SDE-SQL: 通过使用 SQL 探针进行自驱动探索, 增强大型语言模型中的文本到 SQL 生成

Wenxuan Xie[1], Yaxun Dai[2], Wenhao Jiang[3]*

[1]South China University of Technology [2]Soochow University [3]Guangdong Laboratory of AI and Digital Economy (SZ)

lancelotxie601@gmail.com, cswhjiang@gmail.com

Abstract

大型语言模型（LLMs）的最新进展显著提升了文本到 SQL 任务的性能。然而，以往的方法通常依赖于推理时提供的静态、预处理的数据库信息，这限制了模型对数据库内容的全面理解。没有动态交互，LLMs 被限制在固定的人为提供的上下文中，无法自主探索底层数据。为了解决这一限制，我们提出了 SDE-SQL，一个可以让大型语言模型在推理期间对数据库进行自我驱动探索的框架。这是通过生成和执行 SQL 探针来实现的，使模型能够主动从数据库中检索信息并迭代更新其对数据的理解。与以往方法不同，SDE-SQL 在零样本的情况下操作，不依赖于任何问题和 SQL 对作为上下文示范。在以 `Qwen2.5-72B-Instruct` 评估的 BIRD 基准测试中，SDE-SQL 在执行准确度上相比于基础的 vanilla `Qwen2.5-72B-Instruct` 基线取得了 8.02 % 的相对提升，在不依赖监督微调（SFT）或模型集成的开源模型方法中成就了新的最先进水平。此外，通过 SFT，SDE-SQL 的性能可以进一步提高，带来额外的 0.52 % 的提升。

1 引言

文本到 SQL 是自然语言处理领域中的一个长期研究任务，旨在将自然语言问题翻译成可以由数据库执行的 SQL 查询。它不仅能让非专业人员轻松地与数据库交互，还通过提供数据库中存储的知识，帮助缓解问答系统中的幻觉问题。

最近在大型语言模型（LLM）方面的发展大大提高了文本到 SQL 应用程序的性能和准确性。基于大型语言模型的方法已经在早期的 Spider 数据集 (**?**) 上实现了超过 90 % 的执行准确性，并且在更复杂和具有挑战性的 BIRD 数据集上也表现良好。然而，这些方法与人类专家之间仍然存在差距，这在最近提出的 Spider 2.0 基准 (**?**) 中尤为明显。目前主流的大模型基础的文本到 SQL 方法通常由三个关键模块组成：模式链接、SQL 生成和 SQL 优化。在模式链接阶段，以往的工作主要集中于将更精确和相关的数据库信息匹配到特定的自然语言问题。在 SQL 生成阶段，各种问题分解方法和推理策略被提出。在 SQL 优化阶段，错误类型被更彻底地分类，从而开发出更有针对性的纠正技术。然而，SQL 的一个重要但常常被忽略的方面是其作为与数据库交互界面的角色，能够实现快速而高效的执行。这个特性在当前的方法中仍未被充分利用，可能部分解释了基于大型语言模型的系统与人类专家之间的性能差距。

因此，我们提出了 SDE-SQL，在生成和改进阶段引入了自驱动探索技术，如图 **??** 所示。除了与自然语言问题相对应的最终 SQL 查询之外，在任务过程中还生成和执行了一系列专为数据库探索设计的 SQL 查询。这些 SQL 查询被称为 SQL 探针。对于模式链接，我们采用基于实体的值检索和基于实体的软链接方法。在生成阶段，我们允许大语言模型根据当前的自然语言问题和数据库模式独立探索数据库，然后使大语言模型能够利用探索结果进行零样本学习。

在生成之后，我们使用两种策略来改进不正确的 SQL 查询。对于那些返回显式错误信息的查询，LLMs 会根据反馈直接修改查询。对于返回空结果的 SQL，我们应用基于规则的分解来生成一系列子查询（Sub-SQLs）。然后，LLMs 分析它们的执行结果并进行有针对性的探索以识别和修复潜在问题。

在 BIRD 数据集上的评估中，基于 Qwen2.5-72B-Instruct 的 SDE-SQL 实现了 67.67 % 的执行准确率。在 SFT 之后，性能进一步提高到 68.19 %。

总之，我们的贡献如下：

- 我们提出了 SDE-SQL，这是一个新颖的框架，利用自驱探索来释放大型语言模型在文本到 SQL 任务中的真正潜力，使它们能够使用 SQL 作为探针来探索数据库，从而缩小大型语言模型与人类专家之间的

