Cost-based data integration with disjunction

Lucas Willems

5 septembre 2017

Introduction

Informally, the PDQ system (Proof-Driven Querying) works as follow:

- inputs:
  - a *schema* consisting of *relations*.
  - the data described by the schema is not freely available to a user. Instead it can be obtained only via a set of *access methods*. Each access method can expose some data for a given cost.
  - a user *query* written over the schema.
- output: if possible, the **lowest-cost** plan that answers the query; otherwise, nothing.

My intership purpose was to add support for *disjunction* in the constraints of these schemas in order to support more realistic relationships. This would add *union* at the plan level.

A *database* is given by:

- a *schema* $\texttt{Sch}$ (the "organisation"):
  - a finite collection of *relations*. A *relation* is a *relation name* associated with an *arity*, i.e. a positive number.
  - a finite collection of *schema constants* denoted $\vec{\texttt{cst}}$.
  - a collection of *integrity constraints*, i.e. sentences in some logic.
- an *instance* (the "data") for this schema $\texttt{Sch}$ that is a collection of *facts* such that all integrity constraints of $\texttt{Sch}$ are satisfied. A *fact* is the association of a relation R with a tuple $\vec{c}$ of the proper arity.

To request for information from a database, PDQ uses *conjuctive queries* of the form: $Q(\vec{x}) = \exists \vec{y} \bigwedge_{i=1}^{n} A_i(\vec{x}, \vec{y}, \vec{\texttt{cst}})$ where A_i are relational atoms.

- a *binding of Q* is a mapping from its free variables to some values.
- an *homomorphism of Q in an instance I* is a binding that witnesses that Q holds in I.
- the *output of Q over I*, denoted $[\![Q]\!]_I$, is the collection of the homomorphisms of Q over I.

These constraints are built up from *atoms*, which can be either:

- *relational atoms* of the form $R(\vec{t})$, where R is a relation name and each t_i in $\vec{t}$ is a constant or a variable;
- *equality atoms* of the form $t_i = t_j$ with t_i, t_j a constant or a variable.

Below, φ will denote to a conjunction of atoms and ρ and ρ_i to conjunctions of relational atoms.
Before starting my internship, only support for:

- *equality-generating dependencies* (EGDs): $\forall \vec{x}\ [\varphi(\vec{x}) \rightarrow x_i = x_j]$ where x_i and x_j are distinct variables.
- *tuple-generating dependencies* (TGDs): $\forall \vec{x}\ [\varphi(\vec{x}) \rightarrow \exists \vec{y}\ \rho(\vec{x}, \vec{y})]$.

After my internhsip, additional support for *disjunctive TGDs* (DTGDs):

$$\forall \vec{x}\ \left[\varphi(\vec{x}) \rightarrow \exists \vec{y}\ \bigvee_{i=1}^{n} \rho_i(\vec{x}, \vec{y})\right]$$

.

Example

A possible schema for a library:

- relations: `Books(id, title)`, `Members1(id, name)`, `Members2(id, name)`, `Loans(bid, mid)`.
- schema constants: `Dumbo`.
- integrity constraints:
  - some EGDs to make the IDs unique.
  - $\forall x\, y\ [\texttt{Loans}(x, y) \rightarrow \exists z\ \texttt{Books}(x, z)]$;
  - $\forall x\, y\ [\texttt{Loans}(x, y) \rightarrow \exists z\ \texttt{Members1}(y, z) \vee \texttt{Members2}(y, z)]$.

With some instance:
$\{$`Books(1, Dumbo)`, `Books(2, Aladdin)`, `Members(1, Michael)`, `Members(2, Efi)`, `Loans(1, 1)`, `Loans(2, 1)`$\}$.
We may request for:

$$Q(x) = \exists y\, z\ \texttt{Books}(y, \texttt{Dumbo}) \wedge \texttt{Loans}(y, z) \wedge \texttt{Members}(z, x).$$

A common database problem is called *query containment with constraints*. Given two queries Q and Q^* and a conjunction of integrity constraints Σ, we want to determine if $Q \wedge \Sigma \models Q^*$, i.e. if, for every instance I that satisfies Σ, $[\![Q]\!]_I \subseteq [\![Q^*]\!]_I$.

