

SQL による数独の解法とクエリオプティマイザの有効性

Sudoku Solver in SQL and effectiveness of SQL query optimizer

矢吹 太郎 ♡

佐久田 博司 ◆

Taro YABUKI

Hiroshi SAKUTA

SQL による数独の解法を示す。SQL は非手続き型の言語であるため、この解法には、数独の特定の問題におけるルールつまり空白に入る数に関する制約条件のみを記述すればよく、数独を解くための具体的な手続きを記述する必要はないという利点がある。Ingres と MySQL, Derby, SQLite, PostgreSQL, Oracle, SQL Server, Firebird, DB2 を使って実験し、Firebird と DB2 以外のリレーショナルデータベース管理システム (RDBMS) ではこの解法が有効であることを確認する。ただし、一部の RDBMS では、クエリヒント等によってクエリオプティマイザの動作を調整する必要がある。その原因を調査し、具体的な調整方法を示す。

We introduce a Sudoku Solver in SQL. Its strong point is that we need not describe a procedure to solve Sudoku, because SQL is a non-procedural language. We have only to describe rules—constraints between variables—of an instance of Sudoku. We test our method on Ingres, MySQL, Derby, SQLite, PostgreSQL, Oracle, SQL Server, Firebird, and DB2. As a result, we confirm that the method is available on all systems except Firebird and DB2. However, some systems require query hints that adjust behaviors of query optimizers. We show the query hints concretely and discuss the reasons for them.

1. 序論

数独は 1979 年に Garns によって発表されたパズルである [1]。このパズルの目標は、図 1 のようなグリッドの空白部分に、次の条件 (ルール) に合う数を当てはめることである。

1. 空白には 1 から 9 のいずれかの整数が入る
2. 各行と各列、各ブロック (図 1 の太線で囲まれた 3 行 3 列の領域) の数はすべて異なる

1				7		9		
	3			2				8
		9	6			5		
		5	3			9		
	1			8				2
6				4				
3								1
	4							7
		7				3		

図 1 数独の問題例 [2]

Fig. 1 Sudoku problem example [2].

1.1 SQL で数独を解く意義

さまざまな数独の解法が多くのプログラミング言語で実装されている¹。それにも拘わらず、本稿で SQL による数独の解法を提示するのは、SQL が利用されていない領域に、SQL を活用する余地があることを、具体例を使って示すためである。

手続き的なプログラミング言語と SQL のどちらで実装しても良い処理は、非手続き型言語である SQL を使った方が簡単に実装できる可能性が高い。手続き的手法を採用するプログラマは、解を求める具体的な方法 (how) を記述しなければならないのに対して、非手続き的手法を採用するプログラマは、解の性質 (what) のみを記述すればよいからである。

しかし、構成要素にリレーショナルデータベースを含む、つまり SQL を利用するシステムにおいて、SQL の利用は単純なデータ処理に限定されることが多い。Web アプリケーションはその典型例である。確かに、SQL の利用を限定的なものにすることは利点がある。たとえば、SQL の仕様は各リレーショナルデータベース管理システム (RDBMS) によって異なっており、SQL を多用することには、システムが特定の RDBMS に依存することになる危険が伴う。SQL の利用を必要最低限にとどめることで、そのような危険を回避できる。しかし、この方針には欠点もある。SQL なら簡単になるはずの実装が、SQL を避けたために複雑になってしまう危険である。

システムの実装において SQL が占める割合は、SQL の利点と欠点を考慮して決めなければならないのだが、上で述べたような SQL の非手続き性による利点が強調されることは少ない。そこで本稿では、数独というこれまで SQL で扱われたことがほとんどない問題を SQL による非手続き的な実装で解く具体的な方法を示すことによって、SQL を使うことの利点を際立たせることを目指す。

1.2 想定する状況

数独のような組み合わせ問題に対して、プログラマが単純なしらみつぶしによる解法しか思いつかない状況を想定してほしい。そのためのプログラムは、ループのための記述が不要な分、手続き的なものよりも非手続き的なものの方が簡単で、実装しやすい (2.2 節を参照)。さらに、手続き的に実装した単純なしらみつぶし法は、実行効率が悪く、実際に数独を解くことができないのに対して、SQL で実装したしらみつぶ

