复现 QueryCheetah 并针对查询搜索器进行优化

摘要

本研究旨在复现并优化针对基于查询的系统（QBS）的全自动隐私攻击发现方法——QueryCheetah。基于查询的系统（QBS）是共享数据的关键方法之一。QBS 允许查询者基于私有保护的数据集查询聚合信息。及时发现会泄露用户隐私的查询（攻击）是确保 QBS 可以做到隐私保护的关键之一。然而，这类攻击的开发和测试的人力成本较高，无法应对系统日益增长的复杂性。自动化方法已被证明很有前景，但目前极其计算密集，限制了其在实践中的适用性。原文提出了更快更强的全自动隐私攻击发现方法，不仅比之前的方法快了 18 倍，而且支持更广的查询语法空间。我从中学习到了“多阶段本地搜索”的核心优化思想，并发现其在实现查询搜索器的部分依然存在一些算法效率和性能上的不足，我在复现其完整系统的同时对这部分进行了优化。

关键词：自动化隐私攻击；属性推断攻击；算法优化

1 引言

在当今数据驱动的世界中，隐私保护已成为信息技术领域的一个核心议题。基于查询的系统（QBS）作为一种流行的数据分享和查询处理模式，尽管提供了数据获取的便利性，但亦引发了一系列隐私安全问题。这些系统面临着如何在提供必要数据的同时保护个人隐私的双重挑战。近年来，针对这些系统的隐私攻击层出不穷，暴露了现有防御机制的不足。因此，发展能够自动发现并防御这类攻击的技术是急需解决的问题。

QueryCheetah 作为一种先进的自动化隐私攻击发现工具，已经在属性推断攻击方面显示出比传统方法更快、更有效的优势。然而，随着攻击手段的不断演进和系统复杂度的增加，原有的方案需要进一步的优化和扩展，以应对更广泛的攻击场景和更高效的防御需求。通过复现并优化 QueryCheetah，不仅可以加深对其工作机制和效能的了解，还可以探索将其应用于更复杂系统的可能性。

通过复现并优化 QueryCheetah，本研究不仅旨在提升该工具的性能，更重要的是通过这一平台深入分析和理解基于查询的系统中的隐私保护问题。这将有助于发现当前隐私保护措施的不足，进而推动相关技术的发展，为 QBS 系统的使用者提供更为安全、高效的服务。此外，本研究的成果还将对其他类似技术的研发提供理论指导和实践参考，具有一定的学术价值和应用前景。

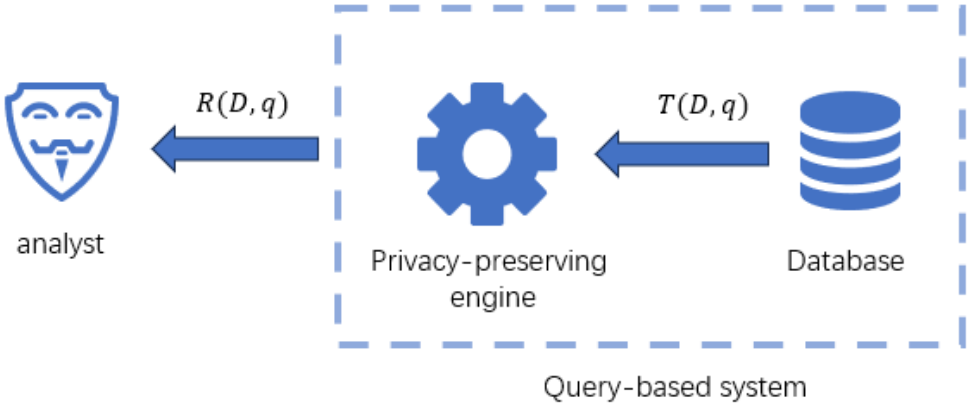


图 1. QBS 的处理过程

2 背景

2.1 基于查询的系统

令 **U** 表示所有用户，比如一项服务的所有客户或者一个国家的人口，令 **D** 表示一个分布。已知一组用户 $U \subset \mathbf{U}$，一个数据集 $D \sim \mathbf{D}$，该数据集包含这些用户在属性集 $A = \{a_i, \ldots, a_n\}$ 上的值，其中每个属性 a_i 取自集合 $\mathbf{V}_i$。用 $s_D = |U| = |D|$ 表示数据集大小。对于一个给定的用户 $u \in U$，他在数据集 D 上的表示为 $r_u = (r_u^1, r_u^2, \ldots, r_u^n)$，其中 r_u^i 表示该用户在属性 a_i 上的取值。

考虑一位数据管理员，他要确保任何人都可以对数据集 D 进行数据分析，但又不能泄露数据集中任何个人的隐私。他通过实现了 SQL 接口的基于查询的系统（QBS）来开放对数据集 D 的访问权。任何数据分析师都可以通过这个接口提交查询并得到答案。

