软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(9):4132−4152 [doi: 10.13328/j.cnki.jos.006646]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

正反例归纳合成 SQL 查询程序*

张 健[1,2], 李 弋[1,2], 彭 鑫[1,2], 赵文耘[1,2]

[1](复旦大学 软件学院, 上海 200433)

[2](上海市数据科学重点实验室 (复旦大学), 上海 200433)

通信作者: 李弋, E-mail: liy@fudan.edu.cn

摘　要: SQL 是一种被广泛应用于操作关系数据库的编程语言, 很多用户 (如数据分析人员和初级程序员等) 由于缺少编程经验和 SQL 语法知识, 导致在编写 SQL 查询程序时会碰到各种困难. 当前, 使用程序合成方法根据<输入-输出>样例表自动生成相应的 SQL 查询程序, 吸引了越来越多人的关注. 所提 ISST (正反例归纳合成) 方法, 能够根据用户编辑的含有少量元组的<输入-输出>示例表自动合成满足用户期望的 SQL 查询程序. ISST 方法包括 5 个主要阶段: 构建 SQL 查询程序草图、扩展工作表数据、划分正反例集合、归纳谓词和验证排序. 在 PostgreSQL 在线数据库上验证 SQL 查询程序, 并依据奥卡姆剃刀原则对已合成的 SQL 查询程序候选集打分排序. 使用 Java 语言实现了 ISST 方法, 并在包含 28 条样例的测试集上进行验证, ISST 方法能正确合成其中的 24 条测试样例, 平均耗时 2 s.

关键词: SQL; 程序合成; PBE; 归纳; 正例; 反例

中图法分类号: TP311

中文引用格式: 张健, 李弋, 彭鑫, 赵文耘. 正反例归纳合成SQL查询程序. 软件学报, 2023, 34(9): 4132–4152. http://www.jos.org.cn/1000-9825/6646.htm

英文引用格式: Zhang J, Li Y, Peng X, Zhao WY. Inductive SQL Synthesis with Positive and Negative Tuples. Ruan Jian Xue Bao/Journal of Software, 2023, 34(9): 4132–4152 (in Chinese). http://www.jos.org.cn/1000-9825/6646.htm

Inductive SQL Synthesis with Positive and Negative Tuples

ZHANG Jian[1,2], LI Yi[1,2], PENG Xin[1,2], ZHAO Wen-Yun[1,2]

[1](Software School, Fudan University, Shanghai 200433, China)

[2](Shanghai Key Laboratory of Data Science (Fudan University), Shanghai 200433, China)

Abstract: SQL is a programming language that is widely used to operate relation databases. Many users (such as data analysts and junior programmers) will encounter various difficulties when writing SQL query programs due to the lack of programming experience and knowledge of SQL syntax. Currently, the research on the automatic synthesis of SQL query programs from the <input-output> (I/O) example tables has attracted more and more attention. The inductive SQL synthesis with positive and negative tuples (ISST) method proposed in this study can automatically synthesize SQL query programs that meet the users' expectations by the I/O example tables edited by users and containing a small number of tuples. The ISST method contains five main stages: constructing the SQL query program sketches, expanding the worksheet data, dividing the sets of positive and negative examples, inductively synthesizing selection predicates, and sorting after verifying. The candidate set of SQL query programs is verified on the online database PostgreSQL, and the candidate set of synthesized SQL query programs is scored and sorted according to the principle of Occam's razor. The ISST method is implemented using the Java language and then is evaluated on a test set containing 28 samples. The results reveal that the ISST method can correctly synthesize 24 of the samples, which takes an average of 2 seconds.

Key words: structured query language (SQL); program synthesis; program by example (PBE); induction; positive example; negative example

---

* 基金项目: 上海市科委项目 (19511132000)
收稿时间: 2021-08-04; 修改时间: 2021-12-10; 采用时间: 2022-01-18; jos 在线出版时间: 2023-01-04
CNKI 网络首发时间: 2023-01-05

1 引 言

SQL (structured query language) 是一种广泛用于关系数据库数据存取操作的领域程序语言, 长期位列 TIOBE 编程语言排行榜[1]前 10. 随着信息化时代的发展, 越来越多的业务人员参与到数据分析工作中, 他们需要频繁的查询数据库, 然而由于缺少编程经验以及 SQL 查询语法的灵活性, 用户在编写 SQL 查询时常常碰到很多困难, 在知名的 IT 问答平台 Stack Overflow[2]上, 我们使用关键字“SQL”搜索到了 56.5 万多条相关提问.

考虑下面的情况, 某公司需要了解员工劳务支出情况, HR 经理要根据年龄段、性别、级别、入职时间等各个维度统计查询每月员工的平均劳务支出, 如果公司财务系统没有提供这种复杂的统计功能, HR 经理要编写出正确的 SQL 查询语句是个很大的挑战. 在 Stack Overflow 上的 SQL 问题大多都是以<输入-输出>示例表对的形式描述问题, 虽然用户不知道如何编写正确的 SQL 查询, 但用户知晓 SQL 查询的意图, 可以通过编写含有少量元组的<输入-输出>示例表来表达 SQL 查询的意图. 在用户不熟悉编程语言的情况下, 可以通过编辑<输入-输出>样例规约来表达用户的期望, 程序合成器将自动合成出能成功将输入示例转化为输出示例的程序, 这种方法称为 PBE (program by example) 方法.

程序合成是一种智能化软件开发的技术, 它能够根据用户规约自动合成满足用户期望的程序代码, 最终目标是实现计算机根据需求自主编程, 用户规约的形式主要有自然语言描述规约、DSL (domain specified language) 逻辑规约和<输入-输出>样例规约. 程序合成主要有演绎程序合成 DPS (deductive program synthesis)、归纳程序合成 IPS (inductive program synthesis) 和基于深度学习概率预测的智能程序合成等途径[3]. IPS 是程序合成的重要方法, 已被成功应用于微软 Excel 2013 中的快速填充功能 (Flashfill[4]、FlashExtract[5]).

根据用户编辑的含有少量元组的<输入-输出>示例表对自动合成 SQL 查询是程序合成技术的一个重要应用场景. 对非专业编程人员有很大的帮助, 例如熟悉数据的数据分析人员和企业业务人员等, 通过构造 SQL 查询的输入输出示例, PBE 可以帮助不熟悉 SQL 语法的用户快速编写出正确的 SQL 查询程序. 对于专业编程人员同样也会有很大帮助, 虽然知悉 SQL 语法, 但是编写某些特定功能的 SQL 查询依然可能存在很大困难, 也同样可以使用基于 PBE 方法快速合成出正确的 SQL 查询程序.

最近几年, 有不少工作提出了一些自动合成 SQL 查询的方法[6−13]. 例如, SCYTHE[9,10]就具有很强的表达能力, 支持投影、连接、分组、聚合函数和合并等功能. 给定<输入-输出>示例表, 一些算法在 SQL 语法空间中枚举大量的候选语句, 再消去那些不可能的候选项[6,8,9,12,13]. 一些算法把合成问题转化为约束求解问题[11]或者是机器学习问题[7], 一些算法会基于 SQL 查询模板或者草图 (sketch) 来降低算法的搜索空间[7,9,11,12], 还有一些算法利用输出表中的元组来去掉不可行的候选[8,13], EGS[13]则利用输出表中的元组信息来构造关系. 在很多情况下, 用户只需要编写少量的元组, 就可以构造<输入-输出>示例表来表达用户查询的期望. 为了充分利用<输入-输出>示例表中数据元组的关系, 我们分析了一些典型 SQL 语句实例的执行, 观察输入表元组在关系代数作用下的变化, 把 SQL 查询的合成转化为从输出表到输入表的逆向工程问题. 我们进而提出了正反例归纳合成方法 ISST, 把工作表划分成正例元组集合和反例元组集合, 然后归纳出可能的谓词, 最后用输出表中的元组作为约束条件消除不可能的谓词.

我们实现了 ISST 方法, 并在典型的 SQL 查询合成测试上集进行验证, 实验结果表明, ISST 方法合成 SQL 查询的成功率为 85.7%, 每个测试用例的平均耗时约 2 s, 而且 ISST 方法合成的 SQL 查询具有可读性好、通用性和符合用户语义期望等特点.

2 相关工作

在过去的数十年中, 研究者提出了很多根据用户编辑的含有少量元组的<输入-输出>示例表对自动合成 SQL 查询的方法. Gulwani[4]提出了利用<输入-输出>示例对来归纳合成程序代码, 该归纳合成方法在微软产品 Microsoft Excel 2013 的快速填充 (“Flash Fill”) 功能得以实现, 并取得了很大的成功. SQLSynthesizer 方法[7]首先构建一个参数化的 SQL 模板, 然后使用启发式规则和训练 PART[14]决策树方法来合成投影列、谓词等参数, 由于该方法是使用所有元组 (没有经过 WHERE 子句过滤) 进行扩展数据的计算, 导致对同时含有 WHERE 子句和

GROUP-BY 子句的 SQL 查询合成场景支持效果较差. SCYTHE 方法[9,10]是一种枚举搜索和嵌套组合的合成方法, 使用抽象查询等价类的方法剪枝搜索空间, 适用于合成多层嵌套类型的 SQL 查询, 该方法合成的 SQL 查询的语义可解释性较差. SQLSOL 方法[11]是把 SQL 查询合成问题转换成 SMT 约束求解问题, 它首先建立参数化的 SQL 查询模板, 该 SQL 查询模板中包含了未知的投影列参数、聚合函数名参数和谓词参数等, 然后将<输入-输出>示例表的数据和 SQL 查询语法作为约束条件, 使用 Z3[15]求解器拟合出 SQL 查询模板中的未知参数. SQUARES[12]是一种基于程序枚举的合成方法, 包含两个部分: 枚举器和决策器. 枚举器使用 TRINITY[16]框架枚举合成各种可能的 SQL 查询程序, 决策器用于校验枚举合成的 SQL 查询是否正确, 同时枚举器基于等价策略剪枝搜索空间. QRE[6,17−23]聚焦于高效合成 SPJ (SELECT-PROJECT-JOIN) 样式的 SQL 查询程序合成, REGAL[8]研究了包含分组和谓词的单层简单 SQL 查询程序合成, 这些研究侧重于在用户不熟悉数据的前提下, 从业务系统中还原出简单样式的 SQL 查询程序. SQLizer[24]和 SyntaxSQLNet[25]方法是一种基于 NLP 处理技术的合成方法, 但由于自然语言描述固有的灵活性和歧义性, 常常导致合成的准确性不高. 基于 NLP 技术的合成方法需要大量的训练数据, 并且已训练的模型迁移到新的场景上也都存在着很大的挑战. EGS 方法[13]使用“元组共现模式”的方法指导程序空间搜索, 相比较于语法指导的程序空间搜索方法, EGS 方法使用常量共现图, 极大地减少了搜索空间, 提高合成效率.

3 预备知识

我们先简要介绍 SQL 语句的语法和语义特点, 由于不同数据库对 SQL 语法支持有细微的差异, 本文中以 PostgreSQL 数据库的 SQL 语法为标准进行说明.

3.1 SQL 语法

SQL 查询是由一系列灵活的子句组成, 包括: SELECT 子句、FROM 子句、WHERE 子句、GROUP-BY 子句和 HAVING 子句, 这些子句共同确定查询哪些数据, 如图 1 是典型的 SQL 查询样例.

