gideonw14 / sql-compiler

Branch: optimizer_2 sql-compiler / compiler.py Find file Copy path

Gideon Walker added a little comment to the top e706a57 a minute ago

0 contributors

1112 lines (972 sloc) 37.1 KB

```python
"""
Programmer: Gideon Walker and Ryan Leas
Date: 11/30/17
Class: CS 5300 Databases
Assignment: Part 2 of the SQL Compiler Project
"""

from copy import deepcopy

###############################################################################
#                                                                             #
#  LEXER                                                                      #
#                                                                             #
###############################################################################

# Token types

INTEGER         = 'INTEGER'
STRING          = 'STRING'
PLUS            = 'PLUS'
MINUS           = 'MINUS'
MUL             = 'MUL'
LPAREN          = 'LPAREN'
RPAREN          = 'RPAREN'
ID              = 'ID'
ASSIGN          = 'ASSIGN'
SEMI            = 'SEMI'
DOT             = 'DOT'
COLON           = 'COLON'
COMMA           = 'COMMA'
EOF             = 'EOF'
KEYWORD         = 'KEYWORD'
SELECT          = 'SELECT'
FROM            = 'FROM'
WHERE           = 'WHERE'
AS              = 'AS'
IN              = 'IN'
CONTAINS        = 'CONTAINS'
INTERSECT       = 'INTERSECT'
UNION           = 'UNION'
EXCEPT          = 'EXCEPT'
HAVING          = 'HAVING'
GROUP           = 'GROUP'
BY              = 'BY'
AND             = 'AND'
OR              = 'OR'
EQUAL           = 'EQUAL'
GREATER         = 'GREATER'
LESSER          = 'LESSER'
GREATEREQUAL    = 'GREATEREQUAL'
LESSEREQUAL     = 'LESSEREQUAL'
MIN             = 'MIN'
MAX             = 'MAX'
SUM             = 'SUM'
```

```python
COUNT           = 'COUNT'
AVG             = 'AVG'
_NOT            = 'NOT'
EXISTS          = 'EXISTS'

SPACES = 8
RELATIONS = ('SAILORS', 'BOATS', 'RESERVES')
ATTRIBUTES = {RELATIONS[0]: ('SID', 'SNAME', 'RATING', 'AGE'),
              RELATIONS[1]: ('BID', 'BNAME', 'COLOR'),
              RELATIONS[2]: ('SID', 'BID', 'DAY')}

# Helper Function
def flatten(S):
    if S == []:
        return S
    if isinstance(S[0], list):
        return flatten(S[0]) + flatten(S[1:])
    return S[:1] + flatten(S[1:])

class Tree_Node(object):
    def __init__(self, left=None, right=None, value=None):
        self.left = left
        self.right = right
        self.value = value

    def __str__(self):
        return '{} : {} : {}'.format(self.left, self.value, self.right)

    def __repr__(self):
        return self.__str__()

class Token(object):
    def __init__(self, type, value):
        self.type = type
        self.value = value

    def __str__(self):
        """String representation of the class instance.

        Examples:
            Token(INTEGER, 3)
            Token(PLUS, '+')
            Token(MUL, '*')
        """
        return 'Token({type}, {value})'.format(
            type=self.type,
            value=repr(self.value)
        )

    def __repr__(self):
        return self.__str__()


RESERVED_KEYWORDS = {
    'SELECT': Token('SELECT', 'SELECT'),
    'FROM': Token('FROM', 'FROM'),
    'WHERE': Token('WHERE', 'WHERE'),
    'AS': Token('AS', 'AS'),
    'AND': Token('AND', 'AND'),
    'OR': Token('OR', 'OR'),
    'IN': Token('IN', 'IN'),
    'CONTAINS': Token('CONTAINS', 'CONTAINS'),
    'INTERSECT': Token('INTERSECT', 'INTERSECT'),
    'UNION': Token('UNION', 'UNION'),
    'EXCEPT': Token('EXCEPT', 'EXCEPT'),
    'HAVING': Token('HAVING', 'HAVING'),
    'GROUP': Token('GROUP', 'GROUP'),
    'BY': Token('BY', 'BY'),
    'MIN': Token('MIN', 'MIN'),
    'MAX': Token('MAX', 'MAX'),
    'COUNT': Token('COUNT', 'COUNT'),
    'SUM': Token('SUM', 'SUM'),
    'AVG': Token('AVG', 'AVG'),
    'NOT': Token('NOT', 'NOT'),
    'EXISTS': Token('EXISTS', 'EXISTS'),

}


class Lexer(object):
    def __init__(self, text):
        # client string input, e.g. "4 + 2 * 3 - 6 / 2"
        self.text = text
        # self.pos is an index into self.text
        self.pos = 0
        self.current_char = self.text[self.pos]

    def error(self):
        raise Exception('Invalid character near or at "{}"'.format(self.current_char))

    def advance(self):
        """Advance the `pos` pointer and set the `current_char` variable."""
