Optimizing select-project-join expressions

Jan Van den Bussche
Stijn Vansummeren

1 Introduction

The expression that we obtain by translating a SQL query q into the relational algebra is called the *logical query plan* for q. The ultimate goal is to further translate this logical query plan into a *physical query plan* that contains all details on how the query is to be executed. First, however, we can apply some important optimizations directly on the logical query plan. These optimizations transform the logical query plan into an equivalent logical query plan that is more efficient to execute.

In paragraph 16.3.3, the book discusses the simple (but very important) logical optimizations of "pushing" selections and projections; and recognizing joins, based on the algebraic equalities from section 16.2.

In addition, in these notes, we describe another important logical optimization: *the removal of redundant joins.*

2 Undecidability of the general problem

In general, we could formulate the optimization problem of relational algebra expressions as follows:

Input: A relational algebra expression e.

Output: A relational algebra expression e' such that:

1. e' is *equivalent* to e (denoted as $e' \equiv e$), in the sense that $e(D) = e'(D)$ for every database D. Here, $e(D)$ stands for the relation resulting from evaluating e on D.
2. e' is *optimal*, in the sense that there is no expression e' that is shorter than e', yet still equivalent to e.

Of course, we have to define what the *length* of a relational algebra expression is. A good way to do this is to count the number of times a relational algebra operator is applied in the expression.

Example 1. The length of the expression $R \cap (S \cap R)$ is two (there are two applications of $\cap$). This expression is not optimal because it is equivalent to the shorter expression $R \cap S$. □

Unfortunately, it is possible to show that *there is no algorithm that solves the relational algebra optimization problem above.* We can see this by considering the following *equivalence problem* for the relational algebra:

Input: Two relational algebra expressions.

Output: 'Yes' if these two expressions are equivalent; 'no' otherwise.

It is known that this problem is undecidable: there is no algorithm that (1) always returns the correct answer and (2) always returns this answer after a finite amount of time. Actually, the problem is already undecidable for relational algebra expressions over the extremely simple database schema with a single relation $R(A, B)$. The proof that the equivalence problem is undecidable falls outside the scope of this course.

Using the undecidability of the equivalence problem, however, we can show that there is no algorithm that solves the optimization problem. Indeed, suppose, for the purpose of contradiction, that there does exists some algorithm for the optimization problem. We can then construct the following algorithm to decide the equivalence problem:

1. Given two expressions e_1 and e_2, construct the expression $(e_1 - e_2) \cup (e_2 - e_1)$, and optimize it. Let e' be the result.

2. If e' is syntactically equal to one of the following two expressions, output 'Yes', otherwise output 'No'

$$\boldsymbol{\sigma}_{\text{false}}(R) \qquad \text{or} \qquad R - R.$$

Indeed, e_1 and e_2 are equivalent if, and only if, $(e_1 - e_2) \cup (e_2 - e_1)$ expresses the *empty query*, i.e., the query that returns the empty relation on all input databases. And the two expressions $\boldsymbol{\sigma}_{\text{false}}(R)$ and $R - R$ above are exactly the only two shortest expressions that express the empty query.

Conclusion We may hence conclude that there is no algorithm that can optimize *general* relational algebra expressions. In the remainder of these notes, however, we will see that there does exist an algorithm for optimizing expressions in a *fragment* of the relational algebra. This fragment consist of the so-called *select-project-join* expressions (to be formally defined later). Optimizing such expressions consists of eliminating redundant joins. We first motivate why eliminating such joins is important, and then move on to the actual optimization algorithm.

3 Redundant Joins

The relational algebra expression obtained by translating a SQL query can contain redundant joins for various reasons. The three most important reasons are human errors; using views; and integrity constraints.

