


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

```
1  """
2  Programmer: Gideon Walker and Ryan Leas
3  Date: 11/30/17
4  Class: CS 5300 Databases
5  Assignment: Part 2 of the SQL Compiler Project
6  """
7
8  from copy import deepcopy
9
10 #####
11 #                                                                 #
12 # LEXER                                                            #
13 #                                                                 #
14 #####
15
16 # Token types
17
18 INTEGER      = 'INTEGER'
19 STRING       = 'STRING'
20 PLUS         = 'PLUS'
21 MINUS        = 'MINUS'
22 MUL          = 'MUL'
23 LPAREN       = 'LPAREN'
24 RPAREN       = 'RPAREN'
25 ID           = 'ID'
26 ASSIGN       = 'ASSIGN'
27 SEMI         = 'SEMI'
28 DOT          = 'DOT'
29 COLON        = 'COLON'
30 COMMA        = 'COMMA'
31 EOF          = 'EOF'
32 KEYWORD      = 'KEYWORD'
33 SELECT       = 'SELECT'
34 FROM         = 'FROM'
35 WHERE        = 'WHERE'
36 AS           = 'AS'
37 IN           = 'IN'
38 CONTAINS     = 'CONTAINS'
39 INTERSECT    = 'INTERSECT'
40 UNION        = 'UNION'
41 EXCEPT     = 'EXCEPT'
42 HAVING       = 'HAVING'
43 GROUP        = 'GROUP'
44 BY           = 'BY'
45 AND          = 'AND'
46 OR           = 'OR'
47 EQUAL        = 'EQUAL'
48 GREATER      = 'GREATER'
49 LESSER       = 'LESSER'
50 GREATEREQUAL = 'GREATEREQUAL'
51 LESSEREQUAL  = 'LESSEREQUAL'
52 MIN          = 'MIN'
53 MAX          = 'MAX'
54 SUM          = 'SUM'
```

```

55 COUNT          = 'COUNT'
56 AVG            = 'AVG'
57 _NOT           = 'NOT'
58 EXISTS         = 'EXISTS'
59
60 SPACES = 8
61 RELATIONS = ('SAILORS', 'BOATS', 'RESERVES')
62 ATTRIBUTES = {RELATIONS[0]: ('SID', 'SNAME', 'RATING', 'AGE'),
63               RELATIONS[1]: ('BID', 'BNAME', 'COLOR'),
64               RELATIONS[2]: ('SID', 'BID', 'DAY')}
65
66 # Helper Function
67 def flatten(S):
68     if S == []:
69         return S
70     if isinstance(S[0], list):
71         return flatten(S[0]) + flatten(S[1:])
72     return S[:1] + flatten(S[1:])
73
74 class Tree_Node(object):
75     def __init__(self, left=None, right=None, value=None):
76         self.left = left
77         self.right = right
78         self.value = value
79
80     def __str__(self):
81         return '{ } : { } : { }'.format(self.left, self.value, self.right)
82
83     def __repr__(self):
84         return self.__str__()
85
86 class Token(object):
87     def __init__(self, type, value):
88         self.type = type
89         self.value = value
90
91     def __str__(self):
92         """String representation of the class instance.
93
94         Examples:
95             Token(INTEGER, 3)
96             Token(PLUS, '+')
97             Token(MUL, '*')
98         """
99         return 'Token({type}, {value})'.format(
100             type=self.type,
101             value=repr(self.value)
102         )
103
104     def __repr__(self):
105         return self.__str__()
106
107
108 RESERVED_KEYWORDS = {
109     'SELECT': Token('SELECT', 'SELECT'),
110     'FROM': Token('FROM', 'FROM'),
111     'WHERE': Token('WHERE', 'WHERE'),
112     'AS': Token('AS', 'AS'),
113     'AND': Token('AND', 'AND'),
114     'OR': Token('OR', 'OR'),
115     'IN': Token('IN', 'IN'),
116     'CONTAINS': Token('CONTAINS', 'CONTAINS'),
117     'INTERSECT': Token('INTERSECT', 'INTERSECT'),
118     'UNION': Token('UNION', 'UNION'),
119     'EXCEPT': Token('EXCEPT', 'EXCEPT'),
120     'HAVING': Token('HAVING', 'HAVING'),
121     'GROUP': Token('GROUP', 'GROUP'),

```

```

122     'BY': Token('BY', 'BY'),
123     'MIN': Token('MIN', 'MIN'),
124     'MAX': Token('MAX', 'MAX'),
125     'COUNT': Token('COUNT', 'COUNT'),
126     'SUM': Token('SUM', 'SUM'),
127     'AVG': Token('AVG', 'AVG'),
128     'NOT': Token('NOT', 'NOT'),
129     'EXISTS': Token('EXISTS', 'EXISTS'),
130
131 }
132
133
134 class Lexer(object):
135     def __init__(self, text):
136         # client string input, e.g. "4 + 2 * 3 - 6 / 2"
137         self.text = text
138         # self.pos is an index into self.text
139         self.pos = 0
140         self.current_char = self.text[self.pos]
141
142     def error(self):
143         raise Exception('Invalid character near or at "{}'.format(self.current_char))
144
145     def advance(self):
146         """Advance the `pos` pointer and set the `current_char` variable."""