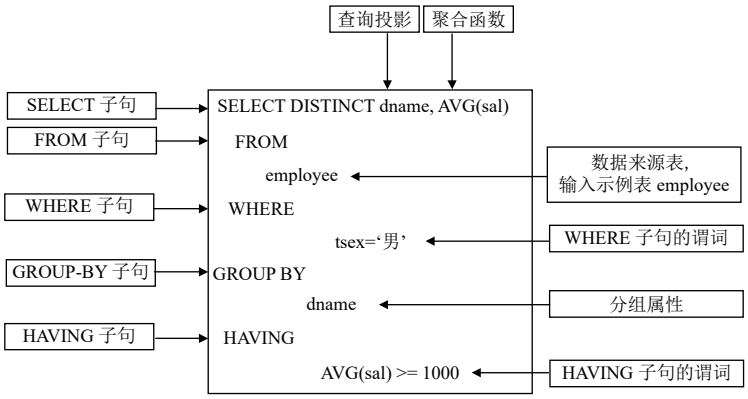


图 1 SQL 查询样例

● SELECT 子句指定了 SQL 查询的投影和聚合函数投影, 投影记为 $PROJ_{colus}$, 聚合函数记为 $PROJ_{aggr}$, 如在图 1 样例中 $PROJ_{colus}$={dname}, $PROJ_{aggr}$={AVG(sal)}, 我们目前支持 COUNT、SUM、MAX、MIN 和 AVG 共 5 种聚合函数, 函数名不区分大小写.

● FROM 子句指定了 SQL 查询的数据来源表.

● WHERE 子句指定了对工作表元组的过滤条件 (谓词), 记为 $PRED_{where}$, 如在图 1 样例中 $PRED_{where}$={tsex=‘男’}, 即选取 employee 表中 tsex 列的值为“男”的元组集合.

● GROUP-BY 子句指定了 SQL 查询的分组, 即根据{dname}列的值进行分组聚合计算.

● HAVING 子句指定了对分组统计的过滤条件 (谓词), 记为 $PRED_{having}$, 如在图 1 样例中 $PRED_{having}$={AVG(sal)>=1000}, 即选取分组统计的 AVG(sal) 值大于等于 1 000 的分组元组集合.

SQL 查询只有 SELECT 子句和 FROM 子句是必须的, 其他的子句都是可选的. 如图 2 展示了 ISST 方法目前已实现的 SQL 查询语法和类型. 根据 SQL 查询的执行过程关键特征, 我们把 SQL 查询分成 4 种类型.

```
<query>:: = <type 1> | <type 2> | <type 3> | <type 4>
<type 1>:: = SELECT <column>+ FROM <table>+ WHERE <pred>+
<type 2>:: = SELECT <aggr>+ FROM <table>+ WHERE <pred>+
<type 3>:: = SELECT <expr>+ FROM <table>+ GROUP BY <column>+ HAVING <pred>+
<type 4>:: = SELECT <expr>+ FROM <table>+ WHERE <pred>+ GROUP BY <column>+ HAVING <pred>+
<table>:: = <table_name> | <table_name> (inner | left | right) join <table_name> on <join>
<column>:: = <table_name>.column_name
<join>:: = <table_name1>.column_name1 = <table_name2>. column_name2
<expr>:: = <column> | <aggr>
<pred>:: = <pred> && <pred> || <column> <op> <const> || <column> <op> <query>
<aggr>:: = MIN(<column>) | MAX(<column>) | COUNT(<column>) | SUM(<column>) | AVG(<column>) | EVERY(<pred>)
<op>:: = >= | <= | = |in
```

图 2　部分 SQL 查询语法和类型

1) <type 1>类型的 SQL 查询程序, 包含 SELECT 子句、FROM 子句和 WHERE 子句, 它首先是使用 WHERE 子句的谓词对工作表元组进行选择过滤, 然后对属性投影得到 SQL 查询结果.

2) <type 2>类型的 SQL 查询程序, 与<type 1>包含的子句相同, 区别是<type 2>类型 SQL 查询的 SELECT 子句中仅含有聚合函数投影, 它是对满足 WHERE 子句谓词条件的元组集合进行聚合计算.

3) <type 3>类型的 SQL 查询程序, 包含 SELECT 子句、FROM 子句、GROUP-BY 子句和 HAVING 子句, 它首先将工作表中元组按照分组属性的值进行分组, 并对每个分组进行聚合计算, 然后根据 HAVING 子句的谓词对分组进行选择过滤, 最后执行查询投影得到 SQL 查询结果.

4) <type 4>类型的 SQL 查询程序, 包含 SELECT 子句、FROM 子句、WHERE 子句、GROUP-BY 子句和 HAVING 子句, 它首先是使用 WHERE 子句的谓词对工作表元组进行过滤, 其次对保留的元组集合按照分组属性的值进行分组, 并对每个分组进行聚合计算, 然后根据 HAVING 子句的谓词对分组进行选择过滤, 最后执行查询投影得到 SQL 查询结果.

3.2 SQL 语义

根据<输入-输出>示例表合成 SQL 查询, 可以看作是 SQL 查询执行过程的逆向工程. SQL 查询执行过程本质是关系代数作用在源数据上, 得到目标数据. 如图 3 所示, 我们给出了一个含有 6 个属性的“employee”数据表, 共有 12 条元组. 然后, 我们为每种类型的 SQL 查询构造了 1 个样例, 并简要说明处理过程. “output”数据表是 SQL 查询的执行结果.

(1) 查询 1 “select tname, age, dname, sal from employee where dname=‘研发部’”是<type 1>类型的 SQL 查询, 用户查询 “研发部雇员的姓名、部门、性别、工资等信息”. SQL 查询执行过程: 首先使用 WHERE 子句的谓词对“employee”表中所有元组进行选择过滤, 仅保留满足谓词“dname=‘研发部’”条件的元组, 然后执行查询投影 $\prod_{\text{tname, age, dname, sal}}$, 得到 SQL 查询目标数据.

(2) 查询 2 “select avg(sal) from employee where dname=‘研发部’”是<type 2>类型的 SQL 查询, 用户查询“研发部雇员的平均薪资”. SQL 查询执行过程: 首先使用 WHERE 子句的谓词对“employee”表中所有元组进行选择过滤, 仅保留满足谓词“dname=‘研发部’”条件的元组; 然后对属性“sal”执行 AVG 聚合函数计算, 得到 SQL 查询目标数据.

(3) 查询 3 “select dname, avg(sal) from employee group by dname having avg(sal) >= 1050”是<type 3>类型的 SQL 查询, 用户查询“雇员平均薪资大于等于 1050 的部门和平均薪资”. SQL 查询执行过程: 首先根据属性“dname”的值对元组进行分组; 其次, 对每个分组的属性“sal”执行 AVG 聚合函数计算, 并过滤掉不满足 HAVING 子句中谓词“avg(sal) >= 1050”条件的分组; 然后, 对满足条件的分组执行查询投影 $\prod_{\text{dname, age(sal)}}$, 得到 SQL 查询目标数据.

(4) 查询 4 “select dname, avg(sal) from employee where tsex=‘男’ group by dname having avg(sal) >= 1050”是<type 4>类型的 SQL 查询, 用户查询“男性雇员平均薪资大于等于 1 050 的部门和男性雇员平均薪资”. SQL 查询执行过程: 首先使用 WHERE 子句的谓词对“employee”表中所有元组进行选择过滤, 仅保留满足谓词“tsex=‘男’”条件的元组; 其次, 根据属性“dname”的值对元组集合进行分组; 然后, 对每个分组的属性“sal”执行 AVG 聚合函数计算, 并过滤掉不满足 HAVING 子句中谓词“avg(sal) >= 1050”条件的分组; 最后, 对满足条件的分组执行查询投影 $\prod_{\text{dname, age(sal)}}$, 得到 SQL 查询目标数据.

“employee”数据表是输入示例表, “output”数据表是输出示例表. 在上述这 4 类 SQL 查询中, 有的输出表是输入元组部分子集的直接投影, 有的输出表中包含一些计算后的元组, 例如聚合函数.

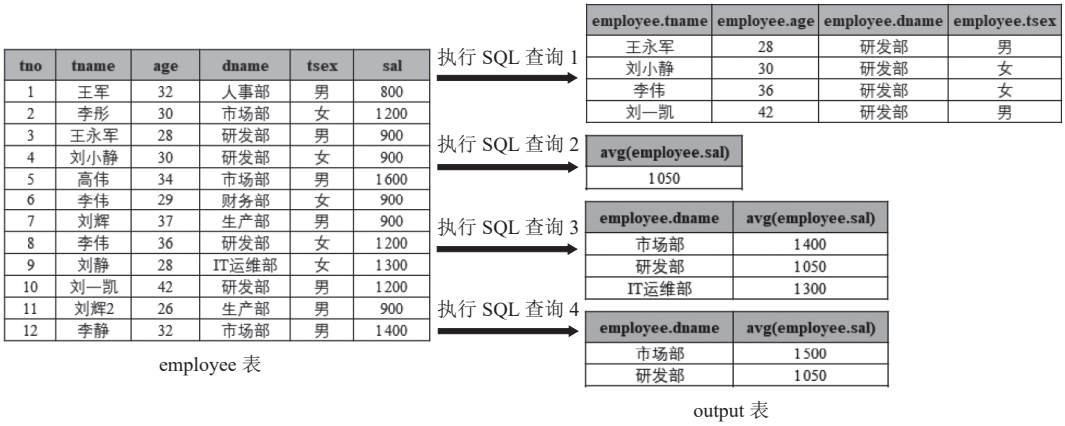


图 3 SQL 查询样例

4 问题和示例

SQL 是一种数据库操作和查询的描述性语言, 按照语法可以生成各种复杂多样的 SQL 查询语句, 这对于根据含有少量元组的<输入-输出>样例表合成 SQL 查询带来了很大的挑战.

4.1 问题和假设

用 PBE 方法合成 SQL 查询, 需要用户提供含有少量元组的<输入-输出>示例表对. 输入示例表是一张或多张数据表, 给出了所有的 SQL 查询源数据, 简称为 IE (input example) 表; 输出示例表是 SQL 查询的结果表, 简称为 OE (output example) 表. 如果一个 SQL 查询, 能够从 IE 表成功查询出 OE 表数据, 就是问题的一个可能的解, 这样的解可能有多个.

从 Stack Overflow 上的很多 SQL 相关提问可以看出, 提问者知晓数据的含义, 以及 SQL 查询的目标和意图, 能够准确使用含有少量数据的样例来描述问题. 但提问者不熟悉 SQL 语法, 或者虽然知悉 SQL 语法, 但是可能编写某些特定功能的 SQL 查询依然存在很大困难. 虽然有 SQL 语法规范, 但具体实现还是存在不同的差异, 也有各自的扩展. 用户除了构造含有少量元组的 IE 表和 OE 表, 用来表达 SQL 查询意图, 还可以基于用户对数据的了解提供一些辅助信息, 如查询投影信息、分组信息和多表连接条件信息, 这些辅助信息可以减少程序搜索空间, 降低合成的代价. 也可以像其他方法[7,11,12]一样, 使用启发式规则 (名称匹配和类型匹配) 和枚举方法自动识别这些辅助信息, 这不是我们的研究重点, 在本文中假设用户提供这 3 类辅助信息.

