$\boxed{\textit{Technical Note: BIRN-DI-TN-2003-01}}$

Query Containment, Minimization, and Semantic Optimization of Conjunctive Queries (or: *More on Uncles and Aunts*)

Bertram Ludäscher

February 2003 (major revision: August 2003)

Abstract

This technical note presents the ubiquitous problem of *query containment* for conjunctive queries (an NP-complete problem), and an elegant implementation CQCP of ***C***onjunctive ***Q***uery ***C***ontainment in ***P***rolog, in just 7 lines of code. We also describe two important applications of conjunctive query containment: *semantic query optimization* and *minimization.* The latter can be implemented by another concise Prolog algorithm, requiring additional 7 lines of code ...

1 Introduction

In BIRN-DI-TN-2002-01 we have given an overview of the overall query planning problem of the mediator, and presented a technique for unfolding (non-recursive) mediated views. In BIRN-DI-TN-2002-02 we then have focused on the problem of *ordering conjunctive queries w.r.t. binding patterns restrictions* to form *feasible plans*, and presented a straightforward implementation in Prolog that computes all maximal feasible solutions. A fundamental problem at the core of most logic-based query optimization techniques is *query containment*, which is the focus of this technical note:

(Conjunctive) Query Containment. A query Q_1 is contained in Q_2, denoted $Q_1 \sqsubseteq Q_2$, if for all possible database instances D, the set of answers to Q_1, is contained in the set of answers to Q_2. For a database instance D and a k-ary query Q, the set of answers to Q is $\{\bar{a} \mid D \models Q(\bar{a})\}$, where $\bar{x}$ is a vector of constants or domain values $a_1, \ldots, a_k$.[1]

Q_1 and Q_2 are called *equivalent* (denoted $Q_1 \equiv Q_2$), if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$. Query containment is undecidable for many languages, in particular for first-order queries. For conjunctive queries (select-project-join queries), however, the problem is decidable and NP-complete (in the size of the query) [CM77]. Since query sizes tend to be "small" (in particular, when compared to database sizes), query containment is still of use in practice – indeed, it is one of the most fundamental tools for logic-based query optimization.

A generalization of the problem considers in addition to Q_1 and Q_2 a set of integrity constraints, and asks whether for all databases satisfying the constraints, each tuple in the result of Q_1 is contained in the result of Q_2. We will not consider this generalization here.

[1] In logic "$\models$" is the model relationship, and $D \models \varphi(\bar{a})$ indicates that the structure D is a *model* for the formula φ in which the free variables have been replaced by the constants $\bar{a}$.

2 Conjunctive Query Containment in Logic

Conjunctive query containment is usually formalized using the notion of *containment mappings*. Here, we choose another, slightly more general way, similar to the "canonical database approach" [Ull97] and based on logic equivalences only.

A *conjunctive query* Q is a formula of the form $(\exists \bar{y})\, \varphi(\bar{x}, \bar{y})$ where φ is a conjunction of atomic formulas $(p_1(\ldots) \wedge \cdots \wedge p_n(\ldots))$ and in which all free variables are among $\bar{x}, \bar{y}$. The free variables $\bar{x}$ of Q are called *distinguished variables*, while the $\exists$-bound variables of Q are called *existential* (or *non-distinguished*) *variables*. It is common to write conjunctive queries in rule form:

$$Q := \quad q(\bar{x}) \leftarrow \underbrace{p_1(\ldots),\ \ldots\ ,\ p_n(\ldots)}_{(\exists \bar{y})\, \varphi(\bar{x}, \bar{y})}$$

We call $q(\bar{x})$ the *head* of the query Q, while $p_1(\ldots), \ldots, p_n(\ldots)$ is called the *body* of Q. Note that in rule form, the non-distinguished variables $\bar{y}$ are implicitly $\exists$-quantified.

Consider two conjunctive queries $Q_\varphi := (\exists \bar{y})\, \varphi(\bar{x}, \bar{x}_1, \bar{y})$ and $Q_\psi := (\exists \bar{z})\, \psi(\bar{x}, \bar{x}_2, \bar{z})$. Here we assume that the conjunctive queries Q_φ and Q_ψ may have some distinguished variables in common (*i.e.*, $\bar{x}$) and some disjoint ones (*i.e.*, $\bar{x}_1$ and $\bar{x}_2$, respectively). Also, *w.l.o.g.*, they have disjoint existential variables $\bar{y}$ and $\bar{z}$, respectively (otherwise rename them apart). We want to know whether $Q_\varphi \to Q_\psi$ is *valid*, *i.e.*, true for any logic interpretation.[2] Thus, we are interested in the following *query implication problem*:

$$\models (\exists \bar{y})\, \varphi(\bar{x}, \bar{x}_1, \bar{y}) \to (\exists \bar{z})\, \psi(\bar{x}, \bar{x}_2, \bar{z}) \tag{1}$$

where φ and ψ are conjunctions of atomic formulas.[3]

Since $\bar{x}, \bar{x}_1, \bar{x}_2$ occur free in (1), this is equivalent to considering the $\forall$-closure:

$$\models (\forall \bar{x})(\forall \bar{x}_1)(\forall \bar{x}_2)\ (\exists \bar{y})\, \varphi(\bar{x}, \bar{x}_1, \bar{y}) \to (\exists \bar{z})\, \psi(\bar{x}, \bar{x}_2, \bar{z}) \tag{2}$$

Note that for $\bar{x}_1 = \bar{x}_2 = \emptyset$ we obtain exactly the containment problem $Q_\varphi \sqsubseteq Q_\psi$, *i.e.*, whether for all databases D, we have that $Q_\varphi(D) \subseteq Q_\psi(D)$, or more precisely, whether for all D:

$$\{\ \langle \bar{x} \rangle \mid D \models (\exists \bar{y})\, \varphi(\bar{x}, \bar{y})\ \} \subseteq \{\ \langle \bar{x} \rangle \mid D \models (\exists \bar{z})\, \psi(\bar{x}, \bar{z})\ \}$$

However, the above formulation also affords the case that that the two queries share no variables (*i.e.*, $\bar{x} = \emptyset$) and the case that they may have different arities:

Example 1 (Semantic Query Optimization via Denials) Consider a semantic *integrity constraint*, expressed in (Datalog) rule form as a *denial*, *i.e.*, a formula which must never be true for any database D:

$$\mathsf{false} \leftarrow \psi(\bar{z})$$

If we can show that the implication $(\exists \bar{y})\, \varphi(\bar{x}_1, \bar{y}) \to (\exists \bar{z})\, \psi(\bar{z})$ is valid, then we know that the first query can never be satisfied (otherwise the "denied situation" $\psi(\bar{z})$ would occur for some values for $\bar{z}$, contradicting the assumed integrity constraint). □

Reasoning with integrity constraints, as illustrated in the previous example, constitutes an important case where the slightly more general query implication problem is useful, since it does not require the queries to have the same arity and shared variables (unlike the classical definition which has this requirement).

---

[2] "*for every database instance*" in database terminology

[3] "*lists of positive goals/atoms*" in logic programming terminology; "*select-project-join queries*" in database terminology

Solving Query Implication via Query Evaluation

Coming back to the above query implication problem (2), we note that $\bar{x}_1$ ($\bar{x}_2$) does not occur in ψ (φ), so we can move those quantifiers "inside" (*i.e.*, "below" the implication symbol "$\rightarrow$"):

$$\models (\forall \bar{x})\ (\forall \bar{x}_1)(\exists \bar{y})\, \varphi(\bar{x}, \bar{x}_1, \bar{y}) \rightarrow (\forall \bar{x}_2)(\exists \bar{z})\, \psi(\bar{x}, \bar{x}_2, \bar{z}) \tag{3}$$

Note that a formula is valid iff its negation is *unsatisfiable*. We denote that a formula (or set of formulas) F is unsatisfiable by writing $F \models \Box$.[4] Thus (3) is equivalent to

$$\neg(\forall \bar{x})\ (\forall \bar{x}_1)(\exists \bar{y})\, \varphi(\bar{x}, \bar{x}_1, \bar{y}) \rightarrow (\forall \bar{x}_2)(\exists \bar{z})\, \psi(\bar{x}, \bar{x}_2, \bar{z}) \models \Box \tag{4}$$

