软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(3):814−831 [doi: 10.13328/j.cnki.jos.006452]

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

AlphaQO: 鲁棒的学习型查询优化器*

余 翔[1], 柴成亮[1], 张辛宁[2], 汤 南[3], 孙 佶[1], 李国良[1]

[1](清华大学 计算机科学与技术系, 北京 100084)

[2](浙江大学 计算机科学与技术学院, 浙江 杭州 310027)

[3](Qatar Computing Research Institute, Hamad Bin Khalifa University, Doha, Qatar)

通信作者: 李国良, E-mail: liguoliang@tsinghua.edu.cn

摘 要: 由深度学习驱动的学习型查询优化器正在越来越广泛地受到研究者的关注, 这些优化器往往能够取得近似甚至超过传统商业优化器的性能. 与传统优化器不同的是, 一个成功的学习型优化器往往依赖于足够多的高质量的负载查询作为训练数据. 低质量的训练查询会导致学习型优化器在未来的查询上失效. 提出了基于强化学习的鲁棒的学习型查询优化器训练框架 AlphaQO, 提前找到学习型优化器做不好的查询, 以提高学习型优化器的鲁棒性. AlphaQO 中存在两个重要部分: 查询生成器和学习型优化器. 查询生成器的目标是生成"难"的查询(传统优化器做得好, 但是学习型优化器反而做得不好的查询). 学习型优化器利用这些生成的查询进行测试和训练,并提供反馈让查询生成器进行更新. 系统迭代交替的运行上述两个部分, 分别进行训练. 目的在于在提供尽量少的信息和消耗足够小的时间下找到足够多"难"的并且未见的查询给优化器训练, 以提高学习型优化器的鲁棒性. 实验结果显示: 该生成器会提供越来越难的训练查询给学习型优化器; 同时, 这些查询能够提升学习型优化器的性能.

关键词: 学习型优化器; 鲁棒性; AI4DB; 数据库; 强化学习; 查询生成

中图法分类号: TP311

中文引用格式: 余翔, 柴成亮, 张辛宁, 汤南, 孙佶, 李国良. AlphaQO: 鲁棒的学习型查询优化器. 软件学报, 2022, 33(3): 814–831. http://www.jos.org.cn/1000-9825/6452.htm

英文引用格式: Yu X, Chai CL, Zhang XN, Tang N, Sun J, Li GL. AlphaQO: Robust Learned Query Optimizer. Ruan Jian Xue Bao/ Journal of Software, 2022, 33(3): 814–831 (in Chinese). http://www.jos.org.cn/1000-9825/6452.htm

AlphaQO: Robust Learned Query Optimizer

YU Xiang[1], CHAI Cheng-Liang[1], ZHANG Xin-Ning[2], TANG Nan[3], SUN Ji[1], LI Guo-Liang[1]

[1](Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

[2](College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China)

[3](Qatar Computing Research Institute, Hamad Bin Khalifa University, Doha, Qatar)

Abstract: Learned database query optimizers, which are typically empowered by (deep) learning models, have attracted significant attention recently, because they can offer similar or even better performance than the state-of-the-art commercial optimizers that require hundreds of expert-hours to tune. A crucial factor of successfully training learned optimizers is training queries. Unfortunately, a good query workload that is sufficient for training learned optimizers is not always available. This study proposes a framework, called AlphaQO, on generating queries for learned optimizers with reinforcement learning (RL). AlphaQO is a loop system that consists of two main components, query generator and learned optimizer. Query generator aims at generating "hard" queries (i.e., those queries that the learned optimizer provides poor estimates). The learned optimizer will be trained using generated queries, as well as providing feedbacks (in terms of numerical rewards) to the query generator. If the generated queries are good, the query generator will get a high reward;

* 基金项目: 国家自然科学基金(61925205, 61632016); 华为和好未来资助项目
本文由"数据库系统新型技术"专题特约编辑李国良教授、于戈教授、杨俊教授和范举教授推荐.
收稿时间: 2021-06-30; 修改时间: 2021-07-31; 采用时间: 2021-09-13; jos 在线出版时间: 2021-10-21

otherwise, the query generator will get a low reward. The above process is performed iteratively, with the main goal that within a small budget, the learned optimizer can be trained and generalized well to a wide range of unseen queries. Extensive experiments show that AlphaQO can generate a relatively small number of queries and train a learned optimizer to outperform commercial optimizers. Moreover, learned optimizers need much less queries from AlphaQO than randomly generated queries, in order to well train the learned optimizer.
Key words: learned optimizer; robustness; AI4DB; database; reinforcement learning; query generation

查询优化器决定着数据库引擎执行一条查询时所采用的具体方案, 是决定数据库系统成功与否的核心部件[1−3]. 传统的查询优化器[4−7]主要使用的是基于代价的方法: 给定一个 SQL 查询, 优化器会尝试枚举不同的可能的执行方案, 然后交给估计器估计每个方案的执行代价, 从中选出最小代价的那个方案. 传统的方法往往受限于两个问题.

- L1: 枚举代价过高. 可能的方案空间往往是指数级的, 基于动态规划的方法需要大量的计算, 而启发式的方法不一定能取得较好的结果.
- L2: 代价估计模型难以维护. 维护一个好的估计器是非常难的, 不同的系统环境、存储结构、数据集需要不同的代价模型参数; 同时, 基数估计的准确性也显著影响着估计器的效果.

近段时间, 一些研究者开始使用基于学习的方法对传统优化器进行改进[8−11], 并在 L1 和 L2 上取得了不错的效果. 他们研究发现, 对比目前已有的数据库管理系统, 学习型优化器可以提供相近甚至更高的性能. 然而学习型优化器往往使用的是深度学习模型, 效果往往依赖于模型训练的情况. 一般来讲有两个难点.

- L3: 冷启动问题. 如果学习型优化器依赖于大量的真实查询来训练, 一直在线上收集查询并训练, 比如 NEO[8], 我们会需要很久的时间让学习型优化器收集数据、训练, 以产生较好的效果.
- L4: 鲁棒性问题. 基于学习的优化器的质量往往取决于训练集的质量, 在收集数据的质量不高, 或者使用随机生成的查询来训练的情况下, 学习型的优化器不一定能够对未来的真实查询取得一个较好的结果, 从而导致系统的失效.

因此, 我们这篇文章主要解决一个问题: 不仅仅依赖于线上收集大量真实的负载, 我们能否提前发现学习型优化器做得不好的查询, 交给学习型优化器训练, 提高其鲁棒性?

一个自然而然的想法是生成大量的随机查询, 尽可能地覆盖所有的情况. 然而, 这个做法会十分消耗时间而并不实用: 首先, 可能的查询数量是十分巨大的, 不同的连接、不同的谓词选择, 会产生指数级别的查询搜索空间; 同时, 学习型优化器训练一条查询的代价也是非常高的, 需要使用不同的方案在数据库管理系统上执行该查询得到足够的训练数据. 这些都导致了单纯的随机生成无法解决我们的问题.

我们需要尽量减小生成查询的数量, 提高生成查询的质量, 选择有效的查询交给学习型优化器作为训练数据. 考虑到学习型优化器一般从那些已经做得好的查询上学不到太多信息, 我们希望查询生成器能够识别学习型优化器目前做得不好的查询(质量高). 我们使用迭代的方式, 首先由查询生成器生成查询提供给学习型优化器测试, 然后得到反馈并改进查询生成器的生成方向.

- 贡献

在这篇文章中, 我们提出了一个通用的鲁棒优化器训练框架 AlphaQO, 目的是基于深度强化学习(deep reinforment learning, DRL)发现学习型优化器做得不好的查询, 并以此提高学习型优化器的鲁棒性. 图 1 总结了我们 AlphaQO 的系统架构, 这是一个循环的结构, 主要包含了一个查询生成器(主体, agent)和一个学习型优化器(环境, environment). 查询生成器迭代地生成查询, 交给学习型优化器. 学习型优化器将测试这些查询, 并给出对应的奖励值(reward), 用于指导查询生成器的生成方向; 之后, 用这些查询训练自己提高性能. 更具体地来讲, 在第 i 个迭代过程中, 查询生成器会根据策略函数生成查询集合 Q_i, 交给学习型优化器. 学习型优化器拿到这个查询集合 Q_i 以后, 会与后端的数据库管理系统(DBMS)交互, 来对比看是否超过了传统优化器的性能, 并根据对比情况返回奖励值集合 R_i 给查询生成器. 最后, 学习优化器利用这些查询 Q_i 作为训练数据进行训练, 并更新系统的状态为 S_i. 这些奖励值 R_i 会被用于生成器的策略函数, 指导查询生成器下一次迭代过程中的结果.

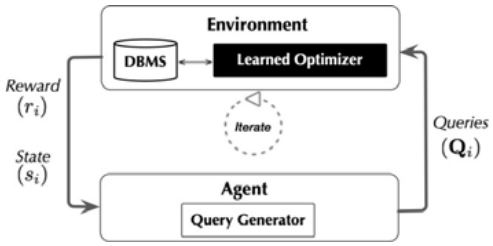


图 1 AlphaQO 系统架构

我们的贡献主要总结如下:

(1) 形式化地将查询生成问题变成了图结构的生成问题. 给出了查询、查询完全图等相关数据结构的定义和符号系统(见第 1 节).

(2) 提出了一个通用框架 AlphaQO, 并借助了强化学习这个机器学习的架构, 去形式化描述了这个查询生成器的生成过程(见第 2 节).

(3) 提出了 AlphaQO 生成器的生成策略模型, 并解决了它的更新方法和训练过程. 策略模型决定着查询生成器的生成方法, 是 AlphaQO 的重要部分. 我们提出了基于 LSTM 模型的策略模型(见第 3.1 节), 并研究了它的更新方法(见第 3.2 节), 给出了如何借助少量真实查询提高生成器生成查询质量的方法(见第 3.3 节), 并给出了整个系统的训练方法(见第 3.4 节).