Human errors. Inexperienced (or careless, or sleepy) SQL-programmers can occasionally be caught writing needlessly complicated SQL statements. Consider the following example.

```
SELECT movieTitle FROM StarsIn S1
WHERE starName IN (SELECT name
                   FROM MovieStar, StarsIn S2
                   WHERE birthdate = 1960
                    AND S2.movieTitle = S1.movieTitle)
```

A moment's reflection shows that there is no need to use the `StarsIn` relation twice in this statement. Indeed, the statement is equivalent to the following, simpler one.

```
SELECT movieTitle FROM StarsIn
WHERE starName IN (SELECT name
                   FROM MovieStar
                   WHERE birthdate = 1960)
```

If we now compare the relational algebra translations of the above two statements, then we see that the first expression has a redundant join:

$$\boldsymbol{\pi}_{S_1.\texttt{movieTitle}}(\boldsymbol{\rho}_{S_2}(\texttt{StarsIn}) \underset{S_2.\texttt{movieTitle}=S_1.\texttt{movieTitle}}{\bowtie} \boldsymbol{\rho}_{S_1}(\texttt{StarsIn}) \underset{S_1.\texttt{starName}=\texttt{name}}{\bowtie} \boldsymbol{\sigma}_{\texttt{birthdate}=1960}(\texttt{MovieStar}))$$

whereas the second does not:

$$\boldsymbol{\pi}_{S_1.\texttt{movieTitle}}(\boldsymbol{\rho}_{S_1}(\texttt{StarsIn}) \underset{S_1.\texttt{starName}=\texttt{name}}{\bowtie} \boldsymbol{\sigma}_{\texttt{birthdate}=1960}(\texttt{MovieStar})).$$

Views. Many applications query the database using views rather than the raw base tables. To evaluate such queries, the query compiler will automatically substitute each view name in a query by its corresponding defining SQL expression. The corresponding relational algebra expression will frequently contain redundant joins.

Consider, for example, the following view definition that bundles all information in the base relations `Movie`, `StarsIn`, and `MovieStar` into a big virtual "universal relation":

```
CREATE VIEW MovieView AS
  SELECT title, year, length, studioName, producerC#,
         name, address, gender, birthdate
  FROM Movie, StarsIn, MovieStar
  WHERE title = movieTitle AND year = movieYear
    AND starName = name
```

This view makes it easier to formulate queries. To retrieve all titles of movies with an actor born in 1960, we could write, for example:

```
SELECT title FROM MovieView WHERE birthdate = 1960
```

The query compiler transforms this statement by replacing `MovieView` by its defining expression:

```
SELECT title FROM (Q) WHERE birthdate = 1960
```

Here, Q denotes the Select-statement from the view definition above.

Now consider the following query, which retrieves all actors that have co-acted with Kevin Bacon:

```
SELECT V1.name
FROM MovieView V1, MovieView V2
WHERE V1.title = V2.title AND V1.year = V2.year
  AND V2.name = 'Kevin Bacon'
```

If we replace the two occurrences of `MovieView` by Q, and subsequently translate the resulting SQL query into the relational algebra, then we obtain the following expression (from which we have omitted the final projection)

$$\boldsymbol{\rho}_{V_1}(\texttt{Movie} \underset{\substack{\texttt{title=movieTitle}\\ \texttt{year=movieYear}}}{\bowtie} \texttt{StarsIn} \underset{\texttt{starName=name}}{\bowtie} \texttt{MovieStar}) \underset{\substack{V_1\texttt{.title=}V_2\texttt{.title}\\ V_1\texttt{.year=}V_2\texttt{.year}}}{\bowtie} \boldsymbol{\rho}_{V_2}(\texttt{Movie} \underset{\substack{\texttt{title=movieTitle}\\ \texttt{year=movieYear}}}{\bowtie} \texttt{StarsIn} \underset{\texttt{starName=name}}{\bowtie} \boldsymbol{\sigma}_{\texttt{name=`Kevin Bacon'}}(\texttt{MovieStar}))$$

Note that, because of the central join, one of the two joins with `Movie` is redundant. The expression is hence equivalent to the following simplification:

$$\boldsymbol{\rho}_{V_1}(\texttt{Movie} \underset{\substack{\texttt{title=movieTitle}\\ \texttt{year=movieYear}}}{\bowtie} \texttt{StarsIn} \underset{\texttt{starName=name}}{\bowtie} \texttt{MovieStar}) \underset{\substack{V_1\texttt{.title=}V_2\texttt{.movieTitle}\\ V_1\texttt{.year=}V_2\texttt{.movieYear}}}{\bowtie} \boldsymbol{\rho}_{V_2}(\texttt{StarsIn} \underset{\texttt{starName=name}}{\bowtie} \boldsymbol{\sigma}_{\texttt{name=`Kevin Bacon'}}(\texttt{MovieStar})) \quad (*)$$