        self.pos += 1
        if self.pos > len(self.text) - 1:
            self.current_char = None  # Indicates end of input
        else:
            self.current_char = self.text[self.pos]

    def peek(self):
        peek_pos = self.pos + 1
        if peek_pos > len(self.text) - 1:
            return None
        else:
            return self.text[peek_pos]

    def skip_whitespace(self):
        while self.current_char is not None and self.current_char.isspace():
            self.advance()

    def integer(self):
        """Return a (multidigit) integer consumed from the input."""
        result = ''
        while self.current_char is not None and self.current_char.isdigit():
            result += self.current_char
            self.advance()
        return int(result)

    def string(self):
        """ Return a string consumed from the input """
        result = ''
        if self.current_char == "'":
            self.advance()
        while self.current_char != "'":
            result += str(self.current_char)
            self.advance()
        if self.current_char == "'":
            self.advance()
        else:
            self.error()
        return result

    def _id(self):
        """Handle identifiers and reserved keywords"""
        result = ''
        while self.current_char is not None and self.current_char.isalnum():
            result += self.current_char
            self.advance()

        token = RESERVED_KEYWORDS.get(result, Token(ID, result)) # Gets the keyword or returns identifier token
        return token

    def get_next_token(self):
        """Lexical analyzer (also known as scanner or tokenizer)

        This method is responsible for breaking a sentence
        apart into tokens. One token at a time.
        """
        while self.current_char is not None:

            if self.current_char.isspace():
                self.skip_whitespace()
                continue

            if self.current_char.isalpha():
                return self._id()

            if self.current_char.isdigit():
                return Token(INTEGER, self.integer())

            if self.current_char == "'":
                return Token(STRING, self.string())

            if self.current_char == ';':
                self.advance()
                return Token(SEMI, ';')

            if self.current_char == '*':
                self.advance()
                return Token(MUL, '*')

            if self.current_char == '(':
                self.advance()
                return Token(LPAREN, '(')

            if self.current_char == ')':
                self.advance()
                return Token(RPAREN, ')')

            if self.current_char == '.':
                self.advance()
                return Token(DOT, '.')

