PAPER Special Section on Knowledge-Based Software Engineering

Graphical Expression of SQL Statements Using Clamshell Diagram

Takehiko MURAKAWA[†a)], *Member and* **Masaru NAKAGAWA**[†], *Nonmember*

SUMMARY Thinking process development diagram is a graphical expression from which readers can easily find not only the hierarchy of a given problem but the relationship between the problem and the solution. Although that has been developed as an idea creation support tool in the field of mechanical design, we referred to the restricted version as clamshell diagram to attempt to apply to other fields. In this paper we propose the framework for drawing the diagram of the SQL statement. The basic idea 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. To verify the usefulness of the diagram expression, we actually drew several clamshell diagrams. For three SQL statements that are derived from the same specification, the resulting diagrams enable us to understand the difference visually.

key words: *program understanding, software inspection, SQL, diagram, information representation*

1. Introduction

In the Internet services such as electronic commerce and ticket reservation, it is typical that one or more servers employ some sort of database management system (DBMS) to hold the key data, while Web servers act as the wicket for responding users' requests. Since the communication between Web servers and the DBMS using SQL is done within the server side, the users enjoy the service without regard to the database.

When constructing such a practical database system, the lightweight language like PHP: Hypertext Preprocessor (PHP) takes part of the logic and the Web servers' behavior, and the SQL statement is often described as a string in the script file. That is why SQL is considered less serious from the viewpoint of source code maintenance.

Object/Relational (O/R) mapping is a mechanism to enable the developers to write the directions about record retrieval and manipulation. ActiveRecord used together with Ruby on Rails is the driving force. However we have to recognize that O/R mappers only supply the convenience of coding but the optimization of the SQL queries is untouched. Actually, ActiveRecord leaves the method for executing any SQL statement directly, from which we can infer that it is a human work that makes the most efficient database access via SQL.

We investigated the application of thinking process development diagrams [1] (TPDDs) to lower process of software development [2]–[4]. Although TPDDs have been developed for supporting mechanical design, the originators made the point that they are also useful in planning or organizing something in wider fields and in analyzing the phenomena or the existing products in detail. The authors agree to their assertion, and have been looking into the graphical expression for program understanding.

In this paper we propose the formalism for drawing the diagram of the SQL statement. The basic idea is to supply the hierarchical code fragments of a given SQL statement in the left side hand of the diagram and to put the meaning written in a natural language in the right. SQL permits a nest of SQL statement, called a subquery, which makes the statement complex. We employ a variable-length, symmetric clamshell diagram which is a variant of TPDD, and give expression to the subquery by means of grafting. By using the proposed method, a long, abstruse SQL statement can be deciphered easily by looking at the diagram with symmetrical two trees. The examples of three SQL statements that are the same in meaning, together with their clamshell diagrams, enable us to understand the difference in the details of the statements. The diagrams will be informative and helpful to the programmers of database manipulation from beginners to experts.

The outline of this paper is shown in Fig. 1, by means of clamshell diagram. Except those seen in Fig. 2, all the clamshell diagrams are drawn with Graphviz.

2. Clamshell Diagram

2.1 Introduction of Clamshell Diagram

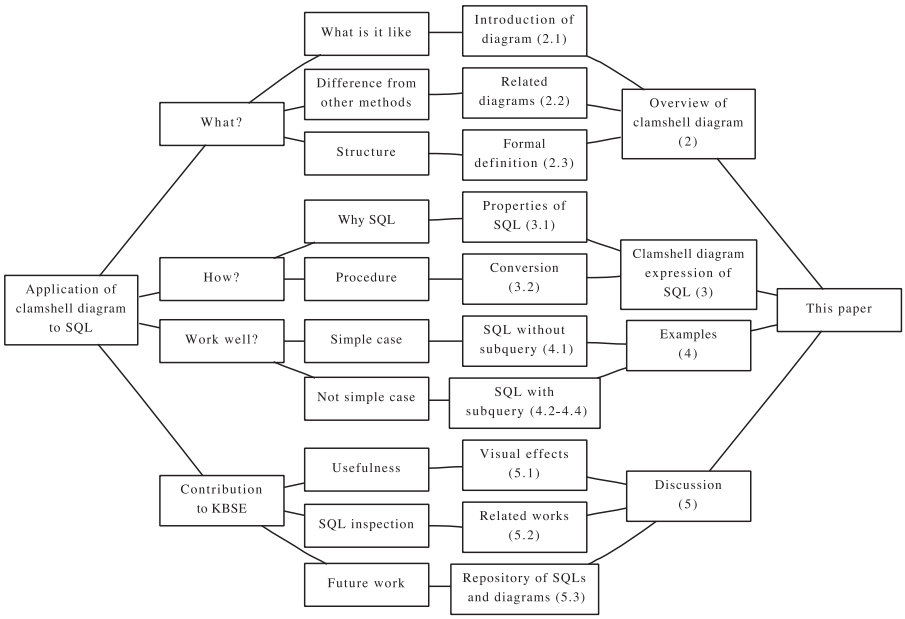
Thinking process development diagram [1] is a diagram of the designer's thought in a certain style. The completed diagram will be represented including two hierarchical relationships and the relationship of issue and resolution. Although the diagrams have been developed in mechanical design, the originators said that they would be available to broader area of design processes, with which we agree.

