Datalog-SQL Translator

Tiezheng Song

November 18, 2018

1 Introduction

This report introduces a Python program to translate Datalog to SQL and vice versa under certain constraints.

1.1 Constraints in Datalog query

1. The datalog query is non-recursive;
2. No negation are involved;
3. Head and body are separated by ':-';
4. Body literals are separated by ', ';
5. Each terms in predicate are separated by ',';

1.2 Constraints in SQL query

1. All SQL queries are based two part naming convetion, i.e. *tableName.columnName*;
2. Join are expressed implicitly;
3. No set operators are involved, i.e. *union, intersect, etc*

2 Datalog To SQL

To translate Datalog query to SQL query, two inputs are required for main function and eleven subfunctions are included. The workflow of main function is listed below. The idea is straightforward, especially for the generation of 'select' statement and 'from' statement, which is to split each part of the datalog query and convert each one based on the logic relation of datalog and sql. To convert the 'where' statement, two dictionaries are created in order to deal with the situation when pre-built predicates or constants are involved in body literals. In those scenarios, we need to match the constants appear in literal with the variables in that particular predicate, then, we can generate the conditions for where clause. For example, if we have 'fact1(a,10)', we should figure out which term of fact1 equals to 10. The sudo-code of 'where' statement generation is also provided.

2.1 Logic of Main Function

a. Input one (string): datalog query

b. Input two (string): schema of each predicate

c. Return (string): sql query

algorithm 1 Main Function for Datalog-SQL translator

1: Generate a schema dictionary for predicates in body via function 7,
2: Convert entry datalog queries to a list via function 1,
3: Get the head via function 2,
4: Extract each terms in head via function 4
5: Get the body via function 2
6: Generate a dictionary for each body literals via function 8,
7: Generate 'Select' clause via function 9
8: Generate 'From' clause via function 10
9: Generate 'Where' clause via function 11
10: Add each 'Select - From - Where' clause to generate the final result

2.2 Sub - Functions

Function 1:

```
def getClauses(datalog_query):
    '''
    Description: Seperate each clause by character '.'
    :type datalog_query: string
    :rtype list
    '''
    return [item for item in datalog_query.split('.') if item != '']
```

Function 2:

```
def seperateHeadBody(clause):
    '''
    Description: get head and body for each clause
    :type clauses: string
    :rtype list
    '''
    head = clause.split(':-')[0]
    body = clause.split(':-')[1]
    return head,body
```

Function 3:

```
def getTermsInParens(a):
    '''
    Description: get each term in parentheses
    :type a: string
    :rtype list
    '''
    results = []
    if a.find('(') != -1 and a.find(')') != -1:
        terms = a[a.find("(")+1:a.find(")")].split(',')
        for term in terms:
            results.append(term)
    return results
```

Function 4:

```
def getColsFromHead(head):
    '''
    Description: extract each term in head
    :type head: string
```

```
    :rtype list
    '''
    cols = getTermsInParens(head)
    return cols
```

Function 5:

```
def getPredPerTerm(term):
    '''
    Description: extract predicate symbol in body
    :type term: string
    :rtype: string
    '''
    predicate = ''
    if term.find('(') != -1 and term.find(')') != -1:
        predicate = term.split('(')[0]
    return predicate
```

Function 6:

```
def getTabsPerBody(body):
    '''
    Description: extract terms in each body literals
    :type body: string
    :rtype list
    '''
    tabs = []
    bodyLiterals = body.split(', ')
    for bodyLiteral in bodyLiterals:
        if bodyLiteral.find('(') != -1 and bodyLiteral.find(')') != -1:
            predicate = bodyLiteral.split('(')[0]
            tabs.append(predicate)
    return tabs
```

Function 7:

```
def getSchemaDict(inputSchemas):
    '''
    Description: Create a dictionary for each literal
    in terms of {key: predicate symbol, value: list of each term}
    :type inputSchemas: list
    :rtype dictionary
    '''
    schemaDict = {}
    for schema in inputSchemas:
        pred = getPredPerTerm(schema)
        if pred in schemaDict:
            schemaDict[pred] += getTermsInParens(schema)
        else:
            schemaDict[pred] = getTermsInParens(schema)
    return schemaDict
```

