OptBench：面向 AI/ML-SQL 协同优化的交互式工作台 [扩展演示提案]

Jaykumar Tandel
亚利桑那州立大学 坦佩，亚利桑那州，美国
jrtandel@asu.edu

Douglas Oscarson 亚利桑那州立大学 坦佩，亚利桑那州，美国
doscarso@asu.edu

Jia Zou 亚利桑那州立大学 坦佩，美国
jia.zou@asu.edu

arXiv:2603.08880v1 [cs.DB] 9 Mar 2026

摘要

数据库工作负载正越来越多地将人工智能（AI）和机器学习（ML）流水线以及 AI/ML 模型推理与数据处理嵌套在一起，从而产生混合的“SQL+AI/ML”查询，这些查询将关系算子与昂贵且不透明的 AI/ML 运算符（通常表示为用户定义函数）混合在一起。这些工作负载难以优化：（1）机器学习算子的行为如同黑盒，依赖于数据的效应（例如稀疏性、选择率或基数）可能主导运行时间；（2）领域专家往往依赖实用的启发式方法，而这些方法很难借助单体优化器来开发；（3）引入 AI/ML 运算符将带来大量协同优化机会，例如因式分解与下推、机器学习到 SQL、线性代数到关系代数等，这将显著扩大等价执行计划的搜索空间，并需要新的优化策略。与此同时，面向 SQL+ML 优化的研究原型难以进行公平的同类对比评估，因为每个优化器通常在不同平台上开发，并使用不同的查询进行评估。

我们介绍 OptBench，这是一个交互式工作台，旨在以透明、公平的方式构建和基准测试用于混合型“SQL+人工智能/机器学习”查询的查询优化器。OptBench 使用 DuckDB，在统一的后端上运行所有优化器，并提供交互式 Web 界面，让用户能够：(i) 利用和/或扩展我们抽象出的逻辑计划重写操作来构建查询优化器；(ii) 在一组多样化的查询上对不同的优化器实现进行基准测试和比较，同时记录决策轨迹和延迟；以及 (iii) 并排可视化不同优化器生成的逻辑计划。本次演示展示了从业者和研究人员如何在单一工作台中，为多模态推理查询快速验证优化器构想、检查计划转换，并定量比较优化器设计。演示视频见：https://youtu.be/hHZAkUXhEWw

ACM 参考格式：
Jaykumar Tandel、Douglas Oscarson 和 Jia Zou。2018年。OptBench：面向 AI/ML-SQL 协同优化的交互式工作台 [扩展演示提案]。载于《请务必填写权利确认邮件中的正确会议标题（会议简称‘*XX*’）》。ACM，纽约州纽约，美国，12页。https://doi.org/XXXXXXX.XXXXXXX

1 引言

SQL 引擎越来越多地被用作关系数据上数据科学和机器学习应用的端到端执行底层平台 [2– 4, 6, 7, 9, 10, 12, 14–18]。在现代流水线中，特征提取、数据转换，甚至模型推理都在“数据库内”执行，并与连接、筛选和聚合并行，生成交错使用关系算子和机器学习算子的 SQL+ML 查询，例如向量变换、深度神经网络、卷积神经网络、Transformer 模型和基于树的模型。虽然这种集成可以简化部署、减少数据移动 [4],并缓解隐私问题 [5], ，但也会引入新的性能问题。例如，错失利用数据稀疏性、对机器学习函数进行因式分解，或将线性代数运算符转换为 SQL 的机会，都可能导致端到端延迟居高不下。

遗憾的是，出于多种原因，优化 SQL+机器学习工作负载仍然颇具挑战。

首先，很难扩展传统查询优化器以支持 SQL-ML 协同优化。一方面，机器学习算子通常表示为用户定义函数或复合表达式，对传统优化器而言不透明；这些优化器主要围绕关系代数以及广为人知的物理算子而设计。此外，一些有效的启发式规则，例如“在调用昂贵模型之前应用强筛选”，或“根据稀疏性在稠密内核与稀疏内核之间切换”，由于传统关系数据库系统中的机器学习算子缺乏一等可扩展性与代价内省，很难在单体优化器中进行编码和验证。

其次，SQL+ML 的研究型和原型优化器很难进行公平比较：每个优化器通常都在不同的系统或代码库中实现，提供不同的计划表示、统计信息接口和评估流水线。例如，MADLib [6] 是 PostgreSQL 的扩展，Raven [10] 是在 SQL Server 上开发的，LMFAO [12] 是从头实现的，Morpheus [2] 则构建于 R 和 Oracle Enterprise 之上。IMBridge [15]是在 OceanBase 和 DuckDB 上开发的，而 InferF [3] 则基于 Velox 实现。因此，无法轻松地以同等条件比较这些系统中实现的 SQL-ML协同优化技术。

允许免费制作本作品全部或部分的数字副本或纸质副本，供个人或课堂使用，前提是这些副本不得为营利或商业利益而制作或分发，且副本须在首页载有本声明及完整引文。本作品中由作者以外他人拥有的组成部分，其版权必须得到尊重。允许在注明出处的情况下进行摘要。若要以其他方式复制或再版、发布到服务器或再分发到列表，则须事先获得特别许可和/或支付费用。请向 permissions@acm.org 申请许可。会议简称‘*XX*，纽约州伍德斯托克© 2018 年 版权由所有者/作者持有。出版权已授权给 ACM。ACM ISBN 978-1-4503-XXXX-X/2018/06 https://doi.org/XXXXXXX.XXXXXXX

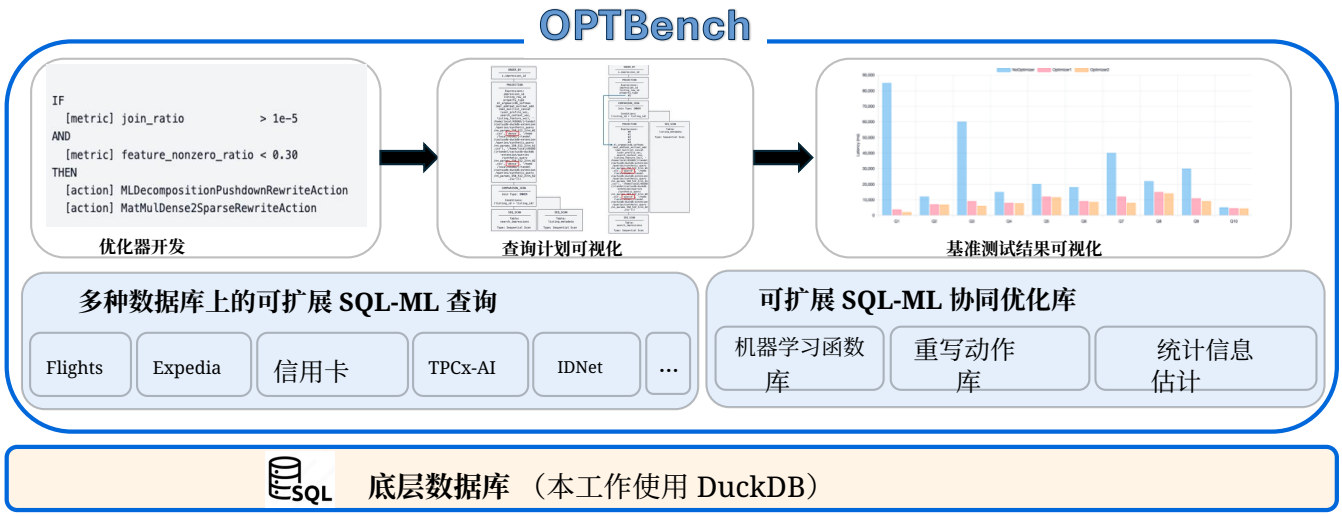

图 1：OptiBench 系统概览

第三，性能通常受数据依赖和工作负载依赖的影响，并且需要一组多样化的工作负载和数据集来评估不同的优化技术。然而，不同的 SQL-ML 协同优化系统在其评估方案中采用了不同的工作负载和数据集，缺乏统一标准。此外，优化过程通常被视为黑盒：用户只能看到最终的运行时间或成本，却很少能了解系统作出了哪些决策、哪些规则被触发，或这些决策如何依赖于数据和工作负载统计信息。

