AN EFFICIENT ALGORITHM FOR CONVERTING A SPREADSHEET OF ROWS WITH IMPLICIT PARENT-CHILD RELATIONSHIPS TO A DATABASE TABLE OF ROWS WITH EXPLICIT PARENT-CHILD RELATIONSHIPS

Pham Van Viet
Le Quy Don Technical University

ARTICLE INFO

Received: 15/7/2022
Revised: 22/8/2022
Published: 23/8/2022

KEYWORDS

Convert spreadsheet to table
Implicit parent-child relationship
Explicit parent-child relationship
Self-referencing table
Parent-child hierarchy

ABSTRACT

Converting difficult-to-use data to easy-to-use one in real world applications is popular. This paper proposes an efficient algorithm for converting data from an input spreadsheet of rows with implicit parent-child relationships to a database table of rows with explicit parent-child relationships and defined birth orders. Here, a row with an implicit parent-child relationship has no reference fields to its parent row, but its parent can be determined by its level and order. The input spreadsheet is hard to query and bind with tree type user interface components for display in applications, while the output database table is convenient. The proposed algorithm's time complexity is *O(n)*, where *n* is the number of rows in the input spreadsheet. The algorithm also uses limited memory space: it uses a few basic data type variables and two integer lists, where their size doesn't exceed the number of different levels used to assign to the spreadsheet's rows.

MỘT THUẬT TOÁN HIỆU QUẢ ĐỂ CHUYỂN ĐỔI MỘT BẢNG TÍNH CÁC HÀNG CÓ QUAN HỆ CHA-CON NGẦM ĐỊNH VÀO MỘT BẢNG CƠ SỞ DỮ LIỆU CÁC HÀNG CÓ MỐI QUAN HỆ CHA-CON RÕ RÀNG

Phạm Văn Việt
Trường Đại học Kỹ thuật Lê Quý Đôn

THÔNG TIN BÀI BÁO

Ngày nhận bài: 15/7/2022
Ngày hoàn thiện: 22/8/2022
Ngày đăng: 23/8/2022

TỪ KHÓA

Chuyển đổi bảng tính thành bảng
Quan hệ cha-con ngầm định
Quan hệ cha-con rõ ràng
Bảng tự tham chiếu
Hệ phân cấp cha con

TÓM TẮT

Việc chuyển đổi dữ liệu khó sử dụng thành dữ liệu dễ sử dụng trong các ứng dụng thế giới thực là phổ biến. Bài báo này đề xuất một thuật toán hiệu quả để chuyển đổi dữ liệu từ một bảng tính đầu vào các hàng có mối quan hệ cha-con ngầm định thành một bảng cơ sở dữ liệu gồm các hàng có mối quan hệ cha-con rõ ràng và thứ tự sinh xác định. Ở đây, một hàng có mối quan hệ cha-con ngầm định không có trường tham chiếu đến hàng cha của nó, nhưng cha của nó có thể được xác định bằng cấp và thứ tự của nó. Bảng tính đầu vào khó truy vấn và liên kết với các thành phần giao diện người dùng kiểu cây để hiển thị trong các ứng dụng, trong khi bảng cơ sở dữ liệu đầu ra rất tiện lợi. Độ phức tạp thời gian của thuật toán được đề xuất là *O(n)*, trong đó *n* là số hàng trong bảng tính đầu vào. Thuật toán cũng sử dụng không gian bộ nhớ hạn chế: nó sử dụng một vài biến kiểu dữ liệu cơ bản và hai danh sách số nguyên, trong đó kích thước của chúng không vượt quá số lượng các cấp khác nhau được sử dụng để gán cho các hàng của bảng tính.

DOI: https://doi.org/10.34238/tnu-jst.6258

Email: v.v.pham2012@gmail.com

1. Introduction

In the real world, an organization may input data in a Excel-like spreadsheet of rows with implicit parent-child relationships as in Table 1. Here, a row with an implicit parent-child relationship has no reference fields to its parent row. The organization then has an application to display and input the data. However, the application's user interface component for displaying and inputting the data is a tree type user interface component. The data's structure as in Table 1 cannot bind with the tree type user interface component. Further, queries on Table 1 (e.g. getting the child rows of any row) are complicated. Converting the spreadsheet to a database table of rows with explicit parent-child relationships as in Table 2 can solve these issues. Each row in the table is added three more fields: "ID" to identify the row, "Parent ID" to refer to the row's parent explicitly and "Birth order" of the row. The table is a self-referencing table with a parent-child hierarchy. This paper proposes an efficient algorithm for the converting problem.