Using elementary logic equivalences we obtain

$$(\exists \bar{x})\ (\exists \bar{x}_1)(\exists \bar{y})\, \varphi(\bar{x}, \bar{x}_1, \bar{y}) \wedge \neg(\forall \bar{x}_2)(\exists \bar{z})\, \psi(\bar{x}, \bar{x}_2, \bar{z}) \models \Box \tag{5}$$

As is well known (Theorem(s) of Herbrand-Löwenheim-Skolem), we can get a formula which is equivalent *w.r.t.* satisfiability (which is what we are interested in here), by replacing the outermost existential variables by Skolem functions (here: constants). This yields:

$$\varphi(\bar{a}, \bar{a}_1, \bar{b}) \wedge \neg(\forall \bar{x}_2)(\exists \bar{z})\, \psi(\bar{a}, \bar{x}_2, \bar{z}) \models \Box \tag{6}$$

φ asserts the existence of some $\bar{a}, \bar{a}_1, \bar{b}$ (for the $\exists$-quantified variables $\bar{x}, \bar{x}_1, \bar{y}$, respectively) for which φ is true (but adding the negation of ψ renders the whole formula unsatisfiable). Using simple rules for "$\models$" on closed formulas (essentially, "$\models$" behaves here like "$\rightarrow$"), we obtain:

$$\varphi(\bar{a}, \bar{a}_1, \bar{b}) \models (\forall \bar{x}_2)(\exists \bar{z})\, \psi(\bar{a}, \bar{x}_2, \bar{z}) \tag{7}$$

Therefore, our original query implication $Q_\varphi \rightarrow Q_\psi$ holds iff we can show that any model of $\varphi(\bar{a}, \bar{a}_1, \bar{b})$ is also a model of $(\forall \bar{x}_2)(\exists \bar{z})\, \psi(\bar{a}, \bar{x}_2, \bar{z})$.

So finally we can reduce the implication problem to a query evaluation problem: The conjunctive formula $\varphi(\bar{a}, \bar{a}_1, \bar{b})$ corresponds to what is known as the "canonical database" $D_\varphi(\bar{a}, \bar{a}_1, \bar{b})$ of Q_φ with the "frozen variables" $\bar{a}, \bar{a}_1, \bar{b}$. This canonical database is a specific (Herbrand) interpretation, and we now only have to evaluate the query Q_ψ over this database:

$$\{\ \langle \bar{a}\, \bar{x}_2 \rangle \mid D_{\varphi(\bar{a}, \bar{a}_1, \bar{b})} \models (\exists \bar{z})\, \psi(\bar{a}, \bar{x}_2, \bar{z})\ \} \tag{8}$$

The original implication $Q_\varphi \rightarrow Q_\psi$ holds iff (8) is not empty.[5]

Putting on those Logic Programming Glasses

Note that $D_{\varphi(\bar{a}, \bar{a}_1, \bar{b})}$ can also be viewed as a logic program comprising only facts. In order to decide our original problem $Q_\varphi \rightarrow Q_\psi$, we can simply evaluate the conjunctive query $(\exists z)\psi(\bar{a}, \bar{x}_2, \bar{z})$ over $D_{\varphi(\bar{a}, \bar{a}_1, \bar{b})}$. The original implication holds, iff the answer is not empty.

From a result in logic programming, the completeness theorem for SLD resolution, we know that $P \models \forall Q\Theta$ (*i.e.*, the universal closure of Q is true in all models of P) iff $P \vdash_{SLD} Q\tau$ where τ subsumes Θ [BD01].[6] This means that we can solve our original problem by asserting the canonical database $D_{\varphi(\bar{a}, \bar{a}_1, \bar{b})}$ of Q_φ as the program P and running Q_ψ as a query over that database. If SLD resolution finds an answer $Q_\psi\tau$ then the original implication holds. Moreover, from the answer substitution(s) τ we can obtain the containment mapping(s) as described in [Ull97].

[4] This notion is borrowed from automated deduction: "$\Box$" denotes the *empty clause*, and corresponds to false.

[5] ***2do*** Well, certainly for $\bar{x}_2 = \emptyset$; otherwise I'm a bit puzzled what to do with those $\bar{x}_2$.

[6] *i.e.*, there is some σ such that $Q\Theta = Q\tau\sigma$

3 Conjunctive Query Containment in 7 Lines of Prolog

While all of the above may sound quite complicated, implementing conjunctive query containment in Prolog is actually surprisingly simple and elegant, and seven lines of code will do (Figure 1).

Before we explain this implementation, let us consider two conjunctive queries $Q_1 := (\exists\bar{y})\varphi(\bar{x}, \bar{y})$ and $Q_2 := (\exists\bar{z})\psi(\bar{x}, \bar{z})$. We are interested in the implication $(\forall\bar{x})\ Q_1(\bar{x}, \bar{y}) \to Q_2(\bar{x}, \bar{z})$, *i.e.*, we consider here the traditional variant of query containment $Q_1 \sqsubseteq Q_2$ in which both queries share the same distinguished variables $\bar{x}$ (so $\bar{x}_1 = \bar{x}_2 = \emptyset$). We represent such conjunctive queries in Prolog as statements of the form:

$$\texttt{Qid} :: \ \ \texttt{q}(\bar{\texttt{X}}) \leftarrow [\ \varphi(\bar{\texttt{X}}, \bar{\texttt{Y}})\ ]\ .$$

Here `Qid` is a unique *query identifier*, the head $\texttt{q}(\bar{\texttt{X}})$ of the query contains the *distinguished variables* $\bar{x}$ (albeit in uppercase $\bar{\texttt{X}}$, due to the Prolog variable notation), and the body contains the *actual query expression* $(\exists\bar{\texttt{Y}})\varphi(\bar{\texttt{X}}, \bar{\texttt{Y}})$ as a Prolog list of atoms (note that the variables $\bar{\texttt{Y}}$ occurring in the body only are implicitly $\exists$-quantified in Prolog).

Example 2 (Prolog Syntax) Here are two conjunctive queries in Prolog syntax:

$$\texttt{q}_1 :: \ \texttt{q}(\texttt{X}_1, \texttt{X}_2, \texttt{X}_3) \leftarrow [\ \texttt{p}(\texttt{Y}_1, \texttt{Y}_2, \texttt{X}_3), \texttt{p}(\texttt{X}_1, \texttt{Y}_2, \texttt{Y}_3), \texttt{p}(\texttt{Y}_4, \texttt{X}_2, \texttt{Y}_3), \texttt{p}(\texttt{X}_1, \texttt{Y}_5, \texttt{Y}_6), \texttt{p}(\texttt{Y}_1, \texttt{Y}_5, \texttt{X}_3)\ ]\ .$$
$$\texttt{q}_2 :: \ \texttt{q}(\texttt{X}_1, \texttt{X}_2, \texttt{X}_3) \leftarrow [\ \texttt{p}(\texttt{Z}_1, \texttt{Z}_2, \texttt{X}_3), \texttt{p}(\texttt{X}_1, \texttt{Z}_2, \texttt{Z}_3), \texttt{p}(\texttt{Z}_4, \texttt{X}_2, \texttt{Z}_3)\ ]\ .$$

Exercise: Which of $\texttt{q}_1 \to \texttt{q}_2$ and $\texttt{q}_2 \to \texttt{a}_1$ hold? □

Note that when representing conjunctive queries in this way, *i.e.*, as Prolog statements of the form $\texttt{Qid} :: \ \texttt{q}(\bar{\texttt{X}}) \leftarrow [\varphi(\bar{\texttt{X}}, \bar{\texttt{Y}})]$ the Prolog system will consider all distinguished variables from different statements to be different (although we may choose the same name for them). This is due to the fact, that in each Prolog rule (just as in logic) we can rename variables and still obtain a logically equivalent statement. For example, the two Prolog rules

$$\texttt{p}(\texttt{X}) \leftarrow \texttt{q}(\texttt{X}, \texttt{Y}).$$
$$\texttt{p}(\texttt{U}) \leftarrow \texttt{q}(\texttt{U}, \texttt{V}).$$

are really equivalent (*e.g.*, we can apply the renaming $\{\texttt{U} \mapsto \texttt{X}, \texttt{V} \mapsto \texttt{Y}\}$ to obtain the former from the latter). Both Prolog rules are equivalent to the same logic formula:

$$(\forall x)\ \ \texttt{p}(x) \leftarrow (\exists y)\, \texttt{q}(x, y)$$

Prolog Operator Declarations. In Example 2, we have made use of two operators "`::`" and "$\leftarrow$".[7] We could have avoided this by writing, *e.g.*,

$$\texttt{query(}\ \texttt{Qid},\ \texttt{q}(\bar{\texttt{X}}),\ [\ \varphi(\bar{\texttt{X}}, \bar{\texttt{Y}})\ ]\ ).$$

instead of

$$\texttt{Qid} :: \ \ \texttt{q}(\bar{\texttt{X}}) \leftarrow [\ \varphi(\bar{\texttt{X}}, \bar{\texttt{Y}})\ ]\ .$$