147         self.pos += 1
148         if self.pos > len(self.text) - 1:
149             self.current_char = None # Indicates end of input
150         else:
151             self.current_char = self.text[self.pos]
152
153     def peek(self):
154         peek_pos = self.pos + 1
155         if peek_pos > len(self.text) - 1:
156             return None
157         else:
158             return self.text[peek_pos]
159
160     def skip_whitespace(self):
161         while self.current_char is not None and self.current_char.isspace():
162             self.advance()
163
164     def integer(self):
165         """Return a (multidigit) integer consumed from the input."""
166         result = ''
167         while self.current_char is not None and self.current_char.isdigit():
168             result += self.current_char
169             self.advance()
170         return int(result)
171
172     def string(self):
173         """ Return a string consumed from the input """
174         result = ''
175         if self.current_char == '"':
176             self.advance()
177             while self.current_char != '"':
178                 result += str(self.current_char)
179                 self.advance()
180             if self.current_char == '"':
181                 self.advance()
182             else:
183                 self.error()
184             return result
185
186     def _id(self):
187         """Handle identifiers and reserved keywords"""
188         result = ''

```

```
189 while self.current_char is not None and self.current_char.isalnum():
190     result += self.current_char
191     self.advance()
192
193 token = RESERVED_KEYWORDS.get(result, Token(ID, result)) # Gets the keyword or returns identifier token
194 return token
195
196 def get_next_token(self):
197     """Lexical analyzer (also known as scanner or tokenizer)
198
199     This method is responsible for breaking a sentence
200     apart into tokens. One token at a time.
201     """
202     while self.current_char is not None:
203
204         if self.current_char.isspace():
205             self.skip_whitespace()
206             continue
207
208         if self.current_char.isalpha():
209             return self._id()
210
211         if self.current_char.isdigit():
212             return Token(INTEGER, self.integer())
213
214         if self.current_char == '"':
215             return Token(STRING, self.string())
216
217         if self.current_char == ';':
218             self.advance()
219             return Token(SEMI, ';')
220
219         if self.current_char == '*':
222             self.advance()
223             return Token(MUL, '*')
224
225         if self.current_char == '(':
226             self.advance()
227             return Token(LPAREN, '(')
228
229         if self.current_char == ')':
230             self.advance()
231             return Token(RPAREN, ')')
232
233         if self.current_char == '.':
234             self.advance()
235             return Token(DOT, '.')
236
237         if self.current_char == '=':
238             self.advance()
239             return Token(EQUAL, '=')
240
241         if self.current_char == '>' and self.peek() == '=':
242             self.advance()
243             self.advance()
244             return Token(GREATEREQUAL, '>=')
245
246         if self.current_char == '<' and self.peek() == '=':
247             self.advance()
248             self.advance()
249             return Token(LESSEREQUAL, '<=')
250
251         if self.current_char == '>':
252             self.advance()
253             return Token(GREATER, '>')
254
255         if self.current_char == '<':
```

```

256         self.advance()
257         return Token(LESSER, '<')
258
259     if self.current_char == ',':
260         self.advance()
261         return Token(COMMA, ',')
262
263     self.error()
264
265     return Token(EOF, None)
266
267
268 #####
269 #                                     #
270 #  PARSER                           #
271 #                                     #
272 #####
273
274 class AST(object):
275     pass
276
277 class Rel_Algebra_Select(AST):
278     def __init__(self, left, op, right, next=None):
279         self.left = left
280         self.token = self.op = op
281         self.right = right
282         self.next = next
283
284     def __eq__(self, other):
285         if isinstance(other, Rel_Algebra_Select):
286             if self.__str__() == other.__str__():
287                 return True
288             return False
289
290     def __str__(self):
291         result = '{} {} {}'.format(self.left.__str__(), self.op, self.right.__str__())
292         if self.next:
293             result += ' {}'.format(self.next)
294         return result
295
296     def str_no_next(self):
297         return '{} {} {}'.format(self.left.__str__(), self.op, self.right.__str__())
298
299     def __repr__(self):
300         return self.__str__()
301
302 class Attr(AST):
303     def __init__(self, attribute, relation=None):
304         self.attribute = attribute.value
