COMPREHENSION SUPPORT OF SQL STATEMENT USING DOUBLE-TREE STRUCTURE

Takehiko Murakawa and Masaru Nakagawa
Faculty of Systems Engineering, Wakayama University, Wakayama, Japan

Keywords: Program understanding, Code review, SQL, Graphical expression, Syntactic analysis.

Abstract: SQL is a practical programming language used mainly in the query to relational database. However there have been rarely met support tools for database programmers' understanding with relation to SQL statements. We have proposed a framework where a clamshell diagram which looks like a symmetrical, double tree is drawn given an SQL statement. In this paper, we report an automatic conversion system of SQL statements into clamshell diagrams. The system parses the given statement and arranges the configuration which differs slightly from the well-known syntax tree. Moreover we actually generated some dozens of diagrams for SQL statements using the system to make sure that it can readily draw diagrams.

1 INTRODUCTION

SQL is a practical programming language used mainly in the query to relational database. Most departments on information science or computer engineering let the students learn basic concepts of database and acquaint themselves with the queries using SQL statements. In the authors' affiliation, the students train in SQL after mastering C and Java. Some of them have a resistance to SQL since, say, the syntax is unprecedented.

Bending our eyes on the software development business, we have rarely met support tools for database programmers' understanding with relation to SQL statements, although we can find such tools for source codes written in general-purpose procedural languages. The approaches to the comprehension support of procedural languages are (Desmond et al., 2006; Chen, 2010; Zest,) and others. Davis (Davis, 2008) attempted to assist the understanding of source files by allocating them on a window.

There exist support tools in fact which analyze any SQL statement to visualize it. For example, SQLProb (Liu et al., 2009) parses given SQL statements and forms the trees to detect the change of configuration which indicates an SQL injection attack. Another case is Visual Explain (VisualExplain, 2006) which generates the access plan as a tree. Both applications, however, would be poor at the code inspection support by a single person or multiple persons. Apart from the tools, a case study is found in (Chan et al., 2005) which made quantitative evaluations of accuracy of the codes that novice SQL programmers wrote.

We have proposed a framework for program understanding using a graphic record that we call a clamshell diagram (Murakawa and Nakagawa, 2010). Roughly summarized, a clamshell diagram is a symmetrical, double tree for representing two hierarchical structure and a problem-solution relationship. After formalizing it and drawing figures (Murakawa et al., 2008), we have focused on SQL. Those diagrams were, however, made by hands. Such way of drawing is inefficient and has a disadvantage for the code reviewer to examine analogous codes.

In this paper we report an automatic conversion system of SQL statements into clamshell diagrams. It parses the given statement and configures the left-hand tree which differs slightly from the well-known syntax tree. And then, it creates the structure of the right-hand tree where the labels of SQL fragments may be changed into words which are more natural with regards to English expression. The system is the improved version of the one (Murakawa et al., 2010).

2 PRELIMINARIES

2.1 SQL

SQL is a programming language for the query to

Murakawa T. and Nakagawa M..
COMPREHENSION SUPPORT OF SQL STATEMENT USING DOUBLE-TREE STRUCTURE.
DOI: 10.5220/0003346103180323
In *Proceedings of the 3rd International Conference on Computer Supported Education* (CSEDU-2011), pages 318-323
ISBN: 978-989-8425-49-2

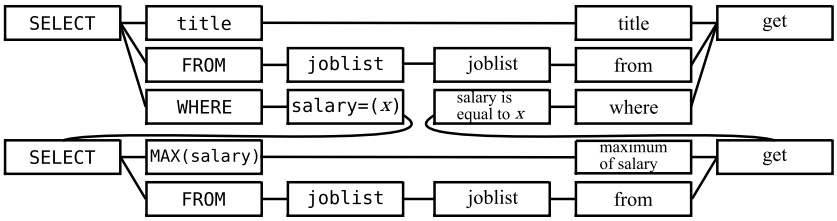

Figure 1: Example of clamshell diagram for an SQL statement (old way of drawing).

database. Although a typical executable unit of SQL is smaller than that of C or popular ones, an SQL statement could be unboundedly lengthened by using a nested query or a subquery. Moreover, as far as the processing performance is concerned, a single, complicated SQL statement is prefer to divided, simplified ones.

