QTrail-DB：面向质量演化的不完美数据库查询处理引擎

Maha Asiri, Mohamed Y.Eltabakh伍斯特理工学院计算机科学系（*WPI*），美国马萨诸塞州。{*mmasiri, meltabakh*}*@wpi.edu*

关键词：不完美数据库、数据质量、质量传播、查询优化。

arXiv:2303.06720v1 [cs.DB] 12 Mar 2023

摘要：由于种种原因，不完美数据库在许多应用中十分常见，这些原因包括数据录入错误、传输或集成错误、仪器读数错误，以及导致结果不正确的实验设置缺陷。管理和查询处理不完美数据库是一项极具挑战性的工作，因为这要求在数据库引擎中纳入数据质量。更具挑战性的是，质量通常并非静态，而是可能随时间演化。遗憾的是，大多数最先进的技术都将数据质量问题视为离线任务，与查询处理引擎完全隔离（在数据库管理系统之外进行）。因此，终端用户在收到查询结果时，无从得知这些结果是否可信，能否用于进一步分析和决策。本文提出了“QTrail-DB”系统，从根本上扩展标准数据库管理系统，以支持质量不断演化的不完美数据库。QTrail-DB 基于“质量轨迹”这一新概念，引入了一种新的质量模型，用于捕捉数据质量随时间的演化。QTrail-DB 扩展了关系数据模型，将质量轨迹纳入数据库系统。我们提出了一种名为“*QTrail*代数”的新查询代数，可在查询流水线中无缝、透明地传播和推导数据质量。因此，查询结果将自动附带元组级别的质量相关信息。QTrail-DB 的传播模型利用了数据库溯源和血缘追踪文献中经过深入研究的传播语义，因此无需开发新的查询优化器。QTrail-DB 基于 PostgreSQL 开发，并使用真实世界数据集进行了实验评估，以展示其效率和实用性。

1 引言

在大多数现代应用程序中，工作数据库可能并不完美，且可能包含低质量数据记录，这几乎已是不争的事实（Batini & Scannapieco，2006；Redman，1998；Zimnyi & Pirotte，1997）。低质量数据的出现有诸多原因，包括值缺失或错误、来自多个来源的冗余信息和冲突信息、数据录入中的人为错误、机器及网络传输错误，甚至科学实验期间的错误假设或仪器校准不当，这些都会导致结果不准确。近期一项科学调查显示，80.3%的参与调查的研究和科学组承认，其工作数据库中包含质量低下的记录，使其分析和探索面临风险（Twombly，2011）。此外，文献（Eckerson，2002）报告称，不完美数据库导致的错误决策和缺乏依据的分析每年给美国企业造成约6000亿美元的损失。

更具挑战性的是，数据元组的质量通常并非静态不变，而是可能随着时间推移（不断演变），取决于数据库中发生的各种事件。新兴的科学应用是其中的典型例子，在这些应用中，跟踪并维护数据的质量至关重要（Radziwill，2006；Twombly，2011）。例如，图1展示了生物数据库中可能发生的一系列操作。首先，一个数据元组r（例如，一个基因元组）可以从外部来源导入本地数据库。此时，将根据来源的可信度为r分配初始质量分数。随后，科学家可能添加一条评论，指出该元组中可能存在错误（例如，基因起始位置似乎不正确），据此应降低r的质量。过了一段时间，一项将本地数据与外部

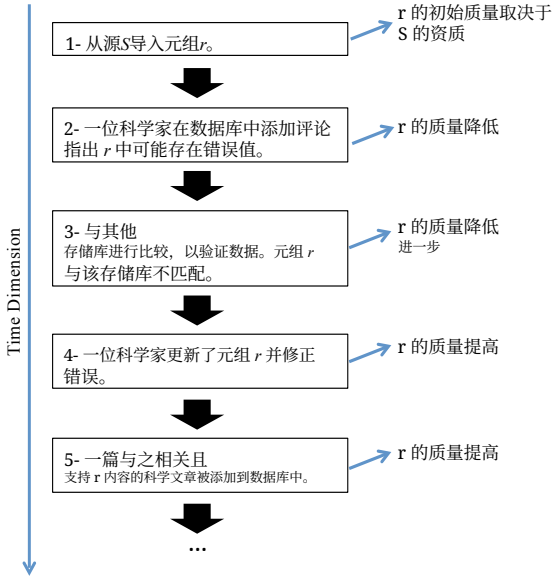


图1：质量随时间演化的数据库元组。

存储库进行比较的验证步骤可能确认r 包含错误值，从而进一步降低r的质量。数据库中的后续操作可能会随着时间推移提高或降低r的质量，例如图1中的步骤4和步骤5：对r 执行更新操作（例如，纠正基因起始位置），以及添加一篇与r的新内容相匹配的科学文章；这两项操作都应提升r的质量。一般而言，数据库中的每个元组都可能基于数据库中发生的不同操作，随着时间推移而改变其质量和可信度。

在这类质量随时间动态演化的不完美数据库中，忽略元组质量而将所有元组一视同仁的标准查询处理，确实是一种非常有限的方法。例如，除了标准数据查询之外，还可能出现一些有趣且具有挑战性的问题，包括：

1. 元组r 在最后一次修订之前的质量如何？2. 为什么r的质量在t时大幅下降？我们采取了哪些措施来修复这一问题？3. 对于我的复杂查询，例如涉及选择、连接、分组、聚合和集合运算符的查询，每个输出元组的质量如何？我能否相信这些结果，并据此开展进一步分析？4. 对于我的查询，如何仅在高质量元组上执行，例如仅处理质量高于某一阈值的元组？如何根据元组的质量进行连接、选择或排序？

5. 在数据库中的低质量元组中，哪些元组更为重要，例如经常参与查询结果的元组，应优先进行调查？

显然，支持这类问题对终端用户和高层应用至关重要。一方面，如果不以系统化的方式对质量信息进行建模并持续跟踪，关键信息就会丢失。另一方面，如果不评估输出结果的质量，科学家和决策者就会对所得结果缺乏信心。因此，基于低质量数据开展进一步分析，不仅可能导致错误决策，还会浪费科学家的努力、资源和预算（Haug、Zachariassen 和 van Liempd，2011；Redman，1998年）。

显然，支持这类问题需要对底层数据库管理系统进行根本性改造。本文提出*“QTrail-DB”*系统，一种面向质量不断演变的不完美数据库的先进查询处理引擎。我们确定了需要解决的三项主要任务：

1.1 任务 1–演变质量的系统化建模

随着现代数据库规模不断扩大，即使低质量数据所占比例很小，也可能对应数量极其庞大的低质量记录。因此，要即时识别、隔离或修复这些记录，将变得极具挑战性且耗时。这意味着底层数据库引擎必须能够以系统化的方式捕获并建模数据质量，同时持续跟踪其随时间的演化，例如质量在何时以及为何发生变化（参见前述问题 1 和 2 ）。

1.2 任务 2–质量传播与查询结果评估

假设应用程序可以冻结其工作数据库，直到所有记录都得到修正，然后再启用数据库供查询，这并不现实。这是一个持续收集和生成不同质量水平数据的过程——期间可能穿插着离线工作，以验证和修正不完美元组。因此，在元组质量各不相同的情况下查询数据不可避免。于是，查询处理引擎必须扩展功能，不仅要处理数据值，还要处理与之相关的质量。输出结果中的每个元组 r 都应

根据参与 r 计算的输入元组推断并导出其质量（参见前文提到的问题 3）。

1.3 任务 3–质量驱动的处理与整理

另一种重要的处理类型——除了传播和推导输出的质量之外——是能够根据数据的质量查询数据，即质量驱动的处理。例如，这包括根据数据元组的质量选择、连接或排序数据元组，并可能在单个查询计划中将此类质量驱动的处理与标准查询运算符结合起来（参见前文提到的问题4）。另一种分析类型是质量驱动的整理，终端用户可能希望通过这种方式，例如，追踪低质量元组如何影响查询结果，或根据低质量元组参与查询的情况对其进行排序，以便调查和修正（参见前文提到的问题5）。要实现这种类型的质量驱动处理，必须对数据库引擎进行核心改造。

本文将重点讨论前两项任务（任务1和任务2），而将任务3留待未来工作。QTrail-DB提出将数据的质量全面集成到数据库管理系统的所有层中。这种集成包括引入一种新的质量模型，用于捕捉每个数据元组随时间演化的质量，称为“质量轨迹”；扩展关系数据模型以涵盖质量轨迹；并提出一种新的关系代数，称为*“QTrail*代数”，以便在查询流水线中无缝、透明地传播和推导数据的质量。因此，复杂查询计划中的每个输出元组都将根据贡献输入元组，为其标注推导和推断出的质量。此外，我们还表明，QTrail-DB扩展不会改变查询优化规则，因此仍可使用标准优化器。

本文的主要贡献总结如下：

- 提出“QTrail-DB”系统，将数据的质量视为关系数据库中的组成部分。与现有相关工作（例如离线质量管理和通用溯源跟踪系统）相比，QTrail-DB 首次对数据质量进行量化和建模，并将其完整地整合到数据处理周期中。（第2节）
- 介绍一种基于“质量轨迹”这一新概念的新质量模型，该模型能够捕捉每个数据元组随时间演变的质量历史。该新模型支持对质量轨迹进行高级处理和查询，这是传统单一分数质量模型无法实现的。（第3节）
- 提出一种新的查询代数，称为*“QTrail*代数”，扩展标准查询操作符的语义，以流水线式且透明的方式处理和传播质量轨迹。每个查询算子都会为其每个输出元组标注质量轨迹，该质量轨迹由贡献输入元组推断并派生得到。我们还提出了新的质量专用算子，并将其整合到其他查询算子中。（第4节）
- 研究将关系数据库强大的查询优化器应用于 QTrail-DB 的可行性。我们证明，在新的 QTrail代数下，并遵循基于溯源的技术所采用的传播语义，标准等价规则和变换规则仍然适用。（第5节）
- 在 PostgreSQL 引擎中开发 QTrail-DB 原型系统，并使用真实世界的生物数据集评估其性能。考虑到新增的增值功能，与传统数据库的标准查询执行相比，QTrail-DB 的开销处于可接受范围内。（第 6 节和第 7 节）

