arXiv:2602.07336v1 [cs.DB] 7 Feb 2026

阿里巴巴 MaxCompute 中的学习型查询优化器：挑战、分析与解决方案

Lianggui Weng#, Dandan Liu#, Wenzhuang Zhu#, Rong Zhu*, Junzheng Zheng*, Bolin Ding, Zhiguo Zhang, Jingren Zhou 阿里巴巴集团

摘要

由于理想化的建模假设和设计选择，现有学习型查询优化器仍不适用于现代分布式、多租户数据仓库。以阿里巴巴的 MaxCompute 为代表，我们揭示了任何可部署的学习型查询优化器都面临的四项根本性、与系统无关的挑战：1）高度动态的执行环境会导致计划成本出现较大方差；2）可能缺少成本估算所需的输入统计信息；3）传统模型改进方式不可行；4）不同工作负载带来的收益难以确定。这些挑战揭示了理论进展与生产环境现实之间的深刻错位，因此需要以部署为先，对学习型优化器进行有原则的重新设计。

为弥合这一差距，我们推出了 LOAM，这是一个面向 MaxCompute 的一站式学习型查询优化框架。其设计原则和技术具有普适性，可轻松适配类似系统。在架构方面，LOAM 引入了无统计信息计划编码，利用算子语义和历史执行记录推断数据分布的细节，并显式编码训练查询的执行环境，以学习这些环境对计划成本的影响。对于预测时环境未知的在线查询，LOAM 为可实现的性能提供了理论界，并提出了一种实用策略，以平滑环境因素对成本估算的影响。在系统运行方面，LOAM 将领域自适应技术融入训练，无需传统的细化阶段即可有效泛化到在线查询计划。此外，LOAM 还包含一个轻量级项目筛选器，用于优先选择收益较高的部署项目。在生产工作负载上，与 MaxCompute 的原生查询优化器相比，LOAM 至多节省 30% 的 CPU 成本，有望转化为可观的实际资源节省。LOAM 和示例工作负载已在 https://drive.google.com/drive/folders/1B0tqOLo2Jt6aBFxaPfjbV0B7sagFPT5u 开源。

1 引言

查询优化是数据库系统的基石，也是长期以来的研究课题。在过去十年中，利用机器学习技术的学习型查询优化器逐渐成为一种很有前景的方法，既可取代 [8, 9, 11,19, 29]，也可改进传统查询优化器 [3, 18, 30, 34, 35]，因为它们能够更好地捕捉复杂的数据分布，并提供更准确的成本估算。

尽管取得了这些进展，将学习型优化器应用于分布式多租户数据仓库（例如 Microsoft Azure Synapse Analytics [5], Google BigQuery [13], Amazon Redshift Server- less [4],阿里巴巴 MaxCompute [1],和 Snowflake Serverless [12],）仍然存在很大问题。这些平台是当今主要的云数据基础设施。本文聚焦于阿里巴巴的 MaxCompute，这是一个生产级数据仓库，可处理从 100 GB 到 EB 级别的存储和计算工作负载，并已在阿里巴巴集团内部经过大规模实战检验（每天通常处理超过 4000 万次查询）。MaxCompute 体现了先进云原生数据仓库共有的关键架构和运营原则，而这些原则也带来了部署学习型查询优化器时独有的挑战。通过研究 MaxCompute，我们旨在找出系统性障碍，并提出可推广的解决方案，将学习型优化领域的理论进展应用于生产级数据仓库。

MaxCompute 中的挑战MaxCompute。现有的学习型查询优化器通常使用计划探索器为给定查询生成一组候选执行计划，并结合学习型代价模型预测每个计划的代价。随后选择预测代价最低的计划执行。然而，在 MaxCompute 中应用这一范式面临若干关键挑战。

首先，MaxCompute 中的成本估算从根本上受到两项系统现实因素的制约，这些因素也打破了当前方法的核心假设。与以往工作通常假设的本地部署环境或低容量云容器相比，MaxCompute 中的查询执行涉及从集群级资源池进行高度动态的资源分配；该资源池平均包含 5,000 多台负载各异的机器，导致计划成本出现显著变化 [18, 19]（**C1. 环境变化**）。更糟的是，鉴于生产数据规模庞大且数据修改频繁，MaxCompute 默认不会自动、及时地维护有助于理解数据分布的统计信息（例如属性直方图），因此这些信息往往过时或缺失（**C2. 有用统计信息缺失**）。综合这些因素，当前学习型代价模型（其将静态计划特征和输入统计信息确定性地映射为代价）无法应用于 MaxCompute。

从系统运行的角度来看，为了使学习型代价模型能够泛化到计划探索器为在线查询（实时生产查询）生成的候选计划，现有工作会通过以下方式收集计划-代价对来细化模型：离线执行大量候选计划 [9, 19, 29]，或在线部署经过部分训练的模型以服务用户查询，并定期利用观测到的计划-代价对更新模型 [18, 19, 30, 34, 35]。然而，在 MaxCompute 中，生产环境的 OLAP 查询通常会处理多个数据分区，而且往往涉及大量连接，CPU 成本可高达 10^7 甚至更高，因此，仅为细化模型而执行额外的候选计划，代价高昂到难以承受。此外，在线优化

#：贡献相同；*：通讯作者 {red.zr, junzheng.zjz}@alibaba- inc.com

还可能生成效率极低的执行计划，消耗过多计算资源，这在 MaxCompute 的多租户环境中是不可接受的。综上，这些因素使传统的学习型优化器细化策略在 MaxCompute 中难以实施 （**C3. 传统细化策略不可行**）。

最后，除了查询优化之外，MaxCompute 还托管着超过 100,000 个由用户创建的数据库实例，称为 项目，每个项目都具有不同的工作负载特征（例如，连接拓扑）和数据属性（例如，表大小和更新频率）。这种异质性使学习型优化器无法在所有项目中始终保持良好性能。因此，普遍部署学习型优化器既低效也不可持续，因此，自动识别最有可能实现显著性能提升的项目至关重要（**C4. 项目选择**）。

由于 MaxCompute 与其他现代云数据仓库拥有相同的核心设计原则（见附录 A [25]），这些挑战（C1–C4）也适用于类似系统。这凸显了一个更广泛的启示：分布式多租户数据仓库中的学习型优化器，要求我们从根本上重新思考架构设计和系统运行范式。

我们的贡献。本文弥合了学习型查询优化在理论上的进展与生产级数据仓库之间的差距。针对挑战 C1–C4，我们提出了 LOAM，这是阿里巴巴MaxCompute中的一站式学习型查询优化器。它解决了 *MaxCompute* 中的核心挑战，同时为大规模多租户数据仓库中的学习型优化器提供了可迁移的设计范式，并为在生产系统中采用更实用的 *AI* 增强型查询优化技术铺平了道路。

LOAM 采用引导式策略，通过调节参数引导 MaxCompute 的原生优化器生成多样化的候选计划，并选出估计成本最低的计划（见第 3 节）。除此之外，我们提炼出四项核心原则，重新塑造了面向 MaxCompute 类系统的学习型优化器设计，并推动了 LOAM 的关键创新，使其有别于现有方法：

• **环境感知的计划成本建模**。 估计执行计划的成本时，必须明确考虑执行环境的影响（依据 *C1*）。因此，LOAM 会为每个训练计划明确编码运行时环境特征（见第 4 节）。这使成本模型能够学习不同的执行环境如何影响计划成本。关键在于，在线查询在预测（查询优化）时无法获知环境信息。这打破了当前学习型优化器的一项核心假设，即预测前能够观测到所有特征值，并会给任何查询优化器带来固有误差。为量化这种影响，我们推导出在环境变化不可观测的情况下，查询优化器可达到的最佳性能上界。在实践中，LOAM 通过在具有代表性的平均情况条件下进行成本预测来缓解这一误差（见第 5 节）。

• **无统计信息计划编码**。 计划成本估算必须在没有输入统计信息的情况下运行（依据 *C2*）。与此相一致，LOAM 对关键算子的属性进行编码，以便根据历史查询对有关 数据分布的细节进行粗略推断（见第 4 节）。

• **前瞻性泛化**。 在部署前以计算高效的方式，前瞻性地处理优化器对未见过的候选计划的泛化问题（根据 *C3*）。作为数据仓库的一项关键特性，MaxCompute 会保留大量历史数据以供长期分析。利用这一点，LOAM 会基于历史上执行过的查询计划及其代价进行离线训练。由于这些训练计划的特征分布可能与针对在线查询调优的候选计划存在显著差异，我们整合领域自适应技术 [14]，为训练计划和候选计划学习领域不变中间表示。这使针对训练计划优化的 LOAM 能够良好泛化，从而估算候选计划的代价，同时避免执行额外候选计划所需的高昂成本，以及在线优化可能带来的风险（见第 4 节）。

• **自动项目选择**。 除查询优化之外，还需要一种高效机制来自动识别具有潜在较大部署收益的项目（依据 *C4*）。为此，LOAM 通过两阶段流程自动完成项目选择（见第 6 节）：首先利用基于规则的启发式方法过滤掉不适合训练学习型优化器的项目，然后使用学习型排序器对具有高潜在收益的剩余项目进行优先排序。

我们在来自 MaxCompute 的生产工作负载上评估了 LOAM，观察到相较于原生查询优化器，性能提升最高可达 30%。LOAM 的核心技术在提升实际性能和实现更广泛的部署收益方面也证明非常有效。至关重要的是，我们的分析表明，MaxCompute 约 4% 的项目借助 LOAM 可预期获得 $\geq 10\%$ 的性能提升，这在 MaxCompute 当前庞大的服务版图（超过 100,000 个项目）中占有不可忽视的比例，并可转化为可观的现实资源节省。值得注意的是，这一估计较为保守，通过纳入更多策略来生成候选计划，还可以大幅改进。

2 背景

MaxCompute 中的查询优化同时受到其生产工作负载的巨大规模以及分布式多租户架构的挑战。在本节中，我们首先描述 MaxCompute 中的查询执行工作流（第 2.1 节），然后简要回顾现有的学习型查询优化器，并解释其假设与运行范式为何在 MaxCompute 中失效（第2.2节）。

2.1 MaxCompute 中的查询执行

项目类似于用户创建的数据库实例，是 MaxCompute 中的主要组织单元。它们提供数据的逻辑隔离和基本访问控制。用户可以通过 SQL 查询在项目内查询和操作数据。提交后，每个查询都会经历四阶段执行过程，如图 1 所示。

1. 查询编译与优化。 首先，MaxCompute 的原生查询优化器会对 SQL 语句进行编译和优化，生成一个物理计划。该计划以树结构表示，每个节点对应一项数据操作，例如表扫描、连接或聚合。MaxCompute 采用基于成本的优化器，通过一组丰富的转换规则（例如连接重排序和分区裁剪）探索等价的查询计划，生成更高效的备选方案，并选择成本模型估算代价最低的计划执行。成本模型高度依赖最新的输入统计信息，例如前 N 个值、不同值的数量（NDV）以及属性的频率直方图。然而，由于数据规模庞大（从太字节到艾字节）且数据频繁更新，为了优先保障系统

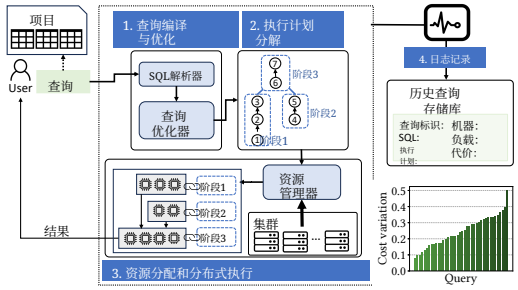


图 1：MaxCompute 中的查询执行工作流。

效率，除非用户明确启用这些统计信息，否则 MaxCompute 默认不会自动维护它们。缺少这些统计信息会严重限制优化器的效果。成本估算只能退而采用粗略的、由元数据驱动的近似方法，例如依据历史表行数进行估算，这往往会导致不可靠的计划选择。此外，许多依赖准确输入统计信息的转换规则（例如连接重排序）会被禁用，从而增加错过更优计划的可能性。

2. 执行计划分解。 随后，物理查询计划会在需要跨机器重新分配数据的算子处被分解为由阶段 组成的树，例如哈希连接和分组聚合，这些算子要求输入按连接键或分组键共同分区。每个阶段由一系列相互连接的算子组成，它们以机器内流水线的方式执行；树中的边则对应阶段之间的数据依赖关系。

3. 资源分配与分布式执行。MaxCompute 在阶段级别管理查询执行，将每个阶段视为资源分配与调度的原子单元。所有父阶段完成后，该阶段即可执行。由阿里巴巴伏羲驱动的资源管理器 [32],会根据阶段的计算需求和当前系统负载，从集群范围的共享资源池中动态分配资源。每个阶段由 1 个至超过 100,000 个并行实例执行，每个实例处理一个数据分区并生成部分结果。所有阶段完成后，系统便会返回最终输出。尽管这种方式效率很高，但阶段级别的资源分配，以及多租户干扰造成的机器间系统负载差异，都会导致查询执行成本出现不可忽视的波动。图 1 中的柱状图展示了 MaxCompute 生产工作负载中重复查询的 CPU 成本相对标准差，这些数据是在一个月内观测到的；同一查询的成本波动最高可达 50%。这给查询优化中的成本预测带来了巨大挑战。