Students of the authors' department study SQL after mastering C and learning Java. When we put a question of writing down a SQL statement for a bit long query in a natural language to the students, the accuracy rate is terribly low. One of the goals in the database classes is that the students learn to write down the right queries by themselves. For this purpose, the objective of our study is throwing a bridge over queries or intentions and SQL statements.

2.2 Clamshell Diagram

A standard clamshell diagram has two tree structures whose roots are located in the both ends of the diagram and the leaves of the trees are connected one-to-one in the center. While a tree presents a mere expansion, a clamshell diagram with double tree visualizes both the divergence and convergence. Moreover we can have a careful look at the completed diagram by several methods, such as changing the viewpoint into one of the adjacent nodes or into the opposite node, or skimming through a path from the left edge to the right. The combination of these ways of viewing helps us understand the object.

If you are interested in the formal definition of clamshell diagrams, see (Murakawa and Nakagawa, 2010). Note that we have only to keep a single tree for a clamshell diagram in a computer, where each node holds the information of the corresponding nodes of both-hand trees.

The basic idea of drawing is to supply the hierarchical code fragments of a given SQL statement in the left side of the diagram and to put the meaning written in a natural language in the right. Figure 1 is an example of clamshell diagram for an SQL statement including a subquery. There exist two nodes which read `SELECT` on the extreme left of this figure, but we can easily see that the lower one is the child node of "`salary=(x)`". The symbol `x` is a parameter which substitutes for the sequence for the subtree. When we traverse the left-hand tree in preorder, it reads "`SELECT title FROM joblist WHERE salary=(SELECT MAX(salary) FROM joblist)`". From the right-hand tree, we can similarly obtain the sequence, "get title from joblist where salary is equal to 'maximum of salary from joblist' ", where the word "get" which occurs except at the root of the right-hand tree is omitted. The query is to retrieve the titles of the maximum salary, under the assumption that there exists a table named `joblist` which consists of two columns by the name of `title` and `salary`.

There exist problems in drawing clamshell diagrams by hand. Firstly, it is tedious and error-prone. Then, it is not easy to understand expressions which may be too long and complicated. Lastly, it is not straightforward to identify the column names, especially in the statement where a table name appears twice or more. To overcome these difficulties, we developed the software for converting SQL statements into clamshell diagrams.

3 AUTOMATIC CONVERSION OF SQL STATEMENT INTO CLAMSHELL DIAGRAM

3.1 Overview

The overview of generating the clamshell diagram for a given SQL statement is shown in Fig. 2.

We would like to enumerate the features of the program we have developed. Firstly, it takes an SQL statement as input to make the clamshell diagram. It thereby automates the time-consuming job. Then, each token of the given SQL statement is associated with a node of the left-hand tree of the clamshell diagram, apart from what is introduced for readability. This property hopes that we can understand the roles of the tokens in the connection of others more easily. Lastly, if a single column name is specified in the given SQL statement, then it is augmented with

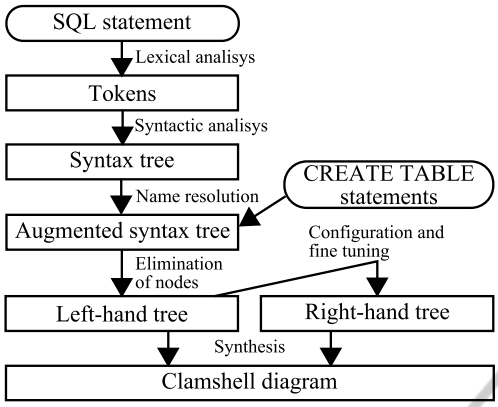

Figure 2: Overview of generating clamshell diagram using the system.

the appropriate table name and the dot symbol before the column name. Such an enhancement helps one to understand the origin of the column name.

The system was implemented using Ruby. Whereas employing free, well-established library and software, we wrote several Ruby script files for the various processes.

