The Datalog Interpreter

CS236 Project Five

Datalog Interpreter

With the addition of two operations, *union* (U) and *join* (|x|), we are able to extend our relational database management system so that it can fully interpret all of 236-Datalog.

Project Description

Write an RDBMS-based interpreter for a 236-Datalog program.

Processing queries in Project 5 is the same as for Project 4 except that we now also have rules. The output is the same as for Project 4 except that there is an additional first and last line. The first line is the string "Schemes populated after x passes through the Rules.", with x replaced by the number of iterations it takes to converge to a solution. The last line is the string "Done!". (This is fitting since there is no more code to write for CS 236. :))

Compare the time it takes to execute the same tests as we did for Project 3. We have provided test files for you as explained below. You should also ask yourself the question: Why do the results came out the way they do? Although you need not turn in an answer to this question, you should hopefully see that the RDBMS-based implementation is more efficient, and you should be able to explain why.

Requirements (Get the Design Right)

The data structure for Project 5 is the same as for Project 4. The methods for relational-algebra operators are also the same except you will be required to add *union* and *join* methods to the *Relation* class. How you chose and used your data structures from Project 4 affects the difficulty of these methods. Each data structure implementation has its benefits, drawbacks and obstacles to overcome.

Project 5 has two steps. First, add all tuples that can be derived using the rules to the existing relations. Second, process each of the queries in the datalog program as we did in Project 4.

Adding Tuples

To add all tuples to the relations, we repeatedly "execute" the set of rules until we have added all possible facts to the database. To execute a set of rules, we "execute" each rule in the set once. The set of rules are executed in the order they were defined. Most students take one of two approaches for executing rules: the interpretation approach or the command approach.

In the interpretation approach, each rule is executed by translating it into a relational query and then executing the query. If the same rule is ever seen again, it is again translated and executed. Thus, if there are five rules and we "execute" the set of rules in a DatalogProgram three times, then there are 15 translations and 15 executions.

In the command approach, each relational operation is thought of an object or command. Thus a relational query can be converted into a relational expression tree where every operation is a node in the tree. The rules are translated once and only once and stored as a set of expression trees. This set of trees is repeatedly executed until all new tuples have been added to the database. Thus, if there are five rules and we "execute" the set of expression trees three times, then there are 5 translations and 15 executions.

Translating Rules

Translate each rule into a query using the following five steps.

1. For every predicate on the right hand side of a rule, translate it into a sequence of *rename*, *select*, and *project* operations exactly as we did for the queries in Project 4. Each sequence can be considered a single sub-query producing a single relation as an intermediate result. If there are *n* predicates on the right hand side of a rule, there should be *n* intermediate results.

2. Next, if there is a single predicate on the right hand side of the rule, we use the single intermediate result from Step 1 as the result for Step 2. If there are two or more predicates on the right-hand side of a rule, *join* all the intermediate results to form the single result for Step 2. Thus, if p_1, p_2, and p_3 are the intermediate results from step one; you should construct a new query: $p_1 |x| p_2 |x| p_3$.

3. It is possible for the predicate list in the original rule to have free variables (variables not defined in the head of the rule). Perform a *project* operation on the result from Step 2 to remove columns corresponding to free variables. That is, project on the result from Step 2 keeping only those columns whose attribute names appear in the head of the rule.

4. Before taking the union in Step 5, we must first make the result of Step 3 union compatible with the appropriate relation in the database. We do this as follows:

    Select the relation *r* from the database whose name matches the name of the head of the rule. For every variable *v* that appears in column *i* of the head of the rule and column *j* of the result from Step 3, rename the attribute for column *j* of the result to the name of the attribute in column *i* of the schema of *r*.

5. Union the results of Step 4 with the relation in the database whose name is equal to the name of the head of the rule. In Step 4 we called this relation in the database, *r*. Add tuples to relation *r* from the result of Step 4.