本文通过 OptBench 弥补了这些不足。OptBench 是一个交互式工作台，旨在针对 SQL 和 SQL+ML 工作负载，实现透明的优化器设计与基准测试。OptBench 源于三种常见的用户场景，用户可能是系统构建者、研究人员或数据科学家：**(1) SQL-ML 优化器开发与性能评估**。用户为 SQL-ML 工作负载设计了一种新的 SQL-ML 协同优化策略（例如，新的启发式规则或新的最优计划搜索算法），并需要在可控地访问计划结构和统计信息的条件下，公平地比较其效率和效果与其他优化策略。**(2) SQL-ML 优化器基准测试与比较**。用户可能希望在一组通用工作负载上，使用相同的硬件资源和软件配置，比较不同的优化器设计决策和算法。**(3) SQL-ML 优化器调试与改进**。用户发现，在关系型数据库中运行某些数据科学流水线时速度异常缓慢，却无法判断问题是源于连接顺序、过早进行机器学习评估，还是错误的基数假设。他们需要修改优化器（例如，启用或禁用某些规则）并运行它以进行调试。OptBench 通过提供统一且可扩展的环境支持所有这些场景，用户可在其中构建、检查和评估优化器。

OptBench 构建于 DuckDB 之上，由几个关键组件组成：**(1) 一个可扩展的机器学习函数库**，封装为 DuckDB C++ 用户定义函数，可用于组合复杂的机器学习推理工作流；(2) 一个可扩展的查询**计划重写操作**，每个操作应用一种特定的 SQL-ML 协同优化技术，将输入查询计划转换为输出查询计划，在保证查询结果等价的同时带来不同的性能表现；**(3) 一个可扩展的统计方法库**，用于计算和缓存数据统计信息、谓词选择率以及中间结果基数；**(4) 一个可扩展的多样化 SQL-ML 查询库**，用于公平评估不同的优化器；**(5) 一个基于 Web 的用户界面**，用于开发、上传、检查和评估优化器。我们选择 DuckDB 作为 OptBench 基础系统的原因在于，它提供了紧凑、现代且高度可改造的优化器和执行引擎，并且对用户定义函数和计划重写具有良好支持，从而能够快速进行 SQL-ML 优化器的原型开发和受控基准测试。此外，DuckDB 的进程内执行模型、Python/R 集成以及较低的部署开销，使 OptBench 能够直接供数据科学家使用，让他们无需搭建复杂的分布式系统，便可试验数据库内机器学习流水线、检查执行计划并诊断性能问题。未来，我们还将把该框架扩展到其他系统。

贡献。 本演示作出以下贡献：(1) 我们提出并实现了 OptBench 原型，这是一款交互式优化器工作台，可轻松开发新的 SQL-ML 查询优化器，并使用不同的优化器执行 SQL+ML 查询，以便透明地检查并进行公平比较；(2) 我们提供了基于 Web 的用户界面，用于开发或上传新的优化器，并使用两个或更多查询优化器执行所选查询，同时并排可视化查询计划和性能结果。我们还提供查询套件级基准测试，支持对异构优化器进行受控、公平的比较；(3) 我们设计了两种不同类型的交互式演示场景：(i) 使用 OptBench 库或直接通过 Web 界面开发并上传新的查询优化器；(ii) 使用 OptBench 用户界面选择不同的查询

表 1：OptBench 演示中用于表示 SQL+机器学习推理查询的机器学习函数示例。

函数	在 SQL+机器学习查询中的作用
矩阵_乘法 (mat_mul)	用于表达神经网络的核心线性算子层。
矩阵_加法（mat_add）	线性层偏置/加法及中间组合。
conv2d	二维卷积算子。
Softmax	用于分类/排序的归一化逻辑值。
Sigmoid函数	用于二元/逻辑推理的激活函数。
距离	两个向量之间的 L1 或 L2 距离。
余弦_相似度	两个向量之间的余弦相似度。
argmax	对类别分数执行的预测算子。
torch_深度神经网络	融合神经网络推理算子（后端支持）。
最小值_最大值_缩放器	特征预处理 / 归一化。
独热_编码器_编码器	类别特征扩展。
K均值	用于客户细分的聚类算子-/分组。
朴素_贝叶斯	文本/特征分类，使用贝叶斯评分。 朴素
llm	基于大语言模型的推理调用（黑盒模型调用）。
决策_树	树模型推理原语。
决策_森林	使用以下模型进行集成推理（多棵树）：XGBoost 或随机森林。

和查询优化器，以便详细比较和分析性能。

2 系统概述

如图 1 所示，OptBench 构建于 DuckDB 之上，由以下组件组成。

2.1 可扩展的机器学习函数库

OptBench面向多模态 *SQL+*机器学习推理查询，这类查询通过用户定义函数 (UDF) 将模型推理执行嵌入 SQL 中。为以自包含的方式支持这一场景，我们基于 DuckDB 构建了原型系统，提供一个可扩展的机器学习算子库，其中的算子以标量/表函数的形式提供，可用于组合 SQL 表达式。这些函数涵盖了推理流水线中的常见构件，包括线性代数算子、数据预处理函数、神经网络算子和基于树的模型。

表 1 展示了 OptBench 支持并随演示一同提供的机器学习函数示例集合。在我们的查询套件中，这些函数经组合后形成更高层次的推理算子（例如，将多层神经网络表示为嵌套的线性代数算子，或将决策森林推理表示为一组决策树推理的聚合）。

2.2 可扩展的重写操作

OptBench 将重写动作 定义为一种可复用、可组合的转换：通过应用 SQL-ML 协同优化技术来重写逻辑查询计划，同时确保相同输入下查询输出等价。这些重写操作是 SQL-ML 查询优化器的"构建模块"。SQL-ML 查询优化器会尝试选择应用这些操作的最优序列，以生成改进后的查询计划。当度量谓词成立时，基于规则的优化器会触发相应操作；而基于成本的优化器则会枚举并为通过应用一系列重写操作得到的计划评分。OptBench 附带一个小型的内置可复用重写操作库，表 2 展示了其中的示例。这些操作可用于开发基于规则和基于成本的优化器。

稠密/稀疏核选择。 此操作针对矩阵乘法表达式，例如 matrix_multiply(X, W)，并根据性能分析/指标（如非零比率）在稠密核和稀疏核之间进行选择。切换核类型可能带来益处：当输入高度稀疏时，稀疏核可减少无效计算；而当输入趋于稠密时，稠密核凭借更好的局部性和向量化，可能运行得更快。此重写在保持语义不变的同时，能够提升稀疏性因数据集而异的查询性能。

将机器学习计算关系化为连接/聚合形式。 Mul2Join-AggRewriteAction 将矩阵乘法计算（例如点积 $\sum_k x_k w_k$）改写为等价的 交叉连接 +聚合计划：将特征/值对与权重物化为关系，按特征索引配对，并用 SUM 聚合乘积。DecisionForestUDF2RelationRewriteAction 同样将决策森林推理用户定义函数替换为 交叉连接 +聚合 形式，计算每棵树的输出并加以聚合，从而得到最终预测。这些改写将机器学习推理表示为标准关系算子，从而支持连接重排序、下推和向量化执行等经典优化，同时保持语义不变。