2 相关工作

鉴于数据质量至关重要，学术界已对其进行了广泛研究。与我们工作最相关的研究如下。

2.1 清理与修复技术

一项主要研究方向是数据清洗、修复与净化：识别潜在的低质量数据记录，并加以修正（Bohannon, Fan, Flaster, & Rastogi, 2005; Bohannon, Fan, Geerts, Jia, & Kementsietsidis, 2007; Cong, Fan, Geerts, Jia, & Ma, 2007; Ebaid et al., 2013; Lopatenko & Bravo, 2007; Yakout, Elmagarmid, Neville, Ouz- zani, & Ilyas, 2011）。这些系统所采用的技术差异很大，从全自动的基于启发式的方法（Bohannon et al., 2005; Cong et al., 2007; Lopatenko & Bravo, 2007）、与外部来源和存储库进行比对的方法（Arasu, R´e, & Suciu, 2009; Beskales, Soli- man, Ilyas, & Ben-David, 2009; Han, Jiang, & Li, 2010），以及规则驱动技术（Bohannon et al.,

2007; Bravo, Fan, & Ma, 2007），到人在环机制（Galhardas, Lopes, & Santos, 2011; Yakout et al., 2011）。随着数据清洗算法和技术日益多样，已有若干可扩展的通用框架被提出，用于整合这些算法，例如（Dallachiesa et al., 2013; Ebaid et al., 2013; Geerts, Mecca, Papotti, & Santoro, 2013）。所有这些系统的共同点是，它们都以离线方式运行，并与查询处理完全隔离。由于所有现代应用中的数据都在快速演变和增长，修复工作因此成为一项永无止境且耗时的任务。

因此，当数据中包含低质量记录时，对其进行查询、分析和决策是不可避免的。然而，在查询处理过程中，上述技术均无法支持评估结果的质量，也无法实现质量感知处理。更糟糕的是，如果这些技术识别出潜在的错误记录，并将其标记为待验证或待修复——而这可能需要很长时间才能完成，那么就没有任何机制能够将这些观察结果整合到查询处理中。

2.2 质量评估技术

另一方面，针对查询时间的质量评估却鲜有人关注。相关研究已涉及挖掘操作（Su、Li 和 Peng，2008）、传感器数据（Klein 和 Lehner，2009；Pon 和 Cardenas，2005）以及关系数据库（Ballou、Chengalur-Smith 和 Wang，2006；Motro 和 Rakov，1996）等领域。这些技术的核心基于对底层数据的统计假设，例如定义完整性、健全性和错误概率等统计度量。随后，各种技术分别研究其特定领域中的操作，以及这些操作如何影响统计度量。

这些系统的一个主要局限是，在许多应用程序中，所假定的统计信息可能并不可用。例如，与 QTrail-DB 最相关的工作（Ballou 等，2006；Motro 和 Rakov，1996）假定数据库中每一列的错误概率都是预先已知的，但许多应用程序并非如此。即使具备这些知识，它也是针对整列的粗粒度知识，而不是与特定元组相关联；例如，列 X 中 1 % 的错误率意味着，在 X 的每 100 个值中，预计会有 1 个错误。因此，估计的输出质量同样是粗粒度的，无法与特定元组关联。此外，这些系统假定采用<style id='5'>单一分数质量模型，没有考虑到数据记录具有长期存在的特性，并且其质量会随时间而变化。

QTrail-DB 与上述两类有着根本区别：(1) 它提出了一种基于质量轨迹、而非单一分数质量模型的更丰富的质量模型；(2) 它不对数据的特征作任何假设，例如估计错误率，而是随着数据库随时间演变，逐步创建并维护质量轨迹；(3) 它的质量模型完全集成在查询处理引擎中。

QTrail-DB 与数据清理和修复技术相辅相成，两者结合可提供更全面的解决方案，分别实现在线的质量感知查询处理和离线修复过程。

2.3 不确定数据库与概率数据库

另一个重要的研究领域聚焦于不确定数据库与概率数据库（Galindo、Urrutia 和 Piattini，2006；Singh 等，2008；Widom，2005a）。在这些系统中，给定的数据值（或整个元组）可能是不确定的，因此会用一组可能的值、某个给定范围上的概率分布函数，或实际存在的概率来表示。在不确定数据库中，查询引擎经过扩展，可以处理这些不确定的值和元组，并确保语义正确（称为 “可能世界”）。尽管从某种意义上说，不确定性与数据质量有关，但这些系统与 QTrail-DB 有着根本区别，因为“质量”的概念并不属于这些系统。因此，不确定数据库和概率数据库既无法建模或追踪数据质量，也无法实现 QTrail-DB 系统所提出的高级质量驱动型查询处理。

2.4 数据沿袭与数据溯源

数据溯源与数据质量直接相关，因为元组的质量显然取决于其数据溯源。已有多个系统研究了数据溯源信息的推导与传播，例如（Buneman、Chapman 和 Cheney，2006；Buneman、Khanna 和 Tan，2001）；甚至还有一些系统（如 Trio（Widom，2005b））将不确定性与数据溯源结合起来。

QTrail-DB 利用了数据溯源文献中经过深入研究的理论基础和传播语义。然而，从捕获原始溯源信息到将这些信息转

化为基于质量的知识和分析，仍存在很大差距。例如，

1. 溯源系统可以报告每个输出元组 r 的根源——其中可能包括其他数百个元组的标识符——但它们无法告知科学家结果是否具有较高质量、是否值得用于进一步分析。目前，获取这些知识需要付出巨大且不切实际的人工努力，即使这确实可行。

2. 溯源系统不提供可量化指标供查询进行交互，即溯源信息通常是难以应用条件或转换的不透明对象。而且

3. 溯源系统无法捕获数据元组的历史（或演变），因此，即使在t_1 与t_2 之间发生的某些变化改变了记录的质量而未改变其内容，例如添加了整理信息，从而提升了记录的价值，在Q 时刻t_1 和t_2 执行同一查询，也会返回相同的溯源信息。

QTrail-DB 解决了这些问题，填补了这一空白。

3 QTRAIL-DB数据与质量模型

QTrail-DB采用扩展的数据模型，其中每个数据元组除携带数据值外，还携带“质量轨迹”，用于编码该元组不断演变的质量。更正式地说，对于具有n个数据属性的关系R，每个数据元组$r \in R$ 的模式为：$r = \langle v_1, v2,..., v_n, Q_r \rangle$，其中$v_1, v2,..., v_n$是$r$的数据值，而$Q_r$ 是r的质量轨迹。Q_r 是形式为$Q_r = \langle q_1, q2,..., q_z \rangle$的向量，其中每个点$q_i$ 都是一个质量转换，定义如下。

定义 3.1（质量转换）。质量转换表示元组质量的变化，由一个四元向量 ⟨分数、时间戳、触发事件、统计信息⟩组成，其中，**“分数”**是介于 *1*（最低质量）和 *MaxQuality*（最高质量）之间的质量分数，**“时间戳”**是分数开始适用的时间，**“触发事件”** 是描述触发此次质量转换的事件的文本字段，而**“统计信息”** 字段包含将在查询处理中维护和更新的各种统计信息。只有**“分数”**和**“时间戳”** 是必填字段，而**“触发事件”**和**“统计信息”**是可选字段。

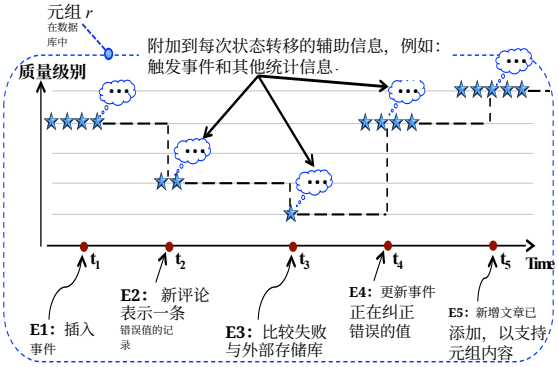


图2：与图1中操作对应的r的质量轨迹示例。

由于r的质量会随时间变化，Q_r的向量长度也会随着时间推移，通过添加新的状态转移而不断增加（参见图2）。质量轨迹正式定义如下。

定义3.2（质量轨迹）。给定元组 $r \in R$ 的质量轨迹记为Q_r，表示为质量转换向量。Q_r 中的转移按时间顺序排列，即$Q_r[i]$.时间戳 $< Q_r[i+1]$.时间戳 $\forall i$。此外，质量转换具有逐步变化的模式，即在时间段 $[Q_r[i]$.时间戳，$Q_r[i+1]$.时间戳）内，$Q_r[i]$是有效的转换。

参见图1中的数据元组r，其对应的质量轨迹如图2所示。随着图1中突出显示的每个操作，r 的质量轨迹将从左侧（插入时间）变化（演化）到右侧（当前时间）。质量轨迹中的每个点都是一次质量转换。例如，在时间t_4,，轨迹中会添加一个新的质量转换，其内容为：⟨4,t_4,“更新错误值” {...}⟩。该转换将一直有效（作为最近的一次转换），直到时间t_5，届时会添加新的转换。统计 字段及其用法将在第4节中进一步详细讨论。