Generating all New Tuples – The Fixed-Point Algorithm

After translating the rules, we should have a set of relational-algebra expressions, one expression for each rule. Executing each expression in the set exactly once is called an *iteration*. The fixed-point algorithm repeatedly performs iterations. Each iteration may change the database by adding at least one new tuple to at least one relation in the database. The fixed-point algorithm terminates when an iteration of the rule expression set does not union a new tuple to any relation in the database.

Comparing Times

When your program is working run:

ant timetest5

This should give you a total elapsed time. Compare this with the time it took to execute the timetest3 for Project 3. *Important: for the time test to work, do not include the "Schemes populated…" line in the string output of the body method. Use System.out.println for that line*

Notes About Pass-off Files

- The head of every rule contains only identifiers (no strings)
- The identifiers are unique. That is, no two identifiers in a rule head are the same.
- Every identifier in the head will appear in at least one predicate on the right hand side of the rule.
- The timetest5 is required for passoff. It does not cover all test cases. There are other more complex files used for passoff.

Examples

These are not sufficient to completely test your program. You must have output formatted exactly like the example outputs below:

Example 50 Input: ex50.txt

```
Schemes:
  snap(S,N,A,P)
  csg(C,S,G)
  cn(C,N)
  ncg(N,C,G)

Facts:
  snap('12345','C. Brown','12 Apple St.','555-1234').
  snap('22222','P. Patty','56 Grape Blvd','555-9999').
  snap('33333','Snoopy','12 Apple St.','555-1234').
  csg('CS101','12345','A').
  csg('CS101','22222','B').
  csg('CS101','33333','C').
  csg('EE200','12345','B+').
  csg('EE200','22222','B').

Rules:
  cn(c,n) :- snap(S,n,A,P),csg(c,S,G).
  ncg(n,c,g) :- snap(S,n,A,P),csg(c,S,g).

Queries:
  cn('CS101',Name)?
  ncg('Snoopy',Course,Grade)?
```

Example 50 Output: out50.txt

```
Schemes populated after 2 passes through the
Rules.
cn('CS101',Name)? Yes(3)
  Name='C. Brown'
  Name='P. Patty'
  Name='Snoopy'
ncg('Snoopy',Course,Grade)? Yes(1)
  Course='CS101', Grade='C'
Done!
```

Example 51 Input: ex51.txt

```
Schemes:
    f(A,B)
    g(C,D)
    r(E,F)

Facts:
```

Example 51 Output: out51.txt

```
Schemes populated after 5 passes through the
Rules.
f('3',Z)? Yes(4)
  Z='1'
  Z='3'
  Z='4'
```

```
    f('1','2').
    f('4','3').
    g('3','2').
    r('1','4').
    r('2','5').
    r('3','5').
    r('4','1').

Rules:
    r(A,B) :- f(A,X),g(B,X).
    f(C,D) :- r(D,C).
    g(E,F) :- f(E,X),r(X,F).

Queries:
    f('3',Z)?
    r(Y,'3')?
    f(W,X)?
```

```
  Z='5'
r(Y,'3')? Yes(4)
  Y='1'
  Y='3'
  Y='4'
  Y='5'
f(W,X)? Yes(18)
  W='1', X='1'
  W='1', X='2'
  W='1', X='3'
  W='1', X='4'
  W='1', X='5'
  W='3', X='1'
  W='3', X='3'
  W='3', X='4'
  W='3', X='5'
  W='4', X='1'
  W='4', X='3'
  W='4', X='4'
  W='4', X='5'
  W='5', X='1'
  W='5', X='2'
  W='5', X='3'
  W='5', X='4'
  W='5', X='5'
Done!
```

