Cost-based data integration with disjunction

Lucas Willems

Under the direction of Michael Benedikt

27th August 2017

Introduction

Informally, the PDQ system (Proof-Driven Querying) takes as input:

- 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.

Then, PDQ determines if the query can be answered using the access methods. If it is the case, a plan can be generated and PDQ returns the lowest-cost one.

For example, PDQ can provide a solution for querying data available through web-based APIs.

Before starting my internship, PDQ supported only simple schemas. My 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.

In the first part of this report, I will introduce the database knowledge required to understand the rest of the report. Then, in the second part, I will present the global working of PDQ. Finally, in the last and main part, I will present my work on the project.

1 Database knowledge

Before starting my internship, as I was not familiar at all with the field of databases and the notions involved in PDQ, I had to read the bibliography sent by Michael Benedikt in order to be able to work on the project.

In particular, I spent a lot of time reading Dr. Benedikt's notes on *modern dependency theory*, explaining from scratch a bunch of database definitions and results, and reading the book "Generating Plans from Proofs", explaining the theory behind PDQ and its main algorithms.

In this part, I will explain the database knowledge required to understand the rest of the report ([1]).

1.1 Database and conjunctive query

Database. A *database* is an organized collection of data. The "organisation" is given by the notion of *schema* and the "data" by the notion of *instance*.

A *schema* consists of:

- 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*. Informally, these represent values that a user might use within accesses, in addition to values exposed from the database.
- a collection of *integrity constraints*, which will be sentences in some logic.

For a schema `Sch`, a *fact* is the association of a relation R with a tuple $\vec{c}$ of the proper arity. An *instance* is a collection of *facts* such that all integrity constraints of `Sch` are satisfied.

Example 1. A library database may have this schema:

- Relations: `Books(id, title)`, `Members(id, name)`, `Loans(bid, mid)`.

- Schema constants: `Dumbo`, in the case we want to get the name of the members who borrowed the books titled Dumbo.
- Integrity constraints: see below.

With this instance:

$$\{\texttt{Books(1, Dumbo)}, \texttt{Books(2, Aladdin)}, \texttt{Members(1, Michael)}, \\ \texttt{Members(2, Efi)}, \texttt{Loans(1, 1)}, \texttt{Loans(2, 1)}\}$$

$\triangleleft$

Integrity constraints of particular interest. 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 either a constant or a variable;
- *equality atoms* of the form $t_i = t_j$ with t_i, t_j either a constant or a variable.

Before starting my internship, only 2 kind of constraints were supported:

- *equality-generating dependencies* (EGDs), given syntactically by:

$$\forall \vec{x}\ [\varphi(\vec{x}) \rightarrow x_i = x_j]$$

  where φ is a conjunction of atoms and x_i and x_j are distinct variables.
- *tuple-generating dependencies* (TGDs), given syntactically by:

$$\forall \vec{x}\ [\varphi(\vec{x}) \rightarrow \exists \vec{y}\ \rho(\vec{x}, \vec{y})]$$

  where φ is a conjunction of atoms and ρ a conjunction of relational atoms.

Example 2. The integrity constraints of the library schema may be composed of EGDs:

- $\forall x\, y\, z\ [\texttt{Books}(x, y) \wedge \texttt{Books}(x, z) \rightarrow y = z]$
- $\forall x\, y\, z\ [\texttt{Members}(x, y) \wedge \texttt{Members}(x, z) \rightarrow y = z]$

and TGDs:

- $\forall x\, y\ [\texttt{Loans}(x, y) \rightarrow \exists z\ \texttt{Books}(x, z)]$
- $\forall x\, y\ [\texttt{Loans}(x, y) \rightarrow \exists z\ \texttt{Members}(y, z)]$

$\triangleleft$

My purpose was to add support for *disjunctive tuple-generating dependencies* (DTGDs), given syntactically by:

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

where φ is a conjunction of atoms and ρ_i conjunctions of relational atoms.

Example 3. Let's imagine that members are not stored in an unique relation `Members` but in two relations `Members1` and `Members2`. In this case, the TGD:

$$\forall x\, y\ [\texttt{Loans}(x, y) \rightarrow \exists z\ \texttt{Members}(y, z)]$$

would be replaced by the DTGD:

$$\forall x\, y\ [\texttt{Loans}(x, y) \rightarrow \exists z\ \texttt{Members1}(y, z) \vee \texttt{Members2}(y, z)]\,.$$

$\triangleleft$

In the rest of the report, only TGDs and DTGDs will be considered.