3.2 Parsing

Lexical and syntactic analyses are done with the aid of the Ruby's library Racc (http://i.loveruby.net/en/projects/racc/). Figure 3 is a part of the Ruby script file for the syntax analyzer. The word without any lower case letters means a special token of SQL, and the one beginning with "K_" indicates a reserved word by removing the prefix. The lines between `prechigh` and `preclow` define the operator precedence together with the associativity. After the line of "`rule`", the word without any upper case letters denotes a non-terminal symbol whose syntax is defined in the file. The vertical bar is used for the disjunction. Inside a pair of braces, the action is described in Ruby. The last line except "`...`" in Fig. 3 suggests that `expr` is defined in a recursive way. We introduced 53 non-terminal symbols while defining 186 syntactic rules (counting the disjunctive rules separately), by referring to the specification of SQL (ISO9075,) and subsidiarily the graphical syntactic rule about SQLite (http://www.sqlite.org/syntaxdiagrams.html).

Racc converts the syntax definition file into a Ruby script which takes a grammatical SQL statement as input to produce an Array object in Ruby which corresponds to the syntax tree.

```
class Sqlp
  prechigh
    right UMINUS
    right UMINUS
    left PERIOD2
    left PERIOD3
    left ASTERISK SLASH PERCENT
    left PLUS MINUS DOUBLE_VERTICAL_BAR
...
  preclow

  options no_result_var

  rule
    sql_stmt: sql_stmt1 sql_stmt2 {...}
    sql_stmt1: {[]}
      | K_EXPLAIN {...}
      | K_EXPLAIN K_QUERY K_PLAN {...}
    sql_stmt2: select_stmt {...}
    expr: literal_value {...}
      | column_name {...}
      | expr ASTERISK expr {...}
...
```

Figure 3: A part of syntax definition for Racc.

3.3 Name Resolution of Columns

When reading a complicated SQL statement, we often have difficulty with identifying the table for a single identifier which should be a column name. Taking the SQL statement in Section 2.2 for example, `joblist` and `salary` respectively appear twice, and we learn that each column `salary` is wired in the table `joblist` in chronological order.

We attempted to add words to the given SQL statement. Just after parsing, if the table name without `AS` modifier occurs more than twice, then `AS` modifier is attached. In addition, for the (sub)expression which is merely the column name, the table name and a dot are inserted just before it. For this purpose the program takes `CREATE TABLE` statements involved with the given SQL statement as an auxiliary input.

The SQL statement in Section 2.2 can be converted into "`SELECT j1.title FROM joblist AS j1 WHERE j1.salary=(SELECT MAX(j2.salary) FROM joblist AS j2)`". Our system is designed so that the rules can be applicable in the SQL statements with inline views as well.

3.4 Making the Left-hand Tree

When the statement "`SELECT 3+4`" passes through the parser, we can obtain the syntax tree (a bit optimized) shown in Fig. 4, where `select_core` and `expr` denote non-terminal symbols. If this tree were adopted as the left-hand tree of the clamshell diagram, then the inner nodes associated with the non-terminal symbols would give no information in the right-hand tree. In general, we would also like to give care to locating the opening and closing parentheses on the chart. These difficulties made us remove the in-

ner nodes associated with non-terminal symbols thoroughly, with the property preserved that the diagram should produce the original SQL statement by traversing the left-half tree in preorder.

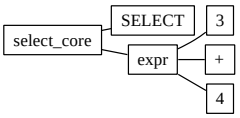

Figure 4: Syntax tree of `SELECT 3+4`.

First, we need to pay attention to the binary operator which frequently occurs in the expression, since it is not compatible with preorder configuration. For this problem, we attempt to reconfigure the involved nodes so as to make them fit in with preorder. The principle leads to the tree shown in Fig. 5 for the expression $x+1*1-1$ where x is an expression. You can hide the operator together with its right operand both from the expression and from the tree. Note that if the expression is $y+1$ and y is a single value, then it corresponds to the chain which ties in "y", "$+$" and "1" in sequence. This way of reconfiguration can be applied as well in listing two items or more with the comma injected.

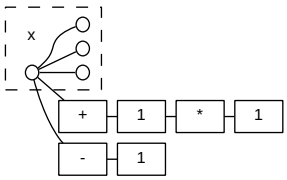

Figure 5: Configuration of the expression $x+1*1-1$.

We would like to propose the marriage of a pair of parentheses, by appending the conversion rules; if the parentheses are attached to the function call, then the label of the root in the subtree about the call consists of the function name and the parentheses; otherwise, the pair of parentheses change themselves into a single node with "( )" on it, and this node is the root of the subtree. When the node of a pair of parentheses has more than one child nodes as the result of reduction, only the first child node should be put in the parentheses whereas the rest follows the subexpression.

