PSR-SQUARES：基于程序空间约简器的 SQL 逆向合成系统

窦全胜[1,2]，张顺[1]，潘浩[1]，王荟贤[1]，唐焕玲[1]
（1. 山东工商学院信息与电子工程学院，山东 烟台 264005；
2. 喀什大学计算机科学与技术学院，新疆 喀什 844006）

摘 要：针对 SQUARES 程序空间增长过快，导致程序合成效率偏低的问题，在 SQUARES 的基础上，增加了以深度神经网络为核心的程序空间约简器，将给定的<被查询表，查询结果>示例表示成二维张量，作为深度神经网络的输入，网络的输出是关于目标 SQL 语句合成规则的相关性标记向量。约简器根据神经网络的输出结果，采用末 N 位淘汰策略，删除与目标 SQL 语句相关性弱的合成规则，以减少候选 SQL 语句的生成和验证，提升系统合成效率。对约简器中深度神经网络的结构设计、训练样本集的生成方法和网络训练过程进行了详细描述。同时将 PSR-SQUARES 与当前有代表性 SQL 逆向合成系统进行实验对比，实验结果表明，PSR-SQUARES 的综合性能不同程度地优于其他合成系统，平均合成时间由 SQUARES 的 251 s 降低至 130 s，目标程序合成成功率由 80% 提升至 89%。

关键词：程序合成；SQL 逆向合成；SQUARES；程序空间约简器；领域特定语言
中图分类号：TP39
文献标志码：A
DOI: 10.11959/j.issn.1000−436x.2023203

PSR-SQUARES: SQL reverse synthesis system based on program space reducer

DOU Quansheng[1,2], ZHANG Shun[1], PAN Hao[1], WANG Huixian[1], TANG Huanling[1]
1. School of Information and Electronic Engineering, Shandong Technology and Business University, Yantai 264005, China
2. School of Computer Science and Technology, Kashi University, Kashi 844006, China

Abstract: In order to address the issue of rapid growth of program space in SQUARES, which led to low efficiency in program synthesis, a program space reducer based on deep neural network (DNN) was introduced into the SQUARES framework. A given <Queried tables, Query result> pair was represented as a 2D tensor which was used as input for a DNN. And the output of the DNN was the relevance vector of the target SQL statement synthesis rules. Based on the output of the DNN, the last N rules with weak correlation to the target SQL statement were eliminated, thereby shrinking the program search space and improving the system synthesis efficiency. The architecture of DNN, the method of generating training datasets, and the training process of DNN were described in detail. Furthermore, experimental comparisons between PSR-SQUARES and other representative SQL reverse synthesis systems were conducted. The results show that the overall performance of PSR-SQUARES is superior to other synthesis systems to varying degrees, with the average synthesis time reduced from 251 s in SQUARES to 130 s and the target program synthesis success rate increased from 80% to 89%.

Keywords: program synthesis, SQL reverse synthesis, SQUARES, program space reducer, domain-specific language

收稿日期：2023−07−08；**修回日期：**2023−10−13
基金项目：国家自然科学基金资助项目（No.61976124, No.61976125）；新疆维吾尔自治区自然科学基金面上项目（No.2022D01A237, No.2022D01A238）
Foundation Items: The National Natural Science Foundation of China (No.61976124, No.61976125), The Natural Science Foundation of Xinjiang Uygur Autonomous Region (No.2022D01A237, No.2022D01A238)

0 引言

程序设计是人类复杂的智能活动。早在 20 世纪，美国麻省理工学院的 Arthur Lee Samuel 教授就曾提出“不用明确编程，让计算机自动学会解题”的设想。如何让计算机自动编写程序，是人工智能领域研究者希望攻克的重大问题。传统研究中，这类问题被统称为“程序合成”（PS, program synthesis）。在 PS 任务中，用户表达意图的规约方式有 3 种：基于代码框架的程序规约[1-2]、基于自然语言描述的程序规约[3-4]和基于<输入，输出>示例的程序规约[5-6]。相比而言，基于<输入，输出>示例的程序规约方式更加简单、方便和自然，用户只需提供一组<输入，输出>示例，计算机便根据这组示例自动归纳合成与之一致的程序。

基于<输入，输出>示例的程序合成被称作归纳程序合成（IPS, inductive program synthesis）。查询逆向工程[7]（QRE, query reverse engineering）是归纳程序合成在数据分析领域的一个重要分支，其任务是根据<被查询表，查询结果>示例，合成与该示例一致的结构化查询语言（SQL）语句。QRE 具有重要的现实意义和应用价值，一经提出就受到相关领域研究者的广泛关注，成为人工智能和数据库领域研究的热点。

QRE 是一项具有挑战性的任务，Das-Sarma 等[8]已经从理论上证明了 QRE 任务是一个多项式空间（PSPACE）难题。2009 年，新加坡国立大学研究团队[9]提出的 TALOS 是早期颇具影响的工作之一。TALOS 根据数据属性的不同取值，对被查询表进行划分，最终产生一个与期望结果一致的数据表。这一过程可以视为一棵决策树，树的根节点对应被查询表，叶节点对应期望得到的查询结果，通过遍历根节点至叶节点的路径，即可获得相应的查询条件，并生成与<被查询表，查询结果>示例一致的 SQL 语句。显然，这样的决策树并不唯一，因此对于相同的<被查询表，查询结果>示例，TALOS 合成的 SQL 语句也不唯一，用户需要在多个输出结果中进行选择，这对于没有专业背景的应用者来说，存在一定困难。2014 年，该团队[10]对 TALOS 进行改进，将<被查询表，查询结果>示例中的元组分为正元组和负元组，通过减少正元组数量，克服数据集划分过程中的过度约束，提高 SQL 语句的合成效率。2015 年，该团队[11]提出 QFE（query for examples）合成系统。该系统增加了交互机制，通过用户的反馈，对合成的 SQL 语句进行过滤，以解决 SQL 语句的最终选择问题。

近年来，QRE 领域的研究不断取得进展，一些新的合成模型和系统，如 SQLSynthesizer[12]、Scythe[13-14]、REGAL[15-16]、UNMASQUE[17]、RELIC[18]、DExPlorer[19]、Trinity[20]、SQUARES[21]、Duoquest[22]等不断涌现。在这些系统中，纽约大学 WANG 等[13-14]提出的 Scythe 和卡内基梅隆大学 Martins 等[20]提出的 Trinity 具有较强代表性。Scythe 采用抽象查询作为合成 SQL 语句的框架。抽象查询的语法结构与 SQL 语句类似，不同之处在于，抽象查询含有若干未确定的“孔”，将这些“孔”实例化，即可得到具体的 SQL 语句。

与 Scythe 不同，Trinity 利用领域特定语言（DSL, domain-specific language），将 SQL 语句的合成问题转化为基于 DSL 的程序合成问题。Trinity 对 DSL 中所有终结符和非终结符进行编号，并利用完全 k-叉树表示 DSL 程序，k-叉树的每个节点对应一个编码变量，对编码变量的取值进行约束，这样一个合法的 DSL 程序即可用一组满足约束的编码变量取值表示。显然，这是一个整数规划问题，Trinity 采用成熟的约束求解器，如 SAT、SMT 对其进行求解，满足约束的编码变量的解即对应一个合法的 DSL 程序。