Integrity constraints Consider the following query:

```
SELECT title
FROM Movie, StarsIn, MovieStar
WHERE title = movieTitle AND starName = name
  AND birthdate = 1960
```

In the presence of the integrity constraint that every `movieTitle` that occurs in the relation `StarsIn` must also occur as the `title` in relation `Movie`, we can simplify this query as follows:

```
SELECT movieTitle
FROM StarsIn, MovieStar
WHERE starName = name AND birthdate = 1960
```

The relational algebra translation for the first SQL query is

$$\boldsymbol{\pi}_{\texttt{title}}(\texttt{Movie} \underset{\texttt{title}=\texttt{movieTitle}}{\bowtie} \texttt{StarsIn} \underset{\texttt{starName}=\texttt{name}}{\bowtie} \boldsymbol{\sigma}_{\texttt{name}=1960}(\texttt{MovieStar})) \tag{$\dagger$}$$

while the translation of the second SQL query has one join less:

$$\boldsymbol{\pi}_{\texttt{movieTitle}}(\texttt{StarsIn} \underset{\texttt{starName}=\texttt{name}}{\bowtie} \boldsymbol{\sigma}_{\texttt{name}=1960}(\texttt{MovieStar})).$$

Note, however, that this join is only redundant when the database satisfies the integrity constraint mentioned above.

4 Select-project-join expressions and conjunctive queries

From now on, we will only occupy ourselves with relational algebra expressions built from the following operators:

- projection;
- selection $\boldsymbol{\sigma}_\theta$, where the condition θ is a conjunction of equalities;
- cartesian product; natural join; or theta-join. For theta-joins, the join condition must again be a conjunction of equalities;
- renaming.

We call such expressions "select-project-join" expressions. To illustrate, all expressions from Section 3 are examples of select-project-join expressions. In contrast, $R \cup S$ and $R - S$ are not select-project-join expressions (because union and difference are not mentioned in the list of operators above), nor are $R \bowtie_{R.A=S.B \vee R.C=S.C} S$ and $R \bowtie_{R.A>S.B} S$ (because the join conditions are not conjunctions of equalities).

There exists an alternate syntax for select-project-join queries that is much easier to work with for the purpose of optimization. This syntax is the syntax of "conjunctive queries".

4.1 Definition of conjunctive queries

An *atom* is an expression of the form $R(u_1, \ldots, u_k)$, where R is the name of a database relation with k attributes, and u_1, ..., u_k are *terms*. A *term* is either a variable or a constant value. For example, the following is an atom:

$$\texttt{MovieStar}(\text{'Kevin Bacon'}, x, y, 1960).$$

Here, x and y are variables, and 'Kevin Bacon' and 1960 are constants.

A *conjunctive query* consists of a *head* and a *body*:

- The body is a finite set of atoms. These atoms in the body are sometimes also called *subgoals*.
- The head is a finite tuple of variables. Each variable from the head must occur in some atom in the body.

We write conjunctive queries as

$$Q(head) \leftarrow body$$

where Q is a name for the query.

Example 2. The following are two examples of conjunctive queries.

$$Q_1(t) \leftarrow \texttt{Movie}(t, y, \ell, i, s, p), \texttt{StarsIn}(t, y_2, n), \texttt{MovieStar}(n, a, g, 1960)$$

$$Q_2(n) \leftarrow \texttt{Movie}(t, y, \ell, i, s, p), \texttt{StarsIn}(t, y, n), \texttt{MovieStar}(n, a, g, b),$$
$$\texttt{StarsIn}(t, y, \text{'Kevin Bacon'}), \texttt{MovieStar}(\text{'Kevin Bacon'}, a_2, g_2, b_2)$$

Notice in particular that the same variable (like t and n) can occur multiple times in a conjunctive query! □

4.2 Semantics of conjunctive queries

There is a very elegant way to define the result of a conjunctive query Q on a database D. For this, we have to look at a relational database through another pair of eyes: we can view every tuple t in a relation R as the atom $R(t)$. Since t contains only constant data values (not variables), all terms in $R(t)$ are constants. Such atoms without variables are called *facts*. We can then view a relational database simply as *a set of facts*.