4. 日志记录。 查询完成后，执行详情（包括 SQL 语句、物理计划、执行环境（例如，已分配的资源和系统负载）、端到端成本以及延迟）会被记录到由项目维护的历史查询存储库中，以供日后检查或下游使用。这一特性使 MaxCompute 有别于传统数据库系统，因为它为优化提供了丰富得多的数据基础。

2.2 现有工作：综述与分析

几乎所有现有的学习型查询优化器都采用相似的架构设计和系统运行范式，同时结合专门的技术来提升其性能。

架构设计框架。 学习型查询优化器 [2, 3, 8, 9, 11, 18, 19, 26, 29, 31, 34, 35]都采用计划探索器来生成候选查询计划，并配备能够预测每个计划代价的学习型成本模型，但它们在计划探索技术和成本模型设计方面有所不同。在计划探索方面，Neo [19]、Balsa [29], 及后续工作 [8, 9, 11]自行采用最佳优先或（ϵ-）束搜索策略，以自底向上的方式构建候选计划。另一种方式是，Bao [18]（及其后续工作 [3, 21, 31]）、HyperQO [30],和 Lero [35], 通过不同的调节参数引导传统查询优化器（即，算子上的提示、连接顺序中的前导提示或缩放后的基数估计），从而生成一组多样化计划。FOSS [34] 沿着这一方向推进，通过使用连接相关提示，迭代优化传统查询优化器生成的查询计划。

在成本模型设计中，成本度量指标可以采取多种形式，包括查询延迟 [8, 9, 18, 19, 30, 34], CPU 成本 [29], , 甚至相对得分 [35], , 从而能够比较不同查询计划的相对质量。然而，它们通常假设执行环境是稳定的，并将计划成本仅视为执行计划与数据的函数。相应地，成本模型采用精心构造的特征向量，用以编码计划结构（例如，算子类型）以及谓词，并附带关于输入列和表的分布的统计信息（例如，属性的直方图、估计基数或代价）。随后，估计成本通过构建于树形长短期记忆网络 [30], 图卷积网络或树卷积网络 [3, 9, 18, 19, 29, 35], 或 Transformer [8, 9, 34], 之上的深度网络得出，这些网络能够捕获复杂的树形或有向无环图形查询计划的结构信息。

在 MaxCompute 中，这些做法会带来问题。首先，如第 2.1 节所述，阶段级资源分配以及不同机器上的系统负载差异会造成不可忽略的成本波动（挑战 *1*），因此，仅依据静态计划结构和数据统计信息进行成本预测并不可靠。更糟的是，根据我们在第 5 节中的分析，由于在线查询的执行环境在查询优化阶段不可见，无论如何改进成本模型，查询优化器都难以避免环境变化带来的固有误差。其次，*MaxCompute* 不保证统计信息始终是最新或准确的，而这些学习型代价模型高度依赖底层数据的统计信息（挑战 *2*），这进一步降低了成本估算的可靠性。

系统运行范式。 几乎所有学习型查询优化器都会针对每个数据库执行实例级 优化。给定特定数据库 D，学习型代价模型基于执行计划 P 与代价 $c(P)$ 的配对样本（P， $c(P)$）进行训练。训练过程分为两个阶段：准备阶段 和细化阶段。在准备阶段，会通过不同方式构建代价模型的初始版本：1）随机初始化 [18, 30, 34]；或 2）使用朴素查询优化器生成的一组执行计划进行训练，其代价为实际代价 [8, 19] 或估算代价 [29, 31, 35] 。该阶段生成的模型非常粗糙，因此关键在于依靠细化阶段进行学习，并泛化到多样的候选计划模式。

具体而言，给定包含大量查询 Q的工作负载，计划探索器会为查询 Q生成一个或多个候选计划 P。执行 P后， P和 $c(P)$会作为经验存储起来，用于更新成本模型。此阶段既可以离线 [9, 19, 29] 进行，也可以通过部署部分训练的模型在线进行，

以处理真实查询，并定期使用观测到的计划-代价对 [18, 19, 30, 34, 35] 更新模型。

然而，MaxCompute 主要服务于 OLAP 查询，这些查询跨多个数据分区执行，并涉及大量连接。例行统计显示，每天有超过 700 万个涉及大量连接的查询，平均每个查询连接 3.8 张表。正如我们在第 7.1 节的评估项目中所展示的，每个查询的 CPU 成本通常在 10^3–10^7 量级。在如此大的规模下，为离线优化执行额外的候选计划，代价高得令人难以承受。在线细化阶段也存在问题，因为部署尚未完成训练的优化器可能会产生严重次优的执行计划，消耗过多资源；在共享的多租户环境中，这尤其不可接受。因此，这两种传统的细化方法在 *MaxCompute* 中都不可行（挑战 *3*）。

尽管解决了上述查询优化中的所有挑战，仍然存在一个关键障碍：即使经过优化，学习型优化器所能带来的收益也受工作负载模式（例如，连接拓扑）和数据属性（例如，表大小和更新频率）的限制，因此因项目而异。能够改善一个项目的优化器，在另一个项目中可能几乎没有收益，甚至毫无收益，或导致性能倒退（见第7.2.1节）。MaxCompute 拥有超过 10 万个项目，为每个项目单独训练并部署学习型优化器，既低效也难以持续。相反，我们必须自动筛选出部署学习型优化器后能显著受益的项目（挑战 *4*）。

我们指出，这些挑战并非 MaxCompute 独有。其他领先的云原生数据仓库也具有相似的工作负载模式和运行特征 [4, 5, 12, 13]（参见附录 A [25]），因此在部署学习型优化器时也会面临类似的障碍。基于这一点，我们制定了第 1 节介绍的一系列基本原则，并提出了应对这些问题的解决方案。尽管本文是在 MaxCompute 中实现并评估这些方案，但它们具有广泛的适用性，也可迁移至其他生产级数据仓库。

3 LOAM 框架

本节将概述 LOAM，这是我们为在阿里巴巴 MaxCompute 上部署学习型查询优化器提供的一站式解决方案

。它解决了上述所有挑战，并在 MaxCompute 的生产工作负载上取得了显著的性能提升。

LOAM 的整体架构如图2所示。我们没有在 MaxCompute 的所有项目中统一部署 LOAM，而是采用项目选择器，根据每个项目的历史查询对其进行评估，并自动选择最有可能从 LOAM 中受益的项目（对应挑战4）。

对于选定的项目，我们使用其历史查询存储库中的数据来训练自适应成本预测器，以估算给定查询计划的代价。根据我们的观察，MaxCompute 等分布式系统中的端到端延迟对排队延迟和网络拥塞等瞬时系统状况高度敏感，因此往往存在较大波动。因此，LOAM 预测 CPU 成本，将其作为执行计划总计算开销和长期效率的更稳定指标。为了适用于 MaxCompute，该预测器会明确建模执行环境的影响，以进行可靠的成本估算，并有意避免使用与底层数据相关的统计信息（遵循挑战 1 和挑战 2）。不同于传统的细化过程，我们还采用了

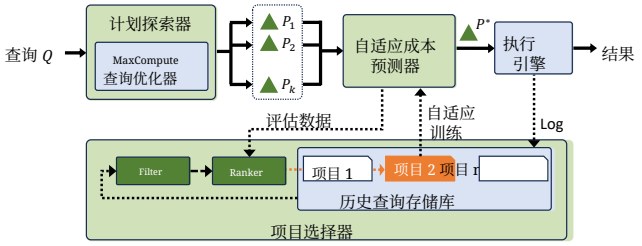


图2：LOAM 框架。

常规细化过程，我们还采用了一种轻量且有效的自适应训练机制，以便将成本预测器泛化到在线查询的候选计划。部署之前，我们会使用从历史查询存储库中抽取的一组测试查询（训练期间未见过）对预测器进行评估。为了获取这些查询的实际代价作为真实值，我们会在 MaxCompute 的飞行环境中执行它们。该环境能够重放用户查询计划，同时不损害隐私，也不会干扰用户项目的正常服务。随后，我们会根据结果判断该预测器是否适合用于生产环境。

部署 LOAM 后，每个传入的用户查询 Q 都会以引导方式进行优化，与 Bao [18] 和 Lero [35]中的方式相同。具体而言，计划探索器与 MaxCompute 原生查询优化器协同工作，生成一组候选计划 $\{P_1、P_2、\cdots P_k\}$。随后，自适应成本预测器会估算这些计划的代价，并选择估计成本最低的最优计划 P^* 进行实际执行。执行完 P^*后，其执行详情会在后台存储到历史查询存储库中，以便今后改进成本预测器。

接下来，我们简要介绍 LOAM 的三个关键组件。**计划探索器。** 对于给定的查询，计划探索器旨在生成一组多样化的候选计划，其中很可能包含代价低于 MaxCompute 原生查询优化器所生成计划的计划，我们将后者称为默认计划。它借鉴了先前研究中的思路来生成候选计划：i）遵循 Bao [18],，切换用于控制 MaxCompute 特定优化的标志。MaxCompute 提供了 75 个可调标志，分为六类（即执行模式、物理实现优化、数据流优化、并行度、存储相关设置及其他优化），从而提供了丰富的搜索空间，以发现更优的候选计划。本文仅使用涵盖连接、洗牌、临时存储和筛选相关优化的六个标志。这些标志由 MaxCompute 的领域专家选定，因为它们更有可能生成多样化的候选计划，同时足够安全，可避免产生糟糕得多的计划，并在实验阶段将评估开销控制在可接受范围内；ii）遵循 Lero [35],，对至少包含三个输入的子查询的估计基数进行缩放，以影响 MaxCompute 对计划结构的选择。值得注意的是，设计计划探索策略本身并非本文的重点。LOAM 不依赖于所采用的具体策略，并支持针对计划生成调节任意类型的参数。**自适应成本预测器。** 该成本预测器基于 MaxCompute 原生查询优化器生成并在历史上执行过的默认计划构建。其架构融入了两个针对 MaxCompute 量身定制的关键设计要点。首先，为弥补缺少属性直方图等统计信息的不足（挑战 2），它对关键算子的属性进行编码，以便从历史查询计划中粗略推断数据分布的细节

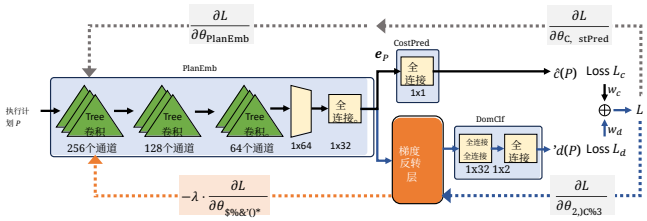


图 3：自适应成本预测器的结构。

。其次，为应对动态执行环境的影响（挑战 1），我们使用历史查询计划及其记录的执行时环境信息，对预测器进行训练，使其在执行期间按阶段粒度，根据计划结构和环境特征估算计划的代价。这样，模型便能学习环境变化如何影响实际执行成本。由于这些默认计划的特征分布可能与针对在线查询、通过各种调节参数调整的候选计划的特征分布大不相同，我们还在训练过程中加入了领域自适应 [14] 和对抗技术。这样，预测器无需传统的细化阶段即可泛化并估算候选计划的代价（对应挑战 3）。成本预测器的设计细节及训练范式将在第 4 节介绍。

然而，在部署时，在线查询的执行环境在预测时尚不可用（因为查询尚未开始执行），这与当前学习型查询优化器（第 2.2节）的假设不同；后者假设所有特征值都可用于成本估算。为解决这一问题，我们将在第 5 节中建立一个理论模型，以量化此类未观测环境特征的影响，并提出一种简单而有效的方法，以应对环境变化，稳健地估算计划成本。

项目选择器。 我们发现，项目不适合部署学习型查询优化器主要有两个原因：1）训练挑战：训练查询不足或频繁的数据修改等问题，会妨碍开发有效的学习型优化器；2）部署收益低：与通过预先调优参数调优的执行计划相比，MaxCompute 的原生查询优化器已经能够生成近乎最优的执行计划，因此部署收益有限。为筛除这些项目，我们采用两阶段、高度自动化且轻量级的项目筛选流程（对应挑战 4）。首先，利用一组明确定义的规则，通过基于规则的过滤器 粗略排除存在训练挑战的项目。随后，学习型项目*Ranker* 会按照潜在部署收益对剩余项目排序。排名靠前（例如，前-N）的项目将被选中部署 LOAM。Ranker 会定期利用多个项目的评估数据训练成本预测器。详情见第 6 节。

4 构建自适应成本预测器

自适应成本预测器旨在通过学习默认计划的历史 CPU 成本，预测计划探索器生成的候选计划的成本。其整体架构如图 3 所示。与以往工作（例如， [18, 19, 29]）一样，预测模块由一个计划嵌入层 （PlanEmb）组成