♡ 正会員 青山学院大学理工学部 yabuki@it.aoyama.ac.jp
◆ 非会員 青山学院大学理工学部 sakuta@it.aoyama.ac.jp

¹ たとえば、文献 [3] の 1.4 節では、「Ruby のパワーと表現力をよく示す」例として数独が取り上げられている。

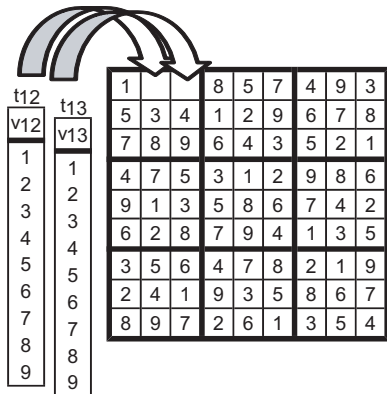


図2 数独の解法、空白に入り得る数の組み合わせを探す。
Fig. 2 Sudoku solving method to search a integer combination applicable to blanks.

し法は、手続き的な実装より単純であるにも拘わらず、多くのRDBMSで実際に数独を解くことができる。RDBMSにはクエリの実行方法(実行プラン)を最適化するクエリオプティマイザが備えられているからである。ただし、オプティマイザが常に最適な実行プランを生成するわけではなく、ユーザがその振る舞いを調整しなければならない場合もある(第4章を参照)。

1.3 既存の研究

SQLを利用して数独を解く方法は、2006年にCelkoによって発表された[4]。Celkoの方法は手続き的で、解ける問題は比較的簡単なものに限定されていた。2007年に矢吹らによって発表された手続き的解法[5]と非手続き的解法[6]は、任意の数独の問題を解けた(対象RDBMSはMySQLとSQL Server)。2008年には、矢吹らの非手続き的解法と同様のものが、Vontobelによって独立に発表された[7]。2009年には、一部のRDBMSのみでサポートされている再帰的サブクエリを用いる解法が、Schefferによって発表された[8]。

本稿では、矢吹らの非手続き的解法[6]におけるオプティマイザの役割を調査し、その結果から得られる知見を元に、より多くのRDBMSで利用できるように手法を改良する。

1.4 利用するRDBMS

文献[9, 10]を参考に、成熟度とプレゼンス、市場シェアが高いと考えられるRDBMSの本稿執筆時点での最新版(Ingres 10.1, MySQL 5.1.41, Derby 10.5.3.0, SQLite 3.6.16.1, PostgreSQL 9.0.1, Oracle 11gR2, SQL Server 2008 SP1, Firebird 2.5.0, DB2 9.7 FP1)を採用する。

2. 手法

空白が2個だけの簡単な問題を例に、数独の解法を説明する(図2)。以下では、 i 行 j 列に入る整数を v_{ij} 、 i 行 j 列に入り得る整数を格納したテーブルを t_{ij} と記述する。

図2のように数独を解くには、空白部分に入る整数の組み合わせ $\{v_{12}, v_{13}\}$ を見つければよい。第1行には1, 3, 4, 5, 7, 8, 9が、第2列には1, 2, 3, 4, 5, 7, 8, 9が、左上のブロックには

```
for (v12=1; v12<=9; v12++) {
  for (v13=1; v13<=9; v13++) {
    if (v12!=1 && v12!=2 && v12!=3 && v12!=4
        && v12!=5 && v12!=7 && v12!=8 && v12!=9
        && v13!=1 && v13!=3 && v13!=4 && v13!=5
        && v13!=6 && v13!=7 && v13!=8 && v13!=9
        && v13!=v12) {
      print v12, v13;
    }
  }
}
```

