UDO: Universal Database Optimization using Reinforcement Learning

- UDO：使用强化学习的通用数据库优化
- ABSTRACT
  - UDO 是一种多功能工具，用于针对特定工作负载对数据库系统进行离线调整。
  - **UDO 可以考虑多种调整选择，从选择事务代码变量到索引选择，再到数据库系统参数调整。**
  - **UDO 使用强化学习来收敛到接近最优的配置，通过实际查询执行创建和评估不同的配置（而不是依赖于简化成本模型）。**
  - 为了迎合不同的参数类型，**UDO 将重参数（更改成本高昂，例如物理设计参数）与轻参数区分开来。UDO使用强化学习算法专门用于优化重参数，允许延迟奖励反馈可用的时间点。**
  - 这使我们可以自由地优化创建和评估不同配置的时间点和顺序（通过对工作负载样本进行基准测试）。
  - **UDO 使用基于成本的规划器来最小化重新配置开销。**
  - 例如，它旨在通过连续评估使用它们的配置来分摊创建昂贵的数据结构。
  - 我们在 Postgres 和 MySQL 以及 TPC-H 和 TPC-C 上评估 UDO，同时优化各种轻量级和重级参数。
- INTRODUCTION
  - 我们提出了 UDO，即通用数据库优化器。**UDO 是一种离线调优工具，可优化各种调优选择（例如，物理设计决策以及数据库系统配置参数的设置），给定示例工作负载和调优时间限制。**
    - **UDO 不依赖于简化成本模型来评估调整选项的质量。**
    - **此外，它不需要任何类型的预先训练数据。**
    - 相反，**它仅依赖于在创建要评估的调整配置之后通过示例运行获得的反馈。**
    - **这使得优化过程成本高昂，但避免了由于错误的成本估计而导致的次优选择，否则这种情况很常见。**
  - 鉴于 UDO 实现的权衡（即高质量、高开销优化），我们看两个主要用例。
    - 首先，**UDO 在通过昂贵的优化获得的配置可以长期使用的情况下很有用。**
      - 如果数据和查询工作负载属性变化不是太频繁，这是可能的。
    - 此外，**UDO 还可用作其他调优方法的分析工具。**
      - 例如，**由于 UDO 不依赖于成本或基数模型，因此可用于发现基于后者的其他推荐工具的弱点。**
      - 在这个场景中，UDO采用了与之前提出的查询优化器测试方法类似的角色，**这些方法通过昂贵的过程生成有保证的最优计划（但特定于查询计划，而不是其他调优选择）。**

- UDO 对各种类型的调整参数进行操作，这些参数通常由单独的调整工具处理。
  - 例如，在我们的实验中，我们考虑了交易查询顺序、索引选择以及数据库系统配置参数的优化。
  - 将各种参数类型一起考虑可能是有利的，因为一种参数类型的最佳选择可能取决于其他参数的设置（例如，我们可能会禁用顺序扫描、配置参数，仅当创建特定索引时）。
  - 因此，我们将通用术语参数用于每个调整选择，术语配置用于从参数到值的分配。
  - UDO 通过统一的方法处理所有参数。
- **UDO 迭代地探索搜索空间：选择要尝试的配置，创建它们（例如，创建索引结构或设置由配置指定的系统参数），并评估它们在工作负载样本上的性能。**
  - 评估可以灵活地合并多个指标，例如吞吐量或延迟。
  - 我们展示了使用不同数据库系统（Postgres 和 MySQL）和标准基准（TPC-C 和 TPC-H）上的两个指标的优化。
  - UDO 使用强化学习 (RL) 来确定接下来要尝试的配置。
  - 性能测量的改进转化为奖励值，这些值在搜索过程中指导 RL 代理进行操作，即配置，最大化累积奖励，即产生的性能。
- **RL 以前特别用于优化数据库系统配置参数。**
  - **UDO 的主要新颖之处在于它将优化范围扩大到更大的参数类别。**
  - 由于我们称之为重参数（我们在下面将它们与轻参数区分开来），这变得特别具有挑战性。
  - **对于繁重的参数，改变参数值的代价是昂贵的。**
  - **例如，与索引创建相关的参数更改起来成本很高。**
  - **创建索引，尤其是聚簇索引，可能会花费大量时间，这在小工作负载样本中占主导地位的查询或事务评估时间。**
  - **同样，需要重新启动数据库服务器的配置参数的更改成本也相对较高。**
  - 正如我们在实验中所展示的，**朴素的 RL 方法受到改变重参数的成本的限制。**
  - **这会导致每次迭代的高成本并减慢收敛速度。**
- **UDO 通过对重参数进行特殊处理来避免这种陷阱。**
  - **UDO 将重参数与轻参数分开，并使用不同的强化学习算法对其进行优化。**
  - **特别是对于重参数，它使用 RL 算法，该算法可以延迟调整，直到先前选择的奖励值可用。**
  - 我们利用以下延迟反馈。
    - RL 算法选择的所有配置都被转发到规划组件。
    - 规划组件决定创建和评估配置的时间和顺序。
    - 根据这些选择，我们能够通过**对许多类似配置的评估来分摊更改重参数的成本。**
  - 例如，**它允许我们创建一个昂贵的索引一次来评估所有包含该索引的多个相似配置。**

