カラム数の大きなテーブルに対する 入出力例からの SQL クエリの合成

竹之内 啓太[1,a)]　岡田 譲二[1,b)]　坂田 祐司[1,c)]

概要：入出力の例から自動的にプログラムを合成する Programming by Example (PbE) は，近年盛んに研究されている領域である．我々の研究チームでは，プログラム合成技術のシステム開発現場への導入を目指し，入出力となるテーブルの例から SQL クエリを合成する技術の研究開発を行っている．入出力例から SQL クエリを合成する先行研究はいくつか存在しているものの，いずれの技術も入出力例となるテーブルのカラム数が大きくなるにつれて，計算時間が爆発的に増加するという課題を抱えている．本研究では，入出力となるテーブルが大きなカラム数をもつ場合であっても，効率的に SQL クエリを合成するアルゴリズムを提案する．SQL クエリの一部を順番に埋めていく一般的なアプローチを採用したうえで，計算量に課題のあったカラムとカラムの組み合わせ計算に工夫がある．ケーススタディでは，実際のバッチ処理システムに存在している SQL クエリ 8 件を対象として提案手法を適用し，7 件が 1 秒以内の合成に成功することを確認した．

Synthesizing SQL Queries Supporting Large Column Tables from Input-Output Examples

Abstract: Programming by Example (PbE) is a technique to automatically synthesize programs from input-output (I/O) examples. PbE has received significant attention from researchers in recent years. Our research team aims to introduce program synthesis techniques to our system development, and we are doing research and development for synthesizing SQL queries from I/O example tables. Although there exists prior work to synthesize SQL queries from I/O examples, the execution time increases significantly as the column sizes become large. In this paper, we propose a novel algorithm to synthesize SQL queries from I/O tables even when they have large column sizes. Our approach adopts a general way that fills each part of SQL queries in order and introduces a new strategy for mitigating the cost that enumerates corresponding column pairs. Our case study shows that our algorithm is able to synthesize seven out of eight queries within a second that are extracted from our batch processing system.

Keywords: Program Synthesis, Programming by Examples, SQL, Domain Specific Language(DSL)

1. はじめに

プログラム合成は，人間が与えた高レベルな仕様をもとに，プログラムを自動生成する技術である [1]．なかでも，Programming by Example (PbE) と呼ばれる合成技術は，実現すべき入出力の例を人間が与えることで，それを満たすプログラムを自動的に生成する手法である [2]．PbE は，近年盛んに研究されている領域であり，Microsoft FlashFill[3] に代表されるように実用化されて始めている技術でもある．

システム開発において，リレーショナルデータベースを操作するため SQL クエリを使用することは一般的である．弊社で開発が行われている Web アプリケーションやバッチ処理システム等においても，リレーショナルデータベースの利用はごく一般的である．それにともない，開発工程では膨大な数の SQL クエリの作成に人や時間が費やされている．システム開発において膨大な数の作成が行われる SQL クエリであるが，その品質を担保するため，多くの場

[1] 株式会社 NTT データ
NTT DATA Corporation
[a)] Keita.Takenouchi@nttdata.com
[b)] Joji.Okada@nttdata.com
[c)] Yuji.Sakata@nttdata.com

合 SQL クエリごとの単体試験が実施される．単体試験では，作成した入力テーブルに対して SQL クエリを実行し，その実行結果が期待値のテーブルと一致することが確認される．すなわち， SQL クエリごとに，その入出力例となるテーブルのデータが存在することが一般的である．我々の研究チームではこの点に着目し，試験時に作成される入出力テーブルを入出力例として PbE の技術を適用することで，SQL クエリを自動的に生成することを目指している．このような開発プロセスが可能となれば，システム開発の飛躍的な生産性の向上が見込まれる．

SQL クエリの合成を実際の開発プロセスに導入するには，手法として以下の要件が重要であると考えている．

(1) 実際に開発される SQL クエリの多くを合成できること．
(2) 短時間 (数秒以内) で合成が完了すること．

(1) について，一般的な SQL クエリは高い表現力を持ち，さまざまな構造をもつ可能性がある．一方で，業務システムのために作成される SQL クエリは単純な構造を持つものが多くを占めると仮定し，本研究ではそのようなボリュームゾーンとなる SQL クエリをカバーする合成手法を目指す．(2) について，Microsoft FlashFill[3] で実現されているように，PbE は，合成結果に応じてユーザが入出力例を更新するインタラクティブな使い方が有効であることが知られている．この点を考慮し，本研究では合成が数秒以内に完了するような手法を目指す．一方で，既存の入出力例から SQL クエリを合成する手法では，カラム数が増加するにつれて計算時間が爆発的に増加するという課題をもっており，(1) や (2) の要件を満たしておらず，実際の開発現場への導入は難しかった．

そこで本研究では，大きなカラム数をもつテーブルを入出力例として， SQL クエリを合成するアルゴリズムを提案する．既存手法では合成アルゴリズムの序盤で行っていたカラムとカラムの対応関係の計算を合成手順の最後に行うことにより，合成にかかる計算量がテーブルのカラム数に比例する程度に抑えられているのが特徴である．本アルゴリズムでは，一般的な SQL クエリのサブセットとなる中間言語を定義し，それを用いて合成を行う．合成時に効率的な探索を実現するため，この中間言語の構文は一般的な SQL より制限されたものとなっている．一方で，実際の開発で頻繁に利用される集約関数や `ORDER BY` 句をサポートするなど，SQL クエリとしての表現力を保っている．