Finally we are presenting the way of eliminating inner nodes in the case other than described above. When we have a subtree whose root is associated with a non-terminal symbol, the first-born node among the child nodes of the subtree should promote to the root. This manipulation keeps the word order through the preorder scanning.

3.5 Making the Right-hand Tree and Whole Diagram

After constructing the left-half tree, we will obtain the right-half tree in a straightforward way. The configuration is just the same as that of left-half tree, and some labels are altered. Table 1 shows the mapping rule for simple tokens. Moreover, all the keywords of SQL are decapitalized, and the function name followed by the parentheses is converted into the function name plus "of". On the other hand, we leave the parentheses other than seen in function calls untouched for readability.

Table 1: Mapping rule of labels for the right-hand tree.

SQL's Token	Substitute word
`SELECT`	get
*	records
,	and
.	's

Note: The token * which appears as the binary operator is not subject to the rule.

We employ Graphviz (http://www.graphviz.org/) for drawing clamshell diagrams. Graphviz basically takes a text file of the graph configuration as input, and produces an image file. You can choose the file format of the output image among vector formats such as SVG (Scalable Vector Graphics) and EPS (Encapsulated PostScript) or pixel formats such as PNG (Portable Network Graphics) and JPEG (Joint Photographic Experts Group). When our program written in Ruby invokes `dot` command that Graphviz furnishes, we finally have an image file of the clamshell diagram.

4 RESULTS

A clamshell diagram which is drawn through the above processes is shown in Fig. 6. The original SQL statement is "`SELECT j2.title FROM (SELECT MAX(j1.salary) AS max_salary FROM joblist AS j1) AS j, joblist AS j2 WHERE j2.salary=j.max_salary`", and intended to calculate the job's titles of the maximum salary in the joblist table using the inline view. The figure presents the inner clamshell diagram whose both edges are the nodes which read "`SELECT`" and "get" in the left-hand and right-hand trees respectively. If a mere tree were displayed, we would be bewildered when reaching a leaf node. Clamshell diagram supplies the right-hand path to convergence, which enables us to

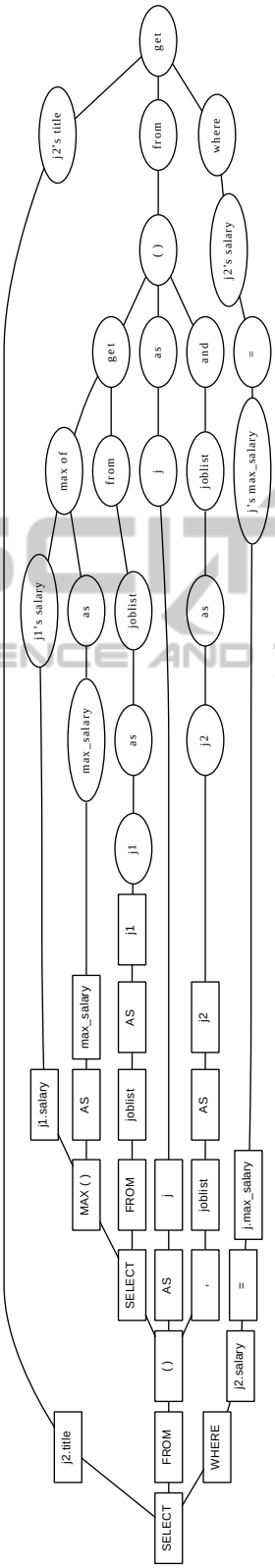

Figure 6: Example of clamshell diagram for an SQL statement.

understand the pieces of the original query through left-to-right stream.

When traversing the right-hand tree of this diagram in consideration for the parentheses' interpretation and the elimination of "get" inside, we obtain "get j2's title from (max of j1's salary as max_salary from joblist as j1) as j and joblist as j2 where j2's salary = j's max_salary."

Dozens of simple queries brought the valid clamshell diagrams. Moreover we applied to the SQL statement for calculating the median (Celko, 2005, p.515) that consists of 132 tokens, to obtain the clamshell diagram with 228 nodes where the length of the path between the both ends is up to 36.