将以 Scythe 和 Trinity 为代表的 2 种合成模式分别称为“框架式”合成和“编码式”合成。目前 QRE 领域的大部分研究采用了上述 2 种模式。2020 年，Baik 等[22]提出了合成系统 Duoquest，该系统首先利用<被查询表，查询结果>示例产生若干 SQL 框架，然后根据用户提供的自然语言描述完成目标语句的合成，这种通过自然语言启发搜索的方式，能有效避免一些无效搜索，但系统需要对自然语言进行处理和理解，这同样是一个不易解决的难题。2020 年，Takenouchi 等[23]提出的 PATSQL 利用关系代数中的转换规则对 SQL 框架进行限制，以减少无关框架的生成，这种策略本质上是利用“知识”来对程序空间进行约简，该系统在一些测试问题上表现出较高合成效率，然而对于“知识”以外的<被查询表，查询结果>示例，合成效率并没有显著提高。2022 年，Zhou 等[24]提出了面向分析型 SQL 的合成系统（QRE），在该系统中，ZHOU 提出了一种新颖的用户规约规范，该规范可以让用户通过动态的单元级计算过程来表达自己的意图，这是对“示例”概念的扩展，具有一定的开创意义。

在诸多合成系统中，Orvalho 等[21]提出的 SQUARES 综合性能表现最突出，该系统采用了与 Trinity 相同的“编码式”合成策略，不同的是，SQUARES 用链式结构来表示 DSL 程序，从而使编码变量和相应约束的增长速度由 Trinity 的指数级降至多项式级，大幅减小了约束求解的复杂度。与 Trinity 相比，SQUARES 的合成效率有较大幅度提升。

自 2006 年深度学习提出以来，人工智能领域研究进入新发展阶段，以深度学习为代表的机器学习技术不断取得突破，利用深度学习技术辅助归纳程序合成，已成为 IPS 领域研究的趋势，面向不同问题的程序合成系统，如 DEEPCODER[25]、CODEGEN[26]、SIGNAL[27]、NPI[28]、TYGAR[29]、CodeRL[30]、NPL[31]、PCoder[32]等不断被提出。相比而言，深度学习与 QRE 相结合的研究工作相对较少。QRE 任务有 2 个关键问题需要解决，一是程序生成，具体而言，就是如何在巨大的程序空间中，获得满足用户规约的 SQL 语句；二是选择偏好，即如何在所有满足用户规约的 SQL 语句中，确定反映用户真实意图的 SQL 语句。本文针对程序生成问题，将深度学习技术引入 QRE 研究领域，以提升目标程序的合成效率。

本文主要的研究工作如下。

1) 在 SQUARES 中增加一个约简器，约简器通过深度神经网络（DNN, deep neural network）实现从<被查询表，查询结果>示例到合成规则相关性标记向量的映射。

2) 给出了相关深度神经网络的结构设计，输入张量的编码方式，训练样本集的生成方法，基于深度神经网络的预测结果，提出了末 N 位淘汰的合成规则约简策略。

3) 通过实验对比，验证了 PSR-SQUARES 的有效性。

1　背景知识和定义

定义 1　表。一个数据表 Γ 可用三元组 (C,R,τ) 表示，其中，C 和 R 为大于 0 的整数，分别表示 Γ 的列数和行数。$\tau=[\tau_1,\cdots,\tau_C]$ 为列表，列表中元素 τ_i 为 Γ 第 i 个属性的类型。

设 $D=\{\Gamma^i\}_{i=1}^m$ 为由 m 个表构成的数据库，对于 $\forall\Gamma^i,\Gamma^j\in D$，若 Γ^i 和 Γ^j 之间存在共享的属性，则称表 Γ^i 和 Γ^j 之间存在联系。

定义 2　查询逆向工程。设 D 为存在联系的被查询表集合，Q 是可在 D 上运行的未知查询语句，Q(D)是在 D 上运行 Q 得到的期望输出，给定<D,Q(D)>生成查询语句 Q 的过程，被称为查询逆向工程。

图 1 和图 2 为 QRE 任务的一个具体范例。

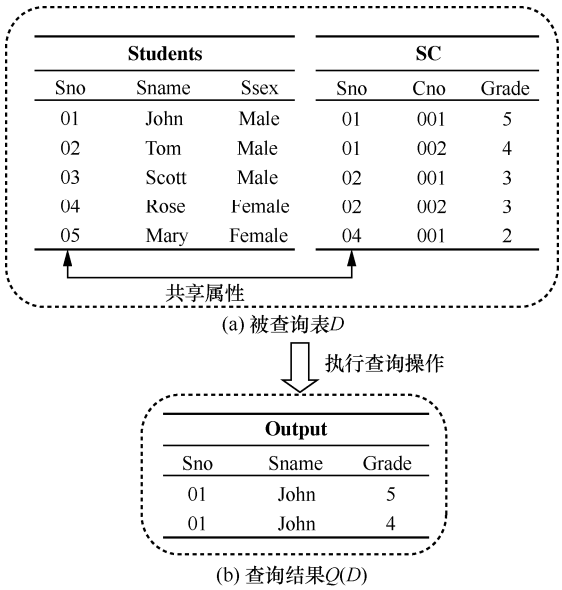

图 1　<被查询表，查询结果>示例

```
SELECT ‘Sno’,‘Sname’,‘Grade ’
FROM
  (SELECT ‘Sno’,‘Sname’,‘Ssex’,‘Cno’,‘Grade ’
       FROM ‘input0’ AS ‘TBL_LEFT’
       INNER JOIN
         (SELECT *
          FROM ‘input1’
          WHERE (‘Grade ’ > 3.0)) AS
‘TBL_RIGHT’ ON (‘TBL_LEFT’.‘Sno’ =
‘TBL_RIGHT’.‘Sno’))
```

图 2　<被查询表，查询结果>示例合成的 SQL 语句

图 1 给出了一个<被查询表，查询结果>示例，其中，图 1(a)为被查询表 D，包括 2 个数据表：Students(Sno，Sname，Ssex)和 SC(Sno，Cno，Grade)，这 2 个表之间存在共享属性 Sno。图 1(b)是将表 Students 和 SC 作为被查询表，执行查询操作后的期望输出表。

图 2 为根据图 1 中的<被查询表，查询结果>示例，逆向合成的 SQL 语句。该语句可以在图 1(a)所示的被查询表上执行，输出图 1(b)所示的查询结果。

2　SQUARES

在诸多 SQL 逆向合成系统中，Orvalho 等[21]提出的 SQUARES 综合性能最突出，本文工作也以该系统为基础。图 3 给出了 SQUARES 的系统结构。

