RISE：基于查询约简的规则驱动 SQL 方言转换

Xudong Xie*中国科学院软件研究所，中国
xiexudong23@otcaix.iscas.ac.cn

Yuwei Zhang*‡中国科学院软件研究所，中国
zhangyuwei@iscas.ac.cn

Wensheng Dou*†‡中国科学院软件研究所，中国
wensheng@iscas.ac.cn

Yu Gao*中国科学院软件研究所，中国
gaoyu15@otcaix.iscas.ac.cn

Ziyu Cui*中国科学院软件研究所，中国
cuiziyu20@otcaix.iscas.ac.cn

Jiansen Song*中国科学院软件研究所，中国
songjiansen20@otcaix.iscas.ac.cn

Rui Yang*
中国科学院软件研究所，中国
yangrui22@otcaix.iscas.ac.cn

Jun Wei*†中国科学院软件研究所，中国
wj@otcaix.iscas.ac.cn

arXiv:2601.05579v1 [cs.DB] 9 Jan 2026

摘要

在不同关系数据库管理系统（RDBMS）之间转换 SQL 方言，对于将基于关系数据库管理系统的应用程序迁移到云端至关重要。传统的 SQL 方言转换工具依赖手工制定的规则，因此要支持新的关系数据库管理系统和方言，需要投入大量人工工作。尽管大语言模型（LLM）能够协助转换 SQL 方言，但它们在处理冗长而复杂的 SQL 查询时往往力有不逮。

本文提出 RISE，这是一种新颖的基于大语言模型的 SQL 方言翻译方法，能够准确处理冗长而复杂的 SQL 查询。给定一个包含某种 SQL 方言的复杂源查询 Q_c d，我们首先采用方言感知查询约简技术，通过移除 d中与 Q_c无关的 SQL 元素，得到简化查询 Q_s。随后，我们利用大语言模型将Q_s翻译为 $Q_{s'}$，并基于 Q_s与$Q_{s'}$之间的关系，自动提取适用于方言 d 的翻译规则 r_d。将r_d应用于 Q_c后，我们便能有效翻译 Q_c中的方言 d，从而避开源查询 Q_c的复杂性。我们在两个真实世界基准测试 TPC-DS 和 SQLProcBench 上评估 RISE，并从翻译准确率方面将其与传统的基于规则的工具和基于大语言模型的方法进行比较。RISE 在 TPC-DS 和 SQLProcBench 上的准确率分别达到 97.98% 和 100%，相比基线方法，平均提升幅度分别为 24.62% 和 238.41%。

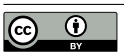
CCS 概念

• **软件及其工程** → **软件维护工具**;
• **计算方法** → 机器翻译。

*隶属于中国科学院系统软件重点实验室、中国科学院软件研究所计算机科学重点实验室和中国科学院大学，北京。CAS 是中国科学院的缩写。

†隶属于南京软件技术研究所和中国科学院大学，南京。

‡Yuwei Zhang 和 Wensheng Dou 是通讯作者。



本作品采用知识共享署名 4.0 国际许可协议授权。国际软件工程会议 '26，巴西里约热内卢© 2026 版权所有归作者/所有者所有。ACM ISBN 979-8-4007-2025-3/26/04 https://doi.org/10.1145/3744916.3773257

关键词

大型语 语言模型、SQL 方言、数据库管理 言模型

ACM 引用格式：
Xudong Xie、Yuwei Zhang、Wensheng Dou、Yu Gao、Ziyu Cui、Jiansen Song、Rui Yang 和 Jun Wei。2026。RISE：通过查询约简实现规则驱动的 SQL 方言转换。载于 *2026* 年 *IEEE/ACM* 第 *48* 届国际软件工程会议（*ICSE '26*），*2026* 年 *4* 月 *12—18* 日，巴西里约热内卢。ACM，纽约，NY，美国，12 页。https://doi.org/10.1145/3744916.3773257

1 引言

如今，超过 50 种关系数据库管理系统（RDBMSs）已在各类组织中广泛使用 [6], ，其中许多已部署在云上，以提升性能并降低成本。不同的关系数据库管理系统通常各具优势；例如，有些在可扩展性方面表现出色，而另一些则在安全性方面更胜一筹。为了实现这些优势，传统的基于关系数据库管理系统的应用程序需要迁移到合适的云端关系数据库管理系统中 [30, 38, 42]。

尽管许多关系数据库管理系统（例如 Oracle [14], 、MySQL [12],、PostgreSQL [15] 和 SQLite [17]）都支持结构化查询语言（SQL），但它们也包含供应商特定的扩展和独有细微差异，因此形成了种类繁多的 SQL 方言。这些不同 SQL 方言的存在带来了语法和语义上的差异，使得在不同关系数据库管理系统之间进行 SQL 方言翻译成为一项具有挑战性的问题。有效的 SQL 方言翻译应确保翻译前后的语义等价性，也就是说，应返回相同的查询结果和数据库状态。已有研究 [9] 表明，SQL 方言翻译通常会耗费 20% 至 40% 的迁移预算。

目前已有多种基于规则的工具 [5, 11, 16, 17]，用于支持不同关系数据库管理系统之间的 SQL 方言翻译。这些工具被广泛采用（例如，SQLGlot [16] 在 GitHub 上获得了 7.8k 个星标），Microsoft 和 Amazon [1, 4] 的专有解决方案也广受使用，这凸显了学术界和工业界 [42]解决 SQL 方言翻译难题的迫切需求。这些工具主要依赖人工编写的翻译规则，往往并不完整且容易出错。例如，现有工具无法正确翻译图 1 所示的复杂查询 Q_c。具体而言，jOOQ 和 SQLines 不支持全外连接子句，而 SQLGlot 则将 UNION 替换为 UNION ALL，生成了错误的查询，导致查询结果不一致。此外，开发和维护这些翻译规则所需的人工投入

相当可观。例如，SQLGlot 中有 1900 行代码专门用于 MySQL 到 PostgreSQL 的翻译。

近期，研究人员探索了使用大语言模型（LLM）来减少对传统工具中手工制定的规则的依赖 [5, 11, 16, 17]。例如，Mallet [30] 利用大语言模型生成翻译规则，随后由开发人员将这些规则手动集成到 SQLGlot [16] 中。因此，Mallet 继承了 SQLGlot 的许多局限性。另一种更直接的方法是提示大语言模型翻译 SQL 查询，称为 LLMTranslator；这是本文设计的一个实验性原型。然而，由于大语言模型常见的幻觉问题，LLMTranslator 难以处理冗长而复杂的查询 [22]。例如，在翻译图 1 中的复杂查询 Q_c 时，LLMTranslator 错误地将 'sum(sum(ws_sales_price))' 重写为 'sum(ws_sales_price)'，改变了查询原本的语义。

为了评估 LLMTranslator 的实际局限性，我们使用两个基准测试（即 TPC-DS [18]和 SQLProcBench [24]）开展了一项实证研究，这两个基准测试均源自真实世界工作负载。结果表明，LLMTranslator 在处理简单查询时表现良好，能够准确转换语法和语义都较为直接的 SQL 方言。然而，在处理冗长复杂的查询时，其效果会显著下降。在这类情况下，LLMTranslator 经常出现幻觉问题，导致翻译错误。这些发现表明，尽管 LLMTranslator 擅长简单的翻译任务，但由于与方言无关的噪声可能会掩盖核心语义，它在处理复杂查询时仍有困难。因此，隔离并过滤掉此类非必要元素，对于提高基于大语言模型的 SQL 方言翻译的准确率至关重要。

受上述观察启发，我们提出了 RISE，这是一种自动化的、基于**规**则的 SQL 方**言**转**换**方法，利用方言感知的查询简**化**技术。其核心思想是通过移除与方言无关的元素来简化复杂 SQL 查询。这种简化技术会生成仅保留目标 SQL 方言特定元素的简化查询。如图 1 所示，源查询 Q_c 长达 42 行，结构复杂，并包含两个 PostgreSQL 特有的方言元素（以红色标示）。通过我们的简化技术，Q_c 首先被转换为简化查询 Q_s，其中仅用两行便提取出 FULL OUTER JOIN 的方言特定元素。随后，我们提示大语言模型将 Q_s 翻译为 $Q_{s'}$；由于复杂度降低，大语言模型能够有效完成此任务。根据 Q_s 与 $Q_{s'}$ 之间的对应关系，我们为第一个方言特定元素 FULL OUTER JOIN 生成一条翻译规则 r_d。最后，我们将 r_d 应用回原始查询 Q_c，从而无需大语言模型处理整个复杂查询，即可准确、高效地转换方言特定元素。

我们在两个真实世界基准测试（即 TPC-DS 和 SQLProcBench）上评估了 RISE，这些基准测试分别要求将 PostgreSQL 方言翻译为 MySQL 和 Oracle 方言。我们将 RISE 与最先进的基线方法进行比较，包括传统的基于规则的工具（即 SQLines、jOOQ 和 SQLGlot）以及基于大语言模型的方法（即我们的原型 LLMTranslator 和 CrackSQL）。实验结果表明，在翻译准确率方面，RISE 大幅超越了所有基线方法。我们已将工具发布于 https://figshare.com/s/5c709f5c0a63d58b0ef4。

总而言之，我们作出以下贡献。

- 我们开展了首项实证研究，以评估 LLMTranslator 的有效性和局限性，为改进基于大语言模型的 SQL 方言翻译提供了宝贵见解。
- 我们提出了 RISE，这是一种新颖的基于大语言模型的 SQL 方言方法，能够准确处理冗长且复杂的 SQL 查询。RISE 将有效的方言感知查询归约技术与大语言模型辅助翻译规则生成器相结合，以克服朴素的大语言模型翻译方法的局限性。
- 我们在两个真实世界基准测试上进行了广泛实验，并对 RISE 的每个组件进行了全面评估。实验结果表明，RISE 优于最先进的基线方法，凸显了其卓越性能。

2 实证研究

鉴于大语言模型拥有广博的知识和强大的自然语言理解能力，利用它们进行 SQL 方言翻译是一种极具前景且灵活的方法。为了评估大语言模型翻译 SQL 方言的效果，我们实现了一个原型 LLMTranslator，并开展实证研究，以回答以下两个研究问题。

- **RQ1（翻译准确率）：** LLMTranslator 能准确翻译 SQL 方言吗?
- **RQ2（失败因素）：** 使用 LLMTranslator 进行 SQL 方言翻译时，哪些因素会导致翻译失败?