该层将经过精心设计的向量化计划 P 转换为多维中间表示，再将其输入成本预测层 （CostPred），以估算 P的代价。值得注意的是，尽管候选计划与默认计划遵循不同的分布关于默认计划，我们的方法并不依赖传统的细化来实现泛化。相反，我们采用自适应训练流程，并结合对抗技术，促使 PlanEmb 为默认计划和候选计划生成分布相似的表示，从而使 CostPred 能够估算候选计划 [23]的成本。本节首先介绍一种专为 MaxCompute 设计的计划向量化方法，然后介绍预测模块的整体设计以及自适应训练范式。**计划向量化。** 回顾一下，MaxCompute 中的每个计划 P都是一棵树，其中每个节点对应一个算子（例如，表扫描和连接）。我们将计划树[1] 中的每个节点与一个向量相关联，该向量从两个方面编码影响执行成本的关键信息：

1. 算子信息。 对于计划树中的每个节点，我们采用独热编码表示算子类型。如图 4 所示，编码的前两个位置用于指示该算子是表扫描还是归并连接。我们注意到，尽管 MaxCompute 中通常缺少统计信息（例如直方图），但我们可以对某些算子的关键属性进行编码，使模型能够根据历史查询推断信息，了解数据分布。其依据有两个方面：

i) 算子属性揭示了所访问的表和列的详细信息。因此，可以通过这些属性推断数据量、基数乃至分布等数据层洞察。例如，对于谓词相同但输入表不同的表扫描算子，我们可以根据观测到的执行成本推断哪个表更大，并粗略估计表大小。同样，对于连接条件相同的连接操作，我们可以预测连接给定的两张表会产生较大还是较小的结果集。

ii) 生产工作负载普遍由参数化的模板查询驱动，其参数会在不同运行之间发生变化。这种稳定且重复的模式为模型提供了充足信号，使其能够观察参数变化如何影响执行代价。例如，通过比较仅过滤谓词不同的查询（例如， $A_1 = a$ where a 是列 A_1取值域中的一个不同值），模型可以学习每个谓词的选择率（甚至可以根据代价差异推断列 A_1的分布）。

在专家知识的指导下，我们对 MaxCompute 支持的 30 种算子类型中最常用且对代价影响最大的几类算子进行编码：

♦ 表扫描算子。在 MaxCompute 中，包含 TB（甚至艾字节）数据的表会按照属性进行逻辑分区，以便高效访问。对于此算子，我们对表标识符以及访问的分区数和列数进行编码，这些信息能够反映处理的数据量。由于 MaxCompute 中临时表的创建和删除十分频繁，我们使用哈希编码（详见附录 B. 1 [25]）对表标识符进行编码，以避免特征维度爆炸。分区数和列数是数值变量，先对其取对数，再使用最小-最大归一化进行归一化。

[1]我们假设采用规范的二叉树；否则，将根据 [18]对执行计划进行相应转换。

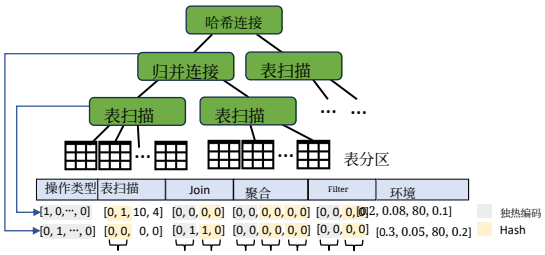


图 4：LOAM 中的执行计划编码。

♦ 所有连接和聚合算子。我们使用两个属性来表示每种连接算子类型（例如哈希连接和归并连接）：1）用于表示连接形式的独热向量（例如内连接、外连接、左连接或右连接）；2）连接列标识符的哈希编码。同样，对于每种聚合算子类型（例如哈希聚合），我们使用一个独热向量表示聚合函数（如求和和计数），并对聚合列标识符和分组列标识符进行哈希编码。这些编码使模型能够推断不同列上连接或聚合的数据量。

♦ 过滤及相关算子。 MaxCompute 中的过滤谓词采用表达式树结构，其中内部节点表示函数（例如 >，<，=），叶节点对应列和常量。这些树可能有数十层甚至数百层，因此对整棵树进行编码会大幅增加复杂度，却无法实质性地提升模型对谓词的理解。因此，我们为 Filter 算子采用一种简化表示，包含两个关键属性：1）对所有涉及函数进行多热编码；2）对所有涉及列的标识符进行哈希编码。其他实现过滤逻辑的算子（例如 Calc算子（兼具过滤和投影操作））也采用类似的编码方式。该表示使模型能够学习谓词的粗粒度选择率相对于谓词。

2. 执行环境。分配的机器的硬件配置及其负载都会影响计划成本。在 LOAM 中，我们重点对负载信息进行编码，并省略显式的单机硬件特征，原因有二：1）在生产环境中，一个阶段可能会在数百台机器上执行，直接对硬件进行编码并不切实际；2）同一集群中的机器经过有意配置，具有相同的硬件，以便于调度和负载均衡。因此，我们合理地假设各机器的计算能力相同，即计划成本与单机硬件配置无关。

我们使用四个标准指标来建模分配的机器上的负载，即 CPU_空闲、I/O_等待、5分钟系统负载和内存_使用率，它们分别反映 CPU、I/O、系统负载和内存的状态（详见附录 B.2 [25]）。在每台机器上，这些指标都高度动态，每 20 秒采样一次。因此，为了给每个计划节点提取具有代表性的环境特征，我们对每个指标在该阶段的执行窗口内取平均值；由于执行通常涉及多台机器，还要对所有分配的机器取平均值。指标 5分钟系统负载经过对数归一化，而其他指标已处于有界范围内，因此直接使用，所有特征均落在 $[0, 1]$ 范围内。值得注意的是，同一阶段内的所有计划节点都在同一组分配的机器上运行（参见第 2.1 节），因此关于执行环境，它们共享相同的编码向量。

这些特征对于在不同生产环境下执行的历史查询进行离线训练至关重要。纳入这些特征后，模型便能够学习执行环境如何影响执行成本的隐含模式，并将环境影响与计划固有特征区分开来。缺少这些特征时，成本标签对模型而言会带有噪声，难以学习从执行计划到成本的稳定映射。然而，在服务阶段，在线查询计划的环境特征无法观测。关于如何在环境特征值不可见时进行在线推理，详见第 5 节。

预测模块设计。执行计划经过向量化后，PlanEmb 将其从输入特征空间映射到一个紧凑的 n维嵌入，记为 $\boldsymbol{e}_P \in \mathbb{R}^n$。我们使用树卷积网络（TCN）构建 PlanEmb，方法类似于 [18, 19, 29, 35]。树卷积在每个树节点及其子节点上应用可学习滤波器，将信息从子节点向上聚合到父节点。通过叠加更多 TCN 层，每个节点会逐步整合来自更深层子树的层级信息。随后，对得到的节点表示进行池化，并将其输入一个全连接层，以生成 $\boldsymbol{e}_P$。CostPred 同样通过一个简单的全连接层实现，将 $\boldsymbol{e}_P$ 映射为估计成本 $\hat{c}(P)$。请注意，预测模块的设计与本文主题正交。我们还评估了其他架构，包括 Transformer [24]、图卷积网络（GCN）[17]和 XGBoost [10]。第7.2.1节报告了比较结果。

自适应训练范式 为了将在历史默认计划上训练的预测模块推广到候选计划，而无需进行传统的细化阶段，我们将无监督域自适应技术 [14] 引入训练过程。核心思想是确保 PlanEmb 生成的嵌入既能有效区分成本预测，又能不受域偏移影响。具体而言，我们希望默认计划的 PlanEmb(P)P 与候选计划的 PlanEmb(P') P' 的分布尽可能相似，从而使 CostPred 能够泛化并估计候选计划的成本 [23]。为此，我们在预测模块中加入了一个域分类器 DomClf，用于区分默认计划和候选计划生成的嵌入（见图 3）。随后，我们在 PlanEmb 和 DomClf 之间进行对抗训练：PlanEmb 旨在生成跨域相似的表示，而 DomClf 则努力将它们区分开来。这种相互作用促使 PlanEmb 生成域不变嵌入。在我们的实现中，DomClf 由两个全连接层组成，用于将 PlanEmb 的嵌入 $\boldsymbol{e}_P$ 映射为一个二维概率向量 $d(P)$，表示该计划 P 是默认计划还是候选计划的可能性。

在训练过程中，我们共同追求两个目标：1）优化 PlanEmb 和 CostPred，以准确估算从历史查询存储库中采样的默认计划的执行成本，记为 $\mathrm{P}_{\mathrm{def}}$；2）对 PlanEmb 和 DomClf 进行对抗训练，使 PlanEmb 为默认计划和候选计划生成域不变表示。第二个目标要求模型接触计划探索器生成的一小组候选计划，记为 $\mathrm{P}_{\mathrm{cand}}$。不过，这些计划无需执行，因为训练过程侧重于将它们与默认计划区分开来，而非估算其实际代价。如第7.2.1节所示，仅生成这些计划所需的开销微不足道。

为简化基于反向传播的对抗训练，我们遵循 [14]，在 PlanEmb 和 DomClf 之间插入一个梯度反转层（*GRL*）G，其作用为 $G(\boldsymbol{e}_P) = \boldsymbol{e}_P$，$\frac{\partial G(\boldsymbol{e}_P)}{\partial \boldsymbol{e}_P} = -\lambda I$，其中 I 为单位矩阵。也就是说，在前向传播过程中，GRL 原样传递特征；而在反向传播过程中，它会将梯度乘以 $-\lambda$ 后反转（其值按照 [14] 自动设置）。通过这一机制，DomClf 仍会被优化以区分执行计划所属的域，因为从最终损失传递到它的梯度不会改变（图 3 中的蓝色虚线箭头）。而从最终损失经由 DomClf 传递到 PlanEmb 的梯度则会被反转（图 3 中的橙色虚线箭头），促使 PlanEmb 生成域不变特征。

令 $L_c(\cdot)$ 和 $L_d(\cdot)$ 分别表示成本预测和域分类所使用的损失函数。总体损失为：

$$L = w_c \sum_{P \in \mathcal{P}_{\text{def}}} L_c\big(\hat{c}(P), c(P)\big) + w_d \sum_{P \in \mathcal{P}_{\text{def}} \cup \mathcal{P}_{\text{cand}}} L_d\big(\hat{d}(P), d(P)\big), \quad (1)$$

其中，$d(P) \in \{(0,1),(1,0)\}$ 表示 P 是默认计划还是候选计划。权重 w_c 和 w_d 用于平衡成本预测与域分类损失的量级，并在训练过程中根据各自的尺度自动调整。我们对 $L_c(\cdot)$ 使用均方误差，对 $L_d(\cdot)$ 使用交叉熵损失。

5 计划代价推理

自适应成本预测器在线部署后，会估计为每个查询生成的候选计划的执行成本。然而，在进行查询优化时，执行时的环境尚不可用，因此我们不知道应使用哪些环境特征值来推理计划成本。本节首先建立一个概率框架，以形式化并量化这种不可见的环境变化所带来的影响。我们将证明，这种变化从根本上限制了查询优化器在实际环境下始终选择真正最优计划的能力，并为实际部署中的可达到的最佳性能确立一个非平凡的上界。随后，我们介绍一种在推理时设置环境特征的实用策略，该策略能够有效减轻这种变化带来的不利影响，并提高计划选择的效率。

理论分析。令 E 为表示系统环境的多维随机变量。对于任意执行计划 P，令 $C_E(P)$ 表示其在环境 E 下的执行成本，而它本身是由 E 决定的随机变量。对于观测到的 E 的一个实例 e，成本 $c(P)$ 是 $C_{E=e}(P)$ 的值，也记作 $C_e(P)$。给定查询 Q 及其候选计划 $\mathcal{P}_Q = \{P_1, P_2, \cdots P_n\}$，以及用于成本预测的（学习得到的）成本模型 M，令 P_M 为 M 选出的执行计划，即估计成本最低的计划。在理想情况下，预言机模型 M_o 在 E 的任意实例 e 下，总能选出成本最低的执行计划，即 $P_{M_o} = \min_{P_i} C_e(P_i)$。

在实践中，M_o 无法实现，因为我们无法在执行计划之前预知 e 的值。因此，我们考虑关于相对于 E 的期望成本，并寻求一个模型 M_b，使其始终能够选出期望成本最低的执行计划，即 $P_{M_b} = \min_{P_i} \mathbb{E}[C_E(P_i)]$。虽然 P_{M_b} 可能无法在每个实例 e 上都使成本最低，但它能在所有可能的环境中平均降低成本。

为了评估不同模型的执行计划选择质量，我们衡量所选执行计划相对于预言机选择的代价偏差。对于在环境

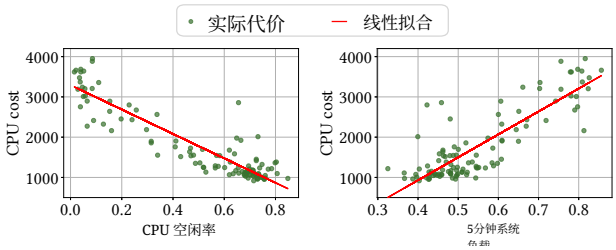


图5：重复查询的 CPU 成本相对于机器负载。

实例 e 下选择执行计划 D_e 的模型 M，我们定义 $D_e(M) = C_e(P_M) - C_e(P_{M_o})$。这会产生一个随机变量 $D_E(M)$，使得 $D_{E=e}(M) = D_e(M)$。期望偏差，即 也就是说，$\mathbb{E}[D_E(M)]$，自然地量化了 M 与预言机模型 M_o 之间的性能差距。由于在计划执行之前无法知道环境实例 e，无论我们如何努力改进模型，以下定理1指出：1）环境不可见所造成的性能差距是内在的，这与第2.2节回顾的现有学习型查询优化器所依据的假设有根本区别；2）使期望成本最小化的模型 M_b 是 可达到的最佳值。证明见附录C[25]。通过测试 MaxCompute 中的典型查询，我们发现 $\mathbb{E}[D_E(M_b)]$ 通常约为预言机代价的 10%（见第 7.2.5 节）。