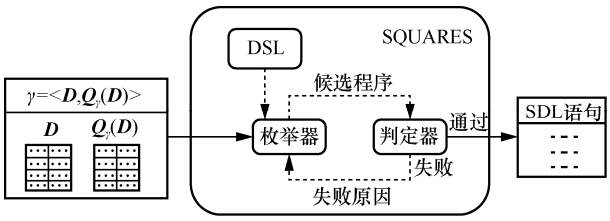

图 3 SQUARES 系统结构

从图 3 可以看出，SQUARES 主要由枚举器和判定器两部分构成，下面就 SQUARES 的结构和 SQL 合成过程进行详细论述。

2.1 领域特定语言

由于 SQL 语句的语法规则复杂，包括 SQUARES 在内的许多 SQL 逆向合成系统并不利用 SQL 语句的语法规则直接合成 SQL 语句，而是通过预定义的 DSL，将 SQL 语句的合成问题转化为基于 DSL 的程序合成问题，再利用 DSL 定义的语义规则，将 DSL 程序转化为对应的 SQL 语句。

领域特定语言可以用二元组(*G*,Ops)表示，其中 *G* 为表 1 所示的上下文无关文法，该文法定义了生成合法 DSL 程序的产生式规则，Ops 为 G 中每个产生式所对应的 SQL 语义。

表 1 DSL 的上下文无关文法

产生式左部	产生式右部
table→	input$^{(0)}$|inner_join(table,table)$^{(1)}$| inner_join3(table,table,table)$^{(2)}$| inner_join4(table,table,table,table)$^{(3)}$| filter(table,filterCondition)$^{(4)}$| filters(table,filterCondition,filterCondition,op)$^{(5)}$| summariseGrouped(table,summmariseCondition,Cols)$^{(6)}$| anti_join(table,table)$^{(7)}$|left_join(table,table)$^{(8)}$| bind_rows(table,table)$^{(9)}$|intersect(table,selectCols,distinct)$^{(10)}$
tableSelect→	select(table,selectCols,distinct)$^{(11)}$
op→	Or$^{(12)}$|And$^{(13)}$
distinct→	true$^{(14)}$|false$^{(15)}$
empty→	ε$^{(16)}$

在表 1 中，每个产生式的右部可以是终结符，也可以是预定义的函数。产生式的左部可以视为这些终结符或函数的返回类型，包括 table、tableSelect、op、distinct 和 empty。产生式规则(0)～规则(10)定义了所有返回值为 table 的产生式规则，考察产生式规则(1)

table→inner_join(table,table)

若将 2 个具有共享属性 *x* 的表 T_A，T_B 作为参数传递给函数 inner_join(table,table)，对应的 SQL 语义为

SELECT * FROM T_A AS `TBL_LEFT` INNER JOIN T_B AS `TBL_RIGHT` ON (`TBL_LEFT`.*x* = `TBL_RIGHT`.*x*)。

考察产生式规则(11)

tableSelect→select(table,selectCols,distinct)

该产生式定义了程序返回的最终结果，这里函数 select 包含参数 table、selectCols 和 distinct，其中 table 和 distinct 分别由产生式规则(0)～规则(10)和规则(14)～规则(15)定义，selectCols 为终结符，可通过用户给定的期望输出表确定。由于 table 的生成可能是多次查询的复合结果，因此，select 函数需要将这些查询合并，返回最终的 SQL 语句。

2.2 枚举器和判定器

设 $\gamma =< D, Q_\gamma(D) >$ 为给定的<被查询表，查询结果>示例，Q_γ 为与 γ 一致的 SQL 语句。枚举器的任务是根据 DSL 语法中定义的产生式规则，生成形式上合法的候选程序 p，并将其传递给判定器。判定器验证程序 p 是否与 γ 一致，若 p 与 γ 一致，则 p 为目标程序，根据 DSL 定义的语义规则，将 p 转化为对应 SQL 语句，该语句即目标 SQL 语句 Q_γ；否则，判定器将失败原因返回给枚举器。

SQUARES 采用了与 Trinity 相同的“编码式”合成策略，将程序合成问题转化成整数规划问题，并利用 SMT 求解器进行求解。不同的是，SQUARES 采用链式结构来表示 DSL 程序，从而使整数规划问题中的编码变量和相应约束的增长速度由 Trinity 的指数级降至多项式级，大幅降低了求解难度。

将整数规划问题中所有编码变量约束的合取式称作 SMT 计算式，相同长度的 DSL 程序与唯一的 SMT 计算式对应，而对于相同 SMT 计算式，可能存在多个不同的解释，每个解释都对应一个形式上合法的 DSL 候选程序。

定义 3 枚举轮数（Ers, enumeration rounds），SMT 计算式的一次更新，称为一轮枚举，SMT 计算式更新的次数称为枚举轮数。

SQUARES 的程序枚举空间与 SMT 计算式中编码变量和约束的数量相关。设 k_i（$i=1,\cdots,10$）为 DSL 产生式 i 右部函数的参数个数，$k_{\max}=\max(k_i)$，|Prod(*D*)|为 DSL 中的产生式数量，|Term(*D*)|为 DSL 中终结符的数量，|IN|为用户提供的对<被查询表，查询结果>示例约束的数量。文献[21]给出了 SMT 计算式编码变量规模 Vs_{size} 和

约束规模 $\mathrm{Cs}_{\mathrm{size}}$ 的估计值

$$\mathrm{Vs}_{\mathrm{size}} \sim O(\mathrm{Ers}\times(k_{\max}+2)) \tag{1}$$

$$\mathrm{Cs}_{\mathrm{size}} \sim O(\mathrm{Ers}\times k_{\max}\times|\mathrm{Prod}(D)|\times(|\mathrm{Term}(D)|+\mathrm{Ers})+|\mathrm{IN}|) \tag{2}$$

由式(1)和式(2)可知，SMT 计算式的复杂度随 Ers 的增加而增大。在一轮枚举中，由于用户提供的输入约束|IN|是确定的，因此，枚举程序空间的规模与参数 $k_{\max}$、|Prod*(D)*|和|Term*(D)*|相关。然而，并非 DSL 中的所有产生式都与 Q_γ 的生成有关，无关的产生式只会使上述参数值变大，增加程序枚举的难度。

显然，控制编码变量和相应约束的增长规模，可以有效提高 SQUARES 合成效率。基于这个思想，本文在 SQUARES 中增加了一个约简器，在程序合成过程中，约简器删除与目标语句不相关的产生式规则，式(1)和式(2)中 $k_{\max}$、|Prod(*D*)|、|Term(*D*)|也会随之减小，程序枚举空间得到了约简，从而提升系统的合成效率。

3　PSR-SQUARES

将具有程序空间约简器的 SQUARES 系统简记为 PSR-SQUARES，其系统结构如图 4 所示。

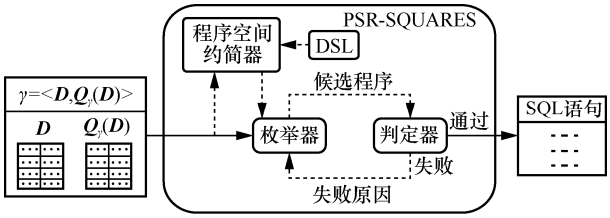

图 4　PSR-SQUARES 系统结构

从图 4 和图 3 的对比中可以看出，PSR-SQUARES 比 SQUARES 多了一个约简器，约简器的核心是一个深度神经网络，训练后的 DNN 可以对<被查询表，查询结果>示例与 DSL 产生式的相关性进行预测，约简器根据 DNN 的预测结果，对相应的合成规则进行约简。