2.1 研究方法

下面介绍本实证研究中使用的基准测试、方法和评估指标。

2.1.1 基准测试。 我们使用以下两个基准测试中的查询来评估大语言模型在SQL 方言翻译a方面的能力:

- **TPC-DS：** TPC-DS [18] 是一项工业界标准基准测试，用于评估关系数据库管理系统在复杂分析查询上的性能。它包含99个查询，涵盖广泛的SQL结构，其中一些查询超过100行。这些查询包括基本运算符、聚合、函数和高级SQL特性（例如WITH 子句、公用表表达式、子查询）。TPC-DS体现了现实世界查询工作负载的复杂性和多样性，每个查询都经过精心设计，用于模拟实际的分析任务。因此，我们采用TPC-DS来评估大语言模型应对工业级SQL 方言翻译多方面挑战的能力。
- **SQLProcBench：** SQLProcBench [24]扩展了 TPC-DS 的数据库模式，重点关注过程式代码，包括在 PostgreSQL、Oracle 和 SQL Server 中实现的存储过程、函数和触发器。在评估中，我们聚焦于存储过程，其中包含 44 条语句，旨在模拟各种应用中的真实过程式代码用法，有些过程超过 50 行 SQL 代码。这些过程具有复杂的控制流（例如 IF-ELSE 条件、WHILE 和 FOR 循环）、独立的 SQL 语句（例如 SELECT、UPDATE、DELETE）、变量管理、异常处理等。通过捕捉翻译过程式逻辑时固有的挑战，SQLProcBench 成为评估 SQL 方言翻译方法在实际数据库迁移场景中有效性的关键基准测试。

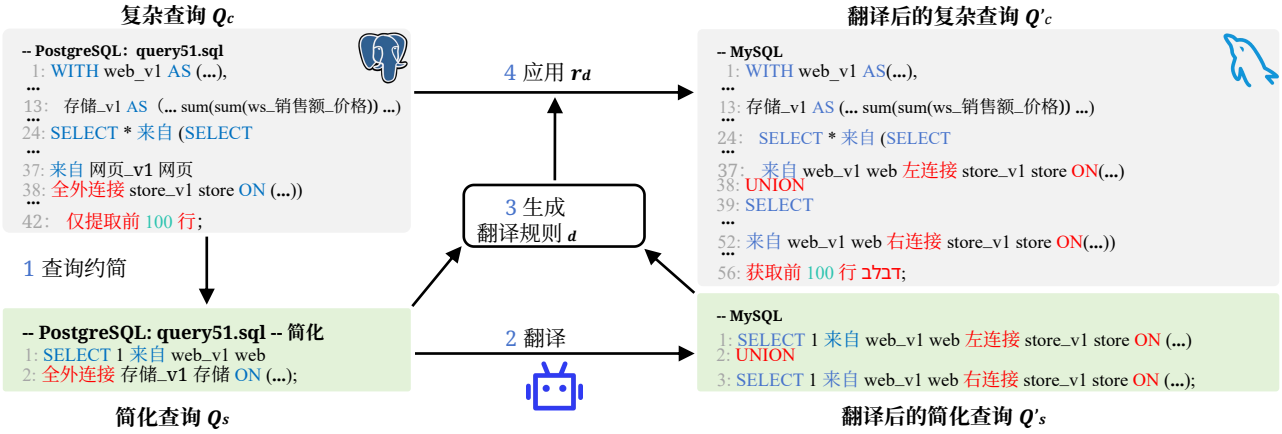


图 1：SQL 方言翻译示例。PostgreSQL 特有方言以红色标示。

2.1.2 方法。 我们开发了一个原型 LLMTranslator，用于评估基于大语言模型的 SQL 方言翻译，并作为后续比较的基线。LLMTranslator 采用思维链（CoT）策略 [27],，将翻译过程分为三个阶段。首先，我们分别在目标数据库和源数据库上执行源查询，以收集反馈，并将其作为错误消息（例如执行错误或结果不一致）。其次，我们使用大语言模型识别并总结源查询中的方言特定特性。最后，大语言模型结合错误反馈和总结出的方言特定特性完成翻译。

为进一步评估错误消息和方言特定特征摘要对转换性能的贡献，我们对 LLMTranslator 进行了消融研究。具体而言，我们评估了三个变体：一个不包含错误消息（不含错误信息），一个不包含特征摘要（不含摘要），以及一个两者均不包含的变体（单一大语言模型）。为确保研究结果具有普适性，我们使用两种具有代表性的大语言模型评估 LLMTranslator：DeepSeek-V3[8]（记为 LLMTranslator$_{DS}$）和 GPT-4o [10]（记为 LLMTranslator$_{GPT}$）。对于每个大语言模型及其变体，我们统计语法错误和不等价查询的数量，每个查询最多进行三次转换迭代。

2.1.3 评估指标。 翻译准确率通过准确率量化，并用作评估指标。我们在目标数据库管理系统上执行翻译后的查询，并将其结果与在源数据库管理系统上运行源查询所得的结果进行比较。如果执行失败或产生不一致的结果，则认为翻译不成功。对于结果一致的查询，我们会进一步进行人工验证，以确认翻译后的查询与源查询在语义上等价。只有在确认语义一致性后，才认为翻译成功。

2.2 翻译准确率

如表1中的实证结果所示，即使直接使用纯大语言模型（即单一大语言模型），也能够成功翻译大多数 SQL 方言，其中 GPT-4o 和 DeepSeek-V3 的准确率分别达到 76.22% 和 71.33%。

表1：LLMTranslator 及其变体的准确率

模型	语法错误	不等价	准确率
LLMTranslator$_{GPT}$	7	12	86.71%
不含错误信息	7	18 (50.00% ↑)	82.52%
无摘要	9 (28.57% ↑)	17 (41.67% ↑)	81.82%
单一大语言模型	13 (85.71% ↑)	21 (75.00% ↑)	76.22%
LLMTranslator$_{DS}$	9	16	82.52%
不含错误信息	10 (11.11% ↑)	24 (50.00% ↑)	76.22%
无摘要	12 (33.33% ↑)	20 (25.00% ↑)	77.62%
单一大语言模型	14 (55.56% ↑)	27 (68.75% ↑)	71.33%

O1：仅凭纯大语言模型即可正确翻译大多数方言。

有错误消息时，大语言模型可以识别方言特定元素的位置和性质，从而提升准确翻译 SQL 方言的能力。同样，特征摘要有助于大语言模型更准确地理解源查询的各种特征，进而减少错误翻译。移除错误消息（即不使用 ErrMsg）或特征摘要（即不使用 Summary），会使语法错误和不等价查询的数量分别增加约 34.09% 和 31.82%。

O2：错误消息和方言特定特征摘要均可提高 *SQL* 方言翻译的准确率。

2.3 失败因素

我们对 LLMTranslator$_{GPT}$ 和 LLMTranslator$_{DS}$生成的错误查询进行了详细分析。针对这 44 个错误案例，我们首先识别出方言特定元素，然后手动移除与方言无关的内容，生成简化版本，并使用 LLMTranslator 重新翻译这些查询。