4 查询结果的质量传播与评估

本节介绍 QTrail-DB 的扩展查询处理引擎，用于在查询计划中传播质量轨迹。为了以透明且流水线化的方式实现此类推导与传播，我们提出了一种新的 SQL 代数，称为 *“QTrail*代数”，其中标准查询操作符经过扩展，可无缝处理与每个元组关联的质量轨迹。在 QTrail-DB 中，每个

操作符都将消费并生成符合第 3 节所述数据模型的元组。在本节中，我们假设质量轨迹已经创建并得到维护（第 6 节的重点），因此现在将重点放在查询时的传播上。

幸运的是，在数据溯源文献中，不同运算符下元组数据血缘的传播语义是一个经过充分研究的问题。具体而言，我们采用与 Trio 系统相同的语义（Widom，2005b）。因此，在每次代数变换之后，我们都可以跟踪对特定输出元组有贡献的输入元组，而无需重新发明轮子。然而，尚未解决的挑战是如何将这些知识转换为关于质量轨迹的推导？例如，假设两个元组r_1 和r_2 将连接起来，生成一个输出元组r_o；根据现有文献，已经确定r_1 和r_2 共同构成r_o的数据血缘。剩下的问题是，给定r_1和r_2, 的质量轨迹，如何推导出r_o的质量轨迹？下文将研究如何根据每条输出记录的贡献输入记录推导其质量轨迹的语义。

4.1 选择算子 ($\sigma_p(R)$)

该算子对关系 R 应用基于数据的选择谓词p ，并报告符合条件的元$_1$,组。谓词 p 仅引用元组内的数据值$v_1, v2, \ldots, v_n$。由于符合条件的元组内容不会改变，因此对选择算子的扩展很直接，输出质量轨迹也保持不变。代数表达式为： $\sigma_p(R) = \{r = \langle v_1, v2, \ldots, v_n, Q_r \rangle \in R \mid p(r) = True\}$

4.2 投影算子$\pi_{a_1,a2,\ldots,an}(R)$

在 QTrail-DB 中，质量轨迹处于元组级别，并不与元组中的特定属性绑定。因此，投影算子不会改变其输入元组的质量。也就是说：$\pi_{a_1, a2,\ldots,an}(R) = \{r' = \langle a_1, a2, \ldots, an, Q_r \rangle\} \; \forall r \in R.$

4.3 合并算子（Ω（Q_1, Q_2，...））

若干关系算子（例如连接、分组、聚合等）需要将多个输入元组合并（或组合）成一个输出元组。因此，相应的输入质量轨迹也可能需要合并和组合。为对质量轨迹执行此合并操作，我们引入新的合并算子$\Omega(Q_1, Q2, \ldots)$。该算子不是物理算子——它是一种在其他物理算子（例如连接、分组和重复项消除）内部执行的逻辑算子。

合并算子 Ω（Q_1,Q_2,...，Q_n）

输出：

- 质量轨迹 Q_o ！ 初始时没有转移

1. 将扫描线 L 放置在 Q_1,Q_2,...，Q_n 中最左侧的转移处
2. **当**（还有更多转移可用时）**执行**
3. 将 L 向右移动到时间 t 处的下一个转移
4. S={$s_1, s_2, \ldots, s_n$} ！ 在 t 时来自 Q_1,Q_2,...，Q_n 的活动转换集合
5. s_{out} ！ 一个新的输出转换
6. s_{out}.分数 = 最小值(S.s_i.分数), $1 \le i \le n$
7. s_{out}.时间戳 = t
8. s_{out}.统计信息 = StatsCombine(S.s_i.统计信息), $1 \le i \le n$
9. s_{out}.触发事件 = Null
10. Q_o.addTransition(s_{out})
11. **结束循环**
12. **返回** Q_o

图3：合并 Ω 算子的伪代码。

合并算子的逻辑如图3所示，其功能通过图4中的示例进行说明。假设将三个元组r_1, r_2, 和r_3 分别与质量轨迹Q_{r1}、Q_{r2}、和Q_{r3}结合。全部质量轨迹通常从右侧（即查询时间Q_t）对齐，也就是说，每条质量轨迹都必须在时间Q_t具有有效的状态转移。然而，由于数据元组可能在不同时间插入数据库，因此这些轨迹不一定从左侧对齐（见图4）。输出元组Q_o 的质量轨迹通过对输入质量轨迹应用扫描线算法导出：从左向右扫描，并跳过状态转移点，如图4所示（图3中的第1—3行）。该算法的基本思想是：输出元组在任意给定时间t 的质量，应为各贡献元组在时间t的质量中的最低值。这基于以下直觉：低质量输入会产生低质量输出；只有当所有贡献输入元组在时间t都具有高质量时，输出元组在时间t 才应具有高质量。

参见图4中的示例，扫描线从位置1开始，此时只有Q_{r1} 存在，其质量级别为*4*星，该质量级别将被输出。随后，扫描线跳转到位置2，此时Q_{r3} 开始以*3*星质量级别参与，因此会向Q_o添加一个*3*星状态转移。扫描线继续移动到后续位置，并在每个位置计算输入参与者中的最低质量分数，将其作为输出在该位置的质量分数（图3第5—7行）。例如，参见图4(a)中的示例，在时间t_4, , 来自Q_{r1}、Q_{r2}和Q_{r3}的输入质量分别为*2*星、*3*星和*5*星，

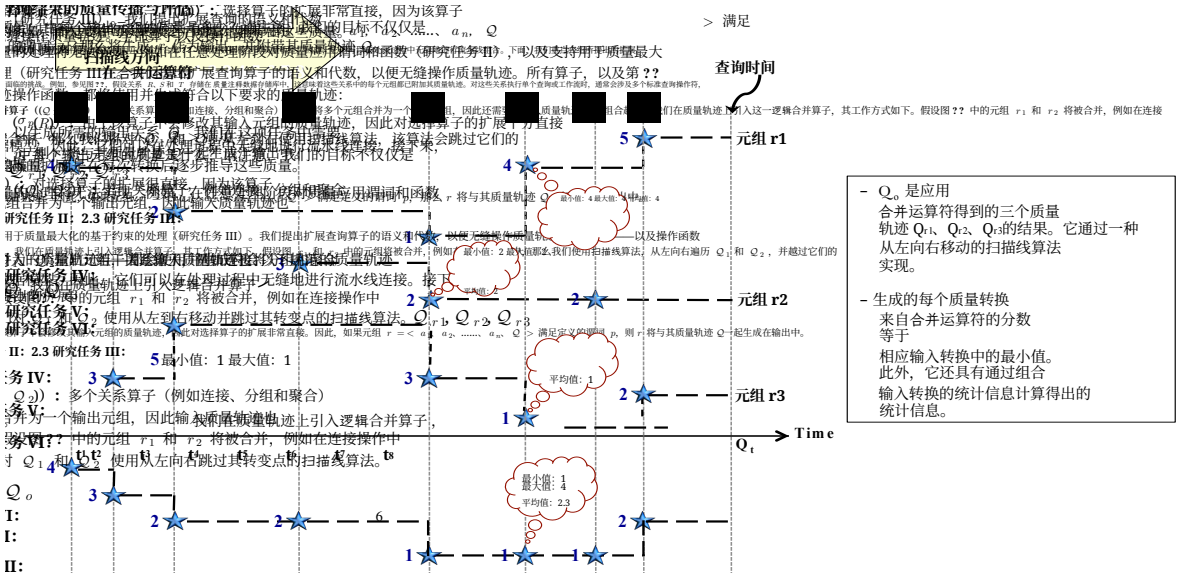


图4：QTrail-D 中合并运算符示例 B.

因此，Q_o 上相应的质量转换将具有*2* 星分数。

尽管Q_o 的质量分数仅反映输入值中的最低分数，但与每个质量转换相关联的统计信息字段旨在深入揭示影响该分数的其他值。最初，与每个质量转换相关联的统计信息（例如 Min、Max、Avg）均设为该转换的分数值，如图3所示。随后，随着各转换被合并，会计算新的统计信息，并将其附加到新的质量转换上。例如，位置 6 处的扫描线会遇到分数为*4* 星、*2* 星和*1* 星的转换，以及它们各自的初始统计信息。请注意，Q_{r2} 在位置 6 处的活动转换仍然是位置 5（时间t_5）处发生的*2* 星转换。合并运算符将组合这些统计信息，以计算Q_o 新转换的统计信息（图3中的第 8 行）。该示例展示了如何维护 Min、Max 和 Avg 统计信息。不过，一般而言，任何能够随着新值加入而增量计算的聚合都可以得到支持，例如代数聚合（如 Avg() 和 Stddev()），或可在插入时进行分配计算的聚合（如 Sum()、计数、Min() 和 Max()）（Palpanas、Sidle、Cochrane 和 Pirahesh，2002 年）[1]。

最后，新创建的状态转移将被添加到输出质量轨迹中（图3中的第10行）。

[1]代数聚合是指在增量计算时只需要少量或恒定额外存储空间的聚合。插入可分配聚合是指添加新数据时无需额外存储空间即可增量计算的聚合。

4.4 θ连接算子 ($R \bowtie_p S$)

给定两个元组 r ∈ R 和 s ∈S，它们分别具有质量轨迹 Q_r 和 Q_s。如果 r 和 s 满足基于数据的连接谓词 p，并生成连接元组 z，那么 z 将继承其两个组成元组 r 和 s 的合并质量。合并算子的相同直觉也适用于此：在任意给定时间点 t，z 的质量应当是各贡献元组中的最低值。代数表达式为：$R \bowtie_p S = \{\langle r.v1,..., r.v_n, s.v',..., s.v'_m, \Omega(Q_r, Q_s)\rangle \mid r \in R, s \in S, and\ p(r,s) = True\}$