将神经网络用户定义函数链融合为*Torch*算子。 这些操作针对以嵌套标量用户定义函数调用形式编写的神经网络推理（通常每一层或每个子表达式对应一个用户定义函数）。TwoLayerUDF2TorchNNRewriteAction 处理常见的两层模式，而 MultiLayerUDF2Torch-NNRewriteAction 则推广到更深的网络；二者都将用户定义函数链替换为表示完整模型的单个融合Torch神经网络算子。这降低了逐元组解释开销，改进了向量化执行，并为优化器提供了更清晰的规划代价边界。MultiLayerUDF2TorchNNCUDArewriteAction 在 GPU 支持可用且有益时，进一步选择该融合算子的 CUDA 后端实现，从而在保持语义的同时加速推理。

机器学习分解与细粒度下推。 MLDecomposition-PushdownRewriteAction 将复合机器学习推理表达式分解为较小的子运算符，并在安全的情况下将它们沿计划下推（例如，跨越投影并靠近筛选/连接），从而利用选择率和连接基数特性，让开销高昂的推理仅针对更少的行运行。MLFactorizationRewriteAction 进一步支持对线性/仿射形式（例如，XW +b）进行更细粒度的下推：通过在连接后的数据源之间因式分解计算，当 $X = [X_1 \mid X_2 \mid \cdots]$, 时，它会拆分 $W = [W_1; W_2; \cdots]$，并重写 $XW + b = X_1W_1 + X_2W_2 + \cdots + b$。这些重写共同将机器学习计算移至更靠近特征生成的位置，在保持语义不变的同时，减少连接后计算和中间数据移动。

利用约束降低树推理成本（TreeModelPruning-RewriteAction）。此操作适用于基于树的模型，并利用查询中已有的约束（例如过滤谓词或已知的特征边界）来识别永远不会被采用的分支。在剪枝不改变语义的情况下，它会在求值时移除不可达分支。这减少了每个元组所需执行的决策逻辑，从而加快推理且不改变结果。

表2：OptBench演示动作库中的内置重写操作。

重写动作	用途（推理重写）
MatMulDense2Sparse	切换/标注矩阵乘法为 当稀疏度指标 表明这样做有益时。
DecisionForestUDF2Relation	将决策森林推理 UDF 重写 为等价的关系形式，以便 启用下推并复用。
MatMul2Relation	将符合条件的乘法/聚合 模式重写为连接+聚合形式，以 改进执行结构。
ConvNN2MatMul	将卷积算子重写为 展平后的图像与 拼接滤波器之间的矩阵乘法。
MultiLayerUDF2TorchNN	将多层神经网络用户自定义函数表达式替换为 融合神经网络算子。
MLDecompositionPushdown	分解复合机器学习推理 表达式，并在安全的情况下将计算 下推至特征来源附近。
Fuse2TorchNN	针对两层神经网络的专用重写 将 UDF 模式转换为融合神经网络算子。
MLFactorization	因子化共享的机器学习 子表达式/权重，以减少 冗余的推理工作。
TreeModelPruning	剪枝树模型中的冗余部分 （在安全的情况下）以降低推理成本。

重写动作定制。 重要的是，动作可通过简单的注册接口进行扩展，使研究人员能够轻松开发并接入新的重写操作。

算法 1 展示了将 A 一个动作应用于重写以树结构存储、根节点为 p 的查询计划的通用方法，该过程通过树遍历实现。在迭代式树遍历过程中，每次迭代将执行以下步骤：(1) 通过 A 的 MatchesPlan 方法检查当前计划节点是否符合应用 A 的条件；如果匹配，则应用 RewritePlan；(2) 随后遍历该节点处的表达式树（例如，表示定制选择算子中过滤谓词的表达式树，或定制投影算子的表达式树），并对每个表达式应用A的 RewriteExpr 方法；(3) 移动到查询计划树中的下一个节点。该算法同时返回重写后的计划和一个布尔标志，用于指示是否发生了任何修改，从而使优化器能够迭代地应用操作（例如，直到达到不动点或耗尽搜索预算）。

以 MatMulDense2SparseRewriteAction 为例，其 RewriteExpr 实现了对矩阵_乘法

算法 1: ActionApplyPlanRewrite(A, p)

输入： 操作 A，查询计划树节点 p
输出： 元组 $(modified, p')$，其中 $modified \in \{\text{true}, \text{false}\}$

$modified \leftarrow$ false;
如果 $A.MatchesPlan(p)$ **则**
  $p \leftarrow A$.RewritePlan(p);
  $modified \leftarrow$ true;
遍历 表达式 e，包含于 p **执行**
  $(m_e, e') \leftarrow A$.RewriteExpr(e);
  将 e 替换为 e'，位于 p中;
  $modified \leftarrow modified \vee m_e$;
遍历 子计划节点 c，对 p **执行**
  $(m_c, c') \leftarrow$ ActionApplyPlanRewrite(A, c);
  在 p 中将 c 替换为 c';
  $modified \leftarrow modified \vee m_c$;
返回 $(modified, p)$;

算子的模式匹配（输入数据的密度高于可调阈值）；匹配成功后，它会将算子的执行模式/注解切换为稀疏变体，同时保持语义不变。

2.3 可扩展统计估计

OptBench 提供了一个可在各优化器之间共享的统计估计方法库，以促进优化器开发。各类统计信息来源于：(1) 数据统计信息（例如关系或中间结果的稀疏性、基数和元数），(2) 谓词选择率以及连接输入/输出基数之比（记为连接比率），以及 (3) 机器学习算子的时间与空间复杂度。我们以 DuckDB 原生的基数估计作为关系算子的基线估计，并辅以定向分析来计算与机器学习相关的统计信息（例如稀疏性和结构零值，以及机器学习算子的参数规模），这些信息通常不存在于关系目录中。示例见表 3。

为了在不执行完整查询的情况下估计面向机器学习的指标，OptBench 会对计划所引用的特征输入发起小型采样查询。这些样本用于计算稳健的汇总统计量，以指导推理重写，例如将稠密矩阵乘法切换为稀疏内核，或将高开销推理推迟到选择性算子之后。采样使指标收集保持轻量，同时仍能敏感地反映依赖于查询和数据的特性。

2.4 多样且可扩展的推理查询

OptBench 包含十个 SQL+机器学习推理查询，取自两个广泛使用的来源（表 4）。前三个查询改编自 Raven[10] 的预测查询工作负载，这些工作负载在数据科学任务中常用的真实世界数据集上进行评估，包括 Project Hamlet 的 Expedia 和 Flights 关系数据集，以及来自 Kaggle 的 CreditCard 欺诈数据集。这些工作负载有意设计为连接密集型（例如，Expedia 是三路连接，且

表3：OptBench 中支持的统计信息示例。

统计信息	计算方法	
基数 / 选择率估计	DuckDB 估算器（基线）	
非零元素比例_	非零元素总数	
零_行，零_列	全零行/列的比例	
机器学习模型浮点运算次数	通过机器学习算子的 get_shape()方法	派生
num_parameters	通过机器学习算子的 get_shape() 方法	派生
森林 _num_树	森林模型元数据	