ケーススタディとして，実際のバッチ処理システムから抽出した SQL クエリ 8 件とその試験データを用いて，提案手法の実用化に向けた検証を行った．既存手法では，いずれの SQL クエリも 300 秒以内に合成できなかった一方で，提案手法では 7 件を 1 秒以内に合成できることを確認した．さらに，合成に成功した SQL クエリは実際の SQL クエリと同程度の可読性をもつことも分かった．

表 1 実際のシステムにおけるテーブルのカラムの大きさの値.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
5	11	15	19.5	22	111

以降，2 章で先行研究とその課題について，3 章で提案手法の概要や使用する中間言語について述べる．4 章で合成アルゴリズム，5 章で実施したケーススタディについて説明する．さいごに，6 章でまとめと今後の展望を述べる．

2. 先行研究とその課題

入出力例を用いて SQL クエリを合成する先行研究はいくつか存在する．`SQLSynthesizer`[4] や `SCYTHE`[5] は，入出力例となるテーブルから SQL クエリを合成する手法である．`SQLSynthesizer`[4] は，合成する SQL クエリとして，`SELECT` 句や `WHERE` 句のほか，`GROUP BY` 句や `ORDER BY` 句など実際に高い頻度で使用される構文をサポートしている．それに加えて `SCYTHE`[5] では，サブクエリや `UNION`, `LEFT JOIN` など，さらに高い表現力をもつ SQL クエリの合成がサポートされている．また，`MORPHEUS`[6] は，データ分析の前処理のため，R 言語を用いて，データフレームを操作するプログラムを合成する手法である．SQL クエリより一般的な操作を合成することを目的とした手法であるが，評価実験において，`SQLSynthesizer` より高い SQL クエリの合成能力をもつことが示されている．

これらの既存手法には共通の課題が存在する．それは，入力となるテーブルのカラム数に対して，合成にかかる計算量が爆発的に増加することである．たとえば `SCYTHE` では，以下のような「抽象クエリ」という，通常の SQL クエリの一部を抽象化した中間言語を用いて合成を行う．クエリ中の □ 記号が `WHERE` 句の条件式を抽象化した述語である．

```
SELECT c1, c2, c3 FROM TBL1 WHERE □
```

合成の手順として，まずはじめに SQL クエリの構造を表現した抽象クエリの構築を行う．そして，抽象クエリ中の □ に入るべき適切な条件式を埋め，具体的な SQL クエリを得る．このような 2 段階の手順により，複雑なサブクエリの構造をもつ SQL クエリを合成することが可能である．しかしながら，□ 以外の部分は抽象クエリの段階で決定されるため，`SELECT` 句によって射影されるカラムのすべての並び `[c1, c2, c3]`,..., `[c3, c2, c1]` を列挙する必要がある．上記のクエリの例ではカラム数は 3 であるが，たとえば入力テーブルのカラム数が 10 である場合，まずはじめに $10! = 3,628,800$ 通りの並びに対応した抽象クエリを列挙する必要がある．なお，`SCYTHE` 以外の先行研究でも，このような入力テーブルのカラム数のスケーラビリティの課題は同様である．一方で実際のシステムでは，これらの手法が扱える範囲を超える大きなテーブルの存在が一般的であり，既存手法を実際の開発プロセスに導入する

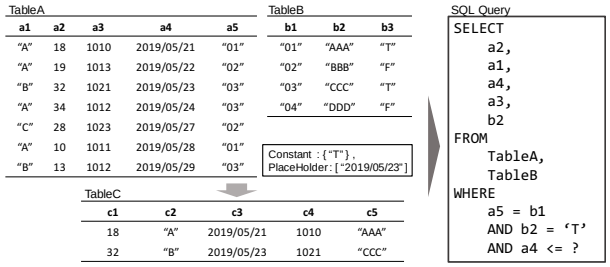

図 1 提案手法の例. 入出力例から SQL クエリを合成.

障壁となっている．実際，表 1 は 5 章の評価実験で用いたシステムに含まれる 163 のテーブルのカラム数の統計量である．中央値で 15 であり，既存手法をそのまま適用するのが難しいことが分かる．

3. 提案手法

本章では，はじめに手法の概要を示し，本研究で解くべき問題を定義したのち，合成時に使用する中間言語について述べる．

3.1 概要

実際のシステムに組み込まれている SQL クエリは，実行時に外部の値を埋め込むためのプレースホルダ (?として記述される) をもつことが一般的である．そのため本手法では，入出力となるテーブルとプレースホルダに入る値を入力とし，それを満たす SQL クエリを合成する．プレースホルダへの対応は，先行研究には見られない独自の提案である．なお，SQL クエリで使用される定数はユーザによって与えられるものとする．

図 1 の例では，入力テーブル $TableA, TableB$ とプレースホルダに埋め込まれる値“2019/05/23”がリストとして与えられ．その結果として得られる出力値がテーブル $TableC$ として与えられている．また，合成時に使う定数として“T”という文字列型のリテラルが与えられている．これらの情報をもとに，図中の SQL クエリを合成する手法を提案する．この例は小規模なテーブルを入出力例としているが，たとえば，$TableA$ や $TableC$ のカラム数が 30 であっても，短時間で合成することが可能である．なお，実際の SQL クエリに頻繁に使用される GROUP BY 句による集約計算や ORDER BY 句にも対応している．

