目录

第一章　需求分析

1. 1. 需求背景

在当今数据驱动的商业环境中，结构化查询语言（SQL）是访问和管理关系数据库的关键工具。然而，随着数据量的不断增长和数据库复杂性的增加，编写和维护 SQL 代码变得越来越具有挑战性，这限制了数据的可访问性和分析效率，降低了专业人士的工作效率，也提升非专业人士的使用门槛。为了解决这一问题，Text-to-SQL 系统应运而生，旨在通过自然语言处理技术将用户的自然语言查询转换为相应的 SQL 语句。Text-to-SQL 系统能够自动理解和解析这些复杂的数据库结构，大大降低了逻辑梳理的时间成本。通过自然语言输入，系统可以快速生成准确的 SQL 查询，即使面对超大规模的数据库表也能高效工作。

Text-to-SQL 任务本身充满挑战。首先，由于 SQL 语句的结构严谨性和复杂性，系统必须能够生成遵循严格语法规则的 SQL 语句。其次，自然语言的多样性和歧义性要求系统必须具备精准的语义理解能力，以确保正确捕捉用户的查询意图。此外，系统还必须理解数据库的结构和内容，这牵涉到复杂的实体识别、关系映射以及逻辑推理等问题。

早期的 Text-to-SQL 研究方法多借鉴机器翻译技术，包括基于规则的方法基于机器学习的方法，以及近年来兴起的基于深度学习的方法。这些方法虽然在特定的数据集和场景下取得了一定的成功，但是在处理复杂查询、多变的自然语言表达以及大规模数据库表的情况下，往往表现不佳。尤其是对于需要大量标注数据的有监督学习方法，它们的效果高度依赖于标签数据的质量和覆盖范围，而这在实际应用中往往难以满足。

大模型具备的自然语言生成计算机语言能力能够解决 Text-to-SQL 方向上的诸多痛点，给该领域带来了显著的进步。大模型在经过大规模的 Text-to-SQL 数据集的预训练，能够更好地理解复杂的自然语言查询，并将其转化为结构化的 SQL 语句。这不仅提高了从自然语言到 SQL 转换的准确性，还使得系统能够处理更加复杂的查询逻辑。自注意力机制等技术使得大语言模型可以捕捉句子中的长距离依赖关系和深层语义信息。这意味着它们能更好地理解用户的意图，即使在模糊或不明确的情况下也能生成合理的 SQL 语句。同时，基于大模型的 Text-to-SQL 解决方案能够实现更高程度的自动化，减少人工干预的需求，并且大语言模型通常具有较强的泛化能力，能够在不同的数据库模式下工作。

因此，一套基于大语言模型的 Text-to-SQL 系统能使当前大数据时代的数据

加工与分析工作更加高效。大语言模型通过自然语言处理技术将用户的自然语言查询转换为准确的 SQL 语句，不仅显著提高了专业人士（如数据分析师和数据库管理员）的查询编写效率和支持复杂逻辑的能力，提升整体的数据访问和分析效率，还降低了非专业人士（如业务人员和技术支持人员）使用数据库的技术门槛，使他们能够轻松进行数据分析和决策支持。

1.2. 用户需求与功能性需求

项目将用户人群分为专业人士与非专业人士。专业人士是指工作中主要的语言就是 SQL 语言或者对数据库系统有较深入理解的人群，比如数据开发工程师、数据分析师和技术开发人员等。非专业人士是指工作中几乎用不到 SQL 或者对数据库系统完全不了解的人，比如业务、销售和财务等。

1.2.1. 专业人士对 Text-to-SQL 系统的需求

专业人士已经能熟练使用 SQL 完成工作，但是企业繁杂的业务线导致数据库表的梳理是一件十分令人头疼的工作，后续编写复杂的 SQL 查询也是一项常见但耗时的任务。对于数据分析师、数据库管理员和其他 IT 专业人士来说，拥有以下功能的 Text-to-SQL 系统能大幅度提升专业人士的工作效率：

（1）支持跨多个数据库表的关联查询：数据分析师日常工作中经常会从多个数据库中抽取数据进行关联查询，分析数据之间的相关性以及计算相关的指标。因此 Text-to-SQL 系统需要能根据自然语言快速的判断需要找到哪些表、需要使用哪些字段进行分析和需要使用哪些字段进行关联。

（2）快速生成 SQL 语句：数据分析师、数据库管理员和其他 IT 专业人士经常需要编写复杂的 SQL 查询。基于大模型的 Text-to-SQL 系统需要能通过自然语言输入自动生成准确的 SQL 代码，减少手动编写的时间和工作量。

（3）支持复杂查询逻辑：专业人士需要处理涉及多个表联查、子查询、聚合函数等复杂逻辑的 SQL 查询。基于大模型的 Text-to-SQL 系统需要能理解并生成符合这些复杂逻辑的 SQL 语句，确保结果的准确性和完整性。这可以减少出错的可能性，提升查询的质量。

（4）灵活性与可扩展性：基于大模型的 Text-to-SQL 系统需要具备高度的灵活性和可扩展性，可以适应不同类型的数据库和查询需求，支持多种主流数据库（如 MySQL, PostgreSQL, Oracle 等），并且允许用户定义自定义函数和视图。此外，插件机制使得用户可以添加自定义的功能模块或第三方工具，进一步增强系统的功能。

1.2.2. 非专业人士对 Text-to-SQL 系统的需求

Text-to-SQL 系统更重要的意义是赋能非专业人士进行数据分析。许多业务人员和技术支持人员虽然具备丰富的领域知识，但缺乏编写复杂 SQL 查询的能力。这限制了他们从数据中提取价值的能力，并影响了数据驱动决策的效率。因此，非专业人士对 Text-to-SQL 系统有以下需求：

（1）降低技术门槛： Text-to-SQL 系统需要让非技术人员能够通过自然语言与数据库进行交互，无需掌握复杂的 SQL 语法即可执行数据查询和分析任务。通过直观的查询方式极大地降低了数据使用的门槛，使更多人能够参与到数据分析过程中。

（2）提升数据分析能力：非专业人士需要通过简单的自然语言描述来获取所需的数据来支持业务决策。例如，市场部门的经理可以通过简单的自然语言查询如“列出上个月销售额增长最快的地区”来获取所需的数据，而无需依赖 IT 团队的支持。这不仅提高了数据分析的效率，还增强了业务人员的数据驱动决策能力。

（3）增强用户体验：传统的 SQL 查询界面对于非技术人员来说不够友好。Text-to-SQL 系统需要通过基于大模型的自然语言接口提升用户体验，使用户能够以更加自然和直观的方式与数据进行交互，从而提高用户满意度和系统的使用率。

