30. Query rewriting in relational databases

In chapter “Relational database design” basic concepts of relational databases were introduced, such as relational schema, relation, instance. Databases were studied from the designer point of view, the main question was how to avoid redundant data storage, or various anomalies arising during the use of the database.

In the present chapter the schema is considered to be given and focus is on fast and efficient ways of answering user queries. First, basic (theoretical) query languages and their connections are reviewed in Section 30.1.

In the second part of this chapter (Section 30.2) views are considered. Informally, a view is nothing else, but result of a query. Use of views in query efficiency, providing physical data independence and data integration is explained.

Finally, the third part of the present chapter (Section 30.3) introduces query rewriting.

30.1. Queries

Consider the database of cinemas in Budapest. Assume that the schema consists of three relations:

$$\mathbf{CinePest} = \{\mathit{Film},\mathit{Theater},\mathit{Show}\}\ . \tag{30.1}$$

The schemata of individual relations are as follows:

$$\begin{array}{rcl} \mathit{Film} & = & \{\mathbf{T}\mathit{itle},\mathbf{D}\mathit{irector},\mathbf{A}\mathit{ctor}\}\ , \\ \mathit{Theater} & = & \{\mathbf{Th}\mathit{eater},\mathbf{A}\mathit{ddress},\mathbf{P}\mathit{hone}\}\ , \\ \mathit{Show} & = & \{\mathbf{Th}\mathit{eater},\mathbf{T}\mathit{itle},\mathbf{Ti}\mathit{me}\}\ . \end{array} \tag{30.2}$$

Possible values of instances of each relation are shown on Figure 30.1.

Typical user queries could be:

30.1 Who is the director of “Control”?

30.2 List the names address of those theatres where Kurosawa films are played.

30.3 Give the names of directors who played part in some of their films.

These queries define a mapping from the relations of database schema **CinePest** to some other schema (in the present case to schemata of single relations). Formally, ***query*** and

Film

T_itle_	**D**_irector_	**A**_ctor_
Control	Antal, Nimród	Csányi, Sándor
Control	Antal, Nimród	Mucsi, Zoltán
Control	Antal, Nimród	Pindroch, Csaba
⋮	⋮	⋮
Rashomon	Akira Kurosawa	Toshiro Mifune
Rashomon	Akira Kurosawa	Machiko Kyo
Rashomon	Akira Kurosawa	Mori Masayuki

Theatre

Th_eater_	**A**_ddress_	**P**_hone_
Bem	II., Margit Blvd. 5/b.	316-8708
Corvin	VIII., Corvin alley 1.	459-5050
Európa	VII., Rákóczi st. 82.	322-5419
Művész	VI., Teréz blvd. 30.	332-6726
⋮	⋮	⋮
Uránia	VIII., Rákóczi st. 21.	486-3413
Vörösmarty	VIII., Üllői st. 4.	317-4542

Show

Th_eater_	**T**_itle_	**Ti**_me_
Bem	Rashomon	19:00
Bem	Rashomon	21:30
Uránia	Control	18:15
Művész	Rashomon	16:30
Művész	Control	17:00
⋮	⋮	⋮
Corvin	Control	10:15

Figure 30.1. The database **CinePest**.

query mapping should be distinguished. The former is a syntactic concept, the latter is a mapping from the set of instances over the input schema to the set of instances over the output schema, that is determined by the query according to some semantic interpretation. However, for both concepts the word "query" is used for the sake of convenience, which one is meant will be clear from the context.

Definition 30.1 *Queries q_1 and q_2 over schema R are said to be* ***equivalent,*** *in notation $q_1 \equiv q_2$, if they have the same output schema and for every instance $\mathcal{I}$ over schema R*

$q_1(\mathcal{I}) = q_2(\mathcal{I})$ holds.

In the remaining of this chapter the most important query languages are reviewed. The expressive powers of query languages need to be compared.

Definition 30.2 *Let Q_1 and Q_2 be query languages (with appropriate semantics). Q_2 is **dominated by** Q_1 (Q_1 is **weaker,** than Q_2), in notation $Q_1 \sqsubseteq Q_2$, if for every query q_1 of Q_1 there exists a query $q_2 \in Q_2$, such that $q_1 \equiv q_2$. Q_1 and Q_2 are **equivalent,** if $Q_1 \sqsubseteq Q_2$ and $Q_1 \sqsupseteq Q_2$.*

Example 30.1 *Query.* Consider Question 30.2. As a first try the next solution is obtained:

> **if** there exist in relations *Film*, *Theater* and *Show* tuples $(x_T,$ "Akira Kurosawa"$, x_A)$, (x_{Th}, x_{Ad}, x_P) and (x_{Th}, x_T, x_{Ti})
> **then** put the tuple $(\mathit{Theater} : x_{Th}, \mathit{Address} : x_A)$ into the output relation.

$x_T, x_A, x_{Th}, x_{Ad}, x_P, x_{Ti}$ denote different variables that take their values from the domains of the corresponding attributes, respectively. Using the same variables implicitly marked where should stand identical values in different tuples.

30.1.1. Conjunctive queries

Conjunctive queries are the simplest kind of queries, but they are the easiest to handle and have the most good properties. Three equivalent forms will be studied, two of them based on logic, the third one is of algebraic nature. The name comes from first order logic expressions that contain only existential quantors (∃), furthermore consist of atomic expressions connected with logical "and", that is conjunction.

Datalog – rule based queries

The tuple $(x_1, x_2, \ldots, x_m)$ is called ***free tuple*** if the x_i's are variables or constants. This is a generalisation of a tuple of a relational instance. For example, the tuple $(x_T,$ "Akira Kurosawa"$, x_A)$ in Example 30.1. is a free tuple.

Definition 30.3 *Let **R** be a relational database schema. **Rule based conjunctive query** is an expression of the following form*

$$ans(u) \leftarrow R_1(u_1), R_2(u_2), \ldots, R_n(u_n) \ , \tag{30.3}$$

where $n \geq 0$, $R_1, R_2, \ldots, R_n$ are relation names from $\mathbf{R}$, ans is a relation name not in $\mathbf{R}$, $u, u_1, u_2, \ldots, u_n$ are free tuples. Every variable occurring in u must occur in one of $u_1, u_2, \ldots, u_n$, as well.

The rule based conjunctive query is also called a ***rule*** for the sake of simplicity. $ans(u)$ is the ***head*** of the rule, $R_1(u_1), R_2(u_2), \ldots, R_n(u_n)$ is the ***body*** of the rule, $R_i(u_i)$ is called a ***(relational) atom***. It is assumed that each variable of the head also occurs in some atom of the body.

A rule can be considered as some tool that tells how can we deduce newer and newer ***facts,*** that is tuples, to include in the output relation. If the variables of the rule can be assigned such values that each atom $R_i(u_i)$ is true (that is the appropriate tuple is contained in the relation R_i), then tuple u is added to the relation *ans*. Since all variables of the head occur in some atoms of the body, one never has to consider ***infinite*** domains, since the variables can take values from the actual instance queried. Formally. let $\mathcal{I}$ be an instance over relational schema $\mathbf{R}$, furthermore let q be a the query given by rule (30.3). Let $var(q)$ denote the set of variables occurring in q, and let $dom(\mathcal{I})$ denote the set of constants that occur in $\mathcal{I}$. The ***image of $\mathcal{I}$ under*** q is given by

$$q(\mathcal{I}) = \{\nu(u) | \nu\colon\ var(q) \ \rightarrow\ dom(\mathcal{I}) \text{ and } \nu(u_i) \in R_i \ i = 1, 2, \ldots, n\}. \tag{30.4}$$

An immediate way of calculating $q(\mathcal{I})$ is to consider all possible valuations ν in some order. There are more efficient algorithms, either by equivalent rewriting of the query, or by using some indices.

An important difference between atoms of the body and the head is that relations $R_1, R_2, \ldots, R_n$ are considered given, (physically) stored, while relation *ans* is not, it is thought to be calculated by the help of the rule. This justifies the names: R_i's are ***extensional relations*** and *ans* is ***intensional relation***.

Query q over schema $\mathbf{R}$ is ***monotone,*** if for instances $\mathcal{I}$ and $\mathcal{J}$ over $\mathbf{R}$, $\mathcal{I} \subseteq \mathcal{J}$ implies $q(\mathcal{I}) \subseteq q(\mathcal{J})$. q is ***satisfiable,*** if there exists an instance $\mathcal{I}$, such that $q(\mathcal{I}) \neq \emptyset$. The proof of the next simple observation is left for the Reader (Exercise 30.1-1.).

Proposition 30.4 *Rule based queries are monotone and satisfiable.*

Proposition 30.4 shows the limitations of rule based queries. For example, the query *Which theatres play only Kurosawa films?* is obviously not monotone, hence cannot be expressed by rules of form (30.3).

Tableau queries

If the difference between variables and constants is not considered, then the body of a rule can be viewed as an instance over the schema. This leads to a tabular form of conjunctive queries that is most similar to the visual queries (QBE: Query By Example) of database management system Microsoft Access.

Definition 30.5 *A **tableau** over the schema $\mathbf{R}$ is a generalisation of an instance over $\mathbf{R}$, in the sense that variables may occur in the tuples besides constants. The pair $(\mathbf{T}, u)$ is a **tableau query** if $\mathbf{T}$ is a tableau and u is a free tuple such that all variables of u occur in $\mathbf{T}$, as well. The free tuple u is the **summary**.*

The summary row u of tableau query $(\mathbf{T}, u)$ shows which tuples form the result of the query. The essence of the procedure is that the pattern given by tableau $\mathbf{T}$ is searched for in the database, and if found then the tuple corresponding to is included in the output relation. More precisely, the mapping $\nu\colon\ var(\mathbf{T})\ \rightarrow\ dom(\mathcal{I})$ is an ***embedding*** of tableau $(\mathbf{T}, u)$ into instance $\mathcal{I}$, if $\nu(\mathbf{T}) \subseteq \mathcal{I}$. The output relation of tableau query $(\mathbf{T}, u)$ consists of all tuples $\nu(u)$ that ν is an embedding of tableau $(\mathbf{T}, u)$ into instance $\mathcal{I}$.

Example 30.2 *Tableau query* Let **T** be the following tableau.

Film	*Title*	*Director*	*Actor*
	x_T	"Akira Kurosawa"	x_A

Theater	*Theater*	*Address*	*Phone*
	x_{Th}	x_{Ad}	x_P

Show	*Theater*	*Title*	*Time*
	x_{Th}	x_T	x_{Ti}

The tableau query $(\mathbf{T}, \langle Theater\colon x_{Th}, Address\colon x_{Ad}\rangle)$ answers question 30.2. of the introduction.

The syntax of tableau queries is similar to that of rule based queries. It will be useful later that conditions for one query to contain another one can be easily formulated in the language of tableau queries.

Relational algebra*

A database consists of relations, and a relation is a set of tuples. The result of a query is also a relation with a given attribute set. It is a natural idea that output of a query could be expressed by algebraic and other operations on relations. The ***relational algebra**** consists of the following operations.[1]

Selection: It is of form either $\sigma_{A=c}$ or $\sigma_{A=B}$, where A and B are attributes while c is a constant. The operation can be applied to all such relations R that has attribute A (and B), and its result is relation ans that has the same set of attributes as R has, and consists of all tuples that satisfy the ***selection condition***.

Projection: The form of the operation is $\pi_{A_1,A_2,\ldots,A_n}$, $n \geq 0$, where A_i's are distinct attributes. It can be applied to all such relations whose attribute set includes each A_i and its result is the relation ans that has attribute set $\{A_1, A_2, \ldots, A_n\}$,

$$val = \{t[A_1, A_2, \ldots, A_n] | t \in R\} ,$$

that is it consists of the restrictions of tuples in R to the attribute set $\{A_1, A_2, \ldots, A_n\}$.

Natural join: This operation has been defined earlier in chapter "Relational database design". Its notation is $\bowtie$, its input consists of two (or more) relations R_1, R_2, with attribute sets V_1, V_2, respectively. The attribute set of the output relation is $V_1 \cup V_2$.

$$R_1 \bowtie R_2 = \{t \text{ tuple over } V_1 \cup V_2 | \exists v \in R_1, \exists w \in R_2, t[V_1] = v \text{ és } t[V_2] = w\} .$$

Renaming: Attribute renaming is nothing else, but an injective mapping from a finite set of attributes U into the set of all attributes. Attribute renaming f can be given by the list of pairs $(A, f(A))$, where $A \neq f(A)$, which is written usually in the form $A_1A_2 \ldots A_n \rightarrow B_1B_2 \ldots B_n$. The ***renaming operator*** δ_f maps from inputs over U to outputs over$f[U]$. If R is a relation over U, then

$$\delta_f(R) = \{v \text{ over } f[U] | \exists u \in R, v(f(A)) = u(A), \ \forall A \in U\} .$$

[1]The relational algebra* is the ***monotone*** part of the (full) relational algebra introduced later.

Relational algebra* queries are obtained by finitely many applications of the operations above from ***relational algebra base queries,*** which are

Input relation: R.

Single constant: $\{\langle A : a \rangle\}$, where a is a constant, A is an attribute name.

Example 30.3 *Relational algebra* query.* The question 30.2. of the introduction can be expressed with relational algebra operations as follows.

$$\pi_{Theater,Address}\left((\sigma_{Director=\text{“Akira Kurosawa”}}(Film) \bowtie Show) \bowtie Theater\right) .$$

The mapping given by a relational algebra* query can be easily defined via induction on the operation tree. It is easy to see (Exercise 30.1-2.) that non-satisfiable queries can be given using relational algebra*. There exist no rule based or tableau query equivalent with such a non-satisfiable query. Nevertheless, the following is true.

Theorem 30.6 *Rule based queries, tableau queries and satisfiable relational algebra* are equivalent query languages.*

The proof of Theorem 30.6 consists of three main parts:

1. Rule based $\equiv$ Tableau
2. Satisfiable relational algebra* $\sqsubseteq$ Rule based
3. Rule based $\sqsubseteq$ Satisfiable relational algebra*

The first (easy) step is left to the Reader (Exercise 30.1-3.). For the second step, it has to be seen first, that the language of rule based queries is closed under composition. More precisely, let $\mathbf{R} = \{R_1, R_2, \ldots, R_n\}$ be a database, q be a query over $\mathbf{R}$. If the output relation of q is S_1, then in a subsequent query S_1 can be used in the same way as any extensional relation of $\mathbf{R}$. Thus relation S_2 can be defined, then with its help relation S_3 can be defined, and so on. Relations S_i are ***intensional*** relations. The ***conjunctive query program*** P is a list of rules

$$\begin{array}{rcl} S_1(u_1) & \leftarrow & body_1 \\ S_2(u_2) & \leftarrow & body_2 \\ & \vdots & \\ S_m(u_m) & \leftarrow & body_m , \end{array} \tag{30.5}$$

where S_i's are pairwise distinct and not contained in $\mathbf{R}$. In rule body $body_i$ only relations $R_1, R_2, \ldots R_n$ and $S_1, S_2, \ldots, S_{i-1}$ can occur. S_m is considered to be the output relation of P, its evaluation is is done by computing the results of the rules one-by-one in order. It is not hard to see that with appropriate renaming the variables P can be substituted by a single rule, as it is shown in the following example.

Example 30.4 *Conjunctive query program.* Let $\mathbf{R} = \{Q, R\}$, and consider the following conjunctive query program

$$\begin{array}{rcl} S_1(x,z) & \leftarrow & Q(x,y), R(y,z,w) \\ S_2(x,y,z) & \leftarrow & S_1(x,w), R(w,y,v), S_1(v,z) \\ S_3(x,z) & \leftarrow & S_2(x,u,v), Q(v,z). \end{array} \tag{30.6}$$

S_2 can be written using Q and R only by the first two rules of (30.6)

$$S_2(x,y,z) \leftarrow Q(x,y_1), R(y_1,w,w_1), R(w,y,v), Q(v,y_2), R(y_2,z,w_2). \tag{30.7}$$

It is apparent that some variables had to be renamed to avoid unwanted interaction of rule bodies. Substituting expression (30.7) into the third rule of (30.6) in place of S_2, and appropriately renaming the variables

$$S_3(x,z) \leftarrow Q(x,y_1), R(y_1,w,w_1), R(w,u,v_1), Q(v_1,y_2), R(y_2,v,w_2), Q(v,z). \tag{30.8}$$

is obtained.

Thus it is enough to realise each single relational algebra* operation by an appropriate rule.