笛卡尔积 ($R \times S$) 和自然连接 ($R \bowtie S$) 算子遵循与 θ连接算子相同的代数表达式。唯一的区别在于，对于笛卡尔积算子，p 为 *True* ；而对于自然连接算子，p 由公共属性上的等值谓词组成。

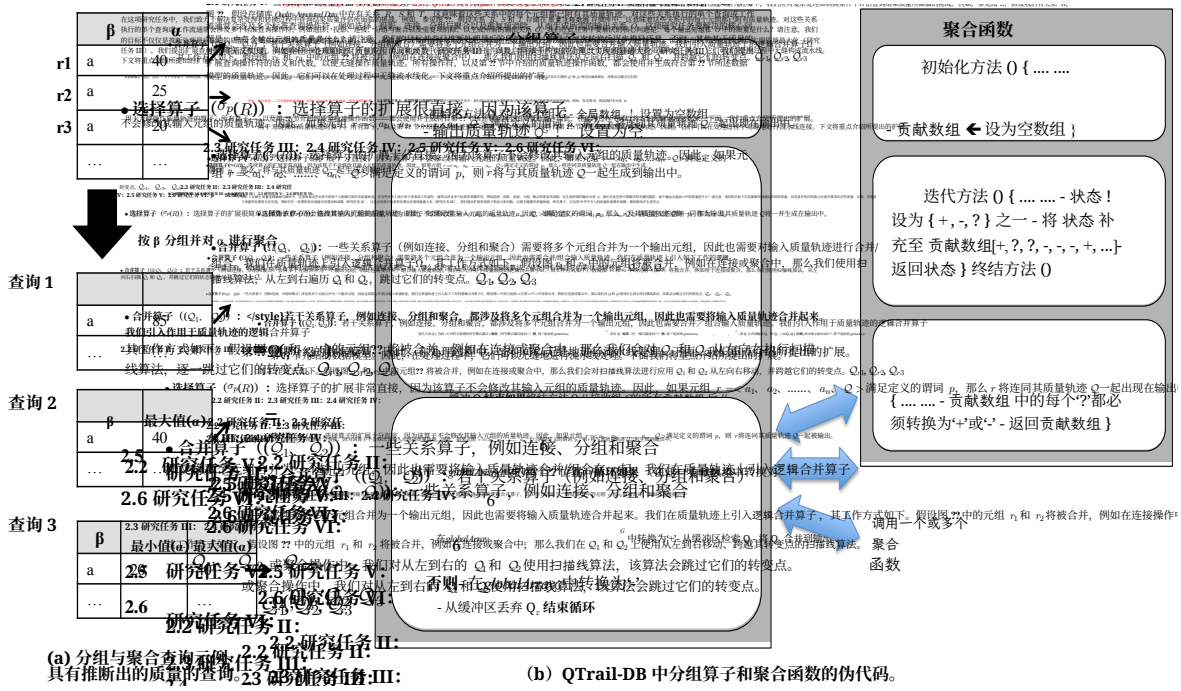
外连接算子（左外连接、右外连接和全外连接）是对上述内连接算子的直接扩展。不失一般性，假设元组 $r \in R$ 会出现在输出中，且没有来自 S 的匹配连接元组，即 S 的值将为 null，那么 r 将继承其自身的质量轨迹。也就是说，r 对应的输出元组为：

$\{\langle r.v_1,...,r.v_n, null ...,null, Q_r\rangle\}, \quad \forall\ r \in R \quad \nexists\ s \in S \mid p(r,s) = 真$

4.5 分组与聚合

$$(\gamma_{a_1,a_2,...,a_m,F_1(\beta_1),...F_z(\beta_z)}(R))$$

分组和聚合是 QTrail-DB 中最有趣且最具挑战性的运算符之一。一个关键



(a) 分组与聚合查询示例，具有推断出的质量的查询。

(b) QTrail-DB 中分组算子和聚合函数的伪代码。

图 5：QTrail-DB 中分组算子与聚合函数之间的扩展反馈机制。

挑战在于，为了准确推断输出元组的质量，我们需要跟踪实际参与聚合的元组。例如，对于某些聚合函数，每个组中只有少数元组会影响输出值。参考图 5，假设我们根据列 β 对元组r_1, r_2, 和 r_3 进行分组，并对列 α进行聚合。图 5(a) 中展示的三个查询分别计算不同的聚合函数，即 Sum()、Max() 以及 { Min() & Max()}。显然，对于 Sum() 函数，所有元组r_1, r_2, 和 r_3都对输出元组作出了贡献。相比之下，对于 Max() 函数，只有r_1 对输出作出了贡献；而在同时计算 Min() 和 Max() 的*Query 3* 中，r_1, 和 r_3 都作出了贡献。

在 QTrail-DB 中，我们支持两种语义：基于粗粒度溯源的语义，以及一种新的细粒度语义。在粗粒度语义中，聚合函数被视为黑盒，因此假定每个组中的全部元组都参与生成输出（不了解黑盒聚合的方法）。在这种情况下，每个组中所有相应的质量轨迹都会被合并，以推导输出的质量。这是溯源跟踪 Trio 系统所采用的方法（Widom，2005b）。尽管这种语义简单直接，但它属于粗粒度且较为悲观的语义，可能会导致不必要地低估输出的质量。回想一下根据*Merge* 运算符，合并的质量轨迹越多，推导出的质量只会降低而不会提高。当这些悲观的推导在查询计划中不断累积时，最终结果可能会变得毫无用处，也无法被信任，因为其质量较低且推导不够精确。

QTrail-DB 还支持另一种开放式聚合感知语义，该语义能够识别影响所生成聚合的最小输入元组集，并且仅使用这些元组推导输出的质量。例如，如图 5(a) 所示，不同查询将根据所涉及的聚合函数生成不同的关联质量轨迹；例如，在 Query 2 中，由于只有元组$r1$参与其中，输出元组被分配了Q_{r1} 质量轨迹。这种语义的优点在于避免不必要地降低输出的质量，从而提高所生成结果的价值和使用率。然而，实现开放式语义需要扩展聚合函数与分组算子之间的通信机制，具体如下所述。

一般而言，QTrail-DB 中的任何聚合函数——QTrail-DB 依赖于 PostgreSQL 数据库管理系统（Stonebraker、Rowe 和 Hirohama，1990）——都包含三种基本方法，即 初始化()、迭代() 和 *Finalize()*（见图 5(b)）。初始化() 函数在输入记录开始流式传输之前执行一次；对于每条输入记录，都会调用 迭代() 以更新聚合器的内部

状态；在处理完所有元组后，*Finalize()* 执行一次，以最终确定聚合器的输出值。所提出的扩展要求每个聚合函数维护一个字符数组，称为 贡献数组。随后，对于每个输入元组，该函数根据聚合器的语义判断该元组是否：(1) 确定对输出有贡献，即向数组中添加“+”；(2) 确定对输出没有贡献，即向数组中添加“-”；或 (3) 该元组处于不确定状态，目前尚不明确，即向数组中添加“?”（见图 5(b) 右侧的 迭代*()* 方法）。除了在 贡献数组中累积每个元组的状态外，该状态还会返回给分组算子，以便进行增量更新，详情将在下文讨论。最后，如图所示，*Finalize()* 方法会将 贡献数组 返回给分组算子。

在聚合函数操作贡献数组时，QTrail-DB 强制执行分组算子赖以确保正确性的两个不变量：(1)“+”和值“-”是永久性的，不得更改；(2) 从聚合器的*Finalize()* 函数返回贡献数组 时，数组中不得包含任何不确定元组，即任何“?”条目都必须转换为“+”或“-”。这是合乎逻辑的，因为 *Finalize()* 方法的执行意味着聚合器已经看到了其所有输入元组，并确切知道哪些元组实际对输出做出了贡献。

另一方面，分组算子可能会在单个查询中调用多个聚合函数，因此会在不同聚合函数之间维护全局状态（见图 5(b) 左侧）。更具体地说，对于每个构建的组G，该算子都会维护一个名为 $globalArray_G$ 的字符数组。随后，对于属于G的给定输入元组r ，如果所有聚合函数都返回“-”状态，则r的质量轨迹Q_r 会立即被清除，并将“-”追加到$globalArray_G$中。这是因为r 肯定不会对G的输出产生贡献。相反，如果任一聚合函数返回“+”状态，则Q_r 会立即合并到G的输出质量轨迹Q_G 中。否则，r的状态尚未确定，因此会将“？”追加到$globalArray_G$ 中，并由分组算子缓冲Q_r （见图 5(b) 中分组算子的迭代方法）。最后，在该算子的*Fi-nalize()* 方法中，所有聚合函数完成工作后，分组算子会检查不确定元组（例如r）的最终状态。如果任一聚合函数将其“？”状态更改为“+”，则会从缓冲区中取回r的质量轨迹，并将其合并到Q_G中；否则，r的质量轨迹将被丢弃。

所提出的分组算子扩展属于数据库系统内部实现，对外部世界完全透明。对于聚合函数，我们扩展了内置函数（例如 Min()、Max() 和 Avg()），以实现新的语义。对于新的用户定义聚合函数，数据库开发人员需要根据该函数的语义实现对贡献数组 的操作。在我们扩展的大多数聚合函数中，这种操作只需极少的工作，例如增加不到 10 行代码。作为回报，这对推断出的质量的准确性所产生的影响可能非常显著 [2]。