3.2 問題の定義

本手法では，ユーザからの入力として (I, T_{out}, C) の 3 つ組が与えられるものとする．ただし，$I = (\{T_1, \ldots, T_n\}, P)$ は入力テーブルの集合とプレースホルダに入る値のリストの組であり，T_{out} は出力テーブル，$C = \{v_1, \ldots, v_k\}$ は WHERE 句の条件式に用いられる定数の集合である．$T_1, \ldots, T_n$ や T_{out} は複数のカラムをもつテーブルであり，各カラムは属性として名前と型の情報をもつ．カラムの型として，文字型，数値型，日付型が存在する．また，プレースホルダ P や定数 C はこれらの型に加えそれぞれの集合型が存在する．なお，テーブルはレコードの順序に意味があるものとする．本研究では，この 3 つ組 (I, T_{out}, C) を入力として，$q(I) = T_{out}$ となる SQL クエリ q を 1 つだけ求めるアルゴリズムを提案する．ただし，SQL クエリ q の WHERE 句の条件式に用いられる定数はすべて C に含まれるものとする．

入出力例のほかに合成のヒントとなる定数の集合 C が与えられる制約は SCYTHE[5] と同様である．これにより，探索空間を大幅に削減することができるうえ，ユーザの意図をより正確に反映した SQL クエリを合成することにつながる．先行研究 [5] では，Stack Overflow 上に投稿される質問において，質問者によって SQL クエリ内で使用すべき定数が示されることが一般的であることが言及されている．また，我々の研究チームでも開発者にヒアリングを行い，開発者が使用する定数を事前に知っていることは自然であるという見解が得られており，この制約は実際の開発プロセスに導入するにあたって問題にならないと判断している．

```
<query> := SELECT <columns> FROM <tables>
           <wherePart>?
           <groupbyPart>?
           <orderbyPart>?
<columns> := <column>+ | *agg
<column> := col_name | <funcCol>
<funcCol> := <func> ( col_name )
<func> := MAX | MIN | COUNT
<tables> := table_name+
<wherePart> := WHERE <joinConds> AND <whereConds>
<joinConds> := <joinCond> (AND <joinCond>)*
<joinCond> := col_name = col_name
<whereConds> := <whereCond> (AND <whereCond>)*
<whereCond> := col_name <whereOp> (const|placeholder)
<whereOp> := <= | >= | < | > | = | IN
<groupbyPart> := GROUP BY (col_name|NIL)
<orderbyPart> := ORDER BY col_name (ASC|DESC)
```

図 2 中間言語の構文

3.3 中間言語

近年のプログラム合成では，合成のための部品を中間表現（ドメイン固有言語など）を用いて表現し，適切な部品の組み合わせを探索する手法が主流である [1]．任意のプログラムを合成する場合と比較し，限られた部品の組み合わせのみを探索することになるため，探索空間を大幅に削

減することにつながる．本研究でも，一般的な SQL クエリのサブセットとなる中間言語を定義し，そのうえで合成を行う．さらに，合成が成功した後に中間言語から実際の SQL クエリへ変換する設計にすることで，データベースマネジメントシステムに依存した SQL の方言へ対応する狙いもある．

本手法で用いる中間言語の構文を図 2 に示す．<> で囲まれた記号は文法の非終端記号であり，*col_name* は具体的なカラム名，*table_name* は入力テーブル名，*const* は定数，*placeholder* はプレースホルダ (?) を意味する．*agg や NIL は独自に導入した特殊な記号であり，説明は後述する．この構文によって表現される SQL クエリは，SQLSynthesizer[4] と同様，GROUP BY 句や ORDER BY 句など一般的に高い頻度で使用される機能をサポートしている．一方で，中間言語の構文のうち，以下の 2 点は本研究独自の特徴的な部分である．

- SELECT 文によって射影されるカラム <columns> として，特殊なワイルドカード *agg をもつ．
- WHERE 句に含まれる条件式 <whereCond> が厳しく制限されている．

以下，それぞれについて述べる．

3.3.1 特殊なワイルドカード

SELECT 文によって射影されるカラムを意味する <columns> を左辺にもつ規則では，右辺としてワイルドカード *agg が入ることが許容されている．この記号の意味を説明する．

まず，SQL クエリに GROUP BY 句が存在しない場合，ワイルドカード *agg は一般的な SQL で用いられるワイルドカード * と同じ意味をもつ．つまり，入力テーブルのすべてのカラムがそのまま射影される．

SQL クエリに GROUP BY 句が存在する場合，そのキーとなるカラムと，各カラムに集約関数が適用された結果となる集約カラムが追加される．この例を図 3 に示す．この例では，入力テーブルが *TableD*, 実行結果が *TableE* であり，入力テーブルと実行結果の間に一時的に作成されるテーブルが *TableTmp* である．中間テーブル *TableTmp* の MAX(a1) より右側のカラムが集約カラムであり，SQL クエリの GROUP BY 句で指定されているカラム a1 によってレコードがグループ化され，各集約関数 MAX, MIN, COUNT を適用した結果が格納される．中間テーブルから重複するレコードを取り除いたのが実行結果 *TableE* である．なお，すべてのカラムを 1 つのグループとする SQL クエリを表現するため，グループ化のキーとして NIL が許されている．このときの実行結果は常に 1 レコードとなり，すべてのレコードを 1 グループとしたときの集約結果の値をもつカラムのみからなる．