Table 1. *Input spreadsheet sample*

Number	Title	Other data
I	Row 1	Row data 1
1	Row 2	Row data 2
a	Row 3	Row data 3
b	Row 4	Row data 4
-	Row5	Row data 5
-	Row 6	Row data 6
2	Row 7	Row data 7
II	Row 8	Row data 8
1	Row 9	Row data 9
a	Row 10	Row data 10
-	Row 11	Row data 11
-	Row 12	Row data 12
-	Row 13	Row data 13
b	Row 14	Row data 14
2	Row 15	Row data 15
a	Row 16	Row data 16
b	Row 17	Row data 17
c	Row 18	Row data 18

There has been a variety of studies on the problem of converting difficult to use data to easy to use one in real world applications. An algorithm is developed to convert tabular data into images for deep learning in [1]. The study in [2] imports data from MySQL to Hadoop. A user-friendly application is introduced to convert source data to brain imaging data structure for analyses of brain function and anatomy in [3]. The studies in [4] - [6] convert spreadsheets into a database, but they do not consider spreadsheets of rows with implicit parent-child relationships. The studies in [7], [8] propose a two-phase semiautomatic system that extracts accurate relational metadata in spreadsheets while minimizing user effort. They consider implicit parent-child relationships between rows. However, the relationships are checked for all row pairs, where typographic style and geometric distance between two rows are first checked for no more checking. The studies in [9], [10] also search for parent-child pairs among row labels, but they don't describe the algorithm for parent-child pairs search in details. If the number of rows in a spreadsheet is n, the number of all row pairs is $n*(n-1)/2$. Therefore, if an algorithm checks the parent-child relationships of all row pairs, its time complexity is $O(n^2)$. The algorithm proposed in this paper uses the level and order of each row for determining its parent. The parent of a row is determined directly by its level, thus the number of pairs to be considered is the number of rows in the spreadsheet (i.e. the number of row pairs to be considered is minimum). The algorithm's time complexity is $O(n)$.

Table 2. *Database table sample*

ID	Parent ID	Birth order	Number	Title	Other data
1		1	I	Row 1	Row data 1
2	1	1	1	Row 2	Row data 2
3	2	1	a	Row 3	Row data 3
4	2	2	b	Row 4	Row data 4
5	4	1	-	Row 5	Row data 5
6	4	2	-	Row 6	Row data 6
7	1	2	2	Row 7	Row data 7
8		2	II	Row 8	Row data 8
9	8	1	1	Row 9	Row data 9
10	9	1	a	Row 10	Row data 10
11	10	1	-	Row 11	Row data 11
12	10	2	-	Row 12	Row data 12
13	10	3	-	Row 13	Row data 13
14	9	2	b	Row 14	Row data 14
15	8	2	2	Row 15	Row data 15
16	15	1	a	Row 16	Row data 16
17	15	2	b	Row 17	Row data 17
18	15	3	c	Row 18	Row data 18

The following sections include: section 2 presents the proposed algorithm, section 3 presents the algorithm's analysis, and the last section is conclusion.