Example 3. Consider the following toy database D:

R

1	2
2	3
2	5
6	7
7	5
5	5

S

2
7

Viewed through our other set of eyes, this database is the following set of facts:

$$\{R(1,2), R(2,3), R(2,5), R(6,7), R(7,5), R(5,5), S(2,7), S(2), S(7)\} \quad \square$$

A *substitution* of a conjunctive query Q in a database D is a function f that maps each variable occurring in Q to a constant occurring in D.

We can apply f also on tuples of variables; atoms; and sets of atoms as follows. If t is a tuple of variables (e.g., the head of Q), then we write $f(t)$ for the tuple of constants obtained by replacing each variable x in t by $f(x)$. Similarly, we can apply f to whole atoms: simply replace each variable x occurring in the atom by $f(x)$ and leave the constants untouched. Note that the result is always a fact. Finally, we can also apply f to sets of atoms: simply apply f on every atom in the set. Note that the result is always a set of facts.

Example 4. Consider again the database D from Example 3. Consider the following conjunctive query over the relations $R(A, B)$ en $S(C)$:

$$Q(x, y) \leftarrow R(x, y),\ R(y, 5),\ S(y).$$

Then f defined as follows is a substitution of Q into D:

$$f\colon \quad \begin{array}{lcl} x & \mapsto & 1 \\ y & & 2 \end{array}$$

When we apply f to the atom $U(1, x, 2, y)$ we get the fact $U(1, 1, 2, 2)$. When we apply f on the body of Q (which, by definition, is a set of atoms), we get:

$$f(\{R(x, y),\ R(y, 5),\ S(y)\}) = \{R(1, 2), R(2, 5), S(2)\}.$$

$\square$

We say that f is a *matching* if $f(\mathit{body}) \subseteq D$. We then define the result of Q on D as the following relation:

$$Q(D) := \{f(\mathit{head}) \mid f \text{ is a matching of } Q \text{ into } D\}.$$

Example 5. Let D be the database from Example 3 and let Q be the conjunctive query from Example 4. Then the following two substitutions of Q into D are the only matchings of Q into D:

$$f\colon \quad \begin{array}{lcl} x & \mapsto & 1 \\ y & & 2 \end{array} \qquad \text{and} \qquad g\colon \quad \begin{array}{lcl} x & \mapsto & 6 \\ y & & 7 \end{array}$$

Indeed, $f(\mathit{body}) = \{R(1, 2),\ R(2, 5),\ S(2)\}$, and this set of facts is indeed a subset of the set of facts in D. In the same way we have $g(\mathit{body}) = \{R(6, 7),\ R(7, 5),\ S(7)\}$ which is again a subset of D. So,

$$Q(D) = \{f(x, y),\ g(x, y)\} = \{(1, 2),\ (6, 7)\}.$$

$\square$

4.3 Translating select-project-join expressions into conjunctive queries

Each select-project-join expression can easily be translated into an equivalent conjunctive query:

- For each relation in the expression we add a new atom to the body of the conjunctive query.
- Initially, all atoms have distinct variables.
- However, when a select condition, or a natural join, or a join condition specifies that two attributes are equal, we *identify* the corresponding variables of the corresponding atoms.
- Moreover, when a select condition specifies that the value of an attribute is equal to a constant, we replace the corresponding variable in the corresponding atom by that constant.
- Finally, the head of the conjunctive query consists of those variables that correspond to the output-attributes of the expression.

Example 6. Q_1 from Example 2 corresponds in this sense to the conjunctieve query for expression (†) on page 5. Q_2 from Example 2 corresponds in this sense to the conjunctive query for expression (∗) on page 4. □

Conversely, we can translate each conjunctive query into an equivalent select-project-join expression:

- We take the cartesian product consisting of a relation for each atom in the body. When multiple atoms in the body share the same relation name, the corresponding occurrences of relation names in the cartesian product must be renamed ($\boldsymbol{\rho}$).
- Whenever two atoms share a variable, we add to the select-project-join expression constructed so far a select (σ) condition that equates the corresponding attributes.
- Whenever an atom mentions a constant, we add to the select-project-join expression constructed so far a select (σ) condition that equates the corresponding attribute to that constant.
- Finally, we add a projection (π) on the attributes that correspond to the variables in the head.

