Syntax Validation of the SQL Conditional Expression Based on Postfix-Expression Evaluation

Wei Lijun[1,2,a], Pan Yang[1,2,b],Wang Hao[1,c], Zhang Yan[1,d]

[1]School of Computer and Information Engineering, Fuyang Normal College, Fuyang, Anhui, China

[2] Northern Anhui Culture Research Center, Fuyang Normal College, Fuyang, Anhui, China

[a]email: 32182158@qq.com, [b]email:77694303@qq.com, [c]email: 393950571@qq.com, [d]email: 34013062@qq.com

Keywords: Conditional Expression; Syntactic Logic; Data Type; Postfix Expression; Syntax Validation

Abstract. The SQL conditional expression is not a constant expression, making it impossible to judge its legitimacy through the final value. However, there exists one alternative way where the elements in expression should be identified first to obtain the corresponding postfix data type expression, and the algorithms of SQL data types should be set to evaluate the postfix expression, then we can judge the syntactic legitimacy of the conditional expression in accordance with the final results. Tests conducted in this research have proved that this method can effectively validate the legitimacy of the SQL conditional expression and significantly serve for custom query and data analysis patterns.

Introduction

The SQL-Select is very powerful in statistics and analysis. However, its users are confined to professionals who are knowledgeable in structures and relationship within database, and proficient in Select instructions, hence the ordinary users can obtain data merely through integrated reports. In this context, various requirements of data analysis are far more difficult to meet. However, there is one means available where users need to define query choices and query conditions according to their need, meanwhile programs themselves should interpret data dependency, the legitimacy of query logic and query conditions, as well as submit the query instruction. The research focuses on the syntax validation of the SQL conditional expression.

Model of Syntax Validation

The SQL-Select conditional expression is an infix expression which comprises of three valid elements, namely, operand, operator and delimiter, such as “(student.sex=’M’ and student.age<20) or score. grade>80”, or more complex form. To be specific, the operand consists of database fields and the feature data; the operator includes different types involving arithmetical, relational, logic and reference type; delimiter contains left parenthesis, right parenthesis and so on. The conditional expression must conform to the SQL syntax rules in order to be correctly interpreted [1]. The syntax rules of conditional expression include legitimate elements, matching parenthesis, legitimate operation and valid logic final value etc.

However, the conditional expression is not necessarily constant expression, making it impossible to validate its legitimacy through evaluating the final value. Despite of this, researches have found this validation actually can be achieved if the three works are done: the operands within the conditional expression are replaced by the corresponding data type; algorithms of SQL data types are established and the final value of postfix data type expressions is evaluated. The model can be shown as Figure 1.