$P \bowtie Q$: Let $\overrightarrow{x}$ denote the list of variables (and constants) corresponding to the common attributes of P and Q, let $\overrightarrow{y}$ denote the variables (and constants) corresponding to the attributes occurring only in P, while $\overrightarrow{z}$ denotes those of corresponding to Q's own attributes. Then rule $ans(\overrightarrow{x},\overrightarrow{y},\overrightarrow{z}) \leftarrow P(\overrightarrow{x},\overrightarrow{y}), Q(\overrightarrow{x},\overrightarrow{z})$ gives exactly relation $P \bowtie Q$.

$\sigma_F(R)$: Assume that $R = R(A_1,A_2,\ldots,A_n)$ and the selection condition F is of form either $A_i = a$ or $A_i = A_j$, where A_i, A_j are attributes a is constant. Then

$$ans(x_1,\ldots,x_{i-1},a,x_{i+1},\ldots,x_n) \leftarrow R(x_1,\ldots,x_{i-1},a,x_{i+1},\ldots,x_n) ,$$

respectively,

$$ans(x_1,\ldots,x_{i-1},y,x_{i+1},\ldots,x_{j-1},y,x_{j+1},\ldots,x_n) \leftarrow \\ R(x_1,\ldots,x_{i-1},y,x_{i+1},\ldots,x_{j-1},y,x_{j+1},\ldots,x_n)$$

are the rules sought. The satisfiability of relational algebra* query is used here. Indeed, during composition of operations we never obtain an expression where two distinct constants should be equated.

$\pi_{A_{i_1},A_{i_2},\ldots,A_{i_m}}(R)$: If $R = R(A_1,A_2,\ldots,A_n)$, then

$$ans(x_{i_1},x_{i_2},\ldots,x_{i_m}) \leftarrow R(x_1,x_2,\ldots,x_n)$$

works.

$A_1A_2\ldots A_n \rightarrow B_1B_2\ldots B_n$: The renaming operation of relational algebra* can be achieved by renaming the appropriate variables, as it was shown in Example 30.4..

For the proof of the third step let us consider rule

$$ans(\overrightarrow{x}) \leftarrow R_1(\overrightarrow{x_1}), R_2(\overrightarrow{x_2}), \ldots, R_n(\overrightarrow{x_n}) \tag{30.9}$$

By renaming the attributes of relations R_i's, we may assume without loss of generality that all attribute names are distinct. Then $R = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$ can be constructed that is really a direct product, since the the attribute names are distinct. The constants and multiple occurrences of variables of rule (30.9) can be simulated by appropriate selection operators. The final result is obtained by projecting to the set of attributes corresponding to the variables of relation ans.

30.1.2. Extensions

Conjunctive queries are a class of query languages that has many good properties. However, the set of expressible questions are rather narrow. Consider the following.

30.4. List those pairs where one member directed the other member in a film, and vice versa, the other member also directed the first in a film.

30.5. Which theatres show "La Dolce Vita" or "Rashomon"?

30.6. Which are those films of Hitchcock that Hitchcock did not play a part in?

30.7. List those films whose every actor played in some film of Fellini.

30.8. Let us recall the game "Chain-of-Actors". The first player names an actor/actress, the next another one who played in some film together with the first named. This is continued like that, always a new actor/actress has to be named who played together with the previous one. The winner is that player who could continue the chain last time. List those actors/actresses who could be reached by "Chain-of-Actors" starting with "Marcello Mastroianni".

Equality atoms

Question **30.4.** can be easily answered if equalities are also allowed rule bodies, besides relational atoms:

$$ans(y_1, y_2) \leftarrow Film(x_1, y_1, z_1), Film(x_2, y_2, z_2), y_1 = z_2, y_2 = z_1 . \tag{30.10}$$

Allowing equalities raises two problems. First, the result of the query could become infinite. For example, the rule based query

$$ans(x, y) \leftarrow R(x), y = z \tag{30.11}$$

results in an infinite number of tuples, since variables y and z are not bounded by relation R, thus there can be an infinite number of evaluations that satisfy the rule body. Hence, the concept of ***domain restricted*** query is introduced. Rule based query q is ***domain restricted,*** if all variables that occur in the rule body also occur in some relational atom.

The second problem is that equality atoms may cause the body of a rule become unsatisfiable, in contrast to Proposition 30.4. For example, query

$$ans(x) \leftarrow R(x), x = a, x = b \tag{30.12}$$

is domain restricted, however if a and b are distinct constants, then the answer will be empty. It is easy to check whether a rule based query with equality atoms is satisfiable.

```
SATISFIABLE($q$)
1  Compute the transitive closure of equalities of the body of $q$.
2  if Two distinct constants should be equal by transitivity
3    then return "Not satisfiable."
4    else return "Satisfiable."
```

It is also true (Exercise 30.1-4.) that if a rule based query q that contains equality atoms is satisfiable, then there exists a another rule based query q' without equalities that is equivalent with q.

Disjunction – union

The question **30.5**. cannot be expressed with conjunctive queries. However, if the union operator is added to relational algebra, then **30.5**. can be expressed in that extended relational algebra:

$$\pi_{Theater}\,(\sigma_{Title=\text{“La Dolce Vita”}}(Show) \cup \sigma_{Title=\text{“ Rashomon”}}(Show)) \ . \tag{30.13}$$

Rule based queries are also capable of expressing question **30.5**. if it is allowed that the same relation is in the head of many distinct rules:

$$\begin{aligned} ans(x_M) &\leftarrow Show(x_{Th}, \text{“La Dolce Vita”}, x_{Ti}) \ , \\ ans(x_M) &\leftarrow Show(x_{Th}, \text{“Rashomon”}, x_{Ti}) \ . \end{aligned} \tag{30.14}$$

Non-recursive datalog program is a generalisation of this.

Definition 30.7 *A **non-recursive datalog program** over schema* **R** *is a set of rules*

$$\begin{array}{rcl} S_1(u_1) & \leftarrow & body_1 \\ S_2(u_2) & \leftarrow & body_2 \\ & \vdots & \\ S_m(u_m) & \leftarrow & body_m \ , \end{array} \tag{30.15}$$

where no relation of **R** *occurs in a head, the same relation may occur in the head of several rules, furthermore there exists an ordering $r_1, r_2, \ldots, r_m$ of the rules such that the relation in the head of r_i does not occur in the body of any rule r_j for $j \leq i$.*

The semantics of the non-recursive datalog program (30.15) is similar to the conjunctive query program (30.5). The rules are evaluated in the order $r_1, r_2, \ldots, r_m$ of Definition 30.7, and if a relation occurs in more than one head then the union of the sets of tuples given by those rules is taken.

The union of tableau queries $(\mathbf{T}_i, u)$ $i = 1, 2, \ldots, n$ is denoted by $(\{\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_n\}, u)$. It is evaluated by individually computing the result of each tableau query $(\mathbf{T}_i, u)$, then the union of them is taken. The following holds.

Theorem 30.8 *The language of non-recursive datalog programs with unique output relation and the relational algebra extended with the union operator are equivalent.*

The proof of Theorem 30.8 is similar to that of Theorem 30.6 and it is left to the Reader (Exercise 30.1-5.). Let us note that the expressive power of the union of tableau queries is weaker. This is caused by the requirement having the same summary row for each tableau. For example, the non-recursive datalog program query

$$\begin{aligned} ans(a) &\leftarrow \\ ans(b) &\leftarrow \end{aligned} \tag{30.16}$$

cannot be realised as union of tableau queries.

Negation

The query **30.6**. is obviously not monotone. Indeed, suppose that in relation *Film* there exist tuples about Hitchcock's film *Psycho*, for example (“Psycho”,”A. Hitchcock”,”A. Perkins”), (“Psycho”,”A. Hitchcock”,”J. Leigh”), . . . , however, the tuple (“Psycho”,”A. Hitchcock”,”A. Hitchcock”) is not included. Then the tuple (“Psycho”) occurs in the output of query **30.6**. With some effort one can realize however, that Hitchcock appears in the film *Psycho*, as “a man in cowboy hat”. If the tuple (“Psycho”,”A. Hitchcock”,”A. Hitchcock”) is added to relation *Film* as a consequence, then the instance of schema **CinePest** gets larger, but the output of query **30.6**. becomes smaller.

It is not too hard to see that the query languages discussed so far are monotone, hence query **30.6**. cannot be formulated with non-recursive datalog program or with some of its equivalents. Nevertheless, if the difference (−)operator is also added to relation algebra, then it becomes capable of expressing queries of type **30.6**. For example,

$$\pi_{Title}\,(\sigma_{Director=\text{“A. Hitchcock”}}(Film)) - \pi_{Title}\,(\sigma_{Actor=\text{“A. Hitchcock”}}(Film)) \tag{30.17}$$

realises exactly query **30.6**. Hence, the (full) relational algebra consists of operations $\{\sigma, \pi, \bowtie, \delta, \cup, -\}$. The importance of the relational algebra is shown by the fact, that Codd calls a query language Q ***relationally complete,*** exactly if for all relational algebra query q there exists $q' \in Q$, such that $q \equiv q'$.

If ***negative literals***, that is atoms of the form $\neg R(u)$ are also allowed in rule bodies, then the obtained ***non-recursive datalog with negation,*** in notation ***nr-datalog***$^\neg$ is relationally complete.

Definition 30.9 *A non-recursive datalog*$^\neg$ *(nr-datalog*$^\neg$*) rule is of form*

$$q: \quad S(u) \leftarrow L_1, L_2, \ldots, L_n\ , \tag{30.18}$$

where S is a relation, u is a free tuple, L_i's are ***literals,****, that is expression of form $R(v)$ or $\neg R(v)$, such that v is a free tuple for $i = 1, 2, \ldots, n$. S does not occur in the body of the rule. The rule is* ***domain restricted,*** *if each variable x that occurs in the rule also occurs in a* ***positive literal*** *(expression of the form $R(v)$) of the body. Every nr-datalog*$^\neg$ *rule is considered domain restricted, unless it is specified otherwise.*

The semantics of rule (30.18) is as follows. Let **R** be a relational schema that contains all relations occurring in the body of q, furthermore, let $\mathcal{I}$ be an instance over **R**. The ***image of*** $\mathcal{I}$ ***under*** q is

$$\begin{aligned} q(\mathcal{I}) = \{\nu(u) |\ \nu \text{ is an valuation of the variables and for } i = 1, 2, \ldots, n \\ \nu(u_i) \in \mathcal{I}(R_i),\ \text{if } L_i = R_i(u_i) \text{ and} \\ \nu(u_i) \notin \mathcal{I}(R_i),\ \text{if } L_i = \neg R_i(u_i)\}. \end{aligned} \tag{30.19}$$

A ***nr-datalog***$^\neg$ ***program*** over schema **R** is a collection of nr-datalog$^\neg$ rules

$$\begin{array}{rcl} S_1(u_1) & \leftarrow & body_1 \\ S_2(u_2) & \leftarrow & body_2 \\ & \vdots & \\ S_m(u_m) & \leftarrow & body_m\ , \end{array} \tag{30.20}$$

where relations of schema **R** do not occur in heads of rules, the same relation may appear in more than one rule head, furthermore there exists an ordering $r_1, r_2, \ldots, r_m$ of the rules such that the relation of the head of rule r_i does not occur in the head of any rule r_j if $j \leq i$.

The computation of the result of nr-datalog$^\neg$ program (30.20) applied to instance $\mathcal{I}$

over schema **R** can be done in the same way as the evaluation of non-recursive datalog program (30.15), with the difference that the individual nr-datalog$^\neg$ rules should be interpreted according to (30.19).

Example 30.5 *Nr-datalog$^\neg$ program.* Let us assume that all films that are included in relation *Film* have only one director. (It is not always true in real life!) The nr-datalog$^\neg$ rule

$$ans(x) \leftarrow Film(x, \text{“A. Hitchcock”}, z), \neg Film(x, \text{“A. Hitchcock”}, \text{“A. Hitchcock”}) \tag{30.21}$$

expresses query **30.6**. Query **30.7.** is realised by the nr-datalog$^\neg$ program

$$\begin{array}{rcl} Fellini\text{-}actor(z) & \leftarrow & Film(x, \text{”F. Fellini”}, z) \\ Not\text{-}the\text{-}answer(x) & \leftarrow & Film(x, y, z), \neg Fellini\text{-}actor(z) \\ Answer(x) & \leftarrow & Film(x, y, z), \neg Not\text{-}the\text{-}answer(x)\ . \end{array} \tag{30.22}$$

One has to be careful in writing nr-datalog$^\neg$ programs. If the first two rules of program (30.22) were to be merged like in Example 30.4.

$$\begin{array}{rcl} Bad\text{-}not\text{-}ans(x) & \leftarrow & Film(x, y, z), \neg Film(x', \text{”F. Fellini”}, z), Film(x', \text{”F. Fellini”}, z') \\ Answer(x) & \leftarrow & Film(x, y, z), \neg Bad\text{-}not\text{-}ans(x), \end{array} \tag{30.23}$$

then (30.23) answers the following query (assuming that all films have unique director)

30.9. List all those films whose every actor played in ***each*** film of Fellini,

instead of query **30.7.**

It is easy to see that every satisfiable nr-datalog$^\neg$ program that contains equality atoms can be replaced by one without equalities. Furthermore the following proposition is true, as well.

Proposition 30.10 *The satisfiable (full) relational algebra and the nr-datalog$^\neg$ programs with single output relation are equivalent query languages.*

Recursion

Query **30.8.** cannot be formulated using the query languages introduced so far. Some ***a priori*** information would be needed about how long a *chain-of-actors* could be formed starting with a given actor/actress. Let us assume that the maximum length of a chain-of-actors starting from “Marcello Mastroianni” is 117. (It would be interesting to know the ***real*** value!) Then, the following non-recursive datalog program gives the answer.

$$\begin{array}{rcl} Film\text{-}partner(z_1, z_2) & \leftarrow & Film(x, y, z_1), Film(x, y, z_2), z_1 < z_2\text{[2]} \\ Partial\text{-}answer_1(z) & \leftarrow & Film\text{-}partner(z, \text{“Marcello Mastroianni”}) \\ Partial\text{-}answer_1(z) & \leftarrow & Film\text{-}partner(\text{“Marcello Mastroianni”}, z) \\ Partial\text{-}answer_2(z) & \leftarrow & Film\text{-}partner(z, y), Partial\text{-}answer_1(y) \\ Partial\text{-}answer_2(z) & \leftarrow & Film\text{-}partner(y, z), Partial\text{-}answer_1(y) \\ \vdots & & \vdots \\ Partial\text{-}answer_{117}(z) & \leftarrow & Film\text{-}partner(z, y), Partial\text{-}answer_{116}(y) \\ Partial\text{-}answer_{117}(z) & \leftarrow & Film\text{-}partner(y, z), Partial\text{-}answer_{116}(y) \\ Mastroianni\text{-}chain(z) & \leftarrow & Partial\text{-}answer_1(z) \\ Mastroianni\text{-}chain(z) & \leftarrow & Partial\text{-}answer_2(z) \\ \vdots & & \vdots \\ Mastroianni\text{-}chain(z) & \leftarrow & Partial\text{-}answer_{117}(z) \end{array} \tag{30.24}$$

It is much easier to express query **30.8.** using ***recursion***. In fact, the ***transitive closure*** of the graph *Film-partner* needs to be calculated. For the sake of simplicity the definition of *Film-partner* is changed a little (thus approximately doubling the storage requirement).

$$\begin{array}{lcl} \mathit{Film\text{-}partner}(z_1,z_2) & \leftarrow & \mathit{Film}(x,y,z_1), \mathit{Film}(x,y,z_2) \\ \mathit{Chain\text{-}partner}(x,y) & \leftarrow & \mathit{Film\text{-}partner}(x,y) \\ \mathit{Chain\text{-}partner}(x,y) & \leftarrow & \mathit{Film\text{-}partner}(x,z), \mathit{Chain\text{-}partner}(z,y)\ . \end{array} \tag{30.25}$$

The datalog program (30.25) is ***recursive,*** since the definition of relation *Chain-partner* uses the relation itself. Let us suppose for a moment that this is meaningful, then query **30.8.** is answered by rule

$$\mathit{Mastroianni\text{-}chain}(x) \leftarrow \mathit{Chain\text{-}partner}(x, \text{“Marcello Mastroianni”}) \tag{30.26}$$

Definition 30.11 *The expression*

$$R_1(u_1) \leftarrow R_2(u_2), R_3(u_3), \ldots, R_n(u_n) \tag{30.27}$$

*is a **datalog rule,** if $n \geq 1$, the R_i's are relation names, the u_i's are free tuples of appropriate length. Every variable of u_1 has to occur in one of $u_2, \ldots u_n$, as well. The head of the rule is $R_1(u_1)$, the body of the rule is $R_2(u_2), R_3(u_3), \ldots, R_n(u_n)$. A **datalog program** is a finite collection of rules of type (30.27). Let P be a datalog program. The relation R occurring in P is **extensional** if it occurs in only rule bodies, and it is **intensional** if it occurs in the head of some rule.*

If ν is a valuation of the variables of rule (30.27), then $R_1(\nu(u_1)) \leftarrow R_2(\nu(u_2)), R_3(\nu(u_3)), \ldots, R_n(\nu(u_n))$ is a ***realisation*** of rule (30.27). The ***extensional (database) schema*** of P consists of the extensional relations of P, in notation $edb(P)$. The ***intensional schema*** of P, in notation $idb(P)$ is defined similarly as consisting of the intensional relations of P. Let $sch(P) = edb(P) \cup idb(P)$. The semantics of datalog program P is a mapping from the set of instances over $edb(P)$ to the set of instances over $idb(P)$. This can be defined proof theoretically, model theoretically or as a fixpoint of some operator. This latter one is equivalent with the first two, so to save space only the fixpoint theoretical definition is discussed.