Example 7. By translating the conjunctive query from Example 5 in this way, we obtain:

$$\boldsymbol{\pi}_{R_1.A,R_1.B}\,\boldsymbol{\sigma}_{R_1.B=R_2.A}\,\boldsymbol{\sigma}_{R_2.B=5}\,\boldsymbol{\sigma}_{R_1.B=C}(\boldsymbol{\rho}_{R_1}(R)\times\boldsymbol{\rho}_{R_2}(R)\times S).$$

□

Conclusion We may hence conclude that Select-project-join expressions and conjunctive queries are two different syntaxes for the same class of queries.

5 Containment

We say that a query Q_1 is *contained in* a query Q_2 if, for each database D, we have $Q_1(D) \subseteq Q_2(D)$. We write $Q_1 \subseteq Q_2$ to indicate that Q_1 is contained in Q_2.

Example 8. Consider the following extremely simple conjunctive queries over a relation $R(A, B)$:

$Q_0(x) \leftarrow R(x, 33)$
$Q_1(x) \leftarrow R(x, x)$
$Q_2(x) \leftarrow R(x, y)$

Then $Q_0 \equiv \boldsymbol{\pi}_A \boldsymbol{\sigma}_{B=33}(R)$; $Q_1 \equiv \boldsymbol{\pi}_A \boldsymbol{\sigma}_{A=B}(R)$; and $Q_2 \equiv \boldsymbol{\pi}_A(R)$. Clearly, therefore, $Q_0 \subseteq Q_2$ and $Q_1 \subseteq Q_2$. □

Example 9. Consider the following conjunctive queries:

$A(x, y) \leftarrow R(x, w),\ G(w, z),\ R(z, y)$
$B(x, y) \leftarrow R(x, w),\ G(w, w),\ R(w, y)$

Then $B \subseteq A$. Let us formally prove this by means of the formal definition of the semantics of conjunctive queries. Let D be an arbitrary database, and consider an arbitrary tuple in $B(D)$, the result of B on D. We have to show that this tuple is also in $A(D)$. By definition of $B(D)$, we know that the tuple under consideration is of the form $(f(x), f(y))$, with f a matching from B into D. In particular,

$$f(\{R(x, w),\ G(w, w),\ R(w, y)\}) \\ = \{R(f(x), f(w)), G(f(w), f(w)), R(f(w), f(y))\} \subseteq D.$$

Then the following substitution is a matching from A into D:

$$g\colon \quad \begin{array}{ccc} x & \mapsto & f(x) \\ y & & f(y) \\ w & & f(w) \\ z & & f(w) \end{array}$$

Indeed,

$$g(\{R(x, w),\ G(w, z),\ R(z, y)\}) \\ = \{R(f(x), f(w)), G(f(w), f(w)), R(f(w), f(y))\} \subseteq D.$$

Hence, $(g(x), g(y)) \in A(D)$. As such, $(f(x), f(y)) = (g(x), g(y)) \in A(D)$, as desired. □

Note that $Q_1 \equiv Q_2$ if, and only if, both $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$. We shall now show that containment, and hence also equivalence, of conjunctive queries is decidable. To this end, we require the following definition.

A *homomorphism* of Q_2 to Q_1 is a function h that maps each variable in Q_2 to either a variable or constant in Q_1 such that

$$h(head_2) = head_1 \qquad \text{and} \qquad h(body_2) \subseteq body_1.$$

Example 10. In Example 8:

- $x \mapsto x$ and $y \mapsto 33$ is a homomorphism of Q_2 to Q_0.
- $x \mapsto x$ and $y \mapsto x$ is a homomorphism of Q_2 to Q_1.
- Homomorphisms leave constant values untouched. Hence, because the constant 33 does not occur in Q_1, nor in Q_2, there cannot exist a homomorphism from Q_0 to Q_1 or Q_2. After all, there is no atom of the form $R(h(x), 33)$ in the body of Q_1 or Q_2, whatever the choice of $h(x)$.
- There also cannot exist a homomorphism from Q_1 to Q_2. After all, there is no atom of the form $R(h(x), h(x))$ in the body of Q_2, whatever the choice of $h(x)$ and $h(y)$.