ワイルドカード *agg は合成過程の SQL クエリ中でのみに使用される記号であり，合成結果となる SQL クエリに含まれることはない．この記号を構文に導入することで，適切な集約関数の選択の問題を，射影するカラムの選択の問題として解いている．たとえば，SELECT MAX(A) のような SQL クエリを構築するために，SELECT *agg の実行結果を用いる．本手法では，SELECT 句によって射影されるカラムを効率的に求めることが可能であるため，このような方法は適用すべき集約関数を効率的に求めることにつながる．

TableD

a1	a2
A	12
B	33
A	18
C	48
B	5

TableTmp

a1	MAX(a1)	MIN(a1)	COUNT(a1)	MAX(a2)	MIN(a2)	COUNT(a2)
A	A	A	2	18	12	2
B	B	B	1	33	5	1
A	A	A	2	18	12	2
C	C	C	1	48	48	1
B	B	B	2	33	5	2

deduplication

TableE

a1	MAX(a1)	MIN(a1)	COUNT(a1)	MAX(a2)	MIN(a2)	COUNT(a2)
A	A	A	2	18	12	2
B	B	B	2	33	5	2
C	C	C	1	48	48	1

SQL Query

```
SELECT *agg
FROM TableD
GROUP BY a1
```

図 3 *agg を用いた SQL クエリの実行の例.

3.3.2 条件式の制約

先行研究と比較し，本研究では WHERE 句に含まれる条件式 <whereCond> が厳しく制限された形をもつ．具体的には，演算子の左辺はカラム名，右辺は定数またはプレースホルダのみが許されている．そのため，以下のような条件式は本手法の対象外である．

(1) OR を用いたもの
(2) サブクエリを用いたもの
(3) カラムとカラムの大小比較（DATE1 > DATE2 など）
(4) IS NOT NULL, IS NULL を用いたもの

ただし，(1) については，演算子として IN が使用可能であるため，たとえば条件 COL = "A" OR COL = "B" は条件 COL IN ("A", "B") として表現可能であり，限定的に OR がサポートされていると言える．また，(2) のサブクエリを用いた条件式は，複数テーブルの JOIN によって書き換えることができる場合があり，サブクエリをもつ SQL クエリと同じ意味のすべての SQL クエリが合成できないわけではない．この制約によって明らかに表現することができない (3) と (4) について，実際のシステムの SQL クエリを構文解析し検証を行った．調査対象は，5 章の評価実験で用いたものと同じシステムから抽出した 1172 件の SQL クエリである．その結果，(3) については 2 件，(4) について IS NOT NULL が 38 件，IS NULL が 57 件存在していた．これらの条件式は多くの SQL クエリに含まれるというわけでないが，無視できるほど少ないわけではないため，今後の課題として検討を進めている．

4. 合成アルゴリズム

提案手法の合成アルゴリズムについて述べる．まずはじめに全体の流れを説明し，その後各手順の詳細を示す．

Algorithm 1 合成アルゴリズムの概要

```
Procedure Synthesize(E, C)
  Input: 入出力例 E = (I, T_out)，定数集合 C
  Output: SQL クエリ
q_0 ← createBaseQuery(E)
for q_1 ∈ buildJoinConds(q_0, E) do
  for q_2 ∈ buildWhereConds(q_1, E, C) do
    for q_3 ∈ buildGroupBy(q_2, E) do
      for q_4 ∈ buildOrderBy(q_3) do
        for q_5 ∈ buildSelect(q_4, E) do
          if CHECK(q_5, E) then
            return q_5
return ⊥
```

4.1 全体の流れ

本手法のアルゴリズムの全体像を Algorithm 1 に示す．関数名が *build* から始まるものは， SQL クエリをもとに新たな SQL クエリの集合を返す関数である．各関数において入出力例 E は枝刈りのために使用され，解となる可能性がない SQL クエリはその時点で出力から除外される．関数 $CHECK$ は，与えられた SQL クエリが入出力例 E を満たす場合に真を返す関数である．

アルゴリズムの手順としては，まずはじめにベースとなる最小の SQL クエリを構築する．その後，枝刈りをしながら `WHERE` 句，`GROUP BY` 句，`ORDER BY` 句，`SELECT` 句の順に， SQL クエリを構築していく．そして，最終的に構築された SQL クエリが入出力例を満たす場合，それを合成結果として返す．以降，各手順について述べる．

手順 1. ベースとなる SQL クエリの構築

まずはじめに，Algorithm 1 の関数 $createBaseQuery$ の処理について説明する．この処理では，以降の手順のベースとなる以下の SQL クエリ q_0 を構築する．

$$q_0 = \texttt{SELECT *agg FROM } T_1, \ldots, T_n$$

この SQL クエリは，入出力例 E に含まれるすべての入力テーブル $T_1, \ldots, T_n$ を `FROM` 句にもち，`SELECT` 句の対象として `*agg` をもつ．この SQL クエリを実行すると，各テーブル $T_1, \ldots, T_n$ がもつレコード集合の直積に相当するレコードをもつテーブルが出力となる．この時点では，実行結果は膨大な行数をもつテーブルとなっているが，以降の手順で適切な条件式などが探索され，最終的にはユーザが与えた出力例と等しい出力をもつ SQL クエリが求められる．

手順 2. JOIN のキーの列挙

図 2 の `<joinConds>` にあたる部分を構築する関数 $buildJoinConds$ の処理について説明する．本手法では複合キーによる JOIN をサポートしているため，この処理では JOIN のキーとなるカラム名のペアの集合を列挙する．