305         if relation:
306             self.relation = relation.value
307         else:
308             self.relation = None
309
310     def __str__(self):
311         result = self.attribute
312         if self.relation:
313             result = '{}.{}'.format(self.relation, result)
314         return result
315
316     def __repr__(self):
317         return self.__str__()
318
319
320 class Ag_Function(AST):
321     def __init__(self, function, attribute, alias=None):
322         self.function = function

```

```
323     self.attribute = attribute
324     self.alias = alias
325
326     def __str__(self):
327         result = '{}({})'.format(self.function, self.attribute)
328         if self.alias:
329             result += ' AS {}'.format(self.alias)
330         return result
331
332     def __repr__(self):
333         return self.__str__()
334
335
336 class Rel(AST):
337     def __init__(self, relation, alias=None):
338         self.relation = relation.value
339         if alias:
340             self.alias = alias.value
341         else:
342             self.alias = None
343
344     def __eq__(self, other):
345         if isinstance(other, str):
346             if self.relation.__str__() == other:
347                 return True
348             else:
349                 if self.relation == other.relation:
350                     if not self.alias and not other.alias:
351                         return True
352                     if self.alias and other.alias:
353                         if self.alias == other.alias:
354                             return True
355                 return False
356
357     def same_relation(self, other):
358         if other == self.relation or other == self.alias:
359             return True
360         else:
361             return False
362
363     def __str__(self):
364         result = self.relation
365         if self.alias:
366             result = '{} AS {}'.format(result, self.alias)
367         return result
368
369     def __repr__(self):
370         return self.__str__()
371
372 class Query(AST):
373     def __init__(self, projects, relations, selects=None, groupby=None, having=None, nested=None):
374         self.selects = selects
375         self.projects = projects
376         self.relations = relations
377         self.groupby = groupby
378         self.having = having
379         self.nested = nested
380
381 class Nest_Query(AST):
382     def __init__(self, attribute, op, query):
383         self.attribute = attribute
384         self.op = op
385         self.query = query
386
387 class Set_Op(AST):
388     def __init__(self, left=None, right=None, op=None):
389         self.left = left
```

```

390         self.right = right
391         self.op = op
392
393     # class In(AST):
394     #     def __init__(self, attribute, select):
395     #         pass
396
397     class Parser(object):
398         def __init__(self, lexer):
399             self.lexer = lexer
400             # set current token to the first token taken from the input
401             self.current_token = self.lexer.get_next_token()
402             # previous token used to make error message more helpful
403             self.prev_token = None
404
405         def error(self):
406             from colorama import init, Fore
407             init(autoreset=True)
408             raise Exception(Fore.RED + 'Invalid syntax near or at "{} {}"'.format(self.prev_token.value, self.current_token.value))
409
410         def eat(self, token_type):
411             # compare the current token type with the passed token
412             # type and if they match then "eat" the current token
413             # and assign the next token to the self.current_token,
414             # otherwise raise an exception.
415             if self.current_token.type == token_type:
416                 # print(self.current_token)
417                 self.prev_token = self.current_token
418                 self.current_token = self.lexer.get_next_token()
419             else:
420                 self.error()
421
422         def query(self):
423             # query: compound statement
424             #         | (? compound statement )?
425             if self.current_token.type == LPAREN:
426                 self.eat(LPAREN)
427             node = self.sql_compound_statement()
428             if self.current_token.type == RPAREN:
429                 self.eat(RPAREN)
430             # self.eat(SEMI)
431             return node
432
433         def sql_compound_statement(self):
434             """
435             note: ? means 0 or 1 instances
436             sql_compound_statement: SELECT attribute_list
437                                     FROM relation_list
438                                     (WHERE condition_list)?
439                                     (GROUP BY attribute_list)?
440                                     (HAVING condition_list)?
441                                     (INTERSECT | UNION | EXCEPT | CONTAINS sql_compound_statement)?