我们发现，移除与方言无关的内容后，38.64% 的错误得以直接解决，因为这样可以避免不一致。

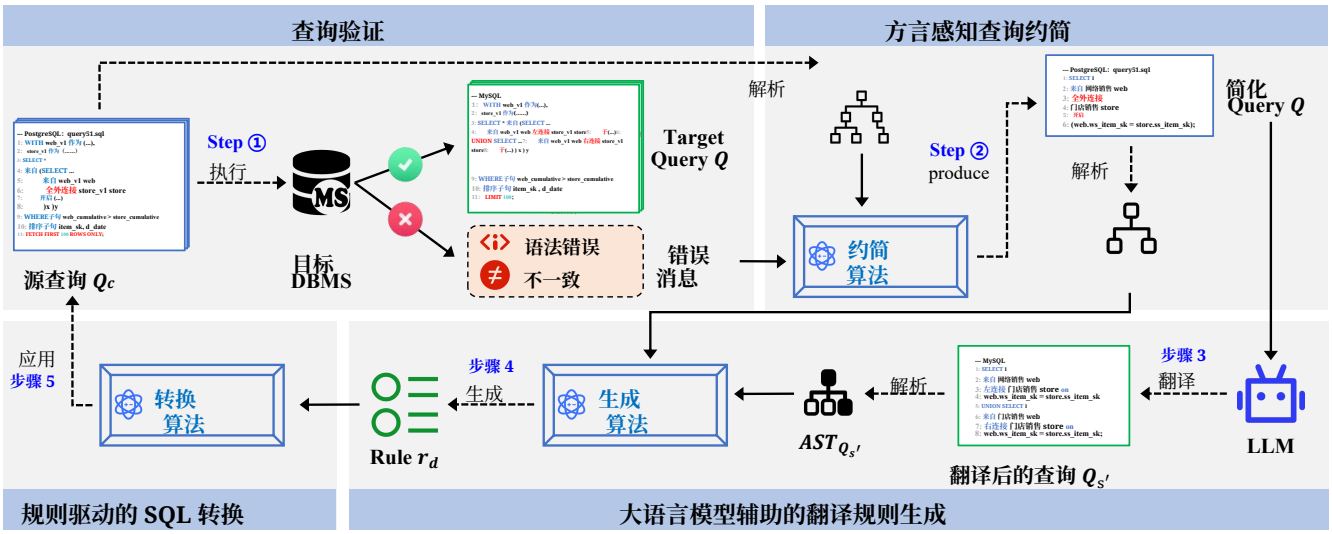


图 2：RISE 概览。

这些不一致是由大语言模型修改与方言无关的内容所导致的。例如，大语言模型会更改 GROUP BY 子句（例如，将 GROUP BY a, b, c 改为 GROUP BY a, b）、修改 SQL 查询中的列名和常量（如字符串），并执行不等价优化。这些优化包括将 Sum(Sum()) 这样的表达式改写为 Sum()，以及将隐式连接转换为显式 JOIN 操作，但连接条件却不正确。

在其余错误中，8 个由 GPT-4o 导致，19 个来自 DeepSeek-V3。应用查询约简后，LLMTranslator$_{GPT}$ 成功翻译了此前 8 个出错案例中的全部案例。在其中 5 个案例中，GPT-4o 最初遗漏了需要处理的方言。通过查询约简，这些方言得到了正确处理。在其余 3 个案例中，GPT-4o 受到冗长信息的误导，错误翻译了语义上不等价的 ROLLUP 函数。然而，约简后，GPT-4o 准确识别出细微的语义差异，并给出了正确的翻译。对于 LLMTranslator$_{DS}$，19 个翻译失败案例中，11 个涉及 ROLLUP 函数处理错误，7 个是由于未处理方言，还有 1 个涉及对全外连接子句的误解。约简后，除涉及 ROLLUP 的查询外，其他查询均成功翻译。这些结果表明，LLMTranslator 能够有效处理简化查询中的大多数 SQL 方言，但在处理更复杂的查询时则难以胜任。为了更好地理解 LLMTranslator 的局限性，我们进一步分析失败原因，并确定了三种会显著降低翻译准确率的主要大语言模型幻觉类型。

- 大语言模型会对与方言无关的部分进行不恰当的优化。
- 尽管大语言模型能够翻译方言，但往往无法准确处理复杂语境。

- 大语言模型会任意修改列名，遗漏某些元素（例如 GROUP BY 子句中的项目），或更改查询对象。

O3：大语言模型在有效处理冗长且复杂的 *SQL* 查询方面存在局限。

3 通过查询约简进行 SQL 方言翻译 我们提出了 RISE，一种基于大语言模型的自动化规则驱动的 SQL 方言转换方法。图 2 展示了 RISE 的概览。设 d 是需要翻译的方言。我们首先在目标数据库管理系统上对源查询 Q_c 执行查询验证（步骤 ①）。如果 Q_c 未产生语法错误，且在其目标数据库管理系统上产生相同结果，则无需进行翻译，并直接将其作为目标查询 Q_t 返回。否则，我们假设方言 d 存在。随后，我们对 Q_c 执行方言感知查询约简，其中自适应解析器将 Q_c 转换为相应的抽象语法树（AST）AST_{Q_c}。利用错误消息和 AST_{Q_c}，查询约简算法生成简化查询 Q_s，同时保留方言 d（步骤 ②）。下一步，我们直接利用大语言模型将 Q_s 翻译为 $Q_{s'}$（步骤 ③）。随后，自适应解析器将简化查询和转换后的查询（即 Q_s 和 $Q_{s'}$）转换为抽象语法树（即 AST_{Q_s} 和 $AST_{Q_{s'}}$），规则生成算法则根据这些抽象语法树推导出方言 d 的新翻译规则 r_d（步骤 ④）。最后，我们利用 r_d 翻译 Q_c 中的方言 d（步骤 ⑤），随后对 Q_c 执行查询验证。如果 Q_c 仍包含未解决的方言元素，或生成的规则无法翻译 Q_c 中的方言 d，我们将重复上述过程。否则，我们将 Q_c 作为目标查询 Q_t 返回。

3.1 查询验证

我们在流程的以下三个阶段进行查询验证。首先，我们将源查询 Q_c 视为目标数据库管理系统上的翻译后查询，以检测其 SQL 方言并获取错误消息。其次，我们检查大语言模型生成的译文，确保 SQL 方言已得到准确翻译。最后，我们评估规则驱动的译文，以确认方言处理得当，并找出尚未解决的 SQL 方言问题。

我们采用动态执行策略，判断翻译后的查询是否包含尚未解决的方言特定元素，并捕获相关错误消息。具体而言，我们分别在各自的

源数据库管理系统和目标数据库管理系统上执行翻译前和翻译后的查询，然后比较执行结果。每次验证前，我们都会将数据库重置为初始状态，以确保评估条件一致，不受可能已执行的任何中间语句影响。我们使用基准测试中包含的表数据进行验证。如果基准测试不包含此类数据，则可以使用现有方法生成这些数据 [19, 23, 26]。在以下情况下，可以判定翻译后的查询中存在尚未解决的方言特定元素：(1) 目标数据库管理系统无法执行翻译后的查询，此时数据库管理系统的反馈作为错误消息。(2) 目标数据库管理系统返回的查询结果与源查询返回的结果不同，此时结果之间的差异作为错误消息。(3) 目标数据库管理系统执行旨在修改数据库状态的查询后，存储的数据库状态有所不同。

采用这种简单却实用的方法（类似于 Mallet [30] 和 VeriEQL [25], 所采用的方法），我们能够有效验证转换前后 SQL 查询的语法有效性和语义等价性。

3.2 方言感知的查询约简

如果在查询验证步骤中识别出方言，我们会使用自适应解析器生成相应的抽象语法树（AST），并执行后续的查询约简。

3.2.1 自适应解析。 受 SQLess [28],启发，我们使用 ANTLR [2] 实现自适应解析器，将 SQL 查询解析为相应的抽象语法树。我们利用源数据库管理系统的 g4 语法文件 [3]，通过 ANTLR 生成基础解析器。随后，使用基础解析器将查询解析为抽象语法树。如果发生解析错误，我们会为原始 g4 语法文件生成补丁，以补全解析规则并解决问题。这一步与 SQLess 类似，旨在实现自适应解析器。然而，SQLess 不会分析无法解析部分的语义，而是直接将无法解析的字符串添加到解析规则的错误模式中，并不针对解析失败的字符串创建更细粒度的解析规则。相比之下，我们使用大语言模型，在解析错误消息的指导下分析无法解析的部分并生成新的解析规则。随后，我们会手动验证这些规则，并将其中有效的规则整合到原始 g4 语法文件中。这种方法有助于构建语义更丰富的补丁，并开发出更稳健、通用性更强的自适应解析器。

3.2.2 查询约简。 此组件旨在从源查询中尽可能多地删除与方言无关的元素，同时仅保留方言特定元素。请注意，已有许多程序规约方法 [29, 33, 34, 36, 39, 40] 被提出。在我们的方法中，我们利用 SQL 的抽象语法树结构进行基于语法的约简。具体而言，我们采用一种轻量但有效的方法[1]，随机删除查询抽象语法树中可移除的子树。随后，我们使用大语言模型进一步约简查询，优化剩余的查询结构。

步骤 1：基于语法的随机缩减。我们首先对从源查询解析得到的抽象语法树进行随机缩减。在每次迭代中，随机选择最多两个子树并将其删除，

[1]在我们的实验中，随机约简策略平均耗时 16 秒即可约简一个 SQL 查询，约简率达到 91%。

从而得到简化后的抽象语法树及其对应的简化查询。得到的简化查询必须满足两个约束：（1）它必须能在源数据库管理系统上成功执行；（2）它在目标数据库管理系统上仍必须产生相同的错误消息。第一个约束确保缩减过程中不会引入新的错误，第二个约束则保证方言特定元素得到完整保留。如果简化查询未能同时满足这两个约束，就会恢复已删除的子树，并在下一次迭代中选择其他子树。此过程会持续进行，直到无法再删除任何子树。