There are no negative literals used in Definition 30.11. The main reason of this is that recursion and negation together may be meaningless, or contradictory. Nevertheless, sometimes negative atoms might be necessary. In those cases the semantics of the program will be defined specially.

Fixpoint semantics

Let P be a datalog program, $\mathcal{K}$ be an instance over $sch(P)$. ***Fact*** A, that is a tuple consisting of constants is an ***immediate consequence*** of $\mathcal{K}$ and P, if either $A \in \mathcal{K}(R)$ for some relation $R \in sch(P)$, or $A \leftarrow A_1, A_2, \ldots, A_n$ is a realisation of a rule in P and each A_i is in $\mathcal{K}$. The ***immediate consequence operator*** T_P is a mapping from the set of instances over $sch(P)$ to itself. $T_P(\mathcal{K})$ consists of all immediate consequences of $\mathcal{K}$ and P.

[2]Arbitrary comparison atoms can be used, as well, similarly to equality atoms. Here $z_1 < z_2$ makes it sure that all pairs occur at most once in the list.

Proposition 30.12 *The immediate consequence operator T_P is monotone.*

Proof. Let $\mathcal{I}$ and $\mathcal{J}$ be instances over $sch(P)$, such that $\mathcal{I} \subseteq \mathcal{J}$. Let A be a fact of$T_P(\mathcal{I})$. If $A \in \mathcal{I}(R)$ for some relation $R \in sch(P)$, then $A \in \mathcal{J}(R)$ is implied by $\mathcal{I} \subseteq \mathcal{J}$. on the other hand, if $A \leftarrow A_1, A_2, \ldots, A_n$ is a realisation of a rule in P and each A_i is in $\mathcal{I}$, then $A_i \in \mathcal{J}$ also holds. ■

The definition of T_P implies that $\mathcal{K} \subseteq T_P(\mathcal{K})$. Using Proposition 30.12 it follows that

$$\mathcal{K} \subseteq T_P(\mathcal{K}) \subseteq T_P(T_P(\mathcal{K})) \subseteq \ldots \ . \tag{30.28}$$

Theorem 30.13 *For every instance $\mathcal{I}$ over schema $sch(P)$ there exists a unique minimal instance $\mathcal{I} \subseteq \mathcal{K}$ that is a **fixpoint** of T_P, i.e. $\mathcal{K} = T_P(\mathcal{K})$.*

Proof. Let $T_P^i(\mathcal{I})$ denote the consecutive application of operator T_P i-times, and let

$$\mathcal{K} = \bigcup_{i=0}^{\infty} T_P^i(\mathcal{I}) \ . \tag{30.29}$$

By the monotonicity of T_P and (30.29) we have

$$T_P(\mathcal{K}) = \bigcup_{i=1}^{\infty} T_P^i(\mathcal{I}) \subseteq \bigcup_{i=0}^{\infty} T_P^i(\mathcal{I}) = \mathcal{K} \subseteq T_P(\mathcal{K}) \ , \tag{30.30}$$

that is $\mathcal{K}$ is a fixpoint. It is easy to see that every fixpoint that contains $\mathcal{I}$, also contains $T_P^i(\mathcal{I})$ for all $i = 1, 2, \ldots,$ that is it contains $\mathcal{K}$, as well. ■

Definition 30.14 *The **result** of datalog program P on instance $\mathcal{I}$ over $edb(P)$ is the unique minimal fixpoint of T_P containing $\mathcal{I}$, in notation $P(\mathcal{I})$.*

It can be seen, see Exercise 30.1-6., that the chain in (30.28) is finite, that is there exists an n, such that $T_P(T_P^n(\mathcal{I})) = T_P^n(\mathcal{I})$. The naive evaluation of the result of the datalog program is based on this observation.

Naïv-datalog(P,$\mathcal{I}$)

```
1  $\mathcal{K} \leftarrow \mathcal{I}$
2  while $T_P(\mathcal{K}) \neq \mathcal{K}$
3       do $\mathcal{K} \leftarrow T_P(\mathcal{K})$
4  return $\mathcal{K}$
```

Procedure Naïv-datalog is not optimal, of course, since every fact that becomes included in $\mathcal{K}$ is calculated again and again at every further execution of the **while** loop.

The idea of Semi-naiv-datalog is that it tries to use only recently calculated new facts in the **while** loop, as much as it is possible, thus avoiding recomputation of known facts. Consider datalog program P with $edb(P) = \mathbf{R}$, and $idb(P) = \mathbf{T}$. For a rule

$$S(u) \leftarrow R_1(v_1), \ldots, R_n(v_n), T_1(w_1), \ldots, T_m(w_m) \tag{30.31}$$

of P where $R_k \in \mathbf{R}$ and $T_j \in \mathbf{T}$, the following rules are constructed for $j = 1, 2, \ldots, m$ and $i \geq 1$

$$\begin{aligned} temp_S^{i+1}(u) \;\leftarrow\;\; & R_1(v_1), \ldots, R_n(v_n), \\ & T_1^i(w_1), \ldots, T_{j-1}^i(w_{j-1}), \Delta_{T_j}^i(w_j), T_{j+1}^{i-1}(w_{j+1}), \ldots, T_m^{i-1}(w_m)\ . \end{aligned} \tag{30.32}$$

Relation $\Delta_{T_j}^i$ denotes the change of T_j in iteration i. The union of rules corresponding to S in layer i is denoted by P_S^i, that is rules of form (30.32) for $temp_S^{i+1}$, $j = 1, 2, \ldots, m$. Assume that the list of *idb* relations occurring in rules defining the *idb* relation S is $T_1, T_2, \ldots, T_\ell$. Let

$$P_S^i(\mathcal{I}, T_1^{i-1}, \ldots, T_\ell^{i-1}, T_1^i, \ldots, T_\ell^i, \Delta_{T_1}^i, \ldots, \Delta_{T_\ell}^i) \tag{30.33}$$

denote the set of facts (tuples) obtained by applying rules (30.32) to input instance $\mathcal{I}$ and to *idb* relations $T_j^{i-1}, T_j^i, \Delta_{T_j}^i$. The input instance $\mathcal{I}$ is the actual value of the *edb* relations of P.

```
Semi-naiv-datalog($P$,$\mathcal{I}$)
 1  $P' \leftarrow$ those rules of $P$ whose body does not contain idb relation
 2  for $S \in idb(P)$
 3      do $S^0 \leftarrow \emptyset$
 4         $\Delta_S^1 \leftarrow P'(\mathcal{I})(S)$
 5  $i \leftarrow 1$
 6  repeat
 7        for $S \in idb(P)$
 8                            ▷ $T_1, \ldots, T_\ell$ are the idb relations of the rules defining $S$.
 9            do $S^i \leftarrow S^{i-1} \cup \Delta_S^i$
10               $\Delta_S^{i+1} \leftarrow P_S^i(\mathcal{I}, T_1^{i-1}, \ldots, T_\ell^{i-1}, T_1^i, \ldots, T_\ell^i, \Delta_{T_1}^i, \ldots, \Delta_{T_\ell}^i) - S^i$
11        $i \leftarrow i + 1$
12  until $\Delta_S^i = \emptyset$ for all $S \in idb(P)$
13  for $S \in idb(P)$
14      do $S \leftarrow S^i$
15  return $\mathcal{S}$
```

Theorem 30.15 *Procedure* Semi-naiv-datalog *correctly computes the result of program P on instance $\mathcal{I}$.*

Proof. We will show by induction on i that after execution of the loop of lines 6–12 ith times the value of S^i is $T_P^i(\mathcal{I})(S)$, while Δ_S^{i+1} is equal to $T_P^{i+1}(\mathcal{I})(S) - T_P^i(\mathcal{I})(S)$ for arbitrary $S \in idb(P)$. $T_P^i(\mathcal{I})(S)$ is the result obtained for S starting from $\mathcal{I}$ and applying the immediate consequence operator T_P i-times.

For $i = 0$, line 4 calculates exactly $T_P(\mathcal{I})(S)$ for all $S \in idb(P)$. In order to prove the induction step, one only needs to see that $P_S^i(\mathcal{I}, T_1^{i-1}, \ldots, T_\ell^{i-1}, T_1^i, \ldots, T_\ell^i, \Delta_{T_1}^i, \ldots, \Delta_{T_\ell}^i) \cup S^i$ is exactly equal to $T_P^{i+1}(\mathcal{I})(S)$, since in lines 9–10 procedure Semi-naiv-datalog constructs S^i-t and Δ_S^{i+1} using that. The value of S^i is $T_P^i(\mathcal{I})(S)$, by the induction hypothesis. Additional new tuples are obtained only if that for some *idb* relation defining S such tuples are considered that are constructed at the last application of T_P, and these are in relations $\Delta_{T_1}^i, \ldots, \Delta_{T_\ell}^i$, also by the induction hypothesis.

The halting condition of line 12 means exactly that all relations $S \in idb(P)$ are unchanged during the application of the immediate consequence operator T_P, thus the algorithm has found its minimal fixpoint. This latter one is exactly the result of datalog program P on input instance $\mathcal{I}$ according to Definition 30.14. ■

Procedure Semi-naiv-datalog eliminates a large amount of unnecessary calculations, nevertheless it is not optimal on some datalog programs (Exercise gy:snaiv). However, analysis of the datalog program and computation based on that can save most of the unnecessary calculations.

Definition 30.16 *Let P be a datalog program. The* ***precedence graph*** *of P is the directed graph G_P defined as follows. Its vertex set consists of the relations of $idb(P)$, and (R, R') is an arc for $R, R' \in idb(P)$ if there exists a rule in P whose head is R' and whose body contains R. P is* ***recursive,*** *if G_P contains a directed cycle. Relations R and R' are* ***mutually recursive*** *if the belong to the same strongly connected component of G_P.*

Being mutually recursive is an equivalence relation on the set of relations $idb(P)$. The main idea of procedure Improved-semi-naiv-datalog is that for a relation $R \in idb(P)$ only those relations have to be computed "simultaneously" with R that are in its equivalence class, all other relations defining R can be calculated "in advance" and can be considered as *edb* relations.

```
Improved-semi-naiv-datalog($P$,$\mathcal{I}$)

1  Determine the equivalence classes of $idb(P)$ under mutual recursivity.
2  List the equivalence classes $[R_1], [R_2], \ldots, [R_n]$
       according to a topological order of $G_P$.
3                              ▷ There exists no directed path from $R_j$ to $R_i$ in $G_P$ for all $i < j$.
4  for $i \leftarrow 1$ to $n$
5     do Use Semi-naiv-datalog to compute relations of $[R_i]$
         taking relations of $[R_j]$ as edb relations for $j < i$.
```

Lines 1–2 can be executed in time $O(v_{G_P} + e_{G_P})$ using depth first search, where v_{G_P} and e_{G_P} denote the number of vertices and edges of graph G_P, respectively. Proof of correctness of the procedure is left to the Reader (Exercise 30.1-8.).

30.1.3. Complexity of query containment

In the present section we return to conjunctive queries. The costliest task in computing result of a query is to generate the natural join of relations. In particular, if there are no indexes available on the common attributes, hence only procedure Full-tuplewise-join is applicable.

```
Full-tuplewise-join($R_1, R_2$)
1 $S \leftarrow \emptyset$
2 for all $u \in R_1$
3     do for all $v \in R_2$
4         do if $u$ and $v$ can be joined
5             then $S \leftarrow S \cup \{u \bowtie v\}$
6 return $S$
```

It is clear that the running time of Full-tuplewise-join is $O(|R_1| \times |R_2|)$. Thus, it is important that in what order is a query executed, since during computation natural joins of relations of various sizes must be calculated. In case of tableau queries the ***Homomorphism Theorem*** gives a possibility of a query rewriting that uses less joins than the original.

Let q_1, q_2 be queries over schema $\mathbf{R}$. q_2 ***contains*** q_1, in notation $q_1 \sqsubseteq q_2$, if for all instances $\mathcal{I}$ over schema $\mathbf{R}$ $q_1(\mathcal{I}) \subseteq q_2(\mathcal{I})$ holds. $q_1 \equiv q_2$ according to Definition 30.1 iff $q_1 \sqsubseteq q_2$ and $q_1 \sqsupseteq q_2$. A generalisation of valuations will be needed. ***Substitution*** is a mapping from the set of variables to the union of sets of variables and sets of constants that is extended to constants as identity. Extension of substitution to free tuples and tableaux can be defined naturally.

Definition 30.17 *Let $q = (\mathbf{T}, u)$ and $q' = (\mathbf{T}', u')$ be two tableau queries overs schema $\mathbf{R}$. Substitution θ is a **homomorphism** from q' to q, if $\theta(\mathbf{T}') = \mathbf{T}$ and $\theta(u') = u$.*

Theorem 30.18 (Homomorphism Theorem). *Let $q = (\mathbf{T}, u)$ and $q' = (\mathbf{T}', u')$ be two tableau queries overs schema $\mathbf{R}$. $q \sqsubseteq q'$ if and only if, there exists a homomorphism from q' to q.*

Proof. Assume first, that θ is a homomorphism from q' to q, and let $\mathcal{I}$ be an instance over schema $\mathbf{R}$. Let $w \in q(\mathcal{I})$. This holds exactly if there exists a valuation ν that maps tableau $\mathbf{T}$ into $\mathcal{I}$ and $\nu(u) = w$. It is easy to see that $\theta \circ \nu$ maps tableau $\mathbf{T}'$ into $\mathcal{I}$ and $\theta \circ \nu(u') = w$, that is $w \in q'(\mathcal{I})$. Hence, $w \in q(\mathcal{I}) \Longrightarrow w \in q'(\mathcal{I})$, which in turn is equivalent with $q \sqsubseteq q'$.

On the other hand, let us assume that $q \sqsubseteq q'$. The idea of the proof is that both, q and q' are applied to the "instance" $\mathbf{T}$. The output of q is free tuple u, hence the output of q' also contains u, that is there exists a θ embedding of $\mathbf{T}'$ into $\mathbf{T}$ that maps u' to u. To formalise this argument the instance $\mathcal{I}_{\mathbf{T}}$ isomorphic to $\mathbf{T}$ is constructed.

Let V be the set of variables occurring in $\mathbf{T}$. For all $x \in V$ let a_x be constant that differs from all constants appearing in $\mathbf{T}$ or $\mathbf{T}'$, furthermore $x \neq x' \Longrightarrow a_x \neq a_{x'}$. Let μ be the valuation that maps $x \in V$ to a_x, furthermore let $\mathcal{I}_{\mathbf{T}} = \mu(\mathbf{T})$. μ is a bijection from V to $\mu(V)$ and there are no constants of $\mathbf{T}$ appearing in $\mu(V)$, hence μ^{-1} well defined on the constants occurring in $\mathcal{I}_{\mathbf{T}}$.

It is clear that $\mu(u) \in q(\mathcal{I}_{\mathbf{T}})$, thus using $q \sqsubseteq q'$ $\mu(u) \in q'(\mathcal{I}_{\mathbf{T}})$ is obtained. That is, there exists a valuation ν that embeds tableau $\mathbf{T}'$ into $\mathcal{I}_{\mathbf{T}}$, such that $\nu(u') = \mu(u)$. It is not hard to see that $\nu \circ \mu^{-1}$ is a homomorphism from q' to q. ■

Query optimisation by tableau minimisation

According to Theorem 30.6 tableau queries and satisfiable relational algebra (without subtraction) are equivalent. The proof shows that the relational algebra expression equivalent with a tableau query is of form $\pi_{\vec{a}}(\sigma_F(R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k))$, where k is the number of rows of the tableau. It implies that if the number of joins is to be minimised, then the number of rows of an equivalent tableau must be minimised.