设 $\gamma=<D,Q_\gamma(D)>$ 为给定的<被查询表，查询结果>示例，$\boldsymbol{S}^{(\gamma)}=(s_1^{(\gamma)},\cdots,s_w^{(\gamma)})$ 为一个向量，其中 $s_i^{(\gamma)}\in[0,1]$（$i=1,\cdots,w$）表示 γ 与产生式 i 的相关度。本文称 $\boldsymbol{S}^{(\gamma)}$ 为示例 γ 与合成规则的相关性标记向量，简称 γ 的相关性标记向量。

将 γ 的张量表示作为 DNN 的输入，DNN 的输出为 γ 的相关性标记向量 $\boldsymbol{S}^{(\gamma)}$，即 $\mathrm{DNN}:\gamma\to\boldsymbol{S}^{(\gamma)}$。从表 1 所示文法可知，数据表是由产生式规则(0)～规则(11)生成的。其中，产生式规则(0)是指从输入读取被查询表，产生式规则(11)定义了 DSL 程序返回的最终结果，这 2 个产生式规则与所有程序都相关，无须预测。而产生式规则(1)～规则(10)则与具体示例 γ 有关。因此，标记向量 $\boldsymbol{S}^{(\gamma)}$ 的长度 w=10。

下面，本文就约简器中深度神经网络的结构和训练过程进行论述。

3.1　DNN 的结构

约简器中深度神经网络由输入层、中间层和输出层构成，具体如图 5 所示。

3.1.1　输入层

输入层包括两路输入：D 和 $Q_\gamma(D)$。首先将<被查询表，查询结果>示例表示成张量形式。设 $\gamma=<D,Q_\gamma(D)>$ 为任意<被查询表，查询结果>示例，其中，D 为被查询表，由相关的 m 个数据表 $\Gamma^1,\cdots,\Gamma^m$ 组成。对于 $\forall\Gamma^i\in D$，$i=1,\cdots,m$，其属性（列）和元组（行）个数分别表示为 $\mathrm{cols}^{(\Gamma^i)}$ 和 $\mathrm{rows}^{(\Gamma^m)}$。$Z^{(\Gamma^i)}$ 是与 Γ^i 相对应的二维张量，其行数为 $\mathrm{rows}^{(\Gamma^i)}+1$，列数与 Γ^i 相同。用 $A_j^{(\Gamma^i)}$ 表示表 Γ^i 的第 j 个属性，$\Gamma^i(k,j)$ 和 $Z^{(\Gamma^i)}(k,j)$ 分别表示 Γ^i 和 $Z^{(\Gamma^i)}$ 第 k 行 j 列元素。

张量 $Z^{(\Gamma^i)}$ 的任意元素 $Z^{(\Gamma^i)}(k,j)$ 按如下方式

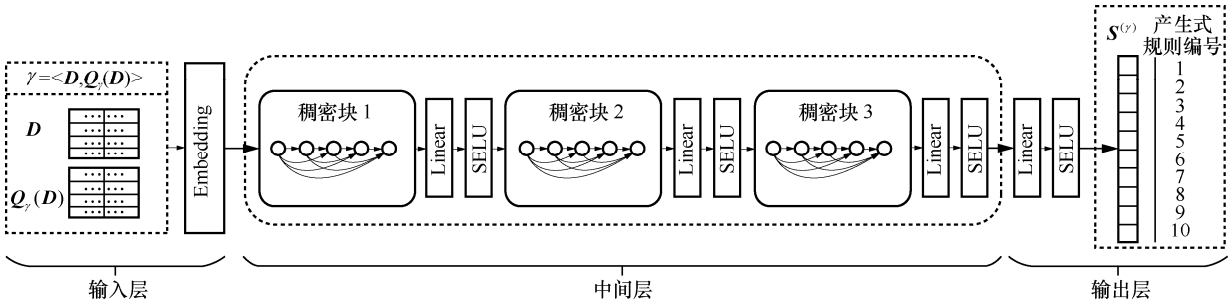

图 5　约简器中深度神经网络的结构

计算。

当 k=1 时，有

$$Z^{(\Gamma^i)}(k,j)=\mathrm{Str2Int}\left(\mathrm{idx}(\Gamma^i)+\mathrm{type}\left(A_j^{(\Gamma^i)}\right)+\mathrm{name}\left(A_j^{(\Gamma^i)}\right)\right) \quad (3)$$

其中，$\mathrm{idx}(\Gamma^i)$ 为表 Γ^i 在整个数据库中的全局索引编码，$\mathrm{type}\left(A_j^{(\Gamma^i)}\right)$ 为属性 $A_j^{(\Gamma^i)}$ 的类型编码，$\mathrm{name}\left(A_j^{(\Gamma^i)}\right)$ 为 $A_j^{(\Gamma^i)}$ 名字的 ASCII 编码串。

当 $k\geqslant 2$ 时，有

$$Z^{(\Gamma^i)}(k,j)=\begin{cases}\Gamma^i(k-1,j), & \Gamma^i(k-1,j)\text{是一个数值}\\ \mathrm{Str2Int}(\text{ASCII code of }\Gamma^i(k-1,j)), & \text{其他}\end{cases} \quad (4)$$

由式(3)和式(4)可知，$Z^{(\Gamma^i)}$ 的第 1 行为表 Γ^i 的各属性信息的编码表示，具体包括表 Γ^i 的全局索引、属性类型和名字等信息。$Z^{(\Gamma^i)}$ 的第 2 行至 $\mathrm{rows}^{(\Gamma^i)}+1$ 行与 Γ^i 中数据相对应。此时分 2 种情况：若属性 $A_j^{(\Gamma^i)}$ 的类型为数字型，$Z^{(\Gamma^i)}(k,j)$ 与 $\Gamma^i(k-1,j)$ 值相等；若属性 $A_j^{(\Gamma^i)}$ 的类型为其他类型，则 $Z^{(\Gamma^i)}(k,j)$ 的值等于 $\Gamma^i(k-1,j)$ 的 ASCII 码串。对于期望输出表 $Q_\gamma(D)$，通过相同的表示方法，亦可得到类似的张量 $Z^{Q(D)}$。

将 $Z^{\Gamma^1},\cdots,Z^{\Gamma^m}$ 合并即可获得第一路的输入张量 INPUT^D，同时将 $Z^{Q(D)}$ 作为另一路输入张量 $\mathrm{INPUT}^{Q(D)}$。由于不同 $<D,Q_\gamma(D)>$ 示例的长度和宽度是不同的，因此，INPUT^D 和 $\mathrm{INPUT}^{Q(D)}$ 需满足最大长度及宽度 $<D,Q_\gamma(D)>$ 示例的需求，对于缺失值则以特殊值 256 填充。