A standard diagram has two tree structures whose roots are the both ends of the diagram and the leaves of the trees are connected one-to-one in the center. The left half of the diagram conservatively shows the problem to be solved, while the right half expresses the solution. Ideally the diagram is bilaterally symmetric, where the corresponding nodes have the relationship of problem-solution. In other words, an incomplete diagram helps us to find the lack of

Manuscript received July 4, 2009.
Manuscript revised October 19, 2009.

[†]The authors are with the Faculty of Systems Engineering, Wakayama University, Wakayama-shi, 640–8510 Japan.
a) E-mail: takehiko@sys.wakayama-u.ac.jp

DOI: 10.1587/transinf.E93.D.713



The parenthetic number means the chapter or the section.
KBSE: Knowledge-Based Software Engineering.

Fig. 1 Outline of this paper using clamshell diagram.

information; if a node appears in the left half but the corresponding one is not observable in the right half, for example, then you will imagine that the means for the specified (sub)goal is unclear. Such missing nodes are expected to complement after a drawer has a careful look at neighboring nodes and thinks of the hierarchical relationship or the homogeneous elements. The spatial thinking support like this is a remarkable feature of this diagram.

The nodes are labeled and put in position from left to right by ordinary. When drawing a diagram, one begins with the deployment of the left root, and then expands the notions or sentences to the right. In the right half, he or she has to converge the thought, and the right edge is single information, namely the conclusion. Thanks to these structural constraints, we can think of the issue and resolution at a time, and produce the design creatively with a moderate strain.

The authors refer to the diagrams as "clamshell diagrams" in this paper, since the word clamshell is the best expressive of the structure; a typical diagram takes the shape of a clamshell where two shells are opened with the joint at both ends.

2.2 Related Diagrams for Idea Creation Support

KJ method (Jiro Kawakita's method) [5] supplies the way of arranging miscellaneous pieces of information for a subject. Yagishita et al. [6] attempted a quantitative evaluation of the resulting contents. KJ method roughly consists of two steps; noting down each idea on a card, and consolidating the cards. The former step corresponds to the act of creating a node in our supporting system reported in [2]. However, the latter step seems to lack a constructive rule of organizing the ideas. Clamshell diagrams present the clear constructions of nodes as well as the whole-part relationship.

A Mind Map is popularly used for expanding a bit of idea [7]. With a single keyword centered, one draws curves in every direction to attach relevant notions. From a view of construction, a diagram is regarded as a directed tree where the centered keyword is the root. The trouble is that, consequently, if a phrase occurs twice or more on the diagram, then they should be separated. Another weak point of Mind Maps is that how widely and deeply the ideas should be expanded lies in the hand of the drawer. Someone might complete a diagram where one direction is further expanded while the others are poor. That is because of wrong caliber of the drawer or due to wrong establishment of the centered subject. The clamshell diagram is so clear since the goal should be a condensed single notion. In addition, if the initial subject get to be inadequate in proportion as the nodes, then one can change the centered topic using tree operations such as deletion, insertion or rotation.

A fast (abbreviation of "Function Analysis System Technique") diagram [8] is an extended version of a logic tree.

A cause effect diagram has a backbone and small bones which directly or indirectly connect to the backbone. The diagram is also known as "fishbone diagram" for its shape, or as "Ishikawa diagram" after the inventor. While used in the Japanese business community, the diagram is now so popular that an example is found in [9]. That is used for presenting all the factors for a given property or result. The cause effect diagram reads the backward reasoning, which distinguishes it from the Mind Map and the fast diagram.

TRIZ is a methodology for creation support based on

the patterns derived from enormous number of patents [10]. That has a matrix consisting of some dozens of parameters to clarify the technical problems for the combinations. Schlueter [11] used it to improve a GUI application written in Perl/Tk.

2.3 Formal Definition of Symmetric Clamshell Diagram

A great number of thinking process development diagrams have been drawn for expressing designers' intention. Through the diagrams on printed publications and made ones with our hands, we recognize various diagram schemata. In this section, after providing a couple of methods for classifying, we identify the schema for expressing SQL statements afterward.