Flights 是四路连接），并具有高维特征，类别输入大多采用独热编码，因此它们能够代表针对优化器在连接、稀疏性和推理位置方面行为的压力测试。

接下来的五个查询基于精选的 TPCx-AI 用例。TPCx-AI 是一种厂商中立的 TPC 标准基准测试，定义了端到端人工智能流水线以及一套包含十项面向业务任务的查询。我们纳入了 (UC01) 客户细分（聚类）、(UC03) 门店销售预测、(UC04) 评论垃圾信息检测（文本分类）、(UC08) 对运营记录进行多类分类，以及 (UC10) 在线欺诈/风险评分。这些查询涵盖结构化/表格数据和文本模态，涉及多种模型族（树/集成模型、提升法、聚类、朴素贝叶斯和深度神经网络推理），共同构成一个精简但多样的目标集，用于对 SQL+机器学习重写操作和优化器进行基准测试。

最后，我们纳入了两个基于 IDNet 的查询，以捕捉 SQL 中以图像为中心的 推理流水线。Q_IDNet1 将身份文件图像与结构化收费审计记录通过共享标识符（例如许可证号）连接起来，并将卷积神经网络推理用户定义函数用作选择性过滤器，仅保留非欺诈案例；该查询考察谓词位置、连接排序，以及在关系运算符之前或之后执行高开销视觉推理所带来的成本影响。Q_IDNet2 对基于大语言模型的欺诈检测器进行建模：使用不同的检索到的参考示例对每张输入图像进行多次判定（通过对演示对进行笛卡尔连接实现），随后进行多数投票聚合以生成最终标签；该查询凸显了重复黑盒推理、聚合密集型后处理，以及围绕重排序、缓存/物化和下推选择性谓词的优化器权衡。这两个查询将查询套件扩展到表格和文本工作负载之外，引入了在现代人工智能赋能的数据系统中日益常见的多模态推理模式。

2.5 Web 界面

OptBench 提供了一个基于 Web 的优化器工作台（图 2），由八个面板组成：(1) 查询选择器，用于从查询套件中选择 SQL+ML 查询；(2) 查询计划可视化 视图，以并排方式呈现逻辑计划，便于直接比较；(3) 统计窗口，用于显示针对已选查询计算出的统计信息；(4) 重写操作窗口，用于列出可用的重写操作；(5) 优化器 面板，用于列出内置和用户定义的优化器配置；(6) 优化器定义 面板，允许用户组合

表 4：基准查询套件概览：数据集、任务（查询返回的内容）以及所调用的推理模型。

查询	数据集	任务 / 输出	机器学习用户定义函数
Q_Expedia	Expedia [11]	酒店搜索排序：返回一个模型每个候选房源的评分，位于一个搜索实例（按（srch_id, prop_id））在选择性筛选条件之后。	决策 Tree
Q_Flights	Flights [11]	航线属性预测：返回每条航线的预测标签（例如代码共享/航线属性），在连接 航空公司 + 出发/目的地机场 元数据后。	决策 森林
Q_信用	信用卡-卡片 [8]	欺诈检测：为每笔交易返回一个预测的风险类别，从 V1..V28 和金额。	XGBoost
Q_UC01	TPCx-AI [1]	客户聚类：返回一个聚类 使用以下特征为每位客户分配 退货率和购买频率 特征（采用最小-最大缩放）。	K均值++
Q_UC03	TPCx-AI	预测式推理：返回一个 针对每个（门店, 部门）的预测结果，基于编码后的 类别特征和归一化 周索引。	DNN
Q_UC04	TPCx-AI	垃圾邮件分类：返回 每条评论预测的_垃圾邮件 文档，使用 分词/词干提取 + 朴素 贝叶斯评分。	朴素贝叶斯
Q_UC08	TPCx-AI	订单分类：返回一个 针对每个订单的多类别预测，来源于 聚合的订单行项目/部门 + 星期特征。	深度神经网络 + Softmax
Q_UC10	TPCx-AI	欺诈/风险评分：返回一个 Sigmoid函数 每笔交易的概率，来自 归一化金额和 营业时段特征。	深度神经网络 + Sigmoid函数
Q_IDNet1	IDNet [13]	连接 + CNN过滤器：仅保留 关联后得到的非欺诈图像 收费审计数据。	CNN
Q_IDNet2	IDNet	大语言模型投票：反复判断是否存在欺诈 利用参考资料评估每张输入图像，然后通过多数投票确定最终标签。	LLM

基于规则的优化器，将统计信息映射到动作；(7) 基准测试结果 视图，用于汇总整个查询套件的端到端延迟；(8) 上传优化器/操作 面板，用于将外部优化器配置和操作定义导入工作台。

3 个用例

在本节中，我们将介绍 OptBench 的两个用例：优化器开发和性能比较。

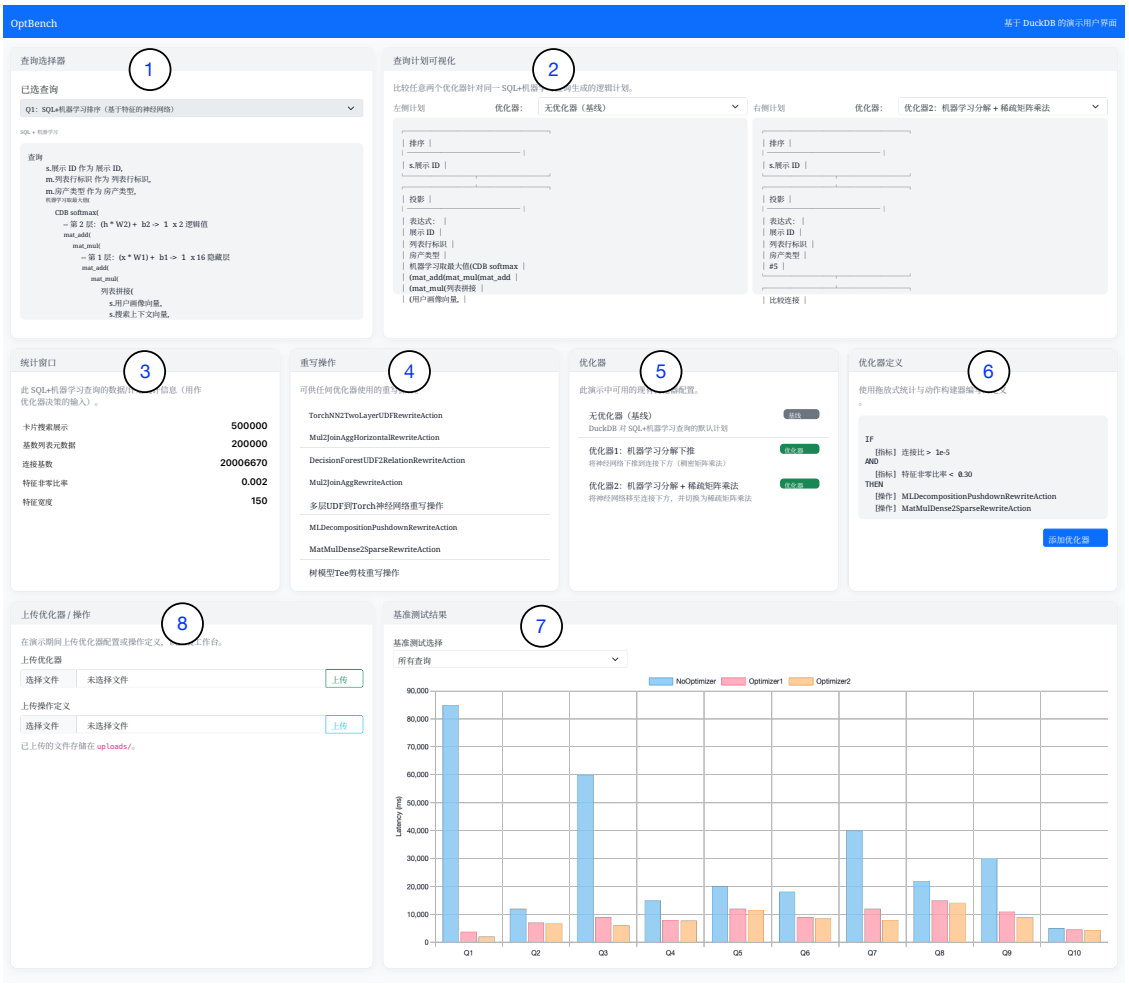

图 2：OptBench 工作台概览。用户选择一个 SQL+机器学习查询（左上），并排比较两个优化器（顶部中央），并检查统计信息、重写操作目录、优化器配置和规则定义（中部）。工作台还支持上传外部优化器/操作定义（左下），以及对查询套件进行延迟基准测试（右下）。