    - 另一种方法是在使用或不使用需要多个索引创建和删除的索引的配置之间交替，效率较低。
- **对于当前重参数的设置，我们再次使用 RL 来寻找轻参数的最佳设置。**
    - 当然，**轻参数的最佳设置取决于重参数的值。**
    - **UDO 考虑到这一点，并将每个重参数设置的轻参数优化建模为一个单独的马尔可夫决策过程 (MDP)，对其应用 RL 算法。**
    - **与重参数相比，我们使用无延迟 RL 算法更快地收敛到轻参数的接近最优设置。**
- 我们提出了一种新的蒙特卡罗树搜索 (MCTS) 变体，称为延迟分层乐观优化 (HOO)，可用于优化重参数和轻参数（有延迟和无延迟）。
    - 我们表明，在使用该方法时，如果有足够的优化时间，UDO 会收敛到接近最优的配置。
- 我们通过实验证明，在给定相同的优化时间的情况下，与参照物相比，生成的系统找到了更好的配置。
    - 我们考虑了多个标准基准（TPC-H 和 TPC-C）、多个优化指标（吞吐量和延迟）以及不同的数据库管理系统（Postgres 和 MySQL）。 总之，我们的原始科学贡献如下。
        - 我们介绍了**一种使用强化学习优化各种数据库调整决策的方法。 这种方法的特点是分解重参数和轻参数，使用接受延迟反馈的 RL 算法，以及通过仔细规划评估顺序来减少重新配置开销的规划器组件。**
        - 我们通过实验证明，在优化时间相同的情况下，UDO 系统找到了比参照物更好的配置。 我们的实验涵盖了各种基准和指标。
        - 我们提出了一种新的 MCTS 变体，即延迟 HOO，可用于优化轻型和重型参数。 我们表明，在适度简化的假设下，使用该方法，UDO 收敛到接近最优的解决方案。

- FORMAL MODEL
    - 定义2.1：调优参数代表一个原子决策，影响特定工作负载的数据库管理系统的性能。 它与一个（离散的）值域相关联，代表可接受的参数值。 它可能受约束，根据其他调整参数的值限制其值。
    - 我们在广义上使用术语“参数”，包括系统配置参数设置以及物理设计决策。 下面，我们给出调整参数的例子。
    - 例子：考虑到给定数据库的一组候选索引，我们将一个调优参数与每个候选关联。此类索引相关参数具有二进制值域，表示是否创建索引。
        - 同样，我们可以引入一个调整参数来表示 Postgres 系统的 random_page_cost 配置参数（以及一组要考虑的值）。
        - 最后，我们可以将事务模板中的查询与一个调整参数相关联，该参数表示评估它的模板内的位置（可允许的位置集通过控制流和数据相关性受到限制）。
    - 定义2.3：给定固定的有序参数，配置 c 是一个向量，为每个参数分配一个特定的值。 配置空间 C 是所有可能配置的集合。
    - 我们的目标是找到优化基准的配置。

