Algorithms for Generating Tables

The algorithms in the section provide developers with a formal specification of translating VQFlow visualizations into SQL programs, i.e., these algorithms formally define the tables corresponding to each mini-query and requirements module (The entire query representation corresponds to a requirements module). Besides, the generated tables would ease the debugging of database queries by providing intermediate tables that correspond to the logical steps of constructing complex queries.

1) Generating Tables for Mini-queries: The pseudo-code of generating the table for a mini-query without the incoming edge is shown in Algorithm 1. In the algorithm, row.`Satisify(`*mq.filter*`)` returns true if row satisfies the condition in $mq.filter$. We call the output $eTable$ the exported table, which contains the rows in $mq.table$ that satisfy the condition in $mq.filter$, and these rows have all columns within $mq.table$. Through extracting the columns in $mq.columnsToDisplay$ from $eTable$, we can get the table to display (This also applies to requirements modules). The displayed table is part of the exported table, only containing the displayed columns $mq.columnsToDisplay$. With the displayed table, developers could easily understand the requirements corresponding to each component (i.e., mini-queries or requirements modules). The extra columns in the exported table are kept because these columns may be used in the edges within the tree starting from the current mini-query. When a mini-query is connected from another mini-query (or requirements module) by its incoming edge, the expanded table can be obtained by Algorithm 3. The key step in Algorithm 3 is the expansion `GetMatchedRows()`, i.e., retrieving the matched rows from the child table for each row in the parent table. The expansion step is demonstrated in Algorithm 2, where the code at Lines 3–12 checks whether a row $childRow$ in the child table matches the inputted row $parRow$ in the parent table. The row $childRow$ whose value at each column $childCol$ (from the pair $(parCol, childCol)$ in $columnsToMatch$) is equal to the corresponding value of $parRow$ is added to the set $matchedRows$. If no matched rows are found for the parent row $parRow$, one row with NULL values at all columns of the child table is added to $matchedRows$ (This strategy is equal to LEFT JOIN in SQL; we adopt this strategy because it is widely used in business reports). Notably, if $columnsToMatch$ is empty, all rows in the child table are added to $matchedRows$. In Algorithm 3, if the input $eInTable$ (i.e., the parent table) is empty, we return an empty set (Lines 1–3), which means that the parent table does not need to be expanded. We then generate the table $childRows$ for the mini-query mq through the `GenerateTableForMqWoE()` method (Line 5) (i.e., the mini-query is regarded as no incoming edges). $mq.incomingEdge$ serves as constraints on $childRows$: $mq.incomingEdge.columnsToMatch$ is used to retrieve the matched rows in $childRows$ for each row in the parent table (Lines 6–11). For each row $parRow$, each row in the matched rows $matchedRows$ is concatenated after $parRow$ (Lines 8–10). The filter in $mq.incomingEdge.filter$ is used to filter the resulting table $eOutTable$ (Lines 12–14). An edge has

---

Algorithm 1: GenerateTableForMqWoE(): Generating the table for a mini-query without the incoming edge

IN : A mini-query mq
OUT : An exported table $eTable$

$eTable \leftarrow \emptyset$
foreach row **in** $mq.table$ **do**
  if row.`Satisify(`*mq.filter*`)` **then**
    $eTable$.`Add(`*row*`)`
  end
end
return $eTable$

Algorithm 2: GetMatchedRows(): Get matched rows for a row in the parent table

IN : A row in the parent table $parRow$,
Filtered rows in the child table $childRows$,
Columns to match $columnsToMatch$
OUT : Matched rows $matchedRows$

1 $matchedRows \leftarrow \emptyset$
2 **foreach** $childRow$ **in** $childRows$ **do**
3   $isMatched \leftarrow true$
4   **foreach** $(parCol, childCol)$ **in** $columnsToMatch$ **do**
5   **if** $parRow.parCol \neq childRow.childCol$ **then**
6       $isMatched \leftarrow false$
7   **Break**
8   **end**
9   **end**
10   **if** $isMatched$ **then**
11     $matchedRows$.`Add(`*childRow*`)`
12   **end**
13 **end**
14 **if** $matchedRows = \emptyset$ **then**
15   $matchedRows$.`Add(`*NULLs*`)`
16 **end**
17 **return** $matchedRows$

relations with only the two exported tables corresponding to the two components connected by the edge. Thus, when developers try to understand the requirements corresponding to the edge, they can focus on only the edge and the connected two components.