However the query statements become much more readable using the operator syntax. It is very easy to make the system understand this operator syntax by using declarations of the form:

```
:- op(600, xfx, ::).
:- op(500, xfx, ←).
```

The first operator declaration declares "`::`" to be a binary operator with precedence `600`.[8] The second declaration defines that "$\leftarrow$" is also a binary operator, however, it binds *stronger* than

[7] "$\leftarrow$" is typed in as `<-` on the keyboard

[8] The `xfx` says that it does not automatically associate to either the left or the right (in contrast to `xfy` and `yfx`), so if we were to use `X::Y::Z` the system would complain, and the user has to disambiguate this as either `(X::Y)::Z` or `X::(Y::Z)`.

"`::`" due to its lower operator precedence number (500). Thus, the statement $\mathtt{X} :: \mathtt{Y} \leftarrow \mathtt{Z}$ will be automatically parsed as $\mathtt{X} :: (\mathtt{Y} \leftarrow \mathtt{Z})$.

We are now getting ready to explain the implementation in Figure 1. Assume that some query statements of the form "$\mathtt{Qid} ::\ \mathtt{q}(\bar{\mathtt{X}}) \leftarrow [\ \varphi(\bar{\mathtt{X}}, \bar{\mathtt{Y}})\ ].$" have been loaded, together with the 7-line program CQCP in Figure 1. We can test whether any two queries Q_A and Q_B are contained in one another by asking the Prolog system:

```
?- QidA :: QA, QidB :: QB, QidA \= QidB, contained(QA, QB).
```

This will bind `QA` and `QB` to two *different* queries (since we required that $\mathtt{QidA} \neq \mathtt{QidB}$, denoted `QidA \= QidB` on the keyboard), and then execute the containment test `contained(QA,QB)`. If it succeeds, the system will print the instantiations of `QA` and `QB` for which the containment holds; otherwise the system will answer "`No`".

Example 3 (Containment Test) Assume the two query statements $\mathtt{q}_1$ and $\mathtt{q}_2$ from Example 2 have been loaded, together with the program in Figure 1. We can test whether $q_1 \rightarrow q_2$ and $q_2 \rightarrow q_1$ by asking the Prolog system:

```
?- QidA :: QA, QidB :: QB, QidA \= QidB, contained(QA, QB).
```

The system first answers as follows:

```
QidA = q1
QA   = q(a0, a1, a2)<-[p(a3, a4, a2), p(a0, a4, a5), p(a6, a1, a5), p(a0, a7, a8), p(a3, a7, a2)]
QidB = q2
QB   = q(a0, a1, a2)<-[p(a3, a4, a2), p(a0, a4, a5), p(a6, a1, a5)]
```

Thus, $\mathtt{q}_1 \rightarrow \mathtt{q}_2$ holds. Observe how the frozen variables $\mathtt{a1}, \mathtt{a2}, \ldots$, originally introduced for `QA`, also occur in the answer for `QB`. Indeed by binding the variables in `QB` to the constants in `QA`, the system makes sure that all atoms in `QB` occur in `QA`. If this can be achieved, then the containment holds, and the constants serve as witnesses for the containment mapping.

If after the first answer, the user hits "`;`" then the system tries to find another answer. In this case, it binds the queries `q1` and `q2` in the opposite order, and indeed that containment holds as well:

```
QidA = q2
QA   = q(a0, a1, a2)<-[p(a3, a4, a2), p(a0, a4, a5), p(a6, a1, a5)]
QidB = q1
QB   = q(a0, a1, a2)<-[p(a3, a4, a2), p(a0, a4, a5), p(a6, a1, a5), p(a0, a4, a5), p(a3, a4, a2)] ;

No
```

Hitting "`;`" one more time shows that there are no more answers (and the sytem replies "`No`"). Observe how all atoms of the longer query instance `q1` occur in the shorter `q2` instance. □

3.1 Algorithm CQCP

How does the algorithm in Figure 1 work? The head of the `contained/2` rule in line (1) guarantees that the distinguished variables (above: $\bar{x}$) of `Q1` and `Q2` are unified. Line (2) calls the built-in predicate `numbervars/4` which freezes all variables in `Q1` by numbering them (starting from 0). More precisely, every variable is replaced by a constant expression of the form $\mathtt{a}(i)$ where i ranges from 0 to $n-1$, if n is the number of variables in the query `Q1`. At this point, the Prolog list `Q1` with its frozen variables constitutes the canonical database, against which we have to evaluate `Q2`. The latter is achieved by the call to `satisfied` in line (3): we want to know whether the conjunctive query `Q2` can be satisfied over the canonical database `Q1`:

Line (4) handles the special case that we have the empty conjunction. The latter is true in any interpretation. Now assume we have a conjunctive query consisting of an atom `A`, and possibly more atoms `As` (denoted `[A|As]`), as well as the canonical database `DB` (line (5)). In order to

```
% contained(+Q1,+Q2) holds iff Q1 is contained in Q2
contained(Vs<-Q1, Vs<-Q2) :-                                  (1)
        numbervars(Vs<-Q1, a, 0,_), % freeze Q1               (2)
        satisfied(Q2,Q1).  % satisfy Q2 over canonical_db(Q1) (3)

% satisfied(+AtomList, +CanonicalDatabase)
satisfied([],_).          % empty conjunction => true         (4)
satisfied([A|As],DB) :-   % else...                           (5)
        member(A,DB),     % ... satisfy first atom wrt DB     (6)
        satisfied(As,DB). % ... same for remaining atoms      (7)
```

Figure 1: Complete Prolog algorithm CQCP for testing conjunctive query containment.

satisfy the query, in line (6), we first make sure that the atom `A` is true in `DB`. Note that `A` may have variables which become bound as we try to unify them with some *ground* (*i.e.*, variable free) atom in `DB`. The `member` predicate successively tries all tuples in `DB`. When `member` succeeds, we finally call `satisfied` recursively in line (7), to satisfy all remaining atoms.

4 Semantic Query Optimization

In Example 1 we have already alluded to the fact that we can use the containment test for semantic query optimization. Let us start with an example from database mediation.

Of Uncles and Aunts. Consider the views in Figure 2, which define family relations such as `uncle/2` and `aunt/2` in terms of other defined views such as `parent/2`, or, ultimately, in terms of base relations such as `male/1` and `spouse/2`. Let us consider that all *base relations* (*i.e.*, which occur in the body only) come from some remote database source (here, we do not care which base relation comes from which source), and that the defined views are exported by the database mediator system. That is, we have defined the global export schema in terms of views on the local source schemata. Hence, this approach is called *global-as-view* (GAV).

In an earlier technical note, we have explained how a user query such as `?- uncle(eva,X)` can be composed with these view definitions, resulting in a logic query plan.[9] Figure 3 shows the resulting logic query plan in Disjunctive Normal Form (DNF): In order to obtain a complete set of answers to the query `?- uncle(E,A)`, at runtime all 24 conjunctive queries have to be executed according to this plan. Then the union of these results will constitute the answer to the original query.[10]

Clearly, even for the simple "mechanics" of family relationships, it is not clear how the lengthy logical plan in Figure 3 can be optimized. However, query containment can come to the rescue: First, we may try to find whether any of the 24 conjunctions is contained in any other conjunction. If that were the case, then the contained conjunction can be eliminated from the plan.

We can solve this query optimization task by asserting the 24 conjunctive queries and simply invoking the same Prolog query as before:

```
?- Qid1 :: Q1, Qid2 :: Q2, Qid1 \= Qid2, contained(Q1, Q2).
```

Since the system answers with "`No`", we know for sure that in the case of our `?-uncle(E,A)` query plan, no conjunctive subplan implies any other one.