*Corresponding author

性能差距。

- 我们在 SQL 生成阶段引入了两阶段探索，使得 LLMs 具有动态与数据库交互的能力，从而突破了信息可用性的限制。
- 在 SQL 优化阶段，我们使用基于规则的方法将 SQL 分解为一系列子 SQL，使得 LLM 可以分析这些子 SQL 的执行结果，通过生成的 SQL 探针的执行结果诊断错误来源，并进一步与数据库交互以根据诊断结论探索解决方案。
- 我们在 BIRD 数据集和 Spider 数据集上评估了我们的框架，并进行了一系列消融实验，证明了自驱动探索对文本到 SQL 任务的重要性和有效性。
- 我们还分别为探索和生成任务构建了一小部分数据，用于监督微调（SFT）。结果表明，单独微调模块可以改善 SDE-SQL 的整体性能。

2 相关工作

将自然语言问题转换为数据库查询是一项经典任务，最早的工作使用归纳逻辑编程和人工设计的模板来完成这一任务 (**?**) 。近年来，文本到 SQL 技术的进步大致可以分为两个阶段，这由自然语言处理的进展所推动。

2.1 传统的 Seq2Seq 模型方法

先前的工作主要集中于改进编码或解码方法，因为 seq2seq 模型框架由两个主要组成部分组成：编码器和解码器。IRNet 采用双向 LSTM 来编码问题，并使用自注意力机制来编码数据库模式，最终使用基于语法的解码器 LSTM (**?**) 。为了有效捕捉数据库模式和问题之间的关系，RAT-SQL 开发了一种具有关联感知自注意力机制的编码器 (**?**) 。此后，**?** 和 **?** 利用图神经网络来编码模式和查询之间的关系。利用预训练语言模型（PLM）在各种 NLP 任务中的出色能力，**?** 是第一个将 BERT 作为编码器的模型。对于解码器的改进，**?** 和 **?** 关注基于草图的解码方法。为了减少推理过程中的时间消耗，SDSQL 提出了模式依赖性学习，并去除了执行指导（EG）解码策略 (**?**) 。

2.2 基于 LLM 的方法

随着 LLMs 的到来，文本到 SQL 领域经历了一次突破性的创新，带来了任务处理方法的显著变化。

基于提示工程的方法 **?** 评估了 LLMs 在文本到 SQL 任务中的潜力，展示了 LLMs 在该任务中的非凡能力。基于上下文学习，DAIL-SQL (**?**) 通过问题表示、示例选择和示例组织，引入了一种新的提示工程方法，改善了 LLMs 在文本到 SQL 任务中的性能。基于链式思维（CoT）推理风格 (**?**) ，DIN-SQL (**?**) ，Divide-and-Prompt (**?**) ，CoE-SQL (**?**) 和 SQLfuse (**?**) 设计了包含推理步骤的 CoT 模板，以激发链式思考。为了提高 LLMs 处理复杂问题的能力，QDecomp (**?**) ，DIN-SQL (**?**) ，MAC-SQL (**?**) 和 MAG-SQL (**?**) 分解复杂的自然语言问题并逐步解决。此外，MCS-SQL (**?**) ，CHASE-SQL (**?**) 和 CHESS (**?**) 通过在推理阶段生成大量候选 SQL 查询并选择最合适的查询来提高性能。

尽管基于闭源模型（如 GPT-4o）的方法在 Text-to-SQL 任务中表现良好，但它们面临着高成本、无法保证隐私和灵活性有限的问题。因此，在 Text-to-SQL 任务中微调开源模型具有重要的实际价值和应用潜力。DTS-SQL 和 SQLfuse 探索了微调 LLMs 用于模式链接和 SQL 生成。SQL-PaLM、Open-SQL、XiYan-SQL 和 CodeS 在精心选择的数据上微调了开源 LLMs，而 CodeS 特别采用了一种增量预训练方法，使用特别策划的以 SQL 为中心的语料库。此外，还有一些新颖的视角。DELLM 特别微调了一个提供领域知识的数据专家语言模型，而 SQL-GEN 则提出了一种新颖的专家混合（MoE）架构以处理多种 SQL 方言。

在 Text-to-SQL 任务中，模式链接是指根据输入的自然语言问题，识别和选择数据库中相关表、列和值的过程。为了提高链接的准确性，我们使用了一种基于实体的链接方法，包括值检索和软模式链接。

类似于 **?** 中的检索模块，我们首先利用一个 LLM 通过少样本学习从自然语言问题中提取实体。然后，值检索器基于局部敏感哈希（LSH）和语义相似性在数据库中识别相似值。

为了提高模式链接阶段的容错性，我们选择了软模式链接方法，就像 **?** 中的方法那样。我们采用一种一挥而就的方式，提示 LLM 根据每个实体选择相关的列。对于选定的列，我们在后续的 SQL 生成过程中提供尽可能详细的信息，包括列名、类型、列描述、值示例和值描述。对于未选定的列，我们仅保留列名和类型。这种方法不仅显著减少了输入长度，使语言模型在生成过程中能够专注于最相关的数据库模式，而且通过防止删除未被选中的有用列来增强容错性。