1. 3. 非功能性需求

为了更好的满足用户在使用系统时的体验，在设计系统时需要对性能指标、接口要求、可维护性与可靠性和用户体验给出具体的要求。具体如下：

1.3.1. 性能指标

(1) 系统应在 5 秒内完成从自然语言输入到生成 SQL 查询的过程。
(2) 对于简单查询，系统应在 1 秒内返回结果；对于复杂查询，系统应在 5 秒内返回结果。
(3) 系统应支持至少 100 个并发用户的查询请求，确保在高负载情况下仍能保持稳定的性能。
(4) 系统每分钟应能够处理至少 1000 次查询请求，确保在高峰期也能提供高效的服务。
(5) 系统在运行时应尽量减少 CPU、内存和 I/O 资源的消耗，确保不会对数据库服务器造成过大的负担。每个查询请求的平均 CPU 使用率不超过 20%，内存使用率不超过 30%。

1.3.2. 接口要求

(1) 系统应提供标准的 RESTful API 接口，支持 HTTP/HTTPS 协议，方便与其他系统进行集成。
(2) API 接口应接受自然语言查询作为输入参数，并支持 JSON 格式的数据传输。
(3) API 接口应支持多种输出格式，如 JSON、CSV 等，以便用户根据需要选择合适的格式。
(4) 系统应提供一个简洁直观的 Web 界面，使用户可以通过浏览器轻松访问和使用系统。
(5) 系统应支持与常见的数据可视化工具（如 Tableau, Power BI 等）和 ETL 工具（如 Talend, Informatica 等）集成，实现无缝的数据分析和处理流程。
(6) 系统应提供插件机制，允许用户添加自定义的功能模块或第三方工具，进一步增强系统的功能。
(7) 系统应提供开放的 API 文档，方便开发者进行二次开发和定制化需求的实现。

1.3.3. 可维护性与可靠性

(1) 系统应定期自动备份历史返回的结果，确保在发生故障时能够快速恢复。
(2) 系统应具备灾难恢复机制，能够在出现重大故障时迅速恢复服务。
(3) 系统应提供实时监控功能，监测系统的运行状态、性能指标和异常情况。
(4) 当系统检测到异常情况时，应通过邮件、短信等方式及时通知管理员。
(5) 系统应记录详细的日志信息，包括系统操作、错误日志、性能日志等，便于问题排查和优化。
(6) 系统应支持自动更新机制，确保用户能够及时获得最新的功能和安全补丁。
(7) 系统应提供版本管理功能，支持回滚到之前的版本，以应对可能出现的问题。

1.3.4. 用户体验

(1) 系统应提供简洁直观的用户界面，易于上手和操作。
(2) 系统应提供详细的在线帮助文档和教程，指导用户如何使用系统。
(3) 系统应提供培训课程和技术支持，帮助用户快速掌握系统的使用方法。
(4) 系统应具备自动补全功能，根据用户的输入提供建议，减少输入错误并

提高查询准确性。

(5) 系统应提供常见查询的示例和模板，帮助用户快速理解和使用系统。

(6) 系统应提供反馈机制，允许用户对查询结果进行评价和反馈，以便不断优化系统性能。

第二章　系统设计

本项目构想一套基于开源大模型接口的真实场景安全可用的NL2SQL方案。通过从业务数据库元数据抽取表和字段的信息并进行映射后建立全域统一规范化数据库，使得大模型更容易通过业务需求生成准确的 SQL 代码。项目通过斯坦纳树算法实现数据库表压缩，同时在进行表搜索时还能获取到表的关联字段，提升 SQL 代码的可用性。最后为了满使生成的 SQL 代码满足规范，项目采用提示词结合大模型的基础能力实现定制化 SQL 代码的需求。项目设计的系统流程图如图 1。

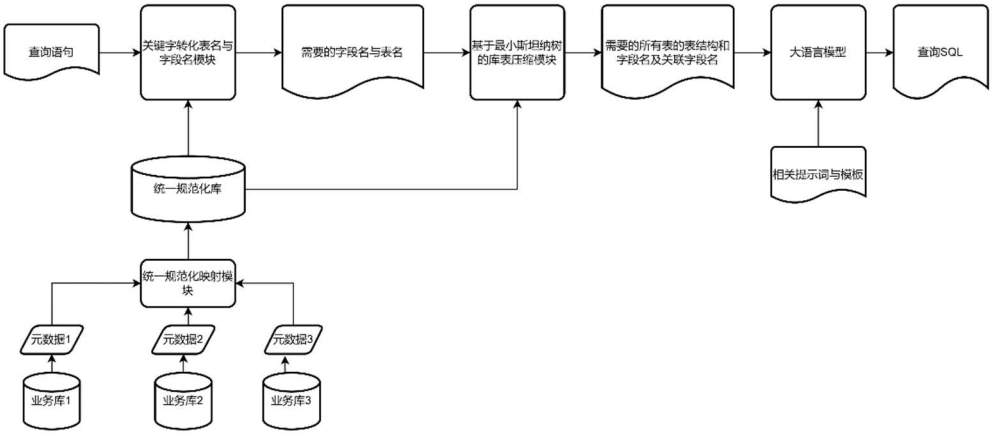


图 1NL2SQL 系统流程图

2.1 设计背景

大模型具备的自然语言生成计算机语言能力能够解决 Text-to-SQL 方向上的诸多痛点，给该领域带来了显著的进步。大模型在经过大规模的 Text-to-SQL 数据集的预训练，能够更好地理解复杂的自然语言查询，并将其转化为结构化的 SQL 语句。这不仅提高了从自然语言到 SQL 转换的准确性，还使得系统能够处理更加复杂的查询逻辑。自注意力机制等技术使得大语言模型可以捕捉句子中的长距离依赖关系和深层语义信息。这意味着它们能更好地理解用户的意图，即使在模糊或不明确的情况下也能生成合理的 SQL 语句。同时，基于大模型的 Text-to-SQL 解决方案能够实现更高程度的自动化，减少人工干预的需求，并且大语言模型通常具有较强的泛化能力，能够在不同的数据库模式下工作。

尽管如此，在实际业务场景中应用这些大型模型时仍面临一些挑战：一是实际业务环境中的数据库往往包含大量的表和字段，将所有信息一次性提供给大模

型处理并不现实，所以需要寻找一个方法将查询需求关联的表和字段提取以缩减信息规模，去除不必要的信息；二是现阶段企业的数据库系统建设的非常庞大且结构复杂，表描述和字段定义没有一个统一的规范，最好的情况是在单个业务线内有统一的规则，但是在公司范围内多条业务线之间的数据库设计没有同一规范，这导致大模型理解业务需求出现困难，且对多表联查时造成极大的障碍；三是企业可能会对 SQL 代码有定制化要求，在线大模型生成的代码可能不满足企业要求。