5 DISCUSSION

We have presented the prospect for the solution through automatic converter while enumerating the problems in drawing clamshell diagrams for SQL statements by hand. We are looking back on them, since we currently have the practical examples of produced clamshell diagram as well as the conversion system. The developed software, taking an SQL statement as input, has several Ruby scripts work together to make the clamshell diagram immediately. Although there are some exceptions for readability, the resulting diagram holds more nodes than the one by hand, which makes each tokens in the statement much clearer. Furthermore, by assigning each column name in the statement to a node where the appropriate table name is qualified, the targets of information reference confidently identified on the diagram. Therefore we consider that the framework of conversion of SQL statements into clamshell diagrams contributes to the SQL comprehension support.

With the utilization of the visualization tool described until now in mind, we would like to explain an idea for making use of the system in the classes of database. There is a teacher or a student who writes an SQL statement using a computer, for retrieving some records from the tables where the schema was defined and the records were stored previously. He or she punches in the query to send the characters or words to the server discretely. When a statement is correct in grammar, the server generates the clamshell diagram and returns it to the client. Referring to the clamshell diagram together with the query, the teacher gives commentary on noteworthy tokens. Moreover, they can find the tables in the diagram which are omitted from a simple SQL statement. If the system can judge whether the statement is correct or not in a moment, and if the time for drawing, transmitting and

rendering the clamshell diagram is negligibly small, then the learners can enjoy the results dynamically changed on the screen. It is not so hard to build up the application which works with a single PC.

Taking the actual use of clamshell diagrams into consideration, we have to recognize that there exists a room to improve the way of visualization. For example, we could change the shapes of nodes, apply colors according as the meanings of labels, and fold and expand the subtrees. In addition, the system becomes more attractive if a user, looking at the generated diagram, can touch a node and change it on site. We expect SVG to be the best image format, since SVG is an XML (Extensible Markup Language) image format which Graphviz supports and the components of the image are controllable through DOM (Document Object Model), using JavaScript.

6 CONCLUSIONS

In this paper we have reported the automatic conversion system of SQL statements into clamshell diagrams. Resulting diagrams enable us to understand the pieces of the original queries thanks to the double tree structure and the left-to-right stream.

Future works include conducting evaluation experiments for readability of clamshell diagrams in comparison with other ways of showing SQL statements. Supporting other languages than English in the right-hand tree is also under consideration.

REFERENCES

Celko, J. (2005). *Joe Celko's SQL for Smarties: Advanced SQL Programming, Third Edition*. Morgan Kaufmann.

Chan, H., Teo, H., and Zeng, X. (2005). An evaluation of novice end-user computing performance: Data modeling, query writing, and comprehension. *Journal of the American Society for Information Science and Technology*, 56(8):843–853.

Chen, X. (2010). Extraction and visualization of traceability relationships between documents and source code. In *IEEE/ACM international conference on Automated software engineering*, pages 505–509.

Davis, S. (2008). Automatic juxtaposition of source files. Master's thesis, University of British Columbia.

Desmond, M., Storey, M.-A., and Exton, C. (2006). Fluid source code views for just in-time comprehension. In *Workshop on Software Engineering Properties of Languages and Aspect Technologies*.

ISO9075. Information technology—database languages—SQL—part 1: Framework (SQL/framework). ISO/IEC 9075-1:1999.

Liu, A., Yuan, Y., Wijesekera, D., and Stavrou, A. (2009). SQLProb: a proxy-based architecture towards preventing SQL injection attacks. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 2054–2061.

Murakawa, T., Kawasaki, T., Mizuochi, H., and Nakagawa, M. (2008). Formulation of clamshell diagram and its application to source code reading. In *Proceedings of the 8th Joint Conference on Knowledge-Based Software Engineering (JCKBSE 2008)*, pages 474–483.

Murakawa, T. and Nakagawa, M. (2010). Graphical expression of SQL statements using clamshell diagram. *IEICE TRANSACTIONS on Information and Systems*, E93-D(4):713–720.

Murakawa, T., Tsujimoto, A., Matsuo, K., and Nakagawa, M. (2010). Automatic conversion of SQL statement into clamshell diagram. In *Proceedings of the 9th Joint Conference on Knowledge-Based Software Engineering (JCKBSE'10)*, pages 100–101.

VisualExplain (2006). DB2 9.1 Visual Explain tutorial. IBM.

Zest. Zest: The eclipse visualization toolkit. http://www.eclipse.org/gef/zest/.