DataBases & Data Mining: Project
A Tiny SQL-Based Data Exchange Engine

March 10, 2016

Abstract

This project will let you delve into the details of the implementation of a (tiny) SQL-based data exchange engine. The project consists of two main parts.

The first part (about 14 points) concerns a data exchange setting with only s-t tgds (without target constraints): a universal solution can be computed via a set of SQL queries.

The second part (about 6 points) focuses on some extensions of the main program. We propose 2 different extensions. You need to answer at least 1 of them to obtain the maximal grade. Apart from that, you can also conceive your own extra features and, depending on their nature, they may let you earn extra points.

This project is to be performed by one or two students. The programming language is to be selected in the following restricted list of authorized languages, at the students' convenience: `C/C++`, `Java`, `Haskell`, `OCaml` or `Python3`. Evaluation criteria (not exhaustively) include: code clarity, elegance, robustness and testing. A small companion report (at most 2 pages, not including annexes) may be included in your project.

As a part of the evaluation will be automatized, it is mandatory to stick precisely to the input file format given in Annex A.1.

The project is due on March 29, 2016 at 1am. You can submit it by mail to `angela.bonifati@univ-lyon1.fr`.

1 The Data Exchange Problem

A schema mapping $\mathcal{M}$ is a triple $\mathcal{M} = \langle \mathbf{S}, \mathbf{T}, \Sigma \rangle$ with $\mathbf{S}$ a *source* schema, $\mathbf{T}$ a *target* schema and Σ a set of *source-to-target tuple-generating dependencies* (s-t tgds), i.e., a set of first-order formulas of the following form :

$$\forall \overline{x}.\forall \overline{y}.(\phi(\overline{x}, \overline{y}) \rightarrow \exists \overline{z}.\psi(\overline{y}, \overline{z}))$$

In the remainder, quantifiers will be omitted, namely s-t tgds are encoded as follows:

$$\phi(\overline{x}, \overline{y}) \rightarrow \psi(\overline{y}, \overline{z})$$

A s-t tgd whose *body* $\phi(\overline{x}, \overline{y})$ consists of a single atom is LAV (Local-as-View) tgd. A s-t tgd whose *head* $\psi(\overline{y}, \overline{z})$ consists of a single atom is a GAV (Global-as-View) tgd. A s-t tgd with conjunctions of atoms in the *body* and in the *head* is a GLAV (Global-Local-as-View) tgd.

In Section 2, the formula $\phi(\overline{x}, \overline{y})$ (resp. $\psi(\overline{y}, \overline{z})$) is a first-order formula built from conjunction and relational atoms having $\overline{x}$ and $\overline{y}$ (resp. $\overline{y}$ and $\overline{z}$) as free variables. These expressions are also known as Conjunctive Queries (CQ).

Example 1.1. Consider the following target (a.k.a. global) schema $\mathbf{T}$ made of the following relations:

- $\mathit{works}(\mathit{Person}, \mathit{Project})$
- $\mathit{area}(\mathit{Project}, \mathit{Field})$

And the following sources relations of $\mathbf{S}$:

- $\mathit{hasjob}(\mathit{Person}, \mathit{Field})$ supposedly from some source 1;
- $\mathit{teaches}(\mathit{Professor}, \mathit{Course})$ and $\mathit{in}(\mathit{Course}, \mathit{Field})$ supposedly from some source 2;

- $get(Researcher, Grant)$ and $for(Grant, Project)$ supposedly from some source 3.

Schemas $\mathbf{S}$ and $\mathbf{T}$ consitute a schema mapping $\mathcal{M} = \langle \mathbf{S}, \mathbf{T}, \Sigma \rangle$ with Σ the following set of s-t tgds :

- $m_1 : hasjob(i, f) \rightarrow works(i, p) \wedge area(p, f)$, a LAV mapping;
- $m_2 : teaches(i, c) \wedge in(c, f) \rightarrow works(i, p) \wedge area(p, f)$, a GLAV mapping;
- $m_3 : get(i, g) \wedge for(g, p) \rightarrow work(i, p)$, a GAV mapping.

2 Tiny SQL-Based Data Exchange Engine

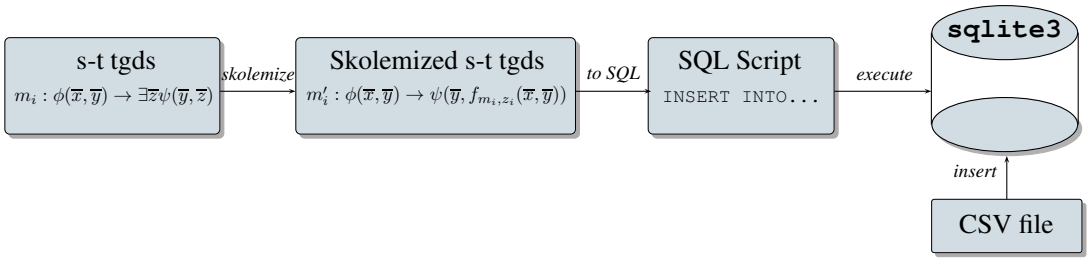

Figure 1: Evaluation pipeline

The whole integration process is divided into several steps. The overall workflow is graphically depicted by Figure 2. The goal of the project is to implement this sequence of steps via a Linux command-line program that:

- takes a `.txt` text file describing a mapping $\mathcal{M} = \langle \mathbf{S}, \mathbf{T}, \Sigma \rangle$ on the standard input stream (`stdin`);
- outputs a `.sql` script file executable by `SQLite3` on the standard output stream (`stdout`).