図3 図2の手続き的な実装
Fig. 3 Procedural implementation of Fig. 2.

```
for (v12=1; v12<=9; v12++) {
  if (constraints on v12) {
    for (v13=1; v13<=9; v13++) {
      if (v12!=v13 && constraints on v13) {
        print v12, v13;
      }
    }
  }
}
```

図4 図2の手続き的な実装(改良版)
Fig. 4 Procedural implementation of Fig. 2. (revised)

1, 3, 4, 5, 7, 8, 9がすでにあるため、数独のルール2(第1章)によって、 v_{12} は1, 2, 3, 4, 5, 7, 8, 9, v_{13} のすべてと、 v_{13} は1, 3, 4, 5, 6, 7, 8, 9, v_{12} のすべてと異なっていなければならない。

2.1 手続き的解法

図2の数独の解法は、図3の手続き的なプログラムで実現できる。図3の2個のfor文が数独のルール1(第1章)のための手続き、if文がルール2のための手続きになっている²。

図3のプログラムは単純であり、任意の数独の問題は、簡単にこの形式に変換できる。しかし、このプログラムは明らかに非効率である。なぜなら、外側のループで判断できる変数 v_{12} についての条件が、内側のループで判断されているからである。このため、この方法で実際の数独の問題を解くことはできない。たとえば、図1のような空白が58個ある問題の解をこの方法で見つけるためには、最悪の場合 9^{58} 通りの組み合わせを調べなければならないが、これは現実的ではない。この問題を回避するためには、図3のプログラムを図4のように書き直し、変数 v_{12} についての条件を外側のループで判断するようにすればよい。

実行時の効率という点では、図4のプログラムは図3のプ

² バックトラックを実装できるプログラマなら、数独を手続き的手法で簡単に解けるが、1.2節で述べたように、プログラマが単純なしらみつぶし法しか思いつかなかった場合をここでは想定している。

```
SELECT v12, v13
FROM t12, t13
WHERE v12<>1 AND v12<>2 AND v12<>3 AND v12<>4
      AND v12<>5 AND v12<>7 AND v12<>8 AND v12<>9
      AND v13<>1 AND v13<>3 AND v13<>4 AND v13<>5
      AND v13<>6 AND v13<>7 AND v13<>8 AND v13<>9
      AND v12<>v13
```

図5 SQLによる図2の非手続き的実装

Fig. 5 Non-procedural SQL implementation of Fig. 2.

プログラムより優れている。しかし、図3のプログラムでは1カ所 (if文) にまとめられていた数独のルール2のための記述が、図4のプログラムでは複数の場所に分散している。そのため、プログラミングの容易さという点では、図4のプログラムは図3のプログラムに劣っている。

2.2 SQLによる解法

2個のテーブル t_{12}, t_{13} があれば、図2の数独の解法は、図5のSELECT文で実現できる。テーブル t_{ij} が数独のルール1 (第1章) に、WHERE句がルール2に対応する。FROM句における2個のテーブルのコンマによる結合は直積である³。このSELECT文によって、直積で生成されるすべての数の組み合わせの中から、WHERE句の条件に合うもの、つまり数独の解が選択される。

SQLを用いるならば、図3のような手続き的実装の場合と異なり、図5のような実装でも数独を解けるということが、本稿における非手続き性の意味である。図3のプログラムにおける数独のルール1のための手続き (変数の初期値やループの方法) の記述は、図5のSELECT文では不要である。図4のように分散させる必要があった数独のルール2のための記述は、図5のSELECT文では1カ所 (WHERE句) にまとめられている。ルール2のための記述を1カ所にまとめた図3のプログラムは現実的ではなかったが、この点に関しては同様に見える図5のSELECT文は現実的である。なぜなら、SELECT文を実行するRDBMSには、効率のよい実行方法を探索するオプティマイザが備えられており、その働きによって、図5のSELECT文が図4のように実行されることが期待できるからである。

SQLによる解法は、(1) テーブルの作成、(2) SELECT文の生成、(3) SELECT文の実行、という手順で実現される (これらの手順は手続き的に実行されるが、このことは本稿の主題とは無関係である)。2.2.1項でテーブルの作成方法を、2.2.2項でSELECT文の生成方法を説明する。