Algorithm 3: GenerateTableForMqWE(): Generating the table for a mini-query with the incoming edge

```
   IN   : An exported table eInTable,
          A mini-query mq
   OUT  : An exported table eOutTable
1  if eInTable is empty then
2  |  return ∅
3  end
4  eOutTable ← ∅
5  childRows ←GenerateTableForMqWoE(mq)
6  foreach parRow in eInTable do
7  |  matchedRows ←GetMatchedRows(parRow,
   |    childRows, mq.incomingEdge.columnsToMatch)
8  |  foreach matchedRow in matchedRows do
9  |  |  eOutTable.Add(parRow + matchedRow)
10 |  end
11 end
12 if mq.incomingEdge.ContainFilter() then
13 |  eOutTable ←
   |    eOutTable.GetSatisfiedRows(
   |    mq.incomingEdge.filter)
14 end
15 return eOutTable
```

2) Generating Tables for Requirement Modules: Mini-queries are the constituent elements of requirements modules. After introducing the inputs and outputs of mini-queries, we now turn to the generation of tables corresponding to requirements modules. In a requirements module, the internal mini-queries are connected by edges, constituting a tree, and each internal mini-query can be replaced by a requirements module because of the same inputs and outputs. A requirements module has the input of 0 or 1 incoming edge (which is the incoming edge of the root component within the requirements module) and the output of a table; thus, the inputs and outputs of requirements modules are the same as mini-queries. Same with mini-queries, for generating the tables corresponding to requirements modules, there are also two cases: without the incoming edge (Algorithm 4) and with the incoming edge (Algorithm 5). In Algorithm 4, we first generate the $outRows$ table for the root component (Lines 2–7). Then we traverse the component tree in the requirements module in the way of depth-first search $DFS()$, and remove the first component in the resulting list (Line 8). Finally, for each component in the component list $compList$, we generate its table (Lines 10–14). Notably, the table $outRows$ for a component $comp$ (corresponding to the first input $eInTable$ of the `GenerateTableForMqWE()` method at Line 11 or `GenerateTableForRMWE()` at Line 13) are usually not the exported table of the source component that is directly connected to the component $comp$.

Algorithm 4: GenerateTableForRMWoE(): Generating the table for a requirements module without the incoming edge

```
   IN   : A requirements module rm
   OUT  : An exported table eOutTable
1  eOutTable ← ∅
2  outRows ← ∅
3  if rm.root is a mini-query then
4  |  outRows ←GenerateTableForMqWoE(rm.root)
5  else if comp is a requirements module then
6  |  outRows ←GenerateTableForRMWoE(rm.root)
7  end
8  compList ← DFS (rm.tree) −rm.root
9  foreach comp in compList do
10 |  if comp is a mini-query then
11 |  |  outRows ←GenerateTableForMqWE(outRows,
   |  |    comp)
12 |  else if comp is a requirements module then
13 |  |  outRows ←GenerateTableForRMWE(outRows,
   |  |    comp)
14 |  end
15 end
16 eOutTable.Add(outRows)
17 return eOutTable
```

Algorithm 5: GenerateTableForRMWE(): Generating the table for a requirements module with the incoming edge

```
   IN   : An exported table eInTable,
          A requirements module rm
   OUT  : An exported table eOutTable
1  if eInTable is empty then
2  |  return ∅
3  end
4  eOutTable ← ∅
5  compList ←DFS (rm.tree)
6  foreach rootRow in eInTable do
7  |  outRows ← [rootRow]
8  |  foreach comp in compList do
9  |  |  if comp is a mini-query then
10 |  |  |  outRows ←GenerateTableForMqWE(
   |  |  |    outRows, comp)
11 |  |  else if comp is a requirements module then
12 |  |  |  outRows ←GenerateTableForRMWE(
   |  |  |    outRows, comp)
13 |  |  end
14 |  end
15 |  eOutTable.Add(outRows)
16 end
17 return eOutTable
```

For a requirements module with the incoming edge, compared with a requirements module without the incoming edge, no special consideration is required for its root component. All we need to do is expand each row in the input $eInTable$ according to the component tree within the requirements module. The expansion way is the same as that for requirements modules without the incoming edge.