2. Proposed algorithm

The proposed algorithm for converting data is in Algorithm 1, in Python pseudocode. It has an input of a spreadsheet of *n* rows with implicit parent-child relationships and outputs a database table of *n* rows with explicit parent-child relationships. The algorithm goes through each row in the input spreadsheet and works out *id*, *parentID* and *birthOrder* that are the current row's identifier, parent identifier and birth order respectively. It uses two integer lists *idList* and *birthOrderList* with elements being numbered from *0*. The *idList* and *birthOrderList* store *IDs* and *birthOrders* of rows at lower levels or at the same level with the current row. Each level has only one element in the lists. The elements *idList[i]* and *birthOrderList[i]* are the *id* and *birthOrder* of the row at level *i* in the branch from a row at level *0* to the current row. A branch is a set of ordered rows, a row at level *i* is the parent of a row at level *i + 1*. The variable *id* is initialized as *0* to assign to a row's *id*. In each step of the for loop, *id* is increased by *1*, and is assigned to the current row'*id*; the current row's *number*, *title* and *otherData* fields are determined, based on the cell_0, cell_1 and cell_2 of the current row. The current row's *level* is worked out, based on its *number*. For example, the row's level is 0 if its number is a Roman number, is 1 if its number is a natural number, is 3 if its number is a alphabet leter from 'a' to 'z', is 4 if its number is a hyphen. This paper assumes that a row's level is determined correctly. A system can have a different spreadsheet content format, hence a different method can be applied for row-level determination. Then the current row's *id*, *parentID* and *birthOrder* are determined as follows:

- **Line 10-19**: If the current row's *level* is equal to 0, its parent does not exist, hence its *parentID* is assigned as *None*.
  - If the *birthOrderList* has no elements, the current row's *birthOder* is *1*.
  - Otherwise the current row's *birthOrder* is *birthOrderList[0] + 1*, i.e. its *birthOrder* is the *birthOrder* of its adjacent elder brother row at level *0*; the *idList* and *birthOderList* are then cleared.
  - Store the current row's *id* and *birthOder* in the *idList* and *birthOrderList* for working out *parentID* and *birthOrder* of the next rows.

Algorithm 1. *Data converting algorithm*

Input: *A spreadsheet containing rows with implicit parent-child relationships. Its structure is presented in Table 1. Assume that the spreadsheet has **n** rows being numbered from **0**.*

Output: *A database table containing **n** rows with parent-child relationships and birth orders explicitly identified by **id**, **parentID** and **birthOrder** fields.*

```
1  idList = []
2  birthOrderList =[]
3  id = 0
4  for i in range(0, n):
5          id = id + 1     // the current row's id
6          number = get the current row's number from cell_0 of row_i
7          title = get the current row's title from cell_1 of row_i
8          otherData = get the current row's other data from cell_2 of row_i
9          level = determine the current row's level from cell_0 of row_i (i.e. its number)
10         if level == 0:
11                 parentID = None
12                 if (len(birthOrderList)==0):    // birthOrderList is empty
13                         birthOrder = 1
14                 else:
15                         birthOrder = birthOrderList[0] + 1
16                         idList = []
17                         birthOrderList = []
18                 idList.append(id)
19                 birthOrderList.append(birthOrder)
20         if level > 0:
21                 parentID = idList[level-1]
22                 while (level<len(idList)-1):
23                         idList.pop()
24                         birthOrderList.pop()
25                 if (level==len(idList)-1):
26                         idList [len(idList)-1] = id
27                         birthOrder = birthOrderList[len(birthOrderList)-1] + 1
28                         birthOrderList[len(birthOrderList)-1] = birthOrder
29                 elif (level>len(idList)-1):
30                         idList.append(id)
31                         birthOrder = 1
32                         birthOrderList.append(1)
33         Insert a record with information of the current row including id, parentID,
           birthOrder, number, title and otherData to the output database table.
```

- **Line 20-32**: If the current row's *level* is greater than *0*, its *parentID* is *idList[level – 1]*, i.e. the *id* of the row at the previous level.
    - ***Line 22-24***: if the current row's *level* < *len(idList) – 1* (i.e. it's *level* is less than the *level* of the row corresponding to the last element in the *idList*), elements at levels higher than the current row's *level* are removed from the *idList* and *birthOrderList*, until the current row's *level* is equal to the *level* of the row corresponding to the last element in the *idList*).
    - ***Line 25-28***: If the current row's *level = len(idList)-1* (i.e. it's *level* is equal to the *level* of the row corresponding to the last element in the *idList* or *birthOrderList*), its *birthOrder* is the *birthOrder* of the last element in *birthOderList* plus *1*. Then, the last elements in the *idList* and *birthOrderList* are updated as the current row's *id* and *birthOrder*.

o ***Line 29-32***: If the current row's *level > len(idList)-1* (i.e. the current row's *level* is greater than the *level* of the row corresponding to last element in the *idList* or *birthOrderList*), its *level* is *1*, since the current row is the first row at its level. The current row's *id* and *birthOrder* are stored in the *idList* and *birthOrderList* for determining the next rows' *id* and *birthOrder*.