The tableau query $(\mathbf{T}, u)$ is ***minimal,*** if there exists no tableau query $(\mathbf{S}, v)$ that is equivalent with $(\mathbf{T}, u)$ and $|\mathbf{S}| < |\mathbf{T}|$, that is S has fewer rows. It may be surprising, but it is true, that a minimal tableau query equivalent with $(\mathbf{T}, u)$ can be obtained by simply dropping some rows from $\mathbf{T}$.

Theorem 30.19 *Let $q = (\mathbf{T}, u)$ be a tableau query. There exists a subset $\mathbf{T}'$ of $\mathbf{T}$, such that query $q' = (\mathbf{T}', u)$ is minimal and equivalent with $q = (\mathbf{T}, u)$.*

Proof. Let $(\mathbf{S}, v)$ be a minimal query equivalent with q. According to the Homomorphism Theorem there exist homomorphisms θ from q to $(\mathbf{S}, v)$, and λ from $(\mathbf{S}, v)$ to q. Let $\mathbf{T}' = \theta \circ \lambda(\mathbf{T})$. It is easy to check that $(\mathbf{T}', u) \equiv q$ and $|\mathbf{T}'| \leq |\mathbf{S}|$. But $(\mathbf{S}, v)$ is minimal, hence $(\mathbf{T}', u)$ is minimal, as well. ■

Example 30.6 *Application of tableau minimisation* Consider the relational algebra expression

$$q = \pi_{AB}\,(\sigma_{B=5}(R)) \bowtie \pi_{BC}\,(\pi_{AB}(R) \bowtie \pi_{AC}\,(\sigma_{B=5}(R))) \tag{30.34}$$

over the schema $\mathbf{R}$ of attribute set $\{A, B, C\}$. The tableau query corresponding to q is the following tableau $\mathbf{T}$:

R	A	B	C
	x	5	z_1
	x_1	5	z_2
	x_1	5	z
u	x	5	z

(30.35)

Such a homomorphism is sought that maps some rows of $\mathbf{T}$ to some other rows of $\mathbf{T}$, thus sort of "folding" the tableau. The first row cannot be eliminated, since the homomorphism is an identity on free tuple u, thus x must be mapped to itself. The situation is similar with the third row, as the image of z is itself under any homomorphism. However the second row can be eliminated by mapping x_1 to x and z_2 to z, respectively. Thus, the minimal tableau equivalent with $\mathbf{T}$ consists of the first and third rows of $\mathbf{T}$. Transforming back to relational algebra expression,

$$\pi_{AB}(\sigma_{B=5}(R)) \bowtie \pi_{BC}(\sigma_{B=5}(R)) \tag{30.36}$$

is obtained. Query (30.36) contains one less join operator than query (30.34).

The next theorem states that the question of tableau containment and equivalence is NP-complete, hence tableau minimisation is an NP-hard problem.

Theorem 30.20 *For given tableau queries q and q' the following decision problems are NP-complete:*

30.10. $q \sqsubseteq q'$?

30.11. $q \equiv q'$?

30.12. Assume that q' is obtained from q by removing some free tuples. Is it true then that $q \equiv q'$?

Proof. The Exact cover problem will be reduced to the various tableau problems. The input of Exact cover problem is a finite set $X = \{x_1, x_2, \ldots, x_n\}$, and a collection of its subsets

$\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$. It has to be determined whether there exists $\mathcal{S}' \sqsubseteq \mathcal{S}$, such that subsets in $\mathcal{S}'$ cover X exactly (that is, for all $x \in X$ there exists exactly one $S \in \mathcal{S}'$ such that $x \in S$). Exact cover is known to be an NP-complete problem.

Let $\mathcal{E} = (X, \mathcal{S})$ be an input of Exact cover. A construction is sketched that produces a pair $q_{\mathcal{E}}, q'_{\mathcal{E}}$ of tableau queries to $\mathcal{E}$ in polynomial time. This construction can be used to prove the various NP-completeness results.

Let the schema $\mathbf{R}$ consist of the pairwise distinct attributes $A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m$. $q_{\mathcal{E}} = (\mathbf{T}_{\mathcal{E}}, t)$ and $q'_{\mathcal{E}} = (\mathbf{T}'_{\mathcal{E}}, t)$ are tableau queries over schema $\mathbf{R}$ such that the summary row of both of them is free tuple $t = \langle A_1 : a_1, A_2 : a_2, \ldots, A_n : a_n \rangle$, where $a_1, a_2, \ldots, a_n$ are pairwise distinct variables.

Let $b_1, b_2, \ldots, b_m, c_1, c_2, \ldots c_m$ be another set of pairwise distinct variables. Tableau $\mathbf{T}_{\mathcal{E}}$ consists of n rows, for each element of X corresponds one. a_i stands in column of attribute A_i in the row of x_i, while b_j stands in column of attribute B_j for all such j that $x_i \in S_j$ holds. In other positions of tableau $\mathbf{T}_{\mathcal{E}}$ pairwise distinct new variables stand.

Similarly, $\mathbf{T}'_{\mathcal{E}}$ consists of m rows, one corresponding to each element of $\mathcal{S}$. a_i stands in column of attribute A_i in the row of S_j for all such i that $x_i \in S_j$, furthermore $c_{j'}$ stands in the column of attribute $B_{j'}$, for all $j' \neq j$. In other positions of tableau $\mathbf{T}'_{\mathcal{E}}$ pairwise distinct new variables stand.

The NP-completeness of problem 30.10. follows from that X has an exact cover using sets of $\mathcal{S}$ if and only if $q'_{\mathcal{E}} \sqsubseteq q_{\mathcal{E}}$ holds. The proof, and the proof of the NP-completeness of problems 30.11. and 30.12. are left to the Reader (Exercise 30.1-9.). ∎

Exercises

30.1-1 Prove Proposition 30.4, that is every rule based query q is monotone and satisfiable. *Hint.* For the proof of satisfiability let K be the set of constants occurring in query q, and let $a \notin K$ be another constant. For every relation schema R_i in rule (30.3) construct all tuples $(a_1, a_2, \ldots, a_r)$, where $a_i \in K \cup \{a\}$, and r is the arity of R_i. Let $\mathcal{I}$ be the instance obtained so. Prove that $q(\mathcal{I})$ is nonempty.

30.1-2 Give a relational schema $\mathbf{R}$ and a relational algebra query q over schema $\mathbf{R}$, whose result is empty to arbitrary instance over $\mathbf{R}$.

30.1-3 Prove that the languages of rule based queries and tableau queries are equivalent.

30.1-4 Prove that every rule based query q with equality atoms is either equivalent with the empty query $q^{\emptyset}$, or there exists a rule based query q' without equality atoms such that $q \equiv q'$. Give a polynomial time algorithm that determines for a rule based query q with equality atoms whether $q \equiv q^{\emptyset}$ holds, and if not, then constructs a rule based query q' without equality atoms, such that $q \equiv q'$.

30.1-5 Prove Theorem 30.8 by generalising the proof idea of Theorem 30.6.

30.1-6 Let P be a datalog program, $\mathcal{I}$ be an instance over $edb(P)$, $inC(P, \mathcal{I})$ be the (finite) set of constants occurring in $\mathcal{I}$ and P. Let $\mathbf{B}(P, \mathcal{I})$ be the following instance over $sch(P)$:

1. For every relation R of $edb(P)$ the fact $R(u)$ is in $\mathbf{B}(P, \mathcal{I})$ iff it is in $\mathcal{I}$, furthermore
2. for every relation R of $idb(P)$ every $R(u)$ fact constructed from constants of $C(P, \mathcal{I})$ is in $\mathbf{B}(P, \mathcal{I})$.

Prove that

$$\mathcal{I} \subseteq T_P(\mathcal{I}) \subseteq T_P^2(\mathcal{K}) \subseteq T_P^3(\mathcal{K}) \subseteq \cdots \subseteq \mathbf{B}(P, \mathcal{I}). \tag{30.37}$$

30.1-7 Give an example of an input, that is a datalog program P and instance $\mathcal{I}$ over $edb(P)$,

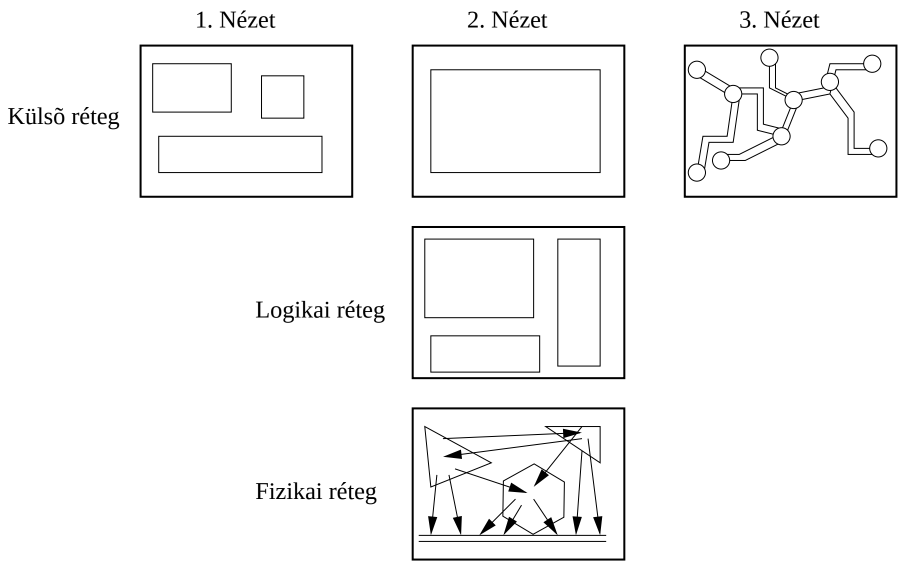

Figure 30.2. The three levels of database architecture.

such that the same tuple is produced by distinct runs of the loop of Semi-naiv-datalog.

30.1-8 Prove that procedure Improved-semi-naiv-datalog stops in finite time for all inputs, and gives correct result. Give an example of an input on which Improved-semi-naiv-datalog calculates less number of rows multiple times than Semi-naiv-datalog.

30.1-9

1. Prove that for tableau queries $q_\mathcal{E} = (\mathbf{T}_\mathcal{E}, t)$ and $q'_\mathcal{E} = (\mathbf{T}'_\mathcal{E}, t)$ of the proof of Theorem 30.20 there exists a homomorphism from $(\mathbf{T}_\mathcal{E}, t)$ to $(\mathbf{T}'_\mathcal{E}, t)$ if and only if the Exact cover problem $\mathcal{E} = (X, \mathcal{S})$ has solution.
2. Prove that the decision problems 30.11. and 30.12. are NP-complete.

30.2. Views

A database system architecture has three main levels:

- physical layer;
- logical layer;
- outer (user) layer.

The goal of the separation of levels is to reach data independence and the convenience of users. The three views on Figure 30.2 show possible user interfaces: multirelational, universal relation and graphical interface.

The ***physical layer*** consists of the actually stored data files and the dense and sparse indices built over them.

The separation of the ***logical layer*** from the physical layer makes it possible for the user to concentrate on the logical dependencies of the data, which approximates the image of the reality to be modelled better. The logical layer consists of the database schema description together with the various integrity constraints, dependencies. This the layer where the database administrators work with the system. The connection between the physical layer and the logical layer is maintained by the database engine.

The goal of the separation of the logical layer and the ***outer layer*** is that the endusers can see the database according to their (narrow) needs and requirements. For example, a very simple view of the outer layer of a bank database could be the automatic teller machine, or a much more complex view could be the credit history of a client for loan approval.

30.2.1. View as a result of a query

The question is that how can the views of different layers be given. If a query given by relational algebra expression is considered as a formula that will be applied to relational instances, then the ***view*** is obtained. Datalog rules show the difference between views and relations, well. The relations defined by rules are called ***intensional***, because these are the relations that do not have to exist on external storage devices, that is to exist extensionally, in contrast to the ***extensional*** relations.

Definition 30.21 *The $\mathcal{V}$ expression given in some query language Q over schema* **R** *is called a* ***view****.*

Similarly to intensional relations, views can be used in definition of queries or other views, as well.

Example 30.7 *SQL view.* Views in database manipulation language SQL can be given in the following way. Suppose that the only interesting data for us from schema **CinePest** is where and when are Kurosawa's film shown. The view **KurosawaTimes** is given by the SQL command

KurosawaTimes

```
1  create view KurosawaTimes as
2           select Theater, Time
3           from Film, Show
4           where Film.Title=Show.Title and Film.Director="Akira Kurosawa"
```

Written in relational algebra is as follows.

$$KurosawaTimes(Theater, Time) = \pi_{Theater,\ Time}(Theater \bowtie \sigma_{Director=\text{"Akira Kurosawa"}}(Film)) \qquad (30.38)$$

Finally, the same by datalog rule is:

$$KurosawaTimes(x_{Th}, x_{Ti}) \leftarrow Theater(x_{Th}, x_T, x_{Ti}), Film(x_T, \text{"Akira Kurosawa"}, x_A). \qquad (30.39)$$

Line 2 of KurosawaTimes marks the selection operator used, line 3 marks that which two relations are to be joined, finally the condition of line 4 shows that it is a natural join, not a direct product.

Having defined view $\mathcal{V}$, it can be used in further queries or view definitions like any other (extensional) relation.

Advantages of using views

- Automatic data hiding: Such data that is not part of the view used, is not shown to the user, thus the user cannot read or modify them without having proper access rights to them. So by providing access to the database through views, a simple, but effective security mechanism is created.
- Views provide simple "macro capabilities". Using the view *KurosawaTimes* defined in Example 30.7. it is easy to find those theatres where Kurosawa films are shown in the morning:

$$KurosawaMorning(Theater) \leftarrow KurosawaTimes(Theater, x_{Ti}), x_{Ti} < 12 \ . \qquad (30.40)$$

  Of course the user could include the definition of *KurosawaTimes* in the code directly, however convenience considerations are first here, in close similarity with macros.
- Views make it possible that the same data could be seen in different ways by different users at the same time.
- Views provide ***logical data independence.*** The essence of logical data independence is that users and their programs are protected from the structural changes of the database schema. It can be achieved by defining the relations of the schema before the structural change as views in the new schema.
- Views make controlled data input possible. The **with check option** clause of command **create view** is to do this in SQL.

Materialised view
Some view could be used in several different queries. It could be useful in these cases that if the tuples of the relation(s) defined by the view need not be calculated again and again, but the output of the query defining the view is stored, and only read in at further uses. Such stored output is called a ***materialised view.*** **Exercises**

30.2-1 Consider the following schema:

$$FilmStar(Name,Address,Gender,BirthDate)$$
$$FilmMogul(Name,Address,Certificate\#,Assets)$$
$$Studio(Name,Address,PresidentCert\#) \ .$$

Relation *FilmMogul* contains data of the big people in film business (studio presidents, producers, etc.). The attribute names speak for themselves, *Certificate#* is the number of the certificate of the filmmogul, *PresidentCert#*) is the certificate number of the president of the studio. Give the definitions of the following views using datalog rules, relational algebra expressions, furthermore SQL:

1. *RichMogul*: Lists the names, addresses,certificate numbers and assets of those filmmoguls, whose asset value is over 1 million dollars.
2. *StudioPresident*: Lists the names, addresses and certificate numbers of those filmmoguls, who are studio presidents, as well.

3. *MogulStar*: Lists the names, addresses,certificate numbers and assets of those people who are filmstars and filmmoguls at the same time.

30.2-2 Give the definitions of the following views over schema **CinePest** using datalog rules, relational algebra expressions, furthermore SQL:

1. *Marilyn*(*Title*): Lists the titles of Marilyn Monroe's films.
2. *CorvinInfo*(*Title*,*Time*,*Phone*): List the titles and show times of films shown in theatre *Corvin*, together with the phone number of the theatre.

30.3. Query rewriting

Answering queries using views, in other words query rewriting using views has become a problem attracting much attention and interest recently. The reason is its applicability in a wide range of data manipulation problems: query optimisation, providing physical data independence, data and information integration, furthermore planning data warehouses.