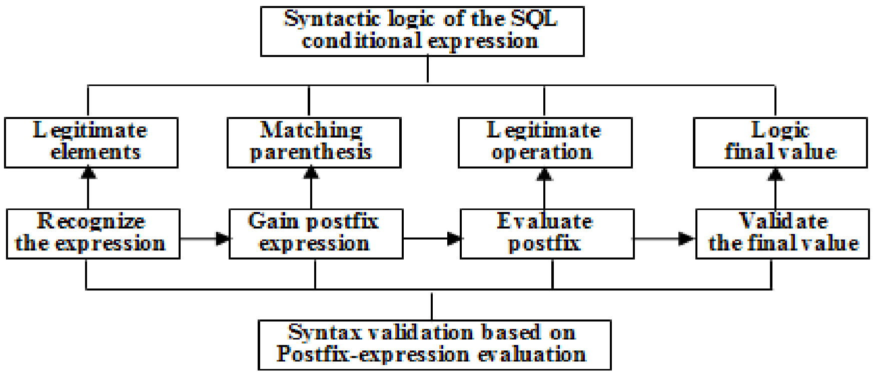

Fig.1. The model of syntax validation

Expression Recognition

In essence, the SQL conditional expression is a sequence of characters, marked by delimiter, operator and separator. In accordance with certain rules of recognition, elements in expressions can be identified and interpreted in turn [2] [3]. Attention: (1) the space character and the tab character out of "’’" or "{}" are generally deletable separators; (2) character "." out of "’’" is the reference operator of table and its fields, or the decimal point in real number; (3) If the nearest valid element before "+" or "-" is not an operand, it should be regarded as an unary operator; (4) If operators overlap like ">", ">=" and "=", they should be matched from long to short; (5) characters like "’" and "{" or "}"should be identified in pairs and in order, and the corresponding elements are either types of text or types of date time. Date time constants include connectors like "/", "-", ":" and there are specific requirements for their formats. In the procedure of identifying elements, the legitimacy of elements should also be verified. Legitimate and valid elements embrace table name, field name, operator, delimiter and feature data. Once invalid elements are recognized, there will appear a prompt, and the validation will stop immediately.

It demands defining the information about database fields first, which will be the gist for judging the validity of the database table name and the field name, shown as Table 1. Broad data categories are used to identify the operations that fields can participate in. Field types of SQL such as char(n), varchar(n), varchar(max) and text belong to text; nchar(n), nvarchar(n), nvarchar(max) and ntext belong to ntext; tinyint, smallint, int and bigint belong to bigint; decimal(p,s), numeric(p,s), smallmoney, money, float(n) and real belong to real; datetime, datetime2, smalldatetime and datetimeoffset belong to datetime, while date and time are themselves in their broad categories respectively. Table 2 presents the basic operators and delimiter of SQL. As it is shown, the smaller the priority is, the higher the rank is; Alternate characters are set to distinguish various meaning and priorities on the same operator. The operators "in", "between and" can be equal to the combination of basic relational and logic operation [4].

Table 1 Example of Fields

Number	Field Name	Alias Name	Source Table	Data Type	Broad Category
F0001	Sno	Number	Student_inf	char(n)	text
F0002	Sxh	Student number	Student_inf	char(n)	text
F0003	Sname	Name	Student_inf	nchar(n)	ntext
…	…	…	…	…	

Table 2 SQL Basic Operators

Type	Operator	Operation	Separato	Priority
binary	.	reference		1
unary	+	pick the positive	⊞	2
unary	-	pick the negative	⊟	2
binary	*	multiply		3
binary	/	divide		3
binary	%	pick the reminder		3
binary	+	plus/connect		4
Binary	-	minus		4
binary	like	like		5
binary	>	more than		5
binary	>=	no less than		5
binary	<	less than		5
binary	<=	no more than		5
binary	=	equal to		5
binary	!=	unequal to		5
unary	not	not		6
binary	and	and		7
binary	or	or		8
delimiter	(	left parenthesis		0
delimiter	)	right parenthesis		0

The following procedures based on features of the first character can help to identify elements in expressions.

(1) If the first character is a separator, it should be cut out and discarded;

(2) If the first character is "("or")", it should be cut out then used as a valid delimiter;

(3) If the first character is an operator or its initial character, identify its validity, cut out it as operator or not if it's invalid;

(4) If the first character is "'" or "{", try to find the first unescaped "'" or "}" in sequence. Once find, cut out the substring from sequence and regard them as constant of text or date/time. If not find, it indicates the expression is not legitimate;

(5) If the first character is a digit, try to find the first character of non-numeric type in sequence, excluding ".". Once find, cut out the substring from sequence and regard it as constant of numeric type;

(6) If the first character is a Chinese character or a letter, try to find the first operator or separator or delimiter or "'" or "{", Once find, cut out the substring from sequence and regard them as database table name or field name or else.

(7) If the first character is different from what mentioned above, thus the expression is not legitimate;

(8) The judgment criteria over elements identified in (4), (5), (6) are naming rules, constitute rules and the inherent information list. If it is legitimate, identify the next one; or the expression is not legitimate.

Finally, classify and store all the legitimate elements in the expression to make it easy in future retrieval and solution. Set N as the number of valid elements, table[N] stores the corresponding table name; fields[N], field name; data[N], features data; operator[N], operator or parentheses. These four sets united together represent the SQL infix expression, and the member will be placed NULL when the expressions without corresponding element.