For Q and Q^* conjunctive queries and Σ a conjunction of TGDs, *the chase* can be used to prove these entailments:

1. start with the *canonical database* of Q ($\texttt{CanonDB}(Q)$): the instance with a fact $R(c_1, ..., c_n)$ for each atom $R(x_1, ..., x_n)$ of Q.
2. iteratively derive consequences with the constraints Σ, i.e repeat:
   1. select a *trigger* for a TGD $\delta = \forall \vec{x}\ [\varphi(\vec{x}) \rightarrow \exists \vec{y}\ \rho(\vec{x}, \vec{y})]$, i.e. a tuple $\vec{e}$ such that $\varphi(\vec{e})$ holds. e is *active* if there is no $\vec{f}$ such that $\rho(\vec{e}, \vec{f})$ holds.
   2. *fire the rule*, i.e. add the facts that make $\rho(\vec{e}, \vec{f})$ true.
3. stop and declare success if we have reached an instance that has a "match" for Q^*, i.e. if $x_i \mapsto c_i$ is an homomorphism of Q^*.

A *chase sequence following a set of TGDs* Σ is a sequence of instances, called *chase configurations*, where a configuration is related to its successor by a rule firing of a constraint in Σ.

In PDQ, several chases are implemented but they only execute the step 2.

PDQ takes as input:

- an *access schema* that consists of:
    - a schema with, originally, TGDs as integrity constraints.
    - for each relation R, a collection of *access methods*. Each access method is associated with an *access cost* and a collection of *input positions* of R, i.e. a collection of numbers between 1 and arity(R).
- a conjunctive query Q written over the input schema.

and then outputs, if possible, a plan that answers Q.

- A *plan* is a sequence of access and middleware query commands.
- It *answers* the query if for every instance I satisfying the constraints of the input access schema, the output of the plan on I is the same as the output of Q.

PDQ's plan search algorithms start by transforming the input access schema `Sch` into a *forward accessible schema for Sch* `AcSch(Sch)`:

- the constants are those of `Sch`.
- the relations are:
  - *original relations*, i.e. the relations of `Sch`.
  - *InfAccCopy relations*, i.e. a copy of each relation R called $\mathtt{InfAcc}R$.
  - a unary relation $\mathtt{accessible}(x)$.
- the constraints are:
  - *original integrity constraints*, i.e. the integrity constraints of `Sch`.
  - *InfAccCopy integrity constraints*, i.e. a copy of each of the integrity constraints of `Sch`, with each relation R replaced by $\mathtt{InfAcc}R$.
  - *accessibility axioms*, i.e for each access method on relation R of arity n with input positions j_1, ..., j_m (universal quantifiers omitted):

$$\left[R(x_1, ..., x_n) \wedge \bigwedge_{k=1}^{m} \mathtt{accessible}(x_{j_k}) \rightarrow \mathtt{InfAcc}R(x_1, ..., x_n) \wedge \bigwedge_{j=1}^{n} \mathtt{accessible}(x_j)\right].$$

At this point, finding plans comes down to exploring the space of proofs of this entailment:

$$Q \wedge \Gamma \models \mathtt{InfAcc}Q$$

where Γ is the conjunction of the constraints of $\mathtt{AcSch(Sch)}$ and $\mathtt{InfAcc}Q$ is a query obtained from copying Q and replacing each relation R by $\mathtt{InfAcc}R$.

In PDQ, several plan search algorithms are implemented. **They use the chase** to explore the space of proofs, represented by a tree of chase configurations.

Depending on the programs, the space of proofs is more or less explored.

My purpose: allowing input access schemas to contain DTGDs.
What differs from using only TGDs?