442             """
443             cond_nodes = list()
444             group_by_list = list()
445             having_list = list()
446             compound_statement = None
447             set_op = ''
448             self.eat(SELECT)
449             attr_nodes = self.attribute_list()
450             self.eat(FROM)
451             rel_nodes = self.relation_list()
452             if self.current_token.type == WHERE:
453                 self.eat(WHERE)
454                 cond_nodes = self.condition_list()
455             if self.current_token.type == GROUP:
456                 self.eat(GROUP)

```

```

457         self.eat(BY)
458         group_by_list = self.attribute_list()
459     if self.current_token.type == HAVING:
460         self.eat(HAVING)
461         having_list = self.condition_list()
462     if self.current_token.type == RPAREN:
463         self.eat(RPAREN)
464     if self.current_token.type in (INTERSECT, UNION, EXCEPT, CONTAINS):
465         set_op = self.current_token.type
466         if self.current_token.type == INTERSECT:
467             self.eat(INTERSECT)
468         elif self.current_token.type == UNION:
469             self.eat(UNION)
470         elif self.current_token.type == EXCEPT:
471             self.eat(EXCEPT)
472         elif self.current_token.type == CONTAINS:
473             self.eat(CONTAINS)
474         compound_statement = self.query()
475     query = Query(attr_nodes, rel_nodes, cond_nodes, group_by_list, having_list)
476     if compound_statement:
477         combined = Set_Op(query, compound_statement, set_op)
478         if query.selects:
479             if combined.op == UNION:
480                 query.selects[-1].next = OR
481             elif combined.op == INTERSECT or combined.op == CONTAINS:
482                 query.selects[-1].next = AND
483             elif combined.op == EXCEPT:
484                 query.selects[-1].next = 'AND NOT'
485
486         if query.relations == compound_statement.relations:
487             for query_condition in compound_statement.selects:
488                 if query_condition in query.selects:
489                     continue
490                 else:
491                     query.selects.append(query_condition)
492         else:
493             for relation in compound_statement.relations:
494                 query.relations.append(relation)
495             for condition in compound_statement.selects:
496                 query.selects.append(condition)
497
498     return query
499
500 def attribute_list(self):
501     """
502     attribute_list : (attribute | ag_function) (COMMA attribute_list)*
503     """
504     if self.current_token.type == ID:
505         node = self.attribute()
506     else:
507         node = self.ag_function()
508     results = [node]
509     while self.current_token.type == COMMA:
510         self.eat(COMMA)
511         if self.current_token.type == ID:
512             next = self.attribute()
513         else:
514             next = self.ag_function()
515         results.append(next)
516     return results
517
518 def ag_function(self):
519     """ag_function: (MIN | MAX | SUM | COUNT | AVG) (attribute) (AS alias):"""
520     function = self.current_token.value
521     if self.current_token.type == MAX:
522         self.eat(MAX)
523     elif self.current_token.type == MIN:

```



```

524         self.eat(MIN)
525     elif self.current_token.type == SUM:
526         self.eat(SUM)
527     elif self.current_token.type == COUNT:
528         self.eat(COUNT)
529     elif self.current_token.type == AVG:
530         self.eat(AVG)
531     else:
532         self.error()
533
534     self.eat(LPAREN)
535     attribute = self.attribute()
536     self.eat(RPAREN)
537
538     if self.current_token.type == AS:
539         self.eat(AS)
540         alias = self.current_token.value
541         self.eat(ID)
542         return Ag_Function(function, attribute, alias)
543
544     return Ag_Function(function, attribute)
545
546 def attribute(self):
547     """
548     attribute : identifier
549               | identifier DOT identifier
550               | STAR aka MUL
551
552     """
553     node = Attr(self.current_token)
554     if self.current_token.type == MUL:
555         self.eat(MUL)
556     else:
557         self.eat(ID)
558         if self.current_token.type == DOT:
559             self.eat(DOT)
560             node.relation = node.attribute
561             node.attribute = self.current_token.value
562             self.eat(ID)
563     return node
564
565 def relation_list(self):
566     """
567     relation_list : relation
568                   | relation COMMA relation_list
569
570     """
571     node = self.relation()
572     results = [node]
573     while self.current_token.type == COMMA:
574         self.eat(COMMA)
575         results.append(self.relation())
576     return results
577
578 def relation(self):
579     """
580     relation : identifier
581              | identifier (AS)? identifier
582
583     """
584     node = Rel(self.current_token)
585     self.eat(ID)
586     if self.current_token.type == AS:
587         self.eat(AS)
588     if self.current_token.type == ID:
589         node.alias = self.current_token.value
590         self.eat(ID)