4.6 内存约束下的分组

如图 5(b) 所示，如果Q_r 的状态尚未确定，分组算子可能会将质量轨迹缓冲到内存中r（*Itera-tive()* 方法的最后两行）。然而，为了保证可扩展性，分组算子必须能够在有限内存下运行，例如在最大大小为M的内存缓冲区中运行，而不会崩溃。为实现这一点，并避免频繁写入磁盘（这会降低处理速度），我们部署了一种名为“*BufferClean()*”的算法。当分配的内存缓冲区已满时，分组算子会执行该算法。BufferClean() 的功能很简单：从聚合函数中获取最新的贡献数组，希望某些已缓冲元组的状态已变为“+”或“-”。在这种情况下，可以将它们的质量轨迹从缓冲区中移除。在最坏情况下，如果*BufferClean()* 未能释放空间，则会将内存缓冲区写入磁盘，并在调用 *Final-ize()* 方法时将其重新取回。有趣的是，*BufferClean()*算法在许多情况下都非常有效，可以完全避免写入磁盘的需求，因为一些可能需要缓冲的常见聚合函数（例如 Min() 和 Max()）在看到更多元组后，可能会开始丢弃之前的候选项（参见第 7 节中的实验）。

4.7 集合运算符

集合运算符 $\cap$ $\cup$和 $-$的语义以与其他运算符相同的方式扩展，即如果一个输出元组同时来自两个关系，那么它将

[2]在最坏的情况下，聚合函数可能会对每个输入都盲目地返回“+”，从而回退到黑盒语义。

继承贡献输入元组合并后的质量轨迹。否则，输出元组仅来自一个关系，因此将携带其自身的质量轨迹。集合运算符扩展后的代数表达式定义如下：

$R \cap S = \{\langle z.v_1, \ldots, z.v_n, \Omega(Q_{R.z}, Q_{S.z})\rangle \mid z \in R \,\&\, z \in S\}$

$R - S = \{\langle r.v_1, \ldots, r.v_n, Q_r\rangle \mid r \in R \,\&\, r \notin S\}$

$$R \cup S = \{z \mid z = \left\{ \begin{array}{ll} \langle z.v_1, \ldots, z.v_n, \Omega(Q_{R.z}, Q_{S.z})\rangle & \text{if } C_1 \\ \langle z.v_1, \ldots, z.v_n, Q_z\rangle & \text{if } C_2 \end{array} \right\}$$

其中

$C_1 = (z \in R \,\&\, z \in S)$，以及$C_2 = (z \in R \text{ xor } z \in S)$

4.8 重复项消除 $\delta(R)$

该运算涉及消除重复元组，仅保留一个副本。在 QTrail-DB 中，当两个元组 $r = \langle v_1, v2, \ldots, v_n, Q_r\rangle$ 和 $s = \langle z_1, z2, \ldots, z_n, Q_s\rangle$在数据部分中完全相同时，即 $r.v_i = s.z_i$, $1 \le i \le n$，才认为它们相同。对于每个相同元组的组G，其输出元组的质量轨迹通过合并G中输入元组的质量轨迹得到。

5 QTrail-DB 中的查询优化

基于数据溯源文献中 Trio 系统（Widom, 2005b）等所采用的传播语义构建系统的一项关键优势是：对输出元组o 的质量有贡献的输入元组，即被视为o的数据血缘的输入元组，能够保留关系数据库中的等价规则和优化规则。然而，由于 QTrail-DB 的某些算子会对质量轨迹应用复杂的合并函数，因此不能直接声称这些规则得以保留（参见第4节中的合并 算子 Ω ）。因此，有必要研究该合并 算子的性质，并以此证明其他代数性质。

引理 5.1（合并算子的性质）。 合并算子具有以下两个性质：

交换律：$\Omega(Q_1,Q_2) = \Omega(Q_2,Q_1)$ 结合律：$\Omega(\Omega(Q_1,Q_2),Q_3) = \Omega(Q1,\Omega(Q_2,Q_3)) = \Omega(Q_1,Q_2,Q_3)$ □

证明概要： 交换律的证明很直接，因为每个新状态转移的计算（图3中的第5—9行）都不受输入质量轨迹顺序的影响，例如Q_1 和Q_2。第6行和第8行中的 *Min()* 与 *StatsCombine()* 函数分别将其输入值视为一个集合，因此顺序并不重要。

证明结合律依赖于这样一个事实：从输入状态转移生成每个新状态转移的基本操作本身具有结合律。也就是说，第6行中的 *Min()* 函数具有结合律，而用于组合底层统计信息的 *StatsCombine()* 函数也涉及具有结合律的函数。请注意，统计信息必须是代数的（例如 *AVG* 和 *STDEV*），或在插入操作下具有可分配性（例如 *SUM*、*COUNT*、*MIN*、*MAX*）；所有这些函数都具有结合律。

根据引理 5.1，我们在下述定理中证明，QTrail-DB 的算子与标准关系算子一样，对输入顺序不敏感。

定理 5.2（集合语义）。*QTrail-DB* 中的每个算子都能保证生成相同的质量标注的输出元组，不受输入顺序影响。□

证明概要：标准关系算子在集合语义或包语义下执行，因此都对顺序不敏感，也就能在输入顺序不同时生成相同的输出。*QTrail-DB* 中扩展后的对应算子可分为三类：（*1*）不对质量轨迹进行任何操作的算子，例如 σ、π 和 −；（*2*）通过合并算子合并质量轨迹的算子，例如 ⋈、γ、∩ 和 δ；（*3*）同时符合上述两种情况的算子，例如外连接和 ∪。1st 类算子默认保留顺序不敏感这一性质。根据引理 *5.1*，2^{nd}类和 3^{rd}类算子也能保证，无论输入顺序如何，都能为每个输出元组推断出相同的质量。因此，定理 *5.2* 得以成立。

定理 5.2 十分重要，因为它保证了每个算子的输出一致性，而不受输入顺序的影响。定理 5.2 的一个直接应用是证明：尽管某些质量轨迹可能会被缓冲，直到算子执行结束时才处理，分组算子仍将生成具有相同推断出的质量的输出元组（参见图 5 中分组算子内部的迭代*()*函数）。

6 质量轨迹的创建与维护

本节介绍质量轨迹的创建和维护机制。由于

函数名称	描述
质量转换[] getQualityTrail()	以质量转换数组的形式返回 r 的质量轨迹。
整数 getSize()	返回 r 的质量轨迹中的转换数量
QTransition[] addTransition(QTransition *q*)	将 *q* 添加到 r 的质量轨迹最右侧。该函数返回扩展后的新质量轨迹。
质量转换[] replaceTransition(整数 位置, 质量转换 *q*)	将位置 *pos* 处的质量转换替换为新的状态转移 *q*。该函数返回新的修改后的轨迹。
QTransition[] trim(Char 方向, Int 数量)	修剪 r 的质量轨迹，仅保留前 数量 个从左侧或右侧开始的转移（取决于 方向）。该函数返回修剪后的新轨迹。
…	…

图6：*r.QTrail* 属性上的操作函数。

QTrail-DB 是一个通用引擎，因此目标是设计一组应用程序接口，作为 QTrail-DB 与外部世界之间的接口，例如能够评估质量分数并将其传递给 QTrail-DB 的工具及应用程序逻辑。更具体地说，QTrail-DB 允许数据库开发人员像操作数据库中的其他属性一样操作质量轨迹，例如，在标准数据库触发器和用户定义函数中，根据数据库的特定变化修改和扩展质量轨迹。质量轨迹被设计为添加到数据库关系中的特殊属性，即每个关系*R*都会自动添加一个名为*"QTrail"*的特殊属性。*QTrail* 属性属于一种新添加的用户定义类型，表示质量转换数组（参见定义3.1和3.2）。在此新类型之上，开发了一组操作函数，如图6所示。这些内置函数绝非面面俱到，但它们是基础函数，数据库开发人员可以在此基础上创建语义更丰富的函数。

图6展示了我们开发的部分函数。例如，假设给定一条质量轨迹 *r.QTrail*，getQualityTrail() 函数会将该质量轨迹作为由质量转移组成的数组返回，而 getSize() 函数会返回质量轨迹中的转移数量。addTransition() 和 replaceTransition() 函数可用于更改现有质量轨迹，分别通过添加新的状态转移，或修改并替换现有状态转移来实现。这些函数会在内部再次检查状态转移的时间戳是否始终单调递增；否则将拒绝修改。trim() 函数支持根据某些条件裁剪给定的质量轨迹，例如指定方向（左侧或右侧）以及要保留的转移数量。QTrail-DB 为 trim() 函数提供了多种不同的签名。如果应用程序不希望保留质量轨迹的完整历史记录（其规模可能很大），而是希望将范围限制为特定数量的转移，或截至过去某个特定时间，那么此函数就很有用。除了这些函数，我们还开发了一组用于操作给定质量转移的函数，例如构建质量转移，以及设置或获取转移中的特定字段[3]。

需要特别强调的是，只有出于维护和更新的目的，元组的质量轨迹才会作为普通属性公开。此功能旨在使 QTrail-DB 成为一个适用范围更广、能够服务于多种应用的通用引擎。然而，这与第4节提出的质量传播和评估完全正交。首先，质量传播不需要显式查询质量轨迹，即用户的查询将按照第4节示例1中的标准方式编写，质量轨迹会随查询结果自动传播。其次，传播语义过于复杂，无法交由终端用户处理，也无法编码到每个查询中。因此，QTrail-DB 会自动且透明地管理传播语义。