            if self.current_char == '=':
                self.advance()
                return Token(EQUAL, '=')

            if self.current_char == '>' and self.peek() == '=':
                self.advance()
                self.advance()
                return Token(GREATEREQUAL, '>=')

            if self.current_char == '<' and self.peek() == '=':
                self.advance()
                self.advance()
                return Token(LESSEREQUAL, '<=')

            if self.current_char == '>':
                self.advance()
                return Token(GREATER, '>')

            if self.current_char == '<':
                self.advance()
                return Token(LESSER, '<')

            if self.current_char == ',':
                self.advance()
                return Token(COMMA, ',')

            self.error()

        return Token(EOF, None)


###############################################################################
#                                                                             #
#  PARSER                                                                     #
#                                                                             #
###############################################################################

class AST(object):
    pass

class Rel_Alg_Select(AST):
    def __init__(self, left, op, right, next=None):
        self.left = left
        self.token = self.op = op
        self.right = right
        self.next = next

    def __eq__(self, other):
        if isinstance(other, Rel_Alg_Select):
            if self.__str__() == other.__str__():
                return True
        return False

    def __str__(self):
        result = '{} {} {}'.format(self.left.__str__(), self.op, self.right.__str__())
        if self.next:
            result += ' {}'.format(self.next)
        return result

    def str_no_next(self):
        return '{} {} {}'.format(self.left.__str__(), self.op, self.right.__str__())

    def __repr__(self):
        return self.__str__()

class Attr(AST):
    def __init__(self, attribute, relation=None):
        self.attribute = attribute.value
        if relation:
            self.relation = relation.value
        else:
            self.relation = None

    def __str__(self):
        result = self.attribute
        if self.relation:
            result = '{}.{}'.format(self.relation, result)
        return result

    def __repr__(self):
        return self.__str__()


class Ag_Function(AST):
    def __init__(self, function, attribute, alias=None):
        self.function = function
        self.attribute = attribute
        self.alias = alias

    def __str__(self):
        result = '{}({})'.format(self.function, self.attribute)
        if self.alias:
            result += ' AS {}'.format(self.alias)
        return result

    def __repr__(self):
        return self.__str__()


class Rel(AST):
    def __init__(self, relation, alias=None):
        self.relation = relation.value
        if alias:
            self.alias = alias.value
        else:
            self.alias = None

    def __eq__(self, other):
        if isinstance(other, str):
            if self.relation.__str__() == other:
                return True
        else:
            if self.relation == other.relation:
                if not self.alias and not other.alias:
                    return True
                if self.alias and other.alias:
                    if self.alias == other.alias:
                        return True
        return False

    def same_relation(self, other):
        if other == self.relation or other == self.alias:
            return True
        else:
            return False

    def __str__(self):
        result = self.relation
        if self.alias:
            result = '{} AS {}'.format(result, self.alias)
        return result

    def __repr__(self):
        return self.__str__()

class Query(AST):
    def __init__(self, projects, relations, selects=None, groupby=None, having=None, nested=None):
        self.selects = selects
        self.projects = projects
        self.relations = relations
        self.groupby = groupby
        self.having = having
        self.nested = nested

class Nest_Query(AST):
    def __init__(self, attribute, op, query):
        self.attribute = attribute
        self.op = op
        self.query = query

class Set_Op(AST):
    def __init__(self, left=None, right=None, op=None):
        self.left = left
```

```python
        self.right = right
        self.op = op

# class In(AST):
#     def __init__(self, attribute, select):
#         pass

class Parser(object):
    def __init__(self, lexer):
        self.lexer = lexer
        # set current token to the first token taken from the input
        self.current_token = self.lexer.get_next_token()
        # previous token used to make error message more helpful
        self.prev_token = None

    def error(self):
        from colorama import init, Fore
        init(autoreset=True)
        raise Exception(Fore.RED + 'Invalid syntax near or at "{} {}"'.format(self.prev_token.value, self.current_token.value))

    def eat(self, token_type):
        # compare the current token type with the passed token
        # type and if they match then "eat" the current token
        # and assign the next token to the self.current_token,
        # otherwise raise an exception.
        if self.current_token.type == token_type:
            # print(self.current_token)
            self.prev_token = self.current_token
            self.current_token = self.lexer.get_next_token()
        else:
            self.error()

    def query(self):
        # query: compound statement
        #      | (? compound statement )?
        if self.current_token.type == LPAREN:
            self.eat(LPAREN)
        node = self.sql_compound_statement()
        if self.current_token.type == RPAREN:
            self.eat(RPAREN)
        # self.eat(SEMI)
        return node

    def sql_compound_statement(self):
        """
        note: ? means 0 or 1 instances
        sql_compound_statement: SELECT attribute_list
                                FROM relation_list
                                (WHERE condition_list)?
                                (GROUP BY attribute_list)?
                                (HAVING condition_list)?
                                (INTERSECT | UNION | EXCEPT | CONTAINS sql_compound_statement)?