Conjunctive query. From a given database, we might request for information using a *query*. PDQ only uses *conjuctive queries* of the form: $Q(\vec{x}) = \exists \vec{y} \bigwedge_{i=1}^{n} A_i$ where A_i are relational atoms, with arguments that are either variables from $\vec{x}$ or $\vec{y}$ or schema constants.

For a query Q, a *binding of* Q is a mapping from the free variables of Q to some values. For an instance I, an *homomorphism of* Q *in* 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.

Example 4. The query to get the name of all the members who borrowed the books titled Dumbo is:

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

It has only one homomorphism $h : x \mapsto$ `Michael` in the instance of the example 1. $\triangleleft$

1.2 Query containment and the chase

Query containment with constraints. 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 the following entailment holds:

$$Q \wedge \Sigma \models Q^*$$

.

In other words, we want to determine if, for every instance I that satisfies Σ, $[\![Q]\!]_I \subseteq [\![Q^*]\!]_I$.

The chase. When Q and Q^* are conjunctive queries and Σ is a conjunction of TGDs, a specialized method has been developed to prove these entailments, called *the chase*.

The chase is working as follows:

1. start with the assumption Q.
   More precisely, take as initial instance I_0 the *canonical database* $\texttt{CanonDB}(Q)$ of the conjunctive query Q: the instance whose elements are the schema constants of Q plus distinct elements c_i for each variable x_i in Q and which has 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.
   More precisely, from the instance I_n:
   (a) 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. This trigger is said *active* if there is no $\vec{f}$ such that $\rho(\vec{e}, \vec{f})$ holds.
   (b) *fire the rule*, i.e. add the facts to I_n that make $\rho(\vec{e}, \vec{f})$ true, where $\vec{f}$ is a tuple of new values distinct from any values in I_n and any schema constants. The new instance is denoted I_{n+1}.
3. stop and declare success if we have reached an instance I_p that has a "match" for Q^*.
   More precisely, a binding from each free variable x_i of Q^* to c_i is a *match* for Q^* in I_p if it is an homorphism of Q^* in I_p.

A *chase sequence following a set of TGDs* Σ is a sequence of instances where an instance is related to its successor by a rule firing of a constraint in Σ. We refer to the instances in the sequence as *chase configurations*.

In PDQ, several different chases are implemented and they only execute the step `2.`. They iteratively derive consequences from an input chase configuration with input constraints. This configuration is modified in place.

The algorithm 1 summarizes the *restricted chase* implemented in PDQ with TGDs. *Restricted* means that only active triggers are selected:

Algorithm 1: PDQ's restricted chase with TDGs

```
  input: A chase configuration config ;
         A collection of TGDs $\Sigma$ ;
1 while there is an active trigger for a TGD $\delta$ do
2     Choose such a trigger ;
3     Fire the rule ;
```

2 The global working of PDQ

In this part, I will present the notions of PDQ required to understand my work presented in the last part.

2.1 Access methods and plans

Let's give a definition of PDQ's inputs and output ([2]).

Access methods. The first input is an *access schema* that consists of:

- a schema. Originally, the integrity constraints used in PDQ were TGDs.
- for each relation R, a collection of *access methods*. Each access method is associated with a collection of *positions* of R, i.e. a collection of numbers between 1 and arity(R). These positions are called the *input positions*. Moreover, each method has an *access cost*.

An *access* consists of an access method of the schema and a method binding. If `mt` is an access method on relation R with arity n, I is an instance for a schema that includes R, and `AccBind` is a method binding on `mt`, then the *output of the access (`mt`, `AccBind`) on* I is the set of n-tuples $\vec{t}$ such that $R(\vec{t}) \in I$ and $\vec{t}$ restricted to the input positions of `mt` is equal to `AccBind`.

The second and last input of PDQ is a conjunctive query Q written over the input schema.

Plans. When it is possible, PDQ outputs a plan that answers the input query Q.

Informally, a *plan* is a reformulation of the input query in a target language that represents the kind of restricted computation done over an interface given by the input access schema. More precisely, it is a sequence of access and middleware query commands, ending with at most one return command. A formal definition of a plan that would involve defining relational algebra is not necessary to understand my work.