(1) 查询投影信息. 查询投影是 SELECT 子句的关键信息, 包括投影列和聚合函数, 是用户查询目标和意图的体现, 我们通过 OE 表和 IE 表中的列名称匹配方式指定 SQL 查询投影, OE 表的列名称采用“表名.列名”和“聚合函数 (表名.列名)”的方式定义了查询投影和聚合函数. 其中“表名.列名”指出了该查询投影的来源, “聚合函数 (表名.列名)”标记了聚合函数名称和列, 这里的聚合函数是作用于 SELECT 子句中查询投影的聚合函数, 是用户的查

询意图体现, 不是 HAVING 子句中谓词的聚合函数. 如图 3 中, “output”表的 “employee.dname”和“avg(employee.sal)”等属性是 SQL 查询投影和聚合函数. SELECT 子句中的聚合函数名称可以使用易于用户理解和记忆的别名, 在本文中使用标准的聚合函数名称.

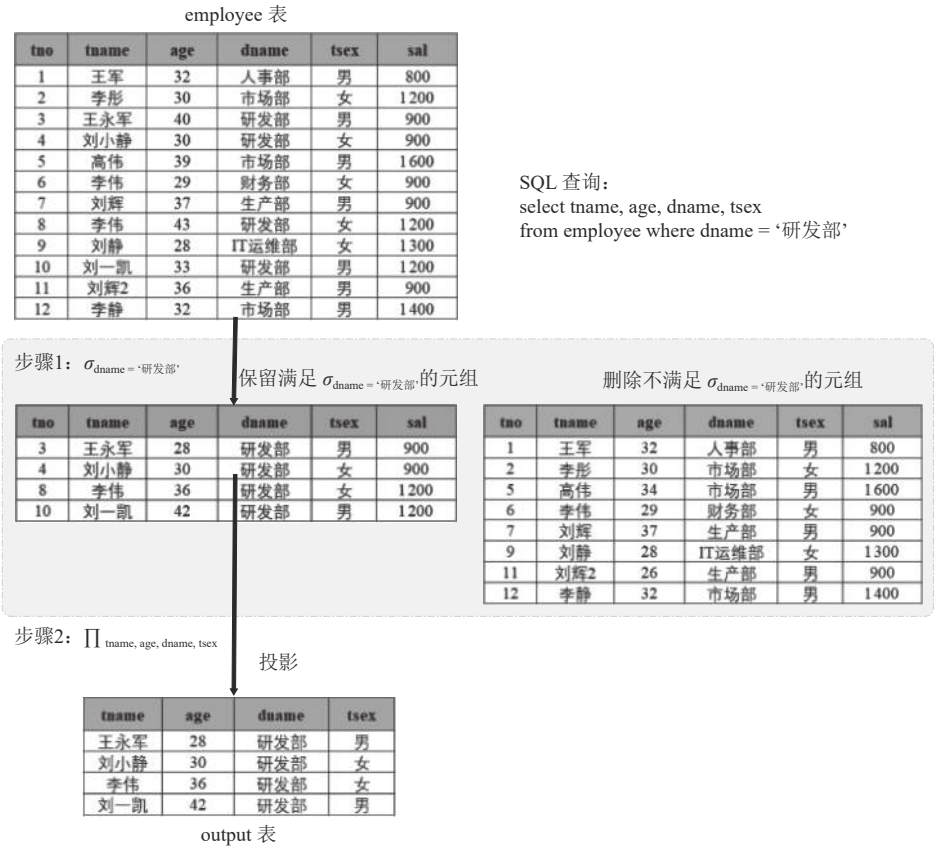
(2) 查询分组信息. 处理数据时, 数据分析员基于某些属性对数据进行分组, 但可能不清楚如何用 SQL 语言来进行分组描述并对数据做进一步的处理. 例如, “查询雇员平均薪资大于 1 000 的部门和平均薪资”“查询男性雇员平均薪资大于 1 050 的部门和男性雇员平均薪资”, 用户易于知道要根据“部门”属性进行分组.

(3) 多表连接条件信息. 多表连接条件是多张输入示例表的连接属性, 我们首先根据多表连接条件将多张数据表连接成一张包含多表数据的连接视图 (包括内连接视图、左外连接视图、右外连接视图), 然后分别基于每个连接视图合成 SQL 查询程序.

这些辅助信息和<输入-输出>示例表对, 我们统称为用户规约. 在这些规约信息的约束下, 结合 SQL 引擎对数据元组的代数计算原理, 我们提出了正反例归纳合成 SQL 查询的方法, 简称为 ISST (inductive SQL synthesis with positive and negative tuples) 方法.

4.2 正反例元组集合划分

SQL 语句的执行过程就是关系代数的运算, 图 4 给出了一个简单 SQL 查询程序的执行过程, 主要分为 2 个阶段: 1) 执行元组过滤操作“$\sigma_{\text{dname}='研发部'}$”, 该操作仅保留满足“dname =‘研发部’”条件的元组集合, 删除不满足该条件的元组集合; 2) 执行查询投影$\prod_{\text{tname, age, dname, tsex}}$, 对满足查询条件的元组集合进行查询投影, 得到 SQL 查询结果“output”表数据.



tno	tname	age	dname	tsex	sal
1	王军	32	人事部	男	800
2	李彤	30	市场部	女	1 200
3	王永军	40	研发部	男	900
4	刘小静	30	研发部	女	900
5	高伟	39	市场部	男	1 600
6	李伟	29	财务部	女	900
7	刘辉	37	生产部	男	900
8	李伟	43	研发部	女	1 200
9	刘静	28	IT运维部	女	1 300
10	刘一凯	33	研发部	男	1 200
11	刘辉2	36	生产部	男	900
12	李静	32	市场部	男	1 400

tno	tname	age	dname	tsex	sal
3	王永军	28	研发部	男	900
4	刘小静	30	研发部	女	900
8	李伟	36	研发部	女	1 200
10	刘一凯	42	研发部	男	1 200

tno	tname	age	dname	tsex	sal
1	王军	32	人事部	男	800
2	李彤	30	市场部	女	1 200
5	高伟	34	市场部	男	1 600
6	李伟	29	财务部	女	900
7	刘辉	37	生产部	男	900
9	刘静	28	IT运维部	女	1 300
11	刘辉2	26	生产部	男	900
12	李静	32	市场部	男	1 400

tname	age	dname	tsex
王永军	28	研发部	男
刘小静	30	研发部	女
李伟	36	研发部	女
刘一凯	42	研发部	男

图 4 SQL 查询的执行过程

从这个例子中可以看出, 执行元组过滤操作“$\sigma_{\text{dname}='研发部'}$”把工作表 (employee 表) 中所有元组划分为两个元组集合: 满足谓词的元组集合和不满足谓词的元组集合. 前者称为正例元组集合, 记为 PT (positive tuples) 集合; 后者称为反例元组集合, 记为 NT (negative tuples) 集合, 且满足 PT∩NT=Φ.

4.2.1 数据扩展

在复杂的 SQL 语句中, HAVING 子句的谓词跟一些聚合函数有关, 例如<type 3>和<type 4>类型的查询. 从用户规约原始数据仅能归纳出常量值的谓词 (如 employee.dname=‘计算机’), 无法归纳出 HAVING 子句的谓词 (如 avg(employee.sal)>=1000) 和包含子查询的谓词 (如 age >= (select avg(employee.age) from employee)). 受 SQLSynthesizer[7]的扩展数据方法启发, 我们对 IE 表的元组数据进行扩展, 以便能归纳含有子查询的谓词或 HAVING 子句的谓词.

本文中把扩展后的数据表称为扩展工作表 (简称为 EW 表), 在后续示例中, 我们基于图 4 中的“employee”表来讨论, 相应的扩展工作表记为 $\text{EW}_{\text{employee}}$. EW 表中来自 IE 表的普通属性记为 EWC_{com}, 扩展属性记为 EWC_{ext}.

(1) 分组聚合数据扩展: 首先, 根据分组属性值对工作表中的元组集合进行分组; 然后, 对每个分组的每个属性计算所有可能的聚合函数值, 数值类型的属性计算 MAX、MIN、AVG、SUM 和 COUNT 等 5 种聚合函数值, 字符串类型的属性计算 COUNT 聚合函数值. 例如, 图 5 是“employee”表根据“employee.dname”属性值分组后的分组聚合数据扩展表.

tno	tname	age	dname	tsex	sal	GROUP BY employee.dname									
						…	MAX(age)	MIN(age)	AVG(age)	…	COUNT(distinct age)	…	MAX(sal)	MIN(sal)	AVG(sal)	…
6	李伟	29	财务部	女	900	…	29	29	29	…	1	…	900	900	900	…
1	王军	32	人事部	男	800	…	32	32	32	…	1	…	800	800	800	…
9	刘静	28	IT运维部	女	1300	…	28	28	28	…	1	…	1300	1300	1300	…
3	王永军	28	研发部	男	900	…	42	28	34	…	4	…	1200	900	1050	…
4	刘小静	30	研发部	女	900	…										
8	李伟	36	研发部	女	1200	…										
10	刘一凯	42	研发部	男	1200	…										
5	高伟	34	市场部	男	1600	…	34	30	32	…	3	…	1600	1200	1400	…
12	李静	32	市场部	男	1400	…										
2	李彤	30	市场部	女	1200	…										
7	刘辉	37	生产部	男	900	…	37	26	31.5	…	2	…	900	900	900	…
11	刘辉2	26	生产部	男	900	…										

图 5 分组聚合数据扩展表

(2) 全表聚合数据扩展: 将工作表中所有元组作为同一个分组集合, 对每个属性计算所有可能的聚合函数值, 数值类型的属性计算 MAX、MIN、AVG、SUM 和 COUNT 等 5 种聚合函数值, 字符串类型的属性计算 COUNT 聚合函数值. 例如, 图 6 是“employee”表的全表聚合数据扩展表.

tno	tname	age	dname	tsex	sal	全表元组									
						…	MAX(age)	MIN(age)	AVG(age)	…	COUNT(distinct age)	…	MAX(sal)	MIN(sal)	AVG(sal)	…
6	李伟	29	财务部	女	900	…	42	26	32	…	9	…	1600	800	1100	…
1	王军	32	人事部	男	800											
9	刘静	28	IT运维部	女	1300											
3	王永军	28	研发部	男	900											
4	刘小静	30	研发部	女	900											
8	李伟	36	研发部	女	1200											
10	刘一凯	42	研发部	男	1200											
5	高伟	34	市场部	男	1600											
12	李静	32	市场部	男	1400											
2	李彤	30	市场部	女	1200											
7	刘辉	37	生产部	男	900											
11	刘辉2	26	生产部	男	900											

图 6 全表聚合数据扩展表

4.2.2 元组划分

我们将 EW 表划分为 PT 集合和 NT 集合, PT 集合和 NT 集合是两个不相交的元组集合, 其中 PT 集合包含了满足谓词条件的元组, NT 集合是被谓词条件过滤掉的元组. OE 表是对 PT 集合的部分属性或全部属性的查询投影.

当合成不同类型的 SQL 查询程序时, 对 EW 表要采用不同的划分方法. 如果 PT 和 NT 集合划分失败, 则判定不存在该类型的 SQL 查询, 中断该类型的程序合成.