        """
        cond_nodes = list()
        group_by_list = list()
        having_list = list()
        compound_statement = None
        set_op = ''
        self.eat(SELECT)
        attr_nodes = self.attribute_list()
        self.eat(FROM)
        rel_nodes = self.relation_list()
        if self.current_token.type == WHERE:
            self.eat(WHERE)
            cond_nodes = self.condition_list()
        if self.current_token.type == GROUP:
            self.eat(GROUP)
```

```
                self.eat(BY)
                group_by_list = self.attribute_list()
            if self.current_token.type == HAVING:
                self.eat(HAVING)
                having_list = self.condition_list()
            if self.current_token.type == RPAREN:
                self.eat(RPAREN)
            if self.current_token.type in (INTERSECT, UNION, EXCEPT, CONTAINS):
                set_op = self.current_token.type
                if self.current_token.type == INTERSECT:
                    self.eat(INTERSECT)
                elif self.current_token.type == UNION:
                    self.eat(UNION)
                elif self.current_token.type == EXCEPT:
                    self.eat(EXCEPT)
                elif self.current_token.type == CONTAINS:
                    self.eat(CONTAINS)
                compound_statement = self.query()
            query = Query(attr_nodes, rel_nodes, cond_nodes, group_by_list, having_list)
            if compound_statement:
                combined = Set_Op(query, compound_statement, set_op)
                if query.selects:
                    if combined.op == UNION:
                        query.selects[-1].next = OR
                    elif combined.op == INTERSECT or combined.op == CONTAINS:
                        query.selects[-1].next = AND
                    elif combined.op == EXCEPT:
                        query.selects[-1].next = 'AND NOT'

                if query.relations == compound_statement.relations:
                    for query_condition in compound_statement.selects:
                        if query_condition in query.selects:
                            continue
                        else:
                            query.selects.append(query_condition)
                else:
                    for relation in compound_statement.relations:
                        query.relations.append(relation)
                    for condition in compound_statement.selects:
                        query.selects.append(condition)

            return query

        def attribute_list(self):
            """
            attribute_list : (attribute | ag_function) (COMMA attribute_list)*
            """
            if self.current_token.type == ID:
                node = self.attribute()
            else:
                node = self.ag_function()
            results = [node]
            while self.current_token.type == COMMA:
                self.eat(COMMA)
                if self.current_token.type == ID:
                    next = self.attribute()
                else:
                    next = self.ag_function()
                results.append(next)
            return results

        def ag_function(self):
            """ag_function: (MIN | MAX | SUM | COUNT | AVG) (attribute) (AS alias):"""
            function = self.current_token.value
            if self.current_token.type == MAX:
                self.eat(MAX)
            elif self.current_token.type == MIN:
                self.eat(MIN)
            elif self.current_token.type == SUM:
                self.eat(SUM)
            elif self.current_token.type == COUNT:
                self.eat(COUNT)
            elif self.current_token.type == AVG:
                self.eat(AVG)
            else:
                self.error()

            self.eat(LPAREN)
            attribute = self.attribute()
            self.eat(RPAREN)

            if self.current_token.type == AS:
                self.eat(AS)
                alias = self.current_token.value
                self.eat(ID)
                return Ag_Function(function, attribute, alias)

            return Ag_Function(function, attribute)

        def attribute(self):
            """
            attribute : identifier
                      | identifier DOT identifier
                      | STAR aka MUL

            """
            node = Attr(self.current_token)
            if self.current_token.type == MUL:
                self.eat(MUL)
            else:
                self.eat(ID)
                if self.current_token.type == DOT:
                    self.eat(DOT)
                    node.relation = node.attribute
                    node.attribute = self.current_token.value
                    self.eat(ID)
            return node

        def relation_list(self):
            """
            relation_list : relation
                          | relation COMMA relation_list
            """
            node = self.relation()
            results = [node]
            while self.current_token.type == COMMA:
                self.eat(COMMA)
                results.append(self.relation())
            return results

        def relation(self):
            """
            relation : identifier
                     | identifier (AS)? identifier
            """
            node = Rel(self.current_token)
            self.eat(ID)
            if self.current_token.type == AS:
                self.eat(AS)
            if self.current_token.type == ID:
                node.alias = self.current_token.value
                self.eat(ID)
            return node

```

```
    def condition_list(self):
        """
        condition_list : condition
                       | condition (AND | OR) condition_list
        """
        node = self.condition()
        results = [node]
        while self.current_token.type in (AND, OR):
            results[-1].next = self.current_token.value
            if self.current_token.type == AND:
                self.eat(AND)
            else:
                self.eat(OR)
            results.append(self.condition())
        return results

    def condition(self):
        """
        condition : attribute (EQUAL | GREATER | LESSER | GREATEREQUAL | LESSEREQUAL) (attribute | INTEGER | STRING)
                  | attribute (IN | NOT EXISTS) LPAREN sql_compound_statement RPAREN
        """
        # Left is always attribute
        if self.current_token.type in (SUM, COUNT, MAX, MIN, AVG):
            left = self.ag_function()
        elif self.current_token.type == _NOT:
            token = 'AND NOT'
            self.eat(_NOT)
            self.eat(EXISTS)
            self.eat(LPAREN)
            node = self.query()
            if self.current_token.type == RPAREN:
                self.eat(RPAREN)
            sub_query = Nest_Query(attribute=None, op=token, query=node)
            return sub_query

        else:
            left = self.attribute()
        if self.current_token.type in (IN,EQUAL, GREATER, LESSER, GREATEREQUAL, LESSEREQUAL):
            # Comparison
            token = self.current_token.value
            if self.current_token.type == EQUAL:
                self.eat(EQUAL)
            elif self.current_token.type == GREATER:
                self.eat(GREATER)
            elif self.current_token.type == LESSER:
                self.eat(LESSER)
            elif self.current_token.type == GREATEREQUAL:
                self.eat(GREATEREQUAL)
            elif self.current_token.type == LESSEREQUAL:
                self.eat(LESSEREQUAL)
            elif self.current_token.type == IN:
                self.eat(IN)


            # Right: integer, string, or attribute
            if self.current_token.type == INTEGER:
                right = self.current_token.value
                self.eat(INTEGER)
            elif self.current_token.type == STRING:
                right = self.current_token.value
                self.eat(STRING)
            elif self.current_token.type == LPAREN:
                self.eat(LPAREN)
                node = self.query()
                if self.current_token.type == RPAREN:
                    self.eat(RPAREN)
                sub_query = Nest_Query(left, token, node)
                return sub_query
            else: # attribute
                right = self.attribute()
            return Rel_Alg_Select(left, token, right)


    def parse_sql(self, check):
        """
        query: sql_compound_statement
        sql_compound_statement: SELECT attributes FROM (relations | query) WHERE (conditions | attributes IN query)
        """
        node = self.query()
        if not check == 'j':
            self.check_syntax(node)
        if self.current_token.type != EOF:
            self.error()
        self.eat(EOF)
        return node

    def check_syntax(self, query):
        _relations = list()
        _aliases = list()
        for relation in query.relations:
            if not relation.relation in RELATIONS:
                raise Exception('Relation {} not in the database.'.format(relation.relation))
            else:
                _relations.append(relation.relation)
                if relation.alias:
                    _aliases.append(relation.alias)

        for attribute in query.projects:
            if isinstance(attribute, Attr):
                self.check_attribute(attribute, _relations, _aliases)
            #else: Ag Function

        for condition in query.selects:
            if isinstance(condition, Nest_Query):
                self.check_syntax(condition.query)
                if condition.attribute:
                    self.check_attribute(condition.attribute, _relations, _aliases)
            elif isinstance(condition, Rel_Alg_Select):
                self.check_attribute(condition.left, _relations, _aliases)
                if isinstance(condition.right, Attr):
                    self.check_attribute(condition.right, _relations, _aliases)
            #else its an Ag function

    def check_attribute(self, attribute, _relations, _aliases):
        if attribute.relation:
            if not (attribute.relation in _relations or attribute.relation in _aliases):
                raise Exception('Relation or alias {} is not used in this query'.format(attribute.relation))
            else:
                if attribute.relation in _aliases:
                    relation = _relations[_aliases.index(attribute.relation)]
                else:
                    relation = attribute.relation
                attributes = ATTRIBUTES[relation]
                if not attribute.attribute in attributes:
                    raise Exception(
                        'Attribute {} is not in the attributes for relation {}'.format(attribute.attribute, relation))
        else:
            red_flag = True
            for relation in _relations:
                attributes = ATTRIBUTES[relation]
                if attribute.attribute in attributes:
                    red_flag = False
            if red_flag:
                raise Exception('Attribute {} is not an any of the relations in this query'.format(attribute.attribute))
```

```
###############################################################################
#                                                                             #
#  INTERPRETER                                                                #
#                                                                             #
###############################################################################

class NodeVisitor(object):
    def visit(self, node):
        method_name = 'visit_' + type(node).__name__
        visitor = getattr(self, method_name, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        raise Exception('No visit_{} method'.format(type(node).__name__))


class Interpreter(NodeVisitor):

    GLOBAL_SCOPE = {}
    QUERIES = list()
    SET_OPS = list()

    def __init__(self, parser):
        self.parser = parser

    def visit_Set_Op(self, set_op):
        left = self.visit(set_op.left)
        op = set_op.op
        right = self.visit(set_op.right)
        return Set_Op(left, right, op)

    def visit_Nest_Query(self, nest_query):
        if nest_query.attribute:
            left = nest_query.attribute
            if nest_query.op == 'IN':
                op = '='
            else:
                op = nest_query.op
            if isinstance(nest_query.query, Query):
                right = nest_query.query.projects.pop(0) #Only one ever
                condition = Rel_Alg_Select(left, op, right, 'AND')
                nest_query.query.selects.insert(0, condition)
        return self.visit(nest_query.query)

    def visit_Query(self, query):
        selects = list()
        projects = list()
        relations = list()
        for item in query.projects:
            projects.append(self.visit(item))
        for item in query.relations:
            relations.append(self.visit(item))
        new_query = Query(projects, relations)
        for item in query.selects:
            if isinstance(item, Nest_Query):
                nested_query = self.visit(item)
                if isinstance(nested_query, Query):
                    for itemx in nested_query.relations:
                        relations.append(itemx)
                    for itemx in nested_query.selects:
                        selects.append(itemx)
                else:
                    new_query.nested = nested_query
            else:
                selects.append(self.visit(item))
        new_query.selects = selects
        if query.groupby:
            new_query.groupby = query.groupby
        if query.having:
            new_query.having = query.having
        return new_query


    def visit_Rel_Alg_Select(self, node):
        return node

    def visit_list(self, node):
        for item in node:
            self.visit(item)

    def visit_Compound(self, node):
        for child in node.children:
            self.visit(child)

    def visit_Attr(self, node):
        return node

    def visit_Ag_Function(self, node):
        return node

    def visit_Rel(self, node):
        return node

    def interpret(self, check):
        tree = self.parser.parse_sql(check)
        if tree is None:
            return ''
        return self.visit(tree)

def print_rel_alg(interpreter, end=''):
    from colorama import init, Fore, Back, Style
    init()
    if interpreter.having:
        print(Fore.MAGENTA + 'HAVING [', end='')
        for idx, item in enumerate(interpreter.having):
            if idx == len(interpreter.having) - 1:
                print('{}] ('.format(item), end='')
            else:
                print('{}, '.format(item), end='')

    if interpreter.groupby:
        print(Fore.GREEN + 'GROUP BY [', end='')
        for idx, item in enumerate(interpreter.groupby):
            if idx == len(interpreter.groupby) - 1:
                print('{}] ('.format(item), end='')
            else:
                print('{}, '.format(item), end='')
        print(Fore.RESET, end='')

    print(Fore.LIGHTYELLOW_EX + 'PROJECT [', end='')