The *output of a plan on* I is a collection of tuples. The plan returned by PDQ *answers* the input query Q 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.

2.2 Finding plans using the chase

Now let's present the strategy developed in "Generating Plans from Proofs" for finding plans. The idea is to:

1. reformulate the problem of finding plans into a problem of query containment with TGD constraints;
2. solve this query containment problem using the chase.

Query containment reformulation. To find this query containment reformulation, we need to recall:

- how a plan works. For a query Q, an access schema `Sch` and an instance I for `Sch`, a plan first exposes facts of I using the access methods and then returns the output of Q over this "exposed instance".
- what kind of plan we are looking for. PDQ outputs a plan that answers Q, i.e. such that, for all instance I that satisfies `Sch`, the output of Q over I is equal to the output of the plan and, therefore, is equal to the output of Q over the "exposed instance".

From the first recall, we can deduce what will be the constraints of the query containment problem. Recall that an access schema consists of a schema and access methods. The idea is to transform the access methods into constraints. This leads to transform the access schema `Sch` into a *forward accessible schema for `Sch`*, denoted `AcSch(Sch)`, defined as follows ([3]):

- 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 of `Sch` called `InfAcc`R (the "inferred accessible version of R").
  - a unary relation `accessible`(x) ("x is an accessible value"). We have `accessible`(c) for each constant c of `Sch`.
- 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 $\texttt{InfAcc}R$.
– *accessibility axioms*: for each access method `mt` on relation R of arity n with input positions j_1, ..., j_m, we have the rule:

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

Informally, the accessibility axioms of `AcSch(Sch)` are what replace the access methods of `Sch`, the `InfAccCopy` relations and `InfAccCopy` constraints can be seen as the "exposed schema" of the "exposed instance", and finally, $\texttt{accessible}(c)$ indicates that the value c can be returned by some sequence of accesses.

From the second recall, we can now deduce the query containment reformulation. For an input access schema `Sch` and an input query Q, plans will be generated from proofs of this entailment:

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

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

PDQ's plan search. In PDQ, several different plan search programs are implemented. Their strategy is to explore the space of proofs of the entailment $Q \wedge \Gamma \models \texttt{InfAcc}Q$ ([4]). This space of proofs is represented by a tree of chase configurations. A configuration B is a child of a configuration A if B is related to A by a rule firing of an accessibility axiom. Each configuration maps to a plan and a status that is either `opened` or `closed`.

Depending on the programs, the space of proofs is more or less explored. The naive program explores the full space of proofs whereas other programs use optimizations to shortcut some explorations.

The naive plan search program is summarized in the algorithm 2. To make it easier to write, a breadth-search first of the tree configurations is used in this algorithm to explore them whereas a depth-search first is used in PDQ.

In the rest of the report, we refer to line 14 as *saturation* and to line 15 as *success checking.*

3 My work on PDQ

My intership's purpose was to add support for DTGDs in PDQ's plan search programs, i.e. to allow DTGDs in the integrity constraints of the input access schema of the plan search programs. Recall that a DTGD is an integrity constraint of the form:

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

What are the consequences on the implementation of these programs? Let's take the algorithm 2 as an example. Allowing DTGDs in the access schema impacts the chase call in line 2 and, by construction of the forward accessible schema, the chase call in line 14. Therefore, the consequences of allowing DTGDs in PDQ's plan search programs depend on the consequences of allowing DTGDs in PDQ's chases.

Hence, the first step of my work was to modify PDQ's chases to handle DTDGs. Then, I modified PDQ's plan search programs to handle DTGDs. Finally, I had time to integrate an external alternative tool to PDQ's chases, called DLV.

3.1 Chases with DTGDs

Before starting modifying the code of PDQ, I first quickly read a book to master Java, the language used. Then, after Dr. Benedikt's quick presentation of all the projects in PDQ, I read the code of the project containing the chases in order to get used to its architecture and have a global vision. Finally, I wrote down my plan to add support for DTGDs in the various chases and showed it to Dr. Benedikt who validated it.