令 **Q** 是满足 QBS 查询语法的查询集，则 QBS 在接收到一个查询 q 且 $q \in \mathbf{Q}$ 时，执行以下步骤：首先基于数据集给出真实回答 $T(D, q)$，然后（如果要进行隐私保护的话）对结果进行扰动得到 $R(D, q)$，并将扰动后的结果返回。大致流程如图 1所示。令 $Y(D, q) \subseteq U$ 表示满足查询 q 的用户集合。当 QBS 接收到一项非法的查询（不满足查询语法限制）时，即 $q \notin \mathbf{Q}$ 时，则直接返回 0。返回 0 相比于返回一个“语法不符合”等这样特定的内容其实回应了更少的信息，因为查询者其实并不知道 0 是查询结果还是查询语句非法。

QBS 支持的查询语法通常可以表示成

$$\text{SELECT agg FROM D Where } a_1 c_1 v_1 O_1 \ldots O_{n-1} a_n c_n v_n$$

其中：

- agg 表示一个聚合函数。$agg \in AGG$，比如 $AGG = \{count(), min(a_j), max(a_j), sum(a_j)\}$ 其中 $j \in \{1, \ldots, n\}$
- c_i 表示一个比较符。$c_i \in C, i \in \{1, \ldots, n\}$，比如 $C = \{=, \neq, \perp\}$，其中 $\perp$ 表示跳过该属性，也就是该属性不出现在查询的 WHERE 子句中
- v_i 表示一个实数值。$v_i \in R, i \in \{1, \ldots, n\}$

- O_i 表示一个逻辑操作符。$O_i \in O$，比如 $O = \{AND, OR\}, i \in \{1, \ldots, n-1\}$

举一个例子，当 $AGG = \{count()\}, C = \{=, \perp\}, O = \{AND\}$，则该 QBS 唯一支持的查询就是列联表。

为了实现较高的实用性，在满足查询语句要求的前提下，扰动后的回答 $R(D, q)$ 不能和真实回答 $T(D, q)$ 有太大的差别。为了预防潜在的隐私漏洞，实际情况下的的 QBS 还会采用正式或者特殊的隐私保护机制，比如限制查询语法，给结果增添有界或者无界的无偏噪声和限制返回结果中涉及到的用户的数量，即对于阈值 T，当 $|Y(D, q)| < T$ 时直接返回 0。

2.2 Diffix

原文中的攻击方法是针对实际应用的一个 QBS——Diffix 提出的。这里仅对其做简单介绍。

Diffix [5] 在上节提到的 QBS 系统的基础上还集成了多种隐私保护机制，这些机制的隐私损失在实践中难以手动测试。Diffix 是研究和开发最深入、但不提供正式隐私保证的 QBS 之一。为了通过发现攻击来逐步修补并改进系统，QBS 的作者组织了两次赏金计划，邀请专家对系统的隐私保证进行对抗性测试，并给予金钱奖励。在这个过程中发现了四种手动或半自动化的漏洞：基于先前工作的成员推断攻击（MIA）、两种重建攻击以及属性推断攻击（AIA）。此外，一种全自动方法发现了针对 Diffix 的具有更强推断能力的 AIAs。

Diffix 的主要防御措施包括：查询集大小限制、无偏无界噪声和答案取整。查询语法的隐私保证仅在有限的查询语法范围内经过测试，但实际支持的查询语法要比测试的范围更丰富，留下了大部分当前未探索的语法可能导致隐私漏洞（QueryCheetah 即针对此，实现了基于更广阔的语法空间进行攻击）。为了抵御前面发现的有效攻击，Diffix 实施了额外的防御措施，称为缓解措施。这些措施包括隔离属性、Shadow table、无条件时增加噪声以及动态噪声种子等方法，以增强系统的隐私保护能力。

3 相关工作

在此综述针对 QBS 的敌对攻击的研究，并特别关注自动化隐私攻击的发展以及差分隐私作为防御机制的应用。

3.1 对 QBS 的攻击

关于 QBS 的攻击已有丰富的研究。自 1979 年 Denning [2] 等人首次提出利用多重查询集对仅实现桶抑制（查询结果数量和查询语法限制）机制的 QBS 进行攻击。2003 年，Dinur [4] 等人提出了一种重建攻击，该攻击通过对每个答案添加至多 $O(\sqrt{n})$ 的噪声，来回答 $nlog^2n$ 个查询，其中每个查询随机选择数据集中的用户子集。随后的研究改进了该攻击在扭曲下的鲁棒性，并减少了所需的查询数量。Chipperfield [1] 等人和 Rinnot [8] 等人针对实现桶抑制和有界种子噪声添加的 QBS 提出了手动的属性推断攻击（AIA）。

3.2 针对 Diffix 的攻击

在真实的 QBS 中，Diffix 受到了最多的专注于攻击方法的研究。当前已知的有三种类型的攻击：属性推断攻击（AIA），成员推断攻击（MIA）和重建攻击。Gadotti [6] 等人针对 Diffix 提出了一种 AIA。早期版本的 Diffix 被发现容易受到 MIA 和重建攻击的威胁。Pyrgelis [7] 等人基于他们早期的工作，提出了一种针对位置数据的 MIA。