    for idx, item in enumerate(interpreter.projects):
        if idx == len(interpreter.projects) - 1:
            print('{}] ('.format(item), end='')
        else:
            print('{}, '.format(item), end='')
    print(Fore.LIGHTBLUE_EX + 'SELECT [', end='')
    for idx, item in enumerate(interpreter.selects):
        if idx == len(interpreter.selects) - 1:
            print(item, end='')
        else:
            print('{} '.format(item), end='')
    print('] (' + Fore.WHITE, end='')

    print(Fore.RED, end='')
    for idx, rel in enumerate(interpreter.relations):
        if idx == len(interpreter.relations) - 1:
            print(rel, end='')
            print(']'*idx, end='')
        else:
            print(rel, end='')
            print(' X [', end='')


    print(Fore.LIGHTBLUE_EX + ')' + Fore.LIGHTYELLOW_EX + ')', end='')
    if interpreter.having:
        print(Fore.GREEN + ')', end='')
    if interpreter.groupby:
        print(Fore.MAGENTA + ')', end='')

    print(Style.RESET_ALL + end, end='')

def build_set_op_tree(set_op):
    return Tree_Node(build_query_tree(set_op.left), build_query_tree(set_op.right), set_op.op)

def build_query_tree(interpreter, tokenized=None):
    select_optimize = dict()
    join_optimize = list()
    project_optimize = set()
    for project in interpreter.projects:
        if isinstance(project, Attr):
            if not project.relation:
                for key in ATTRIBUTES:
                    for item in ATTRIBUTES[key]:
                        if project.attribute == item:
                            project.relation = key
            project_optimize.add(project)
    remove_later = list()
    for cond in interpreter.selects:
        if not isinstance(cond.right, Attr):
            if cond.left.relation in select_optimize.keys():
                select_optimize[cond.left.relation].append(cond.str_no_next())
            else:
                select_optimize[cond.left.relation] = list()
                select_optimize[cond.left.relation].append(cond.str_no_next())
            remove_later.append(cond)
        else:
            join_optimize.append(cond)
            project_optimize.add(cond.left)
            project_optimize.add(cond.right)
            remove_later.append(cond)
    for item in remove_later:
        interpreter.selects.remove(item)
    having_node = None
    groupby_node = None
    if interpreter.having:
        having_node = Tree_Node(None, None, 'HAVING {}'.format(interpreter.having.__str__()))
    if interpreter.groupby:
        groupby_node = Tree_Node(None, None, 'GROUP BY {}'.format(interpreter.groupby.__str__()))
    project = 'PROJECT ['
    for idx, item in enumerate(interpreter.projects):
        if idx == len(interpreter.projects) - 1:
            project += item.__str__()
        else:
            project += '{}, '.format(item.__str__())
    project += ']'
    tree = Tree_Node(None, None, project)
    if interpreter.selects:
        select = 'SELECT ['
        for idx, item in enumerate(interpreter.selects):
            if idx == len(interpreter.selects) - 1:
                select += item.__str__()
            else:
                select += '{} '.format(item.__str__())
        select += ']'
        select_node = Tree_Node(None, None, select)
        tree.left = select_node
    cross_node = build_cross_tree(interpreter.relations, select_optimize, project_optimize, join_optimize)
    if interpreter.selects:
        tree.left.left = cross_node
    else:
        tree.left = cross_node
    if groupby_node:
        groupby_node.left = tree
        if having_node:
            having_node.left = groupby_node
            return having_node
        return groupby_node
    return tree

def build_cross_tree(cross_prods, select_optimize, project_optimize, join_optimize):
    node = Tree_Node(None, None, None)
    project_left = Tree_Node(value=list())
    select_left = Tree_Node(value=list())
    left = Tree_Node()
    project_right = Tree_Node(value=list())
    select_right = Tree_Node(value=list())
    right = Tree_Node()
    if len(cross_prods) == 1:
        node.value = select_optimize[cross_prods[0].alias]
        node.left = Tree_Node(value=cross_prods[0])
        return node
    elif len(cross_prods) == 2:
        left.value=cross_prods[0]
        right.value=cross_prods[1]
        for item in project_optimize:
            if item.relation == cross_prods[0].alias or item.relation == cross_prods[0].relation:
                project_left.value.append(item)
            elif item.relation == cross_prods[1].alias or item.relation == cross_prods[0].relation:
                project_right.value.append(item)
        if cross_prods[0].alias in select_optimize.keys():
            select_left.value.append(select_optimize[cross_prods[0].alias])
        elif cross_prods[0].relation in select_optimize.keys():
            select_left.value.append(select_optimize[cross_prods[0].relation])
        if cross_prods[1].alias in select_optimize.keys():