Algorithm 2: PDQ's naive plan search with TGDs

```
rootConfig := Canonical database of $Q$ ;
Chase rootConfig with original integrity constraints ;
rootConfig's plan := $\emptyset$ ;
rootConfig's status := opened ;
tree := Tree with rootConfig as root ;
bestPlan := null ;
while there is an opened configuration config in tree do
    Select such a configuration ;
    config's status := closed ;
    foreach active trigger for an accessibility axiom $\delta$ associated with method mt do
        newConfig := a copy of config ;
        Chase newConfig with $\delta$ ;
        newConfig's plan := config's plan with method mt appended ;
        Chase newConfig with InfAccCopy integrity constraints ;
        if InfAccQ has a match in newConfig then
            newConfig's status := closed ;
            if bestPlan cost is higher than newConfig's plan cost then
                bestPlan := newConfig's plan ;
        else
            newConfig's status := opened ;
        Add newConfig as a child of config in tree ;
return bestPlan ;
```

Then, I began working on PDQ. The very first step was to define DTGDs in PDQ's first-order logic. This consisted in creating a class for this integrity constraint. At this point, I was able to start adding support for DTGDs, firstly, in the restricted chase and, secondly, in the other chases.

The restricted chase with DTGDs. What differs between a chase with TGDs and a chase with DTGDs is how these rules are fired.

Given an instance I, for a TGD:

$$\forall \vec{x}\ [\varphi(\vec{x}) \rightarrow \exists \vec{y}\ \rho(\vec{x}, \vec{y})]$$

where $\rho(\vec{x}, \vec{y}) = \bigwedge_{i=1}^{n} A_i(\vec{x}, \vec{y})$, and for a trigger $\vec{e}$, firing the rule would add $\{A_1(\vec{e}, \vec{f}), ..., A_n(\vec{e}, \vec{f})\}$ to I, with $\vec{f}$ a tuple of fresh values.

For a DTGD:

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

where $\rho_i = \bigwedge_{j=1}^{n} A_{i,j}(\vec{x}, \vec{y})$, and for a trigger $\vec{e}$, firing the rule would add as few consequences as possible, i.e. would add 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. At the end, firing the rule won't give us a unique instance but a collection of instances, the number of instances being the number of disjuncts in the head of the DTGD.

Hence, a *chase sequence following a set of DTGDs* Σ is a sequence of collections of chase configurations where a collection A is related to its successor by a rule firing of a constraint in Σ on one of the chase configurations of A. We refer to the collections in the sequence as *disjunctive chase configurations.*

This difference between the rule firing of a TGD and a DTGD implies a modification of the restricted chase that was summarized in algorithm 1. The input configuration is not a chase configuration anymore; it is a disjunctive chase configuration. The algorithm 3 sums up the restricted chase with DTGDs. In this algorithm, *splitting a DTGD* λ into TGDs means transforming λ into a collection of TGDs δ_i where the body of δ_i is the body of λ and its head is one of the disjuncts of the head of λ.

Algorithm 3: PDQ's restricted chase with DTGDs

```
input: A disjunctive chase configuration disConfig ;
       A collection of DTGDs $\Sigma$ ;
1 while there is an active trigger for a DTGD $\lambda$ on a configuration config of disConfig do
2     Choose such a trigger ;
3     Remove config from disConfig ;
4     foreach TGD $\delta$ of the splitted DTGD $\lambda$ do
5         newConfig := a copy of config ;
6         Fire the rule of $\delta$ on newConfig ;
7         Add newConfig to disConfig ;
```

Example 5. The algorithm 3 called with:

- the disjunctive configuration `disConfig` $= \{C_0\}$ where $C_0 = \{R(c)\}$;
- the DTGDs $\Sigma = \{\lambda_1, \lambda_2\}$ where $\lambda_1 = \forall x\ R(x) \to S(x) \lor T(x)$ and $\lambda_2 = \forall x\ S(x) \to T(x) \lor U(x)$.

will do the following:

1. As (c) is an active trigger for λ_1 on C_0, it will split λ_1 into $\{\forall x\ R(x) \to S(x), \forall x\ R(x) \to T(x)\}$ and then replace C_0 in `disConfig` by the possible configurations $C_1 = \{R(c), S(c)\}$ and $C_2 = \{R(c), T(c)\}$. At this point, `disConfig` $= \{C_1, C_2\}$.
2. As (c) is an active trigger for λ_2 on C_1, it will split λ_2 into $\{\forall x\ S(x) \to T(x), \forall x\ S(x) \to U(x)\}$ and then replace C_1 in `disConfig` by the possible configurations $C_3 = \{R(c), S(c), T(c)\}$ and $C_4 = \{R(c), S(c), U(c)\}$. At this point, `disConfig` $= \{C_2, C_4, C_5\}$.
3. As there is no other active trigger for a configuration of `disConfig`, the algorithm returns the disjunctive configuration `disConfig` $= \{\{R(c), T(c)\}, \{R(c), S(c), T(c)\}, \{R(c), S(c), U(c)\}\}$.