Additional messages have to by printed exclusively on the standard error stream (`stderr`).

2.1 Input File Format

In the first part, we assume an input file containing in textual form the s-t tgds, whose syntax is specified in BNF in Annex A.1. We have to first build a parser for this file that checks the syntactic correctness of the s-t tgds and store them in adequate structures in memory.

Example 2.1. Example 1.1 is written according to the BNF grammar as follows:

```
SOURCE
hasjob(Person, Field)
teaches(Professor, Course)
in(Course, Field)
get(Researcher, Grant)
for(Grant, Project)

TARGET
works(Person, Project)
area(Project, Field)

MAPPING
hasjob($i, $f), works($i, $p) -> area($p, $f).
teaches($i, $c), in($c, $f) -> works($i, $p), area($p, $f).
get($i, $g), for($g, $p) -> work($i, $p).
```

2.2 Skolemization

Then, we need to take the set of s-t tgds and transform them in their Skolemized form. The goal of Skolemization is to ensure that existentially quantified shared between different atoms in the head are kept linked and unique once inserted as (labeled null) values into the DBMS. Precisely, given a infinite enumerable supply of function symbols $\mathcal{F}$, for each s-t tgds m_i and for each existential variable $z_i \in \overline{z}$, we replace z_i with a term $f_{m_i,z_i}(\overline{y})$ with $f \in \mathcal{F}$.

Example 2.2. Continuing Example 1.1, we obtain the following set of Skolemized s-t tgds:

- $m_1' : hasjob(i, f) \rightarrow works(i, f_{m_1,p}(i, f)) \wedge area(f_{m_1,p}(i, f), f)$
- $m_2' : teaches(i, c) \wedge in(c, f) \rightarrow works(i, f_{m_2,p}(i, f)) \wedge area(f_{m_2,p}(i, f), f)$
- $m_3' : get(i, g) \wedge for(g, p) \rightarrow work(i, p)$

2.3 SQL Generation

Finally, we need to generate an SQL statement of the type `INSERT` for each fact in the conjunction of atoms in the head of m_i, i.e. $\psi(\overline{y}, \overline{z})$. If n is the number of conjuncts in $\psi(\overline{y}, \overline{z})$, we will get n SQL `INSERT` statements for the s-t tgd m_i.

As the BNF grammar in Annex A.1 does not include identifiers for s-t tgds, some synthetic fresh ones has to be generated. In the DBMS, Skolemized terms are represented by strings, where subscripts are written into brackets, as in the following example.

Example 2.3. Assuming the existence of all relations in **S** and in **T** in the DBMS, the mapping m_2' in Example 2.2 is translated into the following SQL `INSERT` statements:

```
INSERT INTO Works(Person, Project)
  SELECT Teaches.Person, 'f[m2,Works.Project](' || Teaches.Person || ',' || In.Field || ')'
  FROM Teaches INNER JOIN In ON Teaches.Course = In.Course;

INSERT INTO Area(Project, Field)
  SELECT 'f[m2,Works.Project](' || Teaches.Person || ',' || In.Field || ')', In.Field
  FROM Teaches INNER JOIN In ON Teaches.Course = In.Course;
```

Note that `'f[m2,Works.Project](' || Teaches.Person || ',' || In.Field ||')'` (the double pipe `||` operator is string concatenation) represents the formal term $f_{m_2,p}(i, f)$ with variables i and f replaced by their concrete values from the instance.

3 Optional extensions

Integration with `SQLite3` The main program generates a `.sql` script file to be run manually. An optional extension will add an optional `-sqlite3 file.db` command-line option. When this option is used, the program establishes a connection to the `SQLite3` database named `file.db`, creates the target schema and runs the integration script.

Integration Scenarios Another optional extension concerns the datasets. The goal is to provide an illustrative integration scenario inspired by those included in existing tools[1] or benchmarks[2] to highlight the benefits and capabilities of your project, (ideally) on real-life datasets.

[1] E.g., `http://www.db.unibas.it/projects/llunatic/`
[2] E.g., `http://www.cs.iit.edu/~dbgroup/research/ibench.php`

A Formal Grammars

A.1 BNF for s-t tgds

The BNF for input files is given below. Note that, for sake of simplicity, usual skippable characters (spaces, tabulations, carriage returns) have been omitted. Comments (also omitted and to be skipped) start with two dashes (--) and finish at the end of the line.

```
START     ::= "SOURCE" SCHEMA "TARGET" SCHEMA "MAPPING" TGDS

SCHEMA    ::= RELATION SCHEMA
            | RELATION

RELATION  ::= NAME "(" ATTS ")"

ATTS      ::= NAME "," ATTS
            | NAME

TGDS      ::= TGD
            | TGDS TGD

TGD       ::= QUERY "->" QUERY "."

QUERY     ::= ATOM "," QUERY
            | ATOM

ATOM      ::= NAME "(" ARGS ")"

ARGS      ::= VALUE "," ARGS
            | VALUE

VALUE     ::= VARIABLE
            | CONSTANT

VARIABLE  ::= "$" NAME

NAME      ::= LETTER
            | LETTER NAME2

NAME2     ::= LETTER_OR_DIGIT
            | LETTER_OR_DIGIT NAME2

CONSTANT  ::= DIGITS

DIGITS    ::= DIGIT DIGITS
            | DIGIT

LETTER_OR_DIGIT ::= LETTER
                  | DIGIT

LETTER    ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j"
            | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t"
            | "u" | "v" | "w" | "x" | "y" | "z"
            | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"
            | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"
            | "U" | "V" | "W" | "X" | "Y" | "Z"
            | "_"

DIGIT     ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```