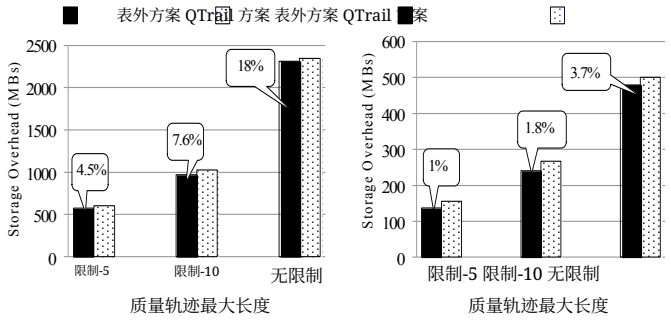
7 实验

设置： QTrail-DB 在 PostgreSQL 数据库管理系统中实现（Stonebraker 等，1990）。质量轨迹被建模为一种新的数据类型，即由质量转换构成的动态数组。默认存储方案称为*"QTrail 方案"*，其中每个用户的关系都会按照第6节所述自动增加一个新的*"QTrail"*列。QTrail-DB 使用一台配备两颗 16 核 AMD CPU、128GB 内存和 2TB SATA 硬盘的 AMD Opteron Quadputer 计算服务器进行实验评估。通过与普通 PostgreSQL 数据库管理系统进行比较，研究新功能所带来的开销（包括时间和存储开销）。文献中的其他工作都没有提供与 QTrail-DB 类似的功能可供比较，尤其是（Ballou 等，2006；Motro 和 Rakov，1996）中的工作尚未进行实验评估。

7.1 应用数据集

我们使用了经过整理的 UniProt 真实世界生物数据库子集（The UniversalProtein Resource

[3]还可以采用其他存储方案，而不影响 QTrail-DB 的核心功能。可能变化的只有应用程序接口的实现。



（a）完整转换下的存储开销（b）最小转换下的存储开销

图7：质量轨迹存储开销。

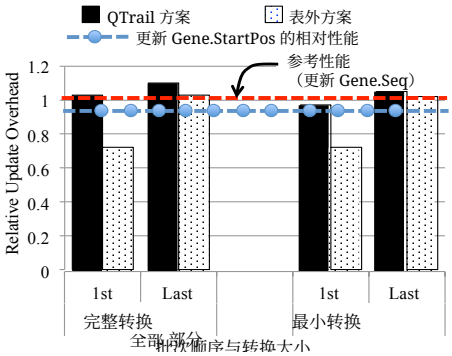


图 8：质量轨迹更新性能。

Databases，2023年）。UniProt 为各种物种提供了涵盖蛋白质及其功能信息的综合资源库。我们提取了四个主要表，包括 Protein 表、Gene 表、Publication 表和 Comment 表。这些表通过以下关系相互连接：Protein 表与 Gene 表具有多对一关系，与 Publication 表具有多对多关系，与 Comment 表具有一对多关系。Gene 表同样与 *Publication*具有多对多关系，与 Comment 表具有一对多关系。Comment 表包含与基因和蛋白质相关的自由文本评论和注释。这些评论提取自 UniProt 条目中标记为“CC”的字段，该标记表示自由文本评论字段。我们的数据集包括约 750,000 条蛋白质记录（≈4.7GBs）、1.3 x 10^6 条基因记录（≈ 8GBs）、12 x 10^6条出版物记录（≈ 4.5GBs）和 8 x 10^6 条评论记录（≈ 6.5GBs）。因此，该数据集的总大小约为 24GBs。

7.2 工作负载

我们重点跟踪在新增出版物和评论的情况下，Gene 表和 Protein 表中元组的质量。质量分数介于 1（最低质量）和 10（最高质量）之间。为了构建质量轨迹，我们在 Publication 表和 Comment 表中的每个表上实现了一个“插入后”数据库触发器，使得插入新的出版物或评论时，相应基因或蛋白质的质量将得到更新。我们假设插入新的出版物会提高质量（除非质量分数已经达到最大值；在这种情况下，新的质量转换将与前一个转换具有相同的分数）。对于评论值，UniProt 中的每条评论都有一个代码，用于表示该评论的类型。其中一种类型是“*CAUTION*”，表示数据中可能存在错误或混淆。全部具有“*CAUTION*”类型的评论都被假设会降低质量（除非质量分数已经达到最小值；在这种情况下，新的质量转换将与前一个转换具有相同的分数）。对于其他评论值，我们将每条评论随机标记为“+”、“-”或“~”，分别表示提高、降低或保持之前的质量分数 [4]。

除非另有说明，我们假设如下：（1）质量分数每次增加或减少一级，即 ±1；（2）如果质量转换存储统计信息——称为“完整转换”——则会维护三种统计信息，分别是 {最小值、最大值、平均值 = （总和、计数）}。由于数据库不会按照全局时间戳对记录创建时间排序，因此出版物记录和评论记录的插入顺序是随机交错的。评估 QTrail-DB 的查询性能时，我们将其与标准查询处理进行比较；后者甚至不会在数据库中存储质量轨迹。最后，我们没有触碰或修改 PostgreSQL 的查询优化器，因此评估中使用的查询均按照标准方式进行优化。

7.3 存储与维护评估

在图7中，我们研究质量轨迹带来的存储开销。为了更全面地比较，我们将“*QTrail 方案*”与另一种方案进行比较：在该方案中，给定表*R*的质量轨迹存储在单独的表*R-QTrail (OID, QTrail)*中，该表与*R*具有一对一关系。此方案称为“表外方案”。我们研究以下两种情况下的存储开销：（1）完整转换，其中每个质量转换都包含

[4]由于开发自由文本语义提取工具（或利用现有工具）并非本文重点，因此我们采用了随机标注。

所有字段的内容，包括必填字段（分数和时间戳）以及可选字段（触发事件和统计信息）（图7(a)）。在这种情况下，触发事件字段是一个长度在50到100字节之间的字符串。（2）最小转换，其中每个质量转换仅包含必填字段的内容（图7(b)）。

在这两幅图中，我们测量了质量轨迹最大长度（横轴）受到不同限制时的存储开销。限制-*5*、限制-*10*和无限制 分别表示最多只保留最后 5 次、10 次或 ∞ 次转移。纵轴显示绝对存储开销，矩形框中的百分比则表示开销——更具体地说，是 QTrail 方案的开销占基因和蛋白质大小总和（≈12.7GB）的百分比。如图 7(a) 和图 7(b) 所示，在所有情况下，两种存储方案之间都没有明显差异。表外 方案的开销略高，这是因为需要存储唯一的元组标识符值（*OID* 列）。总的来说，即使在存储完整质量历史这一最坏情况下，质量轨迹引入的存储开销也不大，例如，完整转换的开销约为 18%，最小转换的开销约为 3.7%。值得指出的是，在无限制 情况下，最长的质量轨迹包含 37 次转移。

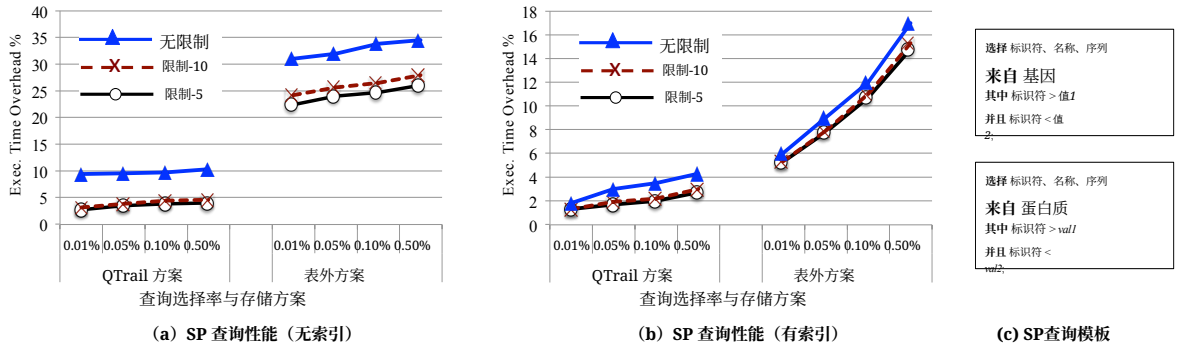
在图 8 中，我们研究了质量轨迹的维护开销。我们考虑质量轨迹的无限制情况。为进行公平比较，我们测量更新质量轨迹（例如添加新的状态转移）所需的时间，并将其与更新其他传统字段（例如更新文本字段或整数型字段）所需的时间进行比较。在所有情况下，更新操作均在 Comment 表上通过实验工作负载中所述的插入后触发器执行。在触发器内部，系统检索相应的基因，以更新其数据或质量轨迹（参见第 6 节中的示例 2）。从数据库中检索数据时，使用 Gene.ID 列上的 B 树索引。每项操作重复执行 20 次，并报告其平均值。如图 8 所示，我们以更新字符串字段的操作作为参考操作；更具体地说，是将 Gene.Seq 字段中的一个片段替换为另一个片段。也就是说，将该操作的执行时间归一化为 1，其他操作则相对于该操作进行测量。图中还展示了更新整数型字段（更具体地说是 Gene.StartPos）的相对性能。平均而言，更新整数型字段的开销约为更新字符串字段的 94%。

为更新质量轨迹，我们同时考虑*QTrail* 方案、表外方案，以及完整转换和最小转换两种情况状态转移。对于每种情况，我们运行一个批次，其中包含 20 个事务，这些事务向 Comment 表中插入记录，从而更新基因的质量轨迹。在横轴上，我们报告两种场景下整个批次的平均性能：(1) 该批次是1^{st}，即所有质量轨迹均为空；(2) 该批次是最后一次，即所有其他记录均已插入，质量轨迹几乎完整。图 8 中的结果表明，对质量轨迹结构进行操作和更新，与更新其他数据库字段的性能非常接近。在*QTrail* 方案下，待查询和更新的表为 Gene 表，相对开销从 0.98%（1st 批次，包含最小转换）到 1.11%（最后一个批次，包含完整转换）不等。表外方案的性能几乎相同；在 1st 批次情况下例外，此时 Gene-QTrail 表为空。