定理1。对于任何在执行计划之前无法预知环境实例 e 的模型 M，我们始终有 $\mathbb{E}[D_E(M)] \geq \mathbb{E}[D_E(M_b)] \geq \mathbb{E}[D_E(M_o)] = 0$。

模型推理的实用方法。根据定理1，选择使期望成本最低的执行计划 P 能够达到可实现的最佳性能，但在实践中，计算 $\mathbb{E}[C_E(P)]$ 并不可行。在生产环境中，很难以规范形式表达环境分布及其对执行计划成本的影响，因此我们只能通过蒙特卡罗采样来近似执行计划 P 的期望成本，即 在不同的环境设置下多次执行 P，并对成本取平均值。然而，这种方法在稳定性和效率之间存在固有的权衡。要获得稳定的估计，需要大量样本，这要求重复执行执行计划，并显著增加查询优化时间。使用较少的样本可以降低开销，但会带来较高的方差，并可能导致计划选择不稳定或不是最优。

为解决这一问题，我们采用一种更实用的近似方法来估算执行成本的期望值。受 [22], 启发，我们选择一个具有代表性的环境实例 e_r，使得对于任意候选计划 P_i，$C_{e_r}(P_i)$ 都接近期望成本 $\mathbb{E}[C_E(P_i)]$。在此设定下，模型 M_r 会从所有候选计划中选择计划 $P_{M_r} = \arg\min_{P_i} C_{e_r}(P_i)$。我们对 MaxCompute 历史查询的实证分析表明，环境特征对计划成本有明显且大致单调的影响，在成本估算中可粗略地近似为线性关系。图5展示了这一点：图中将 MaxCompute 工作负载中的一个简单生产查询的 CPU 成本，与所有计划节点的 CPU_IDLE 和 5分钟系统负载的平均值进行了对比。这促使我们在成本推理时通过将每个环境特征设为其经验均值来实例化 e_r，具体而言，在查询范围内取归一化区间 $[0,1]$ 中接近 0。5 的平均值（预期 IO_WAIT 值接近 0。05 除外）。直观来说，这相当于在典型的平均系统负载条件下评估所有执行计划，这种做法公平且合理。

这种方法在计算上也很高效，因为它只需进行一次模型推理，无需额外执行计划。我们在第 7.2.5 节中的实验还表明，该策略优于其他替代方案，其期望偏差 $\mathbb{E}[D_E(M_r)]$ 接近最优值 $\mathbb{E}[D_E(M_b)]$。

6 项目选择

MaxCompute 中的项目可能不适合部署学习型查询优化器，原因在于训练挑战 和部署收益低。为了筛除不适合的项目，我们首先从每个项目的历史查询中抽取一小部分具有代表性的工作负载 Q，以捕捉其整体查询模式和工作负载特征。然后，我们应用基于规则的筛选器（Filter），排除可能带来训练挑战的项目，并使用轻量级学习型模型（Ranker）根据预估收益对其余项目进行排序。下面将详细介绍 Filter 和 Ranker。

基于规则的项目筛选器。 Filter 由一组规则组成，用于从多个方面评估项目的历史工作负载是否适合训练有效的成本预测器，例如历史查询的数量、涉及的表的属性以及执行计划结构的多样性。我们使用专门的指标量化这些方面，并要求每项指标都达到支持有效模型训练的阈值。当前的 Filter 采用三条规则：i）R1 和 R2 共同要求有足够数量的历史查询用于训练，研究表明这会显著影响成本预测器的性能（见第 7.2.4 节中的评估结果）；ii）R3 倾向于选择包含长期存在的表的项目，以便将 LOAM 从历史查询中学习到的数据分布复用于预测未来针对同一表的查询的代价。这些规则及其指标的详情见附录 D。1 [25]。我们的统计分析表明，MaxCompute 中有 59.5% 的项目被这些规则筛除，这说明许多项目尚未做好部署学习型优化器的准备，也凸显了预先筛除这类项目的重要性。

学习型项目排序器。给定查询 Q，令 P_d 表示 MaxCompute 原生查询优化器 M_d生成的默认执行计划。如第 5 节所示，期望偏差 $\mathbb{E}[D_E(M_d)]$，（下文简称 $D(M_d)$）用于衡量 M_d与预言机查询优化器之间的性能差距。由于较大的 $D(M_d)$意味着 M_d 的性能较差，因此用学习型优化器替换 M_d可能带来更多收益，我们自然会将 $D(M_d)$作为部署学习型优化器所带来的改进空间的代理指标。对于具有采样工作负载 Q 的项目，只有当 Q 中足够比例的查询的 $D(M_d)$较大时，我们才部署 LOAM。由于直接计算 $D(M_d)$十分耗时，无法扩展至 MaxCompute 各项目的采样工作负载，因此我们训练一个名为 Ranker 的学习型模型，用于估算给定查询 Q的 $D(M_d)$。

Ranker 背后的关键见解是，对于查询 Q，其默认执行计划中可观察到的属性 P_d 揭示了*LOAM*的计划探索策略所带来的优化机会，因此能够反映（即 $D(M_d)$）的改进空间 Q。例如，假设 P_d 包含多个嵌套连接算子，同时执行成本异常高。在这种情况下，这通常表明连接顺序不佳，并意味着引导 M_d 生成具有其他连接顺序的执行计划，可能会带来显著的性能提升。基于这一思路，Ranker 学习根据查询 Q的特征来估算 $D(M_d)$，具体方法是 P_d的特征。这些特征适用于各种执行计划，无需使用项目特定标识符对表或列进行编码，因此 Ranker 可以在来自多个项目的样本对（P_d， $D(M_d)$）上进行训练，然后应用于新项目。为提高可扩展性，Ranker 采用高效的计划向量化方法，对 P_d 的结构、输入特征和执行成本进行编码，并使用轻量级 XGBoost 模型 [10] 进行预测。由于篇幅所限，向量化的详细内容见附录 D。2 [25]。

使用 Ranker，我们估算通过 Filter 的每个项目的采样工作负载 Q 中各查询的改进空间 $D(M_d)$。随后，根据各项目在 Q 上的平均预估 $D(M_d)$对项目进行排序。由于不同项目的$D(M_d)$尺度不一，且 Ranker 的估算值存在固有噪声，因此难以确定一个可靠的阈值来筛选预估收益较高的项目。为提高稳健性，我们改为在排名前N的项目上部署 LOAM（其中 N是由 MaxCompute 专家指定的参数）。值得注意的是，随着越来越多的项目部署 LOAM 并接受评估，我们可以从评估数据中得出新的（P_d， $D(M_d)$）对，并定期用于更新 Ranker，从而持续提升其预测准确率和有效性。

7 实验评估

本节对 LOAM 进行了全面的实证评估。我们首先在第 7.1 节介绍实验设置。第 7.2 节随后报告关键结果，展示 LOAM 的有效性及其设计选择带来的优势。最后，第 7.3 节将回答有关 LOAM 实际贡献的最关键问题：*LOAM* 能够多广泛地 应用于 *MaxCompute* 的项目，以及在实践中能够获得怎样的性能提升？

7.1 实验设置

我们使用来自阿里巴巴集团在 MaxCompute 中的内部项目的生产工作负载，对 LOAM 查询优化器进行评估。实验设置详述如下。

评估项目。我们将项目筛选流程（第 6 节）与随机抽样相结合，构建实验数据集。首先，我们对每个 MaxCompute 项目在连续三天内收集的查询所构成的工作负载应用基于规则的筛选器，得到一个覆盖全部项目约 40.5% 的筛选集。为使评估可行并保持统计代表性，我们从该筛选集中随机选取 30 个项目用于主要实验。由于这些抽样项目可能呈现不同的性能提升，并且为了了解 LOAM 能够达到的效果，我们为每个抽样项目计算精确的改进空间$D(M_d)$（定义见第 6 节，计算细节见附录 E. 1 [25]），并选出平均 $D(M_d)$ 最高的五个项目。出于隐私限制，这些项目被匿名化为项目 1–5，并用于第 7.2.1 至 7.2.5 节的实验，以评估 LOAM 查询优化技术的有效性与效率。为保守起见，我们将这五个项目上的结果视为抽样集合的上界，并将其余 25 个项目视为低收益情形。第 7.3 节采用这一约定来估计 LOAM 在 MaxCompute 各项目上的总体收益。在实践中，精确计算 $D(M_d)$ 并不可行，因为它需要枚举并执行所有候选计划。因此，我们使用 Ranker 作为实用的替代来

表1：实验所用项目的统计信息。

数据集	项目 1	项目 2	项目 3	项目 4	项目 5
表数	253	125	348	209	229
列数	3,782	714	7,382	3,794	3,661
训练查询数	10,000	10,000	10,000	4,187	8,701
测试查询数	184	101	177	573	126
平均 CPU 成本	11,501	1,824,978	3,265	1,354	103,040

估计改进空间。其准确性与稳健性在第7.2.6节中得到验证。在 5 个评估项目中，Ranker 识别出项目 1、2 和 5。

基线。我们将 LOAM 与 MaxCompute 的原生优化器进行比较，并与其他学习型优化器变体进行比较；这些变体将 LOAM 基于 TCN 的成本预测器替换为先前研究中的其他代表性成本模型。具体而言，我们考虑基于 Transformer [33], [15], 和图卷积网络（GCN） [17], 以及 XGBoost [10], 构建的 [2]，分别如 [24],和 [2, 8, 9, 15, 27, 33, 34]所示。Transformer 和 GCN 擅长捕获复杂的执行计划结构，并已在计划成本预测方面展现出出色的 [10]性能。XGBoost [2, 27]近期也被用于估算计划成本，因为它简单、高效，且性能可与深度网络相媲美。在本节余下部分，我们将 MaxCompute 的原生优化器简称为 MaxCompute，将采用我们基于 TCN 的成本预测器的默认 LOAM 称为 LOAM，并以其底层模型名称指代其他学习型优化器基线，即 Transformer、GCN 和 XGBoost。

对于所有学习型优化器基线，我们复用 LOAM 的计划浏览器，并按照 [2], 使用 [33] 和 [15], 中 Transformer 和图卷积网络的公开源代码，以及用于 XGBoost 的标准库 [28]，实现成本预测器。为使这些基线适配 MaxCompute 并确保比较公平，我们通过以下方式调整其计划向量化：i) 移除依赖 MaxCompute 中不可用统计信息的特征；ii) 使用各自原生的特征编码方法，添加 LOAM 的额外特征集。

模型训练与评估。LOAM 和所有学习型优化器基线均针对每个评估项目分别训练。具体而言，对于给定项目，我们从其历史查询存储库中收集连续 30 天内的去重查询，并使用前 25 天的数据进行训练，剩余 5 天的数据用于测试。为公平比较，我们将训练查询的最大数量限制为 10,000。对于训练查询不足 10,000 条的项目 4 和项目 5，我们使用其所有可用查询。表1汇总了这些工作负载的统计信息。

重要的是，若没有自适应训练过程来对齐训练数据（默认计划）与测试查询计划（候选计划）的特征分布，所有学习型优化器基线都会受到分布偏移的影响。这会带来一个关键问题：在训练数据的验证集划分上进行传统的超参数调优将不再有效，因为验证性能无法再可靠地反映测试性能。为保持方法上的一致性，我们不对任何模型进行显式的超参数调优。对于 Transformer 和图卷积网络，我们采用其原始论文中的配置 [15, 33], ；对于 XGBoost，则使用标准库的默认配置 [28]。对于我们提出的、基于树卷积网络并遵循 Bao[18] 和 Lero[35], 的 LOAM，我们沿用它们的参数设置，并采用标准的优化设置：初始学习率为 0.01，每个训练轮次的指数衰减因子为 0.99。

在评估时，计划探索器会为每个测试查询生成多个候选计划，并且不失一般性，始终包含默认计划（由 MaxCompute 的原生优化器生成，未使用任何标志或缩放后的估计基数）。在这些候选计划中，学习型优化器会选择估计成本最低的候选计划。此设置反映了实际部署场景。为控制评估开销，我们根据 MaxCompute 粗略的成本估计，仅保留每个测试查询的前 5 个候选计划。我们根据学习型优化器所选计划的平均端到端（E2E）CPU 成本对其进行评估，而 MaxCompute 原生优化器的性能则通过默认计划的代价来衡量。为获得可靠的测量结果，我们会多次执行每个候选计划，并取平均成本。

环境。所有查询均在 MaxCompute 的飞行环境（见第 3 节）中执行，该环境运行于阿里巴巴内部的分布式集群，覆盖 10,000 多台机器。模型训练和推理在一台 Linux 服务器上进行，该服务器配备 Intel Xeon Platinum 8163 CPU（96 核，2.5 GHz）、768 GB DDR4 RAM、2 TB SSD 和 8 块 NVIDIA Tesla V100-SXM2 GPU。