列表 1：基于大语言模型的缩减示例提示词

```
任务描述：将查询从 {source_db} 迁移到 {target_db}。简化以下查询，仅保留触发 所需的部分，以引发'{original_error}' 错误在{target_db} 中。查询：{sql} 思维链指令：找到错误，分析其原因，并删除无关内容。要求：1：尽可能缩短查询。2：使用原始表/列名。3：确保简化查询能在{source_db} 上编译。4：完整保留方言部分。返回格式：```sql 此处填写简化后的查询 ```
```

步骤 2：基于大语言模型的缩减。接下来，我们使用大语言模型进一步优化查询，因为在复杂约束（例如子查询和公用表表达式）下，仅靠随机缩减可能会保留与方言无关的元素。我们使用大语言模型提取并理解方言特有的部分，仅保留随机缩减方法生成的简化查询中的必要部分。示例提示见清单 1。我们不直接使用大语言模型缩减源查询，因为经验观察表明，大语言模型在处理较长的 SQL 查询时可能会引入各种错误。在此步骤中，大语言模型每次尝试缩减都会生成五个输出。对于每个输出，我们在目标数据库管理系统上执行生成的查询，获取新的错误消息，并计算新旧错误消息之间的相似度，以此为输出评分。如果所有输出的相似度分数都未超过 0.85（根据经验确定的阈值），则说明原始方言尚未得到完整保留。我们会收集失败的尝试及其失败原因，并将这些信息反馈到下一轮的大语言模型提示中。此过程以迭代方式重复，最多进行五次迭代。一旦某个输出符合条件，我们便在源数据库管理系统上执行该输出。如果执行没有错误，则接受该输出作为最终的简化查询；否则将其丢弃。

3.3 大语言模型辅助的翻译规则生成

该模块分为两个主要部分：查询翻译和规则生成。它利用大语言模型准确地翻译简化查询，并通过规则生成算法，根据简化查询与转换后查询之间的关系自动推导翻译规则。

3.3.1 查询翻译。获得简化查询后，我们使用第 2.1.2 节中介绍的 LLMTranslator 执行 SQL 方言翻译。翻译后的查询将经过第 3.1 节所述的验证流程。任何无法执行或生成的结果与简化查询不一致的翻译都会被

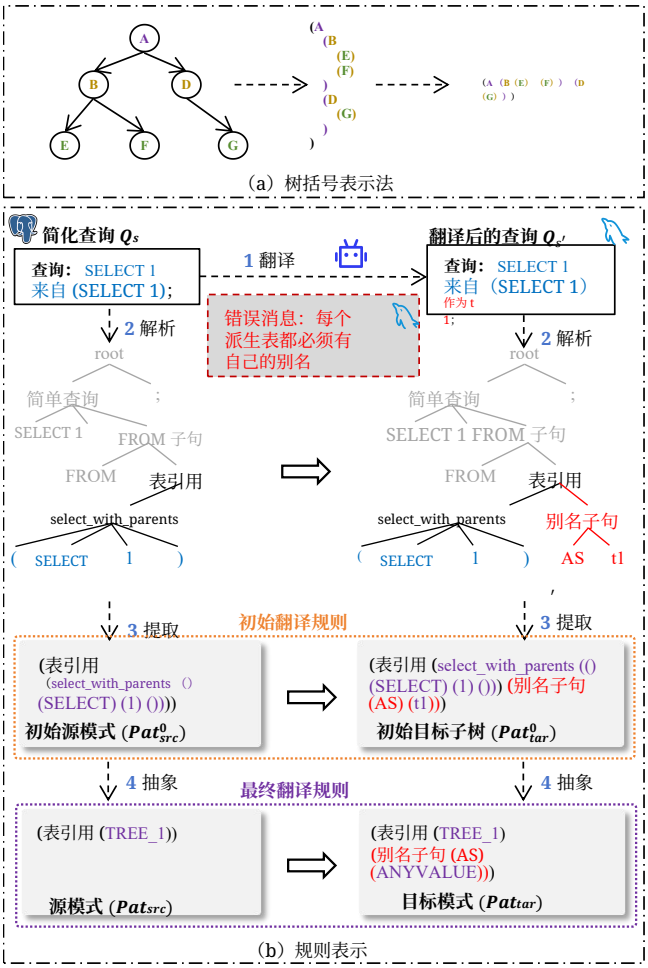


图3：翻译规则生成的示例。

丢弃，并再次向 LLMTranslator 提供简化查询提示，以生成新的翻译结果。

3.3.2 规则生成。该组件旨在根据简化查询和翻译后的查询自动推导出翻译规则。

翻译规则定义。我们将翻译规则定义为从源模式（Pat_{src}）到目标模式（Pat_{tar}）的映射，其中模式指定了表示该方言句法形式的抽象语法树结构。设 AST_{Q_s} 和 $AST_{Q_s'}$ 分别为简化查询和翻译后的查询的抽象语法树。当 AST_{Q_s} 包含与源模式匹配的树结构时，表明其中存在需要转换的方言特定构造。SQL 方言翻译通过将 AST_{Q_s} 中与源模式匹配的片段替换为相应的目标模式来实现，此过程会生成 $AST_{Q_s'}$。

图3展示了 MySQL 中对派生表强制赋予别名的规则，该规则要求所有派生表（例如子查询）都必须具有显式别名。为便于表示，我们使用括号表示法 [20] 来表示树结构。如图所示

算法1： 初始翻译规则提取

输入： 简化查询抽象语法树 AST_{Q_s} 及其对应的翻译后查询抽象语法树 $AST_{Q_{s'}}$

输出： 初始翻译规则 Pat^0_{src}，Pat^0_{tar}

```
1  ⟨Pat^0_src, Pat^0_tar⟩ ← 获取最小差异树(AST_Qs, AST_Qs')
2  函数 获取最小差异树(T_src, T_tar)
3      如果 AST_Qs.remove(T_src) ≠ AST_Qs'.remove(T_tar) 然后
4          返回 NULL
5      else
6          num ← max(T_src.root.childNum, T_tar.root.childNum)
7          对于 i ← 0; i < num; i ← i+1 执行
8              child_1 ← T_src.root.getChild(i)
9              child_2 ← T_tar.root.getChild(i)
10             如果 child_1 ≠ child_2，则
11                 返回 ⟨T_src, T_tar⟩
12             else
13                 rule ←获取最小差异树(child_1.subtree, child_2.subtree)
14                 如果 rule ≠ NULL，则
15                     返回 rule
16         返回 ⟨T_src, T_tar⟩
```

在图3 (a) 中，这种表示法使用嵌套括号来表示节点之间的父子关系，从而唯一确定一棵树。为提取规则，我们找出 T_{src} 中的最小差异子树和 AST_{Q_s} 中的最小差异子树，其中 T_{src} 和 T_{tar} 具有相同的根节点，其余部分的抽象语法树（即 AST_{Q_s}.remove(T_{src}) 和 $AST_{Q_s'}$.remove(T_{tar})）完全相同。如图3 (b) 所示，最小差异子树（即 T_{src} 和 T_{tar}）对应于以表_ref 节点为根的树。我们通过对这两棵子树进行转换来实现方言转换。具体而言，从 AST_{Q_s} 中提取的子树 T_{src} 充当该规则的源模式，而从 $AST_{Q_s'}$ 中提取的另一棵子树 T_{tar} 则构成目标模式。

规则提取。如算法1所述，RISE 通过以下步骤自动提取初始规则：

- 我们使用 getMinDiffTree() 函数，从 AST_{Q_s} 和 $AST_{Q_s'}$ 中提取差异最小的子树，并将其用作初始规则的源模式和目标模式（第 1 行）。
- 在 getMinDiffTree() 中，我们首先检查分别从 AST_{Q_s} 和 $AST_{Q_s'}$ 中移除 T_{src} 和 T_{tar} 后得到的树是否相同（第 3 行）。如果不同，该函数将直接返回 NULL，表示当前子树 T_{src} 和 T_{tar} 不符合要求（第 4 行）。
- 如果树的剩余部分相同，则表示当前子树 T_{src} 和 T_{tar} 符合要求。为了找出其中是否存在更小的差异子树，我们会检查 T_{src} 和 T_{tar} 中根节点的所有子节点是否相同。如果有任意一对子节点不匹配（第 6–10 行），则 T_{src} 和 T_{tar} 就是差异最小的子树，我们会直接返回它们（第 11 行）。
- 否则，我们会递归地对它们各自对应的子树调用该函数（第 12–13 行）。如果返回了非空结果，它就是最小差异子树对，我们会立即将其返回（第 14–15 行）。如果所有递归调用都返回空值，T_{src} 和 T_{tar} 就会被确定为最小差异子树，并直接返回（第 16 行）。在图3中，这一过程依次识别出以下子树对，其

算法 2： 初始转换规则抽象

```
输入：初始翻译规则 Pat^0_src，Pat^0_tar
输出：最终翻译规则 Pat_src，Pat_tar
1  Pat_src ← Pat^0_src
2  Pat_tar ← Pat^0_tar
3  抽象规则(Pat^0_tar.root)
4  函数抽象规则(curNode)
5      如果 isIdentifier(curNode) 则
6          如果 Pat_src.contains(curNode) 则
7              replaceAll(Pat_src, Pat_tar, curNode, $N)
8          else
9              replaceAll(Pat_src, Pat_tar, curNode, ANYVALUE)
10     否则如果 isConst(curNode) 则
11         如果 Pat_src.contains(curNode) 则
12             replaceAll(Pat_src, Pat_tar, curNode, $N)
13     否则如果 isNonTerminal(curNode) 则
14         如果 Pat_src.contains(curNode.subtree) 则
15             replaceAll(Pat_src, Pat_tar, curNode.subtree, TREE_N)
16         else
17             对于每个 childNode ∈ curNode.getChildren() 执行
18                 抽象规则(childNode)
```