3.1.2 中间层

DNN 的中间层由 3 个稠密块[33]构成。每个稠密块含有 5 个全连接层，在同一稠密块内，全连接层神经元个数是相同的，第 i（1<i⩽5）层的输入为第 1 层至第 i−1 层的输出。不同稠密块全连接层神经元个数不同，图 5 所示的 3 个稠密块全连接层神经元个数分别为 500、200 和 100，神经元全部采用 SELU 函数作为激活函数。

3.1.3 输出层

输出层包括 10 个神经元，输出的结果是示例 γ 的相关性标记向量 $\boldsymbol{S}^{(\gamma)}$。DNN 采用交叉熵作为损失函数，即

$$L_{\mathrm{loss}}=-\sum_{\gamma}\boldsymbol{y}^{\gamma}\log\boldsymbol{S}^{(\gamma)} \quad (5)$$

其中，$\boldsymbol{y}^{\gamma}$ 为示例 γ 真实的相关性标记向量。

3.2 DNN 的训练

在文献[11-12,15]提供的数据基础上，随机选取相关联的数据表作为被查询表 D，根据 DSL 产生式规则随机生成可在 D 上运行的程序 p。由于 p 的生成过程已知，因此与 p 对应的 DSL 产生式规则是确定的。程序 p 真实的产生式相关度标记向量 $\mathrm{y}^p=(y_1^p,\cdots,y_{10}^p)$ 为

$$\boldsymbol{y}_j^p=\begin{cases}1, & \text{产生式规则}j\text{ 与}p\text{相关}\\ 0, & \text{其他}\end{cases}$$

其中，$j=1,\cdots,10$。在 D 上运行 p 可得到 $Q(D)$。将 $<D,Q(D)>$ 记为 γ，则 $<\gamma,\boldsymbol{y}^p>$ 即关于程序 p 的一个训练样本。更新 D 中元组并运行 p 可得到关于程序 p 的不同训练样本。不妨将关于程序 p 的样本集记作 DS^p，重复上述过程，生成不同的合法程序 p 和样本集 DS^p，将这些样本集合并，即可获得最终的训练样本集 $\mathrm{DS}=\bigcup_p\mathrm{DS}^p$。

为了避免 DS 产生数据泄露，在 DS 中与文献[21]的测试样本类似的训练样本，保证训练集与测试集没有数据重叠。

设置训练参数，令 optimizer='Adam'，learning_rate=0.001，epochs=400，batch_size=50。选择评估最优 DNN 模型作为约简器。

训练后的 DNN 可以对<被查询表，查询结果>示例 γ 的相关性标记向量进行预测。约简器根据预测结果 $\boldsymbol{S}^{(\gamma)}$，对产生式与示例 γ 的相关性进行判断，删除与 γ 无关的产生式规则。关于如何根据 $\boldsymbol{S}^{(\gamma)}$ 对产生式进行删减的问题，最直接的做法是，设定一个阈值，并根据 $\boldsymbol{S}^{(\gamma)}$ 与阈值的比较结果，删减与 γ 相关性小的产生式。事实上，对于示例 γ，$\boldsymbol{S}^{(\gamma)}$ 体现的是不同产生式与 γ 相关性大小的相对关系，而对于不同示例确定一个统一阈值并不容易。因此，本文未采用阈值设定法，而是采用相对简单的末 N 位淘汰策略，即根据 $\boldsymbol{S}^{(\gamma)}$ 结果，删除排在后面的 N 个与 γ 相关性弱的产生式。这里将使用末 N 位淘汰策略的 PSR-SQUARES 记为 PSR-SQUARES-N。

4　实验分析

在相同计算环境下，选择文献[21]中的 55 个测试问题，对本文方法进行验证。设定程序合成时间的上限为 3 600 s，并取 5 次实验的平均结果作为最终结果。通过实验分析以下 3 个问题。

1) PSR-SQUARES-*N* 中超参数 *N* 对系统性能的影响。

2) PSR-SQUARES 与 SQUARES 综合性能对比。

3) PSR-SQUARES 与其他 SQL 合成系统综合性能对比。

4.1　超参数 *N* 对 PSR-SQUARES-*N* 的影响

图 6 为 PSR-SQUARES-*N* 和 SQUARES 的合成成功率曲线，其中，$N=1,\cdots,6$。

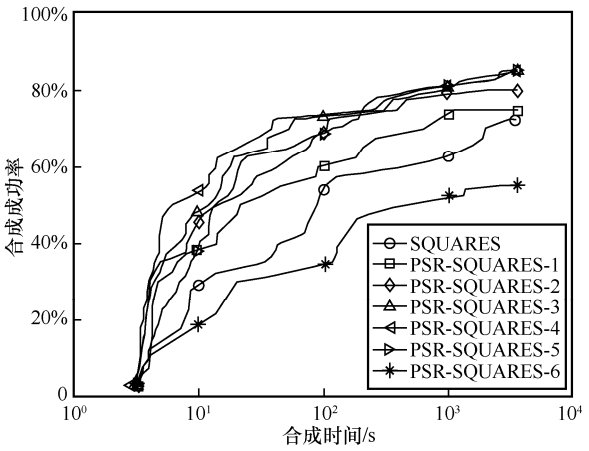

图 6　PSR-SQUARES-*N* 和 SQUARES 的合成成功率曲线

从图 6 中可以看出，当 $N=1,\cdots,5$ 时，PSR-SQUARES-*N* 合成 SQL 语句的成功率不同程度地高于 SQUARES。当 *N*=6 时，PSR-SQUARES-6 的合成成功率低于 SQUARES。这说明 *N* 值过大会淘汰程序合成所必需的合成规则，降低合成成功率。当 $1<N\leqslant 4$ 时，PSR-SQUARES-*N* 的合成成功率始终高于 PSR-SQUARES-(*N*−1)。这说明 *N* 的取值在合理范围内，程序空间得到了有效约简，PSR-SQUARES-*N* 的合成效率和成功率随着 *N* 的增大逐步提高。PSR-SQUARES-5 与 PSR-SQUARES-4 最终成功率相同，但在相同合成时间内，PSR-SQUARES-5 的效率要低于 PSR-SQUARES-4。PSR-SQUARES-6 的合成成功率和效率始终低于 PSR-SQUARES-5。这说明：当 *N* 较大时，部分相关产生式被淘汰，即便存在其他产生式组合可生成满足要求的 SQL 语句，但枚举轮数的增加导致了合成效率下降。更极端地，若合成问题所必需的产生式被删除，则目标程序的合成必会失败，导致成功率下降。

综合上述实验对比结果，最终确定对于本文中的 DSL，PSR-SQUARES-*N* 的最优超参数 *N*=4。

4.2　PSR-SQUARES 与 SQUARES 性能对比

由 4.1 节论述已知，PSR-SQUARES-*N* 的最优超参数 *N*=4，因此本节仅列举 PSR-SQUARES-4 的相关实验结果，并将其与 SQUARES 的进行比较。