(1) <type 1>类型 SQL 查询划分方法比较简单, 因为 OE 表是 EW 表的普通属性 EWC_{com} 的查询投影, 所以直接比较 OE 表元组与 EW 表元组的查询投影属性值即可找到 PT 和 NT 集合的划分情况, 图 7 是第 3.2 节中查询 1 的一种 PT 和 NT 集合划分情况. 如果 OE 表是 PT 集合的部分属性投影, 从 PT 集合中可能找到多条 OE 表元组的映射元组, 所以可能存在多种 PT 和 NT 集合的划分情况.

tno	tname	age	dname	tsex	sal	全表元组			
						…	MAX(age)	…	AVG(sal)
3	王永军	28	研发部	男	900	…	42	…	1100
4	刘小静	30	研发部	女	900				
8	李伟	36	研发部	女	1200				
10	刘一凯	42	研发部	男	1200				

PT 集合

tno	tname	age	dname	tsex	sal	全表元组				
						…	MAX(age)	…	AVG(sal)	…
6	李伟	29	财务部	女	900	…	42	…	1100	…
1	王军	32	人事部	男	800					
9	刘静	28	IT运维部	女	1300					
5	高伟	34	市场部	男	1600					
12	李静	32	市场部	男	1400					
2	李彤	30	市场部	女	1200					
7	刘辉	37	生产部	男	900					
11	刘辉2	26	生产部	男	900					

NT 集合

图 7　查询 1/2 的一种 PT 和 NT 集合划分

(2) <type 2>类型 SQL 查询特点是 SELECT 子句仅包含对某属性的聚合计算. 以第 3.2 节中查询 2 为例, 在 OE 表给出的信息中, 我们知道要对“sal”属性做 AVG 聚合计算, 其他的信息都没有. 因此, 我们需要根据“sal”属性值寻找符合要求的 PT/NT 集合划分, 即 AVG 聚合函数作用在 PT 集合“sal”属性上的值与 OE 表中的值一致. 在我们的实验中是采用格图 (Lattice graph) 方法[8]枚举不同数量的元组组合, 并结合启发式规则进行剪枝, 以快速寻找出所有正确的 PT 和 NT 集合划分. 图 7 是 Lattice graph 方法枚举出的一种划分情况, 验证可知左侧表中的“sal”属性 AVG 聚合函数值等于查询 2 的 OE 表中的值, 记为 PT 集合, 右侧表为对应的 NT 集合. 如果 EW 表元组数量过多, 可能会导致 Lattice graph 方法组合爆炸, 所幸用户手工编辑的 IE 表元组数量一般不会太多, 而且我们结合不同聚合函数的特点, 采用相应的启发式规则剪枝搜索空间, 排除掉不可能的元组组合情况, 减少搜索空间. 例如, 查询 2 是 AVG 聚合函数计算, 我们可以排除掉“sal”属性值全部大于或全部小于“1050”的元组组合.

(3) <type 3>类型 SQL 查询特点是查询投影包含分组属性和聚合函数属性, 无 WHERE 子句, 仅根据 HAVING 子句的谓词对分组进行选择过滤. 对于 EW 表, 我们可以忽略除分组属性之外的 EWC_{com} 属性, 并且去除分组值相同的元组. 与<type 1>类型的划分方法相同, 我们只需比较 OE 表元组与 EW 表元组的查询投影值即可找到 PT 和 NT 集合的划分情况. 例如, 查询 3 中的投影包含分组属性“dname”和属性“sal”的 AVG 聚合函数, 通过比较 EW 表和 OE 表中“dname”和“AVG(sal)”的属性值即可划分 PT 和 NT 集合. 图 8 给出了查询 3 的 PT 和 NT 集合的一种划分情况.

dname	全表元组					GROUP BY employee.dname				
	…	MAX(age)	…	AVG(sal)	…	…	MAX(age)	…	AVG(sal)	…
IT运维部	…	42	…	1100	…	…	28	…	1300	…
研发部						…	42	…	1050	…
市场部						…	34	…	1400	…

PT 集合

dname	全表元组					GROUP BY employee.dname				
	…	MAX(age)	…	AVG(sal)	…	…	MAX(age)	…	AVG(sal)	…
财务部	…	42	…	1100	…	…	29	…	900	…
人事部						…	32	…	800	…
生产部						…	37	…	900	…

NT 集合

图 8　查询 3 的一种 PT 和 NT 集合划分

(4) <type 4>类型 SQL 查询特点是先使用 WHERE 子句的谓词对工作表的元组进行过滤; 其次, 对元组分组, 并对每个分组进行聚合计算; 然后, 根据 HAVING 子句的谓词对分组进行过滤. 合成过程是<type 2>和<type 3>两个类型的组合, 我们使用查询 4 样例进行说明. 我们先分析 SQL 执行过程. 首先, 执行 WHERE 子句的元组过滤操作“$\sigma_{tsex='男'}$”, 排除掉不满足该谓词条件的元组; 其次, 根据“dname” 属性值对数据进行分组, 对每个分组进行聚合计算; 然后, 执行 HAVING 子句的操作“avg(employee.sal) >= 1050”, 排除掉不满足该条件的分组; 最后, 对属性

“dname”和“avg(sal)”投影得到“output”表数据.

基于 SQL 语句执行过程中的代数运算, 我们可以分为下列 3 步来构造查询 4 的 PT 和 NT 集合的划分.

1) 根据“output”表的分组属性值 (“市场部”和“研发部”), 我们先对 $EW_{employee}$ 表进行初步划分, 从 $EW_{employee}$ 表分别找出属于每个分组的元组集合, 那么 $EW_{employee}$ 表的 PT 集合是 $PT_{市场部}$和 $PT_{研发部}$两个集合的并集, 它们分别和“output”表中对应的分组元组构成<IE-OE>对, 如图 9 所示, 易于知道这两个新的<IE-OE>对是<type 2>类型的 SQL 查询合成问题.

2) 对于图 9 中的“市场部”和“研发部”两个分组的<IE-OE>对, 因为分组属性“dname”在 IE 和 OE 表中完全相同, 所以可以忽略这个属性, 那么问题将转化为<type 2>类型的元组划分和谓词归纳问题, 此处不再赘述. “市场部”分组的<$IE_{‘市场部’}$ - $OE_{‘市场部’}$>合成问题划分 PT 和 NT 集合后归纳出谓词为 $PRED_{‘市场部’}$={‘男’}, “研发部”分组的<$IE_{‘研发部’}$ - $OE_{‘研发部’}$>合成问题划分 PT 和 NT 集合后归纳出谓词为 $PRED_{‘研发部’}$={‘男’}, 然后合并所有分组的谓词, 合并方法是谓词条件的值区间取并集, $PRED_{‘市场部’, ‘研发部’}$={‘男’}, 可知该合并后的谓词就是待合成的 SQL 查询的 WHERE 子句的谓词.

3) 使用 2) 合成的 WHERE 子句的谓词 $PRED_{‘市场部’, ‘研发部’}$={‘男’}, 对 $EW_{employee}$ 表元组进行过滤, 并记为 $EW_{after\text{-}where}$, 可知<$EW_{after\text{-}where}$ - output>构成了一个<type 3>类型的 SQL 查询合成问题, PT 和 NT 集合划分方法同上.

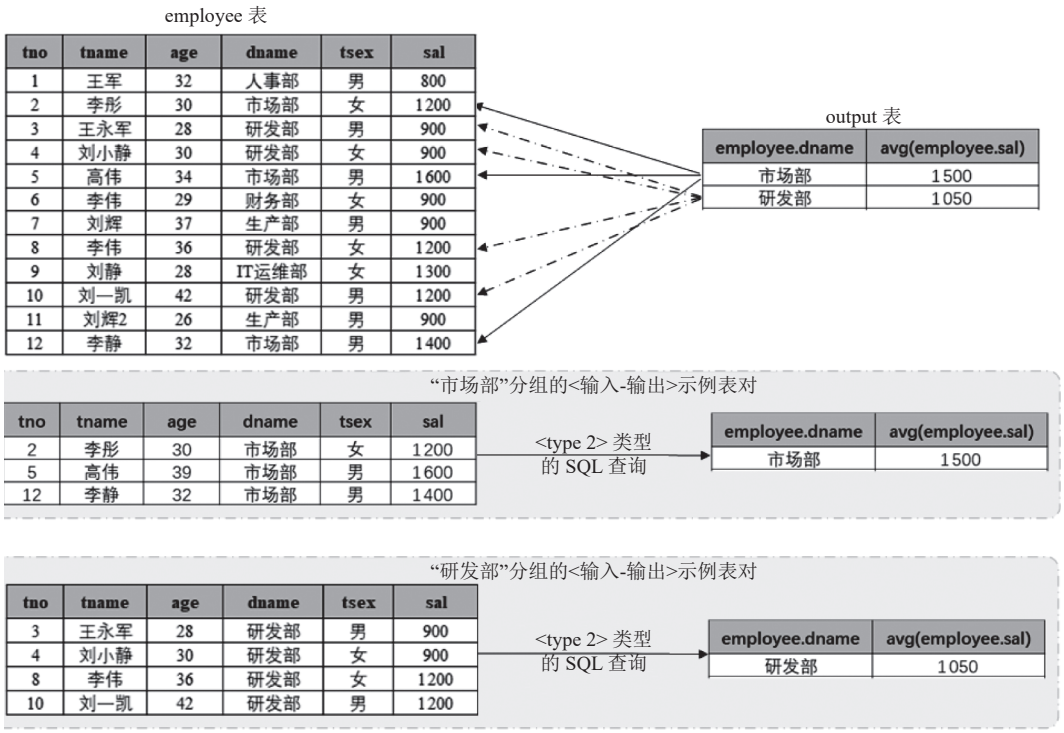


图 9 “市场部”分组和“研发部”分组的<输入-输出>示例表对

4.3 谓词归纳

将 EW 表划分成 PT 和 NT 集合是归纳谓词的前提条件. 标准 SQL 语法定义了多种数据类型, 本论文中仅考虑典型的字符串类型和数值类型, 其他的类型可以类似处理. 如果属性类型为字符串, 我们构造集合 U_p, 是 PT 集中该属性列的所有值集合, 重复值仅保留一个; 同样, 集合 U_n 表示 NT 集合中该属性列的所有值, 重复值仅保留一个. 如果属性为数值类型, 我们构造闭区间 D_p, 覆盖 PT 集合该属性列出现的数值; 集合 V_n 表示 NT 集合中该属性

列出现的所有数值集合, 重复值仅保留一个.

(1) 图 7 是查询 1 样例的一种正反例集合划分情况, 属性“dname”是字符串类型, 那么 $U_{p,\text{dname}}$ = {‘研发部’}, $U_{n,\text{dname}}$ = {‘人事部’, ‘生产部’, ‘市场部’, ‘财务部’, ‘IT 运维部’}, 可知 $U_{p,\text{dname}} \cap U_{n,\text{dname}} = \Phi$, 验证可知该谓词{dname=‘研发部’}可以成功从 $\text{EW}_{\text{employee}}$ 表中选择出 PT 集合, 排除掉 NT 集合.

(2) 图 8 是查询 3 样例的一种正反例集合划分情况, 属性“AVG(sal)”是数值类型, 那么 $D_{p,\text{AVG(sal)}}$ = [1050, 1400], $V_{n,\text{AVG(sal)}}$ = {900, 800}, 可知集合 $V_{n,\text{AVG(sal)}}$ 中的所有数值都不在 $D_{p,\text{AVG(sal)}}$ 区间范围内. 验证可知该谓词{AVG(sal) >= 1050}可以从 $\text{EW}_{\text{employee}}$ 表中选择出 PT 集合, 排除掉 NT 集合. 由 SQL 语法可知, 含有聚合函数的谓词应该放在 HAVING 子句中.

5 ISST 方法合成 SQL 查询程序

在第 4 节, 给定 IE 和 OE 表, 我们通过实例揭示了合成 SQL 查询程序要解决的主要问题和一些特定方法. 在这一节, 我们将讨论符合<type 1–4>类型 SQL 查询的合成方法 ISST.