3.3 隐私攻击的自动化

在 QueryCheetah 之前，QuerySnout 是针对通用 QBS 自动发现隐私攻击的唯一方法 [9]。针对特定于 differential privacy (DP) violations 的自动化攻击研究领域更为丰富。这一系列工作中提出的方法实现了自动化推理或搜索相邻数据集或输出事件。Wang [10] 等人实现了自动推断 DP violations，过程中使用静态程序分析工具分析软件。Ding [3] 等人引入了一种名为 DPStat 的方法，它依赖于统计假设检验来检测 DP 保证的漏洞。

3.4 作为防御手段的差分隐私方法

差分隐私是一种数学上严格定义的隐私保证，可以被用作对抗 Dinur [4] 等人提出的重建攻击。它针对攻击者提供了理论上最坏情况下的安全保证。但是在实践中采用 DP 是比较困难的。首先，通常使用相对较大的 值来获得所需的隐私保护效用。其次，现有放宽的 DP 保证一般提供较弱的事件级保证而不是用户级保证。第三，由于在隐私预算有限的情况下实现定期数据发布很矛盾，许多 DP 部署会定期（例如，每月）重置其隐私预算，因此无法提供长期的隐私保证。

4 本文方法

4.1 本文方法概述

此部分对本文将要复现的工作进行概述，原文全自动属性推断攻击主要体现在两部分：自动找寻整合规则和自动找寻候选查询集。为了方便理解本章将先简要介绍一下属性推断威胁模型和半自动属性推断攻击，然后再介绍全自动属性推断攻击的方法实现。

4.2 属性推断威胁模型

威胁模型主要有以下特性：

- 攻击者可以任意访问保护数据集的 QBS
- 攻击者知道目标用户除了隐私值以外别的属性值 $r_u^{A'}$，也只知道数据集的分布 D_{aux}
- 攻击者的目的是通过大小为 m 的查询集 S($S = \{q_1, \ldots, q_m\}$) 进行 m 次询问，然后通过整合规则 V 来整合这来自 QBS 的 m 个结果来预测目标用户的隐私值 r_u^n

4.3 针对 Diffix 的半自动属性推断攻击：利用差分噪音进行攻击

Gadotti 等人 [6] 提出了一种针对 Diffix 的半自动属性推断攻击（AIA），称为差分噪音利用攻击（Differential Noise-Exploitation Attack）。该攻击利用了 Diffix 分层噪声添加机制的特性，基于手动选择的查询集 q_1 和 q_2，设计了一个针对目标用户敏感属性的攻击方法。具体而言，攻击依赖于如下两个查询：

$$\begin{aligned} q_1 \quad = \quad & \text{SELECT count(*) FROM } D \\ & \text{WHERE } a_{i_1} \neq r^u_{i_1} \text{ AND } a_{i_2} = r^u_{i_2} \text{ AND } \cdots \text{ AND } a_{i_m} = r^u_{i_m} \text{ AND } a_n = v_n \end{aligned}$$

$$\begin{aligned} q_2 \quad = \quad & \text{SELECT count(*) FROM } D \\ & \text{WHERE } a_{i_1} = r^u_{i_1} \text{ AND } a_{i_2} = r^u_{i_2} \text{ AND } \cdots \text{ AND } a_{i_m} = r^u_{i_m} \text{ AND } a_n = v_n \end{aligned}$$

其中，$r \in \{0, 1\}$，$A'' = \{i_1, \cdots, i_m\}$ 是攻击者已知的与目标用户相关的属性子集 $A'' \subseteq A$，且满足以下前提条件：

1. 用户具有唯一性，即用户之间两两不同，$\forall v \in U, v \neq u, r_v^{A''} \neq r_u^{A''}$
2. q_1 和 q_2 都不会受到桶抑制

该攻击的目的即找出使得目标用户特异性的条件，即使得 $\forall v \in U, v \neq u, v \in Y(D, q_2) \vee v \notin Y(D, q_1)$ 通过计算两个查询 q_1 与 q_2 的结果之间的差值，即 $\Delta = R(D, q_2) - R(D, q_1)$，攻击者能够抵消来自 Diffix 的静态噪声，从而利用动态噪声的分布进行攻击。当目标用户不属于查询用户集时，两个查询的差值服从正态分布 $N(0, 2)$；当目标用户属于查询用户集时，差值的均值为 1，服从正态分布 $N(1, 2l + 2)$。为了区分这两种情况，攻击者使用似然比检验的方法，结合目标用户的属性值推断其敏感属性值。

从一个简单的例子（不存在噪声扰动）考虑半自动属性推断攻击即：班里有十个男生和一个女生，想知道女生有没有及格应当如何提问。首先受查询结果限制（桶抑制），问“女生及格人数”会返回结果 0，那么便可以先提问“班上及格人数有多少”再问“除了女生外及格的人数有多少”，通过两个结果的差便可得知女生是否及格。

此外，作者提出了一种自动化搜索候选属性子集 A'' 的方法，确保生成的子集满足唯一性条件 (1) 和避免桶抑制的条件 (2)。然而，在实际操作中，这种攻击仍然需要通过人工构造查询的整合规则（即似然比攻击），因此仍然属于“半自动”攻击。