$\triangleleft$

In the TGD version of the algorithm, only a single chase configuration was stored in a database whereas, in this DTGD version, we need to store a collection of chase configurations. How to handle this problem? A solution could be to create as many databases as chase configurations, but it is not very convenient. The chosen solution consisted in adding an extra attribute to all the relations, that we called `InstanceID`, to identify the configuration containing the facts.

Once I changed the restricted chase and the way chase configurations were stored in the database, I created unit tests to test my code. Immediately, I discovered several bugs: some coming from my code and some coming from the rest of the code. For those coming from my code, I could quickly fix them by printing messages in console. For those coming from the rest of the code, it was much more difficult to find them and to fix them as I didn't know much about the other projects.

For example, before I added the `InstanceID` attribute, an other extra attribute, called `FactID`, was present at the end of all the relations. Several portions of code were referring to this `FactID` by position in the relation (-1 at this moment) rather than by name. Therefore, when I added the `InstanceID` attribute at the end of all the relations, these portions of code were now referring to the `InstanceID` instead of the `FactID`, revealing unexpected behaviors.

For this kind of bug, it was impossible to debug by printing messages in console as it was really hard to find their origin in a code I didn't know much. Hence, George Konstantinidis, a postdoc working with Dr. Benedikt, taught me how to use the powerful debugging tool of Eclipse, a Java IDE, and also taught me some shortcuts to easily move in the different classes. It took me time to fix all the bugs, but, at the end, DTGD support was added in the restricted chase.

All the chases with DTGDs. After adding support for DTGDs in the restricted chase, I had to add support for them in the other chases. A solution is to add support in each chase manually, duplicating code over the different chases. An other solution is to gather what is common to all the chases (including DTGD support) in one file and let the strategy of each chase in its dedicated file.

I chose the second solution. More precisely, this implied a modification of the class architecture of the chases. I created an abstract class called `Chaser` that would contain the common part and each class dedicated to a chase would inherit this abstract class and define a `reason` method, containing the chase strategy.

To test this new architecture, I created other unit tests, discovered other bugs and took time to fix them. At the end, DTGD support was added in all the chases.

3.2 Plan search programs with DTGDs

Before starting to add support for DTGDs in the plan search programs, I read the code of the project containing these programs in order to get used to its architecture and have a global vision. I also had to fix several bugs related to the new `InstanceID` extra attribute which took me some time. I also allowed to manipulate DTGDs in these programs by:

- defining the inferred accessible DTGDs.
- modifying the "parser" of the `schema.xml` file, containing the input access schema, to parse DTGD integrity constraints.

At this point, I was able to start adding support for DTGDs, firstly, in the naive plan search program and, secondly, in the other plan search programs.

The naive plan search program with DTGDs. The DTGD version of the chase now takes as parameter a disjunctive chase configuration instead of a chase configuration as it was the case in the TGD version. This impacts the way to use chases and, in particular, this impacts the algorithm 2. The chase is used twice in this algorithm:

- in line 2, the canonical database is chased with original integrity constraints. We refer to the result of this chase call as *top-level disjunction*.
- in line 14, the configuration obtained after firing a forward accessibility rule is chased with `InfAccCopy` integrity constraints. We refer to the result of this chase call as *follow-up disjunction*.

When I started to modify the naive plan search program to add support for DTGDs, I initially thought that top-level disjunction and follow-up disjunctions could be treated the same. This led me to a really complicated structure which took me a lot of time to implement. Finally, I showed it to Dr. Benedikt who said it made no sense and I realized I had a misunderstanding: the top-level disjunction and follow-up disjunctions should be treated differently because they don't have the same role.

