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 ($-_c$). In this document, we will discuss 4 complementary points: (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, (3) the generality and usability of our operators, and, (4) the discussion about 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 ≤ SP.Units AND ¬ (DP.Price < SP.Price) OR DP.Price / 2 ≥ SP.Price)	 DP.Category SP.Category = DP.Product SP.Product = AND DP.Units SP.Units ≤ DP.Price SP.Price < ¬ AND DP.Price 2 / SP.Price ≥ 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 operations, as well as line 4, adds the closing bracket ")" to the end of our

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;
```

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*

```
1: Create a vector of bits R with size of |_cT_1|;
2: if SetOp is ∩_c then
3:     Initialize R with 0's;
4: else if SetOp is −_c then
5:     Initialize R with 1's;
6: end if
7: if _cT_2 is the table with index structures then
8:     for each tuple t_i of _cT_1 do
9:         if IsCondMember(_cT_2, t_i, c) then
10:            if SetOp is ∩_c then
11:                set R[i] as 1;
12:            else
13:                set R[i] as 0;
14:            end if
15:        end if
16:    end for
17: else if _cT_1 is the table with index structures then
18:    for each tuple t_j in _cT_2 do
19:        S = Index_TupleQuery(_cT_1, t_j, c);
20:        if SetOp is ∩_c then
21:            R = R ∨ S;
22:        else
23:            R = R ∧ ¬S;
24:        end if
25:    end for
26: end if
27: _cT_R = get tuples for all true bits of R;
28: return _cT_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 Generality and Usability

This section presents other applications where our operators are well suitable, thus corroborating their general usability.

3.1 Job Promotion

Let us consider a call for a job promotion supported by data from the Desired Skills (DS) for the job position and one candidate Employee's Skills (ES). Each skill can be quantified by a certification grade on a scale from 1 (beginner) to 5 (expert), with 0 for none; or, by the number of years of experience. Thus, we have relations with schemas $\text{Sch}(DS) = \text{Sch}(ES) = (Skill,\ Grade,\ Exp)$. The traditional set-based operators would be helpless here, as they would not allow to verify if the employee has the minimal certification grade or the minimal experience for each desired skill. However, this condition can be easily treated by our operators; we only have to consider a predicate $c : DS.Skill = ES.Skill \wedge (DS.Grade \leq ES.Grade \vee DS.Exp \leq ES.Exp)$. Now, one may design and execute queries like: a) "Does one skill $t \in ES$ satisfy any desired skill?" with $t \in_c DS$; b) "Does the employee satisfy all the desired skills?" with $DS \subseteq_c ES$; c) "Which desired skills does the employee satisfy?" with $DS \cap_c ES$, and; d) "Which are the desired skills that the employee does not satisfy?" with $DS -_c ES$.

3.2 Internship

For this case let us consider an organization that wants to recruit an intern. Naturally, the organization would like to analyze the best options among the

applicants, and the first step would be to inspect their grades in the Courses Attended (CA) in college. Here, the organization would list the Requested Courses (RC) with minimal grades, and contact an applicant for an interview only if he/she satisfies those requisites. Therefore, the schemas $\text{Sch}(RC) = \text{Sch}(CA) = (Course,\ Score)$. The traditional set-based operators would be helpless here, as they would not allow one to verify if an applicant has the minimal grades for the requested courses. Fortunatelly, our operators are promptly applicable with a predicate $c : RC.Course = CA.Course\ \wedge\ RC.Score \leq CA.Score$. Now, one may design and execute queries like: a) "Does a certain applicant's course $t \in CA$ satisfy any of the requested courses?" with $t \in_c RC$; b) "Does the applicant satisfy all the requested courses?" with $RC \subseteq_c CA$; c) "Which requested courses does the applicant satisfy?" with $RC \cap_c CA$, and; d) "Which are the requested courses that the applicant do not satisfy?" with $RC -_c CA$.

4 Discussion - Conditional Union

In this work, we have expanded different concepts from the Theory of Sets. However, we did not discuss a potential conditional union operation because we could not identify practical utility for it. For example, let us consider once again the motivational case study on sales of products presented in the paper. The union of relations DP and SP would return the products that are either desired by the client or available in the store. In the job promotion case from Section 3.1, the union of DS and ES would be the skills that are either desired by the employer or present in the employee. Also, in the internship example of Section 3.2, the union of RC and CA would be the courses that are either requested to recruit applicants or in the applicant's curriculum. In our humble opinion, none of these results seem to be meaningful for practical use. However, it would be straightforward to define a relational conditional union operator $\cup_c$, if it is required in any future work. The conditional union ${}_c\mathsf{T}_1 \cup_c {}_c\mathsf{T}_2$ of two relational conditional sets ${}_c\mathsf{T}_1$ and ${}_c\mathsf{T}_2$, with regard to a predicate c, would produce a new conditional set that is given by: ${}_c\mathsf{T}_R = {}_c\mathsf{T}_1\ \cup\ \{t_i\ :\ t_i \in {}_c\mathsf{T}_2\ \wedge\ t_i \notin_c {}_c\mathsf{T}_1\}$. As it happens with the traditional union, ${}_c\mathsf{T}_1$ and ${}_c\mathsf{T}_2$ must be union compatible. Also, it would be easy to adapt our algorithms to support the conditional union; this adaptation is described in Algorithm 3.

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*

1: create an array of bits R_1 of size $|{}_c\mathsf{T}_1|$;
2: create an array of bits R_2 of size $|{}_c\mathsf{T}_2|$;
3: **if** *SetOp* is $\cap_c$ **then**
4: initialize R_1 with 0s;
5: **else**
6: initialize R_1 with 1s;
7: **end if**;
8: initialize R_2 with 0s;
9: **for** each tuple $t_i \in {}_c\mathsf{T}_1$ **do**
10: S = Index_TupleQuery(${}_c\mathsf{T}_2$, t_i, c);
11: **if** *SetOp* is $\cap_c$ and S has any bit 1 **then**
12: set $R_1[i]$ as 1;
13: **else if** *SetOp* is $-_c$ and S has any bit 1 **then**
14: set $R_1[i]$ as 0;
15: **else**
16: $R_2 = R_2 \vee S$
17: **end if**;
18: **end for**;
19: ${}_c\mathsf{T}_{R1}$ = get tuples from ${}_c\mathsf{T}_1$ that refer to each bit 1 in R_1;
20: ${}_c\mathsf{T}_{R2}$ = get tuples from ${}_c\mathsf{T}_2$ that refer to each bit 1 in R_2;
21: **return** ${}_c\mathsf{T}_{R1} \cup {}_c\mathsf{T}_{R2}$;