- **Line 33**: Insert a record with information of the current row including *id*, *parentID*, *birthOrder*, *number*, *title* and *otherData* to the output database table.

Running the algorithm with the input spreadsheet in Table 1, we received the outputs step by step as shown in Table 3. The algorithm's first step of is the initialization step, where the *idList* and *birthOrderList* are initialized as empty; and the variable *id* for assigning to a row's *id* is initialized as *0*. Each following step of the algorithm is an iteration of the for loop corresponding to a row. A step has inputs of *id*, *idList* and *birthOrderList* from its previous step and a row in the input spreadsheet and it outputs a row in the database table with the columns of *number*, *level*, *id*, *parentID* and *birthOrder* and updates the *idList* and *birthOrderList*. The following is the explanation for each step corresponding to a row.

- Row 1: the current row's *level* is *0*; its *id* is *1* (*id = id + 1 = 1*); its *level is 0* (i.e. it is not a child of any row), hence its *parentID* is *None*; the step's input *birthOrderList* is empty, hence its *birthOrder* is *1*; the current row's *id* and *birthOrder* are appended to the *idList* and *birthOrderList* respectively for determining the next rows' *parentID* and *birthOrder*, the lists after appending are: *idList = [1]* and *birthOrderList = [1]*.
- Row 2: the current row's *level* is *1*; its *id* is *2* (*id = id + 1 = 2*); its *level* is greater than *0*, hence its *parentID* is *idList[level – 1] = idList[0] = 1* (i.e. its *parentID* is the *id* of the row at the previous level, stored in the *idList*); the current row's *level > len(idList) – 1*, (i.e. its *level* is greater than the *level* of the row corresponding to the last element in the step's input *idList* or *birthOrderList*), hence its *birthOrder* is 1 (the current row is the first row at its *level*); the current row's *id* and *birthOrder* are append to the *idList* and *birthOrderList* respectively for determining the next rows' *parentID* and *birthOrder*, the lists after appending are: *idList = [1, 2]* and *birthOrderList = [1, 1]*.
- Row 3: the current row's *level* is 2; its *id* is *3* (*id = id + 1 = 3*); its *level* is greater than *0*, hence its *parentID* is *idList[level – 1] = idList[1] = 2* (i.e. its *parentID* is the *id* of the row at the previous level, stored in the *idList*); the current row's *level > len(idList) – 1*, (i.e. its *level* is greater than the *level* of the row corresponding to the last element in the step's input *idList* or *birthOrderList*), hence its *birthOrder* is *1* (the current row is the first row at its *level*); the current row's *id* and *birthOrder* are appended to the *idList* and *birthOrderList* respectively for determining the next rows' *parentID* and *birthOrder*, the lists after appending are: *idList = [1, 2, 3]* and *birthOrderList = [1, 1, 1]*.
- Row 4: the current row's *level* is *2*; its *id* is *4* (*id = id + 1 = 4*); its *level* is greater than *0*, hence its *parentID* is *idList[level – 1] = idList[1] = 2* (i.e. its *parentID* is the *id* of the row at the previous level, stored in the *idList*); the current row's *level = len(idList) – 1*, (i.e. its *level* is equal to the *level* of the row corresponding to the last element in the step's input *idList* or *birthOrderList*), hence its *birthOrder* is the last *birthOrder* in the *birthOrderList* plus *1* equal to *2*; the last elements in the *idList* and *birthOrderList* are updated as the current row' *id* and *birthOrder*, the lists after updating are: *idList = [1, 2, 4]* and *birthOrderList = [1, 1, 2]*.
- Row 5: the current row's *level* is *3*; its *id* is *5* (*id = id + 1 = 5*); its *level* is greater than *0*, hence its *parentID* is *idList[level – 1] = idList[2] = 4* (i.e. its *parentID* is the *id* of the row at the previous level, stored in the *idList*); the current row's *level > len(idList) – 1*, (i.e. its *level* is greater than the *level* of the row corresponding to the last element in the step's input *idList* or *birthOrderList*), hence its *birthOrder* is *1* (the current row is the first row at its *level*); the current row's *id* and *birthOrder* are appended to the *idList* and *birthOrderList* respectively for

determining the next rows' *parentID* and *birthOrder*, the lists after appending are: *idList = [1, 2, 4, 5]* and *birthOrderList = [1, 1, 2, 1]*.