(4) 我们通过在两个真实数据集 JOB 和 JOB-ext 进行实验测实, 验证了 AlphaQO 系统的有效性. 对比基线做法, AlphaQO 能够显著地发现学习型优化器做得不好的查询, 提高了学习型优化器在未见查询上的表现, 增强了系统. 同时, 结合小量真实负载(10%)的情况下, 可以将学习优化器的性能提升至使用大量真实负载(70%)训练的效果(见第 4 节).

1 研究背景

在这一章节中, 我们首先在第 1.1 节介绍了一些符号,在第 1.2 节中介绍我们研究的问题, 最后, 我们会在第 1.3 节中引出强化学习, 并解释应用强化学习解决我们这个问题是合理的.

1.1 符 号

- 数据集

数据库 D 包含了许多的表 $T_1,\ldots,T_n$. 符号 $att(T_i)=\{c_{i1},\ldots,c_{im}\}$表示表 T_i 包含的列. 符号 $attr(D)=attr(T_1)\cup\ldots\cup attr(T_n)$是所有列的集合. 需要注意的是, 如果不同表的两个列的名字相同(比如 T_1 的 A 列和 T_2 的 A 列), 它们会被当作不同的列. 表与表之间通过主外键(PK-FK)关联, 这些键用于进行连接操作.

- 查询

在这篇文章中, 我们主要考虑基础的查询-投影-连接(SPJ)查询. GroupBy 语句和对应的聚合操作 Agg=(max,min,count,avg,sum)作为可选的拓展, 提高生成的查询的多样性. 聚类操作一般会被最后执行, 对执行计划的影响比较少. SPJ 查询是 SQL 查询中重要且具有代表性的一个子集, 在数据库的研究中被广泛使用, 比如查询优化[12]、逆向查询工程[13]等等领域. 带有聚合操作的 SPJ 查询更是被广泛地应用在各类真实场景中, 特别是在线任务分析(OLAP)的场景. 我们的模型也可以通过很简单的拓展以产生更加复杂的查询, 但问题的关键的是, 如何在巨大的合法查询空间中找到需要的查询.

- 学习型优化器

一般来讲, 学习型优化器会学习一个函数 f_θ, θ是该函数的参数. 当给定一个查询 q 时, 我们以 q 作为函数的输入, $f_\theta(q)$给出查询 q 的物理计划, 比如决定查询 q 中表的连接顺序.

例 1: 考虑一个查询 q, 一共连接了 3 张表.

Select *

From T_1, T_2, T_3

ERE $T_1.id=T_2.id$ and $T_2.id=T_3.id$.

查询优化中的一个基础问题是连接顺序选择. 不同的连接顺序选择决定了不同的物理执行计划, 不同的计划之间可能有非常大的性能差距.

- (O1) $((T_1 \triangleright\triangleleft T_2) \triangleright\triangleleft T_3$——首先连接 T_1 和 T_2;
- (O2) $T_1 \triangleright\triangleleft (T_2 \triangleright\triangleleft T_3)$——首先连接 T_2 和 T_3.

学习型的优化器训练之后, 能够选择一个“好”的查询计划, 比如(O2).

目前最好的学习型优化器都使用了基于深度学习的模型. ReJoin[10]和 DQ[9]最先提出使用深度强化学习从一堆已有的查询集合 Q 中进行学习模型, 估计每个计划需要的代价, 并从中选取代价最小的计划. NEO[8]和 RTOS[9]则使用了基于树结构的神经网络进行优化, 以便更好地捕捉执行计划的结构特征, 提升模型的准确度, 生成更好的执行计划.

特别地, 学习型优化器的效果严重依赖于训练查询的质量. 但是获取真实的查询负载是困难的, 特别是当系统冷启动时. 而且即使获得了真实的负载, 如果规模不够大, 那么这些负载也不一定会有足够的代表性. 这些困难都会导致学习型优化器不能很好地应对未来的及时查询.

1.2 问题描述

我们令 Q^*表示数据库 D 中所有可能的 SPJ 查询.

- 问题

给定一个数据库 D、一个查询完全图 G 以及谓词集合 P, 针对一个学习型优化器 f_θ, 定义查询生成问题为利用 D, G, P 构造一个查询子集 $Q^g \subseteq Q^*$, 这些查询对于 f_θ是困难的. f_θ使用这些查询进行训练, 能够对未来的查询生成更好的计划. 查询完全图 G 和谓词集合 P 的定义在下文会具体介绍, 它们可以自动生成(比如 PK-FK 连接)或者用户通过辅助工具(比如用户指定目标列进行采样谓词). D, G, P 作为系统信息由用户提前提供, 是 AlphaQO 的输入.

暴力的方案是枚举整个 Q^*的空间, 选取合适的查询 q 交给 f_θ去学习. 这个方法会明显的存在两个问题: (1) 搜索空间是巨大的, 我们难以枚举; (2) 查询之间不是无关的, 一条查询可能被另外一条查询包含, 或者两条查询之间存在着比较大的重叠, 学习型优化器学习了一条查询之后可以做好另外一条查询. 为了解决这些问题, 我们首先需要给出一个数据结构能够描述整个搜索空间(Q^*)和查询子集(Q^g)的生成过程.

- 查询完全图

下面我们引入用于描述整个 SPJ 查询空间的数据结构——查询完全图 $G(V,E)$. $G(V,E)$定义在一个数据库 D 上, 该图包含顶点集合 V 和边集合 E. 一共有两种类型的节点: 列节点 V_C 和操作节点 V_O, $V=V_C \cup V_O$.

(1) 表的每个列 $c \in attr(D)$都是一个节点 $v_c \in V_C$.

(2) 令 O={Project,Predicate,GroupBy,Agg}表示所有操作节点的集合, 每个操作符 $o \in O$ 是一个节点 $v_o \in V_O$.

共有 3 种不同的边: $E=E_T \cup E_J \cup E_O$, 其中, E_T, E_J 和 E_O 分别对应着表内边、连接边和操作符边. 其定义如下:

(1) 表内边 $u,v \in E_T$. 当 u 和 v 都是列节点($u,v \in V_C$), 并且它们来自同一张表.

(2) 连接边 $u,v \in E_J$. 当 u 和 v 都是列节点($u,v \in V_C$), 并且它们来自不同的表, 它们可以被用作连接的键.

(3) 操作符边 $u,v \in E_O$. 当 u 是一个列节点 $u \in V_C$, 并且 v 是一个操作符节点 $v \in V_O$, 代表操作符 v 可以被运用在列 u 上.

表内边 E_T和连接边 E_J都是很容易理解的, 我们进一步来讨论下操作符边 E_O, 考虑操作符边 $e_{u,c}$中不同种类的 u.

(1) 当 u 是投影(project)节点: 边可以被应用在任意列节点 c 上. 当多个列与都连接到该节点时, 表示该

列出现在结果中.

(2) 当 u 是分组(GroupBy)节点: u 可以被连接在一个或者多个列节点(比如列 A 和列 B)上, 表示在该列上进行一个分组操作, 比如 GroupBy(AB).

(3) 当 u 是谓词(predica)节点: u 可以被应用在任意列节点 c 上, 表示对该列进行一个选择操作. 需要注意的是, 我们提前准备了目标列上的候选谓词集合, 需要从中选择合适的谓词. 当某个节点 V_C 多次连接谓词节点时, 我们将多个谓词进行合操作.

(4) 当 u 是聚合(aggregation)节点: 用来表明我们在节点 c 上进行了一个聚合操作:

$$u\in\{\max,\min,\text{count},\text{avg},\text{sum}\}.$$

例 2: 考虑一个小的数据库, 该数据库有如下两个表: $T_1(name,id)$和 $T_2(id,score)$.

T_1 和 T_2 能够使用 id 列进行连接, 对应的连接图如图 2 所示. 它有 4 个列节点和 4 个操作符节点. 有两个表内边: $(T_1.name,T_1.id)$和$(T_2.id,T_2.score)$. 有一条连接边$(T_1.id,T_2.id)$和其他一些操作符边. 这个查询完全图是查询的抽象表示, 但是几乎能够帮我们枚举任意一个 SPJ 查询.

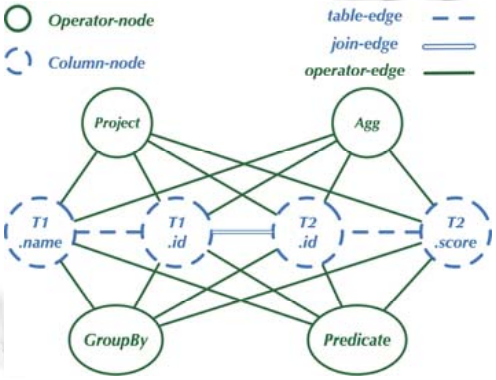


图 2 一个简单的查询完全图

- 查询图

一个查询图 G^q 产生于查询全图 G 的一个连通子图 $G'\subseteq G$, 并且 G'所有的谓词节点都被实例化为具体的谓词. 更具体地讲, 一个连通子图 G'是一个模板, 而 G^q 就是对应这个模板产生的一个具体的真实查询. 我们很自然地能够把 G^q 的生成分成两步: 1) 寻找图 G 的连通子图 G'; 2) 实例化连通子图 G'中的谓词节点, 生成最终的查询图 G^q.

例 3: 图 3 中, 我们生成了一个 SQL 查询. 首先, 我们获得连通全图的连通子图; 然后, 我们实例化了谓词节点. 选择的两个谓词分别是 $T_1.id$>100 和“Mike”. 通过这两步, 我们就生成了一个具体的查询.

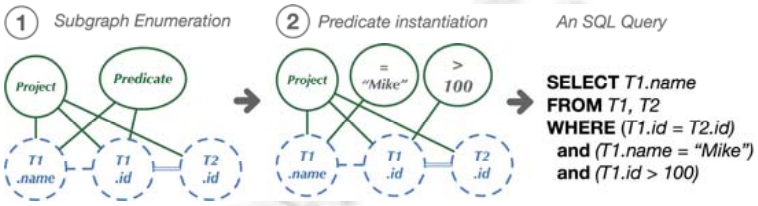


图 3 从连通子图变成查询图

介绍了具体的符号之后, 我们讨论如下具体的查询生成的具体过程.