2.2 解决方案

为了解决大模型在 Text-to-SQL 领域面临的诸如无法直接处理业务所有表和字段、难以理解当前不规范数据库表描述和字段定义和无法满足企业定制化需求。本项目将针对现有基于大模型的方案的问题，依赖开源大模型接口设计一套在真实场景安全可用的 Text-to-SQL 方案。为了解决上述问题，本项目设计下面四大主要模块解决对应的问题。

2.2.1. 全域统一规范化数据库模块

为了提高数据的一致性和可管理性，构建一个全域统一规范化的数据库是必要的。这要求对现有的表描述、字段名及其定义进行标准化处理。然而，许多企业的现有线上业务已经稳定运行多年，直接修改与业务相关的表名、字段名及相关描述可能会引发严重的生产事故，影响业务连续性和数据完整性。

当前实现全域统一规范化库的首选办法是创建一个用于实现现有业务库到标准命名和定义映射的中间数据库，这个中间数据库将作为桥梁，连接现有业务数据库与全域统一规范化的标准。具体来说，可以通过抽取各个业务系统中的元数据（即记录当前数据库中所有表及计算信息的数据）至全域统一规范化库。接着，根据预先设定的统一字段命名规则和字段定义，为这些元数据添加映射字段。通过这种方式，可以建立一套从现有业务库中的表名、字段名及相关描述到全域统一规范化标准之间的映射关系。

通过建立映射库及抽取各业务元数据库的方法不仅避免了直接修改现有业务系统的风险，还为企业提供了一种灵活的方式来逐步过渡到新的标准体系。此外，这种做法也便于维护和扩展，当有新的业务需求或系统变更时，只需更新映射关系即可，而无需大规模重构现有数据库结构。采用这一映射机制，最终可以实现整个组织范围内数据的一致性，为未来的数据分析、报告生成以及跨部门协作打下坚实的基础，更重要的是为后续 Text-to-SQL 的实现提供一个一致的、规范的和灵活的表和字段搜索库，使得大模型能够更准确的定位到相关字段和表。

2.2.2. SQL 查询需求关键字提取与转换表名与字段名模块

后续的库表压缩流程需要指定的表名和字段名，因此从 SQL 查询需求转化到表名与字段名是十分关键的一步。项目将该任务分为两步，第一步是从 SQL 查询需求中获取表名关键字和字段名关键字，第二步是将关键字用于搜索出对应的表名和字段名。

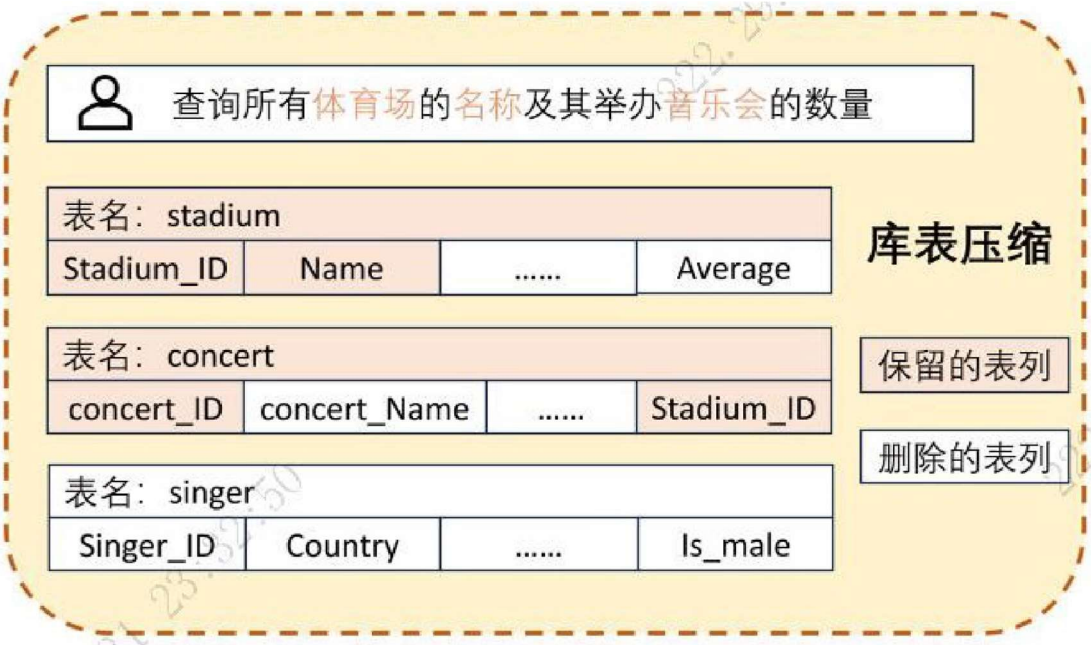
项目采用大模型提取 SQL 查询需求中的表关键字和字段关键字。从 SQL 查询需求中提取的表名关键字和字段名关键字的准确性对后续处理至关重要。本模块通过大语言模型的自然语言处理能力，结合深度语义分析技术，能够精准地从复杂的自然语言表达中提取出相关的关键字信息。提取过程注重上下文的理解，针对多义词和语境依赖的需求，能够灵活适配不同的语句结构。例如，当用户查询中提及“客户”时，系统能够根据语义判断其在具体查询中的实际作用，从而避免错误提取导致的后续处理问题。

项目使用关键字搜索引擎对表和字段的备注内容进行关键字匹配得出对应的表名和字段名。将提取的关键字准确映射为实际数据库中的表名和字段名是本模块的另一关键任务。通过构建高效的关键字搜索引擎，如 ElasticSearch 等引擎，系统能够在大规模的元数据中快速找到对应的表名和字段名。映射过程严格依照统一的规范化规则，通过在表和字段的备注内容中进行搜索，确保生成的结果与

数据库的实际结构保持一致。同时，为了提高模块的实用性，搜索引擎能够适配不同业务场景中的多种数据库结构，确保对动态变化的数据库需求具有良好的兼容性。如果在关键字搜索过程中出现不匹配或多结果不确定的情况，系统还设计了交互式错误提示机制，允许用户快速修正或确认映射结果，从而进一步提高系统的鲁棒性和用户体验。

2.2.3. 基于斯坦纳树算法的库表压缩与关联字段获取模块

库表压缩会根据用户问题从原始库表信息中筛选最相关的表列，删除其余表列，从而达到精简库表信息的目的。在基于大模型的 Text-to-SOL 技术中，库表压缩极为重要，这是因为其可以帮助模型避免几余信息干扰，只关注与问题相关的表列，从而生成正确的 SQL 语句。