根节点分别为： <根节点、根节点>、 <简单_SELECT、简单_SELECT>、<来自_子句、来自_子句>，以及 <表_引用、表_引用>。

然而，图3中提取的初始规则适用性有限，难以扩展到包含相同方言的更广泛查询。规则中包含与方言无关的内容，会对其匹配条件产生不利影响。具体而言，初始源模式中的“(SELECT) (1)”将该规则限制为只能匹配子查询为“SELECT 1”的情况。子查询内容的任何变化都会使该规则不再适用。然而，子查询的内容与将别名分配给子查询这一方言特定操作无关。无论子查询的内部结构如何，别名分配都不会受到影响。这凸显了在规则中抽象与方言无关内容的必要性。此外，所有 SQL 语句都要求标识符（例如表名和列名）在各自作用域内保持唯一。在此示例中，该规则始终为子查询分配相同的别名“t1”。如果一个 SQL 语句中的多个子查询需要别名，该规则势必会导致命名冲突。因此，同样有必要抽象新生成标识符的名称，确保为每个标识符分配一个唯一的随机生成值。

规则抽象。如算法 2 所述，RISE 使用深度优先搜索遍历所有子树（第 17–18 行），检查是否满足三种类型的抽象条件。

- 节点级抽象。 我们假设标识符的名称和常量的值与方言无关。为了规范化模式，我们将两个模式中相同的标识符名称或常量值替换为符号 $N（其中 N 为下标），该符号表示任意一个类型相同的值（第 5–7、10–12 行）。例如，在方言转换“FETCH FIRST 100 ROWS ONLY → LIMIT 100”中，整数值 100 与方言无关。无论该值是 100 还是 10，都不会影响这种方言转换。因此，我们将其替换为 $1，以表示任意整数值，得到“FETCH FIRST $1 ROWS ONLY → LIMIT $1”。

算法 3： 规则驱动的 SQL 转换

```
输入：源查询的抽象语法树 AST_Qc，翻译规则 Pat_src，Pat_tar
输出：转换后查询的抽象语法树 AST_Qc'
1  AST_Qc' ← AST_Qc
2  对每个 subTree ∈ AST_Qc 执行
3  如果 模式匹配(subTree, Pat_src) 那么
4      newSubTree ← instantiate(subTree, Pat_src, Pat_tar)
5      replaceSubTree(AST_Qc', subTree, newSubTree)
6  函数 模式匹配(subTree, Pat_src)
7      /* 抽象节点：$N 和 树_N */
8      如果 isAbstractNode(Pat_src.root) ∧ Pat_src.root.getType() =
         subTree.root.getType() 那么
9          返回 True
10     否则如果 Pat_src.root ≠ subTree.root 那么
11         返回 False
12     else
13         num ←
             max(Pat_src.root.childNum, subTree.root.childNum)
14         对于 i ← 0; i < num; i ← i + 1 执行
15             child_1 ← subTree.root.getChild(i)
16             child_2 ← Pat_src.root.getChild(i)
17             如果 ¬模式匹配(child_1.subtree, child_2.subtree) 则
18                 返回 False
19         返回 True
```

- 新标识符抽象。 如果目标模式包含新引入的标识符，我们就将其替换为 ANYVALUE，它表示随机生成的唯一值（第 8–9 行）。例如，在图3中，标识符 t1 被替换为 ANYVALUE。• 树级抽象。 我们假设在方言转换过程中保持不变的子树与方言无关，因此将其替换为 TREE_N，它表示以特定类型节点为根的任意一棵树（第 13–15 行）。例如，在图3中，子查询“SELECT 1”被抽象为 TREE_1，表示根节点为“select_with_parents”的任意子树。

这一两阶段流程通过识别源抽象语法树与目标抽象语法树在同一节点下最小的差异子树，提取翻译规则。它还通过抽象模式中与方言无关的内容，进一步增强这些规则的通用性。

3.4 规则驱动的 SQL 转换

在此步骤中，我们利用生成的规则执行 SQL 方言翻译。如算法 3 所示，我们在源查询的抽象语法树中搜索与规则的源模式匹配的子树。如果找到匹配的子树，则根据目标模式将其替换。此过程包括以下步骤：

- 源模式匹配。 我们遍历原始源查询抽象语法树中的所有子树 AST_{Q_c}，找出与源模式匹配的子树（第 2–3 行）。在匹配阶段（第 6–19 行），我们使用深度优先搜索遍历输入子树和源模式，检查二者是否相同（第 10–19 行）。如果源模式的根节点是抽象节点（即 $N and TREE_N），只要子树根节点的类型与源模式根节点的类型匹配，我们就认为匹配成功，并跳过后续递归（第 8–9 行）。

- 实例化。我们将目标模式中的抽象符号（例如 $N、TREE_N）替换为从源模式中匹配到的值

并将ANYVALUE替换为一个随机的唯一值。此步骤会生成一个具体的子树（第 4 行）。• 替换。我们将源模式匹配到的子树替换为实例化后的新子树，从而完成方言转换（第 5 行）。

通过遵循这些步骤，我们能够准确识别并转换 SQL 查询中的方言特定结构，从而促进不同 SQL 方言之间的无缝转换。

4 评估

为评估 RISE 的有效性，我们提出以下四个研究问题：

- **RQ3（准确率比较）：** RISE 在准确率方面与最先进的基线方法相比如何?
- **研究问题4（查询约简效果）：** 查询约简在准确率和有效规则生成方面的效果如何?
- **研究问题5（效率比较）：** 与其他基于大语言模型的方法相比，RISE 的效率如何?
- **研究问题6（生成规则分析）：** RISE 能生成哪些翻译规则?

4.1 实验设置

4.1.1 基准测试。由于现有用于评估大语言模型在 SQL 方言翻译方面表现的实验数据往往无法体现真实查询的复杂性 [42], ，我们采用 TPC-DS [18] 和 SQL-ProcBench [24]（见第 2.1.1 节）作为基准测试。

4.1.2 基线方法与指标。我们将 RISE 与以下六种基线方法进行比较。由于 Mallet [30] 未公开发布，我们无法将其纳入基线方法。我们使用准确率评估 RISE 的转换能力，这与第 2.1.3 节所述指标一致。

- **SQLGlot：** SQLGlot [16] 是一款开源 SQL 解析器和转换器，采用手工编写的规则在不同方言之间翻译查询。
- **SQLines：** SQLines [17] 通过基于规则的查询翻译，实现不同数据库管理系统之间的数据库模式和 SQL 迁移。
- **jOOQ：** jOOQ [11] 是一个 Java 库，可用于类型安全的 SQL 构造，并内置了从 Java 代码进行方言转换的功能。
- **`LLMTranslator`：** LLMTranslator 使用思维链策略进行方言转换，并采用两种不同的大语言模型配置：一种使用 DeepSeek-V3（$\text{LLMTranslator}_{DS}$），另一种使用 GPT-4o（$\text{LLMTranslator}_{GPT}$）。详情见第 2.1.2 节。
- **CrackSQL：** CrackSQL [41]使用带有跨方言嵌入的大语言模型，并采用由局部到全局策略，通过句法分割和匹配来翻译查询。

4.1.3 实现。 下文将介绍实现细节和大语言模型设置，以便于复现。

- **自适应解析器构建**。 由于 PostgreSQL [15] 是所有实验的源数据库管理系统，我们基于 PostgreSQLLexer.g4 和 PostgreSQLParser.g4 语法文件 [3], 构建自适应解析器，并使用 ANTLR 4.13.2 [2] 版本进行构建。
- **构建 TPC-DS**。官方 TPC-DS 基准测试[18] 缺少 PostgreSQL 模板。DB2 查询 [7] 与之最为相似，其中 99 条查询有 79 条可在 PostgreSQL 上编译通过（而在 MySQL 上只有 12 条）。经过人工修正后，我们创建了一个兼容 PostgreSQL 的 TPC-DS 版本，其中所有查询都能在 PostgreSQL 上成功执行，而在 MySQL 上只有 12 条能够成功编译并运行。这个调整后的数据集构成了我们的 TPC-DS 基准测试。
- **大语言模型设置**。 RISE 中使用的所有大语言模型均为 **GPT-4o** [10],，这是一种高度先进的大语言模型。对于 SQL 精简，大语言模型的温度设置为 0.7，令牌上限设置为 2048，以确保输出全面。在 LLMTranslator 中，用于特征摘要的大语言模型温度设置为 0，令牌上限设置为 2048，以确保输出具有确定性；用于翻译的大语言模型温度设置为 0.7，令牌上限相同。RISE 和 LLMTranslator 的最大迭代次数均设置为三次。为确保结果可靠，我们对 RISE 进行重复实验，以减轻大语言模型引入的随机性。