在先前的文本到 SQL 研究中，SQL 通常被视为主要的中间结果或最终输出，其固有的功能大多被忽视。因此，我们引入了 SQL Probes

的概念。字面上，SQL Probes 意指作为探测仪的 SQL 查询，专门用于根据当前自然语言问题探索数据库。我们正式将此任务定义为从自然语言查询 Q 和数据库模式 D 映射到对应的 SQL 查询 S 。自然语言查询 Q 由两个部分组成：目标和条件 (?) 。通常情况下，目标对应于 SQL 查询 S 中的主要 SELECT 子句，而条件对应于 S 中的其他子句，例如 WHERE 子句。图 1 是一个例子。

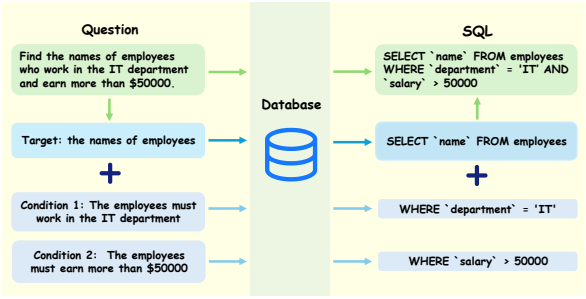

Figure 1: 文本到 SQL 的一个例子。

为了获得具体的 SQL 表示，实体必须首先映射到数据库中的相应列和值。这个步骤是否能够准确执行取决于语言模型对数据库的理解程度。

然而，以前处理的数据库模式提供的信息远远不够。现实世界中的数据库通常非常复杂且混乱。不同的表格可能包含许多具有相同含义（表示同一项目）的列，这些列中的值可能具有不同的格式，某些值甚至只存在于特定表中。在信息不足的情况下，LLM 只能随机识别这些相似列和值中的组合。这也是 LLM 在该任务中表现尤其不稳定的关键原因之一。在评估过程中，经常观察到模型有时可以正确预测某些例子，而在其他时候却在相同例子上失败。在先前的工作中，某些方法涉及使用语言模型生成多个 SQL 候选，然后选择最合适的一个。然而，这种方法未能解决根本问题。我们提出，最根本的解决方案是赋予 LLM 动态与数据库交互的能力。在 SDE-SQL 中，LLM 在生成之前，在数据库中执行两个阶段的自我驱动探索。

2.2.1 候选探索

探索阶段的目标是使大型语言模型能够查询数据库，以获取有关目标和单个条件的信息，然后为每个目标和条件选择合适的候选项。由于自然语言问题中的实体被映射到数据库中的列或值（或者同时是列和值），LLM 需要确定每个实体的候选列和候选值。最初，语言模型生成几个基础 SQL 探针，它们列举了目标的候选列。这些 SQL 仅专注于查询目标而不附加任何条件。接下来，生成条件 SQL 探针，其中每个探针通常通过添加一个列候选项和可能的一个与特定条件对应的值候选项来扩展基础 SQL 探针。假设每组候选项包含两个选项，条件 SQL 探针的生成如图 2 所示，其中每条从根到叶的路径对应于一个特定的条件 SQL 探针。我们将每个条件 SQL 探针的条件描述称为条件描述候选项。例如，如图 2 ，一种条件描述候选项是：

```
SELECT Phone FROM users WHERE name =
'John␣Charlie␣Hinton'AND location = '
United␣Kingdom';
```

基于前一阶段探索的结果，候选者的范围已经缩小。现在，有必要结合所有条件以找到最合适的候选者组合。对于返回无结果的 SQL 查询，相应的候选者组合肯定是不合适的。

2.2.2 零样本生成与探索结果

在我们的实验中，我们发现现有方法没有充分发挥大型语言模型在 SQL 生成中的潜力。例如，设计新的分解方法以使模型逐步解决复杂问题、使用各种提示技术生成多个候选以供选择、或者采用搜索策略如蒙特卡罗树搜索（MCTS）来增强语言模型的推理能力等策略，可以带来模型性能的适度提升。然而，这些提升仍然显著小于通过为模型提供足够信息所获得的提升。

因此，在 SDE-SQL 中，LLM 生成器根据数据库模式和前两个探索阶段的结果生成 SQL，而不依赖任何问题-SQL 对作为 few-shot 示例或使用任何问题分解策略。为了提高 SQL 生成的准确性和鲁棒性，我们采用了一种自一致性策略，通过比较多个生成的 SQL 查询的执行结果，选择最一致的答案。

在过去，基于上下文学习的现有技术已经引入了检测和修复文本到 SQL 错误的解决方案，每种解决方案在错误识别算法和提供的辅助数据上有所不同，以帮助 LLM 理解和纠正这些错误。