过去的 Text-to-SQL 技术会采用基于机器学习的库表压缩方案，将库表压缩视为一个分类问题，训练一个模型来将表列分为与问题相关和不相关两类。然而这种方案的效果十分依赖标注数据的质量，且通用性较差。基于大模型的 Text-to-SQL,技术则直接用大模型本身来进行库表压缩，即在生成 SOL 前，先让大模型去提取和问题相关的表列信息，再用提取后的信息去生成 SQL。但是大模型的输人长度有限，如果原始库表信息很长，那么大模型的压缩效果会显著退化，甚至完全不可用。另一方面，为保证压缩质量，这些方案一般都采用能力更强的闭源大模型(ChatGPT/GPT4),同样存在数据泄露的风险。

本项目使用基于斯坦纳树算法（Steiner Tree Algorithm）的压缩方案，实现高

效且安全的库表压缩，并且在压缩过程中能够得出表关联字段，减少大模型运算压力，提升 SQL 生成的准确性。斯坦纳树算法是一种用于优化网络连接问题的算法。其目标是在给定的点集（称为终端点）中，找到一个连接所有点的最小树，并允许添加额外的中间点（称为斯坦纳点）来减少总连接距离或成本。项目通过把字段和表都抽象为图中的节点，通过斯坦纳树求取连接所有目标字段的最小生成树，达到库表压缩的目的。

具体的设计思路是将数据表之间关联的字段组合成为单独的节点，而数据表也作为节点存在于图中。如图 2 所示，黑色节点是代表数据表，蓝色节点代表字段组合。将表之间关联的字段组合后可以更灵活的连接两张表，如果某张数据表的主键是联合字段，将某个字段作为表之间的连接条件将会造成数据重复的问题，但是将联合字段作为连接条件可以精准的确定单条数据。在求解出最小斯坦纳树之后，表之间的关联字段也可以直接由字段组合节点得出，从而得出满足查询需求的表和表间关联字段。

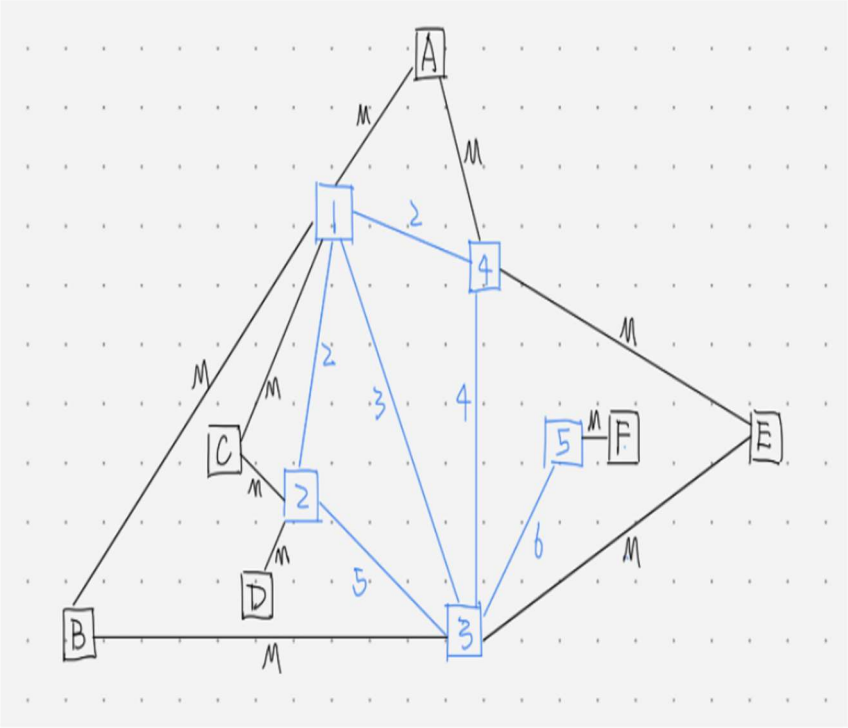


图 2 关系型数据库图

2.2.4. SQL 语句生成模块

根据库表压缩后得出的数据表名得出表的创建语句，同时库表压缩还输出了表之间的关联字段，大模型可以根据 SQL 查询需求以及目标表的表创建语句和表之间的关联字段生成满足需求的 SQL 代码。系统会根据库表压缩得出的表名

查询数据库得出表的创建语句，创建语句中的信息十分完整，将有利于大模型生成准确的 SQL。库表压缩得出的关联字段可以进一步提示大模型如何连接两张数据表，得出更加准确的 SQL 代码。表创建语句和关联关系都是作为提示添加到提问内容中，结合最初的查询需求向大模型发起提问，最终得出 SQL 代码。

对于定制化 SQL 需求，项目选择提示词引导大模型进行 SQL 代码的规范化输出。企业对 SQL 语言编写会有一些相关的规范，比如会限制子查询的使用或限制表关联的层数等，为了满足规范需要对大模型 Text-toSQL 进行特殊的处理。目前主要的方法有两种，一是对大模型进行微调，使其在生成 SQL 代码时能够更好地遵循企业的规范，这可以通过在包含企业规范的数据集上进行额外的训练来实现。二是设计专门的提示词，提示词可以包含具体的示例和模板，引导大模型在生成 SQL 代码时遵守企业的规范。第一种方法需要企业有相关资源建设大模型且有能力对大模型进行微调，这使得该方案耗费的人力物力十分巨大，且本项目也根本不具备相关条件完成大模型微调，因此项目选择提示词引导的方式实现大模型生成满足规范的 SQL 代码。

第三章　系统实现

3. 1. 技术运用

项目采用前后端分离的架构进行实现，前后端及数据库分别采用不同的技术栈实现以最优地满足功能需求、用户需求和性能需求等。项目采用多种开发工具提升开发的效率，同时采用特定的运行环境保证系统的稳定运行。具体内容如下：

3.1.1. 前端技术栈

前端指的是用户可以直接看到并与之交互的部分。前端开发主要涉及网页和移动应用的界面设计与实现，确保用户能够通过浏览器或应用程序流畅地使用软件。前端开发人员利用 HTML 定义网页的内容结构，CSS 控制页面的布局、颜色、字体等视觉表现，而 JavaScript 则为网站添加交互性，比如响应用户的点击、提交表单、动态更新内容等。为了适应不同设备上的正确显示并提供一致的用户体验，前端开发者还会关注响应式设计，确保网站能够在桌面电脑、平板电脑和智能手机上良好呈现。