Any diagram is either the one of the same length of paths from the far left to the right or the one where the length of paths is varied. The former graph is called a fixed-length diagram while the other is variable-length. Another classification is whether it is symmetric or not. Symmetric diagrams are easier to draw and read than asymmetric ones, but they are so restricted and idealized that one might feel ill at ease while drawing.

In this paper, we employ the variable-length, symmetric clamshell diagrams for applying to SQL statements, since SQL statements are complicated unboundedly by using subqueries, and we consider that the symmetrical property is useful in generating the diagram in a certain manner or by automatization. Although earlier clamshell diagrams for expressing C programs permit empty nodes, i.e. the nodes with no description [2], [3], all the nodes are labeled in the diagram drawn by the proposed method in this paper.

We present a formal definition of the configuration of clamshell diagram to express SQL statements.

Definition 1: The structure of the symmetric clamshell diagram is an undirected graph $G = (V, E)$ where V denotes the set of vertices or nodes and E is a set of edges, i.e. pairs of vertices. In addition, the following properties have to hold.

- Both components are divided; $V = V_L \cup V_R$ and $E = E_L \cup E_R \cup E_C$. V_L and V_R are disjoint and so are E_L, E_R and E_C.
- Let $G_L = (V_L, E_L)$, then it is an undirected subgraph of G. For this graph, there exists a rooted directed tree $T_L = (V_L, E'_L)$ such that if $(v1, v2) \in E'_L$ then $(v1, v2) \in E_L$. As for the V_R and E_R, we have the subgraph $G_R = (V_R, E_R)$ and the derived, rooted directed tree $T_R = (V_R, E'_R)$ similarly.
- Two trees T_L and T_R are isomorphic, that is, there exists an isomorphic mapping $f : V_L \rightarrow V_R$ such that $(v1, v2) \in E'_L$ if and only if $(f(v1), f(v2)) \in E'_R$.
- If a node $v \in T_L$ is a leaf of the tree, or there does not exist a node $v' \in T_L$ such that $(v, v') \in E'_L$, then the edge $(v, f(v))$ belongs to C_E.

Intuitively, G_L and G_R are respectively the left and the right half of the whole structure G, while E_C means the binding over the two subgraphs. Although the condition for variance of the paths in the tree is not present, we could supply the definition of fixed length by requiring T_L to be a balanced tree.

To store the configuration of symmetric clamshell diagram in a computer or a database, the sets T_L and T_R rather than G are convenient since they are trees. We hereafter refer to the structure formed by T_L and T_R as the left and the right trees of the whole clamshell diagram, respectively. Using T_L and T_R, we identify the occurrence of any node on a diagram. The root of the left tree is addressed by L. When a node $u \in T_L$ is associated with the address U, and $v \in T_L$ is the i-th child of u where it is assumed that the siblings are numbered, the address of v is $U.i$. The location of each node in the right tree is similarly defined where the prefix symbol is R instead of L. For example, the address of "Why SQL" in Fig. 1 is L.2.1 while the label on R.3 reads "Examples (4)".

We drew the diagrams for a couple of pieces of open source software written in C [2], [3]. The right half of the diagram shows a tree in which the labels are directory names, file names or code segments except comments. The left tree has the inverted connections so that the diagram may be line-symmetric, and the message on some nodes are the comments extracted from the files and the others are empty. The right-and-left allocation is due to a design concept of TPDD, or the belief that his or her thought should be expanded in terms of "what to do" and then converged by making clear "how to do" and integrating them. Using the drawn diagrams, we assessed the sufficiency of comments to make sure that the definition of an important function has more fulfilling comments.

3. Expression of SQL Statements Using Clamshell Diagram

3.1 SQL

In this section we enumerate the properties of the programming language SQL. It is a language for querying and modifying the database, generally used in communicating with DBMSs. When a user sends a query by means of an SQL statement, using a command-line interface or over some application programming interface, the DBMS performs calculates according to the relevant tables and returns a result which is a table if relational database is employed. Since the SQL language is declarative, many programmers in companies and colleges have a resistance to this language after studying procedural languages like C and Java.

Another character of SQL is that the statement which usually ends with a semicolon is apt to be longer, in comparison to that in C or Java. It is difficult to execute a series of SQL statements, only by SQL, where the subsequent query is created dependent on the result of the former one. In addition, a single complicated statement is believed to be more effective than several simple ones for the same purpose since

the overhead between the caller and the DBMS is short. When the tables are normalized already, the statement often includes the product or the join of the tables. Sometimes duplicated tables using self-join are the processing object. It is a good measure in reading or writing a long SQL statement to understand it in a stepwise manner, though, the students who tackle such a puzzling case are in a minority.