对于语法错误和模式错误，执行后的错误反馈已经包含了足够的信息，使得 LLMs 能够有效地完成 SQL 的修正。然而，对于一些其他更复杂的错误，它们通常会导致查询结果为空，且没有错误信息提示。即便是人类尝试修正这些错误时，也无法一次完成；相反，他们需要编写一些 SQL 语句用于调试，并根据这些查询的执行结果来诊断问题。当前的方法涉及不断地重新生成，直到修复成功或达到尝试次数限制。在整个修复过程中，LLM 并没有接收到任何有用的信息，其推理能力也没有得到充分利用。换句话说，错误的原因从未被识别。

因此，在 SDE-SQL 中，我们在 SQL 修订之前引入了一个全面的自驱动探索阶段。对于没

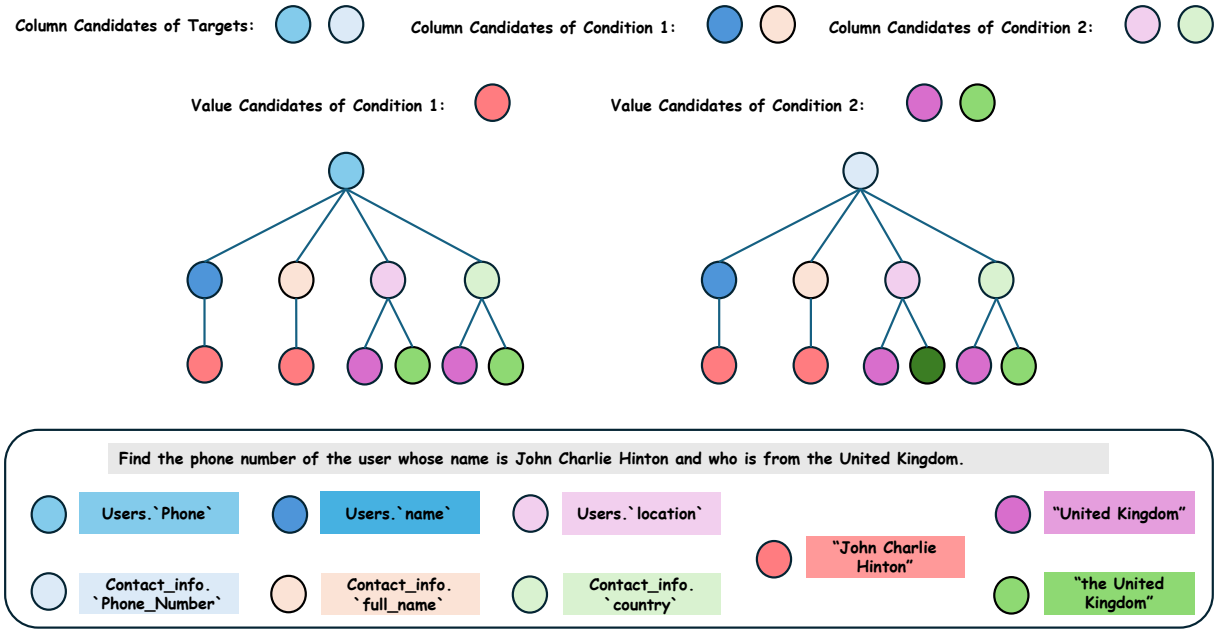

Figure 2: 条件 SQL 探针生成过程使用树结构进行说明。

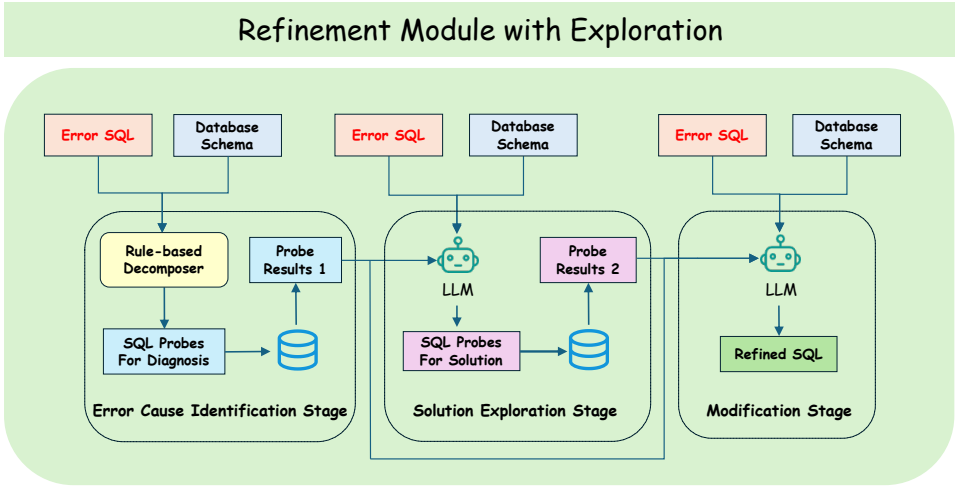

Figure 3: 在 SDE-SQL 中带有探索的提炼过程示意图。

有结果的查询，改进过程分为三个不同的阶段：错误原因识别阶段、解决方案探索阶段和修改阶段，如图 3 所示。

在一个复杂的 SQL 语句中，可能涉及多张表并同时应用多个条件，从而使得通过直接分析整个 SQL 来定位问题变得困难。因此，我们需要进行细粒度的诊断。为了生成一系列用于诊断目的的子 SQL 作为 SQL 探针，我们开发了一个基于 SQLGlot 的分解器。该分解器首先将复杂的 SQL 查询转换为抽象语法树（AST），然后通过分析节点类型及其关系来识别不可分割的条件单元。AST 内识别出的这些子树成为生成语义有效的子 SQL 的基础，这些子 SQL 的执行结果将提供给 LLM，以协助其准确诊断和定位原始查询中的问题。分解结果的一个示例如图 **??** 所示。

2.2.3 解决方案探索阶段

我们总结了五个可能导致查询结果为空的原因。在这个阶段，LLM 需要分析前一阶段探针的结果，以得出可能的错误原因假设，然后生成一系列 SQL 探针来协助探索这些错误的潜在解决方案。

条件冲突或条件重复 该错误指的是在单个条件下执行时可以找到数据，但在多个条件组合时，却找不到满足要求的数据（导致查询结果为空）。造成此错误的可能原因有两个：多个不同条件的组合之间存在冲突或者通过不同列冗余地描述了单个条件。(i) 条件冲突通常发生在一个条件中的实体对应于多个可能的候选列，并且只有一个特定候选列与其他条件组合才能获得相应的数据项。在图 **??** 中展示了一个示例。(ii) 条件重复发生在一个条件中的实

体映射到多个候选列，导致大型语言模型生成的 SQL 无意中使用了这些不同的候选列来描述相同的条件，最终导致不能检索到满足预期条件的数据。

不必要的表连接 SQL 可能包括不必要的表连接，导致在最终交集中没有记录满足条件。