值得注意的是，Gadotti 等人 [6] 通过为查询 q_1 和 q_2 附加过滤条件 $a < v, a > v$ 等扩展了该差分攻击。虽然这些条件不会改变用户集 $Y(D, q_1), Y(D, q_2)$，但它们依赖于领域知识的人工设计，因此在自动化攻击的比较中不予考虑。

4.4 针对 Diffix 的全自动属性推断攻击

结合上一节的内容，属性推断攻击的核心就是找到“问题要怎么问”和“问到的结果怎么处理”，对应的也就是查询集合和整合规则。半自动的方法往往需要手工构造整合规则或是手工构造查询集合，而全自动方法的目标即是实现这两项任务的自动化。

4.4.1 自动寻找整合规则

这里原文采用的是通过机器学习的方式训练出一个线性回归模型来充当整合规则，也是我后面改进的主要部分。首先定义 fitness 作为查询集 S 形成有效攻击（预测隐私值）的能力（准确率）计算过程如下：

1. 将 D_{aux} 分成相等的 D_{train} 和 D_{val}。
2. 从 D_{train} 中随机均匀采样出 2 组用户数据即 $\{r_{v_1}^{A'}, r_{v_2}^{A'}, \ldots, r_{v_n}^{A'}\}$，再补上一组真的 $r_u^{A'}$，构成 T 集合，$|T| = Z + 1$。
3. 对于其中的每组记录分别上一个服从 Bernoulli(0.5) 的值，即添加 0 与 1，其概率一半一半。

$$D_1^{\text{train}} = \{r_{v_1}^{A'} \cup \{b_1\}, \ldots, r_{v_z}^{A'} \cup \{b_z\}, r_u^{A'} \cup \{b_u\}\}$$

   为了区分把 b_u 用 y_1^{train} 表示。

4. 重复 2 和 3 共 f 次，即得到 $\{D_1^{\text{train}}, \ldots, D_f^{\text{train}}\}$ 和 $\{y_1^{\text{train}}, \ldots, y_f^{\text{train}}\}$。
5. 在 D_{val} 重复 2 和 3 共 g 次，即得到 $\{D_1^{\text{val}}, \ldots, D_g^{\text{val}}\}$ 和 $\{y_1^{\text{val}}, \ldots, y_g^{\text{val}}\}$。
6. 期间实例似 QBS 共 $f + g$ 次，为了保护这 $f + g$ 组数据，每次实例化 QBS 都有不同的随机种子来生成噪声。
7. 用一组查询集 S ($|S| = m$) 来询问以上所有的 QBS，得到 $f + g$ 组向量。

$$D^{\text{train}} = \begin{pmatrix} \{R(D_1^{\text{train}}, q_1), \ldots, R(D_f^{\text{train}}, q_m)\} \\ \vdots \\ \{R(D_f^{\text{train}}, q_1), \ldots, R(D_f^{\text{train}}, q_m)\} \end{pmatrix}$$

$$D^{\text{val}} = \begin{pmatrix} \{R(D_1^{\text{val}}, q_1), \ldots, R(D_1^{\text{val}}, q_m)\} \\ \vdots \\ \{R(D_g^{\text{val}}, q_1), \ldots, R(D_g^{\text{val}}, q_m)\} \end{pmatrix}$$

8. 结合对应的 $f + g$个y^{train} 作为标签训练，即可得到一个线性回归模型。
9. 将模型用于 D^{train} 和 D^{val} 得到的准确率中较小值当作 S 的 fitness。

基于不同的查询集即可得到不同的线性回归模型，通过比较不同线性回归模型的 fitness 即可得知查询集的好坏。下一节介绍了如何在拓展后的查询语法空间中生成查询集。

4.4.2 自动寻找候选查询集

原文在找寻候选查询集时主要采用了多阶段本地搜索（multi-stage local-search）的策略。每个阶段都会执行本地搜索，即通过上文提到的 fitness 指标来寻找最佳的查询集，不同的阶段对应不同的查询语法。其多阶段框架如图 2所示。