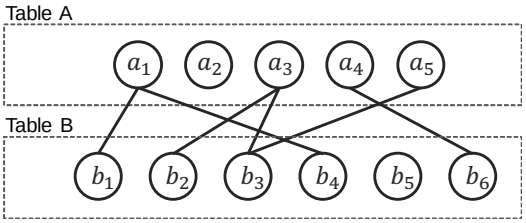

図 4 カラムの対応関係を表した 2 部グラフの例．このグラフの空でないマッチングが複合キーによる JOIN に対応する．

入力テーブルが A, B の 2 つの場合について説明する．カラムのペアの集合の列挙は，図 4 のような 2 部グラフのマッチング（互いに端点を共有しない辺集合）を列挙する問題とみなすことができる．具体的には，テーブル A, B のカラムを頂点，JOIN のキーのペアになりうるカラムの関係を辺とするグラフを考える．1 つのテーブル内のカラム同士が JOIN のキーのペアとなることはないため，このグラフは 2 部グラフとなる．さらに，1 つのカラムが複数の JOIN のキーのペアにはならないと仮定すると，2 部グラフのマッチングを求めることが，JOIN のキーとなるカラムのペアの集合を求めることに対応する．以降，テーブル A, B のカラムをそれぞれ a_i，b_j とする．

まず，2 部グラフを構築する．A, B のカラム名を頂点とするグラフに対し，JOIN のキーのペアとなりうる a_i と b_j のあいだに辺を追加していく．ただし，a_i と b_j はカラムの属性として同じ型をもつものとする．具体的には，以下の SQL クエリを実行したとき，実行結果のテーブルのレコード数が，ユーザの与えた出力例のテーブルのレコード数以上の場合のみ，a_i と b_j のあいだに辺を追加する．

$$\texttt{SELECT *agg FROM } A, B \texttt{ WHERE } a_i = b_j$$

本アルゴリズムでは，これ以降の手順で，実行結果のテーブルのレコード数が増加するような SQL クエリが構築されることはない．そのため，この時点で SQL クエリの実行結果が出力例より小さいレコード数をもつ場合，そのキーによる JOIN をした時点で合成が成功することはない．このように合成過程の SQL クエリの実行結果のレコード数を考慮した枝刈りを以降の各手順で行っており，「実行結果のレコード数による枝刈り」と呼ぶことにする．

次の手順として，構築した 2 部グラフの空でないマッチングを列挙し，その要素を JOIN のキーとして `WHERE` 句の条件式に付加する．その結果，以下のような SQL クエリが得られる．ただし，JOIN のキーのペアの個数を N ，カラムのペアを $(a_{k_1}, b_{l_1}), \ldots, (a_{k_N}, b_{l_N})$ とする．

$$\texttt{SELECT *agg FROM } A, B$$
$$\texttt{WHERE } a_{k_1} = b_{l_1} \texttt{ AND } \ldots \texttt{ AND } a_{k_N} = b_{l_N}$$

このような SQL クエリに対し，実行結果のレコード数による枝刈りを行い，残ったもののみを本手順の出力とする．

入力テーブルが 1 つの場合は，JOIN のための条件式は作られないものとする．また，入力テーブルが 3 つ以上の場合は，上述の手順を再帰的に実行すればよい．たとえば，入力テーブルが A, B, C の 3 つであったとき，はじめにテーブル A と B の JOIN を行い，その結果テーブル A' が得られると考え，つぎにテーブル A' と C の JOIN に対するキーのペアを探索すればよい．

手順 3．WHERE 句の条件式の列挙

図 2 の <whereConds> にあたる部分を構築する関数 $buildWhereConds$ の処理について説明する．本手順では，まず WHERE 句の条件式 <whereCond> を求め，それらを AND でつないだ条件式をもつ SQL クエリを列挙する．ただし，本手順ではある制約を設けている，その制約とは，ユーザによって与えられた定数やプレースホルダは，SQL クエリの WHERE 句の条件式として必ず 1 度だけ使用される，というものである．つまり，使用されない定数がユーザによって与えられることはなく，複数回使用される定数はそれぞれ別の定数として与えられることを仮定している．なお，プレースホルダに入る具体値はユーザによって与えられるが，それは合成時には不変であり，定数と同じ意味をもつ．以下では，簡単のためプレースホルダを定数として扱って説明する．

まずはじめに <whereCond> にあたる条件式を求める．上述の制約に加えて，<whereCond> はただ 1 つの定数をもつことを考慮すると，SQL クエリは定数ごとに <whereCond> にあたる条件式を 1 つだけをもつことになる．つまり，定数の個数を N とすると，SQL クエリは AND でつながれた N 個の <whereCond> にあたる条件式をもつ．個々の <whereCond> の求め方を説明する．入力テーブルを A とし，WHERE 句に使用される比較演算子の集合を O とする．<whereCond> は演算子の左辺にカラム名，右辺に定数をもつ．まず，右辺の定数をユーザから与えられた値 v_i として固定する．つぎに，カラムの型の整合性を保つ比較演算子 $op_j \in O$ とカラム名 $c_k \in A$ を列挙する．これらの 1 つの組み合わせを条件式として，その時点の SQL クエリに付加すると以下が得られる．

SELECT *agg FROM ... WHERE ... AND c_k op_j v_i

この実行結果のレコード数による枝刈りを行い，残った条件式 “c_k op_j v_i” を，v_i に対する条件式集合の要素とする．この操作をすべての v_i に行い，条件式の集合族を求める．