591     return node

```

```

591 def condition_list(self):
592     """
593     condition_list : condition
594                     | condition (AND | OR) condition_list
595     """
596     node = self.condition()
597     results = [node]
598     while self.current_token.type in (AND, OR):
599         results[-1].next = self.current_token.value
600         if self.current_token.type == AND:
601             self.eat(AND)
602         else:
603             self.eat(OR)
604         results.append(self.condition())
605     return results
606
607 def condition(self):
608     """
609     condition : attribute (EQUAL | GREATER | LESSER | GREATEREQUAL | LESSEREQUAL) (attribute | INTEGER | STRING)
610               | attribute (IN | NOT EXISTS) LPAREN sql_compound_statement RPAREN
611     """
612     # Left is always attribute
613     if self.current_token.type in (SUM, COUNT, MAX, MIN, AVG):
614         left = self.ag_function()
615     elif self.current_token.type == _NOT:
616         token = 'AND NOT'
617         self.eat(_NOT)
618         self.eat(EXISTS)
619         self.eat(LPAREN)
620         node = self.query()
621         if self.current_token.type == RPAREN:
622             self.eat(RPAREN)
623         sub_query = Nest_Query(attribute=None, op=token, query=node)
624         return sub_query
625
626     else:
627         left = self.attribute()
628     if self.current_token.type in (IN, EQUAL, GREATER, LESSER, GREATEREQUAL, LESSEREQUAL):
629         # Comparison
630         token = self.current_token.value
631         if self.current_token.type == EQUAL:
632             self.eat(EQUAL)
633         elif self.current_token.type == GREATER:
634             self.eat(GREATER)
635         elif self.current_token.type == LESSER:
636             self.eat(LESSER)
637         elif self.current_token.type == GREATEREQUAL:
638             self.eat(GREATEREQUAL)
639         elif self.current_token.type == LESSEREQUAL:
640             self.eat(LESSEREQUAL)
641         elif self.current_token.type == IN:
642             self.eat(IN)
643
644         # Right: integer, string, or attribute
645         if self.current_token.type == INTEGER:
646             right = self.current_token.value
647             self.eat(INTEGER)
648         elif self.current_token.type == STRING:
649             right = self.current_token.value
650             self.eat(STRING)
651         elif self.current_token.type == LPAREN:
652             self.eat(LPAREN)
653             node = self.query()
654             if self.current_token.type == RPAREN:
655                 self.eat(RPAREN)
656             sub_query = Nest_Query(left, token, node)

```

```

658         return sub_query
659     else: # attribute
660         right = self.attribute()
661     return Rel_Alg_Select(left, token, right)
662
663
664 def parse_sql(self, check):
665     """
666     query: sql_compound_statement
667     sql_compound_statement: SELECT attributes FROM (relations | query) WHERE (conditions | attributes IN query)
668     """
669     node = self.query()
670     if not check == 'j':
671         self.check_syntax(node)
672     if self.current_token.type != EOF:
673         self.error()
674     self.eat(EOF)
675     return node
676
677 def check_syntax(self, query):
678     _relations = list()
679     _aliases = list()
680     for relation in query.relations:
681         if not relation.relation in RELATIONS:
682             raise Exception('Relation {} not in the database.'.format(relation.relation))
683         else:
684             _relations.append(relation.relation)
685             if relation.alias:
686                 _aliases.append(relation.alias)
687
688     for attribute in query.projects:
689         if isinstance(attribute, Attr):
690             self.check_attribute(attribute, _relations, _aliases)
691         #else: Ag Function
692
693     for condition in query.selects:
694         if isinstance(condition, Nest_Query):
695             self.check_syntax(condition.query)
696             if condition.attribute:
697                 self.check_attribute(condition.attribute, _relations, _aliases)
698         elif isinstance(condition, Rel_Alg_Select):
699             self.check_attribute(condition.left, _relations, _aliases)
700             if isinstance(condition.right, Attr):
701                 self.check_attribute(condition.right, _relations, _aliases)
702         #else its an Ag function
703
704 def check_attribute(self, attribute, _relations, _aliases):
705     if attribute.relation:
706         if not (attribute.relation in _relations or attribute.relation in _aliases):
707             raise Exception('Relation or alias {} is not used in this query'.format(attribute.relation))
708         else:
709             if attribute.relation in _aliases:
710                 relation = _relations[_aliases.index(attribute.relation)]
711             else:
712                 relation = attribute.relation
713                 attributes = ATTRIBUTES[relation]
714             if not attribute.attribute in attributes:
715                 raise Exception(
716                     'Attribute {} is not in the attributes for relation {}'.format(attribute.attribute, relation))
717     else:
718         red_flag = True