Postfix Data Type Expression

The following procedures are used to obtain the postfix data type expression corresponding to the SQL conditional expression [5]. The operand is replaced by its data type.

Set suffix_exp[N] as the stored element of postfix expression; operstack[C], temporary stack for operator; i, j and k, variables; the initial value of suffix_exp[N] and operstack[C], Null. Set i=1 and both j and k begin with 0.

(1) If i>N, turn to the eighth step;

(2) Read the unique non-NULL elements with subscript i from table[N], fields[N], data[N] and operator[N]. If the element comes from operator[N], it is operator. Turn to the fourth step;

(3) Identify data type of the operand in following ways. Set k=k+1. Insert it into suffix_exp[k]. Set i=i+1, Turn to the first step.

Identification methods:

①. Any element coming from table[N] is a database table name, thus its type is ”table”.

②. Any element coming from fields[N] is a database field name, thus its type belongs to the corresponding broad category in Table 1.

③. Any element coming from data [N] is a feature data. Type of constant can be interpreted into one among different types below according to the context and the structural features of the element. They are text type（”text”, “ntext”）, numerical type (“bigint”, “real”) and date/time type (“datetime”, “date”, “time”). In numerical type, an element with decimal belongs to “real”, or “bigint” if not. In text type, if it is feasible to identify the type of operand operated in priority with this element, then choose its type, or else choose one from “text” or “ntext”. Date/time type is identified by observing whether “/”, “-” and “:” appear in elements at the same time. If the element is other systematic or self-defined value, convert it to the broad category on the basis of definition types.

(4) If the element is “(”, then j=j+1, push “(” into operstack[j];

(5) If the element is “)”, then pop off operators from operstack[j] one by one and input suffix_exp[k] until “(” appears, also pop off“(” then set j=j-1. Otherwise, if there is no “(”, it indicated the absence of “(” in the infix expression, then prompt its illegitimacy and stop here;

(6) If the element is an operator excluding “(” and “)”, compare the priority between this element and operstack[j] according to Table 2. If its priority ranks higher than (the value is smaller than) that of operstack[j], set j=j+1 and push it into the top of the stack; otherwise, set k=k+1 and pop off operstack[j] then input suffix_exp[k], and push this element into the top of the stack;

(7) Set i=i+1，turn to the first step;

(8) If there still remain operators in stack, pop off them from operstack[j] then input suffix_exp[k] one by one. If there appears “(”, it indicated the absence of “)” in the infix expression, then prompt its illegitimacy and stop here;

(9) Output the postfix expression: suffix_exp[N].

Such postfix expression consists of data type identifiers and operator, with parentheses matching and priorities analyzed simultaneously. For instance, the postfix expression of “(student.sex=’M’ and student.age<20) or score.score>80” is “table ntext . ntext = table bigint . bigint < and table real . bigint > or”.

Syntax Validation

Algorithms of SQL Data types

To evaluate the postfix data type expression, defining the algorithms of SQL data types is required as the basis of evaluating and validating. Such algorithms, as shown in Table 3, covers all the basic data types of SQL, self-defined data types, basic operators, operations in the data type and types of operating results. Among self-defined data types, “table” stands for the type of table; and “logic”, the type of logic. The following table set “>” as an example, which is the same with the other relational operator in Table 2.