查询生成的过程一般可以被分为静态生成和动态生成. 生成模型的目标是生成一个困难查询集合 Q^g, 这些查询将会用于提升学习型优化器 f_θ的性能. 静态生成是指一次性生成我们最终需要的查询, 而动态生成是指一批一批 Q^{b_i} 地生成, 最终形成完整查询 $Q^g=Q^{b_1}\cup\ldots\cup Q^{b_k}$. 考虑到每一条查询进行训练之后, 学习型优化

器可以泛化一些原先做得不好的查询, 那么这里我们使用动态生成的方式更加合适. 更具体地, 在生成查询时, 查询生成器每次生成一个小的查询集合 Q^{b_i}, 然后将这批查询交给学习优化器 f_θ去评估, 得到对应的反馈 R^{b_i}; 同时, 让学习型优化器基于这些查询进行训练, 避免每次都生成类似的困难查询.

1.3 强化学习(reinforcement learning)

强化学习(简称 RL)是机器学习中一类重要的基础方法. 和基于监督的方法不同, 强化学习可以不依赖于提前标注好的数据, 而是通过和环境的交互, 采集数据再进行训练. 强化学习要解决的一个关键问题就是, 如何在探索新的知识和利用已有知识之间取得平衡.

一般来讲, RL 包含了 6 个主要部分: 智能体(agent)、动作(action)、策略函数(policy)、环境(environment)、状态(state)和奖励(reward). “智能体”是整个系统的大脑, 负责学习并且给出系统需要的决策. 智能体给出的决策被称为“动作”. 而“策略函数”就是智能体给出动作的函数, 函数根据目前系统所处的“状态”选出做得好的那个动作. “环境”则是智能体进行学习和决策的地方. 每次智能体采取一个动作之后, 环境负责维护新的状态, 并同时给出一个“奖励”值. 奖励值用于引导策略函数选择最好的动作.

因此, RL 仅仅是提供了一个解决问题的框架, 就类似 Hadoop 和 Spark 在分布式计算中的地位. 但是具体如何把问题抽象成一个强化学习问题、如何设计对应的策略和方法去求解各个问题, 仍存在着许多的选择和困难. 在我们的问题中, 主要可以被总结为两个难点.

(1) 从设计角度来看: RL 主要求解的问题是根据系统的环境给出什么动作, 能够达到获得最大奖励值的状态. 我们需要将查询生成问题与 RL 框架进行抽象, 使该问题对应到强化学习每个部分的输入和输出, 并且提供对应的数据结构进行表示. 这个我们具体在第 2 节进行讨论.

(2) 从算法角度来看: 强化学习中有许多的算法问题需要解决. 比如在我们的场景中, 一个生成过程最终的状态(一个完整的查询)的奖励值是非常直观的, 而中途状态(一个不完整的查询)的奖励值是难以衡量的. 而且如何从这些奖励值进行学习, 引导我们的策略函数在之后生成更有效的查询, 也是一个复杂的问题. 这个我们具体会在第 3 节进行介绍.

2 AlphaQO 的工作流程

下面我们介绍 AlphaQO 这个查询生成器的模型流程. 该系统的整个的流程如图 4 所示.

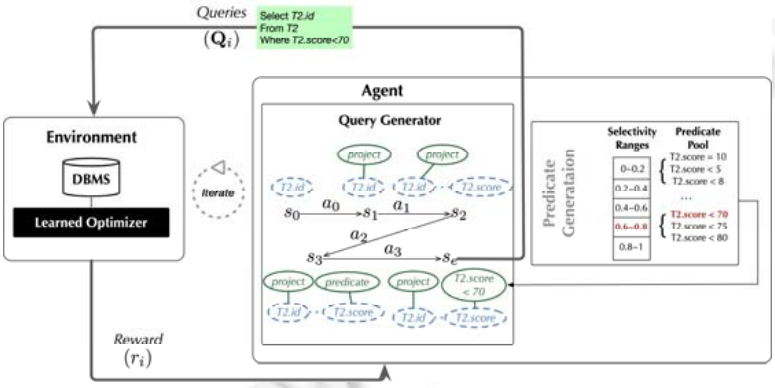


图 4 AlphaQO 工作流程

2.1 智能体和环境

AlphaQO 主要目标是通过智能体和环境的不断交互, 发现学习型优化器做得不好的查询.

环境由一个学习型优化器和一个传统的数据库管理系统得到. 给定一个查询, 学习型优化器生成计划并交给数据库执行引擎去执行. 与传统优化器进行对比得到这条查询的评估, 作为奖励值改进生成器, 同时用

执行的结果训练并提升学习型优化器. 我们选择执行时间(latency)表现作为我们奖励值标准.

智能体就是我们的生成器, 主要任务就是根据目前的构造查询的状态, 确定下一步的构造的操作(比如增加一个连接或者谓词), 一直到我们构造了一个完整的查询. 我们使用长期奖励值来评估智能体的每个可选动作, 比如采取该动作之后, 能够产生的期望查询奖励值.

2.2 AlphaQO的四大要素

- 状态

状态维护着查询生成器在一次生成过程中的所有信息. 一次生成过程由许多的状态组成, 我们使用$\{s_0,s_1,\ldots,s_e\}$表示.

s_0为一个起点状态, 选择查询完全图中的一个任意节点. s_e为一个终止状态, 表示我们已经完成了一个完整的查询 q, 并且 q 将被送到环境中去. 其他状态$\{s_1,\ldots,s_n\}$为查询完全图 G 的一个子图, 或者已经被实例化的查询图 G^q.

例 4: 图 4 给定了 5 个状态$\{s_0,s_1,s_2,s_3,s_e\}$, 其中, 起始状态 s_0 表示了一个单独的节点 $T_2.id$; 第 1 个状态 s_1 在 s_0 的基础上增加了一个投影边; 第 2 个状态 s_2 在 s_1 的基础上连接了一个连接边到 $T_2.score$; 第 3 个状态 s_3 在 s_2 的基础上增加了一个谓词节点; 在终止节点 s_e, 我们把谓词节点实例化成 $T_2.score<70$, 并且将该查询送给学习型优化器中.

- 动作

一个动作 a_i 决定着当给定一个状态 s_i, 它会转移到具体的哪一个状态 s_{i+1}, 在 AlphaQO 表示为加入一条连接边或者增加一个谓词. 在 AlphaQO 中, 我们不停地选择下一个动作, 直到达到了一个预定的最大的动作数量或者抵达了终点 s_e 状态.

例 5: 图 4 分别展示了在状态$\{s_0,s_1,s_2,s_3\}$采取的 4 个动作$\{a_0,a_1,a_2,a_3\}$以及它们如何变成下一个动作的过程. 其中, 谓词的边相对于其他边是特殊的. 与其他边一旦选择就出现在我们最终的查询信息不同, 谓词的边是一个抽象的表达, 连接了谓词的节点之后还需要进行一次实例化操作, 才能够得到最终的具体谓词.

谓词的实例化. 连接了一个谓词的节点之后, 需要将它实例化成一个具体的谓词才能够形成最终的查询. 针对这个问题, 我们首先针对每个目标列 c(比如 $T_2.score$)收集一些谓词集合 P^c(比如 $T_2.score<70$, $T_2.score<75$, …). 考虑传统的优化器往往使用选择度(selectivity)作为谓词的指标, 我们也利用该信息辅助我们谓词的选择. 具体的步骤如下:

(1) 选择度区间划分: 首先将选择度划分为 k 个区间, 比如[0,0.2],…,(0.8,1].

(2) 生成谓词池: 我们将提前准备的谓词集合 P 中选取与列有关的谓词集合 P^c, 并且根据步骤(1)中准备的选择度区间, 将每个谓词 $p\in P^c$ 放到对应的区间中去.

(3) 选择谓词: 每当生成一个谓词时, 我们实际上会先选择一个谓词区间节点, 然后从该区间中采样一个谓词.

需要注意的是, 上述的步骤(1)和步骤(2)步骤会在生成开始前完成, 属于预处理的数据结构部分.

例 6: 我们来具体描述图 4 中 $T_2.score$ 的生成过程. 状态 s_2 采取动作 a_2(连接一个谓词)将状态转移至 s_3, 并且最终在 s_e 产生最终的查询.

(1) 首先, $T_2.score$ 的选择度区间被平均分为 5 份区间[0,0.2],…,(0.8,1], 并且将谓词分到对应的谓词池中;

(2) a_2 是一个选择谓词节点的动作, 实际上它会选择一个具体的选择度区间(0.6,0.8];

(3) a_3 动作会最终实例化出我们的查询, 根据之前的选择度区间(0.6,0.8]采样一个最终的谓词:

$$T_2.score<70.$$

- 策略

策略函数$\pi(s_i)$用于给定状态 s_i 时返回具体的动作 a_i. 一般来讲, 策略函数可以被区分为确定性策略 $a_i=\pi(s_i)$和不确定策略 $a_i\sim\pi(s_i)$. 确定性策略可以被认为是不确定策略的某个动作概率为 1 时的特殊情况. 我们这里使用的是更加一般的不确定策略 $a_i\sim\pi(s_i)$. AlphaQO 使用了一个长短期记忆(LSTM)网络构建了我们的策略

函数，针对每个状态生成采取每个动作的概率$\pi(s_i)$，并根据这些概率值采样出下一步的动作 a_i. 我们会在第 3 节中具体的介绍该网络.

- 奖励值

奖励值 $r(s_i,a_i)$是一个和状态 s_i 和动作 a_i 有关的数值，用于表示我们在该状态 s_i 下选取动作 a_i 的收益. 在这里，我们使用长期收益值，即从状态 s_i 采取动作 a_i 后，最终产生的查询的期望收益来表示. 比如，下棋过程中，我们利用每步最后带来的胜率作为每步的评估.