- 定义2.4：基准指标 f 将配置 $c \in C$ 映射到实值性能结果（即 $f : C \mapsto\to$ R），它根据特定基准的特定指标表示配置的性能。更高的性能结果是可取的。我们假设 f 是随机的（即，两次评估相同的配置可能不会产生完全相同的性能）。
- 我们对 f 的定义特意是通用的，涵盖了不同类型的基准和指标。下面是几个例子。
- 例子：在我们的实验中，我们使用以下两个基准指标。
  - 当处理根据固定分布随机生成的 TPC-C 事务时，我们考虑一个基准指标 $f1$，它将配置映射到在固定时间段内测量的平均吞吐量。
  - 此外，我们考虑了一个基准指标 $f2$，它将配置映射到磁盘空间 d 消耗（例如，对于创建的索引）和所有 TPC-H 查询的运行时间 t（即 $f\,(c) = -d - \sigma \cdot t$ 其中c 是一个配置，$\sigma \in$ R+ 一个用户定义的缩放因子）。
  - 这两个基准指标都作为脚本（从 UDO 的角度来看是一个黑匣子）来实现，它返回一个数值性能结果。
- 我们提出了一个系统 UDO，它解决了以下问题。
- 定义2.6：通用数据库优化的一个实例的特点是一个基准指标 f 和一个配置空间 C。
  - 目标是找到一个最优配置 $c*$，最大化期望中的随机基准指标 f（即，$c* = \text{argmax}c \in C$ E[$f\,(c)$]）。我们将最佳预期性能（c 的）表示为 $f*$。
- 对于迭代方法，问题规范还可能包括用户定义的优化超时。
  - “通用”资格证明我们的方法在参数类型、工作负载和性能指标方面具有广泛的适用性。
  - 我们使用以下定义将 UDO 实例映射到多个情节马尔可夫决策过程，并使用 RL 解决 UDO。
- 定义2.7：情景马尔可夫决策过程 (MDP) 由元组 ⟨S,A,T,R,SD,SE⟩定义，其中 S 是状态空间，A 是一组动作，T : S × A $\mapsto\to$ S 是将状态-动作对连接到新状态的转换函数。R : S $\mapsto\to$ R 是一个将状态映射到奖励值的奖励函数。
  - 我们考虑确定性转换，但考虑随机奖励。优化对执行步骤的代理进行建模。
  - 在每个步骤中，代理选择一个动作，接收奖励，并根据所选动作转换到下一个状态。
  - 优化分为多个阶段。在每一个阶段中，代理从状态 S$D \in$ S 开始。一旦到达结束状态 S$E \subseteq$ S 之一，该阶段就结束。
  - 目标是找到一个策略（这里是我们考虑确定性转换时的一系列动作），使每一步的预期回报最大化。
- 定义2.8：我们根据与更改其值相关的开销来区分重参数和轻参数。重参数具有很高的重新配置开销，轻参数具有可忽略的开销。
  - 我们用 CH 表示重参数的配置空间（即一组表示所有可能的重参数设置的向量）。
  - 因此，整个配置空间C可以写成$C = \{cH \bullet cL | cH \in CH, cL \in CL\} = CH \times CL\} = CH \times CL$，在轻量级参数和轻量级参数之前排序。
  - 目前，UDO 认为所有表示物理数据结构（例如索引）的参数都很重（因为更改此类参数意味着创建或删除关联的数据结构）。

    - 此外，UDO 认为参数很重，需要重新启动数据库服务器才能使值更改生效。 其他参数被认为是轻的。
- 定义2.9：对于重参数 MDP，状态对应于重参数的配置（即 S ⊆ CH ），每个动作将一个重参数更改为一个新值（即，一个动作被定义为一对 $\langle p、v\rangle$ 代表参数 p 和 新值v）。
    - 转换函数将具有参数值更改（即动作）的配置（即状态）映射到新配置，反映更改后的值。
    - 起始状态 SD ∈ CH 代表默认配置（即，没有为所有系统参数创建索引和默认值）。
    - 从具有给定动作数量的开始状态可到达的所有状态都是结束状态（我们通常使用四个动作的阈值）。
    - 奖励函数 R 是特定于场景的，并基于基准指标 f 。
    - 表示重参数配置 cH 的状态的奖励与 argmax cL∈CL f(cH cL) 成正比，即当将 cH 与轻参数的最佳可能配置 cL 结合时，与基准度量的值成正比。
    - 我们通过减去默认配置的奖励来衡量原始奖励（例如，如果 f 测量特定基准的吞吐量，则 UDO 将与默认设置相比的吞吐量改进视为奖励函数）。
- 上面的定义使用了轻参数的最佳配置，导致了第二个 MDP 版本。
- 定义2.10：为每个重参数配置 ch 引入一个轻参数 MDP ML[ch]（实际上，我们将自己限制在 UDO 探索的配置中）。
    - 它的状态代表轻参数的配置，它的动作代表轻参数的值变化（类似于之前的定义）。
    - 与默认值相比，开始状态代表所有轻参数的默认值，结束状态由固定数量的轻参数更改定义。
    - 奖励函数 RL [cH] 定义为 RL[cH](cL)=f(cH•cL)−fD，其中fD 是默认配置的性能。
- 如上所示，我们将轻参数的优化建模为一系列 MDP，其中每个 MDP 与特定的重参数配置相关联。
    - 我们的问题模型假设只有奖励函数（即性能）依赖于重参数。
    - 在这种情况下，还必须为特定的重参数设置实例化状态、转换和动作。
    - 以下示例说明了重参数 MDP 和轻参数 MDP 之间的相互作用。
- 定义2.11：图 1 说明了两级 MDP 的一部分。在图示的场景中，配置空间包含两个重参数（例如，索引创建决策）和一个轻参数（例如，每个运算符使用的最大主内存量）。
    - 矩形代表图中的状态，并用配置向量（按上述顺序报告参数）进行注释。
    - 图的上半部分说明了重参数MDP（注意轻参数没有指定）。
    - 实线标记由于操作更改配置而导致的转换。
    - 虚线标记重参数状态和相关轻参数 MDP 的起始状态之间的映射。
    - 由于与索引相关的参数是二进制的，因此重参数 MDP 有四种状态（显示了其中的三种）。
    - 该图总共说明了三个 MDP（一个用于重参数，一个用于两个重配置的轻参数 MDP）。
    - 最佳状态用红色标记，表明轻参数的最佳设置可能取决于重参数设置。

