Table of Contents

input/output files 1

all code explained 3

to be improved 4

# input/output files

1sqlStatements:

CREATE OR REPLACE TABLE

INSERT INTO

CREATE OR REPLACE VIEW … AS SELECT … (actually SELECT statements, just save records as a view)

e.g.:

SQL

CREATE OR REPLACE TABLE A(a string);

CREATE OR REPLACE TABLE B(b string);

CREATE OR REPLACE TABLE C(a string,b string);

INSERT INTO A VALUES ('a1');

INSERT INTO A VALUES ('a2');

INSERT INTO A VALUES ('a3');

INSERT INTO B VALUES ('b1');

INSERT INTO B VALUES ('b2');

INSERT INTO B VALUES ('a1');

INSERT INTO C VALUES ('a1','a1');

INSERT INTO C VALUES ('a1','b2');

INSERT INTO C VALUES ('a2','b2');

CREATE OR REPLACE VIEW PROJECTION\_RULE AS SELECT \* FROM A;

CREATE OR REPLACE VIEW SELECTION\_RULE AS SELECT a FROM A WHERE a='a2';

CREATE OR REPLACE VIEW INNER\_JOIN\_RULE AS SELECT a FROM A INNER JOIN B on A.a=B.b;

CREATE OR REPLACE VIEW LEFT\_JOIN\_RULE AS SELECT \* FROM A LEFT JOIN B on A.a=B.b;

CREATE OR REPLACE VIEW RIGHT\_JOIN\_RULE AS SELECT \* FROM A RIGHT JOIN B on A.a=B.b;

CREATE OR REPLACE VIEW FULL\_JOIN\_RULE AS SELECT \* FROM A FULL JOIN B on A.a=B.b;

CREATE OR REPLACE VIEW UNION\_RULE AS SELECT \* FROM A UNION SELECT \* FROM B;

CREATE OR REPLACE VIEW DIFFERENCE\_RULE AS SELECT \* FROM A EXCEPT SELECT \* FROM B;

1datalogPrograms:

facts

rules: identified by sign :-

* most rules are Positive
* Negation is applied to one rule: Stratification: Negation on IDB
* no recursive rule
* Semi–Positive: can be seen as positive, not tested this time
* all rules are safe, manually evaluated, more code will be added to evaluate automatically

e.g.:

Datalog

A('a1').

A('a2').

A('a3').

B('b1').

B('b2').

B('a1').

C('a1','a1').

C('a1','b2').

C('a2','b2').

PROJECTION\_RULE(a) :- A(a).

SELECTION\_RULE(a) :- A(a), a='a2'.

INNER\_JOIN\_RULE(a) :- ij(A(a), B(b), a=b).

LEFT\_JOIN\_RULE(a,b) :- lj(A(a), B(b), a=b).

RIGHT\_JOIN\_RULE(a,b) :- rj(A(a), B(b), a=b).

FULL\_JOIN\_RULE(a,b) :- fj(A(a), B(b), a=b).

UNION\_RULE(a,b) :- A(a); B(b).

DIFFERENCE\_RULE(a) :- A(a), ¬B(b).

Extra test files: to be tested later:

(to be done:

…

CREATE OR REPLACE RULE

CREATE OR REPLACE FUNCTION

CREATE INDEX

DROP TABLE

DROP VIEW

DROP TRIGGER

CREATE TRIGGER

…

)

# all code explained

Go to https://github.com/niannianli/TestTranslator/tree/master/testtranslator

From bottom to up:

* File named references.pdf, i found this resource online which includes almost all details about Datalog/SQL/Prolog, used as the main reference for this project
* Two files named details/details.docx, whichever you can open: all details about this project
* File named commandLineTestSession: one command line session to test the code
* File named 1translator.jar: executable file: use below commands to run the file:
  + cd /…DIRECTORY WHRERE YOU SAVE THIS FILE…/
  + java –jar 1translator.jar /…DIRECTORY WHERE YOU SAVE INPUT FILE…/INPUT FILE NAME
* Inside folder inputOutputFiles: files named 1sqlStatements/1datalogPrograms: can be input files; 1sqlStatementsOutput/2datalogProgramsOutput are output files generated after translation:
  + cat /…DIRECTORY WHERE YOU SAVE OUTPUT FILE…/OUTPUT FILE NAME
  + you can see the output contents after translation
  + if input is Datalog, output will be SQL
  + if input is SQL, output will be Datalog
* Inside folder ExtraTestFiles: files will be used for further test
* Go to folder src/com/nian/:
* Folder translator:
  + File TranslatorStart: read input file, run translation function, generate output file
  + File Translator: 2 functions: translate Datalog to SQL; or translate SQL to Datalog; run the function being called by TranslatorStart
* Folder sqltodatalog:
  + File StatementType: categorize SQL statements to 3 types for now: CREATE, INSERT, VIEW; VIEW meaning CREATE OR REPLACE VIEW …AS SELECT…, which includes several SQL statements
  + File SQLToDatalogTranslator: this class is called by Translator, to call build functions to parse input SQL and generate output Datalog
  + File SQLStatement: every line of string (from input) ends with a ; is parsed as a SQLStatement; then we can decide what type of SQL statement it is: CREATE/INSERT/VIEW…
  + File SaveStringsUtilOne: when we are parsing SQL statements, save all table names, variables for each table, values(records) for each table, which can be used for later parsing input/building output
  + File DatalogIDBBuilder: parse SQL VIEW statements, generate Datalog rules
  + File DatalogEDBWithValuesBuilder: parse SQL INSERT statements, generate EDBs/facts with values; a table can have many records, each record is an array of values
  + File DatalogEDBBuilder: parse SQL CREATE OR REPLACE TABLE statements, just save table name and variables; do not generate anything; one table only has one list of variables
  + File DatalogBuilder: has 3 functions, each will be called by SQLToDatalogTranslator to build 2 types of Datalog programs; each function is calling one type of builder to build Datalog program
* Folder datalogtosql:
  + File SQLBuilder: function called by DatalogToSQLTranslator to build SQL statements; all info used to build SQL statements are already saved before while parsing
  + File SaveStringsUtilTwo: evaluate(to be done later) and parse input Datalog program, save all info for further use to build SQL statements
  + File ProgramType: decide Datalog to be POSITIVE, SEMI-POSITIVE, OR STRATIFIABLE…not used for now
  + File ProgramEvaluatorParser: we parse only, evaluate to be done; called by DatalogToSQLTranslator
  + File DatalogToSQLTranslator: called by Translator, to translate Datalog to SQL, not stratify or do recursive datalog program for now
  + File DatalogProgram: a datalog program should come with an input, not checking POSITIVE, SEMI\_POSITIVE, STRATIFIABLE for now, assume all input are safe rules

# to be improved

* Many iterations in the code: running time/space not efficient
* 7 types of sql statements for now: CREATE, INSERT, SELECT \* FROM WHERE, SELECT … FROM WHERE, JOIN, UNION, EXCEPT
* Corresponding to 6 types of EDB clause/IDB rules for now: facts with values, selection rule, projection rule, join rule, union rule, difference rule
* SQL/Datalog: not checked yet:
  + pk
  + fk
  + NOT NULL
  + UNIQUE
  + REFERENCES
  + DETERMINED BY
  + GROUP BY
  + Variable Type: INTEGER, VARCHAR…; for now, all string
  + Cache
  + Cartesion
* Assume all input datalog rules are safe, a datalog rule is safe if every variable appears in some positive relational atom, more code can be added to evaluate/ensure datalog rule safety later, as got too many bugs in this part, removed the code for now
* Rule finite-infinite not checked
* Stratification: not done yet
* No recursive datalog rule yet:
  + e.g.:

RULE\_EXAMPLE(X, Y) :- A\_EXAMPLE(X), RULE\_EXAMPLE(X, Y)

WITH RECURSIVE all super(Major,Minor) AS (SELECT PART, SUBPART

FROM assembly

UNION

SELECT assb.PART, all.Minor FROM assembly assb, all super all WHERE assb.SUBPART = all.Major )

SELECT \*

WHERE Minor=’topTube’

CREATE RECURSIVE VIEW allSubparts(Major,Minor) AS SELECT PART SUBPART

FROM assembly

UNION

SELECT all.Major assb.SUBPART FROM allSubparts all, assembly assb WHERE all.Minor=assb.PART

* A Java framework can be used to parse the datalog rules/sql statements, which is not used yet. For now, I parse the input token by token, which is tedious; can try to use the framework later, then the below syntax can be parsed easily:
  + Atoms. An atom has the form a(t1, ..., tn), where a is a predicate (relation) symbol, and ti (1 ≤ i ≤ n) are terms. If i is 0, then the atom is simply written as a.
  + Restricted atoms. A restricted atom has the form -A, where A is an atom built with no built-in.
  + Conditions. A condition is a Boolean expression containing conjunctions (,/2), disjunctions (;/2), built-in comparison operators, constants and variables.
* Code refactoring…