列与值不匹配 当值的格式不匹配被选择的（但正确的）列，或者选择了一个类似的列但不包含预期值时，会出现此错误。

子查询范围不一致 有时，子查询的范围可能与主查询的不一致，尤其是在子查询中使用 `MIN/MAX` 函数时，这常常导致空查询结果。图 **??** 显示了一个示例，在子查询中检索到的数据行在这两个表的 JOIN 之后的结果中并不存在。

对于一个 SQL 查询，最重要的部分实际上是查询的目标，它指的是在 `SELECT` 子句中被选择的列。如果 SQL 中的查询目标与自然语言问题中的原始查询目标不一致，那么这种转换无疑是失败的。然而，当大型语言模型生成 SQL 时，它们有时会在 `SELECT` 子句中包含不属于查询目标的列，例如用于条件的列。因此，在基于执行结果优化 SQL 后，有必要检查 SQL 中的查询目标是否与自然语言问题中的查询目标匹配。为避免在此阶段引入新的错误，我们允许大型语言模型仅确定 SQL 中是否选择了不必要的目标列。如果发现了这样的列，它们将被移除而不影响执行。该过程在图 **??** 中有所描述。

2.3 监督微调（SFT）

为了进一步增强模型自主探索数据库的能力，并利用探索结果生成更准确的 SQL，我们还对模型进行了监督微调（SFT）。训练数据是从 BIRD 的训练集中抽取的。我们使用了一种基于提示的流程来展开数据，并将最终生成正确 SQL 的示例视为有效数据，用于对模型进行微调。

在 9,428 个数据点中，5,231 个有效样本是通过采样获得的。对于每个示例的推理轨迹，我们提取了两个组件：（ i ）探索阶段，在该阶段生成 SQL 探测；以及（ ii ）预测阶段，在该阶段基于探索结果生成最终的 SQL 查询。

在本节中，我们首先介绍实验设置，然后报告和分析结果。Spider 是一个广泛采用的用于 Text-to-SQL 任务的基准数据集。它是大规模、跨领域和复杂的，包含了 10,181 个自然语言问题和 5,693 个 SQL 查询，涉及 200 个不同的数据库。作为 Text-to-SQL 任务的一个具有挑战性的基准，最近提出的 BIRD 数据集包括了 95 个具有脏值的大规模实际数据库，具有 12,751 对独特的问题- SQL 对。BIRD 数据集中的数据库与现实场景中的类似，表现出了固有的模糊性。因此，每个列都提供了详细的描述，以及外部知识。在这项工作中，我们选择执行准确性（EX）作为指标，因为它反映了执行的 SQL 查询返回结果的准确性。该指标考虑了同一问题的各种 SQL 表述，提供了更精确和公平的结果评估。对于探索任务和生成任务，我们使用 Qwen2.5-72B-Instruct 在 8 块 NVIDIA A800 GPU 上进行了 24 小时的训练。详细的训练超参数在表中提供。为了进行全面比较，我们选择了基于闭源模型的代表性方法和基于开源模型的代表性方法（不采用模型集成）作为基线。