[9]Basically, the query and the views are unfolded until (in the non-recursive view case!!) the original query is expressed solely in terms of base predicates, without mentioning the view predicates.

[10]Also, as part of the mediator's planning task, the goals within each conjunction may have to be reordered, depending on the given binding pattern constraints for base predicates. Moreover, goals from the same source are preferably bundled together in a single execution request, provided they share at least one variable.

```
parent(X,Y) <- father(X,Y).
parent(X,Y) <- mother(X,Y).

son(X,Y)      <- parent(Y,X), male(Y).
daughter(X,Y) <- parent(Y,X), female(Y).

brother(X,Y) <- parent(X,Z), son(Z,Y), X \= Y.
sister(X,Y)  <- parent(X,Z), daughter(Z,Y), X \= Y.

brother_in_law(X,Y) <- sister(X,Z), spouse(Z,Y).
brother_in_law(X,Y) <- spouse(X,Z), brother(Z,Y).
sister_in_law(X,Y)  <- brother(X,Z), spouse(Z,Y).
sister_in_law(X,Y)  <- spouse(X,Z), sister(Z,Y).

uncle(X,Y) <- parent(X,Z), brother(Z,Y).
uncle(X,Y) <- parent(X,Z), brother_in_law(Z,Y).
aunt(X,Y)  <- parent(X,Z), sister(Z,Y).
aunt(X,Y)  <- parent(X,Z), sister_in_law(Z,Y).
```

Figure 2: Example views defining family relations.

Semantic Query Optimization. The previous attempt at optimizing the `uncle` plan did not succeed, *i.e.*, no conjunction is subsumed by any other conjunction. However, this does not preclude the possibility that the union of some conjunctions implies the union of some other set of conjunctions, rendering the former redundant. We will not pursue this path here.

Instead, let us try to directly optimize the DNF plan using some *semantic integrity constraints*, expressed as *denials*.

Example 4 (Disjointness of Mother and Father) It is quite natural to assume that a person cannot be both a mother and a father. This constraint is captured by the closed formula

$$\neg(\exists x)(\exists y)(\exists z)\ mother(x,y) \land father(z,y)$$

i.e., there are no x, y, z such that y is both a mother and a father (of some other persons x and y, respectively). This can be equivalently expressed as a denial as follows:

$$\mathsf{false} \leftarrow (\exists x)(\exists y)(\exists z)\ mother(x,y) \land father(z,y)$$

Along the lines of our above Prolog syntax, we write this denial as follows:

$$\texttt{ic(1)} :: \ \ \texttt{false(}\,\texttt{chtg(Y)}\,\texttt{)} \leftarrow [\ \texttt{mother(X,Y)}, \texttt{father(Z,Y)}\ ]\ .$$

Here we use an extra argument `chtg(Y)` for the special predicate `false/`n to indicate what "went wrong". In this case, `chtg(Y)` indicates that `Y` ***c***annot ***h***ave ***t***wo ***g***enders. The idea is that we can test such an integrity constraint at runtime, *e.g.*, by simply executing the query `?- false(`$\bar{\texttt{W}}$`)`. All bindings that we may obtain for $\bar{\texttt{W}}$ are then witnesses of the occurred integrity violations. □

Semantic integrity constraints such as the one in the previous example cannot only be used as tests at runtime (to check whether indeed the database instance is consistent), but also at compile-time, to optimize query plans, knowing (or assuming) that they hold true.

In Example 1, we already introduced the general idea of using denials for semantic query optimization. Given a denial ψ, *i.e.*, a closed formula which states which situations cannot occur, and a query φ, if $\varphi \to \psi$ is valid, the φ can be ignored.

We can also express this in the conventional form of query containment, but have to assume that the arities of the irrelevant query Q_φ and the containing "denial query" Q_ψ are the same: For $Q_\varphi := (\exists\bar{y})\,\varphi(\bar{x},\bar{y})$ and the denial query $Q_\psi := (\exists\bar{z})\,\psi(\bar{x},\bar{z})$, if $Q_\varphi \sqsubseteq Q_\psi$, then every answer to Q_φ is contained in the answer to Q_ψ. But since the latter is a denial, its answer is empty, and so Q_φ's answer must be empty and we can eliminate Q_φ from any logical plan.

```
1:: uncle(E,A) <- [male(A),father(E,B),father(B,C),father(A,C),neq(B,A)].
2:: uncle(E,A) <- [male(A),mother(E,B),father(B,C),father(A,C),neq(B,A)].
3:: uncle(E,A) <- [male(A),father(E,B),mother(B,C),father(A,C),neq(B,A)].
4:: uncle(E,A) <- [male(A),mother(E,B),mother(B,C),father(A,C),neq(B,A)].
5:: uncle(E,A) <- [male(A),father(E,B),father(B,C),mother(A,C),neq(B,A)].
6:: uncle(E,A) <- [male(A),mother(E,B),father(B,C),mother(A,C),neq(B,A)].
7:: uncle(E,A) <- [male(A),father(E,B),mother(B,C),mother(A,C),neq(B,A)].
8:: uncle(E,A) <- [male(A),mother(E,B),mother(B,C),mother(A,C),neq(B,A)].
9:: uncle(E,A) <- [father(E,B),spouse(C,A),female(C),father(B,D),father(C,D),neq(B,C)].
10:: uncle(E,A) <- [mother(E,B),spouse(C,A),female(C),father(B,D),father(C,D),neq(B,C)].
11:: uncle(E,A) <- [father(E,B),spouse(C,A),female(C),mother(B,D),father(C,D),neq(B,C)].
12:: uncle(E,A) <- [mother(E,B),spouse(C,A),female(C),mother(B,D),father(C,D),neq(B,C)].
13:: uncle(E,A) <- [father(E,B),spouse(C,A),female(C),father(B,D),mother(C,D),neq(B,C)].
14:: uncle(E,A) <- [mother(E,B),spouse(C,A),female(C),father(B,D),mother(C,D),neq(B,C)].
15:: uncle(E,A) <- [father(E,B),spouse(C,A),female(C),mother(B,D),mother(C,D),neq(B,C)].
16:: uncle(E,A) <- [mother(E,B),spouse(C,A),female(C),mother(B,D),mother(C,D),neq(B,C)].
17:: uncle(E,A) <- [male(A),father(E,B),spouse(B,C),father(C,D),father(A,D),neq(C,A)].
18:: uncle(E,A) <- [male(A),mother(E,B),spouse(B,C),father(C,D),father(A,D),neq(C,A)].
19:: uncle(E,A) <- [male(A),father(E,B),spouse(B,C),mother(C,D),father(A,D),neq(C,A)].
20:: uncle(E,A) <- [male(A),mother(E,B),spouse(B,C),mother(C,D),father(A,D),neq(C,A)].
21:: uncle(E,A) <- [male(A),father(E,B),spouse(B,C),father(C,D),mother(A,D),neq(C,A)].
22:: uncle(E,A) <- [male(A),mother(E,B),spouse(B,C),father(C,D),mother(A,D),neq(C,A)].
23:: uncle(E,A) <- [male(A),father(E,B),spouse(B,C),mother(C,D),mother(A,D),neq(C,A)].
24:: uncle(E,A) <- [male(A),mother(E,B),spouse(B,C),mother(C,D),mother(A,D),neq(C,A)].
```

Figure 3: Logical query plan (in Disjunctive Normal Form) for the query `?-uncle(E,A)`

Example 5 (Semantic Query Optimization for `uncle/2`) Consider the conjunctive queries of the DNF plan in Figure 3. We add to this set the above integrity constraint

$$\mathtt{ic}(1) :: \quad \mathtt{false}(\,\mathtt{chtg}(\mathtt{Y})\,) \leftarrow [\ \mathtt{mother}(\mathtt{X},\mathtt{Y}), \mathtt{father}(\mathtt{Z},\mathtt{Y})\ ]\ .$$

In order to compute which queries are irrelevant, we need to simply call `contained/2` with the first argument being any of the 24 queries, and the second argument the denial:

```
?-   Qid :: Q_Head <- Q_Body,                    % pick a query Q
     Qid \= ic(_),                               % ... but not a denial
     ic(ICid) :: Denial_Head <- Denial_Body,     % pick a denial D
     contained(q <- Q_Body , q <- Denial_Body).  % check Q -> D
```

This Prolog call has 12 solutions for `ICid=1`, revealing that 50% of the sub-queries in Figure 3 (*i.e.*, the queries with identifiers `3, 4, 5, 6, 11, 12, 13, 14, 19, 20, 21, 22`) can be discarded. See the appendix for details and more examples. □

5 Conjunctive Query Minimization

The final application of query containment we consider is the *minimization* of conjunctive queries. A conjunctive query Q is called *minimal* if there is no other conjunctive query $Q' \equiv Q$ which has fewer atoms than Q. Computing the minimal conjunctive query is an NP-complete problem and closely related to containment checking.