            select_right.value.append(select_optimize[cross_prods[1].alias])
        elif cross_prods[1].relation in select_optimize.keys():
            select_right.value.append(select_optimize[cross_prods[1].relation])

        flag = True
        for join in join_optimize:
            if cross_prods[0].alias == join.left.relation or cross_prods[0].relation == join.left.relation or  cross_prods[0].alias == join
                node.value = '|><| {}'.format(join.str_no_next())
                flag = False
        if flag:
            node.value = 'X'
        if project_left.value:
            if select_left.value:
                select_left.left = left
                project_left.left = select_left
                node.left = project_left
            else:
                project_left.left = left
                node.left = project_left
        elif select_left.value:
            select_left.left = left
            node.left = select_left
        else:
            node.left = left

        if project_right.value:
            if select_right.value:
                select_right.left = right
                project_right.left = select_right
                node.right = project_right
            else:
                project_right.left = right
                node.right = project_right
        elif select_right.value:
            select_right.left = right
            node.right = select_right
        else:
            node.right = right

        return node
    else:
        cross_prod = cross_prods.pop(0)
        right.value = cross_prod
        for item in project_optimize:
            if item.relation == cross_prod.alias or item.relation == cross_prod.relation:
                project_right.value.append(item)

        if cross_prod.alias in select_optimize.keys():
            select_right.value.append(select_optimize[cross_prod.alias])

        if project_right.value:
            if select_right.value:
                select_right.left = right
                project_right.left = select_right
                node.right = project_right
            else:
                project_right.left = right
                node.right = project_right
        elif select_right.value:
            select_right.left = right
            node.right = select_right
        else:
            node.right = right


        flag = True
        for join in join_optimize:
            if cross_prod.alias == join.left.relation or cross_prod.relation == join.left.relation:
                for cross in cross_prods:
                    if join.right.relation == cross.alias or join.right.relation == cross.relation:
                        node.value = '|><| {}'.format(join.str_no_next())
                        flag = False
            elif cross_prod.alias == join.right.relation or cross_prod.relation == join.right.relation:
                for cross in cross_prods:
                    if join.left.relation == cross.alias or join.left.relation == cross.relation:
                        node.value = '|><| {}'.format(join.str_no_next())
                        flag = False
        if flag:
            node.value = 'X'

        node.left = build_cross_tree(cross_prods, select_optimize, project_optimize, join_optimize)
        return node

def print_query_tree(tree, spaces):
    if tree:
        spaces += SPACES
        print_query_tree(tree.right, spaces)
        spaces -= SPACES
        if tree.right:
            print(' ' * spaces, end='')
            print('/')
```

```python
        if spaces != 0:
            print(' '*(spaces - SPACES), end='')
            print(' |' + '-'*(SPACES-2), end='')
        print(tree.value)
        if tree.left:
            print(' ' * spaces, end='')
            print('\\')
        spaces += SPACES
        print_query_tree(tree.left, spaces)
        spaces -= SPACES
    return

def print_flat_tree(tree):
    if tree:
        print_flat_tree(tree.right)
        if tree.value == 'X':
            end = ' ['
        else:
            end = ' -> '
        print(tree.value, end=end)
        print_flat_tree(tree.left)
    return


def main():
    import sys
    test_case = input('Test case (a-o): ')
    text = open('part2_{}.txt'.format(test_case), 'r').read()
    text = text.upper()
    lexer = Lexer(text)
    parser = Parser(lexer)
    parser_copy = deepcopy(parser)
    tokenized = parser_copy.parse_sql(test_case)
    interpreter = Interpreter(parser)
    result = interpreter.interpret(test_case)

    number_of_relations = len(result.relations)
    print('######################################')
    print('#          Relation Algebra          #')
    print('######################################\n')
    print_rel_alg(result, end='\n\n')
    print('######################################')
    print('#            Query Tree              #')
    print('######################################\n')
    tree = build_query_tree(result, tokenized)
    print_query_tree(tree, 0)
    # print_flat_tree(tree)
    # print(']'*number_of_relations)


if __name__ == '__main__':
    main()
```