2.2.1 テーブルの作成

テーブル t_{ij} に格納する整数は、さまざまな方法で決められるが、その中でも特に簡単なのは、(1) t_{ij} には1から9までのすべての整数を格納する、(2) t_{ij} には数独のルールから直ちに

³ 直積は、ISO/IEC 9075:1989ではコンマで、ISO/IEC 9075:1992では「CROSS JOIN」で表すことが規定されている。PostgreSQLやSQLiteのように、両者の振る舞いが異なるRDBMSも存在する (4.1.3項を参照)。

```
SELECT t12.n AS v12, t13.n AS v13
FROM x t12, x t13
WHERE t12.n<>1 AND t12.n<>2 AND t12.n<>3
      AND t12.n<>4 AND t12.n<>5 AND t12.n<>7
      AND t12.n<>8 AND t12.n<>9 AND t13.n<>1
      AND t13.n<>3 AND t13.n<>4 AND t13.n<>5
      AND t13.n<>6 AND t13.n<>7 AND t13.n<>8
      AND t13.n<>9 AND t12.n<>t13.n
```

図6 SQLによる図2の非手続き的実装 (必要なテーブルは1個)

Fig. 6 Non-procedural SQL implementation of Fig. 2.
Only one table is required.

```
CREATE TABLE x (n INT);
INSERT INTO x VALUES (1);
INSERT INTO x VALUES (2);
...
INSERT INTO x VALUES (9);
```

図7 テーブルxを生成するためのSQL文

Fig. 7 SQL statements to create the table 'x'.

わかる整数のみを格納する、のいずれかであろう。方法(2)は方法(1)に比べて複雑である。そのため、方法(2)を採用すると、解法の中で手続き的な要素の占める割合が大きくなり、実装に手間がかかる恐れがある。それを避けるために、本稿では方法(1)を採用する。

方法(1)を採用することによって、数独の空白部分のために、対応するテーブル t_{ij} を個別に用意する必要はなくなる。テーブル t_{ij} の内容はすべて同じ (1から9の整数) であるから、1から9のすべての整数を格納したテーブルを1個作成し、それに別名を付けて使い回せばよい。図5のSELECT文を図6のように書き換えることでそれが可能になる。図6におけるテーブルxは図7のように生成する。

2.2.2 SELECT文の生成

数独を解くためのSELECT文の生成方法を説明する。SELECT文は選択リストと表参照リスト (FROM句)、探索条件 (WHERE句) で構成される。

選択リストは、数独のすべての空白に対して「 $t_{ij}.n$ AS v_{ij} 」という文字列を生成し、コンマを挿んで連結すれば完成する (iは行、jは列を表す整数、 v_{ij} はi行j列の空白に入る整数。これ以降も同様。例:「 $t_{12}.n$ AS v_{12} 」)。

表参照リストは、数独のすべての空白に対して「 x t_{ij} 」という文字列を生成し、コンマを挿んで連結すれば完成する。本稿ではこの過程を、単純な2重ループ (外側でiが1から9まで変化し、内側でjが1から9まで変化する) で実現する。

探索条件は、数独のすべての空白に対するループを2重に回し、異ならなければならない数のペアに対して「 $t_{ij}.n <> t_{pq}.n$ 」あるいは「 $t_{ij}.n <>$ 整数」という文字列を生成し、ANDを挿んで連結すれば完成する (pは行、qは列を表す数)。

表 1 SELECT 文の実行時間 (秒)

Table 1 Elapsed time for the select statement (sec.)