つぎに，条件式を AND でつないだ条件式をもつ SQL クエリを列挙する．前の手順で得られた集合族の直積を求めると，その要素は条件式の集合となり，対応する SQL クエリとして以下が得られる．ただし，$c_1, \ldots, c_N$ や $op_1, \ldots, op_N$ は，それぞれ $v_1, \ldots, v_N$ に対応するカラム名と比較演算子である．

SELECT *agg FROM ... WHERE ...
AND c_1 op_1 v_1 AND ... AND c_N op_N v_N

この実行結果のレコード数による枝刈りを行い，残ったものを本手順の出力とする．

手順 4．GROUP BY 句の列挙

図 2 の <groupbyPart> にあたる部分を構築する関数 $buildGroupBy$ の処理については，GROUP BY 句のキーとして，入力テーブルの各カラムまたは NIL を列挙する．他の手順と同様，対応する SQL クエリを構築し，実行結果による枝刈りを行うが，これ以降の手順では，SQL クエリの実行結果のレコード数が変化することがないため，現時点での実行結果が出力例と一致するもののみ本手順の出力とする．

手順 5．ORDER BY 句の列挙

図 2 の <orderbyPart> にあたる部分を構築する関数 $buildOrderBy$ の処理については，レコードのソートのキーとなるカラムとして，集約カラムを含むすべてのカラムを列挙する．また，GROUP BY 句をもたない SQL クエリも列挙に加える．そして，ソートの昇順・降順を指定する ASC または DESC それぞれをもつ SQL クエリを構築する．

手順 6．SELECT 句の列挙

本手順では，図 2 の <columnsPart> にあたる部分を構築する関数 $buildSelect$ の処理について説明する．SQL クエリの実行結果のテーブルを出力例と比較することで，SELECT 句によって射影される適切なカラムを求める．まず，現時点での *agg をもつ SQL クエリの実行結果のテーブル T_{tmp} を得る．なお，手順 4 の枝刈りによって，テーブル T_{tmp} と出力例 T_{out} のレコード数が一致することが保証されている．そして，T_{out} の各カラムについて，同じ値の列をもつ T_{tmp} のカラムを求める．T_{out} のすべてのカラムについて，対応する T_{tmp} のカラムが見つかった場合，その対応関係を射影した SQL クエリが本アルゴリズムの解となる．

この手順では，事前に各カラムのハッシュ値を計算しておくことで，カラムの大きさによらず，カラム同士が同じ値をもつかどうかの判定をおよそ定数時間で行うことができる．テーブル T のカラム数を $|T|$ とすると，本手順の計算量は $O(|T_{tmp}| \times |T_{out}|)$ となる．先行研究では，カラムとカラムの対応関係を階乗の組み合わせを試す必要があり，カラム数のスケーラビリティに問題があったが，本手法では入出力例のカラム数に高々比例する程度の計算量で

表 2　ケーススタディの対象となる SQL クエリの一覧.

ID	#InCols	#OutCol	SQL クエリの特徴
#1	[8]	1	MAX
#2	[8]	2	BETWEEN, ORDER BY
#3	[8]	2	サブクエリ
#4	[13, 17]	10	JOIN, ORDER BY
#5	[31]	4	IN, ORDER BY
#6	[16]	1	COUNT
#7	[15, 12, 20, 18, 28, 20]	1	COUNT, IN, BETWEEN
#8	[13]	1	COUNT

抑えられており，大きなカラム数をもつ場合であっても効率的に計算することができる．

5. ケーススタディ

提案手法の有用性を検証するため，実際のバッチ処理のシステムを用いたケーススタディを行った．本ケーススタディで調査する項目は次のとおりである．

(1) 実際の試験データを入出力として SQL クエリを合成できるか．
(2) 合成によって得られた SQL クエリと実際の SQL クエリに差があるか．

対象システムは，弊社で開発を行っている公共系のバッチ処理システムである．システム全体は Java で書かれており，その一部に SQL が埋め込まれバッチ処理の実行が行われる．このシステムには 1172 件の SQL クエリが存在するが，この中から開発担当者が本システムにおいて典型的である，つまり類似の SQL クエリが多く存在する判断したもののうち，さまざまな特徴をもった 8 件の SQL クエリを合成の対象とした．これらの SQL クエリと同等の意味をもつ SQL クエリが合成できるかどうかを検証するため，各 SQL クエリの入出力例を作成した．その一覧を表 2 に示す．この表の #InCols は入力テーブルのカラム数をリストとして記述したものであり，#OutCol は出力テーブルのカラム数である．なお， #InCols リストの大きさは入力テーブルの数に一致する．たとえば，#4, #7 の入力テーブルの数はそれぞれ 2, 6 である．また，「SQL クエリの特徴」列は実際の SQL クエリに含まれる比較演算以外の要素を挙げている．

それぞれの入出力データの作成には既存の試験データを使用した．ただし，たとえば正解となる SQL クエリでは条件式 <= を使用されるべきであるが，代わりに < が使用されるなど，本来の意図と異なる SQL クエリが合成される場合がある．このように試験データとして存在していた入出力データが，実際に存在する SQL クエリ を合成するのに不十分である場合，意図する SQL クエリが合成されるまで入出力データの更新を繰り返した．そして，実際の SQL クエリを正解として，合成された SQL クエリがそれと同じ意味をもつと判断された時点で合成を完了とした．このような合成結果を見て入出力例をインタラクティブに修正するような実験設定は多くの先行研究 [3], [4], [5] と同様である．なお，合成結果が妥当であるかどうかの判断や例外の発見は現行の開発プロセスのレビューに相当するため，SQL クエリを作成する工程に比べると小さなコストで行うことができると考えている．