Firstly, the top-level disjunction gives all the possible "hidden" chase configurations. Recall that we want to prove the entailment $Q \wedge \Gamma \models \texttt{InfAcc}Q$ for all instance. Therefore, for each "hidden" configuration `config`, we must find a plan that answers Q on `config`. Hence, the plan returned by PDQ will be the union of these plans. In practice, we associate to each "hidden" configuration `config` a tree of disjunctive chase configurations with the disjunctive configuration $\{\texttt{config}\}$ as root and then find the best plan for each tree.

Secondly, a follow-up disjunction gives the disjunctive chase configuration after firing the `InfAccCopy` integrity constraints in order to check if this configuration matches $\texttt{InfAcc}Q$. For a disjunctive chase configuration `disConfig`:

- firing a rule means firing the rule on all the configurations of `disConfig`.
- a query Q has a match if Q has a match in each configuration of `disConfig`.

The algorithm 4 summarizes the DTGD version of the naive plan search program:

Example 6. Let's say the algorithm 4 is called with:

- 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 $Q(x) = R(x)$.

Algorithm 4: PDQ's naive plan search with DTGDs

```
config := Canonical database of $Q$ ;
disConfig := config ;
Chase disConfig with original integrity constraints ;
disjunctiveTree := $\emptyset$ ;
foreach config of disConfig do
    disRootConfig := config ;
    disRootConfig's plan := $\emptyset$ ;
    disRootConfig's status := opened ;
    tree := Tree with disRootConfig as root ;
    tree's best plan := null ;
    Add tree to disjunctiveTree ;
while there is an opened disjunctive configuration disConfig in a tree of disjunctiveTree do
    Select such a disjunctive configuration ;
    disConfig's status := closed ;
    foreach active trigger for an accessibility axiom $\delta$ associated to method mt do
        newDisConfig := a copy of disConfig ;
        Chase newDisConfig with $\delta$ ;
        newDisConfig's plan := disConfig's plan with the method mt appended ;
        Chase newDisConfig with InfAccCopy integrity constraints ;
        if InfAccQ has a match in newDisConfig then
            newDisConfig's status := closed ;
            if tree's best plan cost is higher than newDisConfig's plan cost then
                tree's best plan := newDisConfig's plan ;
        else
            newDisConfig's status := opened ;
        Add newDisConfig as a child of disConfig in tree ;
if each tree has a best plan then
    return the union of the best plan of each tree in disjunctiveTree ;
```

The integrity constraints can be summed up in this schema:

$$\begin{tikzcd}
& R \arrow[dl, bend right] \arrow[dr, bend left] & \\
A1 \arrow[ur, bend right] \arrow[dr, bend left] & & A2 \arrow[ul, bend left] \arrow[dl, bend right] \\
& T \arrow[uu] &
\end{tikzcd}$$

Then, the algorithm will do the following:

1. Start with the canonical database of Q that is $C_0 = \{R(c)\}$, then chase `disConfig` $= \{C_0\}$ with the original integrity constraints Γ. At this point, `disConfig` $= \{C_1, C_2\}$ where $C_1 = \{R(c), A1(c)\}$ and $C_2 = \{R(c), A2(c)\}$.
2. Create trees t_1 with $\{C_1\}$ as root and t_2 with $\{C_2\}$ as root. These configurations are `opened` for the moment.
3. Search the best plan for each tree.
   Firstly, for t_1:

(a) Select the root disjunctive configuration `disConfig` $= \{C_1\}$, the only one `opened`.

(b) As there is an active trigger (c) for the accessibility axiom $\delta = \forall x \; A1(x) \rightarrow \texttt{InfAcc}A1(x)$ on C_1:

- create a `newDisConfig` disjunctive configuration that is a copy of `disConfig`.
- chase `newDisConfig` with δ i.e. add $\texttt{InfAcc}A1(c)$ to all the configurations of `newDisConfig`. This configuration now maps to a plan of cost 10.
- chase `newDisConfig` with the `InfAccCopy` integrity constraints. At this point:

$$\begin{aligned}\texttt{newDisConfig} = \{&\{R(c), A1(c), \texttt{InfAcc}A1(c), \texttt{InfAcc}R(c)\},\\ &\{R(c), A1(c), \texttt{InfAcc}A1(c), \texttt{InfAcc}T(c), \texttt{InfAcc}R(c)\}\}.\end{aligned}$$

- as $\texttt{InfAcc}Q(x) = \texttt{InfAcc}R(x)$ has a match in `newDisConfig`, t_1 has a new best plan that is `newDisConfig`'s best plan.
- add `newDisConfig` to t_1 with a `closed` status.