如图 10 所示, 给定<输入-输出>示例表和约束常量, ISST 方法合成 SQL 查询程序的整体流程主要包括 5 个阶段: 构建 SQL 查询程序草图、扩展工作表数据、划分正反例集合、归纳谓词和在线验证排序.

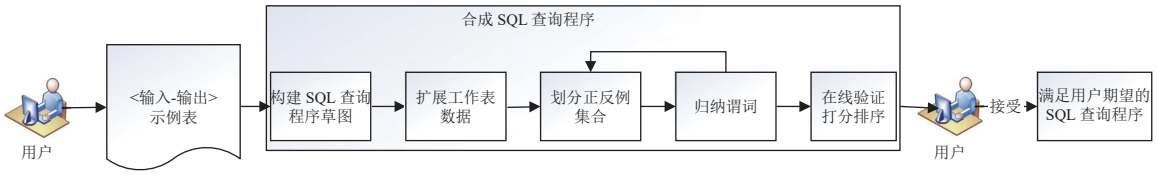


图 10 ISST 方法整体流程图

5.1 构建 SQL 查询草图

SQL 是一种强语法约束的编程语言, 每种类型的 SQL 查询具有严格的语法格式. 由于本文仅处理几种常见的 SQL 查询, 我们可以基于模板来进一步减少搜索空间, 表 1 给出本文关注的 4 种类型 SQL 查询程序草图. 我们对用户规约进行数据分析, 根据启发式规则, 排除掉不可能类型的 SQL 查询.

表 1 SQL 查询程序草图

类型	SQL查询
<type 1>	**SELECT** <column>$^+$ **FROM** <table>$^+$ **WHERE** <pred>$^+$
<type 2>	**SELECT** <aggr>$^+$ **FROM** <table>$^+$ **WHERE** <pred>$^+$
<type 3>	**SELECT** <expr>$^+$ **FROM** <table>$^+$ **GROUP BY** <column>$^+$ **HAVING** <pred>$^+$
<type 4>	**SELECT** <expr>$^+$ **FROM** <table>$^+$ **WHERE** <cond>$^+$ **GROUP BY** <column>$^+$ **HAVING** <pred>$^+$

5.1.1 查询分类

ISST 方法首先分析用户规约, 主要用到<IE-OE>示例表、聚合函数和分组属性等信息. 我们先给出不同类型 SQL 查询的必要条件.

1) <type 1>类型 SQL 查询: OE 表仅含有查询投影, 没有聚合函数, 没有分组属性.

2) <type 2>类型 SQL 查询: OE 表仅含有一个值, 且为聚合函数, 没有分组属性.

3) <type 3>或者<type 4>类型 SQL 查询: OE 表中的查询投影为分组属性和聚合函数.

PENIS 根据必要条件来启发式判定查询类型.

1) 如果用户规约中已指定分组属性, 并且 OE 表中同时包含查询投影和聚合函数, 那么排除掉<type 1>和<type 2>类型的 SQL 查询.

2) 如果用户规约中没有指定分组属性, 且 OE 表中仅含有查询投影、没有聚合函数, 那么排除掉<type 2>, <type 3>和<type 4>类型的 SQL 查询.

3) 如果用户规约中没有指定分组属性, 且 OE 表中仅含有一个聚合函数属性, 那么排除掉<type 1>, <type 3>和<type 4>类型的 SQL 查询.

基于这些启发式的规则, ISST 方法排除掉不可能类型的 SQL 查询.

5.1.2 关键元素推理

确定 SQL 语句可能的所属类型后, 我们需要尽可能地把缺失地信息补充出来, 以减少搜索空间. 基于属性名称匹配规则, 我们从用户规约中推理出 SQL 查询的关键元素: 数据表、查询投影、分组属性和多表连接条件等关键元素, 并补全到 SQL 查询程序草图. 如果用户规约包含多张输入示例表, 我们用 SQLSynthesizer、SQLSOL 等论文中相同的处理方法, 根据连接条件将多张输入示例表连接生成一个包含多表数据的连接视图, 包括内连接视图、左外连接视图和右外连接视图.

5.2 扩展工作表数据

为了更好地构造查询谓词, 我们需要构造一个包含更多信息的数据表. 我们把 IE 表中所有元组作为一个分组, 进行聚合数据扩展; 另外, 如果用户规约中已指定分组属性, 那么我们将根据该分组属性对 IE 表分组后再进行聚合数据扩展. 我们把扩展后的工作表记为 EW 表, 其中来自 IE 表的普通属性记为 EWC_{com}, 扩展属性记为 EWC_{ext}.

(1) 分组聚合数据扩展. 首先, 根据分组属性值对工作表中的元组集合进行分组; 然后, 对每个分组的每个属性计算所有可能的聚合函数值, 如数值类型的属性计算 MAX、MIN、AVG、SUM 和 COUNT 等 5 种聚合函数值, 字符串类型的属性计算 COUNT 聚合函数值.

(2) 全表聚合数据扩展. 将工作表中所有元组作为同一个分组集合, 对每个属性计算所有可能的聚合函数值, 如数值类型的属性计算 MAX、MIN、AVG、SUM 和 COUNT 等 5 种聚合函数值, 字符串类型的属性计算 COUNT 聚合函数值.

5.3 划分 PT 和 NT 集合

SQL 查询能否准确地表达用户的意图, 取决于谓词的准确性. 在扩展工作表数据阶段构造了 EW 表, 接下来我们将设法把 EW 表中所有元组划分为 PT 和 NT 集合, 且 $PT \cap NT=\Phi$. 由于 OE 表是对 PT 集合的部分属性或全部属性的查询投影, 所以我们的划分方法是把 EW 表中可以投影成 OE 表数据的元组集合划分为 PT 集合, 其余划分为 NT 集合. 如果 PT 和 NT 集合划分失败, 则判定不存在该类型的 SQL 查询, 中断该类型的程序合成.

从第 4.2 节查询样例可知, 对于不同类型 SQL 查询, 划分 PT 和 NT 集合的方法不同. 在构建 SQL 查询草图阶段, ISST 方法根据 IE/OE 表和辅助信息识别可能的 SQL 查询类型.

(1) <type 1>类型 SQL 查询: OE 表是 EW 表中普通属性 EWC_{com} 的查询投影, 所以直接比较 OE 表元组与 EW 表元组的查询投影属性值即可找到 PT 和 NT 集合的划分情况, 即根据 OE 表中的元组值从 EW 表中选择对应的某一个元组, 这些元组构成了 PT 集合, 其余元组构成了 NT 集合. 由于 OE 表是 PT 集合的部分属性投影, OE 表中的元组值可能对应了 EW 表中多个元组, 所以可能存在多种 PT 和 NT 集合划分.

(2) <type 2>类型 SQL 查询: 从 EW 表中枚举不同数量的元组集合, 对该元组集合进行聚合计算, 如果 $PROJ_{aggr}$ 聚合函数值与 OE 表中的值一致, 那么该元组集合就是 PT 集合, 其余的元组构成 NT 集合. ISST 方法是采用格图 (Lattice graph) 方法[8]枚举不同数量的元组组合, 并结合启发式规则进行剪枝, 组合复杂度是 $O(2^n)$, n 是元组的个数. 如果 IE 表中的元组数量过多, 可能会导致组合爆炸. 所幸用户手工编辑的 IE 表元组数量一般情况不会太多, 而且我们结合不同聚合函数的特点, 采用相应的启发式规则剪枝搜索空间, 排除掉不可能的元组组合情况, 降低搜索空间.

(3) <type 3>类型 SQL 查询: 与<type 1>类型的划分方法相同, 通过比较 OE 表与 EW 表元组属性值来划分 PT 和 NT 集合. OE 表中的查询投影为分组属性和属性聚合函数, 记为 $PROJ = \{PROJ_{colus}, PROJ_{aggr}\}$. 首先, 根据<type

3>类型 SQL 查询的特点, 我们可以忽略除分组属性之外的 EWC_{com} 属性, 并且去除分组值相同的元组, 记为 EW′表; 然后, 与<type 1>类型的划分方法相同, 我们只需比较 OE 表元组与 EW′表元组的 PROJ 属性值即可找出 PT 和 NT 集合的划分.

(4) <type 4>类型 SQL 查询: 该类型的 SQL 查询合成过程是<type 2>和<type 3>类型的组合, PT/NT 的划分比较复杂, 基于 SQL 查询执行过程的代数运算, 分为两个主要阶段.

1) WHERE 阶段: 首先, 不妨假设 OE 表中的元组为{ t_1, t_2, …, t_j }, 对于其中任一个元组 t_i, 根据分组属性值, 我们把 EW 表中对应的元组提取出来, 构成 EW_{ti} 表; 其次, 每个分组的<EW_{ti}-t_i>构成了一个<type 2>类型的 SQL 查询合成问题, 按照<type 2>类型的处理方式, 分别求出每个分组的 PT_i 和 NT_i 划分; 然后, 归纳出每个分组的谓词, 并合所有分组的谓词, 得到谓词 $PRED_{where}$. 由 SQL 语法可知, 该谓词应填充到 WHERE 子句. 谓词的合并原则是相同属性谓词的值区间取并集.

2) HAVING 阶段: 使用 1) 中合成的 WHERE 子句的谓词 $PRED_{where}$ 对 EW 表中元组进行选择过滤, 排除掉不满足该谓词的元组, 并记为 $EW_{after\text{-}where}$, 可知<$EW_{after\text{-}where}$-OE>构成了一个<type 3>类型的 SQL 查询合成问题, PT 和 NT 集合划分方法同上.

5.4 归纳谓词

SQL 查询执行, 根据谓词排除掉不满足条件的元组, 仅保留满足条件的元组. 构造谓词的候选属性, 既可能是 EW 表的普通属性 EWC_{com}, 也可能是扩展属性 EWC_{ext}. 将 EW 表划分成 PT 和 NT 集合是归纳谓词的前提条件.

5.4.1 单个属性谓词的构造

标准 SQL 语法定义了多种数据类型, 本论文中仅考虑典型的字符串类型和数值类型, 其他的类型可以类似处理. 字符类型和数值类型是常见的数据类型, 本文中字符串类型属性的谓词比较关系支持“=”和“in”, 数值类型属性的谓词比较关系支持“>=”“=”和“<=”.

(1) 字符串类型: 从 PT 集合中汇总该属性的所有字符串值集合 (重复值仅保留一个), 记为 U_p; 从 NT 集合中汇总该属性的所有字符串值集合 (重复值保留一个), 记为 U_n. 如果 $U_p \cap U_n = \Phi$, 那么该属性能够归纳出合法的谓词, 即将 U_p 作为谓词的比较值范围可以排除掉 U_n 中所有字符串; 如果 $U_p \cap U_n \neq \Phi$, 那么该属性不能归纳出合法的谓词, 即将 U_p 作为谓词的比较值范围不能排除掉 U_n 中所有字符串.

(2) 数值类型: 从 PT 集合中归纳该属性的数值区间, 记为 D_p; 从 NT 集合中汇总该列的数值集合, 记为 V_n. 如果 V_n 集合中的所有数值都不在 D_p 区间范围内, 那么根据该列能够归纳出合法的谓词, 即将 D_p 作为谓词的比较值范围可以排除掉 V_n 集合中的所有数值; 如果 V_n 集合中有数值在 D_p 区间范围内, 那么该列不能归纳出合法的谓词 (暂不考虑分段区间), 即将 D_p 作为谓词的数值范围不能排除掉 V_n 中所有数值.