为了提高项目开发的效率和代码质量，本项目前端使用 Vue.js 框架来构建复杂的用户界面。项目使用 Element-ui 组件库来提升开发效率和用户界面的友好性；前端网络请求由 axios 库接收并转发至后端接口进行处理，同样的，后端解决返回数据也由 axios 库接收后在前端页面进行展示；此外，项目使用浏览器内置的 IndexedDB 功能本地暂时存储用户信息与对话内容，使得用户在刷新页面后仍然可以恢复之前的状态和数据。

3.1.2. 后端技术栈与架构

在计算机开发中，后端（Backend）指的是运行在服务器上的应用程序部分，它处理业务逻辑、数据存储、用户认证、服务器配置以及与外部服务的交互等。后端开发者的工作是构建和维护这些服务器端组件，确保它们高效地工作，并能可靠地响应来自前端或其它客户端的请求。

为了提升开发速度和开发质量，项目后端整体采用 Spring Boot 框架进行开发。采用 Spring Boot 框架进行开发具有诸多优势，它简化了基于 Spring 的应用程序的创建过程，提供了开箱即用的功能和自动配置机制，使开发者能够更专注于业务逻辑的实现。Spring Boot 自动化了许多常见的配置步骤，使得开发者无需手动编写大量的 XML 或 Java 配置代码。它根据添加到项目中的依赖关系

自动配置 Spring 应用程序，大大减少了初始设置时间和复杂度。借助其内置的嵌入式服务器（如 Tomcat, Jetty 或 Undertow），Spring Boot 应用可以直接运行，而不需要部署到外部容器中，这不仅加快了开发周期，还简化了部署流程，并且非常适合微服务架构下的快速迭代。Spring Boot 还提供了一套丰富的 starter 依赖项，这些依赖预配置了常用的第三方库和服务集成，例如数据库连接、安全认证、缓存等。通过简单的声明式配置，开发者可以快速集成所需功能，减少了查找和配置第三方库的时间。

系统 web 模块分为 Controller、Service 和 Mapper 三个层次，这种结构为应用带来了显著的好处。该结构明确了各层的职责：Controller 层负责处理用户请求和响应，Service 层处理复杂的业务逻辑，而 Mapper 层专注于数据库交互。分离带来了高度的模块化，每层可以独立开发和测试，极大提高了代码的可维护性和可扩展性。此外，这种分层也有助于解耦，使得各层之间的依赖降低，便于进行单独的优化和修改，而不会影响到其他层。此结构支持更精确的安全控制和更简单的代码管理，特别适用于需要处理复杂业务逻辑和数据操作的中大型项目，使得整个开发过程更加高效和灵活。

项目采用 Nginx 提供 web 服务。Nginx 是一款高性能的 HTTP 和反向代理服务器，同时也支持 IMAP/POP3 代理服务。它以其轻量级、高效能和稳定性而闻名，特别适合处理高并发连接的情况。Nginx 不仅能够作为 Web 服务器直接提供静态内容，如 HTML 页面、图片和 CSS 文件，还可以作为一个高效的反向代理服务器，将请求转发给后端的应用服务器，例如运行 PHP、Python 或 Java 应用程序的服务，并将响应返回给客户端。

后端开发调用已有的库提升开发效率，减少重复劳动和错误的同时提升系统安全性。项目采用 MyBatis 作为 ORM 工具来简化数据库操作，并确保数据库访问的效率和安全。使用 Lombok 库用于减少样板代码，自动化地生成 getter 和 setter 方法、构造函数以及其他常用的方法。

3.1.3. 数据库和数据管理

项目使用 MySQL 数据库存储关系型数据，项目的规范化映射库也是基于 MySQL 建立。MySQL 数据库以其开源免费的特性降低了初期投资成本，同时提供了卓越的性能和可靠性，能够在高并发场景下快速处理大量读写操作，并通过支持 ACID 事务确保数据的一致性和完整性。它不仅易于使用和管理，拥有直观的命令行工具和图形界面客户端，还具备广泛的跨平台兼容性，可以在多种操作系统上顺畅运行。MySQL 的灵活性体现在其多样化的数据类型和可选择的存储引擎上，能够根据具体需求优化数据库性能。此外，它内置了强大的复制和

集群功能，保障系统的高可用性和容错能力，同时提供了丰富的安全措施保护数据免受未授权访问。凭借其活跃的社区支持和与众多编程语言及工具的良好集成，MySQL 成为了构建高效、稳定且经济实惠的数据存储解决方案的理想选择，特别适合 Web 应用和在线事务处理系统。

项目使用 ElasticSearch 引擎支撑关键字搜索。Elasticsearch 是一个分布式、RESTful 风格的搜索引擎和分析引擎，以其卓越的全文搜索能力和实时数据分析功能而闻名。它能够处理海量数据并提供快速的查询响应，非常适合需要高效检索大量信息的应用场景。作为一个高度可扩展的解决方案，Elasticsearch 可以轻松地从单一服务器扩展到由数百个节点组成的集群，确保即使在数据量急剧增长的情况下也能维持高性能。它的灵活性不仅体现在对结构化和非结构化数据的支持上，还在于可以与各种数据源无缝集成，包括关系型数据库、日志文件和其他存储系统。其内置的自动分片和复制机制增强了系统的容错性和高可用性，即使部分节点出现故障也不会影响整体服务的连续性。同时，借助 Kibana 这样的可视化工具，Elasticsearch 让监控和管理变得更加便捷友好。

3.1.4. 运行环境与开发工具

本项目使用可以跨平台的 Java 语言进行开发,具体的开发和运行环境配置如下：

（1） 硬件环境

处理器： Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz2.11 GHz

内存容量：8G

硬盘容量：1T

（2） 软件环境

服务器操作系统：Debian 12

Java 运行环境：JDK 17

数据库系统:MySQL、Elasticsearch

开发工具：Idea、Git、maven 等

3.2. 系统模块实现

3.2.1. 规范化数据库模块

全域统一规范化库是对现有业务数据库元数据的一个统一映射，通过定义一套统一的数据模型、命名规则和字段定义等标准，来标准化数据结构，从而消除不同部门或系统间的数据异构性，实现数据的一致性和互操作性。为了统一规范化的目标而创建的一个中间数据库，将现有业务数据库的数据映射到统一标准上，这样既保护了现有数据资产，又提供了灵活的过渡方式，并便于未来的维护和扩展。