RDBMS	plain	query hints
Ingres 10.1	347	~1
MySQL 5.1.41	~1	
Derby 10.5.3.0	~1	
SQLite 3.6.16.1	~1	
PostgreSQL 9.0.1	incapable	~1
Oracle 11gR2	10	~1
SQL Server 2008 SP1	7	~1
Firebird 2.5.0	incapable	
DB2 9.7 FP1	incapable	

2.3 実験環境

複数の RDBMS 上で、2.2.1 項の方法でテーブル x を作成し、2.2.2 項の方法で生成される図 1 のための SELECT 文を実行する。RDBMS は、Windows 7 64-Bit (Core i7 930 2.80GHz, 主記憶 6GB) 上に構築した仮想マシン (主記憶 2GB, Windows XP SP3 32-Bit) 上で動作させる (仮想化ソフトウェアは VirtualBox 3.2.10)。仮想マシンを用いるのは、各 RDBMS の動作環境を統一しやすくするためである。

3. 結果

2.2 節の方法をそのまま実行した際の、SELECT 文の実行時間を表 1 の “plain” に示す。本稿では 10 分以内に結果が得られなかったものは “incapable” (測定不能) としている。

3.1 実行方法を調整した場合の結果

MySQL と Derby, SQLite 以外の RDBMS は、2.2 節の方法では数独を短時間で解くことはできない (表 1 の “plain”)。第 4 章で考察するように、この原因はテーブルの結合方法にある。そこで、テーブルの結合方法が、短時間で解けた MySQL のそれと同様になるように、クエリヒント等によってオプティマイザの動作を調整した場合の結果を表 1 の “query hints” に示す。Firebird と DB2 以外の RDBMS は⁴、この調整によってほぼすべての数独の問題を解けるはずである⁵。具体的な調整方法は第 4 章で述べる。

4. 考察

SQL を利用する際には一般に、オプティマイザの特性を把握し、それに合わせてテーブルを構築し、SELECT 文を作成・実行する必要がある。このことは、本稿で提案した数独の解法にもあてはまる。数独の解を効率よく求められるかどうかは、オプティマイザが生成する実行プラン、特にテーブルの結合方法に強く依存している。そのため、一部の RDBMS では、

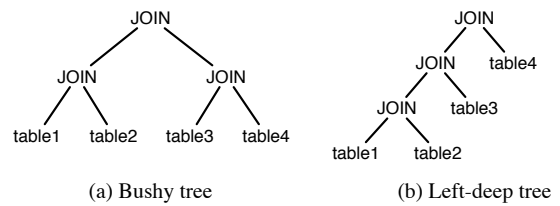


図 8 テーブルの結合方法

Fig. 8 Ways to join tables.

その点を考慮して SELECT 文を作成・実行しなければならない⁶。本章では、各 RDBMS のテーブルの結合方法を調査し、その結果をもとに、数独を効率よく解くための、オプティマイザの調整方法について考察する。

4.1 各 RDBMS の実行プラン

各 RDBMS には実行プランを報告する機能が用意されている。たとえば MySQL なら、「EXPLAIN SELECT ...」のように、調査対象となる SELECT 文の先頭に「EXPLAIN」を付けた EXPLAIN 文を実行することによって、SELECT 文の実行プランを知ることができる。このような機能を利用して、数独のための SELECT 文がどのようなプランで実行されているかを RDBMS ごとに調査し、その結果について考察する。

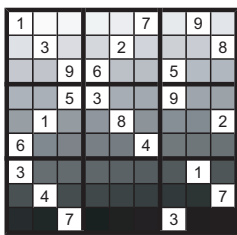
4.1.1 Ingres

Ingres のオプティマイザが生成する実行プランにおけるテーブルの結合方法は、数独を効率よく解けるものではないと推測される。最適化は、(1) 最適な実行プランの全探索、(2) ヒューリスティックによるプランの生成、という 2 段階からなる。第 1 段階で評価すべき実行プランが多すぎると、処理がタイムアウトして第 2 段階に進む。第 2 段階で生成される実行プランにおけるテーブルの結合方法は、図 8(a) のような bushy tree になる (図 1 のための SELECT 文の実行プランは取得できなかったため、小規模な問題のための SELECT 文の実行プランから推測した)。図中のテーブルが配置の近いセルに対応するものとしたときに、数独のための SELECT 文のテーブルを、bushy tree で結合させるのは非効率である。数独では、table3 は table1 と table2 についての制約を含んでいるため、table1 と table2 の結合結果に table3 を結合させることによって、table1 と table2 の候補が早く絞り込まれることが期待できる。そのため、テーブルの結合方法は図 8(b) のような left-deep tree であるのが望ましい。Bushy tree でテーブルを結合させると、処理の途中で一時的に作られるテーブルが増加し、候補の絞り込みは遅くなるだろう。