5.4.2 多个属性谓词的构造

基于 EW 表的普通属性和扩展属性, 可以构造出多属性组合谓词. 一般而言, 我们认为构造谓词的属性越少, 即谓词组合越简单, 越能满足用户的期望.

我们约束 WHERE 子句和 HAVING 子句中分别最多含有 3 个谓词, 既可以防止由于 EW 表中属性过多而导致的属性组合爆炸, 也可以避免合成谓词过多的不必要 SQL 查询. ISST 方法采用格图 (Lattice graph) 方法[8]枚举不同数量的属性组合, 依次递增使用数量为 1、2、3 的候选属性集, 尝试是否能够归纳出合法的谓词. 如图 11 是候选属性集{a_1, a_2, a_3, a_4}的格图, 每个格都是一种可能的属性组合. 从 PT 集合中逐一归纳出每个属性的值区间, 验证该值区间的组合是否能够从 EW 表中排除掉 NT 集合中的所有元组. 目前 ISST 方法只考虑合取关系 (“与”关系) 连接的多谓词组合, 我们在将来的工作中继续实现析取关系 (“或”关系) 连接的多谓词合成.

5.4.3 子查询的构造

在构造谓词时, 如果属性 A 的值依赖于属性 B 的值, 而属性 B 可以通过简单的谓词归纳, 则可以通过构造子查询来表达 A 的选择. 我们可以借助于 EW 表来归纳含有简单子查询的谓词, 归纳方法与第 5.4.1 节中的归纳方法相类似. 从 PT 集合中归纳该列的数值区间, 记为 D_p; 从 NT 集合中归纳汇总该列的数值集合, 记为集合 V_n, 如果

能从 EW 表的扩展属性值中找到划分 D_p 和 V_n 的值, 那么产生该扩展属性值的 SQL 查询即为该属性的子查询. 另外, 我们还实现了 IN、EXISTS、INTERSECT 等关键词连接的子查询.

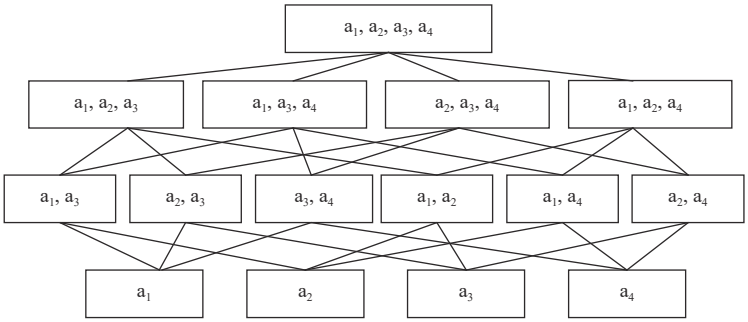


图 11 属性集{a_1, a_2, a_3, a_4}的格图

5.5 在线验证和打分排序

ISST 方法使用在线 PostgreSQL 数据库, 验证合成的 SQL 查询程序是否能从 IE 表中成功查询出 OE 表数据, 以确认 SQL 查询程序的正确性.

ISST 方法将归纳出多条等价的 SQL 查询程序, 一般来说, 语法结构简单的 SQL 查询可读性好、易于被用户采纳. 我们采用奥卡姆剃刀原则, 对多条 SQL 查询程序进行打分排序, 语法结构越简单排序越靠前, 以便于用户挑选最符合语义期望的 SQL 查询程序. 另外, 我们标记了每条 SQL 查询的执行耗时, 作为用户选择 SQL 查询时的参考因素.

5.6 SQL 查询的合成算法

我们介绍了 ISST 方法在不同阶段的任务和方法, 算法 1 给出了完整的 SQL 查询合成算法. 算法的输入包括输入示例表、输出示例表和辅助信息, 输出是 SQL 查询候选集.

ISST 方法先获取分组属性和多表连接条件 (代码第 3, 4 行), 并根据多表连接条件把多张输入示例表连接生成一个包含全部属性数据的视图 (代码第 5 行). 代码第 7, 8 行是构建 SQL 查询程序草图和扩展工作表数据.

根据 SQL 查询程序草图的类型来构造 SQL 查询 (代码第 9–26 行), 分两种情况进行:

1) 合成<type 1/2/3>类型的 SQL 查询 (代码第 9–15 行). 首先根据 OE 表的元组信息把 EW 表划分为 PT 和 NT 集合 (代码第 10 行), 可能存在多种划分; 然后根据 PT 和 NT 集合归纳谓词 (代码第 11 行); 最后生成完整的 SQL 查询并存入候选集 (代码第 13 行).

2) 合成<type 4>类型的 SQL 查询 (代码第 16–26 行), 分为两个阶段. 首先, 归纳合成 WHERE 子句的谓词 $\text{PRED}_{\text{where}}$ (代码第 17 行), 具体实现参见算法 2; 其次, 将 $\text{PRED}_{\text{where}}$ 作用在 EW 表上, 查询生成 $\text{EW}_{\text{after-where}}$ 表 (代码第 19 行), 构造面向 HAVING 的 $\text{PT}_{\text{after-where}}$ 和 $\text{NT}_{\text{after-where}}$ 集合划分 (代码第 20 行); 然后, 根据 $\text{PT}_{\text{after-where}}$ 和 $\text{NT}_{\text{after-where}}$ 集合归纳出 HAVING 子句的谓词 $\text{PRED}_{\text{having}}$ (代码第 19 行); 最后, 生成完整的 SQL 查询并存入候选集中 (代码第 23 行).

最后, 验证 SQL 查询的功能正确性, 并根据简洁性和通用性进行排序 (代码第 28 行). 语法结构越简单、通用性越好的 SQL 查询排序越靠前, 易于用户选取更符合语义期望的 SQL 查询.

算法 2 是合成<type 4>类型 SQL 查询的谓词 $\text{PRED}_{\text{where}}$, 即算法 1 中 INDUWHEREPREDSFORTYPE4 的方法实现. 算法输入为扩展工作表数据、输出示例表和辅助信息. 首先, 从输出示例表中获取分组属性值集合$\{t_i \mid 1 \leqslant i \leqslant n$, 其中 n 为输出示例表元组数量$\}$; 其次, 分别从扩展工作表和输出示例表中, 根据分组属性值 t_i 查询出对应的元组集合 IE_{ti} 和 OE_{ti} (代码第 5, 6 行), 形成<IE_{ti}-OE_{ti}>示例表对, 为<type 2>类型的 SQL 查询合成问题; 然后, 为每个分组<IE_{ti}-OE_{ti}>划分 PT_{ti} 和 NT_{ti} 集合 (代码第 7 行), 并从 PT_{ti} 和 NT_{ti} 集合归纳出该分组的谓词 PRED_{ti} (代码第

8 行); 最后, 合并分组的谓词, 以便能得到满足所有 PT_{ti} 和 NT_{ti} 划分的谓词 $PRED_{where}$. 合并原则是相同属性的谓词值区间取并集. 例如, 字符类型属性, $U = U_{ti1} \cup \ldots \cup U_{tin}$, n 为分组个数; 数值类型属性: $D = D_{ti1} \cup \ldots \cup D_{tin}$, n 为分组个数.

算法 1. SQL 查询程序合成.

输入: 输入示例表 *inputTables*, 输出示例表 *outputTable*, 用户定义约束常量 *userConfig*;
输出: SQL 查询集合.

```
1. function lNDUSYNTSQLQUERYBYPBE(inputTables, outputTable, userConfig)
2.   candidates ← ∅
3.   groupColuNames ← userConfig.groupColuNames
4.   joinColuNamePairs ← userConfig.joinColuNamePairs
5.   joinViews ← CREATEJOINVIEWS(inputTables, joinColuNamePairs)
6.   for each view ∈ joinViews do
7.     sqlSketches ← BUILDSQLQUERYSKETCHES(view, outputTable)
8.     extendDataView ← EXTENDINGVIEWCOLUDATA(view, groupColuNames)
9.     if ISSUITABLEFORQUERY123(extendDataView, outputTable, userConfig) then
10.      pt, nt ← SPLITEW(extendDataView, outputTable)
11.      predList ← INDUSQLQUERYPREDICATES(pt, nt)
12.      for each pred ∈ predList do
13.        candidates ← ASSEMBLESQLQUERY(sqlSketches, pred)
14.      end for
15.    end if
16.    if ISSUITABLEFORQUERY4(extendDataView, outputTable, userConfig) then
17.      predListForWhere ← INDUWHEREPREDSFORQUERY4(extendDataView, outputTable, userConfig)
18.      for each predForWhere ∈ predListForWhere do
19.        viewAfterWherePred ← GENEVIEWAFTERWHEREPRED(extendDataView, predForWhere)
20.        pt, nt ← SPLITIE(viewAfterWherePred, outputTable)
21.        predListForHaving ← INDUSQLQUERYPREDICATES(pt, nt)
22.        for each predForHaving ∈ predListForHaving do
23.          candidates ← ASSEMBLESQLQUERY(sqlSketches, predForWhere, predForHaving)
24.        end for
25.      end for
26.    end for
27.  end for
28.  RANK(candidates)
29.  return candidates
30. end function
```

算法 2. 归纳合成<type 4>类型 SQL 查询的 WHERE 子句的选择谓词.

输入: 扩展工作表 *extendDataView* , 输出示例表 *outputTable* , 用户定义约束常量 *userConfig* ;
输出: SQL 查询的选择谓词.

```
1. function INDUWHEREPREDSFORQUERY4(extendDataView, outputTable, userConfig)
2.    predList ← ∅
3.    groupValueList←GETGROUPVALUELIST(outputTable, userConfig)
4.    for each groupValue ∈ groupValueList do
5.       input ← FETCHTUPLESINGROUP(extendDataView, gourpColuNameList, groupValue)
6.       output ← FETCHTUPLESINGROUP(outputTable, gourpColuNameList, groupValue)
7.       pt, nt ← SPLITEW(input, output)
8.       predsOneGroup ← INDUSQLQUERYPREDS(pt, nt, userConfig)
9.       preds ← MERGEPREDS(preds, predsOneGroup)
10.   end for
11.   return predList
12. end function
```

6 实验和评估

验证环境机器的关键配置信息: Intel Core i5 2.5 GHz CPU, 8 GB 内存.

6.1 工具实现说明

我们从网站上下载了 SQLSynthesizer[7]作者开源的工具代码实现, 在此基础上使用 Java 语言实现 ISST 方法. 我们使用 PostgreSQL 在线数据库来验证合成的 SQL 查询的正确性, 并输出前 10 条正确的等价 SQL 查询. 我们限制 WHERE 子句和 HAVING 子句分别最多含有 3 个谓词.

6.2 合成成功判断标准

除了功能正确, SQL 查询的可读性、效率 (耗时)、通用性、在候选集中排序和语义满足用户期望等因素也是判断合成成功的重要标准. 虽然 SQL 查询的功能正确, 但不满足用户的语义期望、或可读性差、或不具有通用性或在候选集中排序靠后等因素, 往往导致在实际使用中不能被用户选中而认为是合成失败. 我们判定测试样例的 SQL 查询合成成功的标准包括:

(1) 功能正确性: 合成的 SQL 查询能成功从 IE 表中查询出 OE 表数据.

(2) 效率 (耗时): 我们设置每个用例的最大合成时间为 60 s, 超时则判定合成失败.

(3) 排序: 我们根据 Occam's razor 简单有效原则对候选集排序, 如果期望的 SQL 查询在候选集中排序在前 10 位之外, 则判定合成失败.

排序规则如下.