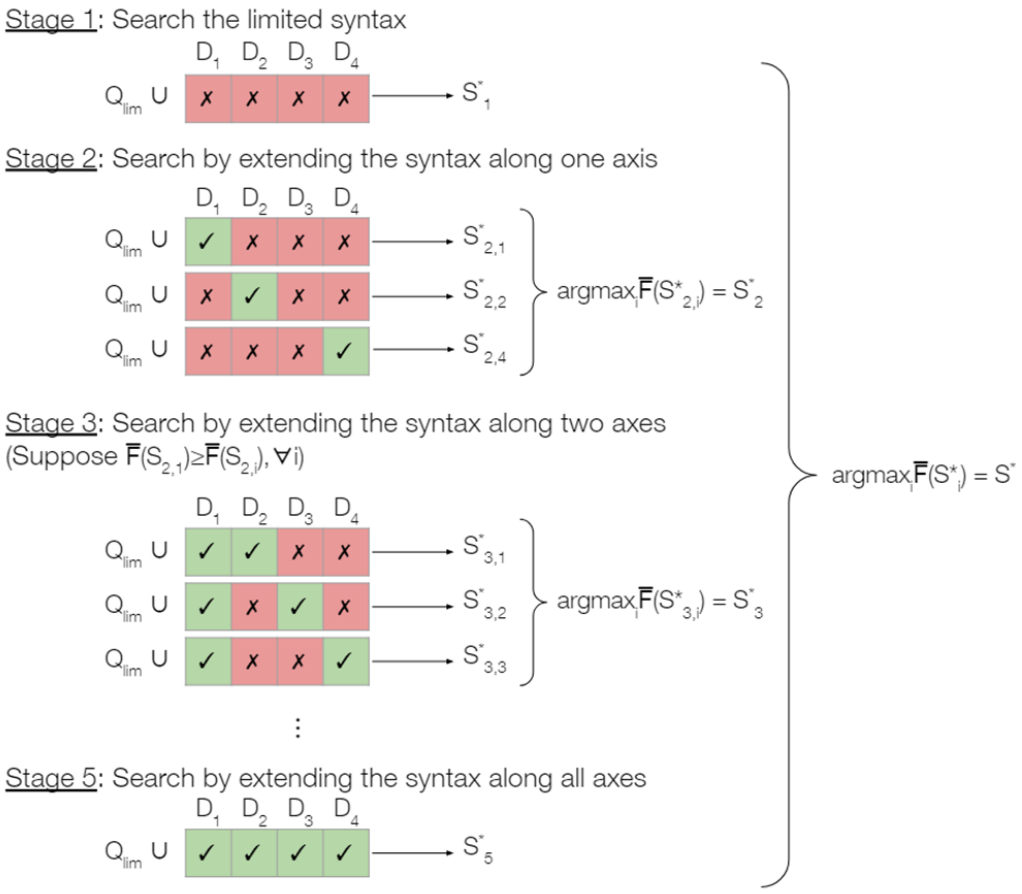


图 2. 多阶段框架

本地搜索过程中将每一阶段得到的最好的查询集 S 称作 solutions。其核心思想为将当前阶段的 solutions 用在下一阶段的选择中，用 M 来表示下一阶段的选择方法，即 $M(S_i, Q) = S_{i+1}$，为了保证查询集的性能要求下一阶段选出的查询集合的 fitness 大于此阶段得到的查询集合，即 $F(S_{i+1}) > F(S_i)$。本地搜索框架如图 3。M 的具体实现分为两部分：

1. 从 S_i 中保留 k 个 queries，即 $r(S_i) = \{q_{j_{1,i}}, q_{j_{2,i}}, \ldots, q_{j_{k,i}}\}$ 选择对应的线性回归模型中 q_j 对应系数的绝对值的大小来进行选择。
2. 在新的语法空间下生成 $m - k$ 个新的 queries，即 $g(S_i, Q) = \{q'_{1,i}, q'_{2,i}, \ldots, q'_{m-k,i}\}$

5 复现细节

5.1 与已有开源代码对比

这里引用的已有的开源代码 https://github.com/computationalprivacy/querycheetah，但是其中存在一些 BUG 无法直接运行。我在复现时优化了在本地查询阶段使用到的 Query-