具体来说，对于最终生成的查询q来讲，给它一个高的奖励值表示它是学习型优化器做得不好的查询. 我们采取和传统优化器进行对比作为评估的标准，即希望发现学习型优化器对比传统的优化器表现得更差的查询. 一个更高的奖励值，意味着我们希望生成越多的和这个查询 q 相关的查询. 与之相对应，一个更低的奖励值代表着这个查询，学习型优化器已经能够做得比较好了，我们希望生成不同的查询.

奖励值会用来帮助我们的策略函数更新参数. 为了能够使奖励值 $r(s_i,a_i)$的定义满足我们上述的要求，对于查询 q，我们定义 r_q 表示学习型优化器和传统优化器的给出计划的执行时间的比值 $r_q=\frac{l_q}{t_q}$. 其中，l_q 是学习型优化器给出计划的执行时间，t_q 是传统优化器给出的计划的执行时间. 对于终止状态 s_e，它对应到一个查询 q，我们可以用 r_q 给出它的奖励值定义，即 $r(s_e)=r_q$. 但是对于中间构造的状态 s_i，我们并没有直观的一个奖励值，这里我们定义它在策略函数指引下产生查询的期望奖励值作为中间状态的奖励值. 具体方法见第 3 节.

3 策略函数: 模型、更新和训练

在这一节中，我们具体介绍针对策略部分设计的神经网络模型，如何使用我们从环境中获得的奖励值更新模型参数的方法，如何在给出小量真实查询下提高生成器的生成质量，以及如何联合生成器和学习型优化器进行 AlphaQO 的迭代训练.

3.1 策略函数的模型

首先，我们的策略函数会根据我们之前访问过的状态序列$\{(s_0,a_0),\ldots,(s_{i-1},a_{i-1})\}$给出下一步的操作 a_i，而生成新的状态 s_{i+1}. 那么自然的一个想法是，选择循环神经网络结构来处理这个序列问题. 我们选择了长短时记忆网络这一能够处理长序列的模型作为我们的策略模型$\pi_\theta(s)$，其中，θ是我们神经网络的参数. 将我们的状态序列作为网络的输入，得到每一次的输出动作.

图 5 展示了我们具体的 LSTM 网络设计，包含了 LSTM(L)单元和动作(action, A)单元. L 单元被用于计算神经网络的中间表示 h_t，记录了目前状态序列的内部信息；A 单元则被用于根据神经网络给出的隐藏信息 h_t，给出具体的动作 a_t.

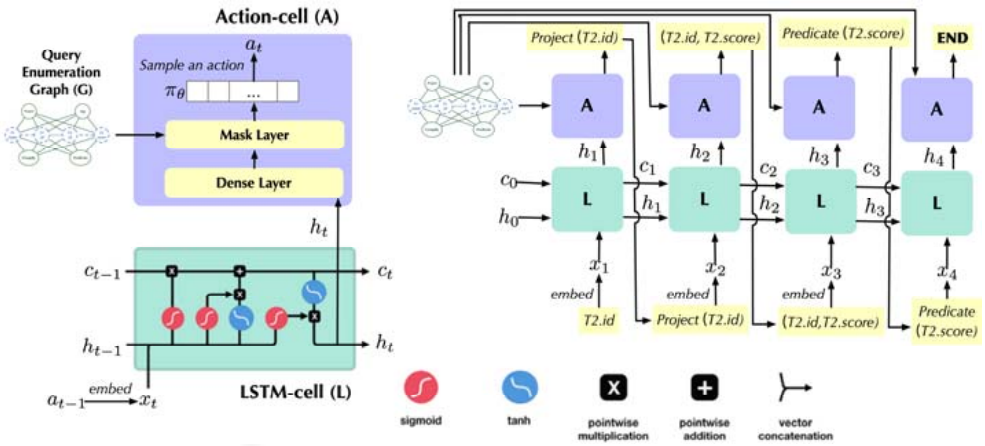


图 5　策略函数的 LSTM 模型

- LSTM 单元

这个就是标准的 LSTM 结构[14]. 在 t 时刻时, 它有 3 个输入 h_{t-1}, c_{t-1}, x_t. h_{t-1} 和 c_{t-1} 分别表示 LSTM 单元在前一个状态时产生的信息表示和记忆表示, 可以被认为当前状态的一个整个表示; x_t 则是我们前一次采取的动作 a_{t-1} 进行向量化表示后的结果. 它有两个输出 h_t 和 c_t, 分别记录了我们采取新的动作之后的状态表示信息和记忆信息.

- 动作单元

它使用 LSTM 单元给出的 h_t 作为网络的输入, 首先通过了一个全连接层进行特征变化, 接着再经过一个 softmax 函数得到我们目前的一个状态分布表示:

$$p(s_t)=softmax(FC(h_t)) \tag{1}$$

$p(s_t)$是一个向量, 表示我们在这一次中每一个操作的可能的概率分布. 通过对这个概率分布进行采样,我们就能得到我们下一步具体要执行的操作 a_t. 然而, 这里需要注意的是, 并不是任意一条边都是合法的.比如在图 4 的状态 s_1 中, 我们不能选择边$(T_2.id,T_2.score)$, 因为这会产生一个不连通的子图. 为了防止我们采取不合法的操作, 我们需要使用一个掩码层来屏蔽目前不合法的边.

- 掩码层

掩码层其实就是一个 01 向量 m_t, 它的长度和动作区间大小一样为$|E|$. 其中, 0 表示对应动作当前不可选, 而 1 表示当前动作可选. 这个向量可以根据之前访问过的所有节点进行维护, 保证每个状态的连通和合法性. 我们将这个向量 m_t 与 $p(s_t)$相乘就得到了我们最终的策略函数: $\pi_\theta(s_t)=p(s_t)\times m_t$. 现在我们就能从$\pi_\theta$采样得到下一个动作 $a_t\sim\pi_\theta(s_t)$了.

3.2 策略函数更新

策略函数$\pi_\theta(s_t)$更新的目标是找到一个参数θ, 在整个的生成过程中, 它能够让我们获得最大的期望的奖励值 $J(\theta)$.

$$\max_\theta J(\theta)=\sum_s \mu(s)\sum_a \pi_\theta(a\mid s)\times r(s,a) \tag{2}$$

其中, s 是任意一个可能的达到状态, $\mu(s)$表示根据策略函数π_θ达到状态 s 的概率. 我们发现, 最大化 $J(\theta)$, 其实就是希望调整θ, 让越大的 $r(s,a)$能够对应更高的概率$\pi_\theta(a|s)$.

- 中间状态的奖励值

为了计算 $J(\theta)$, 我们需要知道每个状态 s 的 $r(s,a)$值. 正如我们前面提到的: 我们对于终点状态的 $r(s,a)$, 可以很容易地得到 $r(s,a)=r_q$; 但是对于中间状态 s 的奖励值 $r(s,a)$, 我们是不清楚的. 这里, 我们给出它的定义: 中间状态的 $r(s,a)$为根据π_θ从 s 抵达的最终状态 q 的期望奖励值.

$$r(s,a)=\sum_q \mu(q\mid s')\times r_q=E_q(r_q) \tag{3}$$

s'为 s 采取动作 a 的后续状态, $\mu(q|s')$表示通过策略π_θ生成查询 q 的概率.

$\nabla_\theta J(\theta)$的计算方法: 为了得到使 $J(\theta)$最大的参数θ, 我们这里使用了梯度下降[15]的方法求解 $J(\theta)$的梯度$\nabla_\theta J(\theta)$进行更新. 然而, 计算$\nabla_\theta J(\theta)$需要枚举整个状态和动作空间的奖励值, 而 $r(s,a)$的计算同样也需要遍历后续的所有状态空间. 直接计算是非常困难的, 因此我们参考了 REINFORCE[16]的方法对它进行转化, 得到$\nabla_\theta J(\theta)$的期望表达式:

$$\begin{aligned}\nabla_\theta J(\theta)\propto&=\sum_s \mu(s)\sum_s r(s,a)\times\nabla_\theta\pi_\theta(a\mid s)\\&=E_s\left(\sum_s r(s,a)\times\pi_\theta(a\mid s)\times\frac{\nabla_\theta\pi_\theta(a\mid s)}{\pi_\theta(a\mid s)}\right)\\&=E_{s,a}(r(s,a)\nabla_\theta\log(\pi_\theta(a\mid s)))\\&=E_{s,a}\left(\sum_q \mu(q\mid s')\times r(q)\times\nabla_\theta\log(\pi_\theta(a\mid s))\right)\\&=E_{s,a,q}(r(q)\nabla_\theta\log(\pi_\theta(a\mid s)))\end{aligned} \tag{4}$$

通过上述推导, 我们可以得到$\nabla_\theta J(\theta)$正比于$r(q)\nabla_\theta\log(\pi_\theta(a|s))$的期望, 而$r(q)\nabla_\theta\log(\pi_\theta(a|s))$仅仅与生成过程

中的中间状态(s,a)和最终生成的查询 q 有关. 对于一次确定的生成过程, 它每一步的$\pi_\theta(a_i|s_i)$是确定的. 因此, 我们可以很容易地得到 $r(q)\nabla_\theta\log(\pi_\theta(a_i|s_i))$的具体数值. 一次生成过程产生的 $r(q)\nabla_\theta\log(\pi_\theta(a_i|s_i))$可以被认为是$\nabla_\theta J(\theta)$的一个采样, 因此我们可以直接使用它作为对$\nabla_\theta J(\theta)$的估计, 对$\theta$使用梯度下降法进行更新: $\theta=\theta+\alpha r(q)\nabla_\theta\log(\pi_\theta(a|s))$, 其中, α是学习率参数用于控制更新的速度.

3.3 从小量真实查询学习

对于现实生活中一个特定的数据库场景来讲, 真实的查询往往是全部可能查询的一小部分(比如基于某些特定的模板), 又或者存在着某些交集(比如物化视图). 给生成器关于真实查询负载的信息, 可以让生成的查询更贴近未来真实的查询, 加快学习型优化器的收敛速度. 从第 3.2 节中我们可以发现, 查询生成器参数的更新只依赖于最终的查询 q 的奖励值 $r(q)$和对应的状态动作序列$\{(s_0,a_0),\ldots,(s_e,a_e)\}$. 当给定一个真实而非生成的查询 q 时, 这启发了我们可以通过构造 q 的动作和状态序列$\{(s_0,a_0),\ldots,(s_e,a_e)\}$让生成器$\pi_\theta$学习到查询 q 中对于学习型优化器有价值的信息, 以生成更多更接近真实负载的复杂查询.