1) 相同类型的 SQL 查询, 谓词数量少的 SQL 查询更简单.

2) <type 1>, <type 2>和<type 3>类型的 SQL 查询比<type 4>类型的 SQL 查询更简单.

3) 谓词的比较关系符号, "="比较关系比"in"比较关系更简单.

4) WHERE 子句中的谓词比 HAVING 子句中的谓词更简单.

5) 多表连接情况下, 多表内连接的 SQL 查询比多表外连接的 SQL 查询简单.

6) 相同类型、相同数量谓词的 SQL 查询, 按字符比较优先级排序.

(4) 可读性: 我们认为过度复杂或深层无语义嵌套的 SQL 查询在实际应用中不会被用户优先选择, 如图 11 所示的 SCYTHE 方法合成的 SQL 查询样例.

(5) 通用性: 典型代表如表 2 中根据关键属性值的定向元组查询, 这种 SQL 查询不具有通用性, 因为一旦迁移到新的数据内容、同样需求的场景下, 该 SQL 查询获取不到用户期望的查询结果. 我们要求 SQL 查询不仅在当前<输入-输出>样例上功能正确, 而且当迁移到新的数据内容、同样需求的场景下, 该 SQL 查询依然能查询出用

户期望的结果.

(6) 满足用户的语义期望: 这点主要通过人工判断, 候选集中包含多条功能正确的 SQL 查询, 只有真正满足用户语义期望的才是正确的 SQL 查询, 例如表 2 中的第 4 条 SQL 查询更符合用户的语义期望, 第 6.4.4 节中 ISST 方法合成的 SQL 查询更符合用户的语义期望.

表 2 “Textbook Ex 5.1.5”测试样例的 SQL 查询候选集

No.	SQL查询
1	select distinct faculty.F_name from class, faculty where (faculty.F_key=class.F_key) and (class.Room in (‘R102’))
2	select distinct faculty.F_name from class, faculty where (faculty.F_key=class.F_key) and class.Room = ‘R103’ INTERSECT select distinct faculty.F_name from class, faculty where (faculty.F_key=class.F_key) and class.Room = ‘R102’
3	select distinct faculty.F_name from class, faculty where (faculty.F_key=class.F_key) and class.Room = ‘R102’ INTERSECT select distinct faculty.F_name from class, faculty where (faculty.F_key=class.F_key) and class.Room = ‘R101’
4	select distinct faculty.F_name from faculty where (NOT EXISTS((select distinct t1.Room from class t1) EXCEPT (select distinct t1.Room from class t1, faculty t2 where (t2.F_key=t1.F_key) and (faculty.F_name = t2.F_name))))

我们参照《Database Management Systems》书籍[26]的课后习题答案和合成成功判断标准, 判断 ISST 方法合成 SQL 查询是否成功, 并记录满足用户期望的 SQL 查询在候选集中的排序位置.

6.3 测试集和对比方法选取

我们的实验测试集来自 SQLSynthesizer[7]工作的 28 条测试样例, 其中第 1–23 条测试样例来自经典书籍《Database Management Systems》[26]中第 5 章的课后习题, 另外第 24–28 条测试样例来自 StackOverflow 等在线问答网站, 我们把这个测试集记为 SQLSynthesizer_Benchmark 测试集. 平均每个测试样例的输入示例表包含 24 条元组、4 个属性, 有 23 条测试样例包含多张输入示例表.

当前 SQL 查询程序合成的最新方法有 SQLSynthesizer[7]、SCYTHE[9,10]、SQLSOL[11]和 SQUARES[12]等. SCYTHE、 SQLSOL 和 SQUARES 方法都使用了 SQLSynthesizer_Benchmark 测试集, 并且给出了测试结果统计情况. SCYTHE 和 SQUARES 方法的合成成功率与 SQLSynthesizer 方法相近 (67.9%), SQLSOL 方法的成功率较高 (89.3%). SCYTHE 和 SQUARES 方法合成的 SQL 查询虽然功能正确, 但可读性差、不具有通用性和语义上不满足用户的期望等缺点. 由于 SQLSOL 方法仅公开了部分实现, 我们补充缺失的代码实现并验证 (合成超时时间设为 60 s), 但未能得出与论文中一致结论. SQLSynthesizer 方法对合成结果的评价标准和我们的方法类似, 并且论文中给出了每个测试样例的详细测试结果. 我们获取了 SQLSynthesizer 作者开源的工具实现, 并且在 SQLSynthesizer_Benchmark 测试集上的测试结果与论文中的测试结果基本一致.

因此, 我们用 SQLSynthesizer_Benchmark 测试集和 SQLSynthesizer 方法, 作为 ISST 方法的测试集和对标方法. SQLSynthesizer_Benchmark 测试集共有 28 条用例, 其中有 5 个用例是<type 1>类型, 1 个用例是<type 2>类型, 5 个用例是<type 3>类型, 6 个用例是<type 4>类型, 其余 10 个是包含嵌套查询的用例, 我们实验中设置嵌套查询最多嵌套 3 层.

6.4 实验结果和分析

表 3 测试结果明细表的属性说明: “Benchmarks” 给出测试样例的统计信息, “SQLSynthesizer” 和“ISST” 分别给出对应方法的测试结果. “#Input Tables”表示输入示例表的数量, “#Tupples”表示输入示例表的元组数量. “#Columns”表示输入示例表的属性数量. “Time cost (s)”表示合成耗时, 单位是秒 (s); “Rank”表示表示满足期望的 SQL 查询在候选集中的排序位置, “X”表示合成失败.

6.4.1 合成正确率和耗时

测试结果如表 3 所示, SQLSynthesizer 方法合成了 19 条测试样例, 成功率为 67.9%, 平均耗时 20 s; ISST 方法

合成了 24 条测试样例, 成功率为 85.7%, 平均耗时 2 s. 在合成成功率和耗时方面, ISST 方法都表现出很大的优势. 我们分析, 由于 SQLSynthesizer 方法过于依赖 PART 决策树, 只有当决策树的准确率达到 100% 时, 才能合成所有正确的谓词, 而受限于训练数据和算法的不足, 导致训练结果往往是无限逼近、却很难达到 100% 的准确率, 所以导致 SQLSynthesizer 方法的成功率不高. ISST 方法采用的是归纳合成的方法, 对不同类型 SQL 查询采用不同的方法划分 PT 和 NT 集合, 然后归纳合成谓词, 优点是对已支持类型和场景下的 SQL 查询合成成功率有保障, 缺点是对新类型的 SQL 查询合成泛化能力不足. 我们分析了 ISST 方法合成失败的 4 条测试样例, 其中 “Textbook Ex 5.1.10”和“Textbook Ex 5.1.12”是含有聚合函数“ALL”的嵌套查询、“Textbook Ex 5.2.11”需要使用 ANY 聚合函数、“Textbook Ex 5.1.4”需要进行行比较归纳, 这些是 ISST 方法暂不支持的 SQL 语法和类型, 我们将在未来的工作中继续研究这些类型和场景下的合成方法.

表 3 Benchmark 测试结果明细表

Benchmarks					SQLSynthesizer		ISST	
ID	Source	#Input Tables	#Tuples	#Columns	Rank	Time cost (s)	Rank	Time cost (s)
1	Textbook Ex 5.1.1	4	28	9	1	31	1	1
2	Textbook Ex 5.1.2	4	35	9	X	16	3	10
3	Textbook Ex 5.1.3	2	52	4	1	20	1	1
4	Textbook Ex 5.1.4	3	49	6	X	20	X	1
5	Textbook Ex 5.1.5	2	17	5	1	12	4	2
6	Textbook Ex 5.1.6	3	44	6	1	28	1	2
7	Textbook Ex 5.1.7	1	10	3	1	9	1	1
8	Textbook Ex 5.1.8	1	9	3	1	8	1	1
9	Textbook Ex 5.1.9	2	22	5	1	34	1	1
10	Textbook Ex 5.1.10	2	19	4	1	19	X	2
11	Textbook Ex 5.1.11	4	22	4	X	14	1	1
12	Textbook Ex 5.1.12	4	14	3	X	6	X	1
13	Textbook Ex 5.2.1	2	12	4	1	14	1	1
14	Textbook Ex 5.2.2	3	18	6	1	19	3	2
15	Textbook Ex 5.2.3	3	21	6	X	25	1	1
16	Textbook Ex 5.2.4	3	28	6	1	22	3	1
17	Textbook Ex 5.2.5	3	24	5	X	7	1	1
18	Textbook Ex 5.2.6	3	19	5	1	12	1	1
19	Textbook Ex 5.2.7	3	25	6	1	16	3	1
20	Textbook Ex 5.2.8	3	38	6	X	27	1	1
21	Textbook Ex 5.2.9	3	31	6	1	19	1	1
22	Textbook Ex 5.2.10	3	35	6	1	69	1	1
23	Textbook Ex 5.2.11	3	30	8	X	15	X	4
24	Forum Question 1	1	11	2	1	7	2	1
25	Forum Question 2	1	5	3	1	8	1	1
26	Forum Question 3	1	24	3	1	10	1	8
27	Forum Question 4	3	10	9	1	120	1	1
28	Forum Question 5	2	7	6	1	6	1	1

6.4.2 RANK 排序

如表 3 所示, 在 28 条测试样例中, 有 18 条用例的 SQL 查询在候选集中排在第 1 位, 最差排序位置是第 4 位. 我们分析, 由于 ISST 方法能够合成多条满足<输入-输出>示例表对的 SQL 查询程序, 其中包含了语法简单的查询出特定元组的 SQL 查询, 即根据元组的某些标识属性值进行定向选择, 形如“column = xxx”之类的定向元组选择. 如表 2 是 “Textbook Ex 5.1.5”测试样例的 4 条 SQL 查询候选集, 其中前 3 条是根据属性值进行的定向选择, 虽然这些 SQL 查询可以从输入示例表中成功查询出输出示例表数据, 但是不具有通用性, 不符合用户的语义期望. 用

户期望 SQL 查询的语义描述为“找出在所有教室都上过课的教师的名字”, 显然候选集中的前 3 条都不符合该语义描述, 只有第 4 条是满足该语义描述的 SQL 查询.

ISST 方法也提供了属性排除接口, 用户基于对数据的了解或修正缩减候选集数量, 可以利用该接口指定排除部分不可能的属性. 也可以通过增加用户自定义约束常量的方法, 排除掉某些常量值, 剪枝程序搜索空间、加快合成速度. 例如, 对“Textbook Ex 5.1.5”测试样例, 可以指定合成过程中不使用“Room”属性或不使用“R101”“R102”“R103”等常量值, 排除此类候选 SQL 查询. 另外, 也可以通过在<输入-输出>示例表中逐步增加反向元组的方法逐步逼近用户期望的 SQL 查询.

6.4.3 主要方法试验结果对比分析

SQLSynthesizer[7]、SCYTHE[9]、SQLSOL[11]和 SQUARES[12]方法都使用了 SQLSynthesizer_Benchmark 测试集, 表 4 统计这些方法的测试结果, 和 ISST 方法进行对比, 给出了解决、未解决的测试用例比例和平均耗时. SQLSynthesizer 方法的成功率为 71.4%, SCYTHE 和 SQUARES 方法成功率均为 67.9%. SQLSOL 支持手动配置 WHERE 子句的选谓词数量, 当谓词数量为 1、2 和 3 时解决的测试样例个数分别为 17、22 和 25 个. SCYTHE 论文和 SQUARES 论文中没有明确给出合成平均耗时, 在表 4 中标记为“X”未知.