719         for relation in _relations:
720             attributes = ATTRIBUTES[relation]
721             if attribute.attribute in attributes:
722                 red_flag = False
723         if red_flag:
724             raise Exception('Attribute {} is not an any of the relations in this query'.format(attribute.attribute))

```

```

725 #####
726 #                                                                 #
727 # INTERPRETER                                                    #
728 #                                                                 #
729 #####
730
731 class NodeVisitor(object):
732     def visit(self, node):
733         method_name = 'visit_' + type(node).__name__
734         visitor = getattr(self, method_name, self.generic_visit)
735         return visitor(node)
736
737     def generic_visit(self, node):
738         raise Exception('No visit_{} method'.format(type(node).__name__))
739
740
741 class Interpreter(NodeVisitor):
742
743     GLOBAL_SCOPE = {}
744     QUERIES = list()
745     SET_OPS = list()
746
747     def __init__(self, parser):
748         self.parser = parser
749
750     def visit_Set_Op(self, set_op):
751         left = self.visit(set_op.left)
752         op = set_op.op
753         right = self.visit(set_op.right)
754         return Set_Op(left, right, op)
755
756     def visit_Nest_Query(self, nest_query):
757         if nest_query.attribute:
758             left = nest_query.attribute
759             if nest_query.op == 'IN':
760                 op = '='
761             else:
762                 op = nest_query.op
763             if isinstance(nest_query.query, Query):
764                 right = nest_query.query.projects.pop(0) #Only one ever
765                 condition = Rel_Alg_Select(left, op, right, 'AND')
766                 nest_query.query.selects.insert(0, condition)
767             return self.visit(nest_query.query)
768
769     def visit_Query(self, query):
770         selects = list()
771         projects = list()
772         relations = list()
773         for item in query.projects:
774             projects.append(self.visit(item))
775         for item in query.relations:
776             relations.append(self.visit(item))
777         new_query = Query(projects, relations)
778         for item in query.selects:
779             if isinstance(item, Nest_Query):
780                 nested_query = self.visit(item)
781                 if isinstance(nested_query, Query):
782                     for itemx in nested_query.relations:
783                         relations.append(itemx)
784                     for itemx in nested_query.selects:
785                         selects.append(itemx)
786                 else:
787                     new_query.nested = nested_query
788             else:
789                 selects.append(self.visit(item))
790         new_query.selects = selects
791         if query.groupby:

```

```

792         new_query.groupby = query.groupby
793     if query.having:
794         new_query.having = query.having
795     return new_query
796
797
798     def visit_Rel_Alg_Select(self, node):
799         return node
800
801     def visit_list(self, node):
802         for item in node:
803             self.visit(item)
804
805     def visit_Compound(self, node):
806         for child in node.children:
807             self.visit(child)
808
809     def visit_Attr(self, node):
810         return node
811
812     def visit_Ag_Function(self, node):
813         return node
814
815     def visit_Rel(self, node):
816         return node
817
818     def interpret(self, check):
819         tree = self.parser.parse_sql(check)
820         if tree is None:
821             return ''
822         return self.visit(tree)
823
824     def print_rel_alg(interpreter, end=''):
825         from colorama import init, Fore, Back, Style
826         init()
827         if interpreter.having:
828             print(Fore.MAGENTA + 'HAVING [' , end='')
829             for idx, item in enumerate(interpreter.having):
830                 if idx == len(interpreter.having) - 1:
831                     print('{}]' (.format(item), end='')
832                 else:
833                     print('{} , '.format(item), end='')
834
835         if interpreter.groupby:
836             print(Fore.GREEN + 'GROUP BY [' , end='')
837             for idx, item in enumerate(interpreter.groupby):
838                 if idx == len(interpreter.groupby) - 1:
839                     print('{}]' (.format(item), end='')
840                 else:
841                     print('{} , '.format(item), end='')
842             print(Fore.RESET, end='')
843
844         print(Fore.LIGHTYELLOW_EX + 'PROJECT [' , end='')
845         for idx, item in enumerate(interpreter.projects):
846             if idx == len(interpreter.projects) - 1:
847                 print('{}]' (.format(item), end='')
848             else:
849                 print('{} , '.format(item), end='')
850         print(Fore.LIGHTBLUE_EX + 'SELECT [' , end='')
851         for idx, item in enumerate(interpreter.selects):
852             if idx == len(interpreter.selects) - 1:
853                 print(item, end='')
854             else:
855                 print('{} ' .format(item), end='')
