Complementary Material for Relational Conditional Set Operations

Alexis I. Aspauza Lescano[1][0000−0001−8685−010X] and Robson L. F. Cordeiro[1][0000−0002−6795−3004]

University of São Paulo – USP, Av. Trab. São Carlense 400, São Carlos, SP, Brazil

This is a complementary document related to the paper: "Relational Conditional Set Operations" submitted to ADBIS 2021. In the paper, we presented the new Relational Conditional Set Operations or **RelCond Set Operations** for short. They include the RelCond Set Membership ($\in_c$), RelCond Subset ($\subseteq_c$), RelCond Intersection ($\cap_c$), and RelCond Difference ($-_c$). Also, there, we mentioned 3 algorithms that would be described in this complementary document. Here, in order to explore those algorithms, we divided this document in three sections: (1) the algorithm to convert an infix expression to postfix, (2) the extended relational conditional set operations which support index structures either in left or right relation, and, (3) the algorithm to support the relational conditional union.

1 Infix To Postifx Expression

We present in this section the Algorithm 1, which converts an infix expression to a postfix expression. We can see some examples of these conversions in Table 1.

Table 1: Examples of Infix and Posfix equivalences.

Infix Expression	Postfix Expression
DP.Category = SP.Category	DP.Category SP.Category =
DP.Category = SP.Category AND DP.Product = SP.Product	DP.Category SP.Category = DP.Product SP.Product = AND
DP.Category = SP.Category AND DP.Product = SP.Product AND (DP.Units $\leq$ SP.Units AND $\neg$ (DP.Price $<$ SP.Price) OR DP.Price / 2 $\geq$ SP.Price)	DP.Category SP.Category = DP.Product SP.Product = AND DP.Units SP.Units $\leq$ DP.Price SP.Price $<$ $\neg$ AND DP.Price 2 / SP.Price $\geq$ OR AND

Starting the algorithm, in lines 1 and 2, we need to create a stack of pending operations and a vector to contain the postfix expression to be returned. Also, in the initialization, line 3 pushes an open bracket "(" onto the stack of pending

Algorithm 1 ToPostfix(*infixExp*)

Input: *infixExp*: Expression in infix notation
Output: *postfixExp*: Expression in postfix notation

```
 1: create stack pendingOps;
 2: create vector postfixExp;
 3: push "(" onto pendingOps;
 4: add ")" to the end of infixExp;
 5: for each token extracted from infixExp do
 6:     if token is an operand then
 7:         add token to postfixExp;
 8:     else if token is "(" then
 9:         add token to pendingOps;
10:     else if token is ")" then
11:         lastToken = pop from pendingOps;
12:         while lastToken != "(" do
13:             add lastToken to postfixExp;
14:             lastToken = pop from pendingOps;
15:         end while
16:     else if token is an operator then
17:         while top of pendingOps is an operator and it has equal or higher precedence than token do
18:             lastToken = pop from pendingOps;
19:             add lastToken to postfixExp;
20:         end while
21:         add token to pendingOps;
22:     end if
23: end for
24: return postfixExp;
```

operations, as well as line 4, adds the closing bracket ")" to the end of our input expression. Lines 5 to 23 are the main loop, which will get every single token extracted from the original expression in infix notation ($infixExp$). The current token being read will be processed according to its type. If the token is an operand (any constant or reference to a column table), it will be added directly to the postfix expression vector. In case the token is an opening bracket "(", it will be added to the stack of pending operations. But if the token is a closing bracket ")", we will pop tokens from the pending operations and add them to the postfix expression until an opening bracket is found, and the bracket will just be discarded. As the last option, if the token is an operator of any type (logical negation, arithmetic operator, logical operator, or logical connector), we will add to the postfix expression all top operators from the pending operations while the top is an operator with higher or equal precedence than the current token; and then, add the token to the stack of pending operations. The next operators are ordered by higher to lower precedence:

1. ¬
2. *, /   both with same precedence

3. $+$, - both with same precedence
4. $<, \leq, >, \geq, =, \neq$ both with same precedence
5. $\wedge$
6. $\vee$

Finally, we will return our postfix expression vector.