Function 8:

```
def getBodyDict(body):
    '''
    Description: create a dictionary to contain predicate-term pair
    for each body term
```

```
    type string: body
    rtype dict: bodyDict
    '''
    bodyDict = {}
    bodyLiterals = body.split(', ')
    for bodyLiteral in bodyLiterals:
        if bodyLiteral.find('(') != -1 and bodyLiteral.find(')') != -1:
            predicate = bodyLiteral.split('(')[0]
            bodyDict[predicate] =
            bodyLiteral[bodyLiteral.find('(')+1:bodyLiteral.find(')')].split(',')
        else:
            bodyDict[bodyLiteral] = bodyLiteral
    return bodyDict
```

Function 9:

```
def getSelectQuery(cols, bodyDict):
    '''
    Description: generate select clause
    :type cols:  list[string]
    :type bodyDict: dict
    rtype string: selectCaluse
    '''
    selectClause = 'select '
    for col in cols:
        for pred in bodyDict.keys():
            if col in bodyDict[pred]:
                selectClause += pred + '.' + col + ', '
                break
    selectClause = selectClause.strip(', ')
    return selectClause
```

Function 10:

```
def getFromQuery(body):
    '''
    Descrption: generate from clause
    :type body: string
    :rtype string
    '''
    fromClause = 'from '
    for tab in getTabsPerBody(body):
        fromClause += tab + ', '
    fromClause = fromClause.strip(', ')
    return fromClause
```

Function 11:

```
def getWhereQuery(bodyDict, schemaDict):
    '''
    Description: generate where clause
    type dict: bodyDict
    type dict: schemaDict
    rtype string: whereClause
    '''
    split_comp = ">|<|="
    dictCol = {}
```

```
    whereClause = 'where '
    for pred in bodyDict.keys():
        if len(re.findall(split_comp, pred))==0:
            for term in bodyDict[pred]:
                if term not in dictCol and term in schemaDict[pred]:
                    dictCol[term] = pred
                elif term in dictCol and term in schemaDict[pred]:
                    whereClause +=
                    dictCol[term] + '.' + term + '=' + pred + '.' + term + ' and '
                # specify constant
                elif term not in schemaDict[pred]:
                    pos = bodyDict[pred].index(term)
                    whereClause +=
                    pred + '.'+ schemaDict[pred][pos] + '=' + term + ' and '
        # translate built-in predicate
        else:
            term = re.split(split_comp,pred)[0]
            term = term.strip(' ')
            whereClause += dictCol[term] + '.' + pred + ' and '
    whereClause = whereClause.strip(' and ')
    return whereClause
```

2.3 SUDO-CODE OF GETWHEREQUERY

To generate each conditions in where statement, each literal in body should be separated and identified based on the predicate type. For those contain symbols such as $>, \leq, <, \geq, =$, will be treated as built-predicate. For those literals which contain constant in parentheses, we need to figure out corresponds variables with the help of the dictionaries defined early. For the general literals such as, fact1(x,y), fact2(x,z), which define the join of two 'tables', which are fact1 and fact2. In implicit join expression, we can translate as 'From fact1, fact2 Where fact1.x=fact2.x'. To get these pair of attributes, I define another dictionary, so the same variable occurs in a literal with different predicate symbol, the join condition can be determined.
In summary, in this mini project, only three conditions will occur in 'where' statement.
First, join condition, e.g. tab1.col1 = tab2.col1
Second, the condition translated by built-in predicate, e.g. predicate.X < 100
Thirld, the equality conditions, e,g. for predicate(X,Y) if we have body literal as predicate(100,Y), we will get predicate.X=100

Input 1: bodyDict, {key: predicate symbol, value: list of terms}
Input 2: schemaDict, {key: predicate symbol, value: list of variables}