当给定一个较小的真实负载 $W=\{q_1,q_2,\ldots,q_k\}$时, 我们让查询生成器π_θ从中学习关于真实负载下对学习型优化器有价值的信息. 对于一条具体的 q_i, 我们构造它的查询图 G^{q_i}. 由于查询图 G^{q_i} 和查询 q_i 是一一对应的, 可以被很容易地构造出来. 为了使用查询 q_i 更新π_θ, 我们还需要求出这张查询子图的动作状态序列$\{(s_0^{q_i},a_0^{q_i}),\ldots,(s_e^{q_i},a_e^{q_i})\}$. 这里, 我们使用了深度优先搜索(depth first search, DFS)的方法, 随机选取一个点作为起点, 对查询图 G^{q_i} 进行搜索, 直到我们找到一条动作状态路径$\{(s_0^{q_i},a_0^{q_i}),\ldots,(s_e^{q_i},a_e^{q_i})\}$能够遍历整张查询图. 最后, 根据第 3.2 节查询生成器π_θ的更新公式, 对θ进行更新:

$$\theta=\theta+\alpha r(q_i)\nabla_\theta\log(\pi_\theta(a^{q_i}\mid s^{q_i})) \tag{5}$$

3.4 与学习型优化器联合的迭代训练过程

生成器π_θ是求解θ最大化生成查询的奖励值. AlphaQO 的最终目标是训练得到一个对比传统优化器 t 更加鲁棒的学习型优化器 l_γ. 因此, 我们可以得到如下的整个系统的目标方程:

$$\min_\gamma\max_\theta G(\theta,\gamma)=E\left(\frac{l_\gamma(q)}{t(q)}\right),\ q\sim\pi_\theta \tag{6}$$

生成器希望调整参数θ, 最大化生成查询对于学习型优化器的难度值(奖励值)$\frac{l_\gamma(q)}{t(q)}$, 而整个系统的目标希望学习参数γ让学习型优化器给出尽量小的难度值. 求解该目标方程, 一般采取的方法就是先生成查询, 再交给优化器训练这样的两步过程. 需要注意的是, 由于学习型优化器具有一定的泛化性, 经过一定的训练之后, 原来困难的查询可能会变得简单. 比如, 学习型优化器本来不能理解 $q_1=A\triangleright\triangleleft B$ 的基数, 但是学习了 $q_2=A\triangleright\triangleleft B\triangleright\triangleleft C$ 的基数之后, 它就可以推断出 q_1 的基数是多少了, 且并不需要再次学习了. 因此, 一次直接全部的查询可能会造成训练资源的浪费. 一个可行的操作是: 我们每次生成一个小批量(比如 25 条)查询, 通过迭代的方式, 根据学习型优化器当次的反馈, 以调整下一次生成查询的方向. 避免优化器在多个同类型的复杂查询下训练, 造成训练时间的浪费.

现在我们可以总结 AlphaQO 的整个迭代训练过程.

(1) 根据查询生成器的策略函数, 生成一个小批量的查询集合$\{q_0,q_1,\ldots,q_k\}$.

(2) 获取目前生成的查询$\{q_0,q_1,\ldots,q_k\}$的奖励值$\{r(q_0),r(q_1),\ldots,r(q_k)\}$, 并将查询交给学习型优化器进行一定时间的训练.

(3) 使用获得的奖励值对查询生成器进行更新.

(4) 回到步骤(1), 继续生成查询.

4 实 验

实验部分的主要目标是验证两个问题: (1) AlphaQO 的查询生成器能否根据给定的查询全图和谓词集合生

成对学习型优化器更难的查询? (2) 学习型优化器经过 AlphaQO 的迭代训练之后是否可以得到提升, 做好未来的查询, 提高系统的鲁棒性.

4.1 实验设置

- 实验环境

我们使用了一台服务器, CPU 为 Intel(R) Xeon(R) CPUE5-2630 v4, 内存为 128 GB 和一张 2 080 ti 的显卡. 使用的实验数据库为 PostgreSQL-12.4. 我们选择了 RTOS[11]为 AlphaQO 后端的学习型优化器.

- 数据集

我们使用了一个真实的数据库 IMDB. 它包含了 3.6 GB 的原始数据, 导入数据库并构建索引之后大小为 11 GB, 该数据库有 21 张表格. 这个数据库相较于一般的 OLAP 数据集(如 TPC-H)更复杂, 因为它的负载往往涉及到更多的表. 同时, 其表的数据存在着相关性, 并且分布不是均匀的. 这些都导致了传统数据库的优化器很难优化好相关的查询. IMDB 上的查询一般使用主外键进行连接. 为了对比优化器的性能, 我们这里选取了该数据库上两个典型的负载进行测试.

(1) JOB[17]: 现实中标准的 IMDB 的查询. 这个负载包含了 113 条查询, 这些查询由 33 个模板生成而来. 它们包含了复杂的谓词情况, 比如字符串查询. 同时, 这些查询的长度(涉及的表的数量)在 4–17 张表之间.

(2) EXT[10]: 一个基于 JOB 数据集针对学习型优化器提出的拓展负载, 它包含了 12 个模板和产生的 24 条具体的查询. 它的查询长度在 3–11 之间, 更加注重测试在一般场景下学习型优化器的泛化表现.

我们预先根据 IMDB 数据集的负载场景(比如 PK-FK 连接)生成了查询完全图 G, 并采样了谓词集合 P, 作为 AlphaQO 的输入. 我们将数据集中的全部查询作为测试集, 来检测 AlphaQO 训练后的学习型优化器 RTOS 的性能. 我们同时也采样了 10%的查询作为一个小的真实负载交给查询生成器, 并将剩下的 90%作为测试集, 来检测我们的生成器是否能够在得到少量负载信息后生成更有效的查询.

- 基准方法

为了测试我们整个 AlphaQO 系统的有效性, 我们分别对比了 AlphaQO 的生成器和学习型优化器的性能.

对于优化器, 我们使用 PostgreSQL 优化器内置的动态规划算法(DP)、ReJOIN[10]、DQ[9]、QuickPick-1000 (QP1000)[6]为基线算法. 动态规划算法枚举所有计划, 并选择估计代价最低的那个代价. 我们设置让 PostgreSQL 内置的优化器选择动态规划算法.

对比生成器, 我们选择随机算法(RANDOM)作为我们的比较方法, RANDOM 方法和 AlphaQO 都会使用同一个查询全图 G 和谓词集合 P 进行查询生成. 与 AlphaQO 使用强化学习选择下一步的动作不同, RANDOM 算法每一步都会随机地产生下一步的动作.

- 运行时间收集

AlphaQO 在收集运行时间时, 我们需要执行查询, 这可能会导致训练的速度过慢. 生成器会尝试生成复杂的查询; 同时, 学习型优化器 RTOS 也会尝试探索不同的计划, 这些都会导致某些查询的执行时间过长. 因此, 我们设置了一个阈值(比如 5 分钟)[8], 将超过这段时间的执行计划停止并标记, 避免太差的计划导致系统训练的时间过长. 具体地, 对于传统优化器超过阈值的查询, 我们会提供给生成器一个–1 的奖励值.

- 神经网络的设定

我们基于 PyTorch 完成了 AlphaQO 系统, 我们设定生成器和学习型优化器隐藏层的大小 hs=128. 这是一个机器学习中很常用的大小, 可以在神经网络的效果和运行时间之间取得一个平衡. 我们使用学习率 lr 为 0.001 的 SGD[18]优化器作为我们的神经网络优化器, 来训练我们的查询生成器和 RTOS. 使用 ReLU[19]作为我们的激活函数. AlphaQO 生成器的每一轮迭代中, RTOS 的强化学习部分会进行 5 个回合的训练.

4.2 训练强化学习生成器

查询生成器指引了 AlphaQO 的查询生成过程, 我们首先评估查询生成器是否能够生成学习型优化器做得

不好的查询. 我们停止 AlphaQO 学习型优化器的训练模块, 单纯测试生成器生成的查询质量随着时间变化的关系. 因为运行时间阈值的存在, 以时间为指标不容易观测这些困难查询的生成情况. 针对 RTOS 和 DP 给出的查询计划, 我们选用数据库的代价模型中估计的代价为这里生成器学习的目标, 即将 $r_q=\frac{l_q}{t_q}$ 中的 l_q 和 t_q 都是用估计器给出的代价代替真实运行时间. 具体来说, 我们使用对数下的相对代价(也就是奖励值 reward)的中值(mean log relative cost, MLRC)作为生成器的指标:

$$MLRC=median\left(\log_2\left(\frac{cost_{RTOS}(q_i)}{cost_{DP}(q_i)}\right)\right) \tag{7}$$

图 6 展示了每个小批次生成的查询的奖励值的中值变化(在 log 的维度下), 并与 RANDOM 算法做了对比. RL 代表了使用了强化学习的 AlphaQO, RD 则是随机化算法. 我们在 40 个迭代的过程中一共生成了 1 000 条查询. 我们可以发现:

- RANDOM 算法和 AlphaQO 算法生成的查询的奖励值在一开始是相似的.
- 但是随着生成过程的推进, RANDOM 生成查询的 MLRC 是没有什么变化, 在 3.59 左右波动; 而图 6 却显示 AlphaQO 在学习了每个小批次的奖励值之后, 会不断地产生对于 RTOS 更难的查询.
- 大概经历了 50 个查询(第 2 个批次)的训练之后, AlphaQO 就能生成更难的查询; 且随着时间的增大, 生成的查询的难度值指数增加.
- 生成 1 000 个之后, AlphaQO 生成的查询的难度中位数更是达到了 10 000 倍(*MLRC*>13).