2 Relational Conditional Set Operations

In the paper, we assumed the index structures are always in the right relation. Here, we present an algorithm that is capable of executing the RelCond Set Operations if there are indexes either for the left or for the right relation.

Algorithm 2 RelCondSetOp($_c\mathsf{T}_1$, $_c\mathsf{T}_2$, c, $SetOp$)

Input: $_c\mathsf{T}_1$, $_c\mathsf{T}_2$: relational conditional sets, c: predicate, $SetOp$: $\cap_c$ or $-_c$
Output: the result $_c\mathsf{T}_R$ from $_c\mathsf{T}_1 \cap_c {}_c\mathsf{T}_2$ or from $_c\mathsf{T}_1 -_c {}_c\mathsf{T}_2$, according to $SetOp$

```
Create a vector of bits $R$ with size of $|_c\mathsf{T}_1|$;
if $SetOp$ is $\cap_c$ then
    Initialize $R$ with 0's;
else if $SetOp$ is $-_c$ then
    Initialize $R$ with 1's;
end if
if $_c\mathsf{T}_2$ is the table with index structures then
    for each tuple $t_i$ of $_c\mathsf{T}_1$ do
        if IsCondMember($_c\mathsf{T}_2$, $t_i$, c) then
            if $SetOp$ is $\cap_c$ then
                set $R[i]$ as 1;
            else
                set $R[i]$ as 0;
            end if
        end if
    end for
else if $_c\mathsf{T}_1$ is the table with index structures then
    for each tuple $t_j$ in $_c\mathsf{T}_2$ do
        $S$ = Index_TupleQuery($_c\mathsf{T}_1$, $t_j$, $c$);
        if $SetOp$ is $\cap_c$ then
            $R = R \vee S$;
        else
            $R = R \wedge \neg S$;
        end if
    end for
end if
$_c\mathsf{T}_R$ = get tuples for all true bits of $R$;
return $_c\mathsf{T}_R$;
```

Algorithm 2 implements both the conditional intersection $\cap_c$ and the conditional difference $-_c$. It receives as parameters the left relation ${}_c\mathsf{T}_1$, the right relation ${}_c\mathsf{T}_2$, the predicate c, and an indicator $SetOp \in \{\cap_c,\ -_c\}$ of the operation of interest. The result ${}_c\mathsf{T}_R$ is either ${}_c\mathsf{T}_1 \cap_c {}_c\mathsf{T}_2$ or ${}_c\mathsf{T}_1 -_c {}_c\mathsf{T}_2$, according to $SetOp$. As it is shown in Lines 1-6, the algorithm begins by creating an array of bits R to be used later to indicate the tuples of ${}_c\mathsf{T}_1$ that should be in ${}_c\mathsf{T}_R$. If $SetOp = \cap_c$, R is initialized with 0s; otherwise, it receives 1s. Now, as a first possibility as it is shown in lines 7-16, is that the right relation is the one with index structures. This is the case we presented in the paper. Here, there is a loop in lines 8-16 iterating for each tuple $t_i \in {}_c\mathsf{T}_1$. The loop updates array R only when the current tuple t_i is a conditional member of ${}_c\mathsf{T}_2$. In this case, $R[i]$ receives 1 if $SetOp = \cap_c$; otherwise, $R[i]$ is set to 0. As the second possibility, we could have the index structures not in the right relation but in the left one, as it is shown in lines 17-26. Here, there is a loop in lines 18-26 iterating for each tuple $t_j \in {}_c\mathsf{T}_2$. The first step in this loop is to compute the sub-result S as a bits vector indicating for each tuple of ${}_c\mathsf{T}_1$ if it satisfies the predicate with t_j. In this case, we will update our result R with a bit-wise operation of $R \vee S$ if $SetOp = \cap_c$; otherwise, R is set to $R \wedge S$. Finally, ${}_c\mathsf{T}_R$ is obtained as being the tuples of ${}_c\mathsf{T}_1$ that refer to each bit 1 in R.

3 Relational Conditional Union

In the paper, we talked about the possibility of having an algorithm for conditional union queries. There we mentioned that we didn't find any utility for it. However, we also pointed that it would be easy to adapt our previous algorithms to support this new operation. Here, we present in Algorithm 3, the extended RelCondSetOp operator to support the two previous operations, relcond intersection ($\cap_c$) and relcond difference ($-_c$), along with the **RelCond Union ($\cup_c$)**.