Example 11. In example 9 there is a homomorphism from A to B, namely:

$$\begin{array}{lccc} h\colon & x & \mapsto & x \\ & y & & y \\ & w & & w \\ & z & & w \end{array}$$

In contrast, there is no homomorphism of B to A. Such a homomorphism would have to map $G(w, w)$ onto $G(w, z)$, which implies that we would have both $w \mapsto w$ and $w \mapsto z$. This is a contradiction, however, since a homomorphism must be function. □

Example 12. Consider the conjunctive queries C_1 and C_2:

$C_1(x) \leftarrow R(x, y),\ R(y, z),\ R(z, w)$
$C_2(x) \leftarrow R(x, y),\ R(y, x)$

There is a homomorphism from C_1 to C_2, namely:

$$\begin{array}{lccc} h\colon & x & \mapsto & x \\ & y & & y \\ & z & & x \\ & w & & y \end{array}$$

However, there is no homomorphism from C_2 to C_1. Indeed, suppose for the purpose of contradiction that such a homomorphism h does exist. Then it would have to map $x \mapsto x$ in order to map the head of C_2 into the head of C_1. In that case, however, $R(h(x), h(y)) = R(x, h(y))$ must occur in the body of C_1. This implies that the first subgoal of C_2 must be mapped onto the first subgoal of C_1 (since this is the only subgoal where x occurs in the first column), and hence, that h must map $y \mapsto y$. Analogously, this implies that the second subgoal of C_2 must be mapped to the second subgoal of C_1. Hence we would have to have $x \mapsto z$. But we had already established that $x \mapsto x$, which yields the desired contradiction since h must be a function. □

Theorem. *$Q_1 \subseteq Q_2$ if, and only if, there is a homomorphism from Q_2 to Q_1.*

Proof. (If) Let $h : Q_2 \to Q_1$ be a homomorphism. Let D be a database. Fix an arbitrary tuple $t \in Q_1(D)$. We have to prove that $t \in Q_2(D)$. Since $t \in Q_1(D)$ we know by definition of the semantics of conjunctive queries that $t = f(\mathit{head}_1)$, with f a matching of Q_1 into D. Now consider the composition $f \circ h$ of f with h. Clearly, this is a substitution of Q_2 into D. Because h is a homomorphism we know that $h(\mathit{body}_2) \subseteq \mathit{body}_1$. Consequently $f(h(\mathit{body}_2)) \subseteq f(\mathit{body}_1) \subseteq D$. In other words, $f \circ h$ is a matching of Q_2 into D, and hence $f(h(\mathit{head}_2)) \in Q_2(D)$. As such, $t = f(\mathit{head}_1) = f(h(\mathit{head}_2)) \in Q_2(D)$, as desired.

(Only if) Suppose that $Q_1 \subseteq Q_2$. If we consider the variables in Q_1 not as variables but as constant data values, then we can look at body_1 as a mini-database D_0. The identity function is a matching of Q_1 into D_0, and hence $\mathit{head}_1 \in Q_1(D_0)$. Then, since $Q_1 \subseteq Q_2$, we know that $\mathit{head}_1 \in Q_2(D_0)$. There hence exists a matching f from Q_2 into D_0 such that $\mathit{head}_1 = f(\mathit{head}_2)$ and $f(\mathit{body}_2) \subseteq D_0 = \mathit{body}_1$. This f is clearly a homomorphism of Q_2 to Q_1 (by considering variables again as variables). □

From the proof above we may synthesize the following surprising conclusion:

Golden Method. *To decide whether $Q_1(D) \subseteq Q_2(D)$ for* **every** *database D, it suffices to evaluate Q_2 on* **one single** *database D_0, namely the body of Q_1, and check that the head of Q_1 is in the result.*

This rule immediate yields an algorithm to test containment of conjunctive queries. We call the database D_0 above also the "canonical database" of Q_1.

Example 13. Let us reconsider Example 12. We have seen that there exists a homomorphism from C_1 to C_2. Hence, by Theorem 5, we know $C_2 \subseteq C_1$. (Careful: C_2 plays the role of Q_1 and C_1 the role of Q_2 in Theorem 5.) We can also check this by means of the Golden Rule above. The canonical

database D_0 is $\{R(x, y), R(y, x)\}$. If we evaluate C_1 on this database we indeed get (x), the head of C_2, in the output through the following matching:

$$\begin{array}{ccc} x & \mapsto & x \\ y & & y \\ z & & x \\ w & & y \end{array}$$

It is no coincidence that this is exactly the homomorphism from C_1 to C_2 that we already knew from Example 12!