第 7.2 节 实验结果

本节中的实验旨在回答有关 LOAM 的查询优化器和项目选择器的几个核心问题：

- 总体性能与开销：与 MaxCompute 的原生优化器及其他最先进的学习型优化器相比，LOAM 能在多大程度上提升查询执行效率？需要额外承担多少部署和运维开销？（第7.2.1节和第7.2.2节）
- 成本预测器训练：自适应训练能在多大程度上改进成本预测器（第 7.2.3 节）？训练数据量如何影响端到端性能（第 7.2.4 节）？
- 计划成本推断：与替代策略相比，我们的计划成本推断方法有多大效果？（第 7.2.5 节）
- 项目选择：Ranker 在优先选择具有较大改进空间的项目方面表现如何？（第7.2.6节）

7.2.1 ***MaxCompute*** 中的端到端性能。图 6 展示了 LOAM 和其他基线在选取的 5 个项目上的端到端性能。我们有以下观察：

1）LOAM 在几乎所有项目上的表现都优于所有基线优化器。具体而言，与 MaxCompute 的原生优化器相比，LOAM 在项目 1、2 和 5 上取得了显著提升，性能分别提高了 10%、23% 和 30%。这种几乎无一例外的优势凸显了 LOAM 在真实生产环境中的实用效果，而在这些环境中，可靠的性能尤为重要。相比之下，其他学习型优化器基线带来的改进有限，甚至不如 MaxCompute。我们认为，LOAM 的优势源于其自适应训练过程，使其能够很好地泛化到候选执行计划；而其他学习型优化器则会受到分布偏移的严重影响，导致成本估计准确性下降。

2) 在所有项目中，学习型优化器（包括 LOAM）所实现的实际性能提升，与改进空间的理论界高度相关，即，$D(M_d)$。具体而言，在项目3和项目4中，可达到的最佳值相对较小（分别为预言机模型代价的 20% 和 23%，而项目2和项目5分别为 43% 和 40%），学习型优化器的性能仅与 MaxCompute 的原生查询优化器相当。这证实了 $D(M_d)$ 是一种有效的

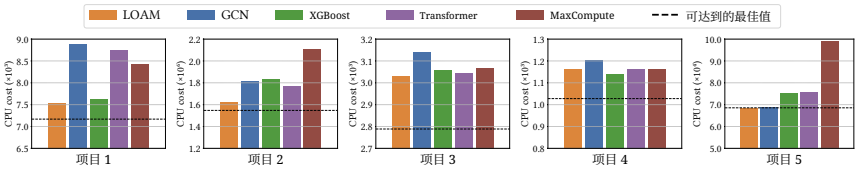


图6：学习型查询优化器与 MaxCompute 的平均 CPU 成本。

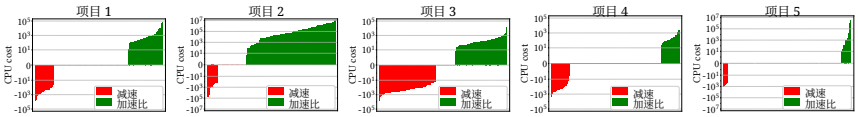


图7：LOAM 与 MaxCompute 的逐查询执行成本对比。

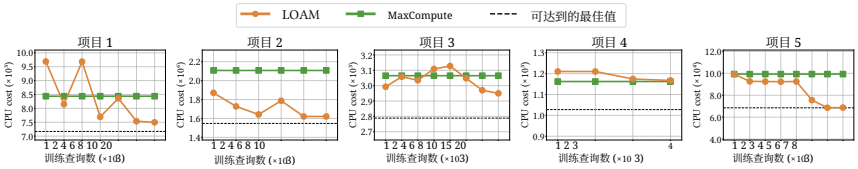


图8：LOAM相对于训练数据规模的性能。

（a）训练开销。

方法	训练时间（秒）				
	项目 1	项目 2	项目 3	项目 4	项目 5
LOAM	2,950	389	3,324	539	3,590
Transformer	1,588	898	3,645	416	129
GCN	206	103	364	115	93
XGBoost	0.12	0.07	0.14	0.05	0.11

(b) 模型占用空间。

方法	模型大小（MB）				
	项目 1	项目 2	项目 3	项目 4	项目 5
LOAM	15	8.1	21	6.4	9.1
Transformer	25.5	25.4	25.9	25.2	25.1
GCN	153	140	155	124	118
XGBoost	11	5.1	17.8	3.4	2.8

(c) 平均值 推理时。

方法	推理时（秒）				
	项目 1	项目 2	项目 3	项目 4	项目 5
LOAM	0.28	0.41	0.27	0.11	0.29
Transformer	0.26	0.16	0.31	0.15	0.14
GCN	0.21	0.24	0.17	0.06	0.20
XGBoost	0.28	0.08	0.13	0.08	0.12

图 9：学习型优化器的额外代价。

用于提前识别前景不佳项目的指标。第 7.2.5 节提供了进一步的实证证据，支持这一观察结果。与定理1一致，可实现的最佳模型（图中的虚线）在项目3和项目4上的成本偏差 $D(M_b)$ 接近 $D(M_d)$，差距约为 7%；而其他项目的差距则在 12% 至 33% 之间。

3) 项目 1 的 $D(M_d)$ 值为 25%，仅略高于项目3和项目4。尽管如此，它仍取得了显著更好的性能。这是因为项目3和项目4还受到训练数据不足的影响，从而阻碍了有效成本预测器的训练。具体而言，项目 3 包含超过 7,000 列，大约是其他项目的两倍，因此需要更多训练查询，以便 LOAM 学习数据分布。与此同时，项目 4 仅有约 4,000 条训练查询，远少于其他项目。

值得注意的是，这五个项目在全部 30 个抽样项目中展现出最大的改进空间，但由于理论和实践上的限制，其中一些仍未带来明显的性能提升。这进一步凸显了第 6 节所研究的项目选择的重要性，以及在实际场景中部署之前仔细评估学习型查询优化器性能的必要性。

尽管 LOAM 在查询优化方面性能出色，但它在 MaxCompute 中的部署开销极小，运行开销也可忽略不计。如图 9 所示，LOAM 所需训练时间不到一小时，内存占用约为 20 MB。在查询优化期间，它在不到 0.1 秒内生成候选计划，并为每条查询增加 0.1-0.5 秒的模型推理代价，仅占来自生产工作负载的测试查询执行时间的 0.23%–0.74%。

7.2.2 性能提升/回退分析。 为提供更多细节，图 7 比较了 LOAM 与 MaxCompute 的每条查询 CPU 成本。测试查询按照代价差异（从减速到加速比）排序，以直观展示 LOAM 相较于 MaxCompute 的性能提升或回退。

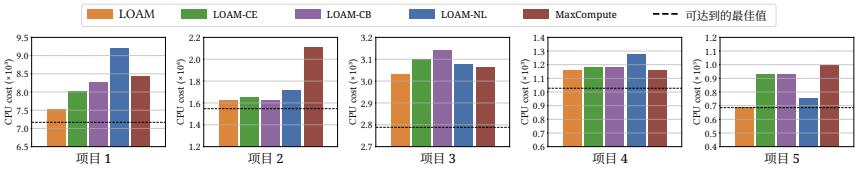
我们观察到，在项目 1 和项目 2 中，LOAM 带来的性能提升远多于性能回退。具体而言，分别只有 26 条（14%）和 8 条（8%）查询出现减速，而分别有 50 条（27%）和 70 条（69%）查询实现了显著提升。项目 5 总体基本持平，表现为 10 条查询加速、 5 条查询减速。重要的是，性能提升的幅度也远超性能回退：在这些项目中，代价节省最多比最严重的回退高出两个数量级。这种明显偏向性能提升的结果凸显了 LOAM 为复杂生产环境查询识别更优执行计划的有效性。

进一步统计发现，在这三个项目的改进查询中，超过一半的查询取得了 17% 至 26% 的相对性能提升。这表明，LOAM 带来的性能提升并非源于对长时间运行查询进行微小优化，而是在各种查询中实现了显著且广泛的改进。

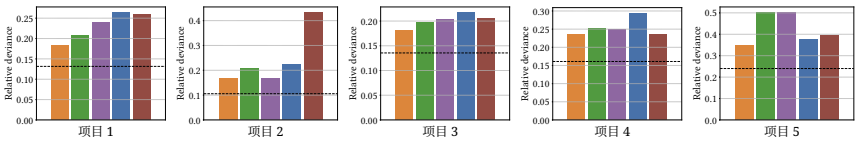
项目3和项目4的结果呈现出不同的模式：LOAM导致的性能退化略多于或与性能提升持平。所有学习型优化器基线也呈现出类似趋势。这与我们在第7.2.1节端到端评估中的发现一致：当改进空间有限且训练数据不足时，学习型优化器难以在 MaxCompute 的基础上实现明确的性能提升。这些结果再次证明，预先筛除前景不佳的项目并确保有充足的训练数据至关重要。

7.2.3 自适应训练的有效性。 我们开展了消融研究，以评估自适应训练过程的有效性。图11比较了 LOAM 与不使用自适应训练的变体 LOAM-NA 的性能。移除 LOAM 中的域分类器和梯度反转层（GRL）即可构建 LOAM-NA；该变体仅通过最小化式（1）中的成本预测损失 $L_c(\cdot)$ 进行训练。实验结果揭示了以下发现。

在项目 1、2 和 5 中，移除自适应训练会导致显著的性能下降：LOAM-NA 的表现明显



(a) 端到端 CPU 成本。

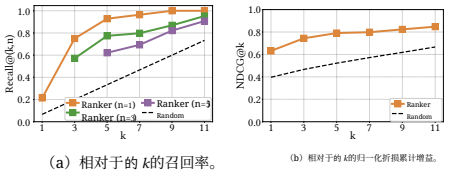


（b）相对偏差。

图 10：查询优化性能相对于成本推断方法。

方法	平均 CPU 成本				
	项目 1	项目 2	项目 3	项目 4	项目 5
MaxCompute	8,438	2,107,260	3,065	1,162	99,284
LOAM-NA	9,136	1,807,823	3,016	1,174	92,488
LOAM	7,537	1,621,243	3,029	1,162	68,617

图11：自适应训练的效果。



（a）相对于的 k的召回率。

（b）相对于的 k的归一化折损累计增益。

图12：Ranker 的性能。

逊于 LOAM，且往往与 MaxCompute 相当，甚至更差。这揭示了默认计划与旋钮调优候选计划之间的分布偏移对学习型优化器性能的严重影响。通过自适应训练，LOAM 避免了对默认计划过拟合，并能很好地泛化，从而估算候选计划的代价，这证实了自适应训练不可或缺的作用。在项目3和项目4中，LOAM-NA 和 LOAM 的表现相近，这与第7.2.1节和第7.2.2节中的发现一致：在改进空间有限且训练数据不足的情况下，任何学习型优化器都难以发挥作用。

除了已观察到的 CPU 成本节省外，自适应训练还使 LOAM 仅凭历史查询计划成本即可完成训练，大幅简化了数据准备阶段。这与以往的工作形成鲜明对比：以往的工作 [18, 19, 29, 35], 需要执行大量候选计划，收集其成本用于训练。

7.2.4 训练数据规模的影响。 在此实验中，我们考察训练数据量对 LOAM 性能的影响。我们逐步将训练集规模从 1,000 条查询增加到 MAX 条查询，其中 MAX 是 25 天内可用历史查询的最大数量，并评估 LOAM 在每种规模的数据集上训练后的性能。图 8展示了实验结果，其中揭示了以下两个关键发现：

1）在项目 1、2 和 5 上，LOAM 的性能通常会随着训练数据增多而提升，并最终趋于稳定。这一现象表明：i）训练查询不足会限制模型的收益，而增加训练数据则能带来明显优势。这一发现促使我们在第 6 节制定规则，筛除历史查询不足以训练 LOAM 的项目；ii）这些优势最终会达到上限。例如，在项目 1 上，将训练集从 10,000 条查询持续扩大到 20,000 条查询，仅带来微小的性能提升。

我们还注意到，无论如何扩展训练集，LOAM 与可实现的最佳模型（图中的虚线）之间都存在固有的性能差距。这是因为我们的项目选择在项目层面进行，并排除了平均改进空间较低的项目，但每个保留的项目中仍包含改进空间有限的查询。扩大训练集几乎无法提升 LOAM 区分此类查询的候选计划之间细微代价差异的能力。这表明我们可以为 LOAM 找到一个最佳平衡点，从而在训练开销与性能提升之间取得最佳平衡。

2）LOAM 的性能在不同项目之间差异显著，每个项目都需要不同的最少训练查询数量才能与 MaxCompute 持平。在项目 1 上，LOAM 只有在超过 6,000 条查询上训练后才能超越 MaxCompute，而在项目2和项目5上，LOAM 在所有采样的训练规模下都始终优于 MaxCompute。这源于两个因素。首先，项目 1 的改进空间（25%）远小于项目2和项目5（超过 40%）。当许多候选计划具有相近的估计成本时，LOAM 更难区分并选择更优的计划。此外，项目 1 的列数约为项目 2 的五倍，因此 LOAM 需要更多训练查询才能有效学习数据分布。在项目3和项目4上，LOAM 的性能保持相对稳定，与 MaxCompute 没有显著差异，如第7.2.1节所述。