4.2 准确率比较

我们使用两个真实世界基准 TPC-DS [18]和 SQLProcBench [24]，将 RISE 与六种基线方法（即 SQLines [17], SQLGlot [16], jOOQ [11], $\text{LLMTranslator}_{DS}$、$\text{LLMTranslator}_{GPT}$ 和 CrackSQL [41]）进行了全面比较。在 TPC-DS 上，我们将查询从 PostgreSQL [15] 翻译为 MySQL [12],；而在 SQLProcBench 上，我们将存储过程从 PostgreSQL 翻译为 Oracle [14]。对于三种 SQL 方言转换工具，我们依靠其在线界面手动翻译查询。对于 CrackSQL，我们使用其开源实现，并采用默认配置和知识库，以确保公平比较。

4.2.1 实验评估。 图 2 展示了 RISE 和选定基线方法在 SQL 方言翻译中的准确率结果。我们的实验揭示了以下三项关键发现。

`RISE` 在两个基准测试中均展现出优于所有基线方法的性能。 总体而言，RISE 在 TPC-DS 上达到了 97.98% 的准确率，在 SQLProcBench 上达到了 100.00% 的准确率。值得注意的是，在 TPC-DS 上，其准确率比最佳基线 $\text{LLMTranslator}_{GPT}$ 高出 7.78%；在 SQLProcBench 上，其准确率比最佳基线 $\text{LLMTranslator}_{DS}$ 高出 22.22%。这些提升充分证明了 RISE 在 SQL 方言翻译任务中的有效性。

总体而言，基于大语言模型的方法优于传统自动化工具。直接使用大语言模型进行 SQL 方言翻译（即 $\text{LLMTranslator}_{DS}$ 和 $\text{LLMTranslator}_{GPT}$），在 TPC-DS 和 SQLProcBench 上的平均准确率分别达到 86.87% 和 79.55%，展现了基于大语言模型的方法的潜力。此外，CrackSQL 在 TPC-DS 上的准确率为 80.81%，但在 SQLProcBench 上表现不佳，准确率仅为 4.55%。这一糟糕表现归因于其缺乏关于存储过程的知识。相比之下，三种自动化 SQL 方言翻译工具在 TPC-DS 上的准确率约为 70%，面对更复杂的 SQLProcBench 时则表现吃力。传统工具依赖手工制定的规则，而这些规则往往不完整且容易出错。这些局限凸显了对更先进、更灵活的翻译机制的需求，以确保 SQL 方言翻译有效且准确。

高级SQL特性（例如存储过程）会显著影响传统SQL 方言转换工具的准确率。值得注意的是，在 SQLProcBench 基准测试中，由于缺少处理存储过程所需的翻译规则，现有工具的准确率接近于零。

表 2：不同方法的翻译准确率

数据集	jOOQ	SQLines	SQLGlot	LLMTranslator$_{DS}$	LLMTranslator$_{GPT}$	CrackSQL	RISE
TPC-DS	73.74%	67.68%	75.76%	82.83%	90.91%	80.81%	**97.98%**
SQLProcBench	13.64%	0.00%	0.00%	81.82%	77.27%	4.55%	**100.00%**



图4：MySQL 与 PostgreSQL 之间的 ROLLUP 语义差异。

4.2.2 错误案例分析。我们进一步对 RISE 和所选基线生成的不正确 SQL 方言翻译中观察到的常见错误类别进行了深入调查。具体而言，我们人工分析了每个错误翻译案例的根本原因，并据此进行分类。根据统计结果，我们的分析得出以下发现。

RISE 的错误分析：RISE。TPC-DS 中只有两个查询因语义不一致而失败。例如，符号'||'在 MySQL 中用作逻辑或运算符，但在 PostgreSQL 中用于字符串拼接。然而，由于结果集为空，RISE 无法自动检测到这种语义不一致，因此未能翻译这些查询。除这些异常外，RISE 成功翻译了所有其他查询。

传统 SQL 方言转换工具的错误分析。我们的分析将传统工具在 TPC-DS基准测试 上的失败分为三类：**解析错误**、**错误的翻译规则**和**缺失的翻译规则**。例如，jOOQ 在处理复杂查询时会出现解析错误（例如，超过 100 行的查询 23），这凸显了它在处理复杂 SQL 特性方面的局限性。在错误翻译规则方面，SQLGlot 在处理'ORDER BY'子句时会引入错误，而 jOOQ 在翻译'INTERVAL'函数时会遇到问题。在缺失翻译规则方面，SQLGlot 不支持为派生表指定别名，jOOQ 无法处理'FULL OUTER JOIN'，而 SQLines 在处理'INTERVAL'函数、'NUMERIC'类型转换和'FULL OUTER JOIN'操作方面存在不足。值得注意的是，SQLines 偶尔无法翻译'提取前 n 行'子句，这表明其针对这一特性的规则缺乏稳健性。如图4所示，'ROLLUP'操作在 MySQL 和 PostgreSQL 之间存在细微的语义差异，而这三种传统工具都无法正确处理'ROLLUP'，这表明处理高级SQL特性面临着更广泛的挑战。值得注意的是，RISE 借助其 LLM辅助翻译规则生成模块 有效解决了这些问题。通过利用大语言模型丰富的领域知识，RISE 自动生成翻译规则，克服了传统工具在规则充分性方面的局限。

基于大语言模型的方法的错误分析。如第 2.3 节所述，LLMTranslator 的错误主要源于大语言模型的幻觉，包括错误地修改与方言无关的内容，以及未能妥善处理方言特定构造。RISE

表3：消融研究结果

模型	TPC-DS		SQLProcBench	
	Acc	NVR	Acc	NVR
RISE	97.98%	13	100.00%	37
不使用 QR	81.82% (16.49% ↓)	3 (76.92% ↓)	72.73% (27.27% ↓)	20 (45.95% ↓)

通过方言感知查询约简应对这些挑战。CrackSQL 也存在明显局限，主要是过度依赖知识库，且对存储过程的处理能力较差。由于缺少存储过程信息，其在 SQLProcBench 上的表现受到严重影响，68.18% 的错误是由错误地判定为"无法翻译！"造成的。在 TPC-DS 上，CrackSQL 出现了 19 个问题：1 个判定错误、2 次崩溃（"列表索引超出范围"和"max_tokens 过大"）、2 个语法错误，以及 14 个语义不一致——这些问题都源于大语言模型的幻觉。例如，CrackSQL 错误地认为 MySQL 不支持 INTERSECT（这与官方文档 [13] 相矛盾），于是将其替换为 UNION。通过移除与方言无关的元素（如 INTERSECT），RISE 能够避免大语言模型引入此类错误，而查询约简则进一步提升了对方言特定构造的准确率。

> ***RQ3*** 的回答：RISE在准确率方面显著优于六种基线方法，凸显了其在翻译 *SQL* 方言方面的有效性。

4.3 查询约简的有效性

为评估查询约简（QR）对 RISE 的影响，我们在 TPC-DS 和 SQLProcBench 上进行了消融研究。该研究评估了移除 QR模块后对 RISE 翻译准确率（Acc）以及生成的有效规则数量（NVR）的影响。

4.3.1 结果分析。如表3所示，移除 QR模块（即 w/o QR）会显著影响 RISE 的整体性能。具体而言，不进行约简而直接处理源查询，会使准确率分别下降 16.49%（TPC-DS）和 27.27%（SQLProcBench），生成的有效规则数量则分别减少 76.92% 和 45.95%。QR 的必要性体现在两个方面。

防止不必要的修改。如果没有 QR模块，大语言模型可能会在翻译过程中无意修改与方言无关的元素。即使这些修改不会改变最终查询的等价性，也可能限制所生成规则的普适性。举例来说，考虑这样一种情况：大语言模型会在表名和

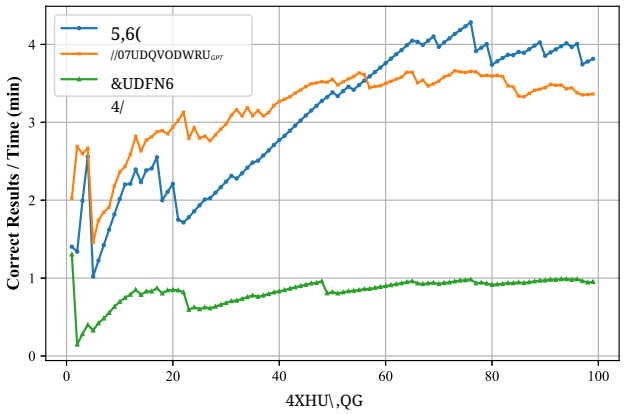


图5：RISE 与基于大语言模型的方法在单位时间内正确翻译数方面的效率比较。

别名之间多余地添加“AS”——这在其输出中很常见。如果规则 r 要翻译方言 d，那么这个多余的“AS”会将 r 限制为 d 中别名前没有“AS”的查询。更关键的是，错误修改（例如第 2.3 节中讨论的将“SUM(SUM())”转换为“SUM()”）可能会破坏查询等价性，却不会触发编译器警告，因此极难自动检测此类语义不一致。

查询长度缩减。 QR模块通过移除与方言无关的元素来简化源SQL查询。如**O3**所示，大语言模型难以处理冗长且复杂的查询。通过简化查询，QR模块提升了大语言模型翻译 SQL 方言的准确率。