856         print(']' ( ' + Fore.WHITE, end='')
857
858         print(Fore.RED, end='')

```

```

859     for idx, rel in enumerate(interpreter.relations):
860         if idx == len(interpreter.relations) - 1:
861             print(rel, end='')
862             print(']' * idx, end='')
863         else:
864             print(rel, end='')
865             print(' X [' , end='')
866
867
868     print(Fore.LIGHTBLUE_EX + ')' + Fore.LIGHTYELLOW_EX + ')', end='')
869     if interpreter.having:
870         print(Fore.GREEN + ')', end='')
871     if interpreter.groupby:
872         print(Fore.MAGENTA + ')', end='')
873
874     print(Style.RESET_ALL + end, end='')
875
876 def build_set_op_tree(set_op):
877     return Tree_Node(build_query_tree(set_op.left), build_query_tree(set_op.right), set_op.op)
878
879 def build_query_tree(interpreter, tokenized=None):
880     select_optimize = dict()
881     join_optimize = list()
882     project_optimize = set()
883     for project in interpreter.projects:
884         if isinstance(project, Attr):
885             if not project.relation:
886                 for key in ATTRIBUTES:
887                     for item in ATTRIBUTES[key]:
888                         if project.attribute == item:
889                             project.relation = key
890                     project_optimize.add(project)
891     remove_later = list()
892     for cond in interpreter.selects:
893         if not isinstance(cond.right, Attr):
894             if cond.left.relation in select_optimize.keys():
895                 select_optimize[cond.left.relation].append(cond.str_no_next())
896             else:
897                 select_optimize[cond.left.relation] = list()
898                 select_optimize[cond.left.relation].append(cond.str_no_next())
899                 remove_later.append(cond)
900         else:
901             join_optimize.append(cond)
902             project_optimize.add(cond.left)
903             project_optimize.add(cond.right)
904             remove_later.append(cond)
905     for item in remove_later:
906         interpreter.selects.remove(item)
907     having_node = None
908     groupby_node = None
909     if interpreter.having:
910         having_node = Tree_Node(None, None, 'HAVING {}'.format(interpreter.having.__str__()))
911     if interpreter.groupby:
912         groupby_node = Tree_Node(None, None, 'GROUP BY {}'.format(interpreter.groupby.__str__()))
913     project = 'PROJECT ['
914     for idx, item in enumerate(interpreter.projects):
915         if idx == len(interpreter.projects) - 1:
916             project += item.__str__()
917         else:
918             project += '{} , '.format(item.__str__())
919     project += ']'
920     tree = Tree_Node(None, None, project)
921     if interpreter.selects:
922         select = 'SELECT ['
923         for idx, item in enumerate(interpreter.selects):
924             if idx == len(interpreter.selects) - 1:
925                 select += item.__str__()

```

```

926         else:
927             select += '{} '.format(item.__str__())
928         select += ']'
929         select_node = Tree_Node(None, None, select)
930         tree.left = select_node
931     cross_node = build_cross_tree(interpreter.relations, select_optimize, project_optimize, join_optimize)
932     if interpreter.selects:
933         tree.left.left = cross_node
934     else:
935         tree.left = cross_node
936     if groupby_node:
937         groupby_node.left = tree
938         if having_node:
939             having_node.left = groupby_node
940             return having_node
941         return groupby_node
942     return tree
943
944 def build_cross_tree(cross_prods, select_optimize, project_optimize, join_optimize):
945     node = Tree_Node(None, None, None)
946     project_left = Tree_Node(value=list())
947     select_left = Tree_Node(value=list())
948     left = Tree_Node()
949     project_right = Tree_Node(value=list())
950     select_right = Tree_Node(value=list())
951     right = Tree_Node()
952     if len(cross_prods) == 1:
953         node.value = select_optimize[cross_prods[0].alias]
954         node.left = Tree_Node(value=cross_prods[0])
955         return node
956     elif len(cross_prods) == 2:
957         left.value=cross_prods[0]
958         right.value=cross_prods[1]
959         for item in project_optimize:
960             if item.relation == cross_prods[0].alias or item.relation == cross_prods[0].relation:
961                 project_left.value.append(item)
962             elif item.relation == cross_prods[1].alias or item.relation == cross_prods[0].relation:
963                 project_right.value.append(item)
964             if cross_prods[0].alias in select_optimize.keys():
965                 select_left.value.append(select_optimize[cross_prods[0].alias])
966             elif cross_prods[0].relation in select_optimize.keys():
967                 select_left.value.append(select_optimize[cross_prods[0].relation])
968             if cross_prods[1].alias in select_optimize.keys():
969                 select_right.value.append(select_optimize[cross_prods[1].alias])
970             elif cross_prods[1].relation in select_optimize.keys():