- Firing a TGD $\forall \vec{x}\ [\varphi(\vec{x}) \rightarrow \exists \vec{y}\ \bigwedge_{i=1}^{n} A_i(\vec{x}, \vec{y})]$ for a trigger $\vec{e}$ on an instance I adds $\{A_1(\vec{e}, \vec{f}), ..., A_n(\vec{e}, \vec{f})\}$ to I, with $\vec{f}$ a tuple of fresh values.
  It is **deterministic**, we use **chase configurations**.
- Firing a DTGD $\forall \vec{x}\ \left[\varphi(\vec{x}) \rightarrow \exists \vec{y}\ \bigvee_{i=1}^{n} \bigwedge_{j=1}^{n} A_{i,j}(\vec{x}, \vec{y})\right]$ for a trigger $\vec{e}$ on an instance I adds either $\{A_{1,1}(\vec{e}, \vec{f}), ..., A_{1,n}(\vec{e}, \vec{f})\}$ to I or ... or $\{A_{n,1}(\vec{e}, \vec{f}), ..., A_{n,n}(\vec{e}, \vec{f})\}$ to I.
  It is **not deterministic**, I introduced the notion of **disjunctive chase configurations**, i.e. collections of chase configurations.

This implies to modify:

- all PDQ's chases;
- all PDQ's plan search algorithms.

One of PDQ's chases is the *restricted chase*, *restricted* meaning that only *active* triggers are selected.
Here is the DTGD version of this algorithm:

input: A disjunctive chase configuration `disConfig` ;
A collection of DTGDs Σ ;
while *there is an active trigger for a DTGD* λ *on a configuration* $\mathtt{config}$ *of* $\mathtt{disConfig}$ **do**
  Choose such a trigger ;
  Remove `config` from `disConfig` ;
  foreach *TGD* δ *of the splitted DTGD* λ **do**
    `newConfig` := a copy of `config` ;
    Fire the rule of δ on `newConfig` ;
    Add `newConfig` to `disConfig` ;
  end
end

Example

Parameters (universal quantifiers omitted):

- $\texttt{disConfig} = \{C_0\}$ where $C_0 = \{R(c)\}$;
- $\Sigma = \{\lambda_1, \lambda_2\}$ with $\lambda_1 = R(x) \to S(x) \vee T(x)$, $\lambda_2 = S(x) \to T(x) \vee U(x)$.

Execution:

1. Active trigger (c) for λ_1 on C_0.
   C_0 replaced by $C_1 = \{\mathrm{R(c), S(c)}\}$ and $C_2 = \{\mathrm{R(c), T(c)}\}$.
   Hence, $\texttt{disConfig} = \{C_1, C_2\}$.
2. Active trigger (c) for λ_2 on C_1.
   C_1 replaced by $C_3 = \{\mathrm{R(c), S(c), T(c)}\}$ and $C_4 = \{\mathrm{R(c), S(c), U(c)}\}$.
   Hence, $\texttt{disConfig} = \{C_2, C_4, C_5\}$.
3. No other active trigger for a configuration of disConfig.
   Finally, $\texttt{disConfig} = \{\{\mathrm{R(c), T(c)}\}, \{\mathrm{R(c), S(c), T(c)}\}, \{\mathrm{R(c), S(c), U(c)}\}\}$.

◁

I encountered several problems:

- storing several instances at the same time in a database: use of an `InstanceID`.
- fixing bugs in the code after creating some unit tests:
  - hard;
  - time-consuming;
  - require the use of debugging tools.

Then, I added support for DTGDs in all the chases:

- an abstract class called `Chaser` that contains the common part
- a class for each different chase that:
  - inherits this abstract class;
  - defines a `reason` method, containing the chase strategy.

My work on PDQ

The naive plan search program with DTGDs - Algorithm

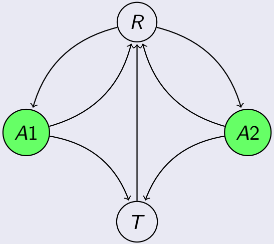
One of PDQ's plan search algorithms is the *naive* one. It explores the full space of proofs of $Q \wedge \Gamma \models \texttt{InfAcc}Q$.
For the TGD version, the space of proof corresponds to a tree of chase configurations. In the DTGD version, it is represented by a **list** of **trees** of **disjunctive chase configurations**:

1. chase `disConfig = {CanonDB(Q)}` with the **original integrity constraints**. For each chase configuration C obtained, create a tree with the disjunctive chase configuration $\{C\}$ as root.
2. iteratively expose facts in the different trees, i.e. repeat:
    1. select an active trigger for an **accessibility axiom** δ on a disjunctive configuration `disC`.
    2. chase `disC` with δ and then with `InfAccCopy` **integrity constraints**.
    3. add this new configuration `newDisC` to `disC`'s children, check if $\texttt{InfAcc}Q$ has a match on `newDisC`, update the best plan of the tree.
3. stop when there is no such trigger anymore, return the union of the best plan found in each tree.