3.1 优化器开发、测试与评估

OptBench 通过提供重写操作和统计信息库，促进基于规则或基于成本的 SQL-ML 查询优化器的开发。此外，OptBench 还实现了一组查询，并通过其 Web 界面提供查询计划可视化，从而有助于查询优化器的测试、评估和调试。

优化器配置档案注册。如清单 1 所示，Opt-Bench 提供了一个轻量级注册接口，使每个优化器都被视为一个命名的优化器配置，可在基于 Web 的用户界面（UI）中启用并列出，以供选择进行基准测试比较。在内部，配置档案将优化器标识符绑定到优化器入口点；该入口点接收初始 DuckDB 查询逻辑计划和查询上下文（包括统计信息列表），并返回优化后的逻辑计划结构。

清单 1：在 OptiBench 中注册优化器配置（示例）

```
1 结构体 OptimizerProfile {2 std :: string name;3 std ::
function <void( OptimizerExtensionInput &,4 std :: unique_
ptr <LogicalOperator >&)>optimize_fn ;5 };67 void
RegisterOptimizer ( OptimizerProfile p);89
RegisterOptimizer ({
```

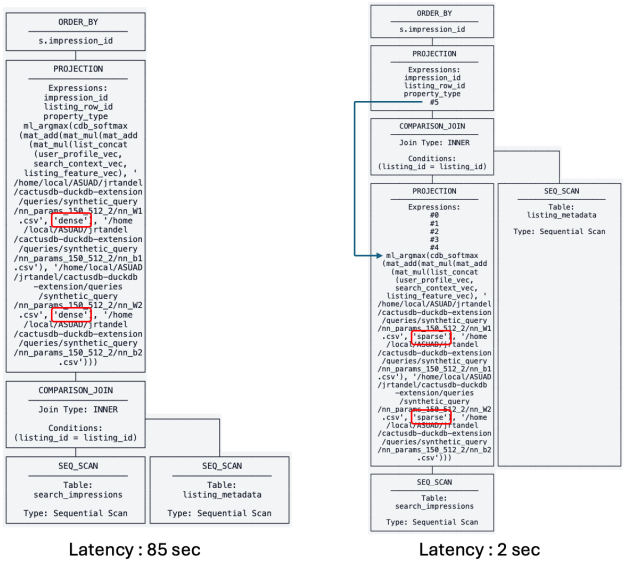

图 3：示例基于规则的优化器对推理查询的影响。从基线计划（无优化器）开始，优化器触发一条指标驱动规则，调用图中突出显示的操作（例如 MLDecompositionPushdownRewriteAction 和 MatMulDense2SparseRewriteAction）。

```
    /* 名称 =*/"DP - CostOpt", /* 优化_
函数 =*/ DPCostOpt::Run});
```

示例 *1*：开发基于规则的优化器。在 OptBench 中，只需指定一组按优先级排列的规则，即可轻松开发基于规则的优化器。这些规则将查询和数据相关统计信息映射为保持语义的推理重写操作。每条规则都是针对统计信息的谓词，或是有关数据、机器学习算子、表达式或查询计划的条件；当规则触发时，会从已注册重写操作库中执行一个或多个动作。重写操作按照算法 1 应用于当前逻辑计划。OptBench 会记录决策轨迹，其中包含已触发的规则以及各项操作引起的结构变化，使用户能够检查领域知识如何转化为具体的计划重写，以及这些重写在统一基准测试下如何影响延迟。

图 3 展示了示例 OptiBench 优化器如何通过应用图中所示的推理重写，将基线逻辑计划（计划 A，无优化器）转换为优化后的计划（计划 B）。该优化器被指定为一条简单的指标驱动规则：当查询呈现出较大的中间连接结果且特征向量足够稀疏时，它会触发 MLDecompositionPushdownRewriteAction，随后触发 MatMulDense2SparseRewriteAction。在运行时，OptiBench 会在计算出的指标向量上评估规则谓词，并将其物化为一个小型条件代码块，该代码块使用算法 1 调用已注册的动作。所得计划将神经网络推理下推到连接之下（因此被评分的元组更少），并将稠密的 mat_mul 算子切换为稀疏变体（因此每个被评分的元组开销更低）。这些改动共同将端到端延迟从图中所示的 85 秒降至 1.976 秒。

示例 *2*：开发基于成本的优化器。 OptBench 中的基于成本的优化器通过枚举 操作库所诱导的备选计划，并在成本模型与搜索策略下选择成本最低的候选方案，从而选出优化后的推理计划。除重写操作外，OptBench 的统计方法库将极大促进成本模型的开发。然而，程序员必须定义成本函数并开发搜索策略。清单 2 概述了使用 OptBench 将基于成本的优化器实现为深度受限的动态规划枚举器的紧凑 C++ 风格实现。

列表 2：紧凑型基于成本的 DP 优化器（枚举推理重写操作）

```
静态 void 运行(优化器扩展输入 &in,
               std::unique_ptr<逻辑运算符>&
                   计划) {
  int L = 2; // 重写 -深度预算
  // 操作：推理重写操作(
      DecomposePushdown, Dense2Sparse, ...)
  // 评分 (p)：成本模型使用 (in.metrics, 计划
      特征)
  // 哈希 (p)：规范 计划哈希（记忆化）

  auto 最佳 = 计划->复制();
  双精度浮点数 最佳_成本 = 评分（最佳, in);

  标准::向量<标准::唯一_指针<逻辑运算符>>
      前沿{ 计划->复制() };
  标准::无序_映射<大小_类型, 双精度>备忘录; 备忘录[
      哈希（前沿[0])] = 最佳_成本;

  对于(整数 d = 0; d < L; d++) {
    std::vector<std::unique_ptr<逻辑运算符>>
        下一个;
    用于(自动 &p : 前沿) {
      for(auto &a : 动作) {                    //
          展开所有动作
        如果(!a.Applicable(p, in)) 继续;
        auto p2 = p->Copy(); a.Apply(p2, in);
        双精度 c = 评分(p2, in); size_t h = 哈希(
            p2);
        如果(!备忘录.count(h) || c <备忘录[h]){    //
            为每个状态保留最佳项
          备忘录[h] = c; 下一步。推送_回退（p2->复制（））;
          如果(c < 最佳_成本){最佳_成本 = c; best
              = p2->复制(); }
        }
      }
    }
    如果(next.empty()) 跳出;
    前沿 = std::move(next);
  }
  计划 = std::move(best);
}
```