实验说明, AlphaQO 能够有效地发现学习优化器做得不好的查询.

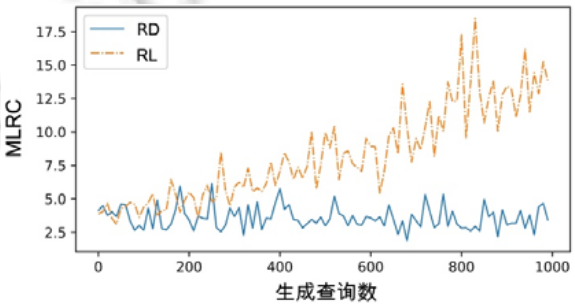


图 6 AlphaQO 和 RANDOM 算法生成查询的奖励值中值的变化情况

4.3 生成器对查询优化器的影响

AlphaQO 的生成器的主要目的是生成训练查询, 提升学习型优化器的性能. 我们这里对于通过实验观察在 AlphaQO 迭代的过程中, RTOS 每次生成的计划的执行时间对比 DP 生成的计划的执行时间的变化, 探讨 AlphaQO 整体的效果. 这里, 我们使用了两个指标来对比 DP 和 RTOS 的性能.

(1) 归一化延迟(normalized latency, NL)[10]: NL 主要关注了 RTOS 对比 DP 在不同负载上的总的运行时间的比值:

$$NL=\frac{\sum_{i=1}^{n}Latency_{RTOS}(q_i)}{\sum_{i=1}^{n}Latency_{DP}(q_i)} \tag{8}$$

(2) 相对延迟几何平均(geometric mean relative latency, GMRL)[11]: GMRL 先算出每个查询 RTOS 和 DP 的相对运行时间, 再做几何平均:

$$GMRL=\left(\prod_{i=1}^{n}\frac{Lantency_{RTOS}(q_i)}{Lantency_{DP}(q_i)}\right)^{\frac{1}{n}} \tag{9}$$

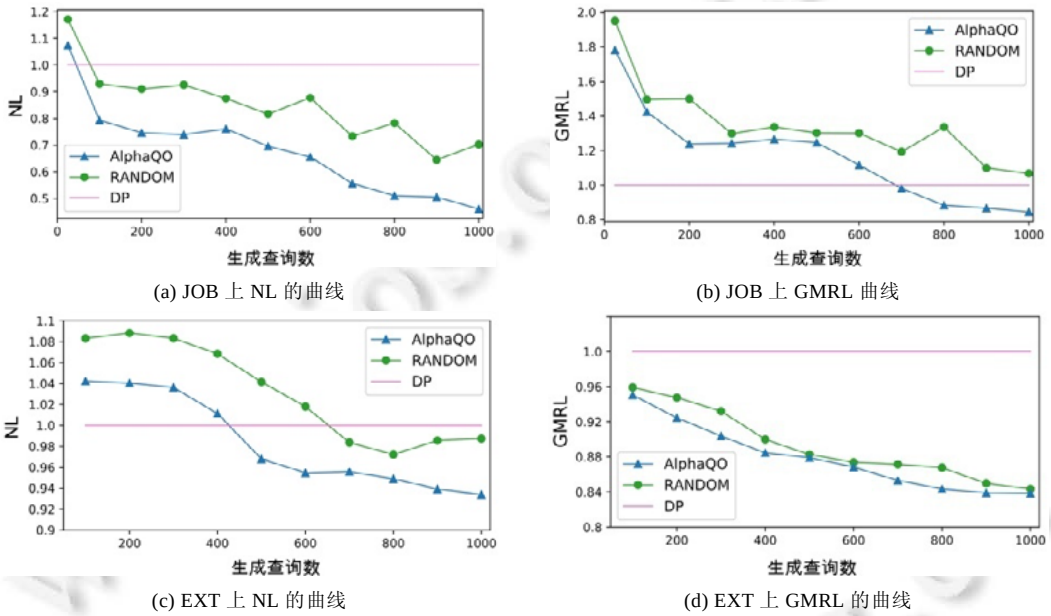
对比 NL 和 GMRL 我们可以发现: NL 指标注重于总的运行时间, 更偏向于复杂查询(运行时间长的查询)

的优化效果; GMRL 指标则不区分复杂和简单的查询而直接先对时间进行归一化处理, 考虑整体的计划的胜负情况. 一般来讲, 我们认为: 值为 1 代表在该指标下, RTOS 取得了和 DP 一样好的结果; 小于 1, 则表示 RTOS 优于 DP.

接下来我们主要从两个方面对 AlphaQO 进行讨论: (1) RTOS 给出计划的质量随着生成查询的数量的改变; (2) 当生成足够多(1 000)条查询时, RTOS 的性能表现.

4.3.1 AlphaQO 对比 RANDOM 算法训练 RTOS 的结果

通过控制生成的查询数量, 我们对比了 RTOS 在经过 AlphaQO 生成的查询和 RANDOM 生成的查询训练之后的效果表现. 图 7 分别以 NL 和 GMRL 指标展示了 RTOS 的在两个数据集上的性能对比. 我们可以发现, 随着生成的训练数据增多, 不管是 RANDOM 还是 AlphaQO 都可以取得更好的效果. 这说明了学习型优化器的效果会随着生成查询数量的增加而提高.



(a) JOB 上 NL 的曲线 (b) JOB 上 GMRL 曲线

(c) EXT 上 NL 的曲线 (d) EXT 上 GMRL 的曲线

图 7 RTOS 随着生成查询数量增多的性能曲线

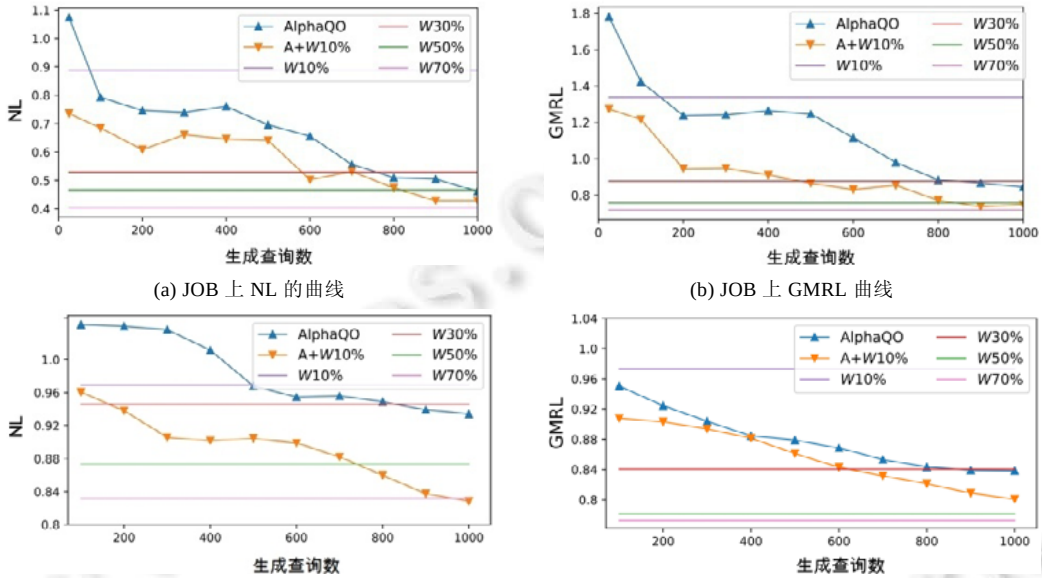
我们首先分析 JOB 数据集上的结果. 我们先看 NL 指标下, AlphaQO 和 RANDOM 算法的表现. 两种算法在最开始都取得了接近的效果, 并且在产生差不多 100 条查询之后, 都能使学习型优化器超过了 DP 的表现. 但是 RANDOM 算法在产生了大概 500 条查询之后, 才能让 RTOS 取得与 AlphaQO 算法类似的效果. 在 GMRL 指标下, AlphaQO 在生成 700 条查询之后, 才让 RTOS 超过了 DP 算法, 而 RANDOM 算法即使 1 000 条查询也未能让 RTOS 达到 DP 效果. 这体现了 AlphaQO 能够产生更好的查询, 让 RTOS 能够更快地学习到有效的知识. 同时, 我们可以很容易地从图片中发现: 一开始, RANDOM 算法和 AlphaQO 算法都能让 RTOS 的指标(NL, GMRL)快速下降; 但是 RANDOM 在产生查询足够大时, 降低的速度明显下降. 这也体现了 AlphaQO 在任何时期都能发现更有效的查询. 值得一提的是, NL 和 GMRL 上对比 DP 的效果, 体现了 JOB 是一个具有挑战性的数据集, 且其中存在了许多传统优化器不能做好的复杂(长)查询. 这解释了为何在 NL 指标下, RTOS 能够使用较少训练查询就超过 DP 算法.

接着我们分析 EXT 数据集. 在 NL 指标下, RANDOM 需要生成 700 条查询, 使得 RTOS 取得和 DP 近似的效果, 而 AlphaQO 只需要接近 400 条查询. 同样地, RANDOM 对 RTOS 效果提升的曲线明显比 AlphaQO 要慢很多. 这都体现了 AlphaQO 算法对比 RANDOM 算法的优势, 能够大大提升学习型优化器训练的效率. 值得一提的是, EXT 数据集上, RTOS 的 GMRL 表现明显好于 NL 的表现. 这说明了 EXT 数据集包含了一些学习

型优化器会产生较差方案的查询, 从而需要产生更多的训练查询.