Example

Parameters:

- an access schema composed of:
  - 4 relations:
    - R and T of arity 1 with no access method;
    - $A1$ of arity 1 with a free access of cost 10;
    - $A2$ of arity 1 with a free access of cost 20.
  - 4 integrity constraints Γ (universal quantifiers omitted):
    $\lambda_1 = (R(x) \to A1(x) \vee A2(x))$ / $\lambda_2 = (A1(x) \to R(x) \vee T(x))$ / $\lambda_3 = (A2(x) \to R(x) \vee T(x))$ / $\lambda_4 = (T(x) \to R(x))$.
- a query $Q(x) = R(x)$.



Example

Execution:

1. Chase $\{\texttt{CanonDB(Q)}\}$ i.e. $\{\{R(c)\}\}$ with the original integrity constraints what gives $\{C_1, C_2\}$ with $C_1 = \{R(c), A1(c)\}$, $C_2 = \{R(c), A2(c)\}$. Create trees t_1 with $\{C_1\}$ as root and t_2 with $\{C_2\}$ as root.
2. Search the best plan for t_1:
    1. Active trigger (c) for the accessibility axiom $\delta = A1(x) \rightarrow \texttt{InfAcc}A1(x)$ on C_1.
    2. Chase $\{C_1\}$ with δ and then with `InfAccCopy` integrity constraints:
    $$\texttt{newDisC} = \{\{R(c), A1(c), \texttt{InfAcc}A1(c), \texttt{InfAcc}R(c)\},\\ \{R(c), A1(c), \texttt{InfAcc}A1(c), \texttt{InfAcc}T(c), \texttt{InfAcc}R(c)\}\}$$
    3. Add `newDisC` to t_1, $\texttt{InfAcc}Q(x) = \texttt{InfAcc}R(x)$ has a match in `newDisC`, t_1 has a new best plan that `newDisC`'s best plan of cost 10.
3. This is the same for t_2, but with a best plan of cost 20.
4. Return the global best plan that has a cost of 30.

I encountered several problems:

- before:
  - defining the inferred accessible DTGDs: double inheritance not possible in Java.
  - modifying the "parser" of the `schema.xml` file.
- during: misunderstanding of how the algorithm worked.
- after: debugging the code.

Then, I added support for DTGDs in all the chases:

- mostly, adapting the definitions of *dominance* and *equivalence* to disjunctive chase configurations.

Michael Benedikt suggested to add support for DLV, a tool that natively handles disjunctive logic programming, as an alternative for:

- saturation, i.e. adding the `InfAccCopy` consequences.
  `DLV -silent {facts} {rules}`
- success checking, i.e. checking if `InfAcc`Q has a match.
  `DLV -silent -cautious {facts} {rules} {query}`

Therefore, I had to:

- create a **DLV chase configuration**, i.e. an abstraction of a collection of DLV models.
- use the DLV reader and printer made by Efthymia Tsamoura to write and parse DLV facts and rules.
- create a DLV mapping parser.
- create an abstraction of DLV, i.e. some classes to abstract the DLV binary and its input files.

Conclusion

This internship helped me to:

- discover the field of databases by:
  - reading from Dr. Benedikt's notes and the book "Generating Plans from Proofs";
  - working on PDQ;
  - attending to conferences organized by the departement of computer science of the University of Oxford.
- continue to develop my coding skills:
  - a new language (Java);
  - a new IDE (Eclipse) and its debugging tool.
- learn to work on a big project and deal with code written by others.
- begin to enter the world of research by talking everyday with Dr. Benedikt, the postdocs and the PhD students.