3.2 性能比较与基准测试

OptBench 在一套共享且可扩展的*SQL+*机器学习多模态推理查询查询套件上对每个优化器进行基准测试，如第 2.4 节所述。所有已注册的优化器，包括 DuckDB 的默认优化器、我们内置的基于规则的和基于动态规划的 SQL-ML 查询优化器，以及用户开发的优化器，均使用相同的执行后端、相同的查询和相同的数据进行评估，从而确保统一的测量环境，并实现同类对等 比较。

OptBench 提供两种视图。首先，计划对比 视图突出显示优化后计划之间的结构差异（例如，算子的位置以及重写导致的变化），从而更容易将性能差异归因于特定的推理重写。其次，基准测试面板 汇总所有选定优化器在相同后端和数据上、针对共享查询套件的端到端延迟（以及其他运行时指标）。这两种视图相结合，支持公平对比的基准测试工作流，将优化器选择与计划层面的变化及可测量的性能结果联系起来。

4 交互式演示设计

在我们提出的演示中，用户将通过以下方式体验 OptBench 的功能：(i) 使用我们的基于 Web 的界面，通过定义基于查询和数据相关统计信息触发重写操作的规则，交互式地开发基于规则的优化器，包括测试、调试和性能评估；(ii) 通过基于 Web 的界面注册并上传我们预定义的基于规则和基于成本的优化器；以及 (iii) 在一个或多个用户选定的 SQL+机器学习查询上，对多个优化器进行基准测试，实现透明且公平的对比：所有优化器均在相同的查询套件上进行评估，并在同一后端系统（我们的基于 DuckDB 的原型）上执行，从而确保延迟和计划结构的差异可归因于优化器设计决策，而非底层系统实现所造成的差异。以下将详细介绍这些交互场景。

4.1 场景 1：使用 OptBench 开发自定义优化器

每位演示参与者都可以运用其数据库系统知识，使用 OptBench 快速开发、测试、调试并评估一个简单的基于规则的 SQL-ML 查询优化器。使用 OptBench，对于仅包含一两条规则的优化器，整个过程可能只需几分钟。例如，某位参与者想要检验如下假设："对于大型连接和稀疏特征向量，将机器学习计算下推并使用稀疏内核是有益的。"他/她可以定义一条新规则，例如"如果稠密矩阵_乘法算子的输入数据超过 $1{,}000{,}000$ 行且稀疏性高于 0.7，则名为 `MatMulDense2SparseRewriteAction` 的重写动作将被触发，以调用相应的稀疏矩阵_乘法算子。随后，参与者可以编写该规则，在优化器定义 *6* 面板中组合出一个简单的查询优化器。保存后，这一新的优化器配置将与已注册并列出的其他优化器一同出现在 优化器 面板和基准测试视图。用户随后可以在计划对比窗格中并排放置任意两个优化器，并在基准测试结果 面板中查看它们的延迟条形图，从而将关于指标驱动的优化的高层假设转化为具体、可度量的实验。在与一名新用户进行的小型试点中，实现上述规则（端到端，从打开工作台到保存优化器）耗时 4 分钟。

4.2 场景 2：比较 SQL+ML 查询的优化器

参与者首先从查询选择器 中选择一个 SQL+ML 查询（图 2），然后选择两个优化器。例如，用户可以选择 DuckDB 的默认优化器，以及我们开发的基于规则或基于成本的 SQL-ML 查询优化器。基线计划可能会将机器学习算子保留为连接上方的单个高级表达式，且该连接具有较大的输出基数；而 SQL-ML 查询优化器则会应用一系列重写操作（例如，分解机器学习算子，并将计算下推至更靠近特征来源的位置）。并排的计划可视化清晰呈现结构差异，而基准测试结果 面板（图 2）则量化了它们在同一后端和数据集上的端到端延迟。此场景展示了 OptBench 如何支持对机器学习密集型工作负载上的异构优化器进行透明比较。

4.3 场景 3：扩展 OptBench

基于成本的优化器或复杂的基于规则的优化器难以仅用规则表达（例如，复杂的搜索策略或专用启发式方法），因此很难在"优化器定义 6 "面板中开发。为支持这些情况，OptBench 提供了扩展途径，可通过后端优化器接口注册外部优化器和所需的重写操作定义。在演示中，用户可以通过上传优化器/操作视图上传优化器配置或操作定义。注册后，外部优化器会显示在优化器 面板中，并可使用相同的计划对比和基准测试工作流，与内置优化器和基于规则的优化器进行比较。

此场景表明，OptBench 并不局限于单一的优化器类别：它提供了统一的评估循环，既可以检查用户编写的基于规则的优化器，也可以检查外部实现的优化器（通过计划差异），并对它们进行定量比较（通过查询套件上的延迟）。

典型的演示工作流也可以分为三个步骤：(1) 选择一个查询并检查其相关统计信息（也可以选择多个查询）；(2) 选择两个或更多优化器，对比它们的计划和延迟；(3) 可选地创建新的优化器配置（通过规则和动作），或注册外部优化器（我们将准备几个此类优化器来演示这一步），然后重新运行基准测试，以量化新设计的影响。

参考资料

[1] 2021. TPCx-AI Express基准测试。（2021年）。https://www.tpc.org/tpcx-ai/default5.asp [2] Lingjiao Chen、Arun Kumar、Jeffrey Naughton和 Jignesh M Patel。2017年。面向规范化数据上的线性代数。*VLDB Endowment* 会议论文集 10, 11 2017年( )。