统一规范的数据结构能够大大降低系统处理异构数据消耗的成本，使得后续在图中求取最小斯坦纳树更加准确与快速。全域统一规范化库模块用于构建一个组织或企业内部建立的集中式数据管理和标准化框架，它为高级数据分析工具和机器学习算法提供了更加一致和可靠的输入源，增强了企业的数据分析能力；同时还简化了新旧系统的集成与迁移过程，降低了成本和复杂度。此外，全域统一规范化库提供了一个规范化的表和字段搜索库，使得自然语言处理模型能够更精准地理解和执行查询指令。

为了满足上面的需求，统一规范化库的数据库结构设计如下：

（1） standard_table

标准表名和源表名之间的映射，还包含表备注、表数据量、表粒度和表 DDL 语句等信息。

字段名称	数据类型	是否为空	是否为主键	注释/描述
standard_table_id	int	NOT NULL	是	标准规范表 id，自动递增，作为主键
original_table_name	varchar(255)	NOT NULL	否	源库表名
standard_table_name	varchar(255)	NULL	否	标准规范表名
table_schema	varchar(255)	NOT NULL	否	源库名
table_comment	varchar(255)	NOT NULL	否	表备注
column_rows	int	NULL	否	表数据量
granularity_name	varchar(255)	NULL	否	表的粒度
granularity_id	int	NULL	否	粒度 ID
original_table_ddl	text	NULL	否	表的 DDL 语句

（2） standard_column

标准字段名和源字段名的映射，还包含字段备注等信息。

字段名称	数据类型	是否为空	是否为主键	注释/描述
standard_column_id	int	NOT NULL	是	主键 id，自动递增
standard_column_name	varchar(255)	NULL	否	标准表名
original_column_name	varchar(255)	NOT NULL	否	源库字段名
column_comment	varchar(255)	NOT NULL	否	字段备注
table_schema	varchar(255)	NOT NULL	否	源库名

（3） granularity

粒度表，存储所有定义的粒度。

字段名称	数据类型	是否为空	是否为主键	注释/描述
granularity_id	int	NOT NULL	是	主键 id，自动递增
granularity_name	varchar(255)	NOT NULL	否	粒度名称
granularity_comment	varchar(255)	NOT NULL	否	粒度备注

（4） standard_column_table

表-字段表，表示每个表包含哪些字段。

字段名称	数据类型	是否为空	是否为主键	注释/描述
standard_column_table_id	int	NOT NULL	是	主键 id，自动递增
standard_table_name	varchar(255)	NOT NULL	否	标准表名
standard_column_name	varchar(255)	NOT NULL	否	标准字段名
standard_column_id	int	NOT NULL	否	标准字段 id
standard_table_id	int	NOT NULL	否	标准表 id

（5） standard_granularity_column

粒度-字段表，表示每个粒度包含哪些字段

字段名称	数据类型	是否为空	是否为主键	注释/描述
granularity_id	int	NOT NULL	否	粒度 id
granularity_name	varchar(255)	NOT NULL	否	粒度名
standard_column_name	varchar(255)	NULL	否	标准字段名
standard_column_id	int	NULL	否	标准字段 id
granularity_column_id	int	NOT NULL	是	主键 id，自动递增

3.2.2. 关键字提取与转化模块

关键字提取与转化模块利用大语言模型的自然语言处理（NLP）能力和深度语义分析技术，从复杂的用户查询中精准提取表名和字段名关键字，并根据上下文理解灵活处理多义词和语境依赖，确保提取的准确性。该模块通过高效的关键字搜索引擎 ElasticSearch ，在大规模元数据中快速匹配并映射关键字为实际数据库中的表名和字段名，严格遵循规范化规则以保持一致性。为了适应不同业务场景和动态变化的数据库结构，搜索引擎具备良好的兼容性，并设计了交互式错误提示机制，允许用户快速修正或确认映射结果，从而提高系统的鲁棒性和用户体验，确保查询需求解析的高效性和准确性。

为了满足上面的要求，模块的程序流程图如下：

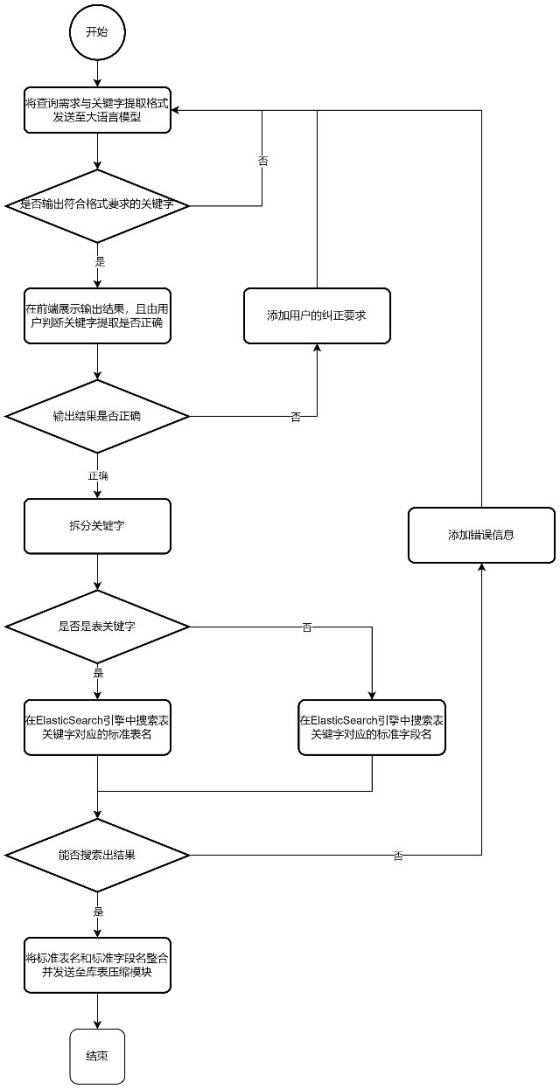


图 3 关键字提取与转化模块流程图

3.2.3. 基于最小斯坦纳树算法的库表压缩与关联关系获取模块

本项目采用基于斯坦纳树算法的压缩方案，通过将数据表和组合关联字段抽象为图中的节点，实现高效且安全的库表压缩。该算法优化网络连接，求取连接所有目标字段的最小生成树，从而减少大模型运算压力并提升 SQL 生成的准确性。通过将表间关联字段组合成单独节点，不仅解决了联合字段作为连接条件时可能引起的数据重复问题，还能够灵活地连接多张表，并在求解出最小斯坦纳树后直接得出满足查询需求的表及其关联字段，确保数据的精准性和查询效率。这种方法既简化了数据库结构，又提高了处理复杂查询的能力。