We use the following fact which is, according to [Koc01, Section 2.4], due to [CM77]: For any conjunctive query Q, there is a minimal query $Q' \equiv Q$ such that their heads (and thus their distinguished variables) are identical, and the body of Q' is a subset of the body of Q. Conjunctive queries can thus be optimized by checking all queries created by dropping body atoms from Q while preserving equivalence and searching for the smallest such query. We will use this in the algorithm CQMP (**C**onjunctive **Q**uery **M**inimization in **P**rolog) below.

Example 6 (Minimal Query) The following queries $\mathtt{q}_1$ and $\mathtt{q}_2$ are equivalent (see Example 2):

$$\mathtt{q}_1 :: \ \mathtt{q}(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3) \leftarrow [\ \mathtt{p}(\mathtt{Y}_1,\mathtt{Y}_2,\mathtt{X}_3), \mathtt{p}(\mathtt{X}_1,\mathtt{Y}_2,\mathtt{Y}_3), \mathtt{p}(\mathtt{Y}_4,\mathtt{X}_2,\mathtt{Y}_3), \mathtt{p}(\mathtt{X}_1,\mathtt{Y}_5,\mathtt{Y}_6), \mathtt{p}(\mathtt{Y}_1,\mathtt{Y}_5,\mathtt{X}_3)\ ]\ .$$
$$\mathtt{q}_2 :: \ \mathtt{q}(\mathtt{X}_1,\mathtt{X}_2,\mathtt{X}_3) \leftarrow [\ \mathtt{p}(\mathtt{Z}_1,\mathtt{Z}_2,\mathtt{X}_3), \mathtt{p}(\mathtt{X}_1,\mathtt{Z}_2,\mathtt{Z}_3), \mathtt{p}(\mathtt{Z}_4,\mathtt{X}_2,\mathtt{Z}_3)\ ]\ .$$

```
% minimal(+Query, -MinimizedQuery)
minimal(Vs<-Body, MinBody) :-                                            (a)
        minimal(Vs<-Body, [], MinBody).                                  (b)

%%% minimal(+Query, +MinimalSoFar,-MinimizedQuery)
% compute minimal version of Q, MinimalSoFar accumulates
minimal(_<-[], Ms, Ms).                                                  (1)
minimal(Vs<-[B|Bs], Ms, MinBody) :-               % pick an atom B          (2)
        append(Ms,Bs,MsBs),                       % everything but B        (3)
        copy_term(Vs<-MsBs, Vs_MsBs_new),         % make fresh copy of Q1   (4)
        (\+ \+ contained(Vs_MsBs_new, Vs<-[B|MsBs]) % \+\+ to undo bindings   (5)
         ->     minimal(Vs<-Bs,Ms,MinBody)        % B can be dropped        (6)
         ;      minimal(Vs<-Bs,[B|Ms],MinBody)    % B is in the minimal query (7)
        ).
```

Figure 4: Prolog algorithm CQMP for mimimizing conjunctive queries.

Indeed the latter is minimal, since dropping any of the three atoms results in a non-equivalent query. □

In a mediator system, non-minimal queries may occur as a result of view unfolding (or when the view designers, or users have come up with non-minimal views/queries). Minimal queries are in general preferable over non-minimal ones, since they require fewer algebraic operations. Moreover, a query which may not be executable because of binding pattern restrictions may become so by minimizing it (the non-executability may be caused by the redundant parts of the query) [Li03].

Minimality can also be generalized to include integrity constraints. The above definition of minimal is then relativized to consider only databases which satisfy the given integrity constraints. It is worth mentioning that when integrity constraints are given, a non-minimal query may be executable, while its minimal query may not be executable:

Example 7 (Non-Minimal Executable vs. Minimal Non-Executable) Consider the query $Q := \ \mathtt{q}(\mathtt{X}) \leftarrow \mathtt{p}(\mathtt{X})$ where `p` has a binding pattern restriction, forcing its argument to be bound at runtime. Also assume that we have the integrity constraint $\mathsf{false} \leftarrow \mathtt{p}(\mathtt{X}), \neg\mathtt{dom}(\mathtt{X})$, *i.e.*, for every `X` for which `p(X)` holds, also `dom(X)` holds.

Clearly Q is minimal but not executable. However the equivalent (*w.r.t.* the given integrity constraints!) and non-minimal query $Q' := \ \mathtt{q}(\mathtt{X}) \leftarrow \mathtt{dom}(\mathtt{X}), \mathtt{p}(\mathtt{X})$ *is* executable, since we can use `dom` to enumerate the domain of `p`. □

5.1 Algorithm CQMP

The basic idea for computing the minimal query Q' for Q is very simple: We obtain the minimal Q' by dropping as many atoms from the body of Q as possible, while maintaining equivalence. To check for the latter it is sufficient to test $Q' \sqsubseteq Q$, as we go (Q' is "shorter" than Q, so $Q' \sqsupseteq Q$ trivially holds).

Lines (a) and (b) are for convenience only and define the predicate `minimal/2`, which given a conjunctive query in its first argument, returns in its second argument the body of the minimal query (the head of the minimal query is `Vs` and thus can be obtained from the input query). However, note that instead of calling `?-minimal(Vs<-Q, Q_min)`, we can avoid lines (a) and (b) simply by calling `?-minimal(Vs<-Q, [], Q_min)`.

Line (1) handles the special case that the remaining input query is empty. Then we are done with minimization, and we return in the third argument whatever we have accumulated for the minimal body (`Ms`) in the second argument. In (2) we consider the recursive case, in which the body of the input query is split into its first atom `B`, and the remaining body list `Bs`. We then consider the query body `MsBs` without `B`, but including all accumulated goals `Ms`, which we know are part of the minimal body (3). To see whether `B` can be dropped, we need to test in line (5) whether ($\texttt{Vs} \leftarrow \texttt{MsBs}$) $\sqsubseteq$ ($\texttt{Vs} \leftarrow [\texttt{B}|\texttt{MsBs}]$), and if this is the case we indeed drop it by not adding it to the second accumulator argument (6), else we add it to the accumulator of required minimal goals (7); then we recurse to minimize `Bs` (6,7).

At the core is the test in (5) whether ($\texttt{Vs} \leftarrow \texttt{MsBs}$) $\sqsubseteq$ ($\texttt{Vs} \leftarrow [\texttt{B}|\texttt{MsBs}]$). Note that the test `contained/2` (explained above) *freezes* the queries by replacing all variables with constants, thereby preventing possible matches of subqueries with other goals. We can avoid this freezing by using double negation (`\+ \+ contained(...)`) in (5).[11]

Finally, we come to line (4): It creates a "fresh copy" of the query $\texttt{Vs} \leftarrow \texttt{MsBs}$ called `Vs_MsBs_new`, which is used in the containment test in (5). Why do we have to use a fresh copy? This is necessary to *rename apart* all non-distinguished variables of the two queries being tested. In the implementation of `contained/2` in Figure 1 we have (tacitly) assumed that the input queries Q_1 and Q_2 do *not share* any non-distinguished variables. Indeed this assumption is valid, *e.g.*, when we read the queries from a file.[12] However, in the algorithm above, the queries to be tested may share non-distinguished variables, leading to incorrect behavior of `contained/2` as illustrated in Example 8. Line (4) effectively renames apart all variables in the two queries being tested, thereby guaranteeing the correct behavior of `contained/2` (note that the latter unifies back the *distinguished variables* as required by the containment algorithm).