4.3.2 加入少量真实查询之后 AlphaQO 训练 RTOS 的结果

在第 4.3.1 节中, 我们已经验证了 AlphaQO 能够产生比 RANDOM 质量高的查询, 并且能够帮助 RTOS 在一段时间的训练后, 取得和数据库内置的动态规划算法接近的结果. 我们下面探讨加入少量真实查询的情况下, 生成器是否能够让 RTOS 取得与使用较多真实负载训练取得接近的效果. 这里我们准备了 4 个真实负载采样 *W*10%, *W*30%, *W*50%, *W*70%, 分别表示抽取了 10%, 30%, 50%, 70%的真实查询作为 RTOS 的训练数据. *A*+*W*10%表示我们在生成查询时, 还结合了已有的 10%的真实负载协同训练我们的生成器. 图 8 展示了我们在不同大小的真实负载、使用结合小量真实负载的 AlphaQO 和只使用查询完全图信息的 AlphaQO 的效果区别(*W x*%表示我们使用了 *x*%的真实负载训练 RTOS).



(a) JOB 上 NL 的曲线 (b) JOB 上 GMRL 曲线

(c) EXT 上 NL 的曲线 (d) EXT 上 GMRL 的曲线

图 8 RTOS 随着生成查询数量增多的性能曲线

我们首先观察 JOB 这一数据集. NL 指标下, *W*10%的值是 0.89 左右, 已经好于了 DP 的表现. 这体现了学习型优化器对比传统数据库的优势. 对比 *A*+*W*10%和 AlphaQO 的表现, 我们发现, *A*+*W*10%可以在额外生成 50 个查询就获得明显好于 DP 和 *W*10%的效果, 而 AlphaQO 需要更多的轮次(300 个查询)才能取得和 *A*+*W*10%近似的效果. 生成 1 000 条查询之后, *A*+*W*10%和 AlphaQO 都超过了 50%的真实负载训练效果, *A*+*W*10%甚至接近了 *W*70%. 在 GMRL 指标上, *A*+*W*10%在生成接近 200 个查询之后就好于了 DP 的表现, 训练速度明显好于 AlphaQO. 同样在 1 000 条查询生成之后, AlphaQO 仅仅只能超过 *W*30%的效果, 而 *A*+*W*10%接近了 *W*70%的效果. *A*+*W*10%的收敛速度和收敛结果都明显好于 AlphaQO. 值得一提的是, 虽然 *A*+*W*10%好于 AlphaQO, 但是随着生成查询数的增多, 两者的差距在越来越小.

我们接着观察 EXT 这一数据集. 与 JOB 数据集的表现类似, *A*+*W*10%比 AlphaQO 收敛得更快、更好. 由于 *W*10%的影响, *A*+*W*10%在 50 个查询训练后, 在 NL 和 GMRL 指标上就比 DP 更好. 值得注意的是, 即使在 EXT 数据集上, *A*+*W*10%在 1 000 条查询生成后, 在 NL 指标上甚至接近了 *W*70%的效果.

对比上述的几组结果, 我们发现, 在引入少量的真实负载之后, 我们的生成器能够更高效地生成更高质量的查询. 但是两者的差距会随着生成数量的变多越来越小, 体现了 AlphaQO 自身的有效性.

4.3.3 学习型优化器 RTOS 在不同生成器设置的性能分析

在本节中, 我们具体探索 RTOS 在不同设置下究竟学习到了什么信息, 哪些查询 RTOS 可以很容易学习

好. 我们在不同的设置下生成了 1 000 条训练查询, 交给学习型优化器 RTOS 进行训练, 将 JOB 和 EXT 负载下的查询作为测试集评估它们的性能表现. 我们按照测试集中查询的长度作为分组的标准, 绘制了不同设置下总执行时间的对比情况. 图 9 和图 10 分别展示了 JOB 和 EXT 数据集不同长度的查询总运行时间(测试集查询按照长度进行分组).

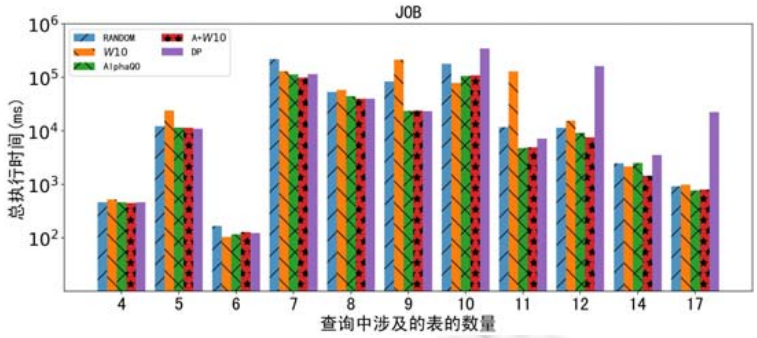


图 9 经过 1 000 条生成查询训练后的 RTOS 的性能表现

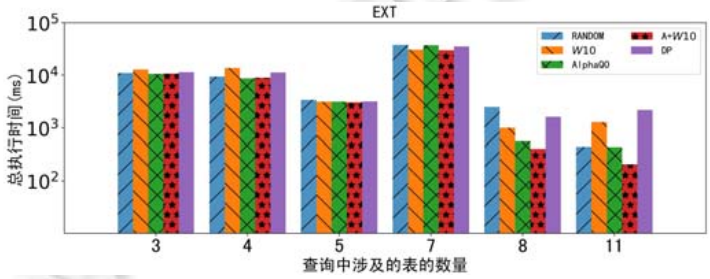


图 10 经过 1 000 条生成查询训练后的 RTOS 的性能表现

我们首先观察 JOB 这个数据集上的表现. JOB 数据集中的查询长度在 4–17 之间. JOB 针对传统优化器因为难以估计过长查询的基数这一特性进行了查询的设计. 我们发现, 几种生成的方法都能让 RTOS 在长度超过 10 的查询中, 轻易地超过动态规划的表现, 甚至随机地生成 1 000 条查询去训练, 都能几乎超过了 DP 的效果. 大小为 10 的组是 JOB 数据集中最难的组, DP 产生的计划需要 346 s 才能执行完该组的查询. 而基于 AlphaQO 方法的 RTOS 在生成 1 000 条训练查询之后, 仅需要 105 s 就可以跑完该组的查询, 节约了 241 s (69.6%)的时间. 这也解释了为何图 7 中 JOB 数据集的 NL 指标能够很快地好于 DP. *W*10%也能够让 RTOS 在第 10 组的查询中好于 DP 算法, 但是由于仅仅 10%的查询难以让 RTOS 泛化到其他的未见查询, 导致在某些组中没有取得较好的结果. 经过 1 000 条生成查询的训练之后, AlphaQO 和 *A*+*W*10%下的 RTOS 取得了接近的结果. *A*+*W*10%由于得到了更多的信息, 会稍好于 AlphaQO. 这两种算法都能够让 RTOS 在第 10 组–第 12 组、第 14 组、第 17 组明显地好于动态规划算法. 对于较短的查询 RTOS 的结果, 和 DP 比较接近. 这也表现了传统的数据库优化器能够较好地解决短查询的连接顺序选择的问题.

我们接着观察 EXT 数据集中, 不同生成器设定下 RTOS 的表现. EXT 的查询长度范围是 3–11, 明显比 JOB 数据集中的负载要短. 和 JOB 数据集中结果相似的是, AlphaQO 和 *A*+*W*10%都明显地在长的查询上(第 8 组和第 11 组)让 RTOS 表现得比 DP 好. 对于短的查询, AlphaQO 和 *A*+*W*10%使得 RTOS 对比 DP 提升不大. EXT 的数据集主要是用来考验学习型优化器, 并且包含了一些传统优化器本身就能做得好的短查询. 所以也解释了经过 AlphaQO 和 *A*+*W*10%训练的 RTOS 在 EXT 上并未获得像 JOB 一样显著的提升.

4.3.4 估计代价和真实时间作为奖励值的对比

在这一节中, 我们对比使用估计代价和真实时间分别作为生成器和优化器的奖励值参数. 使用估计代价为指标时, 我们不执行查询, 直接使用数据库估计器给出的代价代替执行时间作为计划的执行反馈. 使用相

对代价的均值 MRC[9]作为衡量优化器计划质量的指标. 生成器生成 1 000 条训练查询.

表1给出了使用估计代价作为奖励值参数的实验结果. 我们可以发现, 不出意外地, 在求解计划时间允许的情况下, DP 在估计值上取得了最好的效果. 使用结合小量(10%)负载的 AlphaQO 训练的 RTOS 弱于使用了大量(70%负载训练)的 RTOS, 但是好于 QP1000 和其他两种强化学习方法 ReJOIN 及 DQ. 这也体现了生成方法的有效性.

表 1 使用估计代价作为生成器和优化器奖励值参数, 越低越好

指标	*A*+*W*10	*W*50	*W*70	QP1000	ReJOIN	DQ	DP
MRC	1.09	1.06	1.01	1.62	1.75	2.34	1

表 2 给出了使用真实运行时间作为奖励值参数的实验结果. 在使用真实运行时间作为奖励值参数的情况下, DP 没能取得最好的结果. 这说明了估计代价和真实运行时间之间存在着差距, 代价最低的方案不一定是执行时间最小的方案. 而基于学习的优化器 RTOS、DQ、ReJOIN 性能都有所提升. 同时, 结合小量(10%)负载 AlphaQO 训练的 RTOS 在这个部分超过了 DP 的性能, 仅稍弱于大量真实负载(70%)的 RTOS 的效果, 好过了使用 50%真实负载训练的 RTOS, 说明了学习真实运行时间的优点, 也体现了 AlphaQO 生成查询的有效性.

表 2 使用运行时间作为生成器和优化器奖励值参数, 越低越好

指标	*A*+*W*10	*W*50	*W*70	QP1000	ReJOIN	DQ	DP
GMRL	0.71	0.73	0.68	1.90	1.14	1.23	1

5 相关工作

5.1 基于查询的学习型优化算法

在优化器领域中, 许多学习型的方法被提出, 比如基于强化学习的优化器[8−11,20]、基于学习的基数和代价估计算法[21,22]. 可是, 这些方法通常是基于查询的. 具体来说, 基于强化学习的优化器要求用户预先给定一批查询负载, 而基数和代价估计算法则需要给定一些真实的执行计划. 对于给定的训练数据, 这些方法能够获得很好的效果. 但是基数和代价估计往往需要成千上万的训练计划, 获得这些执行计划比较困难. 基于强化学习的优化器算法通常需要与真实查询接近的负载以达到较好的效果. 基于随机生成查询的强化学习优化器无法获得较好的效果[9], 而且过高的训练代价限制了能够生成出的查询数量. 如何给这些基于查询的方法提供查询是非常关键的问题, AlphaQO 的目标是解决这个问题.