971                 select_right.value.append(select_optimize[cross_prods[1].relation])
972
973         flag = True
974         for join in join_optimize:
975             if cross_prods[0].alias == join.left.relation or cross_prods[0].relation == join.left.relation or cross_prods[0].alias == join
976                 node.value = '>|<| {}'.format(join.str_no_next())
977                 flag = False
978         if flag:
979             node.value = 'X'
980         if project_left.value:
981             if select_left.value:
982                 select_left.left = left
983                 project_left.left = select_left
984                 node.left = project_left
985             else:
986                 project_left.left = left
987                 node.left = project_left
988         elif select_left.value:
989             select_left.left = left
990             node.left = select_left
991         else:
992             node.left = left

```

```

993
994     if project_right.value:
995         if select_right.value:
996             select_right.left = right
997             project_right.left = select_right
998             node.right = project_right
999         else:
1000             project_right.left = right
1001             node.right = project_right
1002     elif select_right.value:
1003         select_right.left = right
1004         node.right = select_right
1005     else:
1006         node.right = right
1007
1008     return node
1009 else:
1010     cross_prod = cross_prods.pop(0)
1011     right.value = cross_prod
1012     for item in project_optimize:
1013         if item.relation == cross_prod.alias or item.relation == cross_prod.relation:
1014             project_right.value.append(item)
1015
1016     if cross_prod.alias in select_optimize.keys():
1017         select_right.value.append(select_optimize[cross_prod.alias])
1018
1019     if project_right.value:
1020         if select_right.value:
1021             select_right.left = right
1022             project_right.left = select_right
1023             node.right = project_right
1024         else:
1025             project_right.left = right
1026             node.right = project_right
1027     elif select_right.value:
1028         select_right.left = right
1029         node.right = select_right
1030     else:
1031         node.right = right
1032
1033
1034     flag = True
1035     for join in join_optimize:
1036         if cross_prod.alias == join.left.relation or cross_prod.relation == join.left.relation:
1037             for cross in cross_prods:
1038                 if join.right.relation == cross.alias or join.right.relation == cross.relation:
1039                     node.value = '><| {}'.format(join.str_no_next())
1040                     flag = False
1041             elif cross_prod.alias == join.right.relation or cross_prod.relation == join.right.relation:
1042                 for cross in cross_prods:
1043                     if join.left.relation == cross.alias or join.left.relation == cross.relation:
1044                         node.value = '><| {}'.format(join.str_no_next())
1045                         flag = False
1046         if flag:
1047             node.value = 'X'
1048
1049     node.left = build_cross_tree(cross_prods, select_optimize, project_optimize, join_optimize)
1050     return node
1051
1052 def print_query_tree(tree, spaces):
1053     if tree:
1054         spaces += SPACES
1055         print_query_tree(tree.right, spaces)
1056         spaces -= SPACES
1057         if tree.right:
1058             print(' ' * spaces, end='')
1059             print('/')

```



```
1060         if spaces != 0:
1061             print(' '*(spaces - SPACES), end='')
1062             print(' |' + '-'*(SPACES-2), end='')
1063         print(tree.value)
1064         if tree.left:
1065             print(' ' * spaces, end='')
1066             print('\n')
1067         spaces += SPACES
1068         print_query_tree(tree.left, spaces)
1069         spaces -= SPACES
1070     return
1071
1072 def print_flat_tree(tree):
1073     if tree:
1074         print_flat_tree(tree.right)
1075         if tree.value == 'X':
1076             end = ' ['
1077         else:
1078             end = ' -> '
1079         print(tree.value, end=end)
1080         print_flat_tree(tree.left)
1081     return
1082
1083
1084 def main():
1085     import sys
1086     test_case = input('Test case (a-o): ')
1087     text = open('part2_{}.txt'.format(test_case), 'r').read()
1088     text = text.upper()
1089     lexer = Lexer(text)
1090     parser = Parser(lexer)
1091     parser_copy = deepcopy(parser)
1092     tokenized = parser_copy.parse_sql(test_case)
1093     interpreter = Interpreter(parser)
1094     result = interpreter.interpret(test_case)
1095
1096     number_of_relations = len(result.relations)
1097     print('#####')
1098     print('#          Relation Algebra          #')
1099     print('#####\n')
1100     print_rel_alg(result, end='\n\n')
1101     print('#####')
1102     print('#          Query Tree          #')
1103     print('#####\n')
1104     tree = build_query_tree(result, tokenized)
1105     print_query_tree(tree, 0)
1106     # print_flat_tree(tree)
1107     # print(']'*number_of_relations)
1108
1109
1110 if __name__ == '__main__':
1111     main()
```