在 BIRD 开发数据集上评估时，基于 Qwen2.5-72B-Instruct 的 SDE-SQL 超过了大多数基于 GPT-4 的方法和大多数开源模型，经过微调后实现了 68.19% 的执行准确率，如表 **??** 所示。即使在无训练设置下，它也能实现 67.67% 的强大表现，进一步突出了我们方法的有效性。

如表 **??** 所示，仅在 BIRD 训练集上微调的 SDE-SQL 在 Spider 基准测试中取得了有竞争力的结果，超过了基于 GPT-4 的方法和大多数开源模型，这凸显了其强大的泛化能力。然而，性能提升相对有限，因为 Spider 数据集中的大量 SQL 查询产生空执行结果，从而限制了数据库探索反馈的有效性。

对于 SDE-SQL 的每个模块，我们在 BIRD 基准测试的开发集上进行了消融研究，如表 **??** 所示。此外，我们评估了将微调的探测器和生成器整合到管道中的效果。结果表明，每个组件都起着重要作用，引入两个探索阶段尤其引领了显著的性能提升。此外，在各自子任务上微调的模块可以进一步增强工作流程的整体性能。

在这项工作中，我们提出了 SDE-SQL，这是一种将自驱动探索集成到 SQL 生成和优化阶段的新颖文本到 SQL 框架。通过使 LLMs 能够通过 SQL 探测主动与数据库交互，SDE-SQL 弥合了静态查询生成与动态、基于执行的推理之间的差距。该探索机制使 LLMs 能够发现潜在的模式语义和执行模式，显著提高了生成可执行且语义准确的 SQL 查询的能力。在 BIRD 和 Spider 数据集上的大量实验表明了 SDE-SQL 的有效性，经过监督微调后，该模型在 BIRD 上的执行准确率达到了 68.19。消融研究确认了关键组件的贡献——尤其是探索管道和微调策略。未来工作中，我们计划探索将探索信号更紧密地集成到模型训练中，以进一步增强模型的推理能力。

尽管自驱动探索显著增强了大型语言模型在 Text-to-SQL 任务中的潜力，但我们当前的方法

仍存在若干限制。在 SDE-SQL 中，数据库探索完全依赖于提示驱动——这意味着探索过程的有效性在很大程度上取决于手工设计的提示的设计和质量。构建不当的提示可能导致模型生成无信息或冗余的 SQL 探测，从而限制其获取有意义的模式知识或执行见解的能力。此外，仅依赖提示工程可能会限制模型进行更深入推理的能力，因为它缺乏基于环境反馈进行自适应学习的机制。

另一个限制是模型无法自主地随着时间推移改进其探索策略。由于每个 SQL 探测都是从提示静态生成的，模型无法根据探索过程中的先前成功或失败动态调整其行为。这个限制减少了系统的整体灵活性和学习效率。

为了解决这些问题，未来的工作将重点放在使数据库探索更多地成为模型本身的内在部分。一个有前途的方向是结合强化学习或其他反馈驱动的学习范式，使模型能够根据执行结果迭代地完善其探测策略。通过使模型从其与数据库的交互中学习，我们希望开发出一个更为强大和自适应的框架，能够在复杂的数据库环境中进行更深入的、上下文感知的推理。