7.2.5 我们的计划成本推断实用方法的有效性。 为了评估我们针对计划成本推断中不可见执行环境的实用解决方案，我们将 LOAM 与三个变体进行比较：a) **LOAM-CE** 将集群级环境特征建模为随机变量，其中每个特征均计算为集群中所有机器的平均值。利用过去 24 小时收集的集群范围环境数据拟合分布，并使用其期望特征值进行成本预测；b) **LOAM-CB** 同样利用集群范围环境，但使用查询优化时刻的环境实例进行成本估算；c) **LOAM-NL**从 LOAM 中移除所有环境特征，在训练和成本预测过程中均不使用任何机器负载信息。

我们从端到端 CPU 成本以及与预言机模型的偏差两个方面评估这些方法（详见第 5 节）。为便于比较，报告相对偏差，即用偏差除以预言机模型的期望成本所得的值。图 10 展示了结果。我们观察到：

1) LOAM-NL 在查询执行成本和相对偏差方面的表现始终不如 LOAM，尤其是在项目 1 上。这凸显了执行环境在查询优化中引入的显著噪声，也说明了将环境特征纳入计划成本建模的重要性，正如我们在挑战 1 和第 5 节中所强调的那样。

2) LOAM 在所有项目中的表现都优于 LOAM-CE 和 LOAM-CB 两个变体，尤其是在项目 1 和 5 中。这证实了 LOAM 策略的实际有效性：在平均情况环境条件下估算执行计划

成本。我们将其相较于使用集群级环境特征的优势归因于两个因素。首先，在执行时，查询任务通常会被调度到空闲资源较多的机器上（例如系统负载较低的机器），以实现负载均衡和高效执行。其次，来自不同项目的查询具有不同特征，会在机器上产生不同的负载模式。因此，集群级平均值往往无法反映特定查询所经历的机器级负载。相比之下，LOAM 采用从历史查询中观察到的预期机器级环境条件，能够提供更准确的近似。

3) 可实现的最佳模型在所有测试项目中的相对偏差约为 10%。这一发现为定理1提供了实证支持，并凸显了任何查询优化器与预言机模型之间固有的性能差距。此外，我们观察到，相对偏差不仅能提供更丰富的实际见解，也反映了广泛使用的端到端执行代价指标的表现，而这正是查询优化的最终目标。综上所述，相对偏差有望成为一种新颖且有效的指标，用于未来评估 MaxCompute 等真实系统中的查询优化器。

7.2.6 ***Ranker*** 的有效性。 我们使用第 7.1 节所述的 30 个抽样项目中的 28 个来评估 Ranker 的性能，另有 2 个因内部隐私原因排除。对于每个项目，我们收集由连续 5 天的历史查询构成的工作负载，并计算其改进空间（即， $D(M_d)$），用于训练和测试 Ranker。为获得稳健的结果，我们针对这 28 个项目的多个划分进行交叉验证。在每个划分中，13 个项目用于训练，其余 15 个项目由 Ranker 进行评估；Ranker 会估算这些项目的改进空间并生成排名。

由于 Ranker 旨在优先处理改进空间较大的项目，我们使用两种广泛采用的排名指标对其进行评估：i) $Recall@(k,\ n)$ [7],，用于衡量排名器优先选出的前-k 个项目中包含的 n 真实项目（即改进空间最大的项目）所占的比例；以及 ii) $NDCG@k$ [16]，该指标不仅评估前-k 项排名的质量，还会根据项目相对于真实排名的位置相关性进行考量。

为了评估 Ranker 的性能，我们将其与随机排序模型的预期性能进行比较。下文将该模型称为 Random，它会对所有测试项目进行均匀随机排列。附录 E.2 [25] 详细推导了 Random 的预期 $Recall@(k,n)$ 和 $NDCG@k$。图12展示了所有测试划分的平均结果。我们观察到，在两项指标上，Ranker 在不同的 k 值下始终显著优于 Random。这一优异表现归功于 Ranker 能够有效捕捉默认执行计划中的模式，而这些模式可能揭示优化机会。值得注意的是，在用于评估 LOAM 查询优化性能的 5 个项目中，Ranker 成功识别出其优先级最高的 5 个项目中约 3 个（即取得显著改进的项目 1、2、5）。这凸显了 Ranker 在实际应用中的有效性，能够精准找出最能从 LOAM 部署中受益的项目。此外，随着训练项目增多，预计 Ranker 的性能还会进一步提升（见附录 E.3 [25]）。

7.3 MaxCompute 中的收益

为了了解 LOAM 的实际适用性及其为 MaxCompute 带来的整体收益，我们分析了 MaxCompute 项目中能够从 LOAM 获得显著收益的比例。具体而言，我们要求端到端 CPU 成本相较于 MaxCompute 原生方案至少降低 10%。在 MaxCompute 这样成熟的生产系统中，查询优化器已经过高度优化，因此这一降幅为持续优化留下了可观空间。鉴于 MaxCompute 的计算资源消耗规模巨大，如此幅度的相对降幅可转化为显著的绝对成本节省。

请注意，我们对 MaxCompute 的项目应用 Filter，以筛除可能面临训练挑战的项目，并获得一个子集，其中有 40.5% 的项目通过了所有过滤规则。在第7.2.1节的早期实验中，我们观察到，LOAM 在随机选取的 30 个项目中的约 10%（即项目 1、2 和 5）上实现了显著的性能提升（$\geq 10\%$）。因此，我们可以合理估计，MaxCompute 中至少有 4%（$40.5\% \times 10\%$）的项目预计可通过部署 LOAM 获得 $\geq 10\%$ 的性能提升；鉴于 MaxCompute 目前拥有超过 100,000 个项目，这一比例并不低。

值得注意的是，这一估算受到当前计划探索策略的限制。这些策略非常保守，可能无法生成性能显著优于默认计划的候选计划。例如，对于项目 1 中的查询，其最优候选计划相较于默认计划，平均性能仅提升 15%。这限制了查询优化的潜力，进而制约了 LOAM 的整体性能。不过，通过采用更多样化的计划探索策略，可以大幅提高估算值。

8 结论

本文介绍了 LOAM，这是一个面向 MaxCompute 的一站式框架，旨在弥合学习型查询优化的理论进展与现代分布式多租户数据仓库中的实际部署之间的差距。LOAM 重新思考了开发学习型优化器的架构设计和系统运行范式，并且具有良好的通用性，可轻松适配其他类似的领先系统。具体而言，LOAM 采用无统计信息计划编码，以应对 MaxCompute 中可能缺少输入统计信息的情况；同时将执行时环境纳入计划成本建模。对于在线查询，它既从理论上量化了在未见过的环境变化下可实现的性能，也提供了实用的平滑策略，以减轻环境影响。在系统运行方面，LOAM 依靠自适应训练过程，在离线训练阶段预先泛化到候选计划，从而避免有风险的在线优化，同时无需为执行候选计划付出额外代价。此外，LOAM 还包含一个轻量级项目选择器，用于优先选择收益较高的部署项目。根据我们在 MaxCompute 生产工作负载上的实验，预计 LOAM 能够在 MaxCompute 相当一部分项目中显著优于原生查询优化器，从而有望在实际应用中大幅节省资源。

参考文献

[1] 阿里云。2025。MaxCompute：开展 PB 级数据仓储。https://www.alibabacloud.com/en/product/maxcompute.[2] Remmelt Ammerlaan、Gilbert Antonius、Marc Friedman、HM Sajjad Hossain、Alekh Jindal、Peter Orenberg、Hiren Patel、Shi Qiao、Vijay Ramani、Lucas Rosen-blatt 等。2021。PerfGuard：部署系统机器学习且几乎不会导致性能退化！*VLDB* 期刊论文集14, 13（2021），3362–3375。[3] Christoph Anneser、Nesime Tatbul、David Cohen、Zhenggang Xu、Prithviraj Pandian、Nikolay Laptev 和 Ryan Marcus。2023。Autosteer：适用于任何 SQL 数据库的学习型查询优化。*VLDB* 期刊论文集16, 12（2023），3515–3527。[4] Nikos Armenatzoglou、Sanuj Basu、Naga Bhanoori、Mengchu Cai、Naresh Chainani、Kiran Chinta、Venkatraman Govindaraju、Todd J Green、Monish Gupta、Sebastian Hillig 等。2022。Amazon Redshift 的重新构想。载于*2022* 年国际数据管理会议论文集。2205–2217。[5] 微软 Azure。2025。Azure Synapse Analytics。https://azure.microsoft.com/en-us/products/synapse-analytics/ 访问日期：2025年3月5日。[6] Prasanta Basak、Indrani Basak 和 N. Balakrishnan。2009。基于逐步删失数据的三参数对数正态分布估计。计算统计与数据分析 53, 10 2009 3580 3592 10 1016 2009 03 015 ( )，–。[7] 1985 doi：。/J.CSDA。。David C Blair 和 Melvin E Maron。。全文文档检索系统检索效果评估。*ACM* 通讯28, 3（1985），289–299。[8] Tianyi Chen、Jun Gao、Hedui Chen 和 Yaofeng Tu。2023。Loger：一种旨在生成高效且稳健查询执行计划的学习型优化器。*VLDB* 期刊论文集16, 7（2023），1777–1789。[9] Tianyi Chen、Jun Gao、Yaofeng Tu 和 Mo Xu。2024。GLO：迈向广义学习式查询优化。载于*2024* 年 *IEEE* 第 *40* 届国际数据工程会议（*ICDE*）。IEEE，4843–4855。[10] Tianqi Chen 和 Carlos Guestrin。2016。XGBoost：一种可扩展的树提升系统。载于第 *22* 届 *ACM SIGKDD* 知识发现与数据挖掘国际会议论文集。785–794。[11] Xu Chen、Zhen Wang、Shuncheng Liu、Yaliang Li、Kai Zeng、Bolin Ding、Jingren Zhou、Han Su 和 Kai Zheng。2023。BASE：弥合查询优化中代价与延迟之间的差距。*VLDB* 期刊论文集16, 8（2023），1958–1966。[12] Benoit Dageville、Thierry Cruanes、Marcin Zukowski、Vadim Antonov、Artin Avanes、Jon Bock、Jonathan Claybaugh、Daniel Engovatov、Martin Hentschel、Jiansheng Huang 等。2016。Snowflake 弹性数据仓库。载于*2016* 年国际数据管理会议论文集。215–226。[13] Sérgio Fernandes 和 Jorge Bernardino。2015。什么是 BigQuery？载于第 *19* 届国际数据库工程与应用研讨会论文集。202–203。[14] Yaroslav Ganin 和 Victor Lempitsky。2015。通过反向传播实现无监督域自适应。载于国际机器学习会议。PMLR，1180–1189。[15] Benjamin Hilprecht 和 Carsten Binnig。2022。用于开箱即用学习式代价预测的零样本代价模型。*arXiv* 预印本 *arXiv:2201.00561*（2022）。[16] Kalervo Järvelin 和 Jaana Kekäläinen。2017。检索高度相关文档的信息检索评估方法。载于 *ACM SIGIR*论坛，第 51 卷。美国纽约州纽约市 ACM，243–250。[17] Thomas N Kipf 和 Max Welling。2016。基于图卷积网络的半监督分类。*arXiv* 预印本 *arXiv:1609.02907*（2016）。[18] Ryan Marcus、Parimarjan Negi、Hongzi Mao、Nesime Tatbul、Mohammad Al-izadeh 和 Tim Kraska。2021。Bao：让学习型查询优化切实可行。载于*2021* 年国际数据管理会议论文集。1275–1288。