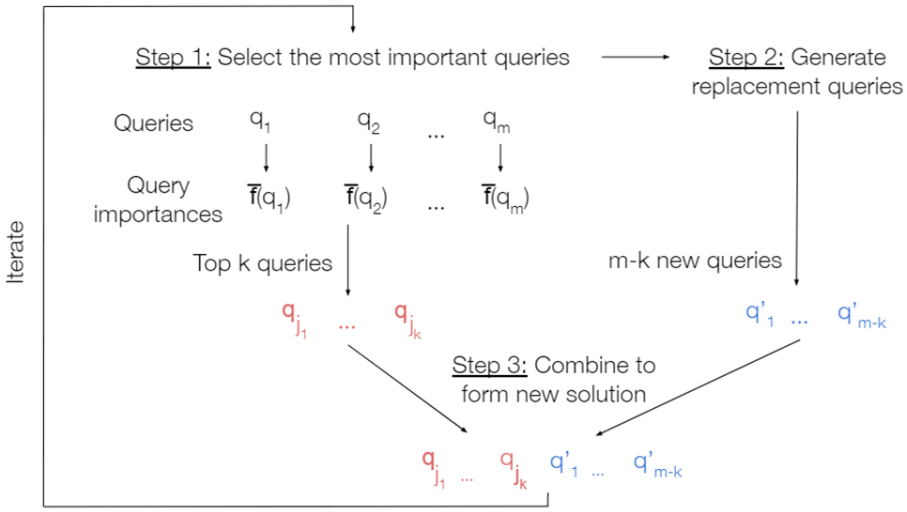


图 3. 本地搜索框架

Searcher 类，其用于根据给定的参数生成查询并通过迭代优化这些查询以达到最大化 fitness。我主要实现了以下方面：

- 迭代优化：优化查询替换的策略，减少不必要的排序。
- 在查询评估时并行处理查询，以减少计算时间。

首先在实现 search_for_this_syntax 方法时使用 multiprocessing.Pool 来并行处理查询评估，这可以大大提高查询评估的效率，特别是当查询数量很大时。修改前如图 4

```
class QuerySearcher:  11 usages
    @staticmethod  2 usages
    def search_for_this_syntax(args, aux_value_probabilities, column_names, continuous_columns, final_qbses, final_ys,
                               seed, target_record, start_queue_of_queries, start_query2query_answers, indexes):
        """
        For easier computation, instead of generating k queries at each iteration to replace the least important
        queries, we generate a queue of queries before we start the search and use them for the replacement.
        First we generate a queue of queries.
        Second, we evaluate their answers on the
        shadow QBSs protecting D_1^{train}, ..., D_f^{train} and D_1^{val}, ..., D_g^{val}.
        Third, we use them in the search.
        """

        # (1) generate a queue of queries
        queue_of_queries = QuerySearcher.construct_queue_of_queries(args, column_names, target_record,
                                                                    continuous_columns, aux_value_probabilities, seed,
                                                                    start_queue_of_queries)
        query2query_answers = {}

        # (2) evaluate the queries
        src.utils.evaluate_queries(queue_of_queries, final_qbses, args.num_procs, query2query_answers, indexes)
        queue_of_queries = start_queue_of_queries + queue_of_queries
        query2query_answers.update(start_query2query_answers)

        # (3) use the queries in the search
        starting_solution = queue_of_queries[:args.num_queries]
        queue_of_queries_pointer = args.num_queries
        all_accuracies, best_solution, best_index = QuerySearcher.search_from_solution(starting_solution, seed: 0,
                                                                                       final_qbses, final_ys,
                                                                                       args,
                                                                                       target_record, column_names,
                                                                                       queue_of_queries,
                                                                                       queue_of_queries_pointer,
                                                                                       query2query_answers,
                                                                                       continuous_columns,
                                                                                       aux_value_probabilities,
                                                                                       args.change_k_queries_at_each_iteration,
                                                                                       indexes)
        query2query_answers_dump = {query: query_answers for (query, query_answers) in query2query_answers.items() if
                                    query in best_solution}
        return all_accuracies, best_index, best_solution, query2query_answers_dump, min(all_accuracies[best_index][0],
                                                                                        all_accuracies[best_index][1])
```

图 4. 修改之前的搜索当前语法空间下的最优查询集

对应修改后的代码如图 5

```
    # (2) evaluate the queries (Parallelize evaluation)
    queue_of_queries = start_queue_of_queries + queue_of_queries
    query2query_answers.update(start_query2query_answers)

    # Parallelize query evaluations
    with Pool(args.num_procs) as pool:
        pool.starmap(src.utils.evaluate_queries,
                     [(queue_of_queries[i:i + args.num_queries], final_qbses, args.num_procs, query2query_answers, indexes)
                      for i in range(0, len(queue_of_queries), args.num_queries)])