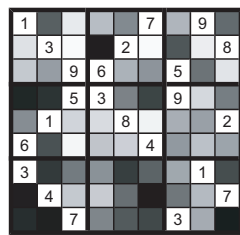
最適化の第 2 段階を無効にすることによって、このような数独には不向きな実行プランを回避できる。そのためには、SELECT 文の実行前に、コマンド「set joinop nogreedy」を実行すればよい [11]。これにより、最適化の第 1 段階がタイ

⁴ Firebird と DB2 では数独を解けないということではない。
⁵ 厳密には、ここで提案した方法で解けるのは、空白の数が、結合できるテーブルの最大数以下であるような問題に限られる。たとえば、MySQL で結合できるテーブルは 61 個までであるため、空白が 62 個以上の問題は本手法では解けない (本稿執筆時点で知られている数独の空白の最大数は 64 である [1])。

⁶ テーブルの作成方法にも注意が必要である。数独の場合にはテーブルにインデックスがない方が効率よく解が求まることが多いが、この点についての詳細は本稿では割愛する。



(a) MySQL, Derby, SQLite



(b) PostgreSQL

図9 テーブルの結合順序 (濃さの近い部分に対応するテーブルが結合される)

Fig. 9 Join ordering. (Tables corresponding to similar color cells are joined.)

ムアウトした時点での最良のプランが採用されることになる (第2段階には進まない)。本稿のための実験環境 (2.3節) においては、この時点での最良の実行プランは、left-deep tree でテーブルを結合させるものになっており、SELECT 文はそれに基づいて効率よく実行される。

4.1.2 MySQL と Derby, SQLite

MySQL のオプティマイザは、FROM 句に記述された通りの順序でテーブルを結合するような実行プランを生成する。本稿では、FROM 句は 2.2.2 項で述べた手続きで生成するため、テーブルの結合順序は図 9(a) のようになる。

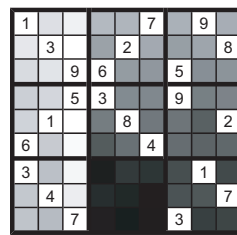
図 7 の方法で作成するテーブルにインデックスはないが、仮にインデックスがあるとすると、オプティマイザがその利用可能性を探索するのにかかる時間のために、SELECT 文の実行時間が測定不能になる。そのため、MySQL を利用して数独を解く際には、テーブルにインデックスを張るべきではない。ただし、インデックスがある場合でも、WHERE 句中の条件を「 $t_{ij}.n < \text{数}$ 」ではなく「 $\text{数} < t_{ij}.n$ 」のように書けば、インデックスの利用可能性は探索されなくなる⁷。SELECT の直後に「`/*! STRAIGHT_JOIN */`」と書くことによって、テーブルの結合順序を強制的に記述通りにしてもよい [12]。

Derby と SQLite では、テーブルは図 9(a) のような順序、図 8(b) のような方法で結合され、解は短時間で得られる。

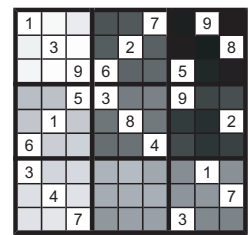
4.1.3 PostgreSQL

PostgreSQL のオプティマイザが生成する実行プランは、数独を効率よく解けるものではない。この実行プランにおけるテーブルの結合順序は 9(b) のようになるが、数独では、離れた空白のためのテーブルを早い段階で結合しても、変数の値を絞り込むことは期待できない。そのため、FROM 句において、コンマではなく「`CROSS JOIN`」を用いてテーブルの結合順序を強制的に記述通りにし、結合方法を探索しないようにする [13]。そうすると、テーブルは図 9(a) のような順序、図 8(b) のような方法で結合されるようになる。

⁷ 環境変数 `optimizer_search_depth` の値を小さくしてもよい。



(a) Oracle



(b) SQL Server

図 10 テーブルの結合順序

Fig. 10 Join ordering.