Example 8 (Variable Name Clash) Consider the following queries:

$$\begin{aligned}\mathtt{q_a} &= \mathtt{q(X)} \leftarrow [\ \mathtt{d(X), p(X,X), p(X,Y)}\ ]\ .\\ \mathtt{q_b} &= \mathtt{q(X)} \leftarrow [\ \mathtt{d(X), p(Z,Z)}\ ]\ .\\ \mathtt{q_c} &= \mathtt{q(X)} \leftarrow [\ \mathtt{d(X), p(Y,Y)}\ ]\ .\end{aligned}$$

If we test $\mathtt{q_a} \sqsubseteq \mathtt{q_b}$, then the test succeeds as expected:

```
?-  contained( q(X)<-[d(X), p(X,X),p(X,Y)] , q(X) <-[d(X),p(Z,Z)] ).
  X = a0
  Y = a1
  Z = a0
Yes
```

However, the direct test $\mathtt{q_a} \sqsubseteq \mathtt{q_c}$ (with $\mathtt{q_a}$ and $\mathtt{q_c}$ replaced by their respective bodies) fails, despite the fact that $\mathtt{q_a} \sqsubseteq \mathtt{q_c}$ holds (note that $\mathtt{q_c} \equiv \mathtt{q_b}$):

```
?-  contained( q(X)<-[d(X), p(X,X),p(X,Y)] , q(X) <-[d(X),p(Y,Y)] ).
No
```

This is because we have *not renamed apart* the (supposedly different) non-distinguished variables (here: `Y`) in $\mathtt{q_a}$ and $\mathtt{q_c}$, so a name clash occurs. In contrast, $\mathtt{q_a}$ and $\mathtt{q_b}$ do *not* share any non-distinguished variables. □

Had we read the queries $\mathtt{q_a}$ and $\mathtt{q_c}$ from Prolog's internal database, as we have assumed and done before, then their variables would have been renamed automatically and the test $\mathtt{q_a} \sqsubseteq \mathtt{q_c}$ would have succeeded:

Example 9 (Automatic Renaming) Recall that we can store query expressions as facts of the form

$$\mathtt{Qid} :: \quad \mathtt{q}(\bar{\mathtt{X}}) \leftarrow [\ \varphi(\bar{\mathtt{X}}, \bar{\mathtt{Y}})\ ]\ .$$

If we `assert` the above $\mathtt{q_a}$ and $\mathtt{q_c}$ in this way, we can retrieve them again as follows:

[11] A well-known technique to undo any variable bindings done by invocation of the Prolog goal G is to call `\+ \+` G: The negated goal `\+` G calls G, and if G is successful fails, undoing any binding for G. However, if G fails, then `\+` G succeeds (without binding variables). Thus, one more application of negation, *i.e.*, `\+ \+` G produces a goal that succeeds iff G succeeds, yet with all variable bindings of the G invocation undone.

[12] The scope of variables is an individual Prolog rule or fact, hence two terms coming from different rules/facts do not share any variables, unless at some point we use unification to equate some of them.

```
?- q_a :: Qa, q_c :: Qc.

  Qa = q(_G233)<-[d(_G233), p(_G233, _G233), p(_G233, _G250)]
  Qc = q(_G255)<-[d(_G255), p(_G265, _G265)]

Yes
```

The system automatically renames apart all variables from separate Prolog clauses. In particular, the name clash on the non-distinguished variable `Y` is avoided, since two distinct variables `_G250` and `_G265` are used in `Qa` and `Qc`, respectively. Now the test succeeds as expected:

```
?- q_a :: Qa, q_c :: Qc, contained(Qa, Qc).

  Qa = q(a0)<-[d(a0), p(a0, a0), p(a0, a1)]
  Qc = q(a0)<-[d(a0), p(a0, a0)]

Yes
```

Observe how the variables `_G250` and `_G265` have been replaced by distinct constants `a1` and `a0`, which was not possible before with the (incorrect) use of the same variable `Y`. □

References

[BD01] G. Brewka and J. Dix. Knowledge Representation with Logic Programs. In *Handbook of Philosophical Logic*. 2001.

[CM77] A. K. Chandra and P. M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *ACM Symposium on Theory of Computing (STOC)*, pp. 77–90, 1977.

[Koc01] C. Koch. *Data Integration against Multiple Evolving Autonomous Schemata*. PhD thesis, Technische Universität Wien, 2001.

[Li03] C. Li. Computing Complete Answers to Queries in the Presence of Limited Access Patterns. *Journal of VLDB*, 2003. conditional acceptance.

[Ull97] J. D. Ullman. Information Integration Using Logical Views. In F. Afrati and P. Kolaitis, editors, *6th Intl. Conference on Database Theory (ICDT)*, number 1186 in LNCS, Delphi, Greece, 1997. Springer.

A Prolog Code for CQCP and CQMP

We list the complete SWI-Prolog code necessary for conjunctive query containment and minimization, together with a some convenience predicates for running those algorithms on some input files.

```
% --------------------------------------------------------
% Conjunctive Query Containment and Minimization Algorithms
% File:          cqcp.swi
% Last modified: 08/06/2003
% Author:        Bertram Ludaescher (ludaesch@sdsc.edu)
% ---------------------------------------------------------

:- op(600,xfx,::).  % for naming queries: Qid :: Query
:- op(500,xfx,<-).  % Query syntax: QueryHead <- [QueryBody].


% query_file(F)
% F contains queries to be processed. Query format is:
%        QueryId ::  QueryHead  <- [ QueryBody ].
% Format for integrity constraints (in denial form) is:
%       ic(DenialId) ::  false(...) <- [ Denial ].
query_file('sample.swi').
query_file('uncle_queries.swi').


% Test everything
go :-
        query_file(F),
        load_queries(F),
        test_contained_all,
        test_irrelevant_all,
        test_minimal_all,
        fail
        ;
        true.

% Some simple profiling. It seems that profile/1 works
% only on "second try", hence the go(1) first ...
go_profile(N) :-
        go(1),
        profile(go(N)).

% Go N times
go(N) :-
        tell('dummy'),
        between(1,N,_),
        go,
        fail
        ;
        told.


% Load queries from first query_file
load_queries :-
        query_file(F),
        load_queries(F).

% Load queries from file F, discarding earlier queries,
% thus files are processed independently
load_queries(F) :-
        format('~n~`=t LOADING ~w ~`=t~78|~n', [F]),
        retractall( _Qid :: _Query ),    % discard any earlier queries
        see(F),
        repeat,
        read(Term),
        (Term = end_of_file
         ->     true
         ;      assert( Term ), % Term = QueryId :: QueryHead <- [ QueryBody ]
                fail
        ),!,
        seen,
        listing( :: ),                   % show what has been loaded
        findall(Id, Id :: _Query , Ids), % get a list of all query ids
        length(Ids, Count),              % and count them
        format('% total of ~w queries loaded.~n',[Count]).

% Test all queries against each other for containment (= O(n^2) tests for n queries)
test_contained_all :-
        format('~n~`-t CONTAINMENT TESTS~`-t~78|~n'),
```

```
        Qid1 :: Q1,
        Qid2 :: Q2,
        Qid1 \= Qid2,       % to avoid testing with oneself
        (contained(Q1, Q2)
         ->     format('~n~w => ~w~n',[Qid1, Qid2]), % containment holds
                format('  ~p =>~n  ~p~n', [Q1,Q2])
         ;      format('.')             % print '.' otherwise
        ),
        fail
        ;
        true.

% Test which queries are irrelevant due to integrity contraints
test_irrelevant_all :-
        format('~n~`-t INTEGRITY CONSTRAINTS TESTS ~`-t~78|~n'),
        Qid :: Q_Head <- Q_Body,                % pick a query
        Qid \= ic(_),                           % ... but not a constraint
        ic(ICid) :: IC_Head <- IC_Body,         % pick a constraint
        (contained(q <- Q_Body , q <- IC_Body)
         ->     format('~n~w is UNSATISFIABLE for ic(~w)~n',[Qid, ICid]),
                format('  ~p =>~n  ~p~n', [Q_Head <- Q_Body, IC_Head <-IC_Body])
         ;      format('.')             % print '.' otherwise
        ),
        fail
        ;
        true.

% Minimize all queries
test_minimal_all :-
        format('~n~`-t MINIMIZATIONS ~`-t~78|~n'),
        Qid :: Vs<-Q,
        minimal(Vs<-Q, M),
        numbervars(Vs<-Q, '$VAR', 0,_),
        length(Q,Q_len), length(M,M_len),
        (Q_len = M_len
         ->     format('~w is minimal:~n    ~w~n',
                   [Qid, Vs<-Q])
         ;      format('~w CAN BE MINIMIZED:~n    ~w~n<=> ~w~n',
                   [Qid, Vs<-Q, Vs<-M])
        ),
        fail
        ;
        true.

%============= CQCP Conjunctive Query Containment in Prolog ==============

% Q1 contained in Q2  <=>  Q2 has an answer over the canonical_db(Q1)
contained(Vs<-Q1,  Vs<-Q2) :-      % unify head variables Vs
        numbervars(Vs<-Q1, '$a', 0,_), % freeze Q1 => canonical_db D
        satisfied(Q2,Q1).          % evaluate Q2 over D

% satisfied(+As, +DB)
% is true if ALL atoms As can be made true in database DB
satisfied([], _).                 % empty conjunction => true
satisfied([A|As], DB) :-
        member(A, DB),            % if atom A can be satisfied in DB => OK; else fail
        satisfied(As, DB).        % continue with rest

%============= CQMP Conjunctive Query Minimization in Prolog ==============

% minimal(+Query, -MinimizedQuery)
minimal(Vs<-Body, MinBody) :-
        minimal(Vs<-Body, [], MinBody).

%%% minimal(+Query, +MinimalSoFar,-MinimizedQuery)
% compute minimal version of Q, MinimalSoFar accumulates
minimal(_<-[], Ms, Ms).
minimal(Vs<-[B|Bs], Ms, MinBody) :-               % pick an atom B
        append(Ms,Bs,MsBs),                       % everything but B
        copy_term(Vs<-MsBs, Vs_MsBs_new),         % make fresh copy of Q1
        (\+ \+ contained(Vs_MsBs_new, Vs<-[B|MsBs]) % \+\+ to undo bindings
         ->     minimal(Vs<-Bs,Ms,MinBody)        % B can be dropped
         ;      minimal(Vs<-Bs,[B|Ms],MinBody)    % B is in the minimal query
        ).
%========================================================================

% For pretty printing '$a'(i) without '$' and parentheses
portray('$a'(X)) :-
        format('a~w',[X]).
```