[19] Ryan Marcus、Parimarjan Negi、Hongzi Mao、Chi Zhang、Mohammad Alizadeh、Tim Kraska、Olga Papaemmanouil 和 Nesime Tatbul。2019。Neo：一种学习型查询优化器。*VLDB* 期刊论文集 12，11（2019），1705–1718。[20] Sergey Melnik、Andrey Gubarev、Jing Jing Long、Geoffrey Romer、Shiva Shivakumar、Matt Tolton 和 Theo Vassilakis。2010。Dremel：对网络规模数据集进行交互式分析。*VLDB* 期刊论文集 3，1-2（2010），330–339。[21] Parimarjan Negi、Matteo Interlandi、Ryan Marcus、Mohammad Alizadeh、Tim Kraska、Marc Friedman 和 Alekh Jindal。2021。引导查询优化器：对大数据工作负载的实用方法。载于*2021* 年国际数据管理会议论文集。2557–2569。[22] Panos Parchas、Francesco Gullo、Dimitris Papadias 和 Franceseco Bonchi。2014。追寻优质可能世界：提取不确定图的代表性实例。载于*2014* 年 *ACM SIGMOD* 国际数据管理会议论文集。967–978。[23] Hidetoshi Shimodaira。2000。通过对数似然函数加权，改善协变量偏移下的预测推断。《统计规划与推断期刊》90，2（2000），227–244。[24] Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N Gomez、Łukasz Kaiser 和 Illia Polosukhin。2017。Attention Is All You Need。神经信息处理系统进展30（2017）。[25] Lianggui Weng、Dandan Liu、Wenzhuang Zhu、Rong Zhu、Junzheng Zheng、Bolin Ding、Zhiguo Zhang 和 Jingren Zhou。2025。阿里巴巴 MaxCompute 中的学习型查询优化器：挑战、分析与解决方案。https://drive.google.com/drive/folders/1B0tqOLo2Jt6aBFxaPfjbV0B7sagFPT5u。[26] Lucas Woltmann、Jerome Thiessat、Claudio Hartmann、Dirk Habich 和 Wolfgang Lehner。2023。Fastgres：让学习型查询优化器提示发挥作用。*VLDB* 期刊论文集 16，11（2023），3310–3322。[27] Ziniu Wu、Ryan Marcus、Zhengchun Liu、Parimarjan Negi、Vikram Nathan、Pascal Pfeil、Gaurav Saxena、Mohammad Rahman、Balakrishnan Narayanaswamy 和 Tim Kraska。2024。Stage：Amazon Redshift 中的查询执行时间预测。载于*2024* 年国际数据管理会议伴随论文集。280–294。[28] XGBoost 开发者。2022。XGBoost 文档 — xgboost 3.0.0 文档。https://xgboost.readthedocs.io/en/release_3.0.0/。[29] Zongheng Yang、Wei-Lin Chiang、Sifei Luan、Gautam Mittal、Michael Luo 和 Ion Stoica。2022。Balsa：无需专家示范即可学习查询优化器。载于*2022* 年国际数据管理会议论文集。931–944。[30] Xiang Yu、Chengliang Chai、Guoliang Li 和 Jiabin Liu。2022。基于代价还是基于学习？一种用于查询计划选择的混合查询优化器。*VLDB* 期刊论文集 15，13（2022），3924–3936。[31] Wangda Zhang、Matteo Interlandi、Paul Mineiro、Shi Qiao、Nasim Ghazanfari、Karlen Lie、Marc Friedman、Rafah Hosn、Hiren Patel 和 Alekh Jindal。2022。在微软生产环境中部署引导式查询优化器。载于*2022* 年国际数据管理会议论文集。2299–2311。[32] Zhuo Zhang、Chao Li、Yangyu Tao、Renyu Yang、Hong Tang 和 Jie Xu。2014。Fuxi：互联网规模下的容错资源管理与作业调度系统。载于*VLDB* 期刊论文集，第 7 卷。VLDB 期刊论文集公司，1393–1404。[33] Yue Zhao、Gao Cong、Jiachen Shi 和 Chunyan Miao。2022。Queryformer：用于查询计划表示的树形 Transformer 模型。*VLDB* 期刊论文集 15，8（2022），1658–1670。[34] Kai Zhong、Luming Sun、Tao Ji、Cuiping Li 和 Hong Chen。2024。FOSS：一种用于查询优化器的自学习医生。载于*2024* 年 *IEEE* 第 *40* 届国际数据工程会议（*ICDE*）。IEEE，4329–4342。[35] Rong Zhu、Wei Chen、Bolin Ding、Xingguang Chen、Andreas Pfadler、Ziniu Wu 和 Jingren Zhou。2023。Lero：一种学习排序查询优化器。*VLDB* 期刊论文集 16，6（2023），1466–1479。

附录

MaxCompute 与类似系统的比较

Google BigQuery [13],Microsoft Azure Synapse Analytics [5], Snowflake Serverless [12], 和 Amazon Redshift Serverless [4],等领先的工业级分布式多租户数据平台，在设计理念和各自生态系统的优化方面各不相同。尽管如此，它们遵循与 MaxCompute 类似的查询执行工作流（如图 1 所示），并且在有效部署学习型优化器时面临着共同的挑战。

常见的查询执行工作流。当查询提交至此类系统时，首先会由基于专家精心设计的成本模型和优化规则构建的原生基于成本的优化器，将其转换为经过优化的物理计划。为便于调度并实现高效执行，这些计划随后会表示为粗粒度树（或更一般地说，有向无环图），其中的顶点代表机器内流水线，作为执行单元运行，类似于 MaxCompute 中的阶段。这些系统通过不同的计算资源抽象和分布机制支持分布式执行。例如，BigQuery 使用 Dremel [20] 引擎，以称为“slot”的单位分配资源（通常为半个 CPU 核心和 1 GB 随机存取存储器）；Snowflake 的引擎则依赖弹性计算集群模型，可在执行期间自动调整资源。尽管存在这些差异，资源始终会动态分配给每个未阻塞的执行单元，并对执行性能产生重大影响。所有执行完成后，相关信息会记录在日志中，以供日后参考和分析。

MaxCompute面临的常见挑战。在多租户系统中，用户通常会维护各自的数据库实例，并提交特征各异的工作负载。鉴于用户工作负载数量众多，且我们观察到学习型优化器难以让所有工作负载都同等受益，因此，首先必须解决如何以自动且高效的方式，选择适合采用学习型查询优化器的工作负载这一问题，这与 MaxCompute 中的挑战 4（项目选择）类似。

针对给定的用户工作负载开发学习型查询优化器时，缺少有用的输入统计信息这一问题仍是一个重大障碍，对应于 MaxCompute 中的挑战 2（有用统计信息缺失）。在这类系统中，生产环境的庞大规模数据（例如，数据量从太字节到拍字节不等）以及频繁的数据修改，使得实时维护统计信息（例如，属性直方图）极具挑战性。作为替代方案，这些系统依赖手动触发来更新统计信息，或在查询优化期间基于采样按需计算统计信息。然而，手动触发往往无法确保统计信息及时更新并反映底层数据分布。同时，基于受控数量的采样记录（例如，少于数千条记录 [4]）计算出的统计信息方差较大，准确性也远远不足。除了这一挑战之外，由于动态资源调度和分布式计算，这些系统中查询执行成本的高方差也不可避免，因此挑战 1（环境变化）同样存在。

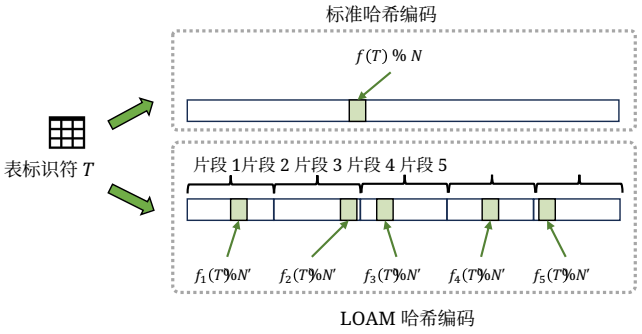


图13：LOAM 中表标识符编码示意图。

最后，在系统运行过程中，鉴于其所针对的 OLAP 查询具有资源密集型特征，且生产级系统对稳定性和可靠性有着严格要求，学习型优化器的传统细化流程完全不可接受。该流程依赖额外的计算资源来执行候选计划，或者部署一个仅经过部分训练的优化器，在其进行长期在线优化的同时为用户查询提供服务。因此，MaxCompute 中的挑战 3（传统细化方式不可行）也必须在此类系统中得到解决。

这些系统与 MaxCompute 之间的共性带来了跨系统迁移的机会：LOAM 中针对 MaxCompute 开发的技术，以及本文介绍的相关经验和见解，都可以作为极具价值的参考，并且能够轻松适配到其他系统。这极大地提升了 LOAM 的实践贡献，使 LOAM 成为一个基础框架，能够推动各种工业级分布式多租户平台中的学习型查询优化发展。

B 自适应成本预测器补充内容

本节介绍自适应成本预测器设计中缺失的细节。

B.1 表标识符和列标识符的哈希编码

在 MaxCompute 的生产工作负载中，表和列的数量可能极其庞大（见表1），因此由于独热编码（或多热编码）可能导致维度爆炸，使用这类编码并不实际。为了解决这一问题，LOAM 基于经典的哈希编码技术，将无界的标识符集合高效映射到具有受控维度的固定大小特征空间中。下面我们重点介绍表标识符的编码方式；列标识符的处理方式与之类似。

令 T 表示表标识符的空间，N 表示特征维度。形式上，标准哈希编码通过哈希函数 $f: \mathrm{T} \rightarrow \mathbb{N}$ 将给定的表标识符 T 映射为一个整数，然后将其转换为一个 N 维二进制向量，其中只有第 $(f(T)\%N)$ 个位置设为 1。

然而，这种标准哈希编码会受到严重的哈希冲突影响，从而限制其可扩展性。例如，使用原生

50 维哈希编码，在碰撞变得极有可能发生之前，只能唯一表示约 50 个不同的表。为解决这一局限，LOAM 引入了一种大幅改进的变体：如图13所示，每个表标识符都被编码为一个 $5 \times N'$ 维向量，其中每个 N' 维片段均通过对表标识符应用独立的哈希函数 $f_i(T)$ 生成，并将第 $(f_i(T)\%N')$ 个位置设为 1。通过在不同片段中使用多个哈希函数，该方法在不牺牲特征紧凑性的前提下，大幅降低了发生碰撞的概率。例如，使用 $N' = 10$ 时，总计 50 维的编码可可靠地编码多达 10^5 个不同的表标识符，远超原生方法的能力。此外，我们的方法还可自然扩展为同时编码多个标识符（例如，过滤算子中的所有列），只需合并它们各自的编码即可。

B.2 机器负载指标

我们将详细介绍 LOAM 执行计划编码中用于建模机器负载的四个指标：

• CPU_IDLE：CPU 保持空闲并可用于新任务的时间百分比。

• IO_WAIT：CPU 用于等待 I/O 操作完成的时间百分比。• 5 分钟系统负载：过去五分钟内的平均系统负载（即使用或等待 CPU、磁盘等资源的进程数量）。

• MEM_USAGE：当前正在使用的内存百分比。

定理1的 C 证明

证明。（定理1）我们首先证明，对于任意模型 M，都有 $\mathbb{E}[D_E(M)] \geq \mathbb{E}[D_E(M_b)]$。根据定义，$M_b$ 选择使期望成本 $\mathbb{E}[C_E(P)]$ 最小的执行计划 P_{M_b}。因此，我们有 $\mathbb{E}[C_E(P_M)] \geq \mathbb{E}[C_E(P_{M_b})]$。从两边减去 $\mathbb{E}[C_E(P_{M_o})]$，得到 $\mathbb{E}[C_E(P_M) - C_E(P_{M_o})] \geq \mathbb{E}[C_E(P_{M_b}) - C_E(P_{M_o})]$,，这直接推出 $\mathbb{E}[D_E(M)] \geq \mathbb{E}[D_E(M_b)]$。

为证明 $\mathbb{E}[R_E(M_b)] > \mathbb{E}[R_E(M_o)] = 0$，注意到 M_o 始终选择在任意环境实例 e 下使代价最小的执行计划 P_{M_o}，并且其偏差值为 0。因此，有 $C_e(P_{M_b}) \geq C_e(P_{M_o})$。对所有环境取期望，可得 $\mathbb{E}[R_E(M_b)] = \mathbb{E}[C_E(P_{M_b}) - C_E(P_{M_o})] \geq 0 = \mathbb{E}[D_E(M_o)]$。综上，定理得证。□

D ProjectSelector 的补充说明

本节介绍项目选择器设计中遗漏的细节。

D.1 Filter 中的规则

在当前的 Filter 中，我们从训练查询的数据量和底层表的稳定性两方面限制项目，以减轻潜在的训练挑战。具体而言：

1) 第 7.2.4 节中的实验表明，用于训练自适应成本预测器的历史查询量会强烈影响查询优化性能。为确保项目拥有足够的历史查询以进行稳健的模型训练，我们在由连续 d 天收集的历史查询组成的采样工作负载 Q 上定义以下两个指标：

• n_查询(Q) = $|Q_d|$。该指标用于计算每天提交的查询平均数量。

• 查询_增_比率(Q) = $\frac{1}{d-1}\sum_{i=2}^{d} \frac{|Q_i|}{|Q_{i-1}|}$，其中 $Q_i \subseteq Q$ 表示在第 i 天（$1 \leq i \leq d$）提交的查询集合。该指标用于衡量连续几天内查询数量的增长率。

query_inc_ratio(Q) 的值接近或大于 1，表明已提交查询的数量保持稳定或稳步增长，因此 n_query(Q) 可作为总体查询量的可靠指标。基于这两个指标，我们协同应用以下两条规则：（R1）n_query(·) $\geq N_0$；以及（R2）query_inc_ratio(·) $\geq r$。如第 7.2.4 节所述，包含至少 10,000 个查询的训练集通常足以实现良好的查询优化性能。因此，在实现中，我们将 $N_0 = 2,000$ 和 r 设置为满足 $N_0 \times r^{30} \geq 10{,}000$ 的最小数量。

2）由于 LOAM 采用的无统计信息计划编码依赖算子语义以及涉及相同表的历史查询，来粗略推断数据级细节，因此，在一个项目中频繁创建和删除表——例如，出于数据分析目的而临时创建的表——可能会妨碍 LOAM 有效利用历史查询中的信息来预测未来查询的计划成本。为了提高可预测性，我们鼓励工作负载涉及更多长期存续表，即也就是生命周期超过 n 天的表。为此，我们引入以下指标，用于评估涉及此类长期存续表的查询比例：stable_table_ratio(Q) = $|\{Q \in Q \mid \forall t \in T_Q, \text{LifeSpan}(t) > n\}|$，其中 T_Q 表示查询 Q 所涉及的表集合，LifeSpan(t) 表示表 t 的生命周期。基于该指标，将规则（R3）stable_table_ratio $\geq \theta$ 纳入 Filter。在当前实现中，我们设置 $n = 30$ 和 $\theta = 0.2$。