- Row 6: the current row's *level* is *3*; its *id* is *6* (*id = id + 1 = 6*); its *level* is greater than *0*, hence its *parentID* is *idList[level – 1] = idList[2] = 4* (i.e. its *parentID* is the *id* of the row at the previous level, stored in the *idList*); the current row's *level = len(idList) – 1,* (i.e. its level is equal to the level of the row corresponding to the last element in the step's input *idList* or *birthOrderList*), hence its *birthOrder* is the last *birthOrder* in the *birthOrderList* plus *1* equal to *2*; the last elements in the *idList* and *birthOrderList* are updated as the current row' *id* and *birthOrder*, the lists after updating are: *idList = [1, 2, 4, 6]* and *birthOrderList = [1, 1, 2, 2]*.
- Row 7: the current row's *level* is *1*; its *id* is *7* (*id = id + 1 = 7*); its *level* is greater than *0*, hence its *parentID* is *idList[level – 1] = idList[0] = 1* (i.e. its *parentID* is the *id* of the row at the previous level, stored in the *idList*); the current row's *level < len(idList) – 1*, (i.e. its *level* is less than the *level* of the row corresponding to the last element in the step's input *idList* or *birthOrderList*), hence the algorithm removes all the elements in the *idList* and *birthOrderList* at levels less than the current row's *level* (these elements are no longer be used for determining the next rows' *parentID* and *birthOrder*), the lists after removing are: *idList = [1, 2]* and *birthOrderList = [1, 1]*; for these new lists, the current row's *level = len(idList) -1* (i.e. its *level* is the *level* of the row corresponding to the last element in the *idList* or *birthOrderList*), hence its *birthOrder* is the last *birthOrder* in the *birthOrderList* plus *1* equal to *2*; the last elements in the *idList* and *birthOrderList* are updated as the current row' *id* and *birthOrder*, the lists after updating are: *idList = [1, 7]* and *birthOrderList = [1, 2]*.
- Row 8: the current row's *level* is *0*; its *id* is *8* (*id = id + 1 = 8*); its *level* is *0* (i.e. it is not a child of any row), hence *parentID* is *None*; the step's input *birthOrderList* is not empty, hence its *birthOrder* is *birthOrderList[0] + 1* (i.e. its *birthOrder* is the *birthOrder* of the above row at *level 0* plus *1*), equal to *2*; all the elements in the *idList* and *birthOrderList* are removed; the current row's *id* and *birthOrder* are appended to the *idList* and *birthOrderList* respectively for determining the next rows' *parentID* and *birthOrder*, the updated lists are: *idList = [8]* and *birthOrderList = [2]*.
- Explanation for the next rows is similar to the above.

Table 3. *The outputs of the algorithm step by step*

Step	Number	Level	ID	parentID	birthOrder	idList	birthOrderList
Init			0				
Row 1	I	0	1		1	1	1
Row 2	1	1	2	1	1	1, 2	1, 1
Row 3	a	2	3	2	1	1, 2, 3	1, 1, 1
Row 4	b	2	4	2	2	1, 2, 4	1, 1, 2
Row 5	-	3	5	4	1	1, 2, 4, 5	1, 1, 2, 1
Row 6	-	3	6	4	2	1, 2, 4, 6	1, 1, 2, 2
Row 7	2	1	7	1	2	1, 7	1, 2
Row 8	II	0	8		2	8	2
Row 9	1	1	9	8	1	8, 9	2, 1
Row 10	a	2	10	9	1	8, 9, 10	2, 1, 1
Row 11	-	3	11	10	1	8, 9, 10, 11	2, 1, 1, 1
Row 12	-	3	12	10	2	8, 9, 10, 12	2, 1, 1, 2
Row 13	-	3	13	10	3	8, 9, 10, 13	2, 1, 1, 3
Row 14	b	2	14	9	2	8, 9, 14	2, 1, 2
Row 15	2	1	15	8	2	8, 15	2, 2
Row 16	a	2	16	15	1	8, 15, 16	2, 2, 1
Row 17	b	2	17	15	2	8, 15, 17	2, 2, 2
Row 18	c	2	18	15	3	8, 15, 18	2, 2, 3