The problem is the following. Assume that query Q is given over schema $\mathbf{R}$, together with views $V_1, V_2, \ldots, V_n$. Can one answer Q using only the results of views $V_1, V_2, \ldots, V_n$? Or, which is the largest set of tuples that can be determined knowing the views? If the views and the relations of the base schema can be accessed both, what is the cheapest way to compute the result of Q?

30.3.1. Motivation

Before query rewriting algorithms are studied in detail, some motivating examples of applications are given. The following university database is used throughout this section.

$$\textbf{University} = \{\textit{Professor,Course,Teach,Registered,Major,Affiliation,Supervisor}\}\ . \tag{30.41}$$

The schemata of the individual relations are as follows:

$$\begin{aligned}
\textit{Professor} &= \{\textit{PName,Area}\} \\
\textit{Course} &= \{\textit{C-Number,Title}\} \\
\textit{Teaches} &= \{\textit{PName,C-Number,Semester,Evaluation}\} \\
\textit{Registered} &= \{\textit{Student,C-Number,Semester}\} \\
\textit{Major} &= \{\textit{Student,Department}\} \\
\textit{Affiliation} &= \{\textit{PName,Department}\} \\
\textit{Advisor} &= \{\textit{PName,Student}\}\ .
\end{aligned} \tag{30.42}$$

It is supposed that professors, students and departments are uniquely identified by their names. Tuples of relation *Registered* show that which student took which course in what semester, while *Major* shows which department a student choose in majoring (for the sake of convenience it is assumed that one department has one subject as possible major).

Query optimisation

If the computation necessary to answer a query was performed in advance and the results are stored in some materialised view, then it can be used to speed up the query answering.

Consider the query that looks for pairs (*Student,Title*), where the student registered for the given Ph.D.-level course, the course is taught by a professor of the *Database* area (the C-number of graduate courses is at least 400, and the Ph.D.-level courses are those with C-number at least 500).

$$\begin{aligned} val(x_S, x_T) \quad \leftarrow \quad & Teach(x_P, x_C, x_{Se}, y_1), Professor(x_P, \text{“database”}), \\ & Registered(x_S, x_C, x_{Se}), Course(x_C, x_T), x_C \geq 500 \ . \end{aligned} \tag{30.43}$$

Suppose that the following materialised view is available that contains the registration data of graduate courses.

$$Graduate(x_S, x_T, x_C, x_{Se}) \leftarrow Registered(x_S, x_C, x_{Se}), Course(x_X, x_T), x_C \geq 400. \tag{30.44}$$

View *Graduate* can be used to answer query (30.43).

$$\begin{aligned} val(x_S, x_T) \quad \leftarrow \quad & Teaches(x_P, x_C, x_{Se}, y_1), Professor(x_P, \text{“database”}), \\ & (x_S, x_T, x_C, x_{Se}), x_C \geq 500 \ . \end{aligned} \tag{30.45}$$

It will be faster to compute (30.45) than to compute (30.43), because the natural join of relations *Registered* and *Course* has already be done by view *Graduate*, furthermore it shelled off the undergraduate courses (that make up for the bulk of registration data at most universities). It worth noting that view *Graduate* could be used eventhough ***syntactically*** did not agree with any part of query (30.43).

On the other hand, it may happen that the the original query can be answered faster. If relations *Registered* and *Course* have an index on attribute *C-Number*, but there exists no index built for *Graduate*, then it could be faster to answer query (30.43) directly from the database relations. Thus, the real challenge is not only that to decide about a materialised view whether it could be used to answer some query logically, but a thorough cost analysis must be done when is it worth using the existing views.

Physical data independence

One of the principles underlying modern database systems is the separation between the logical view of data and the physical view of data. With the exception of horizontal or vertical partitioning of relations into multiple files, relational database systems are still largely based on a one-to-one correspondence between relations in the schema and the files in which they are stored. In object-oriented systems, maintaining the separation is necessary because the logical schema contains significant redundancy, and does not correspond to a good physical layout. Maintaining physical data independence becomes more crucial in applications where the logical model is introduced as an intermediate level after the physical representation has already been determined. This is common in storage of XML data in relational databases, and in data integration. In fact, the Stored System stores XML documents in a relational database, and uses views to describe the mapping from XML into relations in the database.

To maintain physical data independence, a widely accepted method is to use views over the logical schema as mechanism for describing the physical storage of data. In particular, Tsatalos, Solomon and Ioannidis use GMAPs (*Generalised Multi-level Access Paths*) to describe data storage.

def.gmap G1 **as** b$^+$-tree **by**
  given *Student.name*
  select *Department*
  where *Student* major *Department*

def.gmap G2 **as** b$^+$-tree **by**
  given *Student.name*
  select *Course.c-number*
  where *Student* registered *Course*

def.gmap G3 **as** b$^+$-tree **by**
  given *Course.c-number*
  select *Department*
  where *Student* registered *Course* **and** *Student* major *Department*

Figure 30.3. GMAPs for the university domain.

A GMAP describes the physical organisation and the indexes of the storage structure. The first clause of the GMAP (**as**) describes the actual data structure used to store a set of tuples (e.g., a B$^+$-tree, hash index, etc.) The remaining clauses describe the content of the structure, much like a view definition. The **given** and **select** clauses describe the available attributes, where the **given** clause describes the attributes on which the structure is indexed. The definition of the view, given in the **where** clause uses infix notation over the conceptual model.

In the example shown in Figure 30.3, the GMAP G1 stores a set of pairs containing students and the departments in which they major, and these pairs are indexed by a B$^+$-tree on attribute *Student.name*. The GMAP G2 stores an index from names of students to the numbers of courses in which they are registered. The GMAP G3 stores an index from course numbers to departments whose majors are enrolled in the course.

Given that the data is stored in structures described by the GMAPs, the question that arises is how to use these structures to answer queries. Since the logical content of the GMAPs are described by views, answering a query amounts to finding a way of rewriting the query using these views. If there are multiple ways of answering the query using the views, we would like to find the cheapest one. Note that in contrast to the query optimisation context, we ***must*** use the views to answer the given query, because all the data is stored in the GMAPs.

Consider the following query, which asks for names of students registered for Ph.D.-level courses and the departments in which these students are majoring.

$$\begin{aligned} ans(Student,Department) \leftarrow\ & Registered(Student,C\text{-}number,y), \\ & Major(Student,Department), \\ & C\text{-}number \geq 500. \end{aligned} \tag{30.46}$$

The query can be answered in two ways. First, since *Student.name* uniquely identifies a student, we can take the join of G! and G2, and then apply selection operator *Course.c-number* $\geq$ 500, finally a projection eliminates the unnecessary attributes. The other execution plan could be to join G3 with G2 and select *Course.c-number* $\geq$ 500. In fact, this solution may even be more efficient because G3 has an index on the course number and therefore the intermediate joins may be much smaller.

Data integration

A ***data integration system*** (also known as ***mediator system***) provides a ***uniform*** query interface to a multitude of autonomous heterogeneous data sources. Prime examples of data integration applications include enterprise integration, querying multiple sources on the World-Wide Web, and integration of data from distributed scientific experiments.

To provide a uniform interface, a data integration system exposes to the user a ***mediated schema.*** A mediated schema is a set of ***virtual*** relations, in the sense that they are not stored anywhere. The mediated schema is designed manually for a particular data integration application. To be able to answer queries, the system must also contain a set of ***source descriptions.*** A description of a data source specifies the contents of the source, the attributes that can be found in the source, and the (integrity) constraints on the content of the source. A widely adopted approach for specifying source descriptions is to describe the contents of a data source as a ***view*** over the mediated schema. This approach facilitates the addition of new data sources and the specification of constraints on the contents of sources.

In order to answer a query, a data integration system needs to translate a query formulated on the mediated schema into one that refers directly to the schemata of the data sources. Since the contents of the data sources are described as views, the translation problem amounts finding a way to answer a query using a set of views.

Consider as an example the case where the mediated schema exposed to the user is schema **University**, except that the relations *Teaches* and *Course* have an additional attribute identifying the university at which the course is being taught:

$$\begin{aligned} Course &= \{C\text{-}number, Title, Univ\} \\ Teaches &= \{PName, C\text{-}number, Semester, Evaluation, Univ\} \end{aligned} \tag{30.47}$$

Suppose we have the following two data sources. The first source provides a listing of all the courses entitled "Database Systems" taught anywhere and their instructors. This source can be described by the following view definition:

$$\begin{aligned} DBcourse(Title, PName, C\text{-}number, Univ) \leftarrow\ & Course(\ C\text{-}number, Title, Univ), \\ & Teaches(PName, C\text{-}number, Semester, \\ & \qquad Evaluation, Univ), \\ & Title = \text{“Database Systems”}\ . \end{aligned} \tag{30.48}$$

The second source lists Ph.D.-level courses being taught at The Ohio State University (OSU), and is described by the following view definition:

$$\begin{aligned} OSUPhD(Title, PName, C\text{-}number, Univ) \leftarrow\ & Course(\ C\text{-}number, Title, Univ), \\ & Teaches(PName, C\text{-}number, Semester, \\ & \qquad Evaluation, Univ), \\ & Univ = \text{“OSU”}, C\text{-}number \geq 500. \end{aligned} \tag{30.49}$$

If we were to ask the data integration system who teaches courses titled "Database Systems" at OSU, it would be able to answer the query by applying a selection on the source *DB-courses*:

$$ans(PName) \leftarrow DBcourse(Title, PName, C\text{-}number, Univ), Univ = \text{"OSU"}\ . \tag{30.50}$$

On the other hand, suppose we ask for all the graduate-level courses (not just in databases) being offered at OSU. Given that only these two sources are available, the data integration system cannot find ***all*** tuples in the answer to the query. Instead, the system can attempt to find the maximal set of tuples in the answer available from the sources. In particular, the system can obtain graduate ***database*** courses at OSU from the *DB-course* source, and the Ph.D.-level courses at OSU from the *OSUPhD* source. Hence, the following non-recursive datalog program gives the maximal set of answers that can be obtained from the two sources:

$$\begin{aligned} ans(Title, C\text{-}number) &\leftarrow DBcourse(Title, PName, C\text{-}number, Univ), \\ &\quad Univ = \text{"OSU"}, C\text{-}number \geq 400 \\ ans(Title, C\text{-}number) &\leftarrow OSUPhD(Title, PName, C\text{-}number, Univ)\ . \end{aligned} \tag{30.51}$$

Note that courses that are not PH.D.-level courses or database courses will not be returned as answers. Whereas in the contexts of query optimisation and maintaining physical data independence the focus is on finding a query expression that is ***equivalent*** with the original query, here finding a query expression that provides the ***maximal answers*** from the views is attempted.

Semantic data caching

If the database is accessed via client-server architecture, the data cached at the client can be semantically modelled as results of certain queries, rather than at the physical level as a set of data pages or tuples. Hence, deciding which data needs to be shipped from the server in order to answer a given query requires an analysis of which parts of the query can be answered by the cached views.

30.3.2. Complexity problems of query rewriting

In this section the ***theoretical*** complexity of query rewriting is studied. Mostly conjunctive queries are considered. ***Minimal,*** and ***complete*** rewriting will be distinguished. It will be shown that if the query is conjunctive, furthermore the materialised views are also given as results of conjunctive queries, then the rewriting problem is *NP-complete,* assuming that neither the query nor the view definitions contain comparison atoms. Conjunctive queries will be considered in rule-based form for the sake of convenience.

Assume that query Q is given over schema $\mathbf{R}$.

Definition 30.22 *The conjunctive query Q' is a **rewriting** of query Q using views $\mathcal{V} = V_1, V_2, \ldots, V_m$, if*

- *Q and Q' are equivalent, and*
- *Q' contains one or more literals from $\mathcal{V}$.*

*Q' is said to be **locally minimal** if no literal can be removed from Q' without violating the equivalence. The rewriting is **globally minimal,** if there exists no rewriting using a smaller number of literals. (The comparison atoms $=, \neq, \leq, <$ are not counted in the number of literals.)*

Example 30.8 *Query rewriting.* Consider the following query Q and view V.

$$\begin{aligned} Q: q(X, U) &\leftarrow p(X, Y), p_0(Y, Z), p_1(X, W), p_2(W, U) \\ V: v(A, B) &\leftarrow p(A, C), p_0(C, B), p_1(A, D) \end{aligned} \tag{30.52}$$

Q can be rewritten using V:

$$Q' : q(X,U) \leftarrow v(X,Z), p_1(X,W), p_2(W,U) \ . \tag{30.53}$$

View V replaces the first two literals of query Q. Note that the view certainly satisfies the third literal of the query, as well. However, it cannot be removed from the rewriting, since variable D does not occur in the head of V, thus if literal p_1 were to be removed, too, then the natural join of p_1 and p_2 would not be enforced anymore.

Since in some of the applications the database relations are inaccessible, only the views can be accessed, for example in case of data integration or data warehouses, the concept of ***complete rewriting*** is introduced.

Definition 30.23 *A rewriting Q' of query Q using views $\mathcal{V} = V_1, V_2, \ldots, V_m$ is called a **complete rewriting,** if Q' contains only literals of $\mathcal{V}$ and comparison atoms.*

Example 30.9 *Complete rewriting.* Assume that besides view V of Example 30.8. the following view is given, as well:

$$V_2 : v_2(A,B) \leftarrow p_1(A,C), p_2(C,B), p_0(D,E) \tag{30.54}$$

A complete rewriting of query Q is:

$$Q'' : q(X,U) \leftarrow v(X,Z), v_2(X,U) \ . \tag{30.55}$$

It is important to see that this rewriting cannot be obtained ***step-by-step,*** first using only V, then trying to incorporate V_2, (or just in the opposite order) since relation p_0 of V_2 does not occur in Q'. Thus, in order to find the complete rewriting, use of the two view must be considered ***parallel,*** at the same time.

There is a close connection between finding a rewriting and the problem of query containment. This latter one was discussed for tableau queries in section 30.1.3. Homomorphism between tableau queries can be defined for rule based queries, as well. The only difference is that it is not required in this section that a homomorphism maps the head of one rule to the head of the other. (The head of a rule corresponds to the summary row of a tableau.) According to Theorem 30.20 it is NP-complete to decide whether conjunctive query Q_1 contains another conjunctive query Q_2. This remains true in the case when Q_2 may contain comparison atoms, as well. However, if both, Q_1 and Q_2 may contain comparison atoms, then the existence of a homomorphism from Q_1 to Q_2 is only a sufficient but not necessary condition for the containment of queries, which is a Π_2^p-complete problem in that case. The discussion of this latter complexity class is beyond the scope of this chapter, thus it is omitted. The next proposition gives a necessary and sufficient condition whether there exists a rewriting of query Q using view V.

Proposition 30.24 *Let Q and V be conjunctive queries that may contain comparison atoms. There exists a a rewriting of query Q using view V if and only if $\pi_\emptyset(Q) \subseteq \pi_\emptyset(V)$, that is the projection of V to the empty attribute set contains that of Q.*

Proof. Observe that $\pi_\emptyset(Q) \subseteq \pi_\emptyset(V)$ is equivalent with the following proposition: If the output of V is empty for some instance, then the same holds for the output of Q, as well.

Assume first that there exists a rewriting, that is a rule equivalent with Q that contains

V in its body. If r is such an instance, that the result of V is empty on it, then every rule that includes V in its body results in empty set over r, too.

In order to prove the other direction, assume that if the output of V is empty for some instance, then the same holds for the output of Q, as well. Let

$$\begin{array}{rcl} Q\colon q(\tilde{x}) & \leftarrow & q_1(\tilde{x}), q_2(\tilde{x}), \dots, q_m(\tilde{x}) \\ V\colon v(\tilde{a}) & \leftarrow & v_1(\tilde{a}), v_2(\tilde{a}), \dots, v_n(\tilde{a})\ . \end{array} \tag{30.56}$$

Let $\tilde{y}$ be a list of variables disjoint from variables of $\tilde{x}$. Then the query Q' defined by

$$Q'\colon q'(\tilde{x}) \leftarrow q_1(\tilde{x}), q_2(\tilde{x}), \dots, q_m(\tilde{x}), v_1(\tilde{y}), v_2(\tilde{y}), \dots, v_n(\tilde{y}) \tag{30.57}$$

satisfies $Q \equiv Q'$. It is clear that $Q' \subseteq Q$. On the other hand, if there exists a valuation of the variables of $\tilde{y}$ that satisfies the body of V over some instance r, then fixing it, for arbitrary valuation of variables in $\tilde{x}$ a tuple is obtained in the output of Q, whenever a tuple is obtained in the output of Q' together with the previously fixed valuation of variables of $\tilde{y}$. ■

As a corollary of Theorem 30.20 and Proposition 30.24 the following theorem is obtained.