    - 确定重参数的最佳设置需要首先获得轻参数的最佳相关设置。

- RELATED WORK
    - 最近，人们对使用机器学习进行数据库调优产生了浓厚的兴趣。
        - 我们的工作与利用 RL 一样，属于同一广泛的类别。
        - 先前的工作通常侧重于特定的调优选择，例如系统配置参数、索引选择或数据分区。
        - 我们的目标是通过一种统一的方法支持广泛的调优选择。
        - 我们通过实验表明，参数划分和重新配置规划等技术思想在这种情况下是有益的。
    - 传统上，数据库系统中的调整决策是基于简化执行成本模型做出的。
        - 这通常会导致实践中的次优选择。
        - UDO 不使用任何简化成本模型。 相反，它专门使用通过试运行获得的反馈来识别有希望的配置。
        - 在这方面，它也不同于之前使用机器学习进行数据库调优的大部分工作。
        - 许多相应的方法依赖于从代表性工作负载中获得的先验训练数据。
        - UDO 假设没有先前的训练数据，并从头开始学习（接近）最优配置。
        - 这使得优化成本高昂（在我们的实验中以小时为单位），但避免了泛化错误和对训练数据的需求。 UDO 将在即将举行的 SIGMOD'21 会议上展示。
- SYSTEM OVERVIEW
    - UDO 的输入是要优化的基准指标、配置空间和优化时间预算。
        - 配置空间被指定为一组要考虑的索引候选，一组具有要尝试的替代值的数据库系统参数，以及（可选）每个查询或事务模板的一组替代版本。
        - UDO 认为索引参数很重，其他参数很轻。 输出是在时间限制之前找到的最佳配置。
    - UDO 迭代直到达到时间限制。
        - 在每次迭代中，UDO 首先选择一个重参数配置进行探索（组件 A）。
        - UDO 使用强化学习来做出这个决定，以有原则的方式平衡探索（分析可用信息很少的配置）和利用（优化似乎很好的配置）的需求。
        - 然而，评估重参数的新配置可能会很昂贵。 它涉及将当前数据库配置更改为要评估的配置，例如 通过创建索引。
        - 如果当前配置接近要评估的配置，这样做会变得更便宜。
        - 因此，UDO 尝试优化评估重参数配置的时间点。 UDO 使用专门的强化学习算法，不会在选择配置后立即获得评估结果。
        - 相反，它允许一定的延迟（以选择和评估结果之间的迭代次数来衡量）。
        - 重参数的选定配置被添加到缓冲区，与最后期限相关联，在此之前结果必须可用。
        - 请注意，学习算法不会考虑当前数据库状态来决定要探索哪个配置（这样做可能会阻止 UDO 找到远离当前配置的有希望的配置）。
        - 相反，它只是为其他系统组件降低成本创造了机会。
    - 在每次迭代中，UDO 从上述缓冲区（组件 B）中选择一组重参数配置进行评估。

    - 如果配置已到最后期限（在这种情况下，别无选择），或者如果它们的评估比平时便宜（例如，因为它们与必须评估的配置共享索引），则选择配置。
    - 已订购选定的配置进行评估（组件 C）。
    - 评估的目标是通过连续放置相似的配置来降低重新配置成本。
    - 例如，如果连续评估具有相似索引的配置，则可以摊销一些索引创建成本。
- 接下来，UDO 选择轻参数的值（组件 D）。
    - 轻参数的最佳配置可能取决于重参数配置。
    - 例如，我们可能想要启用或禁用特定的连接算法（通过设置 Postgres 的 enable_nestloop 等参数），具体取决于可用的索引。
    - 对于重参数的特定配置，UDO 通过强化学习学习轻参数的合适设置。
    - 在这里，重新配置很便宜。
    - 因此，UDO 使用标准的强化学习算法，没有延迟。
    - 轻型参数针对当前的重型配置进行了优化，用于后一种学习算法的固定迭代次数。
    - 请注意，轻参数的统计信息将被保存，如果再次选择相同的重配置，将用作起点。
    - 通过基准指标评估完全指定的配置（即，用于轻型和重型参数）。
    - 这涉及执行一个脚本，该脚本执行一个示例工作负载并返回性能指标以进行优化。
    - 评估结果用于更新两种学习算法（组件 D 和 A）的统计信息。