Table 3 Algorithms of SQL data types

Operator	Data Type1	Data Type2	Result	Operator	Data Type1	Data Type2	Result
.	table	bigint	bigint	+	ntext	ntext	ntext
.	table	real	real	+	text	ntext	ntext
.	table	text	text	-	bigint	bigint	bigint
.	table	ntext	ntext	-	real	real	real
.	table	datatime	datatime	-	real	bigint	real
.	table	data	data	like	text	text	logic
.	table	time	time	like	ntext	ntext	logic
$\overset{+}{\square}$	bigint		bigint	like	text	ntext	logic
$\overset{+}{\square}$	real		real	>	bigint	bigint	logic
$\overset{-}{\square}$	bigint		bigint	>	real	real	logic
$\overset{-}{\square}$	real		real	>	bigint	real	logic
*	bigint	bigint	bigint	>	text	text	logic
*	bigint	real	real	>	ntext	ntext	logic
*	real	real	real	>	text	ntext	logic
/	bigint	bigint	real	>	datatime	datatime	logic
/	real	real	real	>	data	data	logic
/	real	bigint	real	>	time	time	logic
%	bigint	bigint	bigint	…	…	…	…
+	bigint	bigint	bigint	not	logic		logic
+	real	real	real	and	logic	logic	logic
+	bigint	real	real	or	logic	logic	logic
+	text	text	text				

Evaluation and Validation

The following procedures, similar to what have been mentioned in evaluating the postfix constant expression, are designed for three proposes: to evaluate the postfix data type expression; to validate the legitimacy of the operation in conditional expression, and to validate the final logic value [6] [7].

Set datastack[N] as a place to store operands(here is data type); i and j, variables; suffix_exp[N], the targeted postfix expression; and N, the number of item in the postfix expression.

(1) Set the initial value of datastack[N] NULL, i=1, j=0;

(2) If i>N, turn to the fifth step, or read suffix_exp[i];

(3) If suffix_exp[i] is not an operator, let j=j+1, then push suffix_exp[i] into datastack[j] and let i=i+1, back to the second step;

(4) If suffix_exp[i] is an operator, take the following steps:

① Pop off a certain number of operands required by the operator from datastack[j], regard its quantity as the mesh number of this operator. If that quantity is smaller than the later, it means this operation violates the rules. Prompt its illegitimacy and stop here;

② If the corresponding relations of operands and the operator appears in Table 3, then put the corresponding result type into datastack[j]; if not, this operation must violate the rules, prompt its illegitimacy and stop here;

③ Set i=i+1, then turn to the second step.

(5) Make the final judgment after finishing solution: If there is only one operand “logic” left in datastack[N], it means such conditional expression complies with SQL syntactic rules, then mark its legitimacy; if not, mark the illegitimacy.

This algorithm transfers the process of evaluating postfix expression to operating data type expression, making it possible to validating the syntactic legitimacy of the SQL conditional expression based on the final results.

Validation Tests

Set the below an legitimate example: “table ntext . ntext = table bigint . bigint < and table real . bigint > or”. Here are the procedures of validation.

(1) Take the first element from the above expression, “table”, then push it into stack, datastack[1]=table;

(2) Take the second element, “ntext”, then push it into stack, datastack[2]=ntext;

(3) Take the third, “.”, then pop off “ntext” and “table” from datastack[2] and datastack[1] separately. Both operator and operands are in accordance with Table 3. Push the result into stack, datastack[1]=ntext;

(4) Take the fourth, “ntext”, then push it into stack, datastack[2]=ntext;

(5) Take the fifth, “=”, then pop off “ntext” and “ntext” from datastack[2] and datastack[1] separately. Both operator and operands are in accordance with Table 3(like “>”). Push the result into stack, datastack[1]=logic;

(6) Take the sixth, “table”, then push it into stack, datastack[2]=table;

(7) Take the seventh, “bigint”, then push it into stack, datastack[3]=bigint;

(8) Take the eighth, “.”, pop off “bigint” and “table” from datastack[3] and datastack[2] separately. Both operator and operands are in accordance with Table 3. Push the result into stack, datastack[2]=bigint;

(9) Take the ninth, “bigint”, then push it into stack, datastack[3]=bigint;

(10) Take “<”, pop off “bigint” and “bigint” from datastack[3] and datastack[2] separately. Both operator and operands are in accordance with Table 3(like “>”). Push the result into stack, datastack[2]=logic;

(11) Take “and”, pop off “logic” and “logic” from datastack[2] and datastack[1] separately. Both operator and operands are in accordance with Table 3. Push the result into stack, datastack[1]=logic;

(12) Take “table”, then push it into stack, datastack[2]=table;

