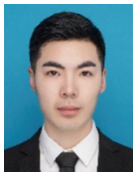
附中文参考文献:

[16] 唐剑琪, 方滨兴, 胡铭曾, 王威. 核外计算中的几种I/O优化方法. 计算机研究与发展, 2005, 42(10): 1820–1825. [doi: 10.1360/crad20051028]

[19] 王云明, 施伯乐. DATALOG程序的组及其应用. 软件学报, 1997, 8(9): 641–646. http://www.jos.org.cn/jos/article/abstract/19970901?st=search [doi: 10.13328/j.cnki.jos.1997.09.001]

张奕裕(1997－), 男, 博士生, 主要研究领域为系统软件, 编译优化.

左志强(1986－), 男, 博士, 副研究员, CCF 专业会员, 主要研究领域为系统软件, 软件工程, 程序语言.



王归航(1997－), 男, 硕士生, 主要研究领域为系统软件, 程序语言.

李宣东(1963－), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为软件工程, 形式化方法.