表 3　各入出力例からの合成に要した実行時間 (秒).
P は提案手法，Q は既存手法である．

	#1	#2	#3	#4	#5	#6	#7	#8
P	0.59	0.20	0.88	0.14	0.13	0.03	×	0.03
Q	×	×	×	×	×	×	×	×

調査項目 (1) に関しては， SQL クエリ合成の最新手法 SCYTHE[5] を比較対象とし，オープンソースとして公開されている実装[*1]を使用した．SCYTHE はプレースホルダを含む SQL クエリの合成はサポートしていないため，代わりにその具体的を定数として与え合成を行った．つまり，? の代わりに具体的な定数が入る SQL クエリを合成するようにした．また，SCYTHE の仕様上，必要とされる集約関数 (MAX, COUNT など) をヒントとして与えた．なお，提案手法は入力テーブルのレコード数の増加に対して計算量は高々比例する程度であるが，SCYTHE は急激に計算量が増える可能性がある．そのため，大きなレコード数をもつ実際の試験データではなく，10 レコード以下になるように入出力データを新たに作成したうえで合成を行った．

本アルゴリズムでは枝刈りのため，入力テーブルに対する中間言語の実行結果を使用するが，SQL クエリを実際のデータベースに発行するのではなく，中間言語のインタプリタを自ら実装した．これにより，データベースアクセスにかかるオーバーヘッドを削減している．インタプリタや本手法のアルゴリズムは Java を用いてを実装を行った．なお，実行環境として CPU-Intel Xeon 2.20GHz, RAM-8GB のマシンを使用した．

5.1 実際の SQL クエリを合成できるか

各入出力例を用いた合成の実行時間を表 3 に示す．表中の P, Q の行がそれぞれ提案手法，SCYTHE の実行時間 (秒単位) である．また，× はタイムアウトを意味しており，300 秒以内に合成が終了しなかったものである．合成結果に応じて入出力データを更新したものに関しては，最終的な SQL クエリの合成に要した時間のみ示している．

結果として，提案手法では入出力例 #7 を除く 7 件に対して 1 秒以内に合成が成功した．その中で，合成結果が実際の SQL クエリがもつ意味と異なり，入出力データを更新したものは次のとおりである．入出力例 #2, #4, #5 は，最初は ORDER BY のない SQL クエリが合成されたため，

[*1] https://github.com/Mestway/Scythe

#4
```
SELECT
    B.c1, B.c2, B.c3,
    B.c4, B.c5, B.c6,
    B.c7, B.c8, B.c9,
    B.c10
FROM
    TBL1 AS A,
    TBL2 AS B
WHERE
    A0.c3 = B0.c1
    AND A0.c1 = ?
    AND A0.c2 <= ?
ORDER BY
    A0.c2 DESC
```

#2
```
SELECT
    c1, c2
FROM
    TBL1
WHERE
    c1 >= ?
    AND c1 <= ?
ORDER BY
    c1 ASC
```

#3
```
SELECT
    MIN(c1), c2
FROM
    TBL1
WHERE
    c2 = '01'
    AND c1 > ?
GROUP BY
    c2
```

#3 (original)
```
SELECT
    c1, c2
FROM
    TBL1
WHERE
    c1 = (
        SELECT
            MIN(c1)
        FROM
            TBL1
        WHERE
            c2 = '01'
            AND c1 > ?
    )
```

図 5 合成された SQL クエリの例．識別子名や定数値は実際のものから置換されている．

入力テーブルのレコードの順序を変えて再度合成を実行した．また，#4 は WHERE 句に含まれる条件式が <= であるべきだが，>= として合成されたため，入力テーブルにレコードを追加することで合成の意図の明確化を行った．これらの作業には大きな手間はかからず，試験データを作成する能力のある開発者であれば，合成のための入出力データを用意するのは難しくないと考えている．以上のように，入力テーブルのカラム数が 10 を超えるような場合でも 1 秒以内に高速に合成が完了するうえ，使用するデータの用意に大きな手間がかからないことから，本手法の開発現場への導入に向けた実現性が十分に確認できたと考えている．300 秒以内に合成が完了しなかった #7 について原因を分析したところ，この入出力例では大きな 6 つのテーブルの JOIN を合成する必要があるが，そのキーの計算 (2 部グラフのマッチングの列挙) で組み合わせ爆発を起こしていたことが分かった．この課題に対しては，アルゴリズムの改善や制約の追加などの対策を検討している．

一方で，既存手法 SCYTHE では，すべての入出力例に対してタイムアウトとなった．これは，テーブルのカラム数が増加するにつれて計算量が爆発的に増加することが原因であると考えられる．入出力テーブルの規模がもっとも小さく条件式も簡単な #1 について，タイムアウトを設けずに合成を実行したところ約 34 分で合成が完了した．その他の，それ以上の複雑さをもつ入出力例に対しては，合成に膨大な時間がかかることが推測され，既存手法を開発現場へ導入することは難しいといえる．

5.2 実際の SQL クエリとの比較

本ケーススタディでは，元の SQL クエリと同じ意味をもつ SQL クエリが合成された時点で合成を完了としているため，合成が成功した SQL クエリは元の SQL クエリと同様の意味をもつ．しかしながら，本手法では図 2 で定義される，一般的な SQL クエリのサブセットとなる中間言語を用いて合成を行っているため，構文として異なる SQL クエリが合成結果として得られる可能性がある．そこで，合成が成功した SQL クエリと実際の SQL クエリの比較を行うことで，中間言語の妥当性に関する分析を行った．