SQL statements are embedded in another programming language under a practical development or operational environment. The programmer is required to have a good command of two different language there. When he or she is rather poor at SQL programming and diverts existing codes without review, there will be a cause for inefficiency or security flaws. This paper does not however aims at efficiency or security of SQL since such problems are close to the tables and the indices as well as the DBMS and the concerned SQL statement.

3.2 Procedure of Conversion

The outline of the conversion of a given SQL to the symmetric clamshell diagram is shown in Fig. 2 (a)–(d).

We concentrate on the SELECT statement in this paper, since that is in heavier usage than any other statement provided by the standard SQL such as INSERT, UPDATE or CREATE TABLE statement [12]. Besides the frequency or utility, we are interested in the fact that SQL statements permit nested SQL statements, or subqueries, which must be a SELECT statement. It is true that the INSERT statement and others are able to include a SELECT one as subquery, but we now explore the space constructed by SELECT statements.

When a given SELECT statement has no subquery, the allocation of the left tree is straightforward; the keyword SELECT corresponds to the address L, the expression between SELECT and FROM to L.1, the keyword FROM to L.2, tables specified just after FROM to L.2.1. Other optional clauses such as WHERE, GROUP BY, HAVING clauses are addressed same as the FROM clause. Even though, for example, the expression at L.1 is too complicated, its segmentalization is out of account.

In contrast to the way of drawing a conventional TPDD, our formalism arranges the code segments on the left half and the corresponding comments on the right. We adopt the reverse arrangement since the tree expression of an SQL statement on the left comes more naturally to SQL programmers who find the desired query by sending ones many times. Although our policy goes against the traditional way of TPDD, the authors believe that this placement gains an advantage in a practical sense.

Now we describe how to cope with the statement with subqueries. Keep in mind that the parentheses are typically put around the subquery but they do not matter. The parentheses which occur in function call, specify the priority order, or enumerate the values just after the IN predicate are regarded as a part of the expression. When the subquery out of parentheses appears before or after the UNION operator, to the contrary, it should be dealt with appropriately.

We express a subquery in terms of *grafting*; the subquery is transformed into the subtree of the left tree. The SQL expression with a subquery is reduced by replacing the words of the subquery with a symbol which is not used in the SQL statement. And then the directed edge is added from the node associated with the expression to the root of the subtree. When an expression has two subqueries or more, the node owns the descending links of the same number and the distinct symbols are seen in the reduced expression.

Here we clarify the addresses in the case of using subquery. Under the assumption that there exists a left tree constructed only by subquery, where will the node addressed by $L.v$ (L denotes the left root of the subtree.) be mapped in the finished diagram? Assume in addition that the subtree is the i-th child of the node whose address is $L.u$ (L represent the left root of the whole diagram.), then the address which we would like to know is $L.u.i.v$.

Mapping the SQL fragment to the description in some natural language is a manual handling task right now. However we are able to present several patterns of translation in English, based on our experience in drawing the diagrams. The initial word SELECT suits with “get”. If the expression just after SELECT is merely *, then the desirable word is “record”. When the table and its byname is connected directly in the FROM clause, the word “alias” should be inserted after the word-for-word translation.

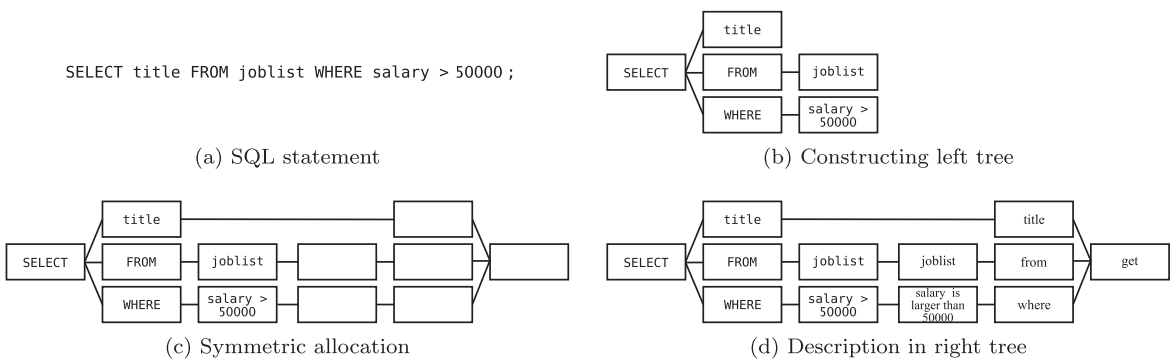


Fig. 2 Steps of drawing clamshell diagram for SQL statement.