> ***RQ4*** 的回答：*SQL* 精简显著提升了 *SQL* 方言翻译在准确率和有效规则生成方面的表现。

4.4 效率比较

研究表明，RISE 的准确率高于其他方法。鉴于此，我们进一步比较 RISE 与其他基于大语言模型的方法（即 LLMTranslator$_{GPT}$ 和 CrackSQL [41]）在 TPC-DS [18]上的翻译效率。为确保公平，所有方法均使用相同的大语言模型 GPT-4o [10]。基准测试指标定义如下：

单位时间内的正确翻译数。 令 n 表示正确翻译的查询数，t 表示总翻译时间（分钟）。效率指标通过计算比率 $\frac{n}{t}$ 得出。该指标表示每分钟正确翻译的查询数，可用于定量比较 SQL 方言翻译的效率。

如图5所示，由于 RISE 从空规则集开始，且复杂度较高，因此生成有效规则需要更长时间，效率低于 LLMTranslator$_{GPT}$。不过，在积累了足够的规则后（约在第 57 个查询时），它便可直接应用这些规则进行翻译，达到每分钟 3.78 条正确译文的稳定速率，超过每分钟处理 3.3 条译文的 LLMTranslator$_{GPT}$；后者会独立处理每个查询。

随着数据集规模增大，预计性能差距会进一步扩大。然而，由于 CrackSQL 的复杂度很高，其效率始终最低。最后，RISE、LLMTranslator$_{GPT}$ 和 CrackSQL 完成 99 条翻译所需的总时间分别为 25.42 分钟、26.75 分钟和 84.02 分钟。

> 研究问题***5***的答案：与其他基于大语言模型的方法相比，RISE 最终实现了更高的效率。

4.5 生成规则分析

如表3所示，RISE 为 TPC-DS [18]生成了 13 条规则，为 SQLProcBench [24]生成了 37 条规则。经人工检查，确认所有生成的规则均有效。

对于 TPC-DS，RISE 生成了 6 条规则来处理方言特定构造：全外连接、时间间隔、转换为 NUMERIC/INT、派生表别名以及提取前 N 行。对于 SQLProcBench，它生成了 22 条规则，涵盖类型转换（整数、字符、十进制数、可变长字符、时间类型）和存储过程构造（例如，声明、游标、返回值、异常、如果-否则、赋值、LIMIT）。以下是评估过程中生成的两条简化规则。

清单2：INTERVAL 函数翻译规则

```
源模式：(...( A_expr_add(TREE_1)(+) (...(
Constinterval(INTERVAL))(Sconst(Anysconst('60_ days '))))) 目
标模式：(...( Type_function_name (Identifier( DATE_ADD))...( A_
expr_add(TREE_1)))(,)( Func_arg_expr(A_expr(INTERVAL)(
Interval_value( Iconst (60))(Time_unit(DAY)))))
```

清单2展示了 RISE 在 TPC-DS 中生成的一条规则，用于翻译 INTERVAL 函数；现有工具（如 jOOQ [11] 和 SQLines [17]）无法正确处理该函数。PostgreSQL 要求将 INTERVAL 中的时间值用引号括起来（例如 ‘60 days’），而 MySQL 禁止给此类值加引号。然而，简单的字符串匹配方法可能会修改与时间值相似、但无关的字符串字面量。RISE 使用规则驱动的 SQL 转换算法（算法 3）准确识别 INTERVAL 参数并应用正确的转换，从而解决了这一问题。

清单 3：过程声明转换规则

```
源模式 :( Createfunc_opt_item(LANGUAGE)(plpgsql)(AS)(Func_
as(Plsql_block($$)...($$)))) 目标模式 :( Createfunc_opt_
item(IS)(Func_as(Plsql_block...)))
```

清单 3 展示了一条将 PostgreSQL 存储过程转换为 Oracle 存储过程的规则：移除“LANGUAGE plpgsql”，将“AS $$”替换为“IS”，并省略结尾的“$$”分隔符。具体而言，PostgreSQL 使用“LANGUAGE plpgsql”指定过程语言，并使用“$$”作为过程体分隔符；而 Oracle 省略语言指定，并使用“IS”开始过程体，且不使用结束分隔符。例如，声明为“CREATE OR REPLACE PROCEDURE ... LANGUAGE plpgsql AS $$ ... $$”的 PostgreSQL 过程，在 Oracle 中会变为“CREATE OR REPLACE PROCEDURE ... IS ...”，从而确保语法兼容性，同时保留过程逻辑。

> 对研究问题 ***RQ6*** 的回答：在 *TPC-DS* 和 *SQLProcBench* 上，RISE 生成了大量多样的有效规则。

5 讨论

本节讨论我们的方法所面临的有效性威胁和局限性。

5.1 有效性威胁

RISE 有效性的主要威胁在于用于评估的 SQL 查询多样性，以及其对其他数据库管理系统的普适性。本文从 TPC-DS [18] 和 SQLProcBench [24], 中收集了共 143 条 SQL 查询，确保评估具备合理的多样性和质量。为进行比较，我们选取了三种传统的 SQL 方言转换工具 [11, 16, 17]，以及三种基于大语言模型的方法：CrackSQL [41] 和我们的实验原型 LLMTranslator$_{DS}$，以及 LLMTranslator$_{GPT}$。未来，我们计划扩充基准测试，以更全面地评估 RISE 对各类数据库管理系统和新兴 SQL 方言的普适性。

5.2 局限性

语义不一致的存在。目前，RISE 无法完全解决语义不一致问题，因为仅依赖数据库管理系统的执行结果无法确保完全准确。例如，如 4.2.2 节所述，RISE 错误地识别了"||"运算符；该运算符在 PostgreSQL 中表示字符串拼接，在 MySQL 中则表示逻辑或。这一错误导致源查询与转换后的查询之间出现语义不一致，进而造成翻译失败。此外，由于数据不满足查询谓词条件，两个数据库管理系统均返回了空结果，导致 RISE 错误地认定这些查询在语义上等价。要在不同数据库管理系统之间自动验证源查询及其对应转换后查询的语义等价性颇具挑战。解决 SQL 方言翻译任务中的语义不一致问题仍是我们未来的工作。

无法处理语义存在歧义的方言。某些 SQL 方言存在语义歧义，因为其行为取决于数据库中的模式信息，无法仅根据 SQL 查询确定。例如，PostgreSQL 中的除法运算符（'/'）有两种可能的语义：整数除法和浮点除法。具体行为取决于操作数：如果操作数中包含浮点数，则表示浮点除法；否则表示整数除法。由于目前定义的翻译规则未考虑模式信息，RISE 无法稳健地翻译诸如 SELECT t1.c1 / t2.c2 这样的查询，因此翻译规则可能不可靠。未来，我们计划扩展规则，将模式信息纳入考量。

6 相关工作

基于手工编写规则的 SQL 方言翻译。目前大多数 SQL 方言转换工具 [5, 11, 16, 17] 都由规则驱动，利用了 SQL 方言语法有限且具有确定性的特点。应对反复出现的错误案例的一种直接策略是设计新规则，然后将其集成到转换工具中，以支持全面的 SQL 迁移。然而，这种方法存在一个明显的局限：规则必须手动构建。这个过程不仅耗时，还要求对所涉及的 SQL 方言有深入了解。

基于大语言模型的 SQL 方言翻译。大语言模型展现出了显著的代码编写能力 [31], ，通过减少人工工作，为 SQL 方言翻译提供了有前景的解决方案。Mallet[30] 利用大语言模型自动生成翻译规则，从而减少对手动构建规则的依赖，以支持尚未支持的 SQL 方言。然而，生成的规则往往相对简单，限制了 Mallet 处理存储过程等复杂 SQL 特性的能力。此外，Mallet 无法独立执行 SQL 方言翻译；它依赖传统工具 SQLGlot [16] 来执行，这使其适用范围仅限于 SQLGlot 已支持的数据库管理系统。为缓解这些问题，近期出现的一种名为 CrackSQL [41] 的方法引入了更结构化的框架。该方法将复杂查询分割成更小且功能连贯的单元，并应用函数规范化和无关查询抽象来简化翻译，从而提高稳健性并减少错误。然而，大语言模型生成输出的可靠性仍然是一个重要问题 [31, 35]。例如，尽管 CrackSQL [41] 将大语言模型与知识库结合起来进行验证，但我们的结果显示，其仍频繁出现语法错误和语义不一致。这些发现凸显了仅依赖大语言模型进行准确 SQL 翻译的局限性。

程序约简。人们从不同角度提出了许多程序约简方法。HDD [29]和 T-PDD [34] 利用层次结构和概率建模来指导约简过程。LPR [39] 引入大语言模型，以提升灵活性和自动化程度。Perses [33] 和 T-Rec [36]利用形式文法和词法文法指导，确保约简过程中的语法正确性。相比之下，PPR [40] 通过成对程序约简探索了一种截然不同的方法。其中，PPR 由于依赖成对比较，与 SQL 查询的单查询特性相冲突，因此不太适用于 SQL 查询约简。其他方法则更适用，并有望提高 SQL 精简率。尽管现有数据库测试方法 [21, 32, 37] 主要依赖简单的抽象语法树随机删除，但将这些技术应用于 SQL 精简仍是一个重要的研究方向。