その結果，入出力例 #1, #4, #5, #6, #8 で合成された SQL クエリは，実際の SQL クエリと構文として等しいものであった．たとえば，図 5 の #4 は合成された SQL クエリの例である．一方で，入出力例 #2 と #3 では，実際の SQL クエリとは異なるものが合成された．入出力例 #2 の合成結果を図 5 の #2 に示す．実際の SQL クエリでは条件式として BETWEEN が使用されていたが，合成結果はその範囲の上限と下限を表す条件式が AND でつながれたものとなった．これらについて可読性の観点で大きな差はないと考えている．入出力例 #3 の合成結果は図 5 の #3 ，実際の SQL クエリは図中の #3(original) である．実際の SQL クエリではサブクエリを用いている一方で，合成結果では GROUP BY 句を用いている．これらはどちらも「分類 c2 が '01' であり，日付 c1 が起点 (?) より後であるもののうち，最小の日付 c1 をもつレコード」を検索する SQL クエリである．SQL クエリの書き方の慣習に依存するものの，これらの可読性に大きな差はないと考えている．

以上により，本手法によって合成が成功する SQL クエリは，実際の SQL クエリに近いものであることが分かった．本手法で使用する中間言語は，一般的な SQL の構文のサブセットとなっており，サブクエリなどはサポート外である．しかしながら，サブクエリと同じ意味をもつ SQL クエリを，可読性を大きく落とすことなく表現できる場合があることが分かった．

6. まとめと今後の展望

本研究では，入出力例から SQL クエリを合成するアルゴリズムを提案した．既存研究で課題となっていた，大きなカラム数をもつテーブルを入出力とする SQL クエリの合成を可能とした．組み合わせ爆発の原因となっていたカラムとカラムの対応関係の計算をアルゴリズムの終盤に行うことで，入出力テーブルのカラムのサイズに大きな影響を受けずに合成を行うことができるのが特徴である．ケー

ススタディでは，実際のバッチ処理システムに存在する SQL クエリ 8 件を対象として提案手法を適用し，そのうち 7 件が 1 秒以内の合成に成功することを確認した．

さいごに，今後の展望を以下に挙げる．

- 実際の開発において作成される数多くの SQL のうち，本手法でどの程度合成可能であるかを定量的に評価する．また，さまざまな性質をもつ SQL クエリを対象とした評価実験を行い，提案手法の合成能力を明らかにする．
- 本手法で使用する中間言語では，一般的な SQL クエリに比べかなり制限された構文をもつ．そのため，サブクエリや `NOT NULL` 条件など，表現できない SQL クエリの要素が存在する．合成の性能を保ちつつ，より表現力の高い SQL クエリを合成する手法を検討する．
- より表現力の高い SQL クエリの合成を実現するためには，効率的なプログラム空間の探索が必要となる．近年盛んに研究されている確率モデルや機械学習を用いた合成技術 [7], [8], [9], [10] を取り入れることでアルゴリズムを洗練する．
- SQL クエリがシステムの中で使用されるとき，他言語と連携することで 1 つの処理を実現している場合が一般的である．たとえば，あるテーブルから SQL クエリを用いてレコードを取り出し，そのレコードに加工を施し，その結果を他のテーブルに書き込むような場合である．SQL クエリだけでなく，それを呼び出しているプログラムを対象とする合成手法を検討する．
- 現状のシステム開発において，プログラム合成技術の利用は一般的でない．プログラム合成を実際の開発現場へ導入するにあたり，開発プロセスや品質保証に対して，従来と異なる考え方をもつ必要があると考えている．それらの課題を具体的に洗い出し，ベストプラクティスを確立する．

参考文献

[1] Gulwani, S., Polozov, A. and Singh, R.: Program Synthesis, *Foundations and Trends in Programming Languages*, Vol. 4 (2017).

[2] Gulwani, S. and Jain, P.: Programming by Examples: PL meets ML, *Dependable Software Systems Engineering* (2019).

[3] Gulwani, S.: Automating String Processing in Spreadsheets using Input-Output Examples, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2011).

[4] Zhang, S. and Sun, Y.: Automatically Synthesizing SQL Queries from Input-output Examples, *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pp. 224–234 (2013).

[5] Wang, C., Cheung, A. and Bodik, R.: Synthesizing Highly Expressive SQL Queries from Input-output Examples, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 452–466 (2017).

[6] Feng, Y., Martins, R., Van Geffen, J., Dillig, I. and Chaudhuri, S.: Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 422–436 (2017).

[7] Kant, N.: Recent Advances in Neural Program Synthesis, *arXiv:1802.02353* (2018).

[8] Lee, W., Heo, K., Alur, R. and Naik, M.: Accelerating Search-based Program Synthesis Using Learned Probabilistic Models, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 436–449 (2018).

[9] Balog, M., Gaunt, A., Brockschmidt, M., Nowozin, S. and Tarlow, D.: DeepCoder: Learning to Write Programs, *Proceedings of the International Conference on Learning Representations* (2017).

[10] Menon, A. K., , O. T., Gulwani, S., Lampson, B. and Kalai, A. T.: A Machine Learning Framework for Programming by Example, *Proceedings of the 30th International Conference on Machine Learning*, pp. 187–195 (2013).