[3] Kanchan Chowdhury、Lixi Zhou、Lulu Xie、Xinwei Fu和Jia Zou。2025年。InferF：连接上AI/ML推理的声明式因式分解。*arXiv*预印本 *arXiv:2511.20489*（2025年）。[4] Hong Guan、Saif Masood、Mahidhar Dwarampudi、Venkatesh Gunda、Hong Min、Lei Yu、Soham Nag和Jia Zou。2023年。端到端决策森林推理流水线的比较。收录于*2023*年*ACM*云计算研讨会论文集。200–215。[5] Hong Guan、Lei Yu、Lixi Zhou、Li Xiong、Kanchan Chowdhury、Lulu Xie、Xusheng Xiao和Jia Zou。2025年。隐私与准确性感知的AI/ML模型去重。*ACM*数据管理会议录 3, 3（2025年），1–28。[6] Joseph M Hellerstein、Christoper Ré、Florian Schoppmann、Daisy Zhe Wang、Eugene Fratkin、Aleksander Gorajek、Kee Siong Ng、Caleb Welton、Xixuan Feng、Kun Li等。2012年。MADlib分析库。*VLDB Endowment* 会议论文集 5, 12（2012年）。[7] Dimitrije Jankov、Shangyu Luo、Binhang Yuan、Zhuhua Cai、Jia Zou、Chris Jermaine和Zekai J Gao。2019年。关系数据库管理系统上的声明式递归计算：抑或，为何应使用数据库进行分布式机器学习。*VLDB Endowment* 会议论文集 12, 7（2019年），822–835。[8] Kaggle。2021年。信用卡欺诈检测。Kaggle数据集。https://www.kaggle . com/mlg-ulb/creditcardfraud 访问日期：2026年1月13日。[9] Guoliang Li、Ji Sun、Lijie Xu、Shifu Li、Jiang Wang和Wen Nie。2024年。GaussML：端到端数据库内机器学习系统。收录于*2024*年*IEEE*第*40*届国际数据工程会议（*ICDE*）。5198 6014 6 5210 10 1109 2024年 00391 – . doi: . /ICDE . 。[10] Kwanghyun Park、Karla Saur、Dalitso Banda、Rathijit Sen、Matteo Interlandi和Konstantinos Karanasos。2022年。机器学习预测查询的端到端优化。收录于*2022*年国际数据管理会议论文集。587–601。[11] Project Hamlet。2021年。Project Hamlet。项目网站。https://adalabucsd.githu b.io/hamlet.html 访问日期：2026年1月13日。[12] Maximilian Schleich、Dan Olteanu、Mahmoud Abo Khamis、Hung Q. Ngo和XuanLong Nguyen。2019年。面向分析工作负载的分层聚合引擎。收录于*2019*年国际数据管理会议论文集（阿姆斯特丹，荷兰）*(SIGMOD '19)*。计算机协会，纽约，纽约州，美国，1642–1659。doi:10.1145/3299869.3324961[13] Lulu Xie、Yancheng Wang、Hong Guan、Soham Nag、Rajeev Goel、Niranjan Swamy、Yingzhen Yang、Chaowei Xiao、Jonathan Prisby、Ross Maciejewski等。2024年。IDNet：一种经由少样本与质量驱动的合成数据生成得到的新型身份证件数据集。收录于*2024*年*IEEE*国际大数据会议（*BigData*）。IEEE，2244–2253。[14] Binhang Yuan、Dimitrije J ankov、Jia Zou、Yuxin Tang、Daniel Bourgeois和Chris Jermaine。2021年。面向分布式机器学习系统设计的张量关系代数。*VLDB*基金会会议论文集 14, 8（2021年），1338–1350。doi:10.14778/3457390 .3457399[15] Chenyang Zhang、Junxiong Peng、Chen Xu、Quanqing Xu和Chuanhui Yang。2025年。缓解预测查询执行与数据库引擎之间的阻抗失配。*ACM*数据管理会议录 3, 3，文章编号189（2025年6月），28页。https://doi.org/10.1145/3725326[16] Lixi Zhou、Jiaqing Chen、Amitabh Das、Hong Min、Lei Yu、Ming Zhao和Jia Zou。2022年。借助关系数据库去重提供深度学习模型服务。*VLDB*基金会会议论文集 15, 10（2022年），2230–2243。https://www.vldb.org/p vldb/vol15/p2230-zou.pdf [17] Lixi Zhou、Kanchan Chowdhury、Lulu Xie、Jaykumar Tandel、Hong Guan、Zhiwei Fan、Xinwei Fu和Jia Zou。2026年。CACTUSDB：释放SQL与AI/ML推理的协同优化机会。*arXiv*预印本 *arXiv:2602.23469*（2026年）。[18] L Zhou、Q Lin、K Chowdhury、S Masood、A Eichenberger、H Min、A Sim、J Wang、Y Wang、K Wu等。2024年。从关系数据库提供深度学习模型服务。数据库技术进展-*EDBT* 27, 3（2024年），717–724。

基准查询套件

清单 3：Q_Expedia（决策树推理）

```
查询
    Expedia_S_房源_扩展2 .prop_id，Expedia_S_房源_扩展2 .
    srch_id，用户定义函数( prop_location_评分1，prop_l
    ocation_评分2，prop_log_历史_价格，价格_美元，原始_目的地
    _距离，prop_评论_评分，平均_预订_美元，标准差_预订_美元，
    位置，prop_国家_id，prop_星级评分，prop_品牌_布尔值，计
    数_点击，计数_预订，
```

7 年、月、周数、时间、网站_id、访客_位置_国家_id、搜索_目的地_id、8 搜索_住宿_时长_、搜索_预订_提前期、搜索_成人_人数、搜索_儿童_人数、9 搜索_房间_数量、搜索_周六_晚_布尔值、随机_布尔值) 10 从 Expedia_S_房源_扩展2 11 连接 Expedia_R1 _酒店2 12 位于 Expedia_S_房源_扩展2 . prop_id =Expedia_R1 _酒店2 .prop_id 13 连接 Expedia_R2 _搜索 14 位于 Expedia_S_房源_扩展2 . srch_id = Expedia_R2 _搜索 .srch_id 15 其中 prop_location_评分1 > 116 且 prop_location_评分2 > 0.1 17 且 prop_log_历史_价格 > 418 且 计数_预订 > 519 且 srch_预订_提前期 > 1020 且 srch_住宿_时长_ > 1;

清单 4：Q_Flights（决策森林推理）

```
查询
    2 Flights_S_routes_extension2 . airlineid , 3 Flights_S_routes_
extension2 . sairportid , 4 Flights_S_routes_extension2 .
dairportid , 5 用户定义函数( slatitude , slongitude , dlatitude ,
dlongitude , 6 name1 , name2 , name4 , acountry , active , 7 scity ,
scountry , stimezone , sdst , 8 dcity , dcountry , dtimezone , ddst)
作为 codeshare 9 从 Flights_S_routes_extension2 10 连接 Flights_
R1 _airlines2 11 ON Flights_S_routes_extension2 . airlineid =
Flights_R1 _airlines2 . airlineid 12 连接 Flights_R2 _sairports 13
ON Flights_S_routes_extension2 . sairportid =Flights_R2 _
sairports . sairportid 14 连接 Flights_R3 _dairports 15 ON Flights_
S_routes_extension2 . dairportid =Flights_R3 _dairports .
dairportid 16 WHERE name2 ='t' 17 AND name4 ='t' 18 AND
name1 > 2.8;
```

清单 5：Q_Credit（XGBoost 推理）

```
查询时间，金额，用户定义函数(V1，V2，V3，V4，V5，V6，
V7，V8，V9，V10，V11，V12，V13，V14，V15，V16，V17，
V18，V19，V20，V21，V22，V23，V24，V25，V26，V27，
V28，金额)作为类别从信用_卡_扩展其中V1 > 1且V2 < 0.27且V
3 > 0.3;
```

列表 6：Q_UC01（K均值++ 推理，使用最小–最大缩放

购) 1 WITH table_groups AS (2 查询 3 MIN(EXTRACT (YEAR 从 CAST(date AS DATE))) 作为 invoice_year , 4 SUM(quantity * price) 作为 row_price , 5 SUM(COALESCE (or_return_quantity , 0) * price) 作为 return_row_price , 6 o_客户_sk 作为 客户_id , 7 o_订单_id 作为 订单_id 8 从 订单明细 li 9 LEFT JOIN order_returns oret 0 ON li. li_order_id = oret. or_order_id 1 AND li. li_product_id = oret. or_product_id 2 JOIN " 订单 " ord 3 ON li. li_order_id = ord. o_order_id 4 GROUP BY o_客户_sk , o_订单_id 5), 6 table_ratios AS (7 查询 8 客户_id , 9 AVG(return_row_price / NULLIF (row_price , 0)) 作为 return_ratio 0 从 table_groups 1 GROUP BY 客户_id 2), 3 table_频率_groups AS (4 查询 5 客户_id , 6 invoice_year , 7 COUNT (DISTINCT 订单_id) 作为 orders_per_year 8 从 table_groups 9 GROUP BY 客户_id , invoice_year 0), 1 table_frequency AS (2 查询 3 客户_id , 4 AVG(orders_per_year) 作为 frequency 5 从 table_frequency_groups 6 GROUP BY 客户_id 7), 8 特征_raw AS (9 查询 0 r. 客户_id , 1 f. frequency , 2 r. return_ratio 3 从 table_ratios r 4 JOIN table_frequency f 5 ON r. 客户_id = f. 客户_id 6), 7 特征 AS (8 查询 9 客户_id , 0 -- 抽象最小值--最大值缩放（你的 min_max_scaler 用户定义函数） 1 minmax_apply ('uc01_scaler ', ARRAY [frequency , return_ratio]) 作为 特征 2 从 features_raw 3) 4 查询 5 客户_id ,