(c) As there is no other `opened` disjunctive configuration, stop the exploration in t_1. t_1's best plan has a cost of 10.

Secondly, for t_2:

(a) Select the root disjunctive configuration `disConfig` $= \{C_2\}$, the only one `opened`.

(b) As there is an active trigger (c) for the accessibility axiom $\delta = \forall x \; A2(x) \rightarrow \texttt{InfAcc}A2(x)$ on C_2:

- create a `newDisConfig` disjunctive configuration that is a copy of `disConfig`.
- chase `newDisConfig` with δ i.e. add $\texttt{InfAcc}A2(c)$ to all the configurations of `newDisConfig`. This configuration now maps to a plan of cost 20.
- chase `newDisConfig` with the `InfAccCopy` integrity constraints. At this point:

$$\begin{aligned}\texttt{newDisConfig} = \{&\{R(c), A2(c), \texttt{InfAcc}A2(c), \texttt{InfAcc}R(c)\},\\ &\{R(c), A2(c), \texttt{InfAcc}A2(c), \texttt{InfAcc}T(c), \texttt{InfAcc}R(c)\}\}.\end{aligned}$$

- as $\texttt{InfAcc}Q(x) = \texttt{InfAcc}R(x)$ has a match in `newDisConfig`, t_2 has a new best plan that is `newDisConfig`'s best plan.
- add `newDisConfig` to t_2 with a `closed` status.

(c) As there is no other `opened` disjunctive configuration, stop the exploration in t_2. t_2's best plan has a cost of 20.

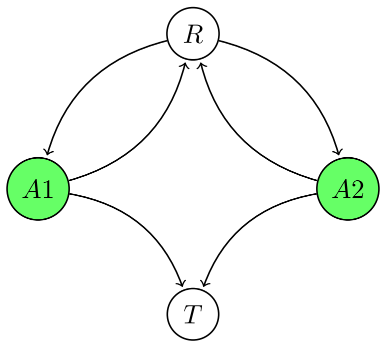
4. As t_1 and t_2 map to a best plan, return the global best plan that has a cost of 30.

$\triangleleft$

Example 7. Let's say the algorithm 4 is called with slightly different parameters than those of example 6:

- an access schema composed of:
  - the same relations.
  - the same integrity constraints except that λ_4 is omitted.
- the same query.

The integrity constraints can be summed up in this schema:



The algorithm will behave slightly differently:

- in the tree t_1 of example 6, the root configuration was chased with an accessibility axiom and then chased with the `InfAccCopy` integrity constraints. At the end, we got this disjunctive configuration:

$$\{\{R(c), A1(c), \texttt{InfAcc}A1(c), \texttt{InfAcc}R(c)\}, \{R(c), A1(c), \texttt{InfAcc}A1(c), \texttt{InfAcc}T(c), \texttt{InfAcc}R(c)\}\}.$$

which matched $\texttt{InfAcc}Q$. This time, with these new parameters, we get this one:

$$\{\{R(c), A1(c), \texttt{InfAcc}A1(c), \texttt{InfAcc}R(c)\}, \{R(c), A1(c), \texttt{InfAcc}A1(c), \texttt{InfAcc}T(c)\}\}.$$

which doesn't match $\texttt{InfAcc}Q$ anymore. Therefore, t_1 doesn't have a best plan anymore.
- the same thing happens in the tree t_2.

At the end, no plan will be returned by the algorithm. ◁

Finally, I created several tests for it.

All the plan search programs with DTGDs. After adding support for DTGDs in PDQ's naive plan search program, I had to add support for them in the other plan search programs. These programs use some optimizations based on the notions of *dominance* and *equivalence*. Given C and C' chase configurations:
- C *is dominated by* C' if the inferred accessible facts of C are included in the inferred accessible facts of C'. Hence, if during the plan search, the plan cost of C is higher than the plan cost of C' and C is dominated by C', we can stop the exploration from the configuration C.
- C *is equivalent to* C' if C is dominated by C' and C' is dominated by C, i.e. if the inferred accessible facts of C are equal to the inferred accessible facts of C'.