Theorem 30.25 *Let Q be a conjunctive query that may contain comparison atoms, and let $\mathcal{V}$ be a set of views. If the views in $\mathcal{V}$ are given by conjunctive queries that do not contain comparison atoms, then it is NP-complete to decide whether there exists a rewriting of Q using $\mathcal{V}$.*

The proof of Theorem 30.25 is left for the Reader (Exercise 30.3-1.).

The proof of Proposition 30.24 uses new variables. However, according to the next lemma, this is not necessary. Another important observation is that it is enough to consider a subset of database relations occurring in the original query when locally minimal rewriting is sought, new database relations need not be introduced.

Lemma 30.26 *Let Q be a conjunctive query that does not contain comparison atoms*

$$Q\colon q(\tilde{X}) \leftarrow p_1(\tilde{U}_1), p_2(\tilde{U}_2), \dots, p_n(\tilde{U}_n)\ , \tag{30.58}$$

furthermore let $\mathcal{V}$ be a set of views that do not contain comparison atoms either.

1. *If Q' is a locally minimal rewriting of Q using $\mathcal{V}$, then the set of database literals in Q' is isomorphic to a subset of database literals occurring in Q.*
2. *If*

$$q(\tilde{X}) \leftarrow p_1(\tilde{U}_1), p_2(\tilde{U}_2), \dots, p_n(\tilde{U}_n), v_1(\tilde{Y}_1), v_2(\tilde{Y}_2), \dots v_k(\tilde{Y}_k) \tag{30.59}$$

is a rewriting of Q using the views, then there exists a rewriting

$$q(\tilde{X}) \leftarrow p_1(\tilde{U}_1), p_2(\tilde{U}_2), \dots, p_n(\tilde{U}_n), v_1(\tilde{Y'}_1), v_2(\tilde{Y'}_2), \dots v_k(\tilde{Y'}_k) \tag{30.60}$$

such that $\{\tilde{Y'}_1 \cup \dots \cup \tilde{Y'}_k\} \subseteq \{\tilde{U}_1 \cup \dots \cup \tilde{U}_n\}$, that is the rewriting does not introduce new variables.

The details of the proof of Lemma 30.26 are left for the Reader (Exercise 30.3-2.). The next lemma is of fundamental importance: A minimal rewriting of Q using $\mathcal{V}$ cannot ***increase*** the number of literals.

Lemma 30.27 *Let Q be conjunctive query, $\mathcal{V}$ be set of views given by conjunctive queries, both without comparison atoms. If the body of Q contains p literals and Q' is a locally minimal rewriting of Q using $\mathcal{V}$, then Q' contains at most p literals.*

Proof. Replacing the view literals of Q' by their definitions query Q'' is obtained. Let φ be a homomorphism from the body of Q to Q''. The existence of φ follows from $Q \equiv Q''$ by the Homomorphism Theorem (Theorem 30.18). Each of the literals $l_1, l_2, \ldots, l_p$ of the body of Q is mapped to at most one literal obtained from the expansion of view definitions. If Q' contains more than p view literals, then the expansion of some view literals in the body of Q'' is disjoint from the image of φ. These view literals can be removed from the body of Q' without changing the equivalence. ■
Based on Lemma 30.27 the following theorem can be stated about complexity of minimal rewritings.

Theorem 30.28 *Let Q be conjunctive query, $\mathcal{V}$ be set of views given by conjunctive queries, both without comparison atoms. Let the body of Q contain p literals.*

1. *It is NP-complete to decide whether there exists a rewriting Q' of Q using $\mathcal{V}$ that uses at most k $(\leq p)$ literals.*
2. *It is NP-complete to decide whether there exists a rewriting Q' of Q using $\mathcal{V}$ that uses at most k $(\leq p)$ database literals.*
3. *It is NP-complete to decide whether there exists a complete rewriting of Q using $\mathcal{V}$.*

Proof. The first statement is proved, the proof of the other two is similar. According to Lemmas 30.27 and 30.26, only such rewritings need to be considered that have at most as many literals as the query itself, contain a subset of the literals of the query and do not introduce new variables. Such a rewriting and the homomorphisms proving the equivalence can be tested in polynomial time, hence the problem is in NP. In order to prove that it is NP-hard, Theorem 30.25 is used. For a given query Q and view V let V' be the view, whose head is same as the head of V, but whose body is the conjunction of the bodies of Q and V. It is easy to see that there exists a rewriting using V' with a single literal if and only if there exists a rewriting (with no restriction) using V. ■

30.3.3. Practical algorithms

In this section only complete rewritings are studied. This does not mean real restriction, since if database relations are also to be used, then views mirroring the database relations one-to-one can be introduced. The concept of ***equivalent*** rewriting introduced in Definition 30.22 is appropriate if the goal of the rewriting is query optimisation or providing physical data independence. However, in the context of data integration on data warehouses equivalent rewritings cannot be sought, since all necessary data may not be available. Thus, the concept of maximally contained rewriting is introduced that depends on the query language used, in contrast to equivalent rewritings.

Definition 30.29 *Let Q be a query, $\mathcal{V}$ be a set of views, $\mathcal{L}$ be a query language. Q' is a* ***maximally contained rewriting*** *of Q with respect to $\mathcal{L}$, if*

1. *Q' is a query of language $\mathcal{L}$ using only views from $\mathcal{V}$,*

2. *Q contains Q',*
3. *if query $Q_1 \in \mathcal{L}$ satisfies $Q' \sqsubseteq Q_1 \sqsubseteq Q$, then $Q' \equiv Q_1$.*

Query optimisation using materialised views

Before discussing how can a traditional optimiser be modified in order to use materialised views instead of database relations, it is necessary to survey when can view be used to answer a given query. Essentially, view V can be used to answer query Q, if the intersection of the sets of database relations in the body of V and in the body of Q is non-empty, furthermore some of the attributes are selected by V are also selected by Q. Besides this, in case of equivalent rewriting, if V contains comparison atoms for such attributes that are also occurring in Q, then the view must apply logically equivalent, or weaker condition, than the query. If logically stronger condition is applied in the view, then it can be part of a (maximally) contained rewriting. This can be shown best via an example. Consider the query Q over schema **University** that list those professor, student, semester triplets, where the advisor of the student is the professor and in the given semester the student registered for some course taught by the professor.

$$\begin{aligned} Q:\ q(\textit{Pname,Student,Semester}) \leftarrow\ & \textit{Registered}(\textit{Student,C-number,Semester}),\\ & \textit{Advisor}(\textit{Pname,Student}),\\ & \textit{Teaches}(\textit{Pname, C-number,Semester}, x_E),\\ & \textit{Semester} \geq \text{“Fall2000”}\ . \end{aligned} \tag{30.61}$$

View V_1 below can be used to answer Q, since it uses the same join condition for relations *Registered* and *Teaches* as Q, as it is shown by variables of the same name. Furthermore, V_1 selects attributes *Student, PName, Semester*, that are necessary in order to properly join with relation *Advisor*, and for select clause of the query. Finally, the predicate *Semester* > “Fall1999” is weaker than the predicate *Semester* ≥ “Fall2000” of the query.

$$\begin{aligned} V_1:\ v_1(\textit{Student,PName,Semester}) \leftarrow\ & \textit{Teaches}(\textit{PName,C-number,Semester}, x_E),\\ & \textit{Registered}(\textit{Student,C-number,Semester}),\\ & \textit{Semester} > \text{“Fall1999”}\ . \end{aligned} \tag{30.62}$$

The following four views illustrate how minor modifications to V_1 change the usability in answering the query.

$$\begin{aligned} V_2:\ v_2(\textit{Student,Semester}) \leftarrow\ & \textit{Teaches}(x_P, \textit{C-number,Semester}, x_E),\\ & \textit{Registered}(\textit{Student,C-number,Semester}),\\ & \textit{Semester} > \text{“Fall1999”}\ . \end{aligned} \tag{30.63}$$

$$\begin{aligned} V_3:\ v_3(\textit{Student,PName,Semester}) \leftarrow\ & \textit{Teaches}(\textit{PName, C-number}, x_S, x_E),\\ & \textit{Registered}(\textit{Student,C-number,Semester}),\\ & \textit{Semester} > \text{“Fall1999”}\ . \end{aligned} \tag{30.64}$$

$$\begin{aligned} V_4:\ v_4(\textit{Student,PName,Semester}) \leftarrow\ & \textit{Teaches}(\textit{PName, C-number,Semester}, x_E),\\ & \textit{Adviser}(\textit{PName}, x_{St}), \textit{Professor}(\textit{PName}, x_A),\\ & \textit{Registered}(\textit{Student,C-number,Semester}),\\ & \textit{Semester} > \text{“Fall1999”}\ . \end{aligned} \tag{30.65}$$

$$\begin{aligned} V_5:\ v_5(\mathit{Student,PName,Semester}) \leftarrow\ & \mathit{Teaches}(\mathit{PName,\ C\text{-}number,Semester},x_E),\\ & \mathit{Registered}(\mathit{Student,C\text{-}number,Semester}),\\ & \mathit{Semester} > \text{“Fall2001”}\ . \end{aligned} \tag{30.66}$$

View V_2 is similar to V_1, except that it does not select the attribute *PName* from relation *Teaches*, which is needed for the join with the relation *Adviser* and for the selection of the query. Hence, to use V_2 in the rewriting, it has to be joined with relation *Teaches* again. Still, if the join of relations *Registered* and *Teaches* is very selective, then employing V_2 may actually result in a more efficient query execution plan.

In view V_3 the join of relations *Registered* and *Teaches* is over only attribute *C-number*, the equality of variables *Semester* and x_S is not required. Since attribute x_S is not selected by V_3, the join predicate cannot be applied in the rewriting, and therefore there is little gain by using V_3.

View V_4 considers only the professors who have at least one area of research. Hence, the view applies an additional condition that does not exists in the query, and cannot be used in an equivalent rewriting unless union and negation are allowed in the rewriting language. However, if there is an integrity constraint stating that every professor has at least one area of research, then an optimiser should be able to realise that V_4 is usable.

Finally, view V_5 applies a stronger predicate than the query, and is therefore usable for a contained rewriting, but not for an equivalent rewriting of the query.

System-R style optimisation

Before discussing the changes to traditional optimisation, first the principles underlying the ***System-R style optimiser*** is recalled briefly. System-R takes a bottom-up approach to building query execution plans. In the first phase, it concentrates of plans of size 1, i.e., chooses the best access paths to every table mentioned in the query. In phase n, the algorithm considers plans of size n, by combining plans obtained in the previous phases (sizes of k and $n - k$). The algorithm terminates after constructing plans that cover all the relations in the query. The efficiency of System-R stems from the fact that it partitions query execution plans into ***equivalence classes,*** and only considers a single execution plan for every equivalence class. Two plans are in the same equivalence class if they

- cover the same set of relations in the query (and therefore are also of the same size), and
- produce the answer in the same interesting order.

In our context, the query optimiser builds query execution plans by accessing a set of views, rather than a set of database relations. Hence, in addition to the meta-data that the query optimiser has about the materialised views (e.g., statistics, indexes) the optimiser is also given as input the query expressions defining the views. Th additional issues that the optimiser needs to consider in the presence of materialised views are as follows.

A. In the first iteration the algorithm needs to decide which views are ***relevant*** to the query according to the conditions illustrated above. The corresponding step is trivial in a traditional optimiser: a relation is relevant to the query if it is in the body of the query expression.

B. Since the query execution plans involve joins over views, rather than joins over database relations, plans can no longer be neatly partitioned into equivalence classes which can be explored in increasing size. This observation implies several changes to the traditional algorithm:

1. ***Termination testing:*** the algorithm needs to distinguish ***partial query execution plans*** of the query from ***complete query execution plans.*** The enumeration of the possible join orders terminates when there are no more unexplored partial plans. In contrast, in the traditional setting the algorithm terminates after considering the equivalence classes that include all the relations of the query.

2. ***Pruning of plans:*** a traditional optimiser compares between pairs of plans ***within*** one equivalence class and saves only the cheapest one for each class. I our context, the query optimiser needs to compare between ***any pair*** of plans generated thus far. A plan p is pruned if there is another plan p' that

   (a) is cheaper than p, and
   (b) has greater or equal contribution to the query than p. Informally, a plan p' contributes more to the query than plan p if it covers more of the relations in the query and selects more of the necessary attributes.

3. ***Combining partial plans:*** in the traditional setting, when two partial plans are combined, the join predicates that involve both plans are explicit in the query, and the enumeration algorithm need only consider the most efficient way to apply these predicates. However, in our case, it may not be obvious a priori which join predicate will yield a correct rewriting of the query, since views are joined rather than database relations directly. Hence, the enumeration algorithm needs to consider several alternative join predicates. Fortunately, in practice, the number of join predicates that need to be considered can be significantly pruned using meta-data about the schema. For example, there is no point in trying to join a string attribute with a numeric one. Furthermore, in some cases knowledge of integrity constraints and the structure of the query can be used to reduce the number of join predicates to be considered. Finally, after considering all the possible join predicates, the optimiser also needs to check whether the resulting plan is still a partial solution to the query.

The following table summarises the comparison of the traditional optimiser versus one that exploits materialised views.

Conventional optimiser	**Optimiser using views**
Iteration 1	***Iteration 1***
a) Find all possible access paths.	a1)Find all views that are relevant to the query. a2) Distinguish between partial and complete solutions to the query.
b) Compare their cost and keep the least expensive.	b) Compare all pairs of views. If one has neither greater contribution nor a lower cost than the other, prune it.
c) If the query has one relation, **stop**.	c) If there are no partial solutions, **stop**.
Iteration 2	***Iteration 2***
For each query join:	
a) Consider joining the relevant access paths found in the previous iteration using all possible join methods.	a1) Consider joining all partial solutions found in the previous iteration using all possible equi-join methods and trying all possible subsets of join predicates. a2) Distinguish between partial and complete solutions to the query.
b) Compare the cost of the resulting join plans and keep the least expensive.	b) If any newly generated solution is either not relevant to the query, or dominated by another, prune it.
c) If the query has two relations, **stop**.	c) If there are no partial solutions, **stop**.
Iteration 3	***Iteration 3***
⋮	⋮

Another method of equivalent rewriting is using transformation rules. The common theme in the works of that area is that replacing some part of the query with a view is considered as another transformation available to the optimiser. These methods are not discussed in detail here.

The query optimisers discussed above were designed to handle cases where the number of views is relatively small (i.e., comparable to the size of the database schema), and cases where equivalent rewriting is required. In contrast, the context of data integration requires consideration of large number of views, since each data source is being described by one or more views. In addition, the view definitions may contain many complex predicates, whose goal is to express fine-grained distinction between the contents of different data sources. Furthermore, in the context of data integration it is often assumed that the views are not complete, i.e., they may only contain a subset of the tuples satisfying their definition. In the foregoing, some algorithms for answering queries using views are described that were developed specifically for the context of data integration.

The Bucket Algorithm

The goal of the Bucket Algorithm is to reformulate a user query that is posed on a mediated (virtual) schema into a query that refers directly to the available sources. Both the query and the sources are described by conjunctive queries that may include atoms of arithmetic comparison predicates. The set of comparison atoms of query Q is denoted by $C(Q)$.

Since the number of possible rewritings may be exponential in the size of the query, the main idea underlying the bucket algorithm is that the number of query rewritings that need to be considered can be drastically reduced if we first consider each ***subgoal*** – the relational atoms of the query – is considered in isolation, and determine which views may be relevant to each subgoal.