2.4 EXAMPLES

Exampl 1
input 1: datalog = 'r1(a,b):-fact1(x,a,20), fact2(a,y), fact3(y,b), x >= 10.'
input 2: inputSchemas = 'fact1(x,a,c), fact2(a,y), fact3(y,b)'
result: select fact1.a, fact3.b from fact1, fact2, fact3 where fact1.c = 20 and fact1.a=fact2.a and fact2.y=fact3.y and fact1.x >= 10
Exampl 2
input 1: datalog = 'r1(a,b,c):-fact(x,a,c), fact2(a,y), fact3(y,b).'
input 2: inputSchemas = 'fact1(x,a,c), fact2(a,y), fact3(y,b)'
result: create view r1 as (select fact1.a, fact3.b, fact1.c from fact1, fact2, fact3 where fact1.a=fact2.a and fact2.y=fact3.y)

algorithm 2 Sudo-Code of getWhereQuery

```
dictCol = {} # to determine the equality condition in where clause
whereStatement = ' where '
for literal in body do
    if literal contains no Comparision Symbols then
        for term in literal do
            if term not in dictCol and term is variable then dictCol ← term-predicate pair
            end if
            if term in dictCol and term is variable then whereStatement += join condition
            end if
            if term is constant then  whereStatement += equality condition
            end if
        end for
    else whereStatement += built-in condition
    end if
end for
```

3 SQL TO DATALOG

To translate SQL query to Datalog query, two inputs are required for main function, and five functions are required. The head literal can be determined from 'select' statement. And by analyzing 'from' and 'where' statements, the body of the Datalog query can be determined.

3.1 LOGIC OF MAIN FUNCTION

Scan the input SQL query, separate 'select-from-where' syntax then convert each part.
a. Input one (string): Sql query

b. Input two (string): schema of each predicate

c. Return (string): sql query

Main Code

```
def mainCode(sqlQuery, inputSchemas):
    headLiteral, sqlFrom, sqlWhere = splitSQL(sqlQuery)
    bodyLiteral = getBody(inputSchemas, sqlFrom, sqlWhere)
    print("DataLog Query: ")
    print(headLiteral + ":- " + bodyLiteral)
```

3.2 SUB - FUNCTIONS

Function 1

```
def getHeadTerms(sqlSelect):
    '''
    Descritption: extract head terms from select clauses
    :type string: sqlSelect
    :rtype string: headTerms
    '''
    results = []
    headTerms = '('
    for term in sqlSelect.split(','):
        results.append(term.split('.')[1])
    for result in results:
```

```
        headTerms += result + ','
    headTerms = headTerms.strip(',')
    headTerms += ")"
    return headTerms
```

Function 2

```
def splitSQL(sqlQuery):
    '''
    Description: return head literal and split sql query
    type string: sqlQuery
    rtype list
    '''
    sqlQuery = sqlQuery.lower()
    split_sql = "select|from|where"
    head = sqlQuery.split('as ')[0]
    sqlWhere = ''
    if sqlQuery.find('where') != -1:
        [sqlSelect, sqlFrom, sqlWhere] = [item.strip(' ') for item in re.split(split_sql, sqlQ
    else:
        [sqlSelect, sqlFrom] = [item.strip(' ') for item in re.split(split_sql, sqlQuery.split
    headRule = head[head.find('create view')+ 11:sqlQuery.find('as')].strip(' ')
    headLiteral = headRule + getHeadTerms(sqlSelect)
    return headLiteral, sqlFrom, sqlWhere
```

Function 3

```
def splitSQL(sqlQuery):
    '''
    Description: generate a dictionary for body literals
    :type string: inputSchemas
    :rtype schemaDict
    '''
    schemaDict = {}
    for schema in inputSchemas:
        schemaDict[schema.split("(")[0]] =  schema
    return schemaDict
```

Function 4

```
def list2str(a_list):
    return ', '.join(list(map(str, a_list)))
```