These definitions don't hold for disjunctive chase configuration. Therefore, I had to give new ones. Given $C = \{C_1, ..., C_n\}$ and $C' = \{C'_1, ..., C'_m\}$ disjunctive chase configurations, where C_1, ..., C_n, C'_1, ..., C'_m are chase configurations:
- C *is dominated by* C' if $\forall i \in [\![1, n]\!]$, $\exists j \in [\![1, m]\!]$ such that C_i is dominated by C'_j.
- C *is equivalent to* C' if C is dominated by C' and C' is dominated by C.

After giving a new definition of these notions, I implemented them in PDQ and then tested them. At the end, all the plan search programs were supporting DTGDs.

3.3 Integration of DLV as an alternative for saturation and success checking

To end my internship, Dr. Benedikt suggested that I integrate DLV ([5]) in PDQ as an alternative for saturation, i.e. adding the `InfAccCopy` consequences, and success checking, i.e. checking if $\texttt{InfAcc}Q$ has a match. DLV is a deductive database system based on disjunctive logic programming. In particular, it natively implements a chase with disjunctive rules.

DLV basics. The DLV's user manual first describes its core language and then presents its two main modes:
- a saturation mode accessible with a command line of this kind: `DLV -silent {facts} {rules}` where `facts` corresponds to the name of the file containing the DLV facts and `rules` to the file containing the DLV rules. This outputs a collection of DLV models, i.e. a collection of collection of DLV facts.
- a success checking mode accessible with a command line of this kind: `DLV -silent -cautious {facts} {rules} {query}` where `{query}` corresponds to the name of the file containing the DLV query. With `-cautious` specified, this outputs all the bindings that witness that the query holds in all the models.

DLV integration in PDQ. The integration of DLV in PDQ consisted in implementing a DLV chase configuration, i.e. an abstraction of a collection of DLV models. This configuration would have the same interface as a disjunctive chase configuration, giving us the choice between using a DLV or a disjunctive chase configuration.

To respect this interface, I had to use:
- DLV saturation. First, I wrote PDQ's facts and integrity constraints in files, respecting the DLV language. For this, I used the DLV printer coded by Efthymia Tsamoura, who worked with Dr. Benedikt. Then, I called DLV with these files as input. Finally, I read the output models and parsed them using a DLV reader also coded by Dr. Tsamoura.

- DLV success checking. First, I wrote PDQ's facts and query in files, then called DLV and, finally, read the output bindings. I just coded a simple binding parser.

In order to make DLV manipulation easier, I also coded some classes to abstract the DLV binary and its input files.

Conclusion

I chose this internship in order to discover the field of databases. I started by reading Dr. Benedikt's notes and the book "Generating Plans from Proofs" that introduced me to the database subject, some theorems and problems of interest. I continued to learn but also practice by working on PDQ. I also attended various conferences organized by the departement of computer science of the University of Oxford where researchers presented their current work and results.

Working on PDQ allowed me to continue to develop my coding skills: I learned a new language (Java) and discovered a new IDE (Eclipse) and its powerful debugging tool. I also worked for the first time on such a huge project that required me to deal with code written by others. Therefore, I needed to version my code and had to intensively use Git.

Moreover, this internship was also a good opportunity to begin to enter the world of research. Working on a research project during eight weeks and talking every day with Dr. Benedikt, the postdocs and the PhD students about research or the researcher life helped me to get a perspective on research work.

Finally, I want to thank Dr. Benedikt for welcoming me. He presented me the building and always shared the link to the conferences. I also received advice on what I should visit in the town, what are the most beautiful colleges... Dr. Benedikt and his team always took time to answer to my questions and help me. It is certain that I will have some very good memories from this internship!

References

[1] Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamoura. Generating plans from proofs. chapter 1. Morgan & Claypool, 2016.

[2] Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamoura. Generating plans from proofs. pages 79–83. Morgan & Claypool, 2016.

[3] Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamoura. Generating plans from proofs. pages 96–97. Morgan & Claypool, 2016.

[4] Michael Benedikt, Julien Leblay, Balder ten Cate, and Efthymia Tsamoura. Generating plans from proofs. pages 154–157. Morgan & Claypool, 2016.

[5] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dlv system for knowledge representation and reasoning. 2002.