Example 52 Input: ex52.txt

```
Schemes:
   snap(S,N,A,P)
   csg(C,S,G)
   cp(C,Q)
   cdh(C,D,H)
   cr(C,R)
   before(C1,C2)
   tthCourses(C)
   Schedule(N,C,R,D,H)
   Grades(N,C,G)
   Students(C,R,N,G)

Facts:
   snap('12345','C. Brown','12 Apple St.','555-1234').
   snap('67890','L. Van Pelt','34 Pear Ave.','555-
5678').
   snap('22222','P. Patty','56 Grape Blvd.','555-
9999').
   snap('33333','Snoopy','12 Apple St.','555-1234').
   csg('CS101','12345','A').
   csg('CS101','67890','B').
   csg('EE200','12345','C').
   csg('EE200','22222','B+').
   csg('EE200','33333','B').
   csg('CS101','33333','A-').
   csg('PH100','67890','C+').
   cp('CS101','CS100').
   cp('EE200','EE005').
   cp('EE200','CS100').
   cp('CS120','CS101').
   cp('CS121','CS120').
   cp('CS205','CS101').
   cp('CS206','CS121').
   cp('CS206','CS205').
   cdh('CS101','Tu','10AM').
   cdh('EE200','M','10AM').
   cdh('EE200','W','1PM').
   cdh('EE200','F','10AM').
   cdh('PH100','Th','11AM').
   cr('CS101','Turing Aud.').
   cr('EE200','25 Ohm Hall').
   cr('PH100','Newton Lab.').
```

Example 52 Output: out52.txt

```
Schemes populated after 3 passes through the
Rules.
snap(Id,'Snoopy',A,P)? Yes(1)
  Id='33333', A='12 Apple St.', P='555-1234'
csg(Course,'33333',Grade)? Yes(2)
  Course='CS101', Grade='A-'
  Course='EE200', Grade='B'
cp(Course,'CS100')? Yes(2)
  Course='CS101'
  Course='EE200'
cdh('EE200',Day,Hour)? Yes(3)
  Day='F', Hour='10AM'
  Day='M', Hour='10AM'
  Day='W', Hour='1PM'
cr('CS101',Room)? Yes(1)
  Room='Turing Aud.'
tthCourses(C)? Yes(2)
  C='CS101'
  C='PH100'
before('CS100','CS206')? Yes(1)
before('CS100',prerequisiteTo)? Yes(6)
  prerequisiteTo='CS101'
  prerequisiteTo='CS120'
  prerequisiteTo='CS121'
  prerequisiteTo='CS205'
  prerequisiteTo='CS206'
  prerequisiteTo='EE200'
Schedule('Snoopy',C,R,D,H)? Yes(4)
  C='CS101', R='Turing Aud.', D='Tu', H='10AM'
  C='EE200', R='25 Ohm Hall', D='F', H='10AM'
  C='EE200', R='25 Ohm Hall', D='M', H='10AM'
  C='EE200', R='25 Ohm Hall', D='W', H='1PM'
Grades('Snoopy',C,G)? Yes(2)
  C='CS101', G='A-'
  C='EE200', G='B'
Students('CS101',R,N,G)? Yes(3)
  R='Turing Aud.', N='C. Brown', G='A'
  R='Turing Aud.', N='L. Van Pelt', G='B'
  R='Turing Aud.', N='Snoopy', G='A-'
Done!
```

```
Rules:
   before(C1,C2) :- cp(C2,C1).
   before(C1,C2) :- cp(C3,C1),before(C3,C2).
   tthCourses(C) :- cdh(C,'Tu',H).
   tthCourses(C) :- cdh(C,'Th',H).
   Schedule(N,C,R,D,H) :-
snap(S,N,A,P),csg(C,S,G),cr(C,R),cdh(C,D,H).
   Grades(N,C,G) :- snap(S,N,A,P),csg(C,S,G).
   Students(C,R,N,G) :-
snap(S,N,A,P),csg(C,S,G),cr(C,R).

Queries:
   snap(Id,'Snoopy',A,P)?
   csg(Course,'33333',Grade)?
   cp(Course,'CS100')?
   cdh('EE200',Day,Hour)?
   cr('CS101',Room)?
   tthCourses(C)?
   before('CS100','CS206')?
   before('CS100',prerequisiteTo)?
   Schedule('Snoopy',C,R,D,H)?
   Grades('Snoopy',C,G)?
   Students('CS101',R,N,G)?
```