模块的程序流程图如下：

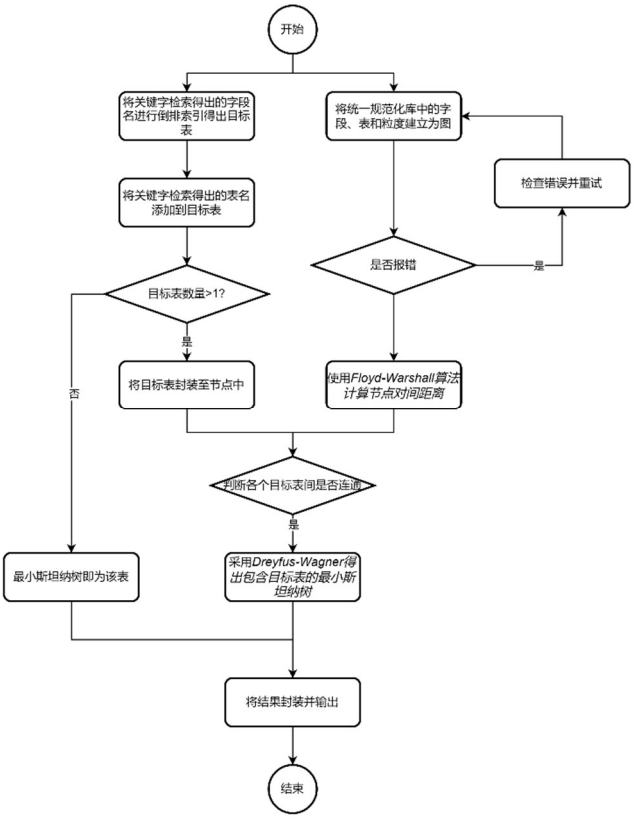


图 4 库表压缩模块

3.2.4. SQL 生成模块

SQL 生成模块基于库表压缩后得出的数据表名和表间关联字段，结合完整的表创建语句信息，为大模型提供准确的提示，以生成满足查询需求的 SQL 代码。系统通过查询数据库获取表的创建语句，并利用库表压缩输出的关联字段指导大模型如何连接数据表，确保生成的 SQL 代码更加精准。为了应对定制化 SQL 需求并遵守企业的编写规范，项目选择了提示词引导的方式，而非资源密集型的大模型微调。具体来说，设计了专门的提示词，包含示例和模板，以引导大模型在生成 SQL 代码时遵循企业特定的规范，如限制子查询的使用或表关联的层数等。这种方式不仅降低了实施成本，还确保了生成的 SQL 代码既符合企业标准又高效准确。

模块的程序流程图如下：

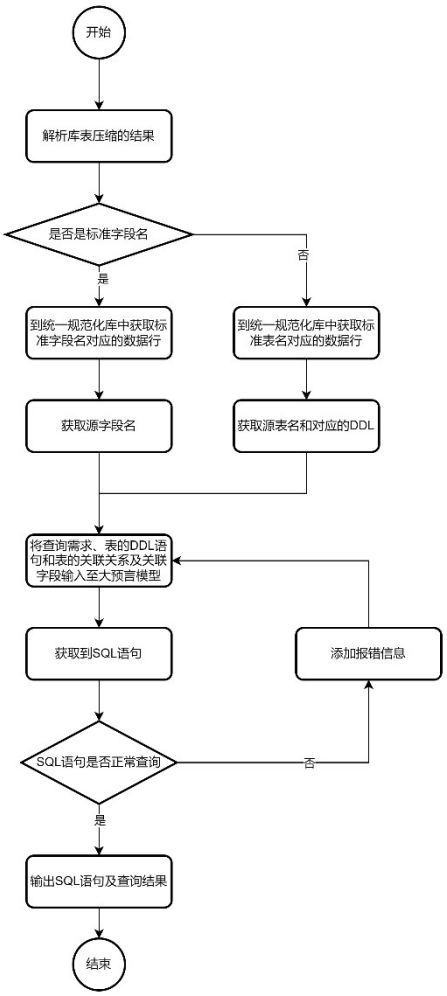


图 5 SQL 生成模块流程图

3.3. 系统部署

项目的部署使用 Docker 完成自动化部署，使用 Docker Compose 进行容器编排。项目后端的 Maven 构建与编译、MySQL 安装配置与数据导入、Elasticsearch 安装配置与数据导入、Nginx 安装配置等服务都使用容器化部署。前端基于 Vue 开发，因此 Node.js 等服务也通过容器化部署。通过容器化，可以将这些服务集成到一个统一的部署环境中，简化复杂系统的管理和维护。

后端部分首先需要创建一个 Dockerfile，将 Maven 构建的 Java 应用打包为 Docker 镜像，确保项目的运行环境与依赖一致性。MySQL 和 Elasticsearch 则可以直接通过官方镜像使用，并根据项目需求进行基本配置。前端的 Vue 应用在构建完成后，也可以容器化处理，将其打包后的静态资源托管到 Nginx 容器中，以提供高效的静态文件服务。

将代码托管到 Git 平台（如 GitHub 或 GitLab）后，可以借助 CI/CD 工具实现从代码提交到部署的自动化流程。每当有新代码提交时，流水线会自动触发以下步骤：编译后端 Java 代码、打包前端 Vue 应用、构建 Docker 镜像并推送至容器注册表。接着，通过 Docker Compose 拉取最新镜像并启动或更新相关容器。

在部署过程中，Docker Compose 是核心工具，用于定义和管理项目所需的所有服务。Compose 文件中可以明确指定后端服务如何连接 MySQL 和 Elasticsearch，以及前端服务如何通过 Nginx 代理访问后端 API。通过这种方式，可以确保各服务之间的网络通信和依赖关系配置正确。此外，Compose 支持将环境变量注入到容器中，用于动态配置敏感信息如数据库用户名、密码以及 Elasticsearch 节点地址。

为提高部署的稳定性，建议配置容器的重启策略和健康检查。例如，MySQL 容器可以自动重启，确保在崩溃后能够快速恢复；后端和前端容器则通过健康检查定期监测服务状态，自动移除或重启异常实例。此外，为避免用户访问中断，可以结合 Docker Compose 的滚动更新策略，在发布新版本时逐步替换旧版本服务，保证整个系统的平滑过渡。

通过这一自动化部署流程，可以实现 NL2SQL 项目从代码提交到生产环境上线的高效协作。所有服务的容器化处理和 Docker Compose 的统一编排，不仅简化了复杂依赖关系的管理，还提供了可靠的扩展性和容灾能力，为项目的持续迭代提供了坚实的技术基础。

第四章　系统测试与评估

本章围绕 NL2SQL 系统的测试与评估展开，详细介绍系统从单元测试到集成测试，再到端到端测试的整体测试框架与实施策略，同时探讨了基于 Spider 数据集的系统性能评估方法。通过层次分明的测试流程和权威数据集的评估，本章为 NL2SQL 系统的功能完善与优化提供了系统性的指导。