B Test Runs

```
Welcome to SWI-Prolog (Multi-threaded, Version 5.2.3)
Copyright (c) 1990-2003 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- % c:/my/Prolog/QueryContainment/cqcp.swi compiled 0.00 sec, 6,840 bytes

Yes
?- go.

============================ LOADING sample.swi ============================

:- dynamic (::)/2.

q_1::q(A, B, C)<-[p(D, E, C), p(A, E, F), p(G, B, F), p(A, H, I), p(D, H, C)].
q_2::q(A, B, C)<-[p(D, E, C), p(A, E, F), p(G, B, F)].
q_3::q(A)<-[p(A, A), p(A, B)].
q_4::q(A)<-[p(B, B), p(B, A)].
q_5::q(A, B)<-[p(A, A), p(A, B)].
q_6::q<-[p(A, A), p(A, B)].
% total of 6 queries loaded.

------------------------------ CONTAINMENT TESTS------------------------------

q_1 => q_2
  q(a0, a1, a2)<-[p(a3, a4, a2), p(a0, a4, a5), p(a6, a1, a5), p(a0, a7, a8), p(a3, a7, a2)] =>
  q(a0, a1, a2)<-[p(a3, a4, a2), p(a0, a4, a5), p(a6, a1, a5)]
....
q_2 => q_1
  q(a0, a1, a2)<-[p(a3, a4, a2), p(a0, a4, a5), p(a6, a1, a5)] =>
  q(a0, a1, a2)<-[p(a3, a4, a2), p(a0, a4, a5), p(a6, a1, a5), p(a0, a4, a5), p(a3, a4, a2)]
......
q_3 => q_4
  q(a0)<-[p(a0, a0), p(a0, a1)] =>
  q(a0)<-[p(a0, a0), p(a0, a0)]
.................
------------------------ INTEGRITY CONSTRAINTS TESTS -------------------------

------------------------------- MINIMIZATIONS --------------------------------
q_1 CAN BE MINIMIZED:
    q(A, B, C)<-[p(D, E, C), p(A, E, F), p(G, B, F), p(A, H, I), p(D, H, C)]
<=> q(A, B, C)<-[p(G, B, F), p(A, E, F), p(D, E, C)]
q_2 is minimal:
    q(A, B, C)<-[p(D, E, C), p(A, E, F), p(G, B, F)]
q_3 CAN BE MINIMIZED:
    q(A)<-[p(A, A), p(A, B)]
<=> q(A)<-[p(A, A)]
q_4 is minimal:
    q(A)<-[p(B, B), p(B, A)]
q_5 is minimal:
    q(A, B)<-[p(A, A), p(A, B)]
q_6 CAN BE MINIMIZED:
    q<-[p(A, A), p(A, B)]
<=> q<-[p(A, A)]

======================== LOADING uncle_queries.swi ==========================

:- dynamic (::)/2.

1::uncle(A, B)<-[male(B), father(A, C), father(C, D), father(B, D), neq(C, B)].
2::uncle(A, B)<-[male(B), mother(A, C), father(C, D), father(B, D), neq(C, B)].
3::uncle(A, B)<-[male(B), father(A, C), mother(C, D), father(B, D), neq(C, B)].
4::uncle(A, B)<-[male(B), mother(A, C), mother(C, D), father(B, D), neq(C, B)].
5::uncle(A, B)<-[male(B), father(A, C), father(C, D), mother(B, D), neq(C, B)].
6::uncle(A, B)<-[male(B), mother(A, C), father(C, D), mother(B, D), neq(C, B)].
7::uncle(A, B)<-[male(B), father(A, C), mother(C, D), mother(B, D), neq(C, B)].
8::uncle(A, B)<-[male(B), mother(A, C), mother(C, D), mother(B, D), neq(C, B)].
9::uncle(A, B)<-[father(A, C), spouse(D, B), female(D), father(C, E), father(D, E), neq(C, D)].
10::uncle(A, B)<-[mother(A, C), spouse(D, B), female(D), father(C, E), father(D, E), neq(C, D)].
11::uncle(A, B)<-[father(A, C), spouse(D, B), female(D), mother(C, E), father(D, E), neq(C, D)].
12::uncle(A, B)<-[mother(A, C), spouse(D, B), female(D), mother(C, E), father(D, E), neq(C, D)].
13::uncle(A, B)<-[father(A, C), spouse(D, B), female(D), father(C, E), mother(D, E), neq(C, D)].
14::uncle(A, B)<-[mother(A, C), spouse(D, B), female(D), father(C, E), mother(D, E), neq(C, D)].
15::uncle(A, B)<-[father(A, C), spouse(D, B), female(D), mother(C, E), mother(D, E), neq(C, D)].
16::uncle(A, B)<-[mother(A, C), spouse(D, B), female(D), mother(C, E), mother(D, E), neq(C, D)].
```

```
17::uncle(A, B)<-[male(B), father(A, C), spouse(C, D), father(D, E), father(B, E), neq(D, B)].
18::uncle(A, B)<-[male(B), mother(A, C), spouse(C, D), father(D, E), father(B, E), neq(D, B)].
19::uncle(A, B)<-[male(B), father(A, C), spouse(C, D), mother(D, E), father(B, E), neq(D, B)].
20::uncle(A, B)<-[male(B), mother(A, C), spouse(C, D), mother(D, E), father(B, E), neq(D, B)].
21::uncle(A, B)<-[male(B), father(A, C), spouse(C, D), father(D, E), mother(B, E), neq(D, B)].
22::uncle(A, B)<-[male(B), mother(A, C), spouse(C, D), father(D, E), mother(B, E), neq(D, B)].
23::uncle(A, B)<-[male(B), father(A, C), spouse(C, D), mother(D, E), mother(B, E), neq(D, B)].
24::uncle(A, B)<-[male(B), mother(A, C), spouse(C, D), mother(D, E), mother(B, E), neq(D, B)].
ic(1)::false(chtg(A))<-[father(B, A), mother(C, A)].
% total of 25 queries loaded.

------------------------------ CONTAINMENT TESTS------------------------------
.........................................................................................
.........................................................................................
.........................................................................................
.........................................................................................
.........................................................................................
.........................................................................................

------------------------ INTEGRITY CONSTRAINTS TESTS -------------------------
..
3 is UNSATISFIABLE for ic(1)
  uncle(a1, a0)<-[male(a0), father(a1, a2), mother(a2, a3), father(a0, a3), neq(a2, a0)] =>
  false(chtg(a3))<-[father(a0, a3), mother(a2, a3)]

4 is UNSATISFIABLE for ic(1)
  uncle(a1, a0)<-[male(a0), mother(a1, a2), mother(a2, a3), father(a0, a3), neq(a2, a0)] =>
  false(chtg(a3))<-[father(a0, a3), mother(a2, a3)]

5 is UNSATISFIABLE for ic(1)
  uncle(a1, a0)<-[male(a0), father(a1, a2), father(a2, a3), mother(a0, a3), neq(a2, a0)] =>
  false(chtg(a3))<-[father(a2, a3), mother(a0, a3)]

6 is UNSATISFIABLE for ic(1)
  uncle(a1, a0)<-[male(a0), mother(a1, a2), father(a2, a3), mother(a0, a3), neq(a2, a0)] =>
  false(chtg(a3))<-[father(a2, a3), mother(a0, a3)]
....
11 is UNSATISFIABLE for ic(1)
  uncle(a0, a3)<-[father(a0, a1), spouse(a2, a3), female(a2), mother(a1, a4), father(a2, a4), neq(a1, a2)] =>
  false(chtg(a4))<-[father(a2, a4), mother(a1, a4)]

12 is UNSATISFIABLE for ic(1)
  uncle(a0, a3)<-[mother(a0, a1), spouse(a2, a3), female(a2), mother(a1, a4), father(a2, a4), neq(a1, a2)] =>
  false(chtg(a4))<-[father(a2, a4), mother(a1, a4)]

13 is UNSATISFIABLE for ic(1)
  uncle(a0, a3)<-[father(a0, a1), spouse(a2, a3), female(a2), father(a1, a4), mother(a2, a4), neq(a1, a2)] =>
  false(chtg(a4))<-[father(a1, a4), mother(a2, a4)]

14 is UNSATISFIABLE for ic(1)
  uncle(a0, a3)<-[mother(a0, a1), spouse(a2, a3), female(a2), father(a1, a4), mother(a2, a4), neq(a1, a2)] =>
  false(chtg(a4))<-[father(a1, a4), mother(a2, a4)]
....
19 is UNSATISFIABLE for ic(1)
  uncle(a1, a0)<-[male(a0), father(a1, a2), spouse(a2, a3), mother(a3, a4), father(a0, a4), neq(a3, a0)] =>
  false(chtg(a4))<-[father(a0, a4), mother(a3, a4)]

20 is UNSATISFIABLE for ic(1)
  uncle(a1, a0)<-[male(a0), mother(a1, a2), spouse(a2, a3), mother(a3, a4), father(a0, a4), neq(a3, a0)] =>
  false(chtg(a4))<-[father(a0, a4), mother(a3, a4)]

21 is UNSATISFIABLE for ic(1)
  uncle(a1, a0)<-[male(a0), father(a1, a2), spouse(a2, a3), father(a3, a4), mother(a0, a4), neq(a3, a0)] =>
  false(chtg(a4))<-[father(a3, a4), mother(a0, a4)]

22 is UNSATISFIABLE for ic(1)
  uncle(a1, a0)<-[male(a0), mother(a1, a2), spouse(a2, a3), father(a3, a4), mother(a0, a4), neq(a3, a0)] =>
  false(chtg(a4))<-[father(a3, a4), mother(a0, a4)]
..
------------------------------ MINIMIZATIONS -------------------------------
1 is minimal:
    uncle(A, B)<-[male(B), father(A, C), father(C, D), father(B, D), neq(C, B)]
2 is minimal:
    uncle(A, B)<-[male(B), mother(A, C), father(C, D), father(B, D), neq(C, B)]
3 is minimal:
    uncle(A, B)<-[male(B), father(A, C), mother(C, D), father(B, D), neq(C, B)]
4 is minimal:
    uncle(A, B)<-[male(B), mother(A, C), mother(C, D), father(B, D), neq(C, B)]
5 is minimal:
    uncle(A, B)<-[male(B), father(A, C), father(C, D), mother(B, D), neq(C, B)]
```

```
6 is minimal:
    uncle(A, B)<-[male(B), mother(A, C), father(C, D), mother(B, D), neq(C, B)]
7 is minimal:
    uncle(A, B)<-[male(B), father(A, C), mother(C, D), mother(B, D), neq(C, B)]
8 is minimal:
    uncle(A, B)<-[male(B), mother(A, C), mother(C, D), mother(B, D), neq(C, B)]
9 is minimal:
    uncle(A, B)<-[father(A, C), spouse(D, B), female(D), father(C, E), father(D, E), neq(C, D)]
10 is minimal:
    uncle(A, B)<-[mother(A, C), spouse(D, B), female(D), father(C, E), father(D, E), neq(C, D)]
11 is minimal:
    uncle(A, B)<-[father(A, C), spouse(D, B), female(D), mother(C, E), father(D, E), neq(C, D)]
12 is minimal:
    uncle(A, B)<-[mother(A, C), spouse(D, B), female(D), mother(C, E), father(D, E), neq(C, D)]
13 is minimal:
    uncle(A, B)<-[father(A, C), spouse(D, B), female(D), father(C, E), mother(D, E), neq(C, D)]
14 is minimal:
    uncle(A, B)<-[mother(A, C), spouse(D, B), female(D), father(C, E), mother(D, E), neq(C, D)]
15 is minimal:
    uncle(A, B)<-[father(A, C), spouse(D, B), female(D), mother(C, E), mother(D, E), neq(C, D)]
16 is minimal:
    uncle(A, B)<-[mother(A, C), spouse(D, B), female(D), mother(C, E), mother(D, E), neq(C, D)]
17 is minimal:
    uncle(A, B)<-[male(B), father(A, C), spouse(C, D), father(D, E), father(B, E), neq(D, B)]
18 is minimal:
    uncle(A, B)<-[male(B), mother(A, C), spouse(C, D), father(D, E), father(B, E), neq(D, B)]
19 is minimal:
    uncle(A, B)<-[male(B), father(A, C), spouse(C, D), mother(D, E), father(B, E), neq(D, B)]
20 is minimal:
    uncle(A, B)<-[male(B), mother(A, C), spouse(C, D), mother(D, E), father(B, E), neq(D, B)]
21 is minimal:
    uncle(A, B)<-[male(B), father(A, C), spouse(C, D), father(D, E), mother(B, E), neq(D, B)]
22 is minimal:
    uncle(A, B)<-[male(B), mother(A, C), spouse(C, D), father(D, E), mother(B, E), neq(D, B)]
23 is minimal:
    uncle(A, B)<-[male(B), father(A, C), spouse(C, D), mother(D, E), mother(B, E), neq(D, B)]
24 is minimal:
    uncle(A, B)<-[male(B), mother(A, C), spouse(C, D), mother(D, E), mother(B, E), neq(D, B)]
ic(1) is minimal:
    false(chtg(A))<-[father(B, A), mother(C, A)]

Yes
?-
```