表 4 主要方法验证结果统计

方法	Solved			Unsolved		
	Count	Percentage (%)	Average time (s)	Count	Percentage (%)	Average time (s)
SQLSynthesizer	19	67.9	20	9	32.1	16.5
SCYTHE	18	64.3	X	10	35.7	X
SQLSOL	25	89.3	10	3	10.7	41
SQUARES	19	67.9	X	9	32.1	X
ISST	24	85.7	2	4	14.3	2

6.4.4 可读性和语义一致性分析

我们使用 SQLSOL[11]方法论文中的“Motivating Example”说明 SQL 查询的可读性. 这个样例来自经典书籍《Database Management Systems》[26], 用户需求描述为: “Find the name and average score of each senior student whose average score is greater than 59”. 由 SQLSOL 方法论文中的“Motivating Example”说明可知, 该样例含有 2 个输入示例表: “Student”和“Grade”表, 元组数量总和为 14 条, 共含有 6 个属性, 图 12 为实际用户手写的 SQL 查询, 图 13 为 SCYTHE 方法[9]合成的 SQL 查询, 图 14 为 SQLSOL 方法[11]合成的 SQL 查询, 图 15 为 ISST 方法合成的 SQL 查询, SQUARES 方法[12]未能在 600 s 内合成该 SQL 查询.

```
SELECT student.name AS name, AVG(grade.score) AS average
FROM student JOIN grade ON student.id = grade.s_id
WHERE student.level = ‘senior’
GROUP BY student.name
HAVING AVG(grade.score) > 59
```

图 12 SQLSOL 方法论文 motivating example 的手写 SQL 查询

图 13 为 SCYTHE 方法[9]合成的 SQL 查询, 虽然功能正确, 能成功从“Student”和“Grade”关联表中查询出“Output”表数据, 但合成的 SQL 查询结构非常复杂, 含有多层嵌套查询组合, 而且语义也不符合用户的期望, 它里面含有谓词“score > 59”, 而不是用户语义描述中的“AVG(score) > 59”.

图 14 为 SQLSOL 方法[11]合成的 SQL 查询, 在候选集中排第 20 位, 合成耗时 821 s.

图 15 为 ISST 方法合成的 SQL 查询, 在候选集中排第 1 位, 合成耗时 4 s. 可以看出该 SQL 查询与图 12 中用户手写的 SQL 查询基本相同, 不仅功能正确, 可续性好, 而且语义上满足用户期望, 符合用户语义描述中的“AVG(score) > 59”条件.

```
SELECT t2. student, t2. avg_score
FROM
  (SELECT t4.student, t4.level, t4.avg_score,
     t5.student As student1,
     t5.level As level1, t5.course, t5.score
   FROM ((SELECT
              t3.student, t3.level, AVG(t3.score) As avg_score
          FROM
              (SELECT *
                   FROM
                        input_table_0
                   WHERE input_table_0.score > 59.0) As t3
          GROUP BY
              t3.student, t3.level) As t4 JOIN
     (SELECT *
          FROM
              input_table_0
     WHERE input_table_0.score = 59.0) As t5)) As t2
WHERE t2.level = t2.level1;
```

图 13　SCYTHE 方法合成的 SQL 查询

```
SELECT student.id, student.name, student.level, grade.course, grade.score
FROM student, grade
WHERE student.id=grade.s_id AND student.name >= (strv “stu4”) or student.level = (strv “senior”)
GROUP BY student.id
HAVING AVG (grade.score) > 59
```

图 14　SQLSOL 方法合成的 SQL 查询

```
SELECT distinct student.name, avg (grade.score)
FROM student, grade
WHERE (student.id=grade.s_id) and (student.level = ‘senior’)
GROUP BY student.name
HAVING (AVG (grade.score) >= 60)
```

图 15　ISST 方法合成的 SQL 查询

7 结　论

ISST 方法基于 PBE 和归纳合成的思想, 它能够根据用户编辑的含有少量元组的<输入-输出>示例表对自动合成 SQL 查询程序. ISST 方法首先对用户规约进行语义分析, 构建所有可能类型的 SQL 查询程序草图. 我们通过对 IE 表进行数据扩展, 以支持归纳合成 HAVING 子句的谓词和包含子查询的谓词. 将扩展数据表划分成正反例元组集合和归纳合成谓词是 ISST 方法的重要步骤. 相比较于其他方法, ISST 方法合成的 SQL 查询, 不仅功能正确, 而且在 SQL 查询程序的语义和结构上, 也符合用户的期望, 可读性好、具有通用性. 实验结果表明, 在合成成功率和耗时方面, ISST 方法都表现出很大的优势. 我们在未来的工作中将继续研究合成具有更复杂语法结构的 SQL 查询程序, 如含有析取连接的谓词和复杂聚合函数 (ALL、ANY 等) 的 SQL 查询.

References:

[1] TIOBE. 2022. https://www.tiobe.com/tiobe-index/

[2] Stack Overflow. 2022. https://stackoverflow.com/

[3] Liu BB, Dong W, Wang J. Survey on intelligent search and construction methods of program. Ruan Jian Xue Bao/Journal of Software, 2018, 29(8): 2180–2197 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5529.htm [doi: 10.13328/j.cnki.jos.005529]

[4] Gulwani S. Automating string processing in spreadsheets using input-output examples. In: Proc. of the 38th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Austin: ACM, 2011. 317–330. [doi: 10.1145/1926385.1926423]

[5] Le V, Gulwani S. FlashExtract: A framework for data extraction by examples. In: Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Edinburgh: ACM, 2014. 542–553. [doi: 10.1145/2594291.2594333]

[6] Zhang MH, Elmeleegy H, Procopiuc CM, Srivastava D. Reverse engineering complex join queries. In: Proc. of the 2013 ACM SIGMOD Int'l Conf. on Management of Data. New York: ACM, 2013. 809–820. [doi: 10.1145/2463676.2465320]

[7] Zhang S, Sun YY. Automatically synthesizing SQL queries from input-output examples. In: Proc. of the 28th IEEE/ACM Int'l Conf. on Automated Software Engineering. Silicon Valley: IEEE, 2013. 224–234. [doi: 10.1109/ASE.2013.6693082]

[8] Tan WC, Zhang MH, Elmeleegy H, Srivastava D. Reverse engineering aggregation queries. Proc. of the 2017 VLDB Endowment, 2017, 10(11): 1394–1405. [doi: 10.14778/3137628.3137648]

[9] Wang CL, Cheung A, Bodik R. Synthesizing highly expressive SQL queries from input-output examples. ACM SIGPLAN Notices, 2017, 52(6): 452–466. [doi: 10.1145/3140587.3062365]

[10] Wang CL, Cheung A, Bodik R. Interactive query synthesis from input-output examples. In: Proc. of the 2017 ACM Int'l Conf. on Management of Data. Chicago: ACM, 2017. 1631–1634. [doi: 10.1145/3035918.3058738]

[11] Cheng L. SqlSol: An accurate SQL query synthesizer. In: Proc. of the 21st Int'l Conf. on Formal Engineering Methods Formal Methods and Software Engineering. Shenzhen: Springer, 2019. 104–120. [doi: 10.1007/978-3-030-32409-4_7]

[12] Orvalho P, Terra-Neves M, Ventura M, Martins R, Manquinho M. SQUARES: A SQL synthesizer using query reverse engineering. Proceedings of the VLDB Endowment, 2020, 13(12): 2853–2856. [doi: 10.14778/3415478.3415492]

[13] Thakkar A, Naik A, Sands N, Alur R, Naik M, Raghothaman M. Example-guided synthesis of relational queries. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. ACM, 2021. 1110–1125. [doi: 10.1145/3453483.3454098]

[14] Frank E, Witten IH. Generating accurate rule sets without global optimization. In: Proc. of the 15th Int'l Conf. on Machine Learning. Madison: Morgan Kaufmann Publishers Inc., 1998. 144–151.

[15] De Moura L, Bjørner N. Z3: An efficient SMT solver. In: Proc. of the 14th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. Budapest: Springer, 2008. 337–340. [doi: 10.1007/978-3-540-78800-3_24]

[16] Martins R, Chen J, Chen YJ, Feng Y, Dillig I. Trinity: An extensible synthesis framework for data science. Proc. of the 2019 VLDB Endowment, 2019, 12(12): 1914–1917. [doi: 10.14778/3352063.3352098]

[17] Tran QT, Chan CY, Parthasarathy S. Query by output. In: Proc. of the 2009 ACM SIGMOD Int'l Conf. on Management of Data. Providence: ACM, 2009. 535–548. [doi: 10.1145/1559845.1559902]

[18] Tran QT, Chan CY, Parthasarathy S. Query reverse engineering. The VLDB Journal, 2014, 23(5): 721–746. [doi: 10.1007/s00778-013-0349-3]

[19] Kalashnikov DV, Lakshmanan LVS, Srivastava D. FastQRE: Fast query reverse engineering. In: Proc. of the 2018 Int'l Conf. on Management of Data. Houston: ACM, 2018. 337–350. [doi: 10.1145/3183713.3183727]

[20] Bonifati A, Ciucanu R, Staworko S. Learning join queries from user examples. ACM Trans. on Database Systems, 2016, 40(4): 24. [doi: 10.1145/2818637]

[21] Zloof MM. Query by example. In: Proc. of the 1975 National Computer Conf. and Exposition. Anaheim: ACM, 1975. 431–438. [doi: 10.1145/1499949.1500034]

[22] Panev K, Michel S. Reverse engineering top-K database queries with PALEO. In: Proc. of the 19th Int'l Conf. on Extending Database Technology. Bordeaux: OpenProceedings.org, 2016. 113–124. [doi: 10.5441/002/edbt.2016.13]

[23] Shen YY, Chakrabarti K, Chaudhuri S, Ding BL, Novik L. Discovering queries based on example tuples. In: Proc. of the 2014 ACM SIGMOD Int'l Conf. on Management of Data. Snowbird: ACM, 2014. 493–504. [doi: 10.1145/2588555.2593664]

[24] Yaghmazadeh N, Wang YP, Dillig I, Dillig T. SQLizer: Query synthesis from natural language. Proc. of the ACM on Programming Languages, 2017, 1(OOPSLA): 63. [doi: 10.1145/3133887]

[25] Yu T, Yasunaga M, Yang K, Zhang R, Wang DX, Li ZF, Radev D. SyntaxSQLNet: Syntax tree networks for complex and cross-domain text-to-SQL task. In: Proc. of the 2018 Conf. on Empirical Methods in Natural Language Processing. Brussels: Association for Computational Linguistics, 2018. 1653–1663. [doi: 10.18653/v1/D18-1193]

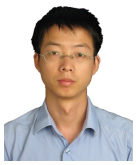
[26] Ramakrishnan R, Gehrke J. Database Management Systems. 3rd ed., Boston: McGraw-Hill, 2002.

附中文参考文献:

[3] 刘斌斌, 董威, 王戟. 智能化的程序搜索与构造方法综述. 软件学报, 2018, 29(8): 2180–2197. http://www.jos.org.cn/1000-9825/5529.htm [doi: 10.13328/j.cnki.jos.005529]

张健(1984－), 男, 硕士, 主要研究领域为程序合成, 机器人软件.

彭鑫(1979－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件智能化开发, 云原生与智能化运维, AI 系统工程.



李弋(1975－), 男, 博士, 讲师, CCF 专业会员, 主要研究领域为机器人软件, 程序分析, 计算机系统架构.

赵文耘(1964－), 男, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件工程.