```

图 5. 修改评估部分为并行评估

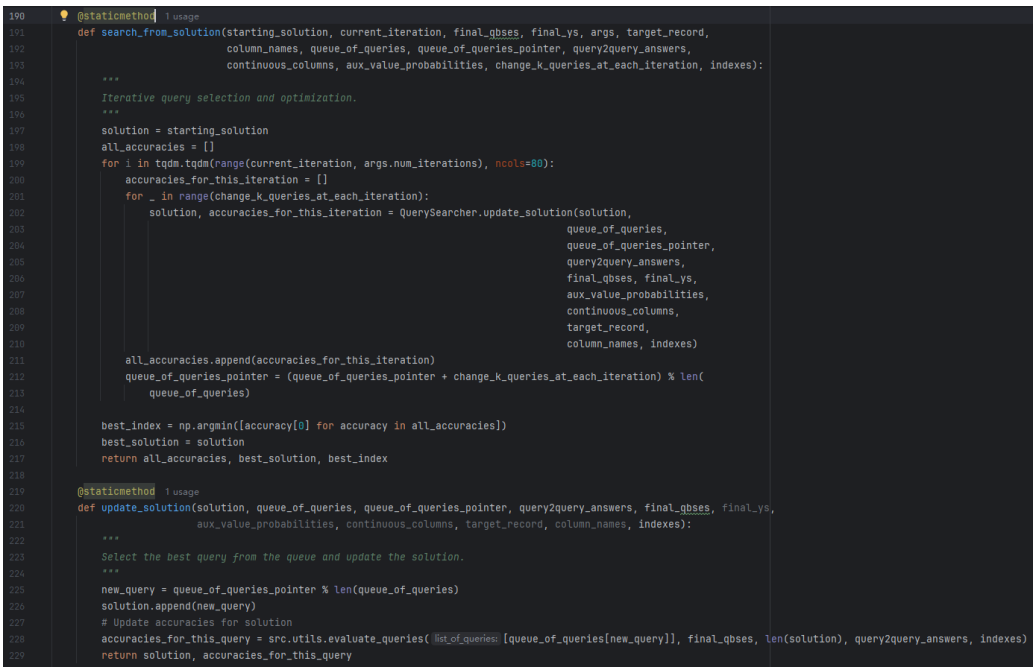
然后对查询集的更新进行了优化，通过 update_solution 方法逐步调整最优解，而不是每次都进行全局排序，减少了计算量：修改前的代码如图 6

```
def search_from_solution(solution, seed, qbses, ys, args, target_record, column_names, queue_of_queries,
    all_accuracies = [
        src.utils.get_accuracy_for_list_of_queries(solution, qbses, ys, query2query_answers, args.num_procs,
                                                   args.num_training_qbses, args.eval_fraction,
                                                   args.num_target_qbses, indexes)
    ]

    best_solution = solution[:]
    best_val_acc = all_accuracies[-1][1]
    best_index = 0
    for iteration in tqdm.tqdm(range(args.num_iterations)):
        coefs = all_accuracies[-1][-1][0]
        argsort_indices = np.argsort(abs(coefs))
        for i in range(len(argsort_indices)):
            if i >= change_k_queries_at_each_iteration:
                break
            if queue_of_queries_pointer == len(queue_of_queries):
                solution[argsort_indices[i]] = QuerySearcher.get_random_query(column_names,
                                                                              args.use_target_user_values,
                                                                              target_record,
                                                                              continuous_columns,
                                                                              aux_value_probabilities,
                                                                              args.use_operator_in,
                                                                              args.use_operator_between,
                                                                              args.use_neq_multiple_times,
                                                                              args.use_limited_syntax_fast_qbs)
            else:
                solution[argsort_indices[i]] = queue_of_queries[queue_of_queries_pointer]
                queue_of_queries_pointer += 1
        all_accuracies.append(src.utils.get_accuracy_for_list_of_queries(solution, qbses, ys, query2query_answers,
                                                                         args.num_procs,
                                                                         args.num_training_qbses,
                                                                         args.eval_fraction,
                                                                         args.num_target_qbses, indexes))
        if min(all_accuracies[-1][0], all_accuracies[-1][1]) > best_val_acc:
            best_val_acc = min(all_accuracies[-1][0], all_accuracies[-1][1])
            best_solution = solution[:]
            best_index = iteration + 1
    return all_accuracies, best_solution, best_index
```

图 6. 查询集更新优化之前

修改后的代码如图 7



```python
@staticmethod 1 usage
def search_from_solution(starting_solution, current_iteration, final_qbses, final_ys, args, target_record,
                         column_names, queue_of_queries, queue_of_queries_pointer, query2query_answers,
                         continuous_columns, aux_value_probabilities, change_k_queries_at_each_iteration, indexes):
    """
    Iterative query selection and optimization.
    """
    solution = starting_solution
    all_accuracies = []
    for i in tqdm.tqdm(range(current_iteration, args.num_iterations), ncols=80):
        accuracies_for_this_iteration = []
        for _ in range(change_k_queries_at_each_iteration):
            solution, accuracies_for_this_iteration = QuerySearcher.update_solution(solution,
                                                                                    queue_of_queries,
                                                                                    queue_of_queries_pointer,
                                                                                    query2query_answers,
                                                                                    final_qbses, final_ys,
                                                                                    aux_value_probabilities,
                                                                                    continuous_columns,
                                                                                    target_record,
                                                                                    column_names, indexes)
        all_accuracies.append(accuracies_for_this_iteration)
        queue_of_queries_pointer = (queue_of_queries_pointer + change_k_queries_at_each_iteration) % len(
            queue_of_queries)

    best_index = np.argmin([accuracy[0] for accuracy in all_accuracies])
    best_solution = solution
    return all_accuracies, best_solution, best_index

@staticmethod 1 usage
def update_solution(solution, queue_of_queries, queue_of_queries_pointer, query2query_answers, final_qbses, final_ys,
                    aux_value_probabilities, continuous_columns, target_record, column_names, indexes):
    """
    Select the best query from the queue and update the solution.
    """
    new_query = queue_of_queries_pointer % len(queue_of_queries)
    solution.append(new_query)
    # Update accuracies for solution
    accuracies_for_this_query = src.utils.evaluate_queries( list_of_queries: [queue_of_queries[new_query]], final_qbses, len(solution), query2query_answers, indexes)
    return solution, accuracies_for_this_query
```

图 7. 查询集更新优化之后

同时在初始下载原工程运行时还存在 BUG，比较琐碎，这里不再赘述。

5.2 实验环境搭建

5.2.1 初始化 Python 环境

```
conda create --name querycheetah_env python=3.9
conda activate querycheetah_env
pip install -r requirements.txt
```

5.2.2 数据集预处理

运行 notebooks/dataset_preprocessing.ipynb。其中既包含了如何下载数据集也包含了如何处理它们。

5.3 创新点

主要是对原查询搜索器进行了优化，提高了运行速度。

- 迭代优化：优化查询替换的策略，减少不必要的排序。
- 在查询评估时采用并行处理查询，以减少计算时间。

6 实验结果分析

运行环境：Win 10 i5 32GB python3.9

因为版权的原因，原文作者并未开放他们测试时的 Diffix，仅给出了没有任何防御措施的 SampleDiffix，因此针对这个系统的攻击基本可以达到完美破解，所以准确率往往可以达到百分之百。加之复刻一个防御完备的 QBS 基本相当于一个新的工程，因此在优化方面选择针对专注于 SampleDiffix 优化速度。优化前的实验结果如图