B.1 Some Profiling Information

The following statistics show that most time is spent, not surprisingly, in `member/2` (25%), which does the actual search for containment mappings (note also the number of 6,200 redos (backtracking) and 13,820 calls). Also freezing the query via `numbervars/4` and outputting results via `format/2` require significant effort (17.5% each). The latter can of course be eliminated in a real application.

```
?- go_profile(10).
=====================================================================
Total time: 0.44 seconds
=====================================================================
Predicate                        Box Entries =    Calls+Redos     Time
=====================================================================
lists:member/2                        20,020 =   13,820+6,200    25.2%
numbervars/4                           7,960 =    7,960+0        17.5%
format/2                               9,920 =    9,920+0        17.5%
satisfied/2                            7,340 =    7,340+0         5.0%
::/2                                   8,150 =      690+7,460     5.0%
contained/2                            8,080 =    8,080+0         5.0%
prolog_listing:portray_head/2            310 =      310+0         2.5%
test_minimal_all/0                        70 =       20+50        2.5%
read/1                                   330 =      330+0         2.5%
put/2                                    310 =      310+0         2.5%
$c_current_predicate/2                   380 =      360+20        2.5%
assert/1                                 310 =      310+0         2.5%
copy_term/2                            1,540 =    1,540+0         2.5%
lists:append/3                         1,540 =    1,540+0         0.0%
...

Yes
```