Conversely, let us check that C_1 is *not* contained in C_2, again by means of the Golden Rule. In this case, the canonical database is $\{R(x, y), R(y, z), R(z, w)\}$. If we evaluate C_2 on this database we see that the result is empty; there is no matching of C_2 into D_0. Since, in contrast, $(x) \in C_1(D_0)$, we see that D_0 is a counterexample to the hypothesis that $C_1(D) \subseteq C_2(D)$ for every database D! □

6 Optimizing conjunctive queries

The number of atoms in the body of a conjunctive query corresponds to the number of joins in the corresponding select-project-join expression. We therefore call a conjunctive query *optimal* if there is no equivalent conjunctive query with fewer atoms in the body. Optimizing a given conjunctive query then consists in finding an equivalent conjunctive query that is optimal.

We could imagine the following optimization algorithm:

1. Given a conjunctive query Q.
2. Pick a random atom in the body.
3. Check whether the chosen atom is redundant as follows: Let Q' be the query we obtain by removing the chosen atom from the body of Q. Then decide whether Q' is equivalent to Q. (It suffices to only check $Q' \subseteq Q$ since $Q \subseteq Q'$ always holds; do you know why?)
4. If Q' is indeed equivalent to Q, then replace Q by Q'.
5. Repeat until we have removed all redundant atoms.

However, it is not clear that this algorithm is correct. Even if we have removed all redundant atoms, it is in principle possible that there exists a completely different equivalent conjunctive query with fewer atoms in its body. Luckily, we can show, however, that this is not the case.

Theorem. *For each conjunctive query we can find an equivalent, optimal conjunctive query by removing atoms from its body.*

Proof. Let Q be a conjunctive query and let P be an arbitrary query that is optimal, and equivalent to Q. Since $Q \equiv P$, certainly $Q \subseteq P$. By Theorem 5 there hence exists a homomorphism h of P to Q. Then let Q' be the query obtained from Q by removing all atoms in its body that are not in the image of h on P (i.e., $\mathit{body}(Q') = h(\mathit{body}(P))$).

Then, on the one hand we have $Q \subseteq Q'$ (the identity function is a homomorphism from Q' to Q), and on the other hand we have $Q' \subseteq P$ (because of h). Therefore, Q' is equivalent to Q. Moreover, Q' is optimal, since it has at most as many atoms in its body as P. □

Example 14. Consider the conjunctive query

$$Q(x) \leftarrow R(x, x),\ R(x, y)$$

If we remove $R(x, y)$ we obtain

$$Q'(x) \leftarrow R(x, x).$$

We have $Q' \subseteq Q$ since $x \mapsto x$, $y \mapsto x$ is a homomorphism. Hence Q' is equivalent to Q. In addition Q' is optimal: we cannot remove more atoms from Q' because it has only one atom. □

Example 15. Consider

$$Q(y) \leftarrow R(x, x),\ R(x, y)$$

We cannot remove $R(x, y)$ because we would then end up with an invalid conjunctive query (variable y in the head would no longer occur in the body). When we remove $R(x, x)$ we obtain:

$$Q'(y) \leftarrow R(x, y).$$

We have $Q' \not\subseteq Q$ (consider the canonical database $\{R(x, y)\}$) and hence Q is already optimal; no atom can be removed. □

7 Closing remarks

The select-project-join expressions (the conjunctive queries) essentially capture the relational algebra without the union and difference operators. We have seen that the equivalence problem for the relational algebra in general is undecidable, while it is decidable for the select-project-join fragment. Can we extend the latter fragment without losing decidability? The asnwer to this question is affirmative: it turns out that we can allow union and difference in a limited manner such that equivalence remains decidable. This falls outside the scope of this course, however.

Note that, as long as the equivalence problem is decidable, one can always (inefficiently) optimize a given expression by systematically enumerating all

shorter expressions (there can be many such expressions, but only a finite number), from smaller to larger, and returning the first equivalent expression found.

Finally, we would like to briefly return on the subject of eliminating redundant joins in the presence of integrity constraints (see the end of Section 3). With the techniques introduced in these notes we can only decide whether two queries are equivalent on *all* databases. Expression (†) on page 5 is *not* equivalent to expression (3) in this sense, however. It is only equivalent on those database satisfying the integrity constraint. There exists a technique called the "chase" that allows deciding equivalence of conjunctive queries only over database that satisfy a given integrity constraint.