When constructing the left tree in the manner described above, we can restore the original SQL statement by traversing the tree in preorder, laying the stumbled words side-by-side and replacing the symbol with the subquery produced by the subtree. If the labels on the right tree are written in English, then we will also be able to make the instruction by traversing it in preorder. It is insufficient, in the right tree restoration, to trade the symbol for the statement derived from the subtree; the initial word "get" should be removed.

Different sorts of statements besides the SELECT statement could be put in position by introducing appropriate conversion rule. For example, the statement "`UPDATE table SET column = value`" is translated into English as "update table to set column at value", from which we can easily obtain the one-to-one relationship for drawing the diagram.

4. Examples

Here we are presenting several clamshell diagrams for concrete SQL statements. Preparatory for showing the examples, we introduce the table to which the statements commonly refer. Imagine that you have a database of well-known jobs and their salaries, namely the table `joblist` which has the attributes `title` and `salary`. The type of `salary` is an integer while `title` ranges variable-length strings. The currency unit of `salary` is ignored.

4.1 Simple SQL Statement

If you would like to know the jobs whose income is 50,000 or more, then execute "`SELECT title FROM joblist WHERE salary > 50000;`". We describe the diagram in Fig. 3. This case is so simple that the differences between the both trees are the roots and the nodes for condition which uses the comparative operator >.

When traversing the left tree in preorder, we obtain the original SQL statement. On the other hand, the right hand tree produces the message "get title from joblist where salary is larger than 50000", which will be a natural instruction by putting in articles.

4.2 SQL Statement Including Subquery

In the rest of this chapter, we describe three SQL statements that are the same in meaning. The query is "get the titles of the maximum salary in a given joblist table." Note that there may exist more than one title which takes the maximum salary. (Otherwise "`SELECT title FROM joblist ORDER BY salary DESC LIMIT 1;`" would be the shortest code.) Without knowing the maximum salary in advance, the SQL statement which satisfies the requirement has to refer the table joblist twice, namely to know the maximum salary and to identify the titles. Therefore, to express the query with a single SQL statement, we have to use the nested one.

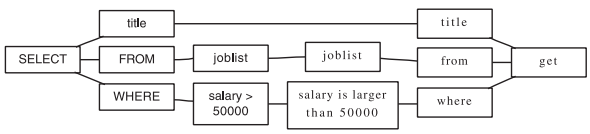


Fig. 3 Clamshell diagram for simple SQL statement.

A straightforward approach is that you get the maximum salary and compare the value with the salary attribute in the joblist table. The SQL statement based on this idea is as follows: "`SELECT title FROM joblist WHERE salary = (SELECT MAX(salary) FROM joblist);`".

You can find the sub-symmetric clamshell diagram which indicates the query "`SELECT MAX(salary) FROM joblist`", which is executable alone. We should remark that if the result of subquery is associated with a relational operator including "=", the value must be single. Instead, in the case of the "IN" predicate, any number of resulting values are permitted although the SELECT subquery has to be specified to get a single column.

Figure 4 is the symmetric clamshell diagram for the statement. The labels in the left half tree forms the given SQL statement, by traveling the left tree in preorder and substitution of the symbol x to the subquery. From the labels in the right half tree, we can obtain the query "get title from joblist where salary is equal to 'maximum of salary from joblist' ", by traveling in preorder and delete the word "get" which occurs at the root of subquery.

4.3 SQL Statement Including EXISTS Predicate

Another way of describing the query for the most highly-paid worker is to determine whether or not there exists a record whose salary is higher than that of the target record; if not, then the target record has the maximum salary. The following is the SQL statement: "`SELECT title FROM joblist j1 WHERE NOT EXISTS (SELECT * FROM joblist j2 WHERE j1.salary < j2.salary);`".

Unlike the statement in the previous section, the subquery "`SELECT * FROM joblist j2 WHERE j1.salary < j2.salary`" is incomplete since "`j1`" is unknown. However the whole statement is legal since `j1` is defined as the alias of the table `joblist`. That is because the scope of the alias assigned in the statement is not directly linked to the dominance relationship in the left or right tree. The corresponding symmetric clamshell diagram is shown in Fig. 5. The right half produces the direction "get title from joblist alias j1 where there does not exist 'record from joblist alias j2 where salary of j1 is smaller than salary of j2' ".

4.4 SQL Statement Including Inline View

The last method is to define an inline view which holds the maximum salary to be compared with the salary of each record in joblist. The SQL statement is as indicated below: "`SELECT title FROM (SELECT MAX(salary) AS max_salary FROM joblist) j, joblist WHERE`

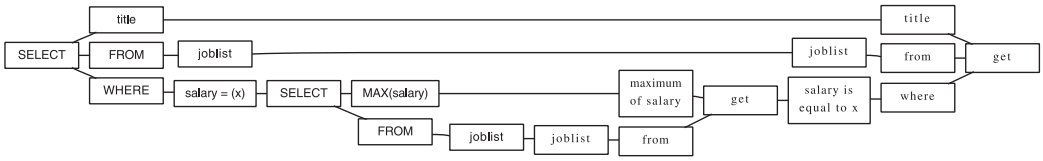


Fig. 4 Clamshell diagram for SQL statement with subquery.