对于不依赖监督训练的方法，精心设计有效的提示模板以在控制范围内引导模型进行自驱探索行为显得尤为重要。在本节中，我们展示了一套全面的提示模板，这些模板在 SDE-SQL 流程的不同阶段被利用以支持此功能。由于空间限制，某些详细元素和具体提示示例已被省略，但基本结构和核心思想已被完整保留。

```
[Instruction]
Your task is to generate a series of SQL Probes to explore the database and
identify the correct columns mentioned the given question. These Probes will
help determine which columns contain the necessary data and ensure that the
final SQL query returns non-empty results. Follow these requirements:

[Requirements]
- In this task, you should identify and list all entities mentioned in the
question, along with their corresponding candidate columns in the database
schema. For each entity, there is only one candidate column unless the
database schema contains multiple columns with the same or extremely similar
meanings that are consistent with the entity. Do not include unnecessary
columns as candidate columns. For each entity, if it corresponds to multiple
candidate columns, generate SQL Probes to check the presence of relevant
values in each candidate column. If a specific value is mentioned for an
entity (e.g., 'Mountain View' district or enrollment > 500), include SQL
Probes to verify the existence of that value in the candidate columns.
- The entities in the question are divided into two types: target entity and
condition entity. The target entity is the ultimate goal of the query, while
the condition entity corresponds to the conditions that the target entity
needs to satisfy. First, you need to generate the corresponding Base SQL
Probes based on the target entity. Then, for each condition entity, generate
the corresponding Condition SQL Probes based on the Base SQL Probes.
Base SQL Probes: At first generate the base SQL Probes that search for the
target entity. All other SQL Probes should be generated based on this base SQL
Probe.
Condition SQL Probes: Generate SQL Probes for each condition entity based on
the Base SQL Probe.

[Attention]
- If the 【Evidence】 specifies a candidate column or candidate value for an
entity, use that column or value as the mapping for the entity directly if
【Evidence】 is reasonable, and there is no need to explore other candidates.
If there is a calculation formula for an entity in the 【Evidence】,
prioritize using this formula to represent the entity. This is very
important!!!
- You don't need to consider SQL Probes that combine multiple conditions.
- Base SQL Probes should only select the targets directly without other
conditions.
- Condition SQL Probes will add new conditions to the Base SQL Probe.

[Note]
...
[SQL Tricks]
...
[Database admin instructions]
...
[Output Format]
...
```

Figure 4: 候选探索的提示模板

```
[Instruction]
The question provided to you can be broken down into a target and several
conditions. Previously, a series of SQL Probes based on the target and
conditions were generated. Among these, the Base SQL Probes are generated for
the target, while the other SQL Probes are based on the Base SQL Probes with
the addition of exploring a specific condition. I will provide you with these
SQL Probes and their corresponding execution results (whether they return
empty or not). What you need to do is combine the conditions based on the
Database schema and the question to generate a new series of SQL Probes. This
will help conduct a more in-depth exploration of the database and assist me in
generating the final SQL for the question.

[Requirements]
- The execution results can be one of two outcomes: NULL or Not NULL. !!!NULL
means that the result of the SQL query is empty (no data matches the
conditions). Not NULL means that the result of the SQL query is not empty
(there is data that matches the conditions)!!!
- You need to analyze the current execution results, eliminate the obviously
invalid candidate columns, and only combine the ones that are potentially
valid.
- You need to combine all the conditions to ensure a comprehensive exploration.
For example, suppose the current question contains a target and three
conditions. After analyzing the execution results, the candidate columns are
as follows: the unique candidate column for the target can be determined from
the Base SQL Probes, the first condition has two possible candidate columns,
the second condition has one possible candidate column, and the third
condition has three possible candidate columns. Therefore, the number of SQL
Probes to be generated after combining them would be 1 * 2 * 1 * 3 = 6.

[Tips]
...

[Output Format]
...
```

Figure 5: 组合探索的提示模板

```
# Task Description
You are an SQLite database expert tasked with generating a SQL query according
to a input user question. You will be provided:
- An input user question, and potentially an evidence
- The database schema
- The descriptions of columns(column name, data_format, description)
- The value retrieved from database
- The SQL Probe result

Your task is to generate the correct SQL query. The input question consists of
a query target and the conditions that the target needs to satisfy. You need
to analyze the semantics of the question and convert it into the corresponding
SQL. You should imitate human, and solve this task step by step.

# Note
...

# SQL Tricks
...

# Database admin instructions
...

# Output Format
...
```

Figure 6: 零样本生成的提示模板

```
[Instruction]
When executing an SQL statement, there may be instances where the execution
result is completely empty.
You need to identify the cause of the error based on the query and database
information and generate some new probe SQLs to find solution.
To help you to find out the reason, I extracted a batch of probe SQLs from
this incorrect SQL and executed them, providing you with the execution results
(NULL or Not NULL). !!!NULL means that the result of the SQL query is empty
(no data matches the conditions). Not NULL means that the result of the SQL
query is not empty (there is data that matches the conditions)!!!
You can use these Probe SQL query results to determine where the issue lies
based on whether they are empty or not.
The revised SQL must be consistent with the Query, and it should not omit any
necessary conditions described in the Query.

[Possible Causes]
Cause 1: Conflicting Conditions or Redundant Descriptions Across Different
Columns
--details: Conflicting: The simultaneous existence of two conditions leads to
null, proving that the column for one of the conditions was chosen incorrectly.
Redundant: For a certain condition, multiple different columns are used to
repeat the description, resulting in a conflict between this condition and
other conditions.
--fix: For the conflicting case, certainly, it seems that a condition might be
described by several columns with similar meanings, but the incorrect column
was selected in the SQL. To resolve this, identify and replace the column name
accordingly. For the Redundant case, remove duplicate descriptions of the same
condition and keep the one that fits best.

Cause 2: Incorrect Condition Values or Case Sensitivity Issues
--details: The conditions in the query may use incorrect values or fail to
account for case sensitivity when comparing strings.
--fix: Try to use `LIKE` because the LIKE keyword is case-insensitive by
default.(table.<column> = 'xxx' -> table.<column> LIKE 'xxx')

Cause 3: Unnecessary Table Joins Resulting in No Satisfying Records.
--details: The query may include unnecessary table joins, resulting in no
records satisfying the conditions in the final intersection.
--fix: Check if every table join is really necessary and discard unnecessary
tables.

Cause 4: Incorrect Column Selection
--details: Among the several tables involved, there may be multiple candidate
columns for a certain condition, but Old SQL selected the wrong one.
--fix: Determine if there is a more suitable column, or use a similar column
from another table.

Cause 5: Misuse of the MAX(MIN) function or `ORDER BY`
--details: Using the MAX(MIN) function or `ORDER BY` in a subquery(nested sql),
the data corresponding to this maximum or minimum value may not be in the
intersection of the two tables, so it may return a null value.
--fix: First JOIN the tables, and then use MAX(MIN) function or `ORDER BY` on
the JOIN results.

[Requirements]
After thinking step by step, you may already have some guesses and potential
solutions about the cause of the error, but you need to validate these guesses
and solutions. Please generate a set of probe SQLs based on your analysis to
help your future self arrive at the correct SQL.
```