7 结论

本文首先通过一项实证研究，探讨大语言模型在 SQL 方言翻译中的能力与局限性。基于观察结果，我们提出了一种新方法 RISE，用于在不同数据库管理系统之间高效、准确地翻译 SQL 方言。具体而言，我们首先采用方言感知查询约简技术，从源查询中移除与方言无关的元素。随后，我们开发了一种大语言模型辅助翻译规则生成算法，根据简化查询自动生成翻译规则，从而无论查询复杂度如何，都能有效实现 SQL 方言翻译。在两个真实世界基准上的全面实验表明，RISE 确实有效。

致谢

本研究部分得到了中国国家自然科学基金（62302493）、中国科学院软件研究所基础研究项目（ISCAS-JCZD-202403）、中国科学院软件研究所重大项目（ISCAS-ZD-202302）以及中国科学院青年创新促进会（Y2022044）的资助。

参考文献

[1] Amazon DMS。（工具）。https://aws.amazon.com/dms/schema-conversion-tool/。访问日期：2025-01。[2] ANTLR。（工具）。https://www.antlr.org/。访问日期：2025-01。[3] ANTLR语法。（语法）。https://github.com/antlr/grammars-v4。访问日期：2025-03。[4] Azure DMS。（工具）。https://learn.microsoft.com/zh-cn/azure/dms/dms-overview。访问日期：2025-03。[5] Datometry。（工具）。https://datometry.com/。访问日期：2025-01。[6] DB-Engines排名。（网站）。https://db-engines.com/en/ranking。访问日期：2025-03。[7] DB2。（数据库管理系统）。https://www.ibm.com/products/db2-database。访问日期：2025-01。[8] DeepSeek-V3。（模型）。https://github.com/deepseek-ai/DeepSeek-V3。访问日期：2025-05。[9] 道琼斯。（网站）。https://aws.amazon.com/cn/solutions/case-studies/dow- jones/。访问日期：2025-01。[10] GPT-4o。（模型）。https://openai.com/index/hello-gpt-4o/。访问日期：2025-01。[11] jOOQ。（工具）。https://www.jooq.org/translate/。访问日期：2025-01。[12] MySQL。（数据库管理系统）。https://dev.mysql.com/doc/refman/8.0/en/。访问日期：2025-01。[13] MySQL。（文档）。https://dev.mysql.com/blog-archive/intersect-and- except-in-mysql-80/。访问日期：2025-05。[14] Oracle。（数据库管理系统）。https://docs.oracle.com/en/database/oracle/oracle-database/23/ sqlrf/。最后访问日期：2025-03。[15] PostgreSQL。（数据库管理系统）。https://www.postgresql.org/docs/15/。访问日期：2025-01。[16] SQLGlot。（工具）。https://github.com/tobymao/sqlglot。访问日期：2025-01。[17] SQLines。（工具）。https://www.sqlines.com/。访问日期：2025-01。[18] TPC-DS基准测试。（TPC）。https://www.tpc.org/tpcds/。访问日期：2025-01。[19] Nicolas Bruno 和 Surajit Chaudhuri。2005。灵活数据库生成器。载于超大型数据库国际会议（*VLDB*）论文集。1097–1107。[20] Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest 和 Clifford Stein。2022。算法导论。麻省理工学院出版社。[21] Ziyu Cui、Wensheng Dou、Yu Gao、Rui Yang、Yingying Zheng、Jiansen Song、Yuan Feng 和 Jun Wei。2025。简单测试即可揭示大多数关键事务漏洞：理解和检测数据库系统中的写操作特定的可串行化违规。载于超大型数据库国际会议（*VLDB*）论文集。2547–2560。[22] Sebastian Farquhar、Jannik Kossen、Lorenz Kuhn 和 Yarin Gal。2024。使用语义熵检测大语言模型中的幻觉。《自然》630，8017（2024），625–630。[23] Jim Gray、Prakash Sundaresan、Susanne Englert、Ken Baclawski 和 Peter J. Weinberger。1994。快速生成十亿条记录的合成数据库。载于*ACM SIGMOD*数据管理国际会议（*SIGMOD*）论文集。243–252。[24] Surabhi Gupta 和 Karthik Ramachandra。2021。SQL过程扩展：理解其在实际应用中的用法。《*VLDB*学报》（*PVLDB*）14，8（2021），1378–1391。[25] Yang He、Pinhan Zhao、Xinyu Wang 和 Yuepeng Wang。2024。VeriEQL：针对带有完整性约束的复杂 SQL 查询进行有界等价性验证。《*ACM*编程语言汇刊》（*PACMPL*）8，OOPSLA1（2024），132:1–132:29。[26] Kenneth Houkjær、Kristian Torp 和 Rico Wind。2006。简单且真实的数据生成。载于超大型数据库国际会议（*VLDB*）论文集。1243–1246。[27] Jia Li、Ge Li、Yongmin Li 和 Zhi Jin. 2025. 结构化思维链 -of-思维提示 - 用于代码生成。*ACM*软件工程与方法学汇刊（*TOSEM*）34, 2（2025），37:1–37:23。[28] Li Lin、Zongyin Hao、Chengpeng Wang、Zhuangda Wang、Rongxin Wu 和 Gang Fan。2024。SQLess：与方言无关的 SQL 查询简化。载于 *ACM SIGSOFT*国际软件测试与分析研讨会（*ISSTA*）论文集。743–754。[29] Ghassan Misherghi 和 Zhendong Su。2006。HDD：分层差分调试。载于 *IEEE/ACM*国际软件工程大会（*ICSE*）论文集。142–151。[30] Amadou Latyr Ngom 和 Tim Kraska。2024。Mallet：利用大语言模型规则生成进行 SQL 方言翻译。载于 利用人工智能技术进行数据管理国际研讨会（*AI4DM*）论文集。5。[31] Shuyin Ouyang、Jie M. Zhang、Mark Harman 和 Meng Wang。2025。ChatGPT 在代码生成中的非确定性实证研究。*ACM*软件工程与方法学汇刊（*TOSEM*）34, 2（2025），1–28。[32] Jiansen Song、Wensheng Dou、Yingying Zheng、Yu Gao、Ziyu Cui、Wei Wang 和 Jun Wei。2025。通过等价数据库构造检测关系型数据库管理系统中的模式相关逻辑错误。载于 超大型数据库国际会议（*VLDB*）论文集。2281–2294。[33] Chengnian Sun、Yuanbo Li、Qirun Zhang、Tianxiao Gu 和 Zhendong Su。2018。Perses：语法引导的程序缩减。载于 *IEEE/ACM*国际软件工程大会（*ICSE*）论文集。361–371。[34] Guancheng Wang、Yiqian Wu、Qihao Zhu、Yingfei Xiong、Xin Zhang 和 Lu Zhang。2023。一种针对抽象语法树的概率差分调试方法。载于 国际软件可靠性工程研讨会（*ISSRE*）论文集。763–773。[35] Justin D. Weisz、Michael Muller、Stephanie Houde、John Richards、Steven I. Ross、Fernando Martinez、Mayank Agarwal 和 Kartik Talamadupula。2021。无需完美？代码翻译中的人机协作。载于 智能用户界面国际会议（*IUI*）论文集。402–412。[36] Zhenyang Xu、Yongqiang Tian、Mengxiao Zhang、Jiarui Zhang、Puzhuo Liu、Yu Jiang 和 Chengnian Sun。2025。T-Rec：由词法语法引导的细粒度语言无关程序缩减。*ACM*软件工程与方法学汇刊（*TOSEM*）34, 2（2025），31。[37] Rui Yang、Ziyu Cui、Wensheng Dou、Yu Gao、Jiansen Song、Xudong Xie 和 Jun Wei。2025。通过等价数据库构造检测关系型数据库管理系统中的模式相关逻辑错误。载于*ACM SIGSOFT*国际软件测试与分析研讨会（*ISSTA*）论文集。1725–1747。[38] Wenhua Yu、Zhenhao Chen、Haibo Liu 和 Yongbin Bai。2024。SQL 方言本地化翻译研究。载于 信息技术、网络、电子与自动化控制会议（*ITNEC*）论文集。1322–1326。[39] Mengxiao Zhang、Yongqiang Tian、Zhenyang Xu、Yiwen Dong、Shin Hwei Tan 和 Chengnian Sun。2024。LPR：大语言模型辅助的程序缩减。载于 *ACM SIGSOFT*国际软件测试与分析研讨会（*ISSTA*）论文集。261–273。[40] Mengxiao Zhang、Zhenyang Xu、Yongqiang Tian、Yu Jiang 和 Chengnian Sun。2023。PPR：成对程序缩减。载于 *ACM*联合欧洲软件工程会议与软件工程基础研讨会（*ESEC/FSE*）论文集。338–349。[41] Wei Zhou、Yuyang Gao、Xuanhe Zhou 和 Guoliang Li。2025。突破 SQL 障碍：基于大语言模型的方言翻译系统。数据管理国际会议（*SIGMOD*）论文集3, 3（2025），26。[42] Ran Zmigrod、Salwa Alamir 和 Xiaomo Liu。2024。为云迁移进行 SQL 方言之间的转换。载于 *IEEE/ACM*国际软件工程大会：软件工程实践（*ICSE SEIP*）论文集。189–191。