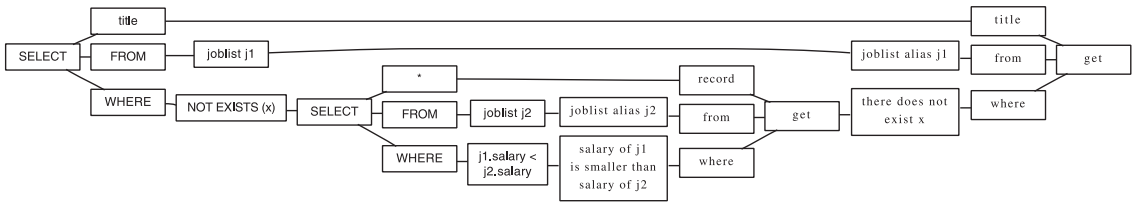
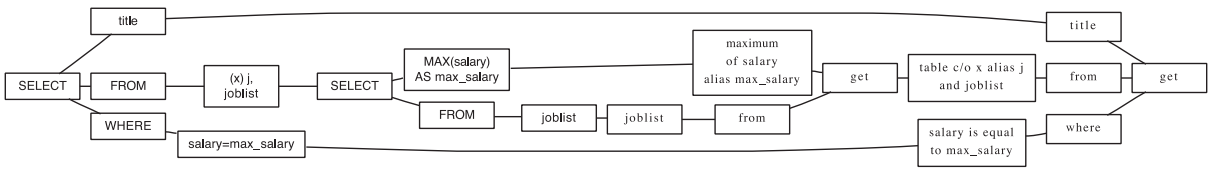


Fig. 5 Clamshell diagram for SQL statement with EXISTS predicate.



c/o: consisting of.

Fig. 6 Clamshell diagram for SQL statement with inline view.

`salary = max_salary;`". The name `j` is not referred but indispensable in the SQL grammar, while `max_salary` is named and used for comparison. The clamshell diagram is shown in Fig. 6. You can restore "get title from table consisting of 'maximum of salary alias max_salary from joblist' alias j and joblist where salary is equal to max_salary" by the right half of the diagram.

Before ending the examples, we would like to make mention of the runtime of the above queries. They all work well using PostgreSQL version 8 on an Ubuntu Linux PC. Where the joblist table contains about 5,000 records randomly generated and is not indexed, each query outputs the correct answer immediately.

5. Discussion

5.1 Visual Effects

Based on the examples last chapter, we are pointing out how to read the runes from the diagram. A simple and effective way is to restore the character strings, namely the query code to the left and the intention to the right. When traversing the right trees on Figs. 4–6, we have had different messages which more or less differ from the original specification, "get the titles of the maximum salary in a given joblist table."

We can also realize the differences of the equivalent SQL statements shown in those figures by focusing attention on the largest box in each diagrams, even though we were unconscious as to the intention of those statements. The largest one in Fig. 5 is located at R.3.1.1.3.1 and reads "salary of j1 is smaller than salary of j2". In addition to the fact that neither the word "maximum" nor the SQL function MAX is found in the diagram, we can make sure that this SQL statement find the record of the maximum salary by a combination of the comparison and the EXISTS predicate. Then, in Fig. 6, the node whose label is "maximum of salary alias max_salary" at R.2.1.1.1 is the largest. The identifier max_salary defined there is seen at R.3.1 and the symmetric positions. While the pair is close apparently, it takes several steps from one to the other via the root, from the viewpoint of graph configuration. This observation leads us to the composition of this SQL statement; the table for the temporary use is defined to compare the sole value with the salary of each record for the selection. Alike those figures, there is not an outstanding box in Fig. 4. It follows that the SQL statement described in Sect. 4.2 is the least confusing, in other words, written in the most straightforward way of the three. We consider that the other characteristic features such as the node with an excess of child nodes or the too long path from end to end could also be the key to learning the statement or the room for improvement to be a comprehensible statement.

The above SQL statements have at most one subquery for simplicity, but our methods can afford more complicated ones. A query for obtaining the median written in [13, p.515] consists of 94 words excluding parenthesis symbols; the keyword SELECT occurs nine times, and some of them forms subquery's subqueries. As the result of drawing, we had a clamshell diagram with 88 nodes where the length of

the path between the both ends is up to 17.