The algorithm proceeds as follows. First, a ***bucket*** is created for each subgoal in the query Q that is not in $C(Q)$, containing the views that are relevant to answering the particular subgoal. In the second step, all such conjunctive query rewritings are considered that include one conjunct (view) from each bucket. For each rewriting V obtained it is checked that whether it is semantically correct, that is $V \sqsubseteq Q$ holds, or whether it can be made semantically correct by adding comparison atoms. Finally the remaining plans are minimised by pruning redundant subgoals. Algorithm Create-bucket executes the first step described above. Its input is a set of source descriptions $\mathcal{V}$ and a conjunctive query Q in the form

$$Q\colon\ Q(\tilde{X}) \leftarrow R_1(\tilde{X}_1), R_2(\tilde{X}_2), \ldots, R_m(\tilde{X}_m), C(Q)\ . \tag{30.67}$$

```
Create-bucket($Q$,$\mathcal{V}$)
1  **for** $i \leftarrow 1$ **to** $m$
2      **do** $Bucket[i] \leftarrow \emptyset$
3          **for** all $V \in \mathcal{V}$
4                              $\triangleright$ $V$ is of form $V\colon\ V(\tilde{Y}) \leftarrow S_1(\tilde{Y}_1), \ldots S_n(\tilde{Y}_n), C(V)$.
5              **do for** $j \leftarrow 1$ **to** $n$
6                  **if** $R_i = S_j$
7                      **then** Let $\phi$ be a mapping defined on the variables of $V$ as follows:
8                          **if** $y$ is the $k^{\text{th}}$ variable of $\tilde{Y}_j$ and $y \in \tilde{Y}$
9                              **then** $\phi(y) = x_k$, where $x_k$ is the $k^{\text{th}}$ variable of $\tilde{X}_i$
10                             **else** $\phi(y)$ is a new variable that does not appear in $Q$ or $V$.
11                         $Q'() \leftarrow R_1(\tilde{X}_1), R_m(\tilde{X}_m), C(Q), S_1(\phi(\tilde{Y}_1)), \ldots, S_n(\phi(\tilde{Y}_n)), \phi(C(V))$
12                         **if** Satisfiable$^{\geq}(Q')$
13                             **then** add $\phi(V)$ to $Bucket[i]$.
14 **return** $Bucket$
```

Procedure Satisfiable$^{\geq}$ is the extension of Satisfiable described in section 30.1.2 to the case when comparison atoms may occur besides equality atoms. The necessary change is only that for all variable y occurring in comparison atoms it must be checked whether all predicates involving y are satisfiable simultaneously.

Create-bucket running time is polynomial function of the sizes of Q and $\mathcal{V}$. Indeed, the kernel of the nested loops of lines 3 and 5 runs $n \sum_{V \in \mathcal{V}} |V|$ times. The commands of lines 6–13 require constant time, except for line 12. The condition of of command **if** in line 12 can be checked in polynomial time.

In order to prove the correctness of procedure Create-bucket, one should check under what condition is a view V put in $Bucket[i]$. In line 6 it is checked whether relation R_i appears as a subgoal in V. If not, then obviously V cannot give usable information for subgoal R_i of Q. If R_i is a subgoal of V, then in lines 9–10 a mapping is constructed that applied to the variables allows the correspondence between subgoals S_j and R_i, in accordance with relations occurring in the heads of Q and V, respectively. Finally, in line 12 it is checked whether the comparison atoms contradict with the correspondence constructed.

In the second step, having constructed the buckets using Create-bucket, the bucket algorithm finds a set of ***conjunctive query rewritings,*** each of them being a conjunctive query that includes one conjunct from every bucket. Each of these conjunctive query rewritings represents one way of obtaining part of the answer to Q from the views. The result of the bucket algorithm is defined to be the union of the conjunctive query rewritings (since each of the rewritings may contribute different tuples). A given conjunctive query Q' is a ***conjunctive query rewriting,*** if

1. $Q' \sqsubseteq Q$, or
2. Q' can be extended with comparison atoms, so that the previous property holds.

Example 30.10 *Bucket algorithm.* Consider the following query Q that lists those articles x that there exists another article y of the same area such that x and y mutually cites each other. There are three views (sources) available, V_1, V_2, V_3.

$$\begin{array}{lcl} Q(x) & \leftarrow & cite(x,y), cite(y,x), sameArea(x,y) \\ V_1(a) & \leftarrow & cite(a,b), cite(b,a) \\ V_2(c,d) & \leftarrow & sameArea(c,d) \\ V_3(f,h) & \leftarrow & cite(f,g), cite(g,h), sameArea(f,g)\ . \end{array} \tag{30.68}$$

In the first step, applying Create-bucket, the following buckets are constructed.

$cite(x,y)$	$cite(y,x)$	$sameArea(x,y)$
$V_1(x)$	$V_1(x)$	$V_2(x)$
$V_3(x)$	$V_3(x)$	$V_3(x)$

(30.69)

In the second step the algorithm constructs a conjunctive query Q' from each element of the Cartesian product of the buckets, and checks whether Q' is contained in Q. If yes, it is given to the answer.

In our case, it tries to match V_1 with the other views, however no correct answer is obtained so. The reason is that b does not appear in the head of V_1, hence the join condition of Q – variables x and y occur in relation *sameArea*, as well – cannot be applied. Then rewritings containing V_3 are considered, recognising that equating the variables in the head of V_3 a contained rewriting is obtained. Finally, the algorithm finds that combining V_3 and V_2 rewriting is obtained, as well. This latter is redundant, as it is obtained by simple checking, that is V_2 can be pruned. Thus, the result of the bucket algorithm for query (30.68) is the following (actually equivalent) rewriting

$$Q'(x) \leftarrow V_3(x,x)\ . \tag{30.70}$$

The strength of the bucket algorithm is that it exploits the predicates in the query to prune significantly the number of candidate conjunctive rewritings that need to be considered. Checking whether a view should belong to a bucket can be done in time polynomial in the size of the query and view definition when the predicates involved are arithmetic comparisons. Hence, if the data sources (i.e., the views) are indeed distinguished by having different comparison predicates, then the resulting buckets will be relatively small.

The main disadvantage of the bucket algorithm is that the Cartesian product of the buckets may still be large. Furthermore, the second step of the algorithm needs to perform a query containment test for every candidate rewriting, which is NP-complete even when no comparison predicates are involved.

Inverse-rules Algorithm

The Inverse-rules Algorithm is a procedure that can be applied more generally than the Bucket Algorithm. It finds a maximally contained rewriting for any query given by arbitrary recursive datalog program that does not contain negation, in polynomial time.

The first question is that for given datalog program $\mathcal{P}$ and set of conjunctive queries

$\mathcal{V}$, whether there exists a datalog program $\mathcal{P}_v$ equivalent with $\mathcal{P}$, whose *edb* relations are relations $v_1, v_2, \ldots, v_n$ of $\mathcal{V}$. Unfortunately, this is algorithmically undecidable. Surprisingly, the best, maximally contained rewriting can be constructed. In the case, when there exists a datalog program $\mathcal{P}_v$ equivalent with $\mathcal{P}$, the algorithm finds that, since a maximally contained rewriting contains $\mathcal{P}_v$, as well. This seemingly contradicts to the fact that the existence of equivalent rewriting is algorithmically undecidable, however it is undecidable about the result of the Inverse-rules Algorithm, whether it is really equivalent to the original query.

Example 30.11 *Equivalent rewriting.* Consider the following datalog program $\mathcal{P}$, where *edb* relations *edge* and *black* contain the edges and vertices coloured black of a graph G.

$$\begin{array}{llll} \mathcal{P}: & q(X,Y) & \leftarrow & edge(X,Z), edge(Z,Y), black(Z) \\ & q(X,Y) & \leftarrow & edge(X,Z), black(Z), q(Z,Y)\ . \end{array} \tag{30.71}$$

It is easy to check that $\mathcal{P}$ lists the endpoints of such paths (more precisely walks) of graph G whose inner points are all black. Assume that only the following two views can be accessed.

$$\begin{array}{lll} v_1(X,Y) & \leftarrow & edge(X,Y), black(X) \\ v_2(X,Y) & \leftarrow & edge(X,Y), black(Y) \end{array} \tag{30.72}$$

v_1 stores edges whose tail is black, while v_2 stores those, whose head is black. There exists an equivalent rewriting $\mathcal{P}_v$ of datalog program $\mathcal{P}$ that uses only views v_1 and v_2 as *edb* relations:

$$\begin{array}{llll} \mathcal{P}_v: & q(X,Y) & \leftarrow & v_2(X,Z), v_1(Z,Y) \\ & q(X,Y) & \leftarrow & v_2(X,Z), q(Z,Y) \end{array} \tag{30.73}$$

However, if only v_1, or v_2 is accessible alone, then equivalent rewriting is not possible, since only such paths are obtainable whose starting, or respectively, ending vertex is black.

In order to describe the Inverse-rules Algorithm, it is necessary to introduce the ***Horn rule***, which is a generalisation of ***datalog program,*** and ***datalog rule***. If ***function symbols*** are also allowed in the free tuple u_i of rule (30.27) in Definition 30.11, besides variables and constants, then ***Horn rule*** is obtained. A ***logic program*** is a collection of Horn rules. In this sense a logic program without function symbols is a datalog program. The concepts of *edb* and *idb* can be defined for logic programs in the same way as for datalog programs.

The Inverse-rules Algorithm consists of two steps. First, a logic program is constructed that may contain function symbols. However, these will not occur in recursive rules, thus in the second step they can be eliminated and the logic program can be transformed into a datalog program.

Definition 30.30 *The **inverse** v^{-1} of view v given by*

$$v(X_1,\ldots,X_m) \leftarrow v_1(\tilde{Y}_1),\ldots,v_n(\tilde{Y}_n) \tag{30.74}$$

is the following collection of Horn rules. A rule corresponds to every subgoal $v_i(\tilde{Y}_i)$, whose body is the literal $v(X_1,\ldots,X_m)$. The head of the rule is $v_i(\tilde{Z}_i)$, where $\tilde{Z}_i$ is obtained from $\tilde{Y}_i$ by preserving variables appearing in the head of rule (30.74), while function symbol $f_Y(X_1,\ldots,X_m)$ is written in place of every variable Y not appearing the head. Distinct function symbols correspond to distinct variables. The inverse of a set $\mathcal{V}$ of views is the set $\{v^{-1}\colon\ v \in \mathcal{V}\}$, where distinct function symbols occur in the inverses of distinct rules.

The idea of the definition of inverses is that if a tuple $(x_1,\dots x_m)$ appears in a view v, for some constants $x_1,\dots x_m$, then there is a valuation of every variable y appearing in the head that makes the body of the rule true. This "unknown" valuation is denoted by the function symbol $f_Y(X_1,\dots,X_m)$.

Example 30.12 *Inverse of views.* Let $\mathcal{V}$ be the following collection of views.

$$\begin{array}{lcl} v_1(X,Y) & \leftarrow & edge(X,Z), edge(Z,W), edge(W,Y) \\ v_2(X)) & \leftarrow & edge(X,Z)\ . \end{array} \tag{30.75}$$

Then $\mathcal{V}^{-1}$ consists of the following rules.

$$\begin{array}{lcl} edge(X, f_{1,Z}(X,Y)) & \leftarrow & v_1(X,Y) \\ edge(f_{1,Z}(X,Y), f_{1,W}(X,Y)) & \leftarrow & v_1(X,Y) \\ edge(f_{1,W}(X,Y), Y) & \leftarrow & v_1(X,Y) \\ edge(X, f_{2,Z}(X)) & \leftarrow & v_2(X)\ . \end{array} \tag{30.76}$$

Now, the maximally contained rewriting of datalog program $\mathcal{P}$ using views $\mathcal{V}$ can easily be constructed for given $\mathcal{P}$ and $\mathcal{V}$.

First, those rules are deleted from $\mathcal{P}$ that contain such *edb* relation that do not appear in the definition any view from $\mathcal{V}$. The rules of $\mathcal{V}^{-1}$ are added the datalog program $\mathcal{P}^-$ obtained so, thus forming logic program $(\mathcal{P}^-,\mathcal{V}^{-1})$. Note, that the remaining *edb* relations of $\mathcal{P}$ are *idb* relations in logic program $(\mathcal{P}^-,\mathcal{V}^{-1})$, since they appear in the heads of the rules of $\mathcal{V}^{-1}$. The names of *idb* relations are arbitrary, so they can be renamed so that their names do not coincide with the names of *edb* relations of $\mathcal{P}$. However, this is not done in the following example, for the sake of better understanding.

Example 30.13 *Logic program.* Consider the following datalog program that calculates the transitive closure of relation *edge*.

$$\begin{array}{llcl} \mathcal{P}: & q(X,Y) & \leftarrow & edge(X,Y) \\ & q(X,Y) & \leftarrow & edge(X,Z), q(Z,Y) \end{array} \tag{30.77}$$

Assume that only the following materialised view is accessible, that stores the endpoints of paths of length two. If only this view is usable, then the most that can be expected is listing the endpoints of paths of even length. Since the unique *edb* relation of datalog program $\mathcal{P}$ is *edge*, that also appears in the definition of v, the logic program $(\mathcal{P}^-,\mathcal{V}^{-1})$ is obtained by adding the rules of $\mathcal{V}^{-1}$ to $\mathcal{P}$.

$$\begin{array}{llcl} (\mathcal{P}^-,\mathcal{V}^{-1}): & q(X,Y) & \leftarrow & edge(X,Y) \\ & q(X,Y) & \leftarrow & edge(X,Z), q(Z,Y) \\ & edge(X, f(X,Y)) & \leftarrow & v(X,Y) \\ & edge(f(X,Y), Y) & \leftarrow & v(X,Y)\ . \end{array} \tag{30.78}$$

Let the instance of the *edb* relation *edge* of datalog program $\mathcal{P}$ be the graph G shown on Figure 30.4. Then $(\mathcal{P}^-,\mathcal{V}^{-1})$ introduces three new constants, $f(a,c), f(b,d)$ és $f(c,e)$. The *idb* relation *edge* of logic program $\mathcal{V}^{-1}$ is graph G' shown on Figure 30.5. $\mathcal{P}^-$ computes the transitive closure of graph G'. Note that those pairs in th transitive closure that do not contain any of the new constants are exactly the endpoints of even paths of G.

The result of logic program $(\mathcal{P}^-,\mathcal{V}^{-1})$ in Example 30.13. can be calculated by procedure NAÏV-DATALOG, for example. However, it is not true for logic programs in general, that

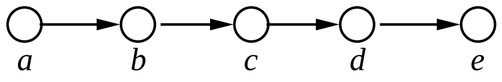

Figure 30.4. The graph G.

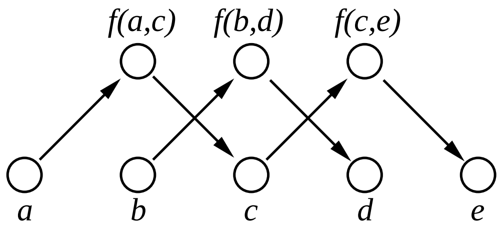

Figure 30.5. The graph G'.

the algorithm terminates. Indeed, consider the logic program

$$\begin{array}{lcl} q(X) & \leftarrow & p(X) \\ q(f(X)) & \leftarrow & q(X)\ . \end{array} \tag{30.79}$$

If the *edb* relation p contains the constant a, then the output of the program is the infinite sequence $a, f(a), f(f(a)), f(f(f(a))), \ldots$. In contrary to this, the output of the logic program $(\mathcal{P}^-, \mathcal{V}^{-1})$ given by the Inverse-rules Algorithm is guaranteed to be finite, thus the computation terminates in finite time.

Theorem 30.31 *For arbitrary datalog program $\mathcal{P}$ and set of conjunctive views $\mathcal{V}$, and for finite instances of the views, there exist a unique minimal fixpoint of the logic program $(\mathcal{P}^-, \mathcal{V}^{-1})$, furthermore procedures* Naïv-datalog *and* Semi-naiv-datalog *give this minimal fixpoint as output.*

The essence of the proof of Theorem 30.31 is that function symbols are only introduced by inverse rules, that are in turn not recursive, thus terms containing nested functions symbols are not produced. The details of the proof are left for the Reader (Exercise 30.3-3.).

Even if started from the *edb* relations of a database, the output of a logic program may contain tuples that have function symbols. Thus, a filter is introduced that eliminates the unnecessary tuples. Let database D be the instance of the *edb*relations of datalog program $\mathcal{P}$. $\mathcal{P}(D)\!\downarrow$ denotes the set of those tuples from $\mathcal{P}(D)$ that do not contain function symbols. Let $\mathcal{P}\!\downarrow$ denote that program, which computes $\mathcal{P}(D)\!\downarrow$ for a given instance D. The proof of the following theorem, exceeds the limitations of the present chapter.

Theorem 30.32 *For arbitrary datalog program $\mathcal{P}$ and set of conjunctive views $\mathcal{V}$, the logic program $(\mathcal{P}^-, \mathcal{V}^{-1})\!\downarrow$ is a maximally contained rewriting of $\mathcal{P}$ using $\mathcal{V}$. Furthermore, $(\mathcal{P}^-, \mathcal{V}^{-1})$ can be constructed in polynomial time of the sizes of $\mathcal{P}$ and $\mathcal{V}$.*

The meaning of Theorem 30.32 is that the simple procedure of adding the inverses of view definitions to a datalog program results in a logic program that uses the views as much as possible. It is easy to see that $(\mathcal{P}^-, \mathcal{V}^{-1})$ can be constructed in polynomial time of the sizes of $\mathcal{P}$ and $\mathcal{V}$, since for every subgoal $v_i \in \mathcal{V}$ a unique inverse rule must be constructed.