(13) Take “real”, then push it into stack, datastack[3]=real;

(14) Take “.”, pop off “real” and “table” from datastack[3] and datastack[2] separately. Both operator and operands are in accordance with Table 3. Push the result into stack, datastack[2]=real;

(15) Take “bigint”, then push it into stack, datastack[3]=bigint;

(16) Take “>”, pop off “bigint” and “real” from datastack[3] and datastack[2] separately. Both operator and operands are in accordance with Table 3. Push the result into stack, datastack[2]=logic;

(17) Take the last element, “or”, pop off “logic” and “logic” from datastack[2] and datastack[1] separately. Both operator and operands are in accordance with Table 3. Push the result into stack, datastack[1]=logic;

(18) The operation upon postfix expression ends here. It is shown that only datastack[1] is left in stack with its value being “logic”. Therefore, such conditional expression abides by the syntactic rules of SQL.

Parentheses are matched successfully on another illegitimate expression “student .age > (20 + student .score >) 60”, whose postfix data type expression is “table bigint . bigint table real . > + bigint >”. Here are the procedures of validation:

(1) Take the first element from the above expression, “table”, then push it into stack, datastack[1]=table;

(2) Take the second, “bigint”, then push it into stack, datastack[2]=bigint;

(3) Take the third, “.”, pop off “bigint” and “table” from datastack[2] and datastack[1] separately. Both operator and operands are in accordance with Table 3. datastack[1]= bigint;

(4) Take the fourth, “bigint”, then push it into stack, datastack[2]=bigint;

(5) Take the fifth, “table”, then push it into stack, datastack[3]=table;

(6) Take “real”, then push it into stack, datastack[4]=real;

(7) Take “.”, pop off “real” and “table” from datastack[4] and datastack[3] separately. Both operator and operands are in accordance with Table 3. datastack[3]=real;

(8) Take “>”, pop off “real” and “bigint” from datastack[3] and datastack[2] separately. Both operator and operands are in accordance with Table 3. datastack[2]=logic;

(9) Take the next one, “+”, pop off “logic” and “bigint” from datastack[2] and datastack[1] separately. Both operator and operands are not in accordance with Table 3. Prompt syntactic error and stop here.

Conclusion

Tests performed in this research have proved that such method can effectively validate the legitimacy of the SQL conditional expression, and have an extensive application potential in custom query and data analysis patterns. In current, this works’ attention focuses on the basic data types and operators, but will shift to more challenging syntax validation of conditional expression where the data type is more complicate and matches are ambiguous in the future.

Acknowledgement

In this paper, the research was sponsored by the Key Projects of National Statistical Research (Project No. 2014LZ32) and the Research Projects of Fuyang Normal College (Project No. 2013KSLZX06，2015KJFH02，2012WBZX01).

References:

[1] Raghu Ramakrishnan / Johannes Gehrke / Raghu Ramakrishnan / Johannes Gehrke .Database Management Systems[M]. McGraw-Hill Science/Engineering/Math, 2010.

[2] Lei Jie, Cheng Lian, Li Ming. Validity judging method of mathematical expression [J]. Journal of Computer Applications, 2015, 35(S1).

[3] Wang Yonghao. Application of expression interpreter in workflow management system [J]. Computer Engineering and Design, 2007, 28(12).

[4] Wu Xiaojun, Gu Jianhua, Zhou Xingshe. Prioritized Operators Algorithm Based Logic Expression Validity Judgement [J]. Mini-Micro System, 2002, 23(10).

[5] Lan Meihui, Yang Ping. Research on Arithmetic Expression Evaluation Algorithm based on Stack Structure [J]. Journal of Qujing Normal University, 2014, 33(3).

[6] Ying Changsheng. Expression Evaluation and Sign Derivation [J]. Journal of Chang Chun University, 2012, 22(10).

[7] Xong Fengguang, Kuang Liqun, Han Yan. Design and implementation of expandable logical expression evaluation [J]. Computer Engineering and Design, 2012, 33(10).