表 2 展示了 PSR-SQUARES-4 在 3 个拥有不同枚举轮数的测试样本上最终合成的目标程序。从表 2 中可以看出，每一轮枚举，DSL 程序都会增加一行，目标 SQL 语句的复杂度也会随之递增。以编号为 32 的测试问题为例，针对该测试问题，合成器进行了 6 轮枚举，因此合成的 DSL 程序的行数也是 6。将 DSL 程序转化成对应的目标 SQL 语句，经过 6 轮枚举获得的目标 SQL 语句已经相当复杂，该语句包括 6 层嵌套、7 个临时数据表、7 个判断条件。这充分说明了 QRE 任务的挑战性。针对该测试问题，SQUARES 的平均合成时间为 11 s，而 PSR-SQUARES-4 在合成过程中，将编号为 1、8、9、10 的 4 个产生式进行了删减，合成时间仅为 4.4 s，合成效率提高了约 60%。

表 2　PSR-SQUARES-4 合成的 DSL 程序示例

序号	样本编号	枚举轮数/轮	DSL 程序
1	8	2	L1=summariseGrouped(input0, meanage = mean(age), level) P=select(L1, level,meanage, true)
2	18	4	L1=summariseGrouped(input0, maxcost =max(cost),P_id) L2=inner_join3(input0, L1, input1) L3=filter(L2,cost==maxcost) P=select(L3,P_id,S_name,distinct,true)
3	32	6	L1=summariseGrouped(input2, S_key,S_name) L2=filters(L1, S_key==S4, S_key!=S4) L3=filters(input0,S_key==S4, S_key!=S4) L4=inner_join4(L3, input1,L2, input0) L5=filter(L4,color ==green) P=select(L5, P_id,n, true)

图 7 和图 8 分别为在全部测试样本上，SQUARES 与 PSR-SQUARES-4 的相对时间和合成时间对比。为规范起见，将测试样本按 SQUARES 的平均合成时间由小到大进行重新排序。在图 7 中，将 SQUARES 的合成时间视为 1，

横坐标为重新排序后的测试样本编号，纵坐标为 PSR-SQUARES-4 合成时间与 SQUARES 的合成时间的比值，从图 7 可以看出，PSR-SQUARES-4 合成时间减小的比例。为更详细地对图 8 中 2 个系统进行比较，表 3 给出了 SQUARES 与 PSR-SQUARES-4 在全部测试样本的合成时间对比。由表 3 可以看出，在 3 600 s 内，SQUARES 和 PSR-SQUARES-4 分别成功完成 55 个测试问题中的 44 个和 49 个，成功率分别为 80%和 89%。对于所有可比较的测试样本，PSR-SQUARES-4 合成时间均优于 SQUARES，平均合成时间由 SQUARES 的 251 s 降低至 130 s。

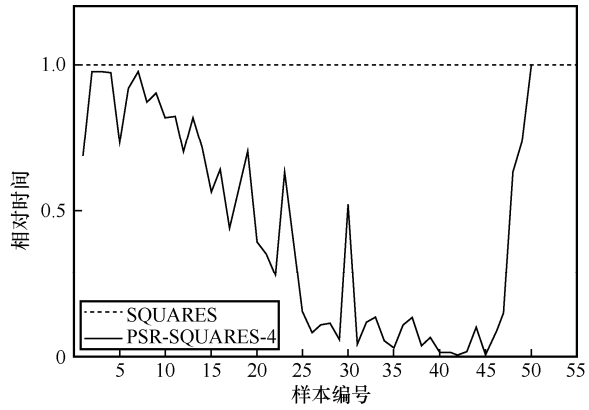

图 7　SQUARES 与 PSR-SQUARES-4 在全部测试样本上相对时间对比

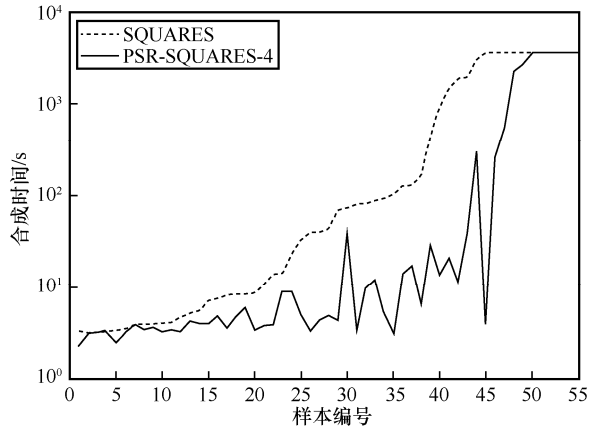

图 8　SQUARES 与 PSR-SQUARES-4 在全部测试样本上合成时间对比

4.3　PSR-SQUARES 与其他 QRE 系统性能对比

采用 4.2 节所述测试问题和实验方法，将 SQUARES、PSR-SQUARES 与另外 2 个有代表性的 QRE 系统 Scythe[14]和 Trinity[20]进行对比，具体结果如表 4 所示。

表 3　SQUARES 与 PSR-SQUARES-4 在全部测试样本的合成时间对比

序号	样本编号	枚举轮数	SQUARES 的合成时间/s	PSR-SQUARES-4 的合成时间/s
1	7	1	3.2	3.1
2	24	4	3.3	3.3
3	1	2	3.7	2.3
4	12	2	3.7	3.6
5	20	5	3.4	3.3
6	21	2	3.4	3.9
7	13	1	3.4	2.5
8	10	3	3.9	3.4
9	37	4	3.9	3.6
10	19	5	4.1	3.3
11	44	5	4.1	5
12	8	2	4.7	3.3
13	36	5	5.2	4.2
14	14	3	5.5	4
15	39	4	7.2	4
16	50	5	7.6	4.8
17	23	5	8.2	3.6
18	28	5	8.4	4.8
19	31	5	8.5	6
20	11	1	8.7	3.4
21	32	6	11	4.4
22	5	3	14	3.8
23	30	6	14	10
24	2	3	23	9
25	29	6	33	5
26	48	4	40	3.3
27	4	3	40	4.3
28	41	5	43	8
29	55	4	69	4.4
30	26	5	74	39
31	45	4	80	3.4
32	3	3	82	9.7
33	54	4	89	12
34	40	5	93	5.2
35	42	5	103	3.9
36	52	4	128	14
37	6	3	128	17
38	15	3	168	6.5
39	53	4	438	29
40	49	4	920	14
41	51	4	1452	21
42	17	4	1898	11.8
43	18	4	1952	38
44	27	4	3065	309
45	35	5	—	3.9
46	47	4	—	256
47	33	6	—	549
48	9	6	—	2271
49	43	4	—	2658
50	46	4	—	—
51	38	5	—	—
52	22	5	—	—
53	16	6	—	—
54	25	6	—	—
55	34	6	—	—

表 4　　不同合成系统性能比较

合成系统	合成成功率	平均合成时间/s
Scythe	60%	893 s
Trinity	66%	724 s
SQUARES	80%	251 s
PSR-SQUARES-4	**89%**	**130 s**