In order to completely solve the rewriting problem however, a datalog program needs to be produced that is equivalent with the logic program $(\mathcal{P}^-, \mathcal{V}^{-1})\downarrow$. The key to this is the observation that $(\mathcal{P}^-, \mathcal{V}^{-1})\downarrow$ contains only finitely many function symbols, furthermore during a bottom-up evaluation like Naïv-datalog and its versions, nested function symbols are not produced. With proper book keeping the appearance of function symbols can be kept track, without actually producing those tuples that contain them.

The transformation is done bottom-up like in procedure Naïv-datalog. The function symbol $f(X_1, \ldots, X_k)$ appearing in the *idb* relation of $\mathcal{V}^{-1}$ is replaced by the list of variables $X_1, \ldots, X_k$. At same time the name of the *idb* relation needs to be marked, to remember that the list $X_1, \ldots, X_k$ belongs to function symbol $f(X_1, \ldots, X_k)$. Thus, new "temporary" relation names are introduced. Consider the the rule

$$edge(X, f(X, Y)) \leftarrow v(X, Y) \tag{30.80}$$

of the logic program (30.78) in Example 30.13.. It is replaced by rule

$$edge^{\langle 1, f(2,3)\rangle}(X, X, Y) \leftarrow v(X, Y) \tag{30.81}$$

Notation $\langle 1, f(2, 3)\rangle$ means that the first argument of $edge^{\langle 1, f(2,3)\rangle}$ is the same as the first argument of *edge*, while the second and third arguments of $edge^{\langle 1, f(2,3)\rangle}$ together with function symbol f give the second argument of *edge*. If a function symbol would become an argument of an *idb* relation of $\mathcal{P}^-$ during the bottom-up evaluation of $(\mathcal{P}^-, \mathcal{V}^{-1})$, then a new rule is added to the program with appropriately marked relation names.

Example 30.14 *Transformation of logic program into datalog program.* The logic program Example 30.13. is transformed to the following datalog program by the procedure sketched above. The different phases of the bottom-up execution of Naïv-datalog are separated by lines.

$$\begin{array}{lcl}
edge^{\langle 1, f(2,3)\rangle}(X, X, Y) & \leftarrow & v(X, Y) \\
edge^{\langle f(1,2),3\rangle}(X, Y, Y) & \leftarrow & v(X, Y) \\
\hline
q^{\langle 1, f(2,3)\rangle}(X, Y_1, Y_2) & \leftarrow & edge^{\langle 1, f(2,3)\rangle}(X, Y_1, Y_2) \\
q^{\langle f(1,2),3\rangle}(X_1, X_2, Y) & \leftarrow & edge^{\langle f(1,2),3\rangle}(X_1, X_2, Y) \\
\hline
q(X, Y) & \leftarrow & edge^{\langle 1, f(2,3)\rangle}(X, Z_1, Z_2), q^{\langle f(1,2),3\rangle}(Z_1, Z_2, Y) \\
q^{\langle f(1,2), f(3,4)\rangle}(X_1, X_2, Y_1, Y_2) & \leftarrow & edge^{\langle f(1,2),3\rangle}(X_1, X_2, Z), q^{\langle 1, f(2,3)\rangle}(Z, Y_1, Y_2) \\
\hline
q^{\langle f(1,2),3\rangle}(X_1, X_2, Y) & \leftarrow & edge^{\langle f(1,2),3\rangle}(X_1, X_2, Z), q(Z, Y) \\
q^{\langle 1, f(2,3)\rangle}(X, Y_1, Y_2) & \leftarrow & edge^{\langle 1, f(2,3)\rangle}(X, Z_1, Z_2), q^{\langle f(1,2), f(3,4)\rangle}(Z_1, Z_2, Y_1, Y_2)
\end{array} \tag{30.82}$$

The datalog program obtained shows clearly that which arguments could involve function symbols in the original logic program. However, some rows containing function symbols never give tuples not containing function symbols during the evaluation of the output of the program.

Relation p is called ***significant,*** if in the precedence graph of Definition 30.16[3] there exists oriented path from p to the output relation of q. If p is not significant, then the tuples of p are not needed to compute the output of the program, thus p can be eliminated from the program.

[3]Here the definition of precedence graph needs to be extended for the *edb* relations of the datalog program, as well.

Example 30.15 *Eliminating non-significant relations.* There exists no directed path in the precedence graph of the datalog program obtained in Example 30.14., from relations $q^{\langle 1,f(2,3)\rangle}$ and $q^{\langle f(1,2),f(3,4)\rangle}$ to the output relation q of the program, thus they are not significant, i.e., they can be eliminated together with the rules that involve them. The following datalog program is obtained:

$$\begin{array}{lcl} edge^{\langle 1,f(2,3)\rangle}(X,X,Y) & \leftarrow & v(X,Y) \\ edge^{\langle f(1,2),3\rangle}(X,Y,Y) & \leftarrow & v(X,Y) \\ q^{\langle f(1,2),3\rangle}(X_1,X_2,Y) & \leftarrow & edge^{\langle f(1,2),3\rangle}(X_1,X_2,Y) \\ q^{\langle f(1,2),3\rangle}(X_1,X_2,Y) & \leftarrow & edge^{\langle f(1,2),3\rangle}(X_1,X_2,Z), q(Z,Y) \\ q(X,Y) & \leftarrow & edge^{\langle 1,f(2,3)\rangle}(X,Z_1,Z_2), q^{\langle f(1,2),3\rangle}(Z_1,Z_2,Y)\ . \end{array} \tag{30.83}$$

One more simplification step can be performed, which does not decrease the number of necessary derivations during computation of the output, however avoids redundant data copying. If p is such a relation in the datalog program that is defined by a single rule, which in turn contains a single relation in its body, then p can be removed from the program and replaced by the relation of the body of the rule defining p, having equated the variables accordingly.

Example 30.16 *Avoiding unnecessary data copying.* In Example 30.14. the relations $edge^{\langle 1,f(2,3)\rangle}$ and $edge^{\langle f(1,2),3\rangle}$ are defined by a single rule, respectively, furthermore these two rules both have a single relation in their bodies. Hence, program (30.83) can be simplified further.

$$\begin{array}{lcl} q^{\langle f(1,2),3\rangle}(X,Y,Y) & \leftarrow & v(X,Y) \\ q^{\langle f(1,2),3\rangle}(X,Z,Y) & \leftarrow & v(X,Z), q(Z,Y) \\ q(X,Y) & \leftarrow & v(X,Z), q^{\langle f(1,2),3\rangle}(X,Z,Y)\ . \end{array} \tag{30.84}$$

The datalog program obtained in the two simplification steps above is denoted by $(\mathcal{P}^-,\mathcal{V}^{-1})^{datalog}$. It is clear that there exists a one-to-one correspondence between the bottom-up evaluations of $(\mathcal{P}^-,\mathcal{V}^{-1})$ and $(\mathcal{P}^-,\mathcal{V}^{-1})^{datalog}$. Since the function symbols in $(\mathcal{P}^-,\mathcal{V}^{-1})^{datalog}$ are kept track, it is sure that the output instance obtained is in fact the subset of tuples of the output of $(\mathcal{P}^-,\mathcal{V}^{-1})$ that do not contain function symbols.

Theorem 30.33 *For arbitrary datalog program $\mathcal{P}$ that does not contain negations, and set of conjunctive views $\mathcal{V}$, the logic program $(\mathcal{P}^-,\mathcal{V}^{-1})\downarrow$ is equivalent with the datalog program $(\mathcal{P}^-,\mathcal{V}^{-1})^{datalog}$.*

MiniCon

The main disadvantage of the Bucket Algorithm is that it considers each of the subgoals in isolation, therefore does not observe the most of the interactions between the subgoals of the views. Thus, the buckets may contain many unusable views, and the second phase of the algorithm may become very expensive.

The advantage of the Inverse-rules Algorithm is its conceptual simplicity and modularity. The inverses of the views must be computed only once, then they can be applied to arbitrary queries given by datalog programs. On the other hand, much of the computational advantage of exploiting the materialised views can be lost. Using the resulting rewriting produced by the algorithm for actually evaluating queries from the views has significant drawback, since it insists on recomputing the extensions of the database relations.

The MiniCon algorithm addresses the limitations of the previous two algorithms. The key idea underlying the algorithm is a change of perspective: instead of building rewritings for each of the query ***subgoals,*** it is considered how each of the ***variables*** in the query can interact with the available views. The result is that the second phase of MiniCon needs to consider drastically fewer combinations of views. In the following we return to conjunctive queries, and for the sake of easier understanding only such views are considered that do not contain constants.

The MiniCon algorithm starts out like the Bucket Algorithm, considering which views contain subgoals that correspond to subgoals in the query. However, once the algorithm finds a partial variable mapping from a subgoal g in the query to a subgoal g_1 in a view V, it changes perspective and looks at the variables in the query. The algorithm considers the join predicates in the query – which are specified by multiple occurrences of the same variable – and finds the minimal additional set of subgoals that ***must*** be mapped to subgoals in V, given that g will be mapped to g_1. This set of subgoals and mapping information is called a ***MiniCon Description (MCD)***. In the second phase the MCDs are combined to produce query rewritings. The construction of the MCDs makes the most expensive part of the Bucket Algorithm obsolete, that is the checking of containment between the rewritings and the query, because the generating rule of MCDs makes it sure that their join gives correct result.

For a given mapping $\tau\colon Var(Q) \longrightarrow Var(V)$ subgoal g_1 of view V is said to ***cover*** a subgoal g of query Q, if $\tau(g) = g_1$. $Var(Q)$, and respectively $Var(V)$ denotes the set of variables of the query, respectively of that of the view. In order to prove that a rewriting gives only tuples that belong to the output of the query, a homomorphism must be exhibited from the query onto the rewriting. An MCD can be considered as a part of such a homomorphism, hence, these parts will be put together easily.

The rewriting of query Q is a union of conjunctive queries using the views. Some of the variables may be equated in the heads of some of the views as in the equivalent rewriting (30.70) of Example 30.10.. Thus, it is useful to introduce the concept of ***head homomorphism***. The mapping $h\colon Var(V) \longrightarrow Var(V)$ is a ***head homomorphism,*** if it is an identity on variables that do not occur in the head of V, but it can equate variables of the head. For every variable x of the head of V, $h(x)$ also appear in the head of V, furthermore $h(x) = h(h(x))$. Now, the exact definition of MCD can be given.

Definition 30.34 *The quadruple $C = (h_C, V(\tilde{Y})_C, \varphi_C, G_C)$ is a* ***MiniCon Description (MCD)*** *for query Q over view V, where*

- *h_C is a head homomorphism over V,*
- *$V(\tilde{Y})_C$ is obtained from V by applying h_C, that is $\tilde{Y} = h_C(\tilde{A})$, where $\tilde{A}$ is the set of variables appearing in the head of V,*
- *φ_C is a partial mapping from $Var(Q)$ to $h_C(Var(V))$,*
- *G_C is a set of subgoals of Q that are covered by some subgoal of $H_C(V)$ using the mapping φ_C (note: not all such subgoals are necessarily included in G_C).*

The procedure constructing MCDs is based on the following proposition.

Proposition 30.35 *Let C be a MiniCon Description over view V for query Q. C can be used for a non-redundant rewriting of Q if the following conditions hold*

C1. *for every variable x that is in the head of Q and is in the domain of φ_C, as well, $\varphi_C(x)$ appears in the head of $h_C(V)$, furthermore*

C2. *if $\varphi_C(y)$ does not appear in the head of $h_C(V)$, then for all such subgoals of Q that contain y holds that*

1. *every variable of g appears in the domain of φ_C and*
2. *$\varphi_C(g) \in h_C(V)$.*

Clause **C1** is the same as in the Bucket Algorithm. Clause **C2** means that if a variable x is part of a join predicate which is not enforced by the view, then x must be in the head of the view so the join predicate can be applied by another subgoal in the rewriting. The procedure Form-MCDs gives the usable MiniCon Descriptions for a conjunctive query Q and set of conjunctive views $\mathcal{V}$.

```
Form-MCDs($Q, \mathcal{V}$)
 1  $C \leftarrow \emptyset$
 2  for each subgoal $g$ of $Q$
 3      do for $V \in \mathcal{V}$
 4          do for every subgoal $v \in V$
 5              do Let $h$ be the least restrictive head homomorphism on $V$,
                   such that there exists a mapping $\varphi$ with $\varphi(g) = h(v)$.
 6                 if $\varphi$ and $h$ exist
 7                   then Add to $C$ any new MCD $C$, that can be constructed where:
 8                        (a) $\varphi_C$ (respectively, $h_C$) is an extension of $\varphi$ (respectively, $h$),
 9                        (b) $G_C$ is the minimal subset of subgoals of $Q$ such that
                              $G_C$, $\varphi_C$ and $h_C$ satisfy Proposition 30.35, and
10                        (c) It is not possible to extend $\varphi$ and $h$ to $\varphi'_C$ and $h'_C$ such that
                              (b) is satisfied, and $G'_C$ as defined in (b), is a subset of $G_C$.
11  return $C$
```

Consider again query (30.68) and the views of Example 30.10.. Procedure Form-MCDs considers subgoal $cite(x, y)$ of the query first. It does not create an MCD for view V_1, because clause **C2** of Proposition 30.35 would be violated. Indeed, the condition would require that subgoal $sameArea(x, y)$ be also covered by V_1 using the mapping $\varphi(x) = a$, $\varphi(y) = b$, since is not in the head of V_1.[4] For the same reason, no MCD will be created for V_1 even when the other subgoals of the query are considered. In a sense, the MiniCon Algorithm shifts some of the work done by the combination step of the Bucket Algorithm to the phase of creating the MCDs by using Form-MCDs. The following table shows the output of procedure Form-MCDs.

$V(\tilde{Y})$	h	φ	G
$V_2(c, d)$	$c \to c, d \to d$	$x \to c, y \to d$	3
$V_3(f, f)$	$f \to f, h \to f$	$x \to f, y \to f$	1, 2, 3

(30.85)

Procedure Form-MCDs includes in G_C only the ***minimal*** set of subgoals that are necessary in order to satisfy Proposition 30.35. This makes it possible that in the second phase of

[4]The case of $\varphi(x) = b$, $\varphi(y) = a$ is similar.

the MiniCon Algorithm needs only to consider combinations of MCDs that cover ***pairwise disjoint subsets*** of subgoals of the query.

Proposition 30.36 *Given a query Q, a set of views $\mathcal{V}$, and the set of MCDs C for Q over the views $\mathcal{V}$, the only combinations of MCDs that can result in non-redundant rewritings of Q are of the form $C_1, \dots C_l$, where*

C3. *$G_{C_1} \cup \dots \cup G_{C_l}$ contains all the subgoals of Q, and*

C4. *for every $i \neq j$ $G_{C_i} \cap G_{C_j} = \emptyset$.*

The fact that only such sets of MCDs need to be considered that provide partitions of the subgoals in the query reduces the search space of the algorithm drastically. In order to formulate procedure COMBINE-MCDs, another notation needs to be introduced. The φ_C mapping of MCD C may map a set of variables of Q onto the same variable of $h_C(V)$. One arbitrarily chosen representative of this set is chosen, with the only restriction that if there exists variables in this set from the head of Q, then one of those is the chosen one. Let $EC_{\varphi_C}(x)$ denote the representative variable of the set containing x. The MiniCon Description C is considered extended with $EC_{\varphi_C}(x)$ in he following as a quintet $(h_C, V(\tilde{Y}), \varphi_C, G_C, EC_{\varphi_C})$. If the MCDs $C_1, \dots, C_k$ are to be combined, and for some $i \neq j$ $EC_{\varphi_{C_i}}(x) = EC_{\varphi_{C_i}}(y)$ and $EC_{\varphi_{C_j}}(y) = EC_{\varphi_{C_j}}(z)$ holds, then in the conjunctive rewriting obtained by the join x, y and z will be mapped to the same variable. Let S_C denote the equivalence relation determined on the variables of Q by two variables being equivalent if they are mapped onto the same variable by φ_C, that is, $xS_Cy \iff EC_{\varphi_C}(x) = EC_{\varphi_C