5.2 对抗训练

在这个工作中, 生成器的角色可以被看作是一个攻击者, 它的目标是寻找利用学习型优化器无法获得较高性能的查询. 工作机理类似于深度学习领域中的对抗攻击[23,24]问题. 对抗攻击尝试寻找深度学习模型的固有弱点, 目标是提升整体的性能. 我们的工作可以被看作是一个黑盒对抗攻击模型[25,26]. 黑盒对抗攻击将深度学习模型看作是一个黑盒子, 它能够产生样本来测试目标学习模型, 并且基于反馈, 生成一些目标模型预测不好的样本, 如此迭代尝试, 最终获得更加健壮的目标模型. 黑盒对抗攻击方法希望这种尝试尽可能地少. 黑盒对抗攻击的过程和 AlphaQO 的目标是很类似的. 目前, 两种生成对抗网络(GAN)[27,28]解决了与 AlphaQO 类似的问题. 不同的是, GAN 需要生成和真实图像类似的图像, 而 AlphaQO 需要生成能够改进优化器性能的查询.

5.3 数据生成

数据生成是数据库领域中一个十分重要的问题, 通过给定相关信息来生成更多的目标数据. 一个重要的用处是保护用户隐私. 在结构化数据库中, 通过生成对抗网络(GAN)对原始数据进行训练, 学习它们的分布等特征[29−31], 然后使用训练好的网络生成数据, 我们可以保持原始数据的一些性质进行后续分析, 同时又避免了原始数据的泄露. 另外一个方向是查询负载的生成, 它们的目标是针对给定的数据库进行正确性和性能

的测试. QGen[32]使用随机组合 SQL 查询的方式, 在短时间内生成大量的合法测试查询. Qrelx[33]使用基于相似度的算法进行搜索, 生成满足用户给定基数条件的查询. ADUSA[34]通过同时生成数据、查询和查询的结果以测试数据库. 而我们的方法 AlphaQO 希望通过强化学习的方式找到当前学习型优化器还不能优化好的查询, 提前交给学习型优化器训练, 以提升学习型优化器的鲁棒性.

6 结论和未来工作

在这篇文章中, 我们解决了一个针对学习型优化器鲁棒性的提升这一新问题. 我们提出了一个 AlphaQO 框架. 该框架使用强化学习生成查询就能更加有效地改进给定学习型优化器的性能. 我们通过实验验证了方法的有效性, 也会尝试在未来给出进一步的理论约束.

未来我们将尝试更多的学习型优化器, 比如 DQ、NEO 和 ReJOIN. 此外, 我们还计划尝试不同的数据库系统, 并且加入更加复杂的查询.

References:

[1] Selinger PG, Astrahan MM, Chamberlin DD, Lorie RA, Price TG. Access path selection in a relational database management system. In: Proc. of the Readings in Artificial Intelligence and Databases. Morgan Kaufmann, 1989. 511−522.

[2] Babcock B, Chaudhuri S. Towards a robust query optimizer: A principled and practical approach. In: Proc. of the 2005 ACM SIGMOD Int'l Conf. on Management of Data. 2005. 119−130.

[3] Trummer I, Koch C. Solving the join ordering problem via mixed integer linear programming. In: Proc. of the 2017 ACM Int'l Conf. on Management of Data. 2017. 1025−1040.

[4] Ioannidis YE, Kang YC. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In: Proc. of the '91 ACM SIGMOD Int'l Conf. on Management of Data. 1991. 168−177.

[5] Chande SV, Sinha M. Genetic optimization for the join ordering problem of database queries. In: Proc. of the 2011 Annual IEEE India Conf. IEEE, 2011. 1−5.

[6] Waas F, Pellenkoft A. Join order selection (good enough is easy). In: Proc. of the British National Conf. on Databases. Berlin, Heidelberg: Springer, 2000. 51−67.

[7] Fegaras L. A new heuristic for optimizing large queries. In: Proc. of the Int'l Conf. on Database and Expert Systems Applications. Berlin, Heidelberg: Springer, 1998. 726−735.

[8] Marcus R, Negi P, Mao HZ, Zhang C, Alizadeh M, Kraska T, Papaemmanouil O, Tatbul N. Neo: A learned query optimizer. arXiv preprint arXiv:1904.03711, 2019.

[9] Krishnan S, Yang ZH, Goldberg K, Hellerstein J, Stoica I. Learning to optimize join queries with deep reinforcement learning. arXiv preprint arXiv:1808.03196, 2018.

[10] Marcus R, Papaemmanouil O. Deep reinforcement learning for join order enumeration. In: Proc. of the 1st Int'l Workshop on Exploiting Artificial Intelligence Techniques for Data Management. 2018. 1−4.

[11] Yu X, Li GL, Chai CL, Tang N. Reinforcement learning with tree-LSTM for join order selection. In: Proc. of the 2020 IEEE 36th Int'l Conf. on Data Engineering (ICDE). IEEE, 2020. 1297−1308.

[12] Chaudhuri S. An overview of query optimization in relational systems. In: Proc. of the 17th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems. 1998. 34−43.

[13] Tan WC, Zhang MH, Elmeleegy H, Srivastava D. Reverse engineering aggregation queries. Proc. of the VLDB Endowment, 2017, 10(11): 1394−1405.

[14] Hochreiter S, Schmidhuber J. Long short-term memory. Neural Computation, 1997, 9(8): 1735−1780.

[15] Sutton RS, McAllester D, Singh S, Mansour Y. Policy gradient methods for reinforcement learning with function approximation. In: Proc. of the NIPs, Vol.99. 1999.

[16] Williams RJ. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning, 1992, 8(3-4): 229−256.

[17] Leis V, Gubichev A, Mirchev A, Boncz P, Kemper A, Neumann T. How good are query optimizers, really? Proc. of the VLDB Endowment, 2015, 9(3): 204−215.

[18] Amari S. Backpropagation and stochastic gradient descent method. Neurocomputing, 1993, 5(4-5): 185−196.

[19] Nair V, Hinton GE. Rectified linear units improve restricted Boltzmann machines. In: Proc. of the ICML. 2010.

[20] Trummer I, Wang JX, Wei ZY, Maram D, Moseley S, Jo S, Antonakakis J, Rayabhari A. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In: Proc. of the 2019 Int'l Conf. on Management of Data. 2019. 1−45.

[21] Kipf A, Kipf T, Radke B, Leis V, Boncz P, Kemper A. Learned cardinalities: Estimating correlated joins with deep learning. arXiv preprint arXiv:1809.00677, 2018.

[22] Marcus R, Papaemmanouil O. Plan-structured deep neural network models for query performance prediction. arXiv preprint arXiv:1902.00132, 2019.

[23] Madry A, Makelov A, Schmidt L, Tsipras D, Vladu A. Towards deep learning models resistant to adversarial attacks. arXiv preprint arXiv:1706.06083, 2017.

[24] Chakraborty A, Alam M, Dey V, Chattopadhyay A, Mukhopadhyay D. Adversarial attacks and defences: A survey. arXiv preprint arXiv:1810.00069, 2018.

[25] Ilyas A, Engstrom L, Athalye A, Lin J. Black-box adversarial attacks with limited queries and information. In: Proc. of the Int'l Conf. on Machine Learning. PMLR, 2018. 2137−2146.

[26] Guo C, Gardner J, You YR, Wilson AG, Weinberger K. Simple black-box adversarial attacks. In: Proc. of the Int'l Conf. on Machine Learning. PMLR, 2019. 2484−2493.

[27] Bojchevski A, Shchur O, Zügner D, Günnemann S. Netgan: Generating graphs via random walks. In: Proc. of the Int'l Conf. on Machine Learning. PMLR, 2018. 610−619.

[28] Yu LT, Zhang WN, Wang J, Yu Y. Seqgan: Sequence generative adversarial nets with policy gradient. In: Proc. of the AAAI Conf. on Artificial Intelligence. 2017.

[29] Fan J, Liu TY, Li GL, Chen JY, Shen YW, Du XY. Relational data synthesis using generative adversarial networks: A design space exploration. arXiv preprint arXiv:2008. 12763, 2020.

[30] Xu L, Veeramachaneni K. Synthesizing tabular data using generative adversarial networks. arXiv preprint arXiv:1811.11264, 2018.

[31] Park N, Mohammadi M, Gorde K, Jajodia S, Park H, Kim Y. Data synthesis based on generative adversarial networks. arXiv preprint arXiv:1806.03384, 2018.

[32] Poess M, Jr. Stephens JM. Generating thousand benchmark queries in seconds. In: Proc. of the 30th Int'l Conf. on Very Large Data Bases, Vol.30. 2004. 1045−1053.

[33] Vartak M, Raghavan V, Rundensteiner EA. Qrelx: Generating meaningful queries that provide cardinality assurance. In: Proc. of the 2010 ACM SIGMOD Int'l Conf. on Management of Data. 2010. 1215−1218.

[34] Khalek SA, Elkarablieh B, Laleye YO, Khurshid S. Query-aware test generation using a relational constraint solver. In: Proc. of the 2008 23rd IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE, 2008. 238−247.

余翔(1994－), 男, 博士生, 主要研究领域为人工智能驱动的数据库优化技术.

汤南(1981－), 男, 博士, 研究员, 主要研究领域为数据准备, 数据可视化.

柴成亮(1992－), 男, 博士, CCF 专业会员, 主要研究领域为人机协作的数据管理, 数据库系统.

孙佶(1994－), 男, 博士, 主要研究领域为分布式相似查询, 人工智能和数据库交叉技术.

张辛宁(2000－), 男, 本科生, 主要研究领域为人工智能驱动的数据库优化技术.

李国良(1981－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为大数据, 数据库, 数据科学.