在表 4 所示的合成系统中，Scythe 采用抽象查询作为合成 SQL 语句框架。这里抽象查询的语法结构与 SQL 语句的语法结构相似，只是在其语法规则中，用未确定的“孔”代替具体的数据表或用于筛选数据记录的过滤谓词，将这些“孔”实例化，即可得到具体的 SQL 语句。Scythe 首先生成形式上合法的抽象查询，再根据<被查询表，查询结果>示例将抽象查询中待确定的“孔”实例化，进而合成具体的 SQL 语句。运行所有 SQL 语句，若某一语句的输出与示例中期望的输出结果一致，则该 SQL 语句即目标语句。否则，若 SQL 语句的查询结果包含期望输出，则保留该查询语句和相应的查询结果，新产生的查询结果可视作临时表，在下一次迭代中，将根据这些临时表与最初的被查询表重新生成抽象查询，并对其“孔”进行实例化。重复上述过程，直至合成目标 SQL 语句或超过预先设定的最大合成时间。Scythe 的合成过程本质上是宽度优先搜索，每一次迭代搜索树深度加 1，随着搜索树深度的增加，搜索空间呈指数级增长，尽管每一次迭代，Scythe 通过删除查询结果不包含期望输出的查询语句来进行剪枝，但从宏观上看，其搜索空间的增长速度依然是指数级的。因此，对于复杂的合成问题，Scythe 在预先定义的时间内无法成功合成相应的 SQL 语句。从表 4 也可以看出，对于本文的测试问题，Scythe 成功合成目标语句的比例仅为 60%，平均合成时间为 893 s。

Trinity 将 SQL 合成问题转化为基于 DSL 的程序合成问题，一个形式合法的 DSL 程序与一个 SQL 语句相对应，通过预定义的语义规则实现两者之间的转化。Trinity 将 DSL 程序表示成完全 k-叉树结构，其中，k 为表 1 所示产生式右部函数参数个数的最大值。完全 k-叉树中的内部节点与返回类型为 table 的函数对应，该内部节点的子节点可以是一个类型匹配的函数或终结符号。这样一个长度为 n 的 DSL 程序，可以表示成深度为 n+1 的 k-叉树。若内部节点对应函数的参数不足 k 个，则其相应的子节点用 ε 占位。

对 DSL 中的所有语法符号用整数编号，并将 k-叉树的所有节点用编码变量表示，同时对这些编码变量赋予相应约束条件，一组满足约束条件的编码变量取值即与一个合法的 DSL 程序对应。这样进一步将程序合成问题转化成整数规划问题，Trinity 采用约束求解器对整数规划问题进行求解，满足约束条件的一组编码变量取值，即可解码为一个合法的 DSL 程序。

不难看出，Scythe 中的“孔”与 Trinity 中的编码变量的本质是相同的，不同之处在于 Scythe 将被查询表（包括中间的临时表）和过滤条件看成抽象查询的“孔”，数据表上的相关操作，如 Select、Join 等，则作为抽象查询的“骨干”，一次迭代要预先生成多个抽象查询。Trinity 则将 DSL 中的函数和函数的参数全部进行编码，表示程序的 k-叉树上的所有节点均对应一个编码变量，通过在编码变量上定义约束来保证 DSL 程序形式上的正确。

由于 Trinity 采用完全 k-叉树式结构表示 DSL 程序，因此其编码变量增长速度同样是指数阶的。从表 4 可以看出，Trinity 综合性能并未比 Scythe 有本质的提高，但由于使用成熟的约束求解器取代遍历搜索，其合成成功率为 66%，平均合成时间为 724 s，较 Scythe 有小幅提升。

SQUARES 采用的合成方案与 Trinity 基本相同，关键的不同是 SQUARES 采用链式结构[21]表示 DSL 程序，与 Trinity 相比，编码变量增长速度由指数阶降至多项式阶，大幅降低了约束求解的复杂度，综合性能较 Trinity 有较大提升，平均合成成功率提升至 80%，平均合成时间减少至 251 s。本文提出的 PSR-SQUARES 在 SQUARES 中增加了一个以深度神经网络为核心的程序空间约简器。对于给定的<被查询表，查询结果>示例，神经网络预测该示例与 DSL 中产生式规则的相关性，约简器根据神经网络的预测结果，删除相关性弱的合成规则，减少无关候选 DSL 程序的生成和验证，以提升程序合成的效率。从表 4 可以看出，PSR-SQUARES 的平均合成的成功率由 SQUARES 的 80%进一步提升至 89%，平均合成时间由 SQUARES 的 251 s 降低至 130 s。

综上所述，与 Scythe、Trinity 和 SQUARE 相比，PSR-SQUARES 的合成成功率分别提升了 29%、23%和 9%，平均合成时间分别降低了 763 s（85%）、

594 s（82%）和 121 s（48%），充分证明了 PSR-SQUARES 的有效性。

5 结束语

程序合成是人工智能领域研究者希望攻克的重大问题，SQL 逆向合成是程序合成在数据分析领域的一个分支，其目标是使没有数据库技术背景的数据分析人员不受数据库查询语言限制，便利地访问数据库，实现数据查询或数据转换操作，具有较强现实意义和应用价值。

在诸多 SQL 逆向合成系统中，马里兰大学研究团队开发的 SQUARES 最具代表性，该系统采用“编码”式合成策略，在大量的测试问题中取得了较理想的合成效果。然而，SQUARES 程序空间随着枚举轮数的增加而快速增长，对于一些复杂的合成问题，其合成效率及合成成功率依然较低。针对这个问题，本文提出了一个基于程序空间约简器的 SQL 逆向合成系统 PSR-SQUARES。该系统在 SQUARES 基础上增加了以深度神经网络为核心的程序空间约简器。通过训练后的神经网络可以对与目标程序相关的产生式规则进行预测，在预测结果基础上，约简器采用了末 N 位淘汰策略，删除与目标程序无关的合成规则，减少无关程序的生成和验证，提升了 SQUARES 系统合成效率。

在相同实验和测试条件下，将 PSR-SQUARES 与 SQUARES、Scythe 和 Trinity 等代表性 QRE 系统进行实验对比，实验结果表明 PSR-SQUARES 的综合性能不同程度地优于其他合成系统。

SQL 逆向合成是一项极具挑战性的任务，目前该领域的研究尚处起步阶段，包括本文所提 PSR-SQUARES 在内的合成系统依然存在一些问题亟待解决，如目标语句的偏好选择、目标语句优化、搜索效率提升、用户真实意图理解等，这些问题都是未来研究的主要方向。

参考文献：

[1] PARISOTTO E, MOHAMED A R, SINGH R, et al. Neuro-symbolic program synthesis[J]. arXiv Preprint, arXiv: 1611.01855, 2016.

[2] SOLAR-LEZAMA A. Program synthesis by sketching[D]. Berkeley: University of California, 2008.