7.4 SSP 查询性能

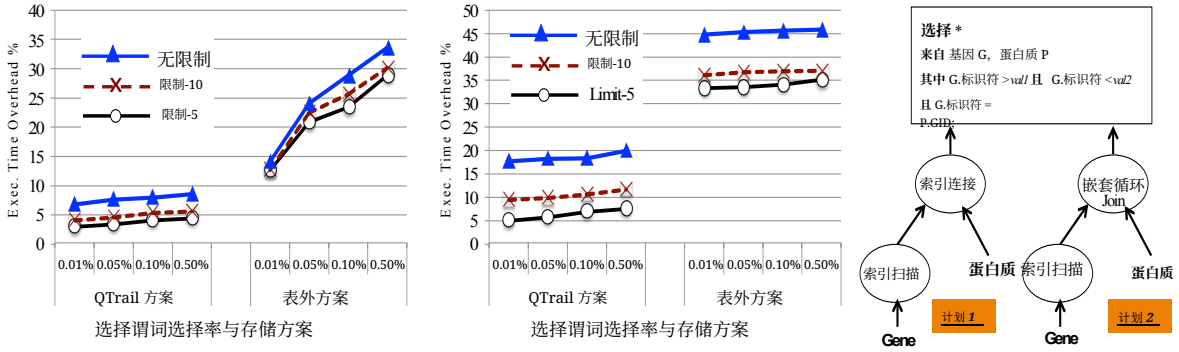
在接下来的实验中，我们评估不同类型查询下质量轨迹在查询时的传播和推导。在图 9 中，我们展示了选择-投影（SP）查询的性能。查询模板如图 9(c) 所示。在图 9(a) 和图 9(b) 的横轴上，我们将查询选择率从 0.01%（对应 75 个蛋白质元组或 130 个基因元组）调整到 0.5%（对应 3,750 个蛋白质元组或 6,500 个基因元组），并考察两种存储方案：*QTrail* 方案 和*Off-Table* 方案。纵轴表示质量轨迹相对于标准查询性能的传播开销，即不存储或传播质量轨迹时的性能。我们考察在限制-*5*、限制-*10*和无限制最大值约束下质量轨迹的传播。在每种配置下，我们分别对 Gene 表和 Protein 表执行 5 个查询，然后报告 10 个查询所得开销的平均值。图 9(a) 和图 9(b) 分别展示了查询在无索引和有索引情况下执行的结果。

结果显示，*QTrail* 方案与表外方案之间存在显著差异。这主要是因为在这两种存储方案下，直接从磁盘读取数据的两个算子，即表扫描和索引扫描算子，具有不同的实现方式。在*QTrail* 方案中，它们经过实现后可以直接从数据元组中读取质量轨迹，而不会产生任何额外开销。相比之下，在表外方案中，它们需要将数据元组与包含质量轨迹的另一张表进行连接。所有其他算子



（a）SP 查询性能（无索引）　（b）SP 查询性能（有索引）　(c) SP查询模板

图 9：选择-投影（SP）查询的性能。



（a）连接查询性能（计划 1）　（b）连接查询性能（计划 2）　(c) 连接查询计划

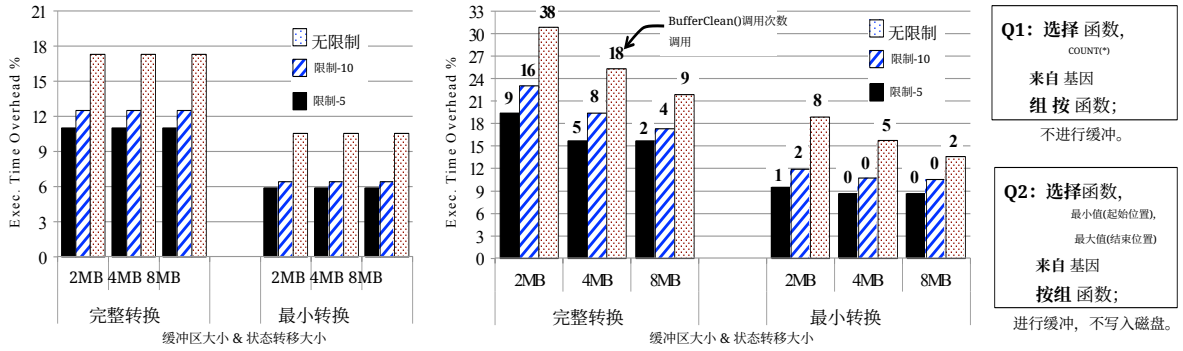
图 10：连接查询的性能。

都不依赖质量轨迹的物理存储方式，因为它们会从查询流水线中的算子缓冲区读取质量轨迹。由于连接是一种开销较高的操作，表外方案会产生更高的开销。在图 9(a) 中，查询选择率并不是重要因素，因为无论选择率如何，都会执行完整表扫描，而完整表扫描占据了大部分成本。相比之下，在图 9(b) 中，系统使用索引来选择满足查询谓词的数据元组，因此性能对查询选择率更加敏感。对于*ff*-表方案，随着选择率提高，开销也会增加，进而导致连接成本上升。如图所示，与 *QTrail* 方案相比，表外方案产生的开销高出 2.5 倍至 6 倍。需要强调的是，选择算子和投影算子不会对质量轨迹进行任何处理，因此产生的开销主要源于质量轨迹引入的额外存储。

7.5 JoinQuery 性能

在图10中，我们测量连接查询的性能。我们考虑使用图-10(c)中突出显示的查询语法连接 Gene 表和 Protein 表。在连接操作之前，会对 Gene 表应用选择谓词，并且如图10(a)和10(b)的横轴所示，该谓词的选择率在0.01%至0.5%之间变化。该查询在两种不同的查询计划下执行，分别代表两种不同的连接类型，即在计划 *1* 中，连接运算符使用 Protein 表中存储的基因 ID（Protein.GID）上的索引；而在计划 *2* 中不存在此类索引，因此连接使用基于块的嵌套循环算法执行。在图10中，我们仅报告完整转换 情况下的结果，因为最小转换 情况下的结果呈现出相似趋势，但开销百分比更低。

在图10(a)中，由于选择和连接操作都在使用索引，质量轨迹的额外存储对执行时间开销的贡献并不大。因此，在质量轨迹的不同最大大小（限制-*5*、限制-*10*和无限制）下，性能没有明显差异。对于*QTrail* 方案，执行开销主要来自连接运算符内对质量轨迹执行的合并操作。而表外方案 则会遇到来自合并操作的相同开销，此外还需要额外的连接来关联数据



（a）聚合查询 Q1 的性能　　（b）聚合查询 Q2 的性能　　(c) 聚合查询

图11：QTrail 方案下聚合查询的性能。

元组与其质量轨迹；该操作隐式地在对两张表执行的索引扫描中完成。连接开销显然占主导地位，高于合并操作的开销。另一方面，图10(b)中的性能结果大多不受查询选择性的影响。这是因为所有情况下都会对 Protein 表执行完整表扫描，而该扫描主导了开销。这也是无限制 情况下的开销明显高于限制-*5* 和限制-*10* 情况的原因；也就是说，I/O 操作数量的差异成为区分这三种情况的因素。

7.6 聚合查询性能

对于分组查询，聚合函数的类型起着重要作用。这是因为某些聚合函数（例如 COUNT、SUM、AVG）不需要缓冲输入质量轨迹，而其他函数（例如 MIN 和 MAX）很可能需要缓冲。图11中，我们研究了这两类聚合器的聚合性能（图11(c)）。查询*Q*1包含一个计数()函数，它总是为每个输入元组返回“+”，因此质量轨迹会被增量合并。相比之下，查询*Q*2包含两个函数 Min() 和 Max()；如果某个元组确定不会影响函数结果，这两个函数都可能返回“-”，如果该元组可能产生影响，则可能返回“？ ”。由于只有看到最后一个元组后才能确定最小值和最大值，因此需要缓冲不确定元组的质量轨迹。在我们实现的 Min()（或 Max()）函数中，如果多个元组具有相同的最小值（或最大值），那么它们都会对输出结果有所贡献。在这组实验中，我们将只关注 *QTrail* 方案，因为它已被证明优于表外方案。

图11(a)中的结果表明，更改 e 最大缓冲区大小（横轴）不会影响性能，因为 COUNT(*) 不需要缓冲。然而，状态转移大小（完整或最小）以及质量轨迹的最大允许大小（限制-*5*、限制-*10*或无限制）都会影响性能。这是因为分组查询需要进行全表扫描，因此随着质量轨迹存储开销的增加，执行时间也会增加。图 11(b) 中的存储开销也会产生相同的影响。不过，*Q*2 会进行缓冲，因此允许的缓冲区大小会影响性能。如第4节所述，当缓冲区已满时，分组运算符将执行*BufferClean()* 函数，试图避免不必要的磁盘写入，并清除那些状态已变为不贡献（“-”）的待定元组（“？ ”）。图11(b)展示了每种情况下 *BufferClean()* 的调用次数。例如，对于缓冲区大小为 8MB 的最小状态转移，无限制情况下仅执行两次*BufferClean()* 调用，而限制-*5*和限制-*10*情况下均不调用。对于本实验而言，这些 *Buffer-Clean()* 调用非常有效，因为随着观察到更多元组，待定元组能够被快速解析，因此即使在最小的 2MB 缓冲区大小下，也不需要进行磁盘写入。因此，根据不同的配置，相对于标准查询的执行开销在 8% 到 31% 之间变化，如图所示。