Function 5

```
def getBody(inputSchemas, sqlFrom,sqlWhere):
    '''
    Description: Generate body literals based on from statement and where statement
    :type string: inputSchemas,
    :type string: sqlFrom,
    :type string: sqlWhere
    :rtype string
    '''
    split_logic = " and | or "
    split_comp = ">|<"
    split_comp1 = ">=|<="
    split_comp2 = ">=|<=|>|<"
    schemaDict = getTableSchema(inputSchemas)
```

```
    body = []
    if sqlWhere != '':
        for whereCondition in re.split(split_logic, sqlWhere): # or
            if whereCondition.find("=") != -1 and len(re.findall(split_comp1,whereCondition))
                # translate join condition
                condLHS = whereCondition.split("=")[0]
                condRHS = whereCondition.split("=")[1]
                if condLHS.find(".") != -1 and condRHS.find(".") != -1:
                    tabl = condLHS.split('.')[0]
                    if schemaDict[tabl] not in body:
                        body.append(schemaDict[tabl])
                    tab2 = condRHS.split('.')[0]
                    if schemaDict[tab2] not in body:
                        body.append(schemaDict[tab2])
                # translate '=' condition
                elif condLHS.find(".") > -1 and condRHS.find(".") == -1:
                    tab = condLHS.split('.')[0]
                    col = condLHS.split('.')[1]
                    body.append(col + "=" + re.split(split_comp, condRHS)[0])
            # translate built-in condition
            elif len(re.findall(split_comp2,whereCondition)) > 0:
                condLHS = re.split(split_comp,whereCondition)[0]
                condRHS = re.split(split_comp,whereCondition)[1]
                if condLHS.find(".") > -1 and condRHS.find(".") > -1:
                    term0 = condLHS.split('.')[1]
                    term1 = condRHS.split('.')[1]
                    temp = whereCondition.replace(condLHS, term0)
                    temp = temp.replace(condRHS, term1)
                    body.append(temp)
                elif condLHS.find(".") > -1 and condRHS.find(".") == -1:
                    term0 = condLHS.split('.')[1]
                    temp = whereCondition.replace(condLHS, term0)
                    body.append(temp)
    else:
        body.append(schemaDict[sqlFrom])
    body = sorted(body)
    bodyLiterals = list2str(body)
    return bodyLiterals
```

3.3 SUDO-CODE OF GETBODY FUNCTION

The main idea of convert SQL to Datalog is similar to the one of converting Datalog to SQL. In this project, three types of conditions are considered, which are the equality condition e.g. fact1.a = 100, join condition e.g. fact1.x=fact2.x and comparison condition e.g. fact1.x>=100. The sudo code is listed below.

3.4 EXAMPLES

Exampl 1

input 1: sqlQuery = 'Create View r1 as(select fact1.a, fact3.b from fact1, fact2, fact3 where fact2.a=fact1.a and fact2.y=fact3.y and fact2.a>=10)'
input 2: inputSchemas = 'fact1(x,a), fact2(a,y), fact3(y,b)'
result: r1(a,b):- a>=10, fact1(x,a), fact2(a,y), fact3(y,b)

algorithm 3 Sudo-Code of getBody Function

```
1: body = []
2: if where statement is not Null then
3:     for condition in sqlWhere do
4:         if condition contains '=' and condition not contains ' >=',' <=',' >',' <' then
5:             if condition doesn't contain constant then bodyLiteral ← translate the join condition
6:             end if
7:             if condition contains constant then bodyLiteral ← translate the equality condition
8:             end if
9:         end if
10:        if condition not contains ' >=',' <=',' >',' <' then bodyLiteral ← get the built-in predicate
11:        end if
12:        body.append{bodyLiteral}
13:    end for
14: else
15: bodyLiteral ← translate the 'from' statement
16: body.append{bodyLiteral}
17: end if
```

Exampl 2
input 1: sqlQuery = 'Create View r1 as(select fact1.a, fact1.x from fact1)'
input 2: inputSchemas = 'fact1(x,a), fact2(a,y), fact3(y,b)'
result: r1(a,x):- fact1(a,x)