[3] CHEN X, LIU C, SHIN R, et al. Latent attention for if-then program synthesis[J]. arXiv Preprint, arXiv: 1611.01867, 2016.

[4] GU X D, ZHANG H Y, KIM S. Deep code search[C]//Proceedings of the 40th International Conference on Software Engineering. New York: ACM Press, 2018: 933-944.

[5] LI C T, TARLOW D, GAUNT A L, et al. Neural program lattices[C]//Proceedings of the International Conference on Learning Representations (ICLR). [S.l.]: OpenReview, 2017: 1-17.

[6] DEVLIN J, UESATO J, BHUPATIRAJU S, et al. RobustFill: neural program learning under noisy I/O[C]//Proceedings of the 34th International Conference on Machine Learning. New York: ACM Press, 2017: 990-998.

[7] GULWANI S, POLOZOV O, SINGH R. Program synthesis[J]. Foundations and Trends in Programming Languages, 2017, 4(1/2): 1-119.

[8] DAS-SARMA A, PARAMESWARAN A, GARCIA-MOLINA H, et al. Synthesizing view definitions from data[C]//Proceedings of the 13th International Conference on Database Theory. New York: ACM Press, 2010: 89-103.

[9] TRAN Q T, CHAN C Y, PARTHASARATHY S. Query by output[C]//Proceedings of the 2009 ACM SIGMOD International Conference on Management of data. New York: ACM Press, 2009: 535-548.

[10] TRAN Q T, CHAN C Y, PARTHASARATHY S. Query reverse engineering[J]. The VLDB Journal, 2014, 23(5): 721-746.

[11] LI H, CHAN C Y, MAIER D. Query from examples[J]. Proceedings of the VLDB Endowment, 2015, 8(13): 2158-2169.

[12] ZHANG S, SUN Y Y. Automatically synthesizing SQL queries from input-output examples[C]//Proceedings of 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). Piscataway: IEEE Press, 2014: 224-234.

[13] WANG C L, CHEUNG A, BODIK R. Synthesizing highly expressive SQL queries from input-output examples[C]//Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York: ACM Press, 2017: 452-466.

[14] WANG C L, CHEUNG A, BODIK R. Interactive query synthesis from input-output examples[C]//Proceedings of the 2017 ACM International Conference on Management of Data. New York: ACM Press, 2017: 1631-1634.

[15] TAN W C, ZHANG M H, ELMELEEGY H, et al. Reverse engineering aggregation queries[J]. Proceedings of the VLDB Endowment, 2017, 10(11): 1394-1405.

[16] TAN W C, ZHANG M H, ELMELEEGY H, et al. REGAL$^+$[J]. Proceedings of the VLDB Endowment, 2018, 11(12): 1982-1985.

[17] KHURANA K, HARITSA J R. UNMASQUE: a hidden SQL query extractor[J]. Proceedings of the VLDB Endowment, 2020, 13(12): 2809-2812.

[18] REHMAN M S, HUANG S L, ELMORE A J. A demonstration of RELIC[J]. Proceedings of the VLDB Endowment, 2021, 14(12): 2795-2798.

[19] QIN X, CHAI C, LUO Y, et al. Interactively discovering and ranking desired tuples by data exploration[J]. The VLDB Journal, 2022, 31(4): 753-777.

[20] MARTINS R, CHEN J, CHEN Y J, et al. Trinity: an extensible synthesis framework for data science[J]. Proceedings of the VLDB Endowment, 2019, 12(12): 1914-1917.

[21] ORVALHO P, TERRA-NEVES M, VENTURA M, et al. SQUARES: a SQL synthesizer using query reverse engineering[J]. Proceedings of the VLDB Endowment, 2020, 13(12): 2853-2856.

[22] BAIK C, JIN Z J, CAFARELLA M, et al. Duoquest: a dual-specification system for expressive SQL queries[C]//Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2020: 2319-2329.

[23] TAKENOUCHI K, ISHIO T, OKADA J, et al. PATSQL: efficient synthesis of SQL queries from example tables with quick inference of projected columns[J]. arXiv Preprint, arXiv: 2010.05807, 2020.

[24] ZHOU X Y, BODIK R, CHEUNG A, et al. Synthesizing analytical SQL queries from computation demonstration[C]//Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. New York: ACM Press, 2022: 168-182.

[25] BALOG M, GAUNT A L, BROCKSCHMIDT M, et al. DEEPCODER: learning to write programs[J]. arXiv Preprint, arXiv: 1611.01989, 2016.

[26] 顾斌, 于波, 董晓刚, 等. 程序智能合成技术研究进展[J]. 软件学报, 2021, 32(5): 1373-1384.
GU B, YU B, DONG X G, et al. Intelligent program synthesis techniques: literature review[J]. Journal of Software, 2021, 32(5): 1373-1384.

[27] 袁胜浩, 杨志斌, 张博林, 等. 同步语言多线程代码生成的语义保持证明方法[J]. 计算机学报, 2020, 43(11): 2216-2226.
YUAN S H, YANG Z B, ZHANG B L, et al. A semantic preservation proving method of multi-threaded code generation for synchronous language[J]. Chinese Journal of Computers, 2020, 43(11): 2216-2226.

[28] REED S, DE-FREITAS N. Neural programmer-interpreters[C]//Proceedings of the 4th International Conference on Learning Representations (ICLR). [S.l.]: OpenReview, 2016: 1-13.

[29] GUO Z, JAMES M, JUSTO D, et al. Program synthesis by type-guided abstraction refinement[J]. Proceedings of the ACM on Programming Languages, 2019, 4: 1-28.

[30] LE H, WANG Y, GOTMARE A D, et al. CodeRL: mastering code generation through pretrained models and deep reinforcement learning[J]. arXiv Preprint, arXiv: 2207.01780, 2022.

[31] GULWANI S, MARRON M. NLyze: interactive programming by natural language for spreadsheet data analysis and manipulation[C]//Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. New York: ACM Press, 2014: 803-814.

[32] SHRIVASTAVA D, LAROCHELLE H, TARLOW D. Learning to combine per-example solutions for neural program synthesis[J]. arXiv Preprint, arXiv: 2106.07175, 2021.

[33] HUANG G, LIU Z, MAATEN L V D, et al. Densely connected convolutional networks[C]//Proceedings of 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Piscataway: IEEE Press, 2017: 2261-2269.

[作者简介]

窦全胜（1971− ），男，黑龙江大庆人，博士，山东工商学院教授，主要研究方向为机器学习、数据挖掘、知识工程与知识处理、智能理论与方法等。

张顺（1992− ），男，山东潍坊人，山东工商学院硕士生，主要研究方向为人工智能、程序合成等。

潘浩（1999− ），男，山东泰安人，山东工商学院硕士生，主要研究方向为人工智能、深度学习、程序合成算法等。

王荟贤（1996− ），男，山东烟台人，山东工商学院硕士生，主要研究方向为人工智能、程序合成、机器学习等。

唐焕玲（1970− ），女，山东烟台人，博士，山东工商学院教授，主要研究方向为机器学习、人工智能、数据挖掘等。