在某些聚合函数中，执行*BufferClean()* 可能并不总是有效，因此分组算子需要将缓冲区内容写入磁盘。为了研究这种影响，我们在图 12 中重复了图 11(b) 所报告的查询*Q*2 的相同实验，但在本实验中禁用了*Buffer-Clean()*函数。因此，缓冲区已满时，分组算子必须将其写入磁盘。图 12 中的结果证实了与另一项实验相同的趋势，但执行时间更高——

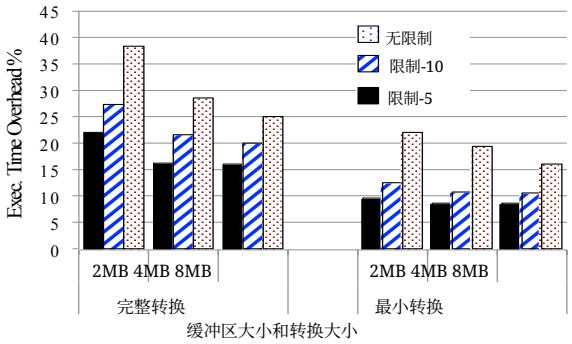


图 12：聚合查询 Q2：禁用的 BufferClean() 和启用的磁盘写入。

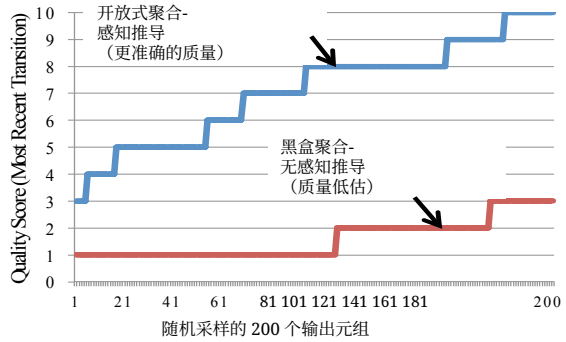


图 13：聚合查询 Q2：粗粒度推导与细粒度推导下的质量分数。

执行开销（最佳情况下介于 0% 到最坏情况下的 6.5% 之间）。值得一提的是，如果某个聚合函数需要为质量轨迹进行大量缓冲，例如 MEDIAN() 函数会缓冲其所有输入质量轨迹，那么该函数很可能也需要对数据值进行大量缓冲。在这种情况下，出于性能考虑，这些函数可以轻松实现粗粒度语义，即为每个输入元组返回“+”，因此无需缓冲质量轨迹。

为了说明为分组和聚合算子实现开放式聚合感知质量推导的优势，我们在黑盒聚合无感知实现下重新执行聚合查询*Q*2；在该实现中，组内所有输入元组都会影响 Min() 和 Max() 函数的输出质量（图 13）。我们从查询结果中随机采样了 200 个元组，并在图 13 中按排序后的顺序报告这些元组最近一次状态转移的质量分数。结果证实，黑盒方法会严重且不必要地低估元组的质量。例如，在开放式方法中，大多数元组的质量分数高于 6；而在黑盒方法中，大多数元组具有最低的质量分数 1。

8 结论

我们提出了 QTrail-DB，这是一种面向质量不断演化的不完美数据库的先进查询处理引擎。在概念层面，QTrail-DB 使高级应用程序能够在数据库系统中建模数据质量，跟踪质量随时间的演化，并基于自动添加质量注释的查询结果做出更明智的决策。在技术层面，QTrail-DB 包含多项创新贡献，包括：(1) 引入一种基于“质量轨迹”这一新概念的质量模型，以区别于常用的“单一分数”质量模型；(2) 扩展关系数据模型以纳入质量轨迹；(3) 提出一种名为“QTrail代数”的新查询代数，它扩展了标准查询算子，并引入新的质量相关算子，用于在查询时传播和推导质量轨迹；(4) 证明 QTrail-DB 的查询优化器无需定制优化器，即可继承并利用标准查询优化器的逻辑。实验评估表明，QTrail-DB 切实可行，其设计选择也具有效率。

参考文献

Arasu, A., R´e, C. 和 Suciu, D.（2009）。使用 Dedupa-log 结合约束进行大规模去重。在*Icde*（第 952–963 页）。Ballou, D. P., Chengalur-Smith, I. N. 和 Wang, R. Y.（2006）。关系数据库环境中查询结果的基于样本的质量估计。*IEEE* 知识与数据工程，*18*（5）。Batini, C. 和 Scannapieco, M.（2006）。数据质量：概念、方法论与技术。*Addison-Wesley*。Beskales, G., Soliman, M. A., Ilyas, I. F. 和 Ben-David, S.（2009）。重复检测中的可能修复建模与查询。《*VLDB*基金会会刊》，*2*（1），598–609。Bohannon, P., Fan, W., Flaster, M. 和 Rastogi, R.（2005）。一种基于成本的模型和用于通过值修改修复约束的有效启发式方法。在*ACM SIGMOD* 国际数据管理会议论文集（第 143–154 页）。

Bohannon, P., Fan, W., Geerts, F., Jia, X., & Ke- mentsietsidis, A. (2007). 用于数据清洗的条件函数依赖。在 *ICDE*（第 746-755 页）。Bravo, L., Fan, W., & Ma, S. (2007). 扩展带条件的依赖关系。在 第*33*届超大型数据库国际会议（第 243–254 页）。Buneman, P., Chap man, A., & Cheney, J. (2006). 整理后的数据库中的溯源管理。在 *SIGMOD*（第 539-550 页）。Buneman, P., Khanna, S., & Tan, W. (2001). 为什么以及何处：数据溯源的特征描述。计算机科学讲义，*1973*，316–333。Cong, G., Fan, W., Geerts, F., Jia, X., & Ma, S. (2007). 提升数据质量：一致性与准确性。在 第*33*届超大型数据库国际会议（*VLDB*）（第 315–326 页）。Dallachiesa, M., Ebaid, A., Eldawy, A., Elmagarmid, A. K., Ilyas, I. F., Ouzzani, M., & Tang, N. (2013). NADEEF：商品化数据清洗系统。在 *SIGMOD* 会议（第 541-552 页）。Ebaid, A., Elmagarmid, A., Ilyas, I. F., Ouzzani, M., Quiane-Ruiz, J.-A., Tang, N., & Yin, S. (2013). NADEEF：通用数据清洗系统。《*VLDB*基金会会刊》，*6*（12），1218–1221。Eckerson, W. (2002). 数据质量与利润底线：通过致力于高质量数据实现商业成功。数据仓库研究院。Galhardas, H., Lopes, A., & Santos, E. (2011). 支持用户参与数据清洗。在第*13*届数据仓储与知识发现国际会议（第 136–151 页）。Galindo, J., Urrutia, A., & Piattini, M. (2006). 模糊数据库：建模、设计与实现。*Idea GroupPublishing*。Geerts, F., Mecca, G., Papotti, P., & Santoro, D. (2013). LLUNATIC 数据清洗框架。《*VLDB*基金会会刊》，*6*（9），625–636。Han, J., Jiang, D., & Li, L. (2010). 多源环境中通过哈希实现自动准确性评估。《专家系统与应用》期刊，*37*（3），2609–2620。Haug, A., Zachariassen, F., & van Liempd, D. (2011). 低质量数据的成本。《工业工程与管理期刊》，*4*（2），168–193。Klein, A., & Lehner, W. (2009). 在传感器数据流环境中表示数据质量。环境。《数据与信息质量期刊》，*1*(2)，10:1–10:28。Lopatenko, A.，& Bravo, L.（2007）。修复不一致数据库的高效近似算法。载于*Icde*（第216–225页）。Motro, A.，& Rakov, I.（1996）。估计关系数据库中的数据质量。载于*1996*年信息质量会议论文集（第94–106页）。麻省理工学院。Palpanas, T.、Sidle, R.、Cochrane, R.，& Pirahesh, H.（2002）。非分配性聚合函数的增量维护。载于*Vldb*（第802–813页）。Pon, R. K.，& Cardenas, A. F.（2005）。数据质量推断。载于信息系统中的信息质量国际研讨会（*iqis*）（第105–111页）。Radziwill, M. N.（2006）。科学数据产品质量管理的基础。质量管理期刊，*13*(2)，7–21。Redman, T. C.（1998）。低数据质量对典型企业的影响。*ACM*通讯，*41*(2)，79–82。Singh, S.、Mayfield, C.、Shah, R.、Prabhakar, S.、Hambrusch, S.、Neville, J.，& Cheng, R.（2008）。数据库对概率属性和元组的支持。载于*2008*年*IEEE*第*24*届数据工程国际会议（第1053–1061页）。Stonebraker, M.、Rowe, L. A.，& Hirohama, M.（1990）。POSTGRES的实现。*TKDE*，*2*(1)，125–142。Su, Y.、Li, D.，& Peng, J.（2008）。数据挖掘中的信息质量风险建模。载于无线通信、网络与移动计算（*wicom*）（第1–4页）。通用蛋白质资源数据库。（2023年）。http://www.ebi.ac.uk/uniprot/。Twombly, M.（2011）。《科学》在线调查：对数据管理的支持。《科学》杂志，*331*。Widom, J.（2005a）。Trio：数据、准确性和数据血缘的综合管理系统。*CIDR*，262–276。Widom, J.（2005b）。Trio：数据、准确性和数据血缘的综合管理系统。*CIDR*，262–276。Yakout, M.、Elmagarmid, A. K.、Neville, J.、Ouzzani, M.，& Ilyas, I. F.（2011）。引导式数据修复。《*VLDB*基金会会刊》，*4*(5)，279–289。Zimnyi, E.，& Pirotte, A.（1997年）。关系数据库中的不完善信息。载于《信息系统中的不确定性管理》（第35–87页）。施普林格美国公司。