4.1. 单元测试

系统整体基于 Java 语言进行编写，在开发过程中对系统的代码单元会进行单元测试以确保系统基本构建块的正确性，提高整个系统的可靠性。由于大语言模型对于系统来说属于黑盒，因此涉及到调用大语言模型接口的结果不适用单元测试，单元测试环节仅应用于接口访问、数据库访问与数据处理等具有白盒特性的方法和模块。测试工具使用 JUnit 编写和组织单元测试，Mockito 用于模拟对象的依赖关系，AssertJ 用于解决 Junit 的断言短板。

对于各个模块的单元测试关注重点不一样。首先，统一规范化映射模块负责将不同业务库中不规范的表名、字段名和外键连接数据转换为统一格式，通过测试不同数据源映射的准确性和一致性来保证系统数据格式的规范性。统一规范化库作为中央存储库，需确保数据存储和检索符合统一格式，单元测试应验证数据格式的一致性和规范性。对于“关键字转化表名与字段名”模块，需要测试关键字能否准确转换为数据库中的表名和字段名，验证该模块在处理不同查询时的灵活性和正确性。最后，库表压缩模块基于最小斯坦纳树算法，负责将数据库表集合压缩为满足查询需求的最小表集合，单元测试应检测输入指定字段和表名时，能否返回具有正确关联关系和相关节点的“最小斯坦纳树”。

4.2. 集成测试

集成测试用于验证系统中各个模块或组件能否正确协同工作，确保系统在整体运行时能够实现预期的功能。单元测试通常仅验证模块的单独功能，而集成测试则进一步检验模块间的交互，帮助识别在接口和数据流中可能存在的问题。对于 NL2SQL 系统，集成测试特别重要，因为该系统包含自然语言解析、关键词转换、库表压缩、SQL 生成等多个模块，只有在这些模块正确协作时才能生成准确的 SQL 查询。

当单独的模块开发完成后，需要集成相邻的两个模块进行测试，确保模块质

检可以正常交互。当完成规范化库模块的开发后，需要将该模块连接至业务库并测试是否能正常抽取业务库元数据，且格式是否满足规范。随后将其连接至规范化库，确保规范化的元数据能够存入库中。完成上述集成测试之后，再将关键字转化模块集成至已测试的链路中，不断解决模块间存在的交互问题，确保系统链路中可以顺利流动正确格式的数据。

4. 3. 端到端测试与数据分析评估

端到端测试旨在模拟真实用户在实际使用场景中的行为，验证整个系统从用户输入到最终输出的完整流程。即使每个单独的模块通过了单元测试和集成测试，系统在整体运行时仍可能遇到问题，而端到端测试可以验证复杂的业务流程，确保系统在处理多步骤操作时各个模块之间可以正确的进行交互。端到端测试还可以发现并修复潜在的性能瓶颈和安全漏洞，提高系统的可靠性和稳定性。

本系统端到端测试更重要的一个内容是验证 NL2SQL 功能的准确度，本项目采用 Spider 数据集对 NL2SQL 的效果进行评估。系统使用的大语言模型接口使得系统整体的测试属于黑盒测试，需要使用测试 Text-to-SQL 的数据集对准确度进行测试。Spider 是一个广泛用于自然语言处理（NLP）领域，特别是文本到 SQL（Text-to-SQL）任务的数据集，主要由 11 名耶鲁大学学生标注，包含 10181 个问题和 5693 个独特的复杂 SQL 查询、200 个具备多个表的数据库，覆盖 138 个不同领域，实际应用性很强。

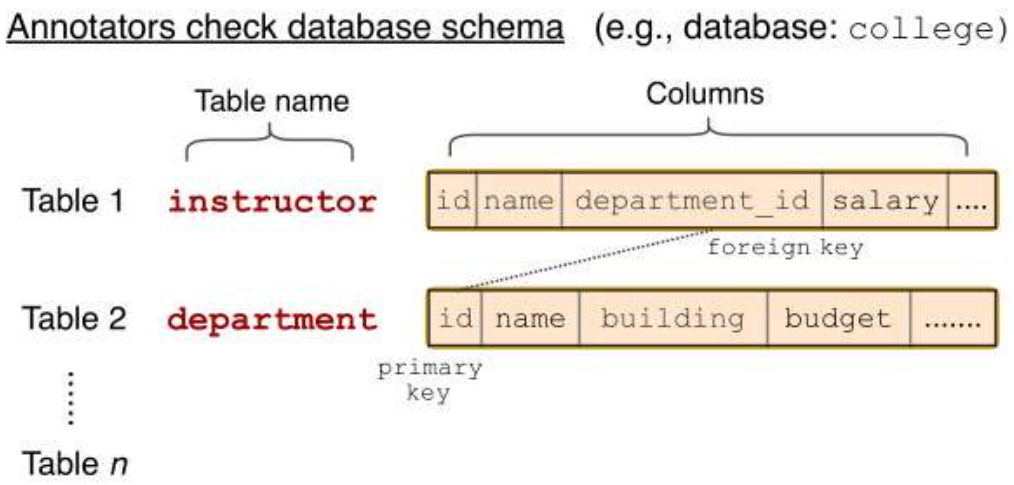


图 6 测试案例

Spider 数据集测试方案

Spider 数据集将 140 个数据库作为训练集，20 个数据库作为验证集，40 个数据库作为测试集。同时，测试集不公开，研究者需要将自己的模型在验证集上调试到最佳性能后将模型发给 Spider 官方，有官方在测试集上进行测试，并将结果公布在 Leaderboard 上以确保模型方法对比的公平性。主要评价指标有两个：

Exact Match(EM): 模型预测的 SQL 语句必须与 ground truth 完全一样。由于 SQL 语句中句式的多样性，完成同样的任务不一定能唯一确定 SQL 语句。因此这是一种过于严格的评价指标。

Execution Accuracy(EX)：模型预测的语句执行后所得的结果与 ground truth 一样。相较于 EM 放宽，但又引入另一种歧义：模型可能预测出语义完全不同的 SQL 语句，但恰好有相同的执行结果。这样的情况也会被该指标纳入正确范围。

如上图所示，假如真实的 SQL 语句为 SELECT COUNT(name) WHERE SSN = 123，只有当预测结果与该语句一模一样时 2，才认为预测正确。假如预测的 SQL 语句为 SELECT COUNT(SSN) WHERE SSN = 123，这事，只要预测 SQL 语句的执行结果与真实 SQL 语句执行结果一致时，就认为预测正确。