图 8. 优化前

同样的指令在优化后运行效果如图

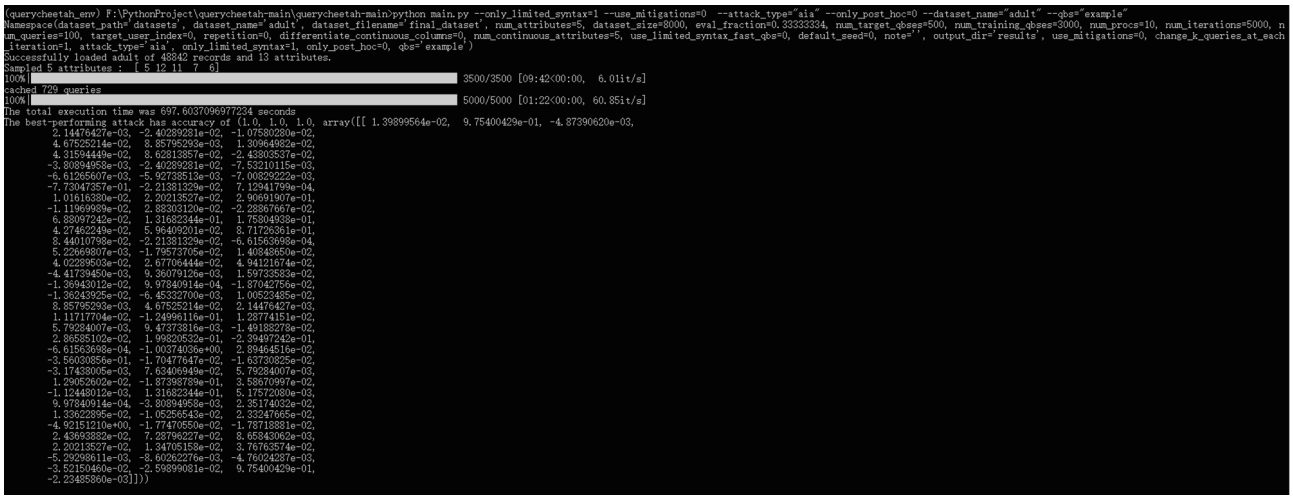


图 9. 优化后

可以看到减少了约 20 秒的运行时间。

7 总结与展望

在本文中复现 QueryCheetah 并针对其中的查询搜索器进行优化了，包括并行优化以及优化查询替换的策略来进行迭代优化，针对某一特定指令可以减少 20 秒左右的运行时间。

后续的优化方向包括：

- 实现完备的 Diffix，在真实的 QBS 上进行测试。
- 更换数据集，多测试各种指令进行统计。

参考文献

[1] James Chipperfield, Daniel Gow, and Bronwyn Loong. The Australian Bureau of Statistics and releasing frequency tables via a remote server. *Statistical Journal of the IAOS*, 32(1):53–64, 2016.

[2] Dorothy E Denning and Peter J Denning. The tracker: A threat to statistical database security. *ACM Transactions on Database Systems*, 4(1):76–96, 1979.

[3] Zeyu Ding, Yuxin Wang, Guanhong Wang, Danfeng Zhang, and Daniel Kifer. Detecting violations of differential privacy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 475–489, 2018.

[4] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 202–210, 2003.

[5] Paul Francis, Sebastian Probst Eide, and Reinhard Munz. Diffix: High-utility database anonymization. In *Privacy Technologies and Policy: 5th Annual Privacy Forum, APF 2017, Vienna, Austria, June 7-8, 2017, Revised Selected Papers 5*, pages 141–158. Springer, 2017.

[6] Andrea Gadotti, Florimond Houssiau, Luc Rocher, Benjamin Livshits, and Yves-Alexandre De Montjoye. When the signal is in the noise: Exploiting Diffix's sticky noise. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1081–1098, 2019.

[7] Apostolos Pyrgelis. On Location, Time, and Membership: Studying How Aggregate Location Data Can Harm Users' Privacy. Accessed: 01-02-2024, 2018.

[8] Yosef Rinott, Christine M O'Keefe, Natalie Shlomo, and Chris Skinner. Confidentiality and differential privacy in the dissemination of frequency tables. *Statistical Science*, 33(3):358–385, 2018.

[9] Bozhidar Stevanoski, Ana-Maria Cretu, and Yves-Alexandrede Montjoye. Querycheetah: Fast automated discovery of attribute inference attacks against query-based systems. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, page 3451–3465, Dec 2024.

[10] Yuxin Wang, Zeyu Ding, Daniel Kifer, and Danfeng Zhang. CheckDP: An automated and integrated approach for proving differential privacy or finding precise counterexamples. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 919–938, 2020.