值得注意的是，这些规则可以动态修改、添加或移除，以更好地适应 LOAM、用户工作负载和系统要求不断变化的特征。这种灵活性确保 Filter 能够长期保持适应性和有效性。

D.2 Ranker 的计划向量化

根据我们在第 6 节中的观察，查询 Q 的改进空间（即，$D(M_d)$）通常可以通过 Q 的默认执行计划 P_d 的可观测属性来体现，包括其结构、输入特征和执行成本，Ranker 通过高效编码以下方面来表示 Q：

1) 的结构。我们采用一个（$1 + n$）维向量来表示 P_d的结构信息，其中第一个位置表示 P_d中算子的总数，其余 n 个位置则编码结构细节。具体而言，我们将每一种不同的父子算子组合父，子视为一种模式，并对每种模式的出现次数进行编码。例如，如图 14 所示，第二和第三个位置分别表示模式 MJ，TS和HJ，MJ的计数，数值分别为 2 和 1。这种方法简化了整体执行计划结构的表示，同时仍能突出可能揭示优化空间的关键结构，例如嵌套连接算子。

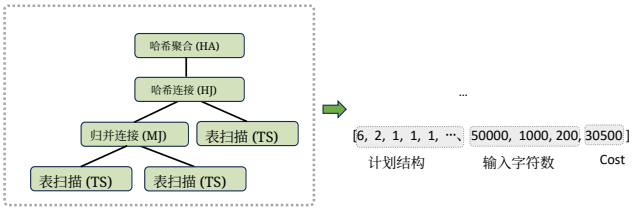


图 14：Ranker 的计划向量化示意图。

值得注意的是，这种编码比单纯统计算子数量包含的信息丰富得多。例如，#HA，MJ $=1$ 可能表明，通过调换 MJ 和 HA 操作的顺序存在潜在的优化机会，而像{#HA : 1，#MJ : 1}这样的编码则无法体现这一点。

2) 输入表的大小。我们用一个向量表示 P_d 中涉及的表的大小（即记录数），其中每个位置对应某个表的大小。这使 Ranker 能够识别输入规模倾斜可能带来优化机会的情况，例如，当大表与小表连接时，引导原生查询优化器先从小表生成谓词，对大表进行过滤，再执行连接，将显著提升连接性能。由于不同查询涉及的表数量不同，我们的实现仅使用一个三维向量表示最大的 3 个表的大小。

3) 的代价。我们还使用单个数值表示 P_d的 CPU 成本，该数值可能反映出不理想的计划选择，例如较差的连接顺序，从而也能揭示优化机会，正如第 6 节所述。

为确保缩放一致，所有特征值均采用最小-最大归一化进行归一化。

E 实验补充材料

本节介绍实验评估中缺失的细节。

E.1 偏差估计

为了估计给定模型 M的偏差，我们首先基于执行计划成本的分布从理论上推导其期望值，然后介绍如何通过采样和分布拟合在实践中对其进行估计。

理论推导。回顾第 5 节，对于查询 Q及其候选计划 $\mathrm{P}_Q=\{P_1, P_2, \cdots P_n\}$，执行计划 M在环境 E下相对于 Q的偏差定义为 $D_E(M)=C_E(P_M)-C_E(P_{M_o})$。

假设已知每个执行计划 $P\in\mathrm{P}$ 的执行成本 $C_E(P)$ 的分布。为简化记号，我们分别用 f_C 和 F_C表示执行成本随机变量 C 的概率密度函数（PDF）和累积分布函数（CDF）。为推导 $D_E(M)$ 的期望值，我们首先将 $D_E(M)$ 改写如下：

$$D_E(M)=\begin{cases}C_E(P_M)-C_E^* & \text{if } C_E^*\le C_E(P_M)\\ 0 & \text{else}\end{cases},$$

其中，C_E^* 是一个随机变量，表示 $\mathrm{P}_Q\setminus\{P_M\}$ 中各计划的最小执行成本。该公式的动机如下：如果 P_M 与预言机模型 M_o 在某些环境实例下选出的执行计划一致，则不会产生偏差，即，$D_E(M)=0$。否则，C_E^*对应于 P_{M_o}的执行成本，因此，根据定义，$D_E(M)=C_E(P_M)-C_E^*$。

由此，$D_E(M)$ 的概率密度函数可表示为：

$$f_{D_E(M)}(z)=\begin{cases}f_{C_E(P_M)-C_E^*}(z) & \text{if } C_E^*\le C_E(P_M)\\ 0 & \text{else}\end{cases},$$

其中，$f_{C_E(P_M)-C_E^*}(z)$ 是 $C_E(P_M)$ 与 C_E^*之差的概率密度函数，可通过卷积计算：

$$f_{C_E(P_M)-C_E^*}(z)=\int_{-\infty}^{\infty} f_{C_E(P_M)}(x)f_{C_E^*}(x-z)\,dx.$$

因此，期望偏差 $D_E(M)$ 可计算为：

$$\mathbb{E}[D_E(M)]=\int_0^{\infty} z f_{D_E(M)}(z)\,dz, \tag{2}$$

其中，积分下限为 0，因为 $D_E(M)\ge 0$。

请注意，要计算该积分，需要知道 C_E^*的分布。我们将在以下引理中推导其概率密度函数。

引理 1. C_E^* 的概率密度函数为

$$f_{C_E^*}(x)=\sum_{C\in\mathrm{C}} f_C(x)\prod_{\substack{C'\in\mathrm{C},\\ C'\neq C}}[1-F_{C'}(x)],$$

其中 $\mathrm{C}=\{C_E(P_i)\mid P_i\in\mathrm{P}_Q\setminus\{P_M\}\}$。

证明。根据定义，$C_E^*=\min\mathrm{C}$。因此，C_E^*的累积分布函数可推导如下：

$$\begin{aligned}F_{C_E^*}(x)&=P(C_E^*\le x)=1-P\Big(\bigcap_{C\in\mathrm{C}}\{C>x\}\Big)\\&=1-\prod_{C\in\mathrm{C}}P(C>x)\ (C\in\mathrm{C}\text{ is independent})\\&=1-\prod_{C\in\mathrm{C}}[1-F_C(x)].\end{aligned}$$

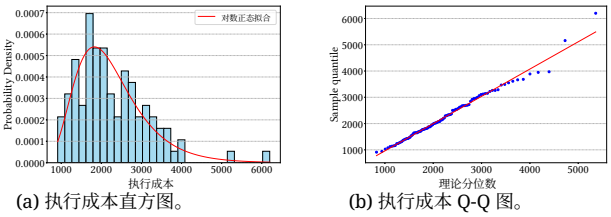
对 $F_{C_E^*}(\cdot)$ 求导，可得 C_E^*的概率密度函数：

$$\begin{aligned}f_{C_E^*}(x)&=\frac{d}{dx}F_{C_E^*}(x)\\&=\frac{d}{dx}\left(1-\prod_{C\in\mathrm{C}}[1-F_C(x)]\right)\\&=\sum_{C\in\mathrm{C}} f_C(x)\prod_{\substack{C'\in\mathrm{C}\\ C'\neq C}}[1-F_{C'}(x)].\end{aligned}$$

因此，引理得证。 □

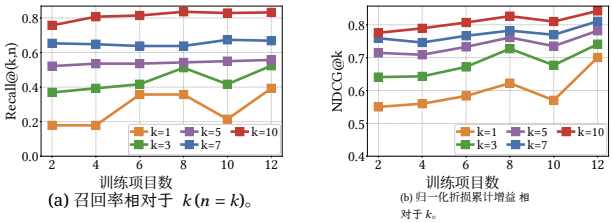
实际估计。虽然给定所有计划 $P\in\mathrm{P}_Q$ 的执行代价分布 $C_E(P)$，可以通过公式 (2) 从理论上计算 $\mathbb{E}[D_E(M)]$，但由于存在众多不可控的运行时因素（例如资源争用），在真实系统中推导精确的分布 $C_E(P)$ 并不可行。因此，我们采用一种系统化的经验方法，分两个阶段估计这些分布：

1)分布建模。为了根据经验刻画查询计划执行成本的分布，我们收集了 MaxCompute 中一组定期执行的重复查询的历史执行成本。对这些样本进行统计分析后发现，执行成本呈现出显著的对数正态模式。图15(a)展示了一个具有代表性的查询计划的代价直方图及其拟合的



(a) 执行成本直方图。 (b) 执行成本 Q-Q 图。

图 15：示例查询计划的成本分布。



(a) 召回率相对于 $k\,(n=k)$。 (b) 归一化折损累计增益 相对于 k。

图 16：Ranker 关于训练项目数量的性能。

对数正态曲线，而图15(b)则展示了相应的分位数-分位数（Q-Q）图。两幅图均表明，经验成本分布与对数正态模型高度吻合。为正式验证这一观察结果，我们对所有收集到的查询计划进行了柯尔莫哥洛夫–斯米尔诺夫检验。平均 p 值约为 0.6，这进一步说明执行成本与理论对数正态分布之间不存在统计显著差异。基于上述观察和结果，我们将每个执行计划的执行成本建模为服从对数正态分布的随机变量。需要注意的是，前文提出的理论框架仍适用于由不同工作负载特征和系统环境决定的任意成本分布。

2) 参数估计。虽然查询计划的执行成本通常服从对数正态分布，但其参数（例如均值和方差）可能因计划结构不同以及系统环境的影响而在不同计划之间存在显著差异。为了估计每个计划 $P \in \mathrm{P}_Q$ 的具体成本分布，我们多次执行 P，并采用 [6] 中描述的最大似然估计方法，根据获得的执行成本拟合参数。

E.2 随机模型的排序指标

在第 7.2.6 节中，我们将 Ranker 与随机模型 Random 进行比较。该模型会对所有测试项目生成均匀随机排序（排列），记为 r。在排序 r 中，每个项目出现在任意位置的概率均为 $1/N$，其中 N 为测试项目总数。接下来，我们推导 Random 的指标 $Recall@(k,\ n)$ 和 $NDCG@k$ 的期望值。

$\mathbb{E}[Recall@(k,\ n)]$：$Recall@(k,\ n)$定义为 Random 优先排序的前 k 个项目中包含的 n 个真实项目所占的比例。由于每个项目出现在 r 任意位置的概率都相同，因此特定项目出现在前 k 个位置的概率为 k/N。因此，在这 n 个真实项目中，位于前 k 个位置的项目数的期望值为 $n\cdot k/N$。由此可得 $\mathbb{E}[Recall@(k,\ n)] = \left(n \times \frac{k}{N}\right)/n = k/N$。

$\mathbb{E}[NDCG@k]$：$NDCG@k$ 评估 r的前k 名排序（记为 r_k）的质量，其方法是考虑项目与真实排序相比。其计算公式为：

$$NDCG@k = \frac{DCG@k}{IDCG@k},$$

其中

$$DCG@k = \sum_{i=1}^{k} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

用于衡量 r_k 的折损累计增益，而 rel_i 表示位置 i 处项目的相对得分（即，改进空间 $D(M_d)$）；$IDCG@k$ 是真实排序中 $DCG@k$ 的理想值。为计算期望值 $\mathbb{E}[NDCG@k]$，注意到 $IDCG@k$ 独立于 r，且对于固定的测试项目集合保持不变，因此有 $\mathbb{E}(NDCG@k) = \frac{\mathbb{E}[DCG@k]}{IDCG@k} = \frac{1}{N}\sum_{i=1}^{k} 2^{r_i-1}$，其中 $\mathbb{E}[DCG@k] = \sum_{i=1}^{k} \frac{\mathbb{E}[2^{rel_i}-1]}{\log_2(i+1)} = \sum_{i=1}^{k} \frac{1}{\log_2(i+1)}$。

E.3 排序性能相对于训练规模

我们考察训练项目的数量如何影响 Ranker 的性能。按照第7.2.6节中的实验设置，我们通过将 28 个可用项目划分为多个测试配置来实施交叉验证。在每种配置中，我们固定 15 个项目用于测试，并将训练项目的数量从 2 个逐步增加到 12 个。所有配置的平均结果如图16所示。我们得到以下观察结果：

1) 即使训练数据极少（例如仅有两个项目），LOAM 仍能显著且稳健地优于 Random 随机基线（参见第7.2.6节中的图12）。这表明，即使在数据受限场景下，Ranker 也能有效学习并捕捉默认计划中的基本模式，从而发现查询优化机会。然而，纳入更多训练项目可带来显著的性能提升，例如，$NDCG@1$ 会在训练项目数量从 2 个增加到 12 个时，从 0.55 提升至 0.7。这证实了增加训练项目对于提升 Ranker 性能至关重要。

2）此外，尽管存在轻微波动，随着训练项目数量增加，LOAM 在两项指标上的性能始终有所提升。这凸显了 Ranker 的良好潜力：随着更多训练项目的可用，性能有望持续提升。