4.1.4 Oracle と SQL Server

Oracle と SQL Server のオプティマイザは、テーブルの結合順序を探し、適切な実行プランを生成している。この実行プランにおけるテーブルの結合順序は図 10 のようになる。この結合順序は、配置に近いセルに対応するテーブルから順に結合されるものであるため効率はよい。しかし、本稿のための実験環境 (2.3節) においては、オプティマイザがこの結論を導くのに時間がかかるという問題がある。

この問題はクエリヒントによって解決できる。Oracle では SELECT の直後に「`/*+ ORDERED */`」というクエリヒントを [14]、SQL Server では SELECT 文の末尾に「`OPTION (FORCE ORDER)`」というクエリヒントを書けば [15]、テーブルの結合順序が FROM 句の記述通り、つまり図 9(a) のような順序、図 8(b) のような方法になる。

単純なループによって生成した図 9(a) の結合順序は、図 10 の結合順序に比べて効率は悪い。しかし、数独のためには十分であることが、クエリヒントを使った場合の実行時間 (表 1 の「query hints」) からわかる。

4.1.5 DB2 と Firebird

DB2 では解も実行プランも得られなかった。そこで、図 1 よりも空白の少ない数独の問題のための、結合するテーブルの少ない SELECT 文を用意し、その実行プランを調べた。その結果をもとに考察すると、提案手法が DB2 では有効でない理由として、以下の 3 点が挙げられる。

第 1 に、テーブルの結合順序がセルの配置の近さに対応したものになっていない (図 9(a) よりも図 9(b) に近い)。そのため、変数の組み合わせを大量に探索しなくてはならなくなっている。第 2 に、テーブルの結合方法が図 8(a) のような bushy tree になっている。これは、Ingres の場合 (4.1.1 項) と同様の理由で非効率である。第 3 に、WHERE 句における変数と定数の関係が、IN を使った形に書き換えられている。たとえば、「 $v_{12} < 1 \text{ AND } v_{12} < 2 \text{ AND } v_{12} < 3$ 」のような条件が、「 $v_{12} \text{ NOT IN } (1, 2, 3)$ 」のように書き換えられている。このため、変数の値の可能性が絞り込まれても条件の数が減らず、探索に時間がかかっている。これらの問題を解決する方法は不明である。

Firebird では解は得られなかった。その原因は不明である。

4.2 クエリオプティマイザへの指示のまとめ

数独を効率よく解くための、オプティマイザに与える指示をまとめると以下のようになる。

- Ingres: コマンド「set joinop nogreedy」を実行する
- MySQL: 「/*! STRAIGHT_JOIN */」を使う
- Derby: なし (2.2節のままでよい)
- SQLite: なし (2.2節のままでよい)
- PostgreSQL: 「CROSS JOIN」を使う
- Oracle: 「/*+ ORDERED */」を使う
- SQL Server: 「OPTION (FORCE ORDER)」を使う
- Firebird: 提案手法は有効ではない
- DB2: 提案手法は有効ではない

以上の指示を与えると、オプティマイザはテーブルの結合方法を探索しなくなり、その結果として、数独の解が短時間で得られるようになる (Firebird と DB2 は除く)。

このような指示を与えても、数独を手続き的に解くことにはならないことに注意してほしい。非手続き的記述で数独を解くためには、オプティマイザのような、実行プランを生成する仕掛けが不可欠である。実行プランにはさまざまな要素が含まれるが、本章で無効にしたのは、それらの要素の中のテーブルの結合方法の探索に関するものだけである。提案手法で数独を解くために重要なのは、図6のような SELECT 文を、図3ではなく図4のように実行することであるが、そのための具体的な実行プランの作成をオプティマイザに任せていることに変わりはない。