-- K均值分配（簇 ID）K均值_预测（'uc01_K均值_模型 '，特征）作为簇_ID从 特征;

列表 7：Q_UC03（深度神经网络推理，基于编码后的门店–部门特征）

```
WITH 基表 作为(
  查询
    门店 ,
    部门 ,
    num_of_week ,
    门店_id_编码器 (CAST( 门店 作为 整数))
        AS 门店 _id_编码后 ,
    部门_编码器（部门） AS
        部门_编码后 ,
    转换为( 数值_的_周 / 156.0 作为 实数) AS
        num_of_week_norm
  FROM 门店_部门
) ,
特征 作为 (
  查询
    门店 ,
    部门 ,
    num_of_week ,
    转换( 拼接 ( 门店_标识符_编码 ,
        部门_编码 , 编号_的_周_标准化）作为
        实数 [ ]) AS 特征
  FROM base
)
查询
  门店 ,
  部门 ,
  num_of_week ,
  深度神经网络_预测（'uc03_模型', 特征） AS
        预测
从特征;
```

清单 8：Q_UC08（基于订单特征的深度神经网络推理；soft-max 输出）

1 2 使用 ord 作为 (3 查询 o_order_id , CAST (date AS 时间戳) 作为 ts 4 从 " order " 5), 6 ord2 作为 (7 查询 o_order_id , ts , day_of_week (ts) 作为 星期几 8 从 ord 9), 10 j1 作为 (11 查询 12 o. o_order_id , o.ts , o.weekday , 13 li. li_product_id , li. quantity 14 从 ord2 o 15 连接 订单明细 li ON o. o_order_id = li. li_order_id 16), 17 j2 作为 (

```
8 查询 9 j1. o_order_id , j1.ts , j1. weekday , 0 j1. quantity , p. 部门
1 从 j1 2 连接 product p ON j1. li_product_id = p. p_product_id
3 ), 4 agg 作为 ( 5 查询 6 o_order_id , 7 ts 作为 date , 8 部门 , 9
quantity , 0 SUM( quantity ) 作为 scan_count , 1 MIN( weekday )
作为 星期几 2 从 j2 3 GROUP BY o_order_id , ts , 部门 , quantity 4 ),
5 特征 作为 ( 6 查询 7 o_order_id , 8 date , 9 CAST ( concat (
quantity , scan_count , 星期几 , 部门_encoder ( 部门 )) AS REAL [])
作为 特征 0 从 agg 1 ) 2 查询 3 o_order_id , 4 date , 5 Softmax (
dnn_predict ('uc08_dnn_model ', 特征 )) 作为 prediction 6 从 特征 ;
```

清单 9：Q_UC10（用于欺诈检测的深度神经网络推理；Sigmoid函数输出）

```
1 WITH tx AS ( 2 查询 3 transaction_id , 4 发送方_id , 5
CAST(hour(CAST(time AS 时间戳 )) AS DOUBLE ) 作为 business_
hour_norm , 6 金额 7 从 financial_transactions 8 ), 9 j AS ( 0 查询
1 tx. transaction_id , 2 tx. business_hour_norm , 3 tx.amount , 4
fa. transaction_limit 5 从 financial_account fa 6 连接 tx ON fa. fa_
客户_sk =tx. 发送方_id 7 ), 8 特征 AS ( 9 查询 0 transaction_id , 1
CAST( concat ( amount / transaction_limit , business_hour_nor
m ) AS REAL []) 作为 features 2 从 j 3 )
```

```
查询
    交易_编号 ,sigmoid（深度神经网络_预测（'uc10_深度神经网
络_模型 ', 特征））作为 预测值 从 特征;
```

清单 10：Q_UC04（对分词后的评论进行朴素贝叶斯垃圾邮件推理）

```
1 WITH docs AS ( 2 SELECT 3 r.id , 4 -- 文本预处理用户定义函数
（已抽象）： clean -> tokenize ->stem 5 ARRAY ( 6 SELECT stem_
token (t) -- 例如， madlib . stem_token (t) 7 FROM UNNEST (
tokenize ( clean ( lower (r.text)))) AS t 8 WHERE t <>'' AND t !~'
^[0-9]+$' 9 ) AS tokens 10 FROM review r 11 ), 12 基表_先验概率
AS ( 13 -- 带标签训练集的基类先验概率（也可以存储为常量） 14
SELECT 15 LN(SUM(CASE WHEN spam = 1 THEN 1 ELSE 0
END) * 1.0 / COUNT (*)) AS log_spam_prior , 16 LN(SUM(CASE
WHEN spam = 0 THEN 1 ELSE 0 END) * 1.0 / COUNT (*)) AS log_
non_spam_prior 17 FROM uc04_train_preprocessed 18 ), 19 token_
评分 AS ( 20 SELECT 21 d.id , 22 SUM(LN( COALESCE (m. spam_
prob , 1e -9))) AS log_spam_score , 23 SUM(LN( COALESCE (m. non_
spam_prob , 1e -9))) AS log_non_spam_score 24 FROM docs d 25
CROSS JOIN UNNEST (d. tokens ) AS tok( token ) 26 LEFT JOIN uc
04_model m 27 ON m. token =tok. token 28 GROUP BY d.id 29 )
30 SELECT 31 s.id , 32 CASE 33 WHEN s. log_spam_score +p. log_
spam_prior 34 > s. log_non_spam_score + p. log_non_
spam_prior 35 THEN 1 ELSE 0 36 END AS predicted_spam 37
FROM token_scores s 38 CROSS JOIN base_priors p 39 ORDER BY
s.id; 清单 11：Q_IDNet（带卷积神经网络谓词过滤的收费审计连接）1
SELECT
```

```
    lst.* ,
    jd.*
FROM idnet AS lst
连接 收费_审计 作为jd
    开启 清单。许可证_数量 =连接。许可证_数量
其中 转换( 卷积神经网络_预测（清单。图像数据）作为 整数 )
    = 0;
```

清单 12：Q_IDNet_大语言模型_嵌套（嵌套的大语言模型调用；多数投票）

```
WITH
输入_图像 AS (
    查询 许可证_数量,
        图像数据
    FROM idnet10k
    使用 样本 10
),
ref1 AS (
    查询 图像数据 AS ref_图像1
    FROM idnet10k
    使用 样本 500
),
ref2 作为(
    查询 图像数据 AS ref_image2
    FROM idnet10k
    使用 样本 500
),
投票数 AS (
    查询
        i. 许可证_数量,
        llm(
            “告诉␣我␣ INPUT ␣图像␣是否␣
                欺诈␣或␣非欺诈␣使用␣参考␣
                图像␣和␣大语言模型␣响应。',
            i. 图像数据,
            r1. ref_图像1,
            大语言模型('这␣张 ␣图像 ␣是否 ␣欺诈? ␣若␣
                是欺诈, ␣否则为0。', r1. ref_图像1 ),
            r2. ref_图像2,
            大语言模型('这␣张 ␣图像 ␣是否 ␣欺诈? ␣若␣
                欺诈 ,␣0␣ otherwise .', r2. ref_图像2 )
        ) AS vote
    FROM input_图像 i
    交叉连接 ref1 r1
    交叉 JOIN ref2 r2
)
查询
    许可证 _数量 ,
    CASE WHEN AVG(CAST(投票 作为 DOUBLE)) >= 0.5 THEN
        1 ELSE 0 END AS is_欺诈 ,
    AVG(CAST( 投票 作为 DOUBLE)) 作为 欺诈_投票_比率,
    计数 (*) 作为 编号_投票数
FROM 投票数
分组依据 许可证 _数量
订单 BY 欺诈_投票_比率 描述 ;
```