The agvantage of the diagrammatic display in this paper derives from the tree expression and from the symmetry. In constructing the tree in the left or the right part, a subquery is transformed into a node with a symbol put in a label and a subtree whose root reads "SELECT" or "get". This dissolution clarifies what the subquery intends to acquire, and the parent node of the subtree demonstrates how the result of the subquery is referred to. Moreover the diagram offers a clue to know the purpose of the parted code in terms of the programming language and the natural language. That is, if the label on a node together with the surroundings is cryptic, then one can switch the viewpoint to the symmetrical location to decipher it. Both benefits never fade for a larger diagram, while the longer statement is harder to understand empirically.

These results show the usefulness of applying our schematic tool to SQL, in other words, applying an idea creation support system to the lower process of software development.

Yet another problem is adaptability to other natural languages than English, say Japanese or Korean that differs in terms of word order. A solution is to change the configuration of nodes in the right tree, after labeling each nodes in the intended language (see Fig. 2 (d)), so that the right tree can be more readable. Note that this approach is in exchange for the symmetry and does not always connect to the leaves. We drew diagrams for the SQL statements in the previous chapter to make sure that Japanese instructions are generated by traversing the right trees in postorder.

5.2 Related Works about SQL Inspection

As far as the authors know, the tool for static analysis of SQL statements is not developed well. That seems to be because each sentence is quite small, in comparison with the ones written in established procedural languages like C or popular embedded languages such as PHP. Actually some programmers describe the SQL statement as a string within the code for the logic, as if he or she attempted to get the Web content by giving a URL. Although solicitous programmers properly write the code on error checking after invoking the database query, few persons take into account the validity of the query; some sentences might be derived from an existing system, without being reviewed; some SQL string has a variable name in it to be replaced though there may be a security hole, or a vulnerability to SQL injection attack, behind the code. Recently supporting methods or systems set up against SQL injection attack have been reported to reduce the programmers' burden [14]–[17]. For example, SQLProb [17] analyzes a given SQL statement including variables to form the tree, and compares it with the one constructed from the query with values into the variables; if the configuration is changed, then it considers that the substitution changes the query and therefore the sender would attempt an SQL injection attack.

Those tools aimed at the automation but not at the support of manual validity check. In addition the SQL example statements in those papers are too small and simple for us to know that they are available in a complicated case. Our approach has an advantage in the sense that a diagram is generated from a given SQL statement which may include confusing subqueries so that the programmers or reviewers can confirm the intention of the query multilaterally, although it takes the automatization into consideration as well. Moreover our framework deals with the query using placeholder where the symbol "?" is mounted in the SQL statement usually, by interpreting the sign as a special meaning.

Visual Explain [18] is, based on a different approach, a visualization tool of SQL statements. It generates the access plan, or the paths of database access, by means of a tree for a given SQL statement. The tree is configured from bottom up and the root means the status where all the internal processes are done. Each node holds the cost which is an estimated figure for the required resource about CPU and input-output. For the same statement, different access plans are drawn according to the presence or absence of indices. This property is useful for making sure that the indexing is effective. Moreover, the node does not have the label of SQL fragment but that of internal process such as scanning or sorting of the table. The above features make us consider that Visual Explain is better suited for the programmers who follow through the efficiency of the database access, but that it would be poor at the code inspection support by a single person or multiple persons.

5.3 Repository of SQL Example Statements

To register and maintain a wide variety of SQL statements, the data store is the cornerstone. We explain a plot of the *repository* for SQL statements. The issue includes (1) from what to collect and (2) how to store.

Collecting and selecting the actual SQL statements holds the key to the success of the constructing code repository. We are making a collection of them not only from books but from our development of database applications as well as the classes of database.

To draw a clamshell diagram from an existing SQL statement, it would be impossible to find comment of the statement except for written in a book. We will annotate the code segments by hand initially, and subsequently implement the feature of the automatic comment. When each statement is divided adequately, the code segments are stored in the database together with the corresponding comments. In addition, the data for composing a clamshell diagram including the references to the code segments are preserved. By managing the parts and the constitution of SQL statements separately, we expect that the parts will be reused and we will know the usage of the keywords and phrases on SQL, beyond mere frequency count.

Many books for writing SQL statements have been published but the readers cannot invoke the code to verify it. We also attempt to construct a collection of verifiable SQL statements together with execution environments and

data set to which the query is applied. Since the learners can see the result quickly, they will be able to know not only the syntax of SQL but good or bad SQL statements in a practical sense efficiently. It may be useful to those who brush up on SQL as well. Finally the repository of SQL statements will be informative and helpful to the programmers of database manipulation from beginners to experts.