Figure 7: 解决方案探索的提示模板

```
[Instruction]
When executing an SQL statement, there may be instances where the execution
result is completely empty.
You need to identify the cause of the error based on the query and database
information and make the necessary corrections.
To help you to find out the reason, I executed a series of SQLs which is
related to this question, providing you with the execution results (NULL or
Not NULL). !!!NULL means that the result of the SQL query is empty (no data
matches the conditions). Not NULL means that the result of the SQL query is
not empty (there is data that matches the conditions)!!!
You can use these Probe SQL query results to determine where the issue lies
based on whether they are empty or not.
Note that your modified SQL still has to correspond one-to-one with the
targets and conditions in the Query.

[Possible Causes]
Cause 1: Conflicting Conditions or Redundant Descriptions Across Different
Columns
-- details: Conflicting: The simultaneous existence of two conditions leads to
null, proving that the column for one of the conditions was chosen incorrectly.
Redundant: For a certain condition, multiple different columns are used to
repeat the description, resulting in a conflict between this condition and
other conditions.
-- fix: For the conflicting case, certainly, it seems that a condition might
be described by several columns with similar meanings, but the incorrect
column was selected in the SQL. To resolve this, identify and replace the
column name accordingly.
For the Redundant case, remove duplicate descriptions of the same condition
and keep the one that fits best.

Cause 2: Incorrect Condition Values or Case Sensitivity Issues
-- details: The conditions in the query may use incorrect values or fail to
account for case sensitivity when comparing strings.
-- fix: Try to use `LIKE` because the LIKE keyword is case-insensitive by
default.(table.<column> = 'xxx' -> table.<column> LIKE 'xxx')

Cause 3: Unnecessary Table Joins Resulting in No Satisfying Records.
-- details: The query may include unnecessary table joins, resulting in no
records satisfying the conditions in the final intersection.
-- fix: Check if every table join is really necessary and discard unnecessary
tables.

Cause 4: Incorrect Column Selection, No Matching Values
-- details: The query may select the wrong column or the column may not have
any values that satisfy the condition.
-- fix: Determine if there is a more suitable column, or use a similar column
from another table.

Cause 5: Misuse of the MAX(MIN) function or `ORDER BY`
-- details: Using the MAX(MIN) function or `ORDER BY` in a subquery(nested
sql), the data corresponding to this maximum or minimum value may not be in
the intersection of the two tables, so it may return a null value.
-- fix: First JOIN the tables, and then use MAX(MIN) function ORDER BY on the
JOIN result, not in a subquery(nested query).
```

Figure 8: 最终改进的提示模板

```
[Instruction]
You are a helpful assistant. Given a question, a SQL statement and probably a
corresponding evidence, you need to determine whether the query goal of this
SQL and the question are consistent.

[Requirement]
1. First, you need to identify the actual entity (target column) that the
question is trying to query.
2. Determine how many columns the question expects to see in the result.
3. Compare the number of columns returned by the SQL query with the expected
number of columns to determine if extra columns were selected.
4. If extra columns were selected, you need to modify the target after the
SELECT keyword in the original SQL statement to remove the unnecessary target
column. If you believe the selected columns are correct or insufficient, no
modification is needed.
5. Your output should be in JSON format:
```json
{{
    "Modification":"<True or False>",
    "Final SQL":"<sql>"
}}
```

[Example]
...

[Attention]
Only modify when you are absolutely certain that there are extra target
columns. If you feel there is no issue or are unsure whether there is an issue,
do not make any changes.
```

Figure 9: 目标检查的提示模板