5. 結論

SQL による数独の解法を示した。本手法は、SQL の非手続き性を利用して、本手法で数独を解く際には、解の性質の記述のみが必要で、解を得るための具体的な方法の記述は不要である。このように、何か計算機プログラムとして実装したい処理があるときに、手続き的な言語ではなく非手続き的な言語を用いると実装が簡単になる可能性がある。

プログラムを実装できることとそれが効率よく動くことは別問題である。特に、非手続き的な言語を用いた場合、プログラムの実行方法に占めるプログラマには見えにくい部分の割合が、手続き的な言語の場合よりも大きくなる。その部分をブラックボックスのままにしておく、プログラムを効率よく実行できない恐れがある。SQL で数独を解く本手法に関して言えば、SELECT 文を実行するための具体的な方法 (実行プラン) はクエリオプティマイザによって決められるが、RDBMS によっては、その機能の一部 (テーブルの結合方法の探索) を無効にした方が、実行効率は良くなる。

このように、非手続き的記述とオプティマイザの調整で簡単に解ける問題があることを、複数の RDBMS (Ingres, MySQL, Derby, SQLite, PostgreSQL, Oracle, SQL Server) で確認した。

SQL はリレーショナルデータベースのための問い合わせ言語であるが、その用途は単純なデータの作成と読み込み、更

新、削除にとどまらない。これまで手続き的なプログラミング言語で実装されていた処理は、SQL の非手続き的な性質を活用して簡単に実装し、効率よく実現できる可能性がある。その可能性を示すために、数独という、SQL で解かれたことがほとんどない例を使って、非手続き的な記述の方法と、実行時に考慮すべき点を紹介した。SQL の非手続き的な側面に注目が集まり、それを利用した応用事例が増えることが期待される。

[謝辞]

本研究を進めるにあたり、貴重な意見をくださった青山学院大学社会情報学部の増永良文教授に感謝いたします。

[文献]

- [1] Jean-Paul Delahaye. The science behind sudoku. *Scientific American*, Vol. 294, pp. 80–87, 2006.
- [2] Arto Inkala. *Ai Escargot*. Lulu.com, 2007.
- [3] David Flanagan and Yukihiko Matz Matsumoto. *The Ruby Programming Language*. O'Reilly & Associates Inc., 2008.
- [4] Joe Celko. *Joe Celko's SQL Puzzles and Answers*. Morgan Kaufmann, 2nd edition, 2006.
- [5] 矢吹太朗. SQL による数独の高速解法. *CodeZine*, 2007.
- [6] 矢吹太朗. 動的 SQL による数独の超高速解法. *CodeZine*, 2007.
- [7] Beat Vontobel. The lost art of the self join. Presentation at MySQL Conference & Expo, 2008.
- [8] Anton Scheffer. Oracle RDBMS 11gR2—solving a sudoku using recursive subquery factoring, 2009.
- [9] 株式会社野村総合研究所. NRI オープンソースマップ, 2006.
- [10] Carl Olofson. Worldwide RDBMS 2006 vendor shares: Preliminary results for the top 5 vendors. IDC, 2007.
- [11] Ingres Corporation. *Ingres 10.0 Database Administrator Guide*, 2010.
- [12] Stefan Hinz, et al. *MySQL 5.1 Reference Manual*.
- [13] The PostgreSQL Global Development Group. *PostgreSQL 9.0.1 Documentation*, 2010.
- [14] Oracle Corporation. *Oracle Database SQL Language Reference 11g Release 2*, 2010.
- [15] Microsoft Corporation. *SQL Server Books Online*, 2010.

矢吹 太朗 Taro YABUKI

青山学院大学理工学部情報テクノロジー学科助教。2004 東京大学大学院新領域創成科学研究科修了。博士 (科学)。研究テーマは情報・計算・進化。

佐久田 博司 Hiroshi SAKUTA

青山学院大学理工学部情報テクノロジー学科教授。1979 東京大学工学系大学院修了。工学博士。