6. Conclusions

In this paper we have proposed the framework for drawing symmetric clamshell diagram provided an SQL statement. The configuration of the diagram with a pair of trees helps one to understand the whole and the parts of the query. In addition, the statement with subquery can be expressed as the grafting. Data structure by means of symmetric trees has two effects. The symmetric clamshell diagram presents more visual attraction than the diagram by a single tree, while all the components can be described by a tree-based format, say as an XML document.

Future works include conducting a quantitative evaluation with regard to the readability and constructing a practical repository of SQL statements so that the diagrams can be efficiently produced. It is also a promising approach to apply the method to shell scripts which have been widely used in the Unix operating systems. The length of a command in a shell does not make much difference from that of an SQL statement, and we can find a similarity in nesting; a command can be made longer by combining commands using && or ||, a pipe, and a subshell.

References

[1] H. Mase, H. Kinukawa, H. Morii, M. Nakao, and Y. Hatamura, "Mechanical design support system based on thinking process development diagram," Trans. Japanese Society for Artificial Intelligence, vol.17, pp.94–103, 2002.
[2] T. Murakawa, T. Kawasaki, H. Mizuochi, and M. Nakagawa, "Formulation of clamshell diagram and its application to source code reading," Proc. Eighth Joint Conference on Knowledge-Based Software Engineering (JCKBSE 2008), pp.474–483, 2008.
[3] T. Kawasaki and T. Murakawa, "Applying thinking process development diagram to source code reading support," FIT2008, pp.117–118, 2008.
[4] T. Murakawa and M. Nakagawa, "Graphical expression of SQL statements using clamshell diagram," J. Japan Society of Information and Knowledge, vol.19, no.2, pp.218–223, 2009.
[5] J. Kawakita, KJ Method: A Scientific Approach to Problem Solving, Kawakita Research Institute, 1975.
[6] K. Yagishita, J. Munemori, and M. Sudo, "A proposal and an application of an evaluation method for sentences of B type KJ method based on contents and structures," Trans. IPSJ, vol.39, no.7, pp.2029–2042, 1998.
[7] T. Buzan and B. Buzan, The Mind Map Book: How to Use Radiant Thinking to Maximize Your Brain's Untapped Potential, Dutton, 1994.
[8] T.J. Snodgrass and M. Kasi, Function analysis: the stepping stones to good value, University of Wisconsin, 1986.
[9] R.B. Grady, "Successfully applying software metrics," Computer, vol.27, no.9, pp.18–25, 1994.
[10] G. Altshuller, L. Shulyak, and S. Rodman, 40 Principles: TRIZ Keys to Technical Innovation, Technical Innovation Center, 1998.
[11] M. Schlueter, "TRIZ for Perl-programming," Proc. TRIZCON2001, 2001.
[12] "Information technology—Database languages—SQL—Part 1: Framework (SQL/framework)," ISO/IEC 9075-1, 1999.
[13] J. Celko, Joe Celko's SQL for Smarties: Advanced SQL Programming, Third Edition, Morgan Kaufmann, 2005.
[14] C. Gould, Z. Su, and P. Devanbu, "JDBC checker: A static analysis tool for SQL/JDBC applications," Proc. 26th International Conference on Software Engineering, pp.697–698, 2004.
[15] W.G.J. Halfond and A. Orso, "AMNESIA: Analysis and monitoring for neutralizing SQL-injection attacks," Proc. 20th IEEE/ACM International Conference on Automated Software Engineering, pp.174–183, 2005.
[16] G. Buehrer, B.W. Weide, and P.A.G. Sivilotti, "Using parse tree validation to prevent SQL injection attacks," Proc. 5th International Workshop on Software Engineering and Middleware, pp.106–113, 2005.
[17] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou, "SQLProb: A proxy-based architecture towards preventing SQL injection attacks," Proc. 2009 ACM Symposium on Applied Computing, pp.2054–2061, 2009.
[18] "DB2 9.1 Visual Explain tutorial," IBM, 2006.

Takehiko Murakawa was born in 1971. He received the Ph.D. degree from Nara Institute of Science and Technology, in 1998. Through a Research Associate of Nara Institute of Science and Technology and Wakayama University, since 2003, he has been an Associate Professor at the Department of Computer and Communication Sciences, Faculty of Systems Engineering, Wakayama University. His research interests include database system and digital archive.

Masaru Nakagawa graduated Osaka University in 1970 and received M.E. degree from the same university in 1972. In 1972–1994, he worked at Musashino Research Laboratory, NTT. Through a Professor of Kinki University, since 1997, he has been a Professor at the Department of Computer and Communication Sciences, Faculty of Systems Engineering, Wakayama University. He is a member of Information Processing Society of Japan, Japan Society of Information and Knowledge, and the Japanese Society for Artificial Intelligence.