3. Algorithm analysis

We can see that the number of executions of a statement in each iteration of the outer for loop is 0 or 1, except for the number of executions of a statement in the inner while loop. The inner while loop takes elements out of the *idList* and *birthOrderList*. The numbers of taken out elements can be different in different iterations of the outer for loop. However, the total number of taken out elements in all the iterations can't be greater than the number of rows in the input spreadsheet, since a row's id and birthOrder can be pushed into the lists only one time. Thus, the algorithm's time complexity is $O(n)$, where n is the number of rows in the input spreadsheet.

For the algorithm's memory space, the algorithm uses the two lists: *idList* and *birthOrderList* with dynamic sizes and a few integer and string variables. However, the maximum size of the *idList* and *birthOrderList* doesn't exceed the number of different levels used to assign to the spreadsheet's rows.

4. Conclusion

In this paper, we proposed an efficient algorithm for converting data from a spreadsheet containing rows with implicit parent-child relationships to a database table containing rows with explicit parent-child relationships and defined birth orders. The algorithm's time complexity is $O(n)$, where n is the number of rows in the input spreadsheet. For the algorithm's memory space, the algorithm uses two integer lists with dynamic sizes and a few number of integer and string variables. However, the maximum size of the integer lists doesn't exceed the number of different levels used to assign to the spreadsheet's rows.

REFERENCES

[1] Y. Zhu, T. Brettin, F. Xia, A. Partin, M. Shukla, H. Yoo, Y. A. Evrard, J. H. Doroshow and R. L. Stevens, "Converting tabular data into images for deep learning with convolutional neural networks," *Scientific Reports*, vol. 11, pp. 1-11, 2021.

[2] D. A. Reddy and M. J. Rihaz, "Importing data from MySQL to Hadoop using Sqoop," *Advances in Computational Sciences and Technology*, vol. 10, no. 9, pp. 2835-2840, 2017.

[3] M. P. Zwiers, S. Moia, and R. Oostenveld, "BIDScoin: A User-Friendly Application to Convert Source Data to Brain Imaging Data Structure," *Frontiers in Neuroinformatics*, vol. 15, pp. 1-12, 2022.

[4] J. Cunha, J. Saraiva and J. Visser, "From Spreadsheets to Relational Databases and Back," in The 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation, Savannah, GA, USA, 2009.

[5] C. Yang, J. Liu, W. Hsu, H. Lu, and W. C. Chu, "Implementation of Data Transform Method into NoSQL Database for Healthcare," in 2013 International Conference on Parallel and Distributed Computing, Applications and Technologies, Taipei, Taiwan, 2013.

[6] A. Awad, R. ElGohary, I. Moawad, and M. Roushdy, "An interactive tool for extracting low-quality spreadsheet tables and converting into relational database," *International Journal of Intelligent Computing and Information Sciences*, vol. 21, no. 1, pp. 1-18, 2021.

[7] Z. Chen and M. Cafarella, "Integrating spreadsheet data via accurate and low-effort extraction," in The 20th ACM SIGKDD international conference on Knowledge discovery and data mining, New York, USA, 2014.

[8] Z. Chen, "Information extraction on para-relational data," PhD. Thesis, University of Michigan, Ann Arbor, Michigan, 2016.

[9] A. O. Shigarov and A. A. Mikhailov, "Rule-based spreadsheet data transformation from arbitrary to relational tables," *Information Systems*, vol. 71, pp. 123-136, 2017.

[10] A. Shigarov, V. Khristyuk, and A. Mikhailov, "TabbyXL: Software platform for rule-based spreadsheet data extraction and transformation," *SoftwareX*, vol. 10, pp. 1-6, 2019.