Algorithm 3 receives as parameters the left relation ${}_c\mathsf{T}_1$, the right relation ${}_c\mathsf{T}_2$, the predicate c, and an indicator $SetOp \in \{\cap_c,\ -_c,\ \cup_c\}$ of the operation of interest. The result ${}_c\mathsf{T}_R$ is either ${}_c\mathsf{T}_1 \cap_c {}_c\mathsf{T}_2$, ${}_c\mathsf{T}_1 -_c {}_c\mathsf{T}_2$, or ${}_c\mathsf{T}_1 \cup_c {}_c\mathsf{T}_2$, according to $SetOp$. As it is shown in Lines 1-8, the algorithm begins by creating the arrys of bits R_1 and R_2 to be used latter to indicate the tuples of ${}_c\mathsf{T}_1$ that should be in ${}_c\mathsf{T}_{R1}$ and the tuples of ${}_c\mathsf{T}_2$ that should be in ${}_c\mathsf{T}_{R2}$ respectively. If $SetOp = \cap_c$, R_1 is initialized with 0s; otherwise, it receives 1s. In all cases, R_2 will be initialized with 0s. The main loop in Lines 9-18 iterates for each tuple $t_i \in {}_c\mathsf{T}_1$. The first step in this loop is to compute the sub-result S as a bits vector indicating for each tuple of ${}_c\mathsf{T}_2$ if it satisfies the predicate with t_i. Then, if $SetOp = \cap_c$ and any bit of S is 1, $R[i]$ receives 1. If instead $SetOp = -_c$ and any bit of S is 1, $R[i]$ receives 1. Otherwise, R_2 is set to $R_2 \vee S$. Finally, ${}_c\mathsf{T}_{R1}$ is obtained as being the tuples of ${}_c\mathsf{T}_1$ that refer to each bit 1 in R_1, ${}_c\mathsf{T}_{R2}$ is obtained as being the tuples of ${}_c\mathsf{T}_2$ that refer to each bit 1 in R_2 and the union of both ${}_c\mathsf{T}_{R1} \cup {}_c\mathsf{T}_{R2}$ is returned.

Algorithm 3 RelCondSetOp(${}_c\mathsf{T}_1,\ {}_c\mathsf{T}_2,\ c,\ SetOp$)

Input: ${}_c\mathsf{T}_1$, ${}_c\mathsf{T}_2$: relational conditional sets, c: predicate, $SetOp$: $\cap_c$ or $-_c$ or $\cup_c$
Output: the result ${}_c\mathsf{T}_R$ from ${}_c\mathsf{T}_1 \cap_c {}_c\mathsf{T}_2$ or from ${}_c\mathsf{T}_1 -_c {}_c\mathsf{T}_2$, according to $SetOp$

create an array of bits R_1 of size $|{}_c\mathsf{T}_1|$;
create an array of bits R_2 of size $|{}_c\mathsf{T}_2|$;
if $SetOp$ is $\cap_c$ **then**
    initialize R_1 with 0s;
else
    initialize R_1 with 1s;
end if;
initialize R_2 with 0s;
for each tuple $t_i \in {}_c\mathsf{T}_1$ **do**
    S = Index_TupleQuery(${}_c\mathsf{T}_2,\ t_i,\ c$);
    if $SetOp$ is $\cap_c$ and S has any bit 1 **then**
        set $R_1[i]$ as 1;
    else if $SetOp$ is $-_c$ and S has any bit 1 **then**
        set $R_1[i]$ as 0;
    else
        $R_2 = R_2 \vee S$
    end if;
end for;
${}_c\mathsf{T}_{R1}$ = get tuples from ${}_c\mathsf{T}_1$ that refer to each bit 1 in R_1;
${}_c\mathsf{T}_{R2}$ = get tuples from ${}_c\mathsf{T}_2$ that refer to each bit 1 in R_2;
return ${}_c\mathsf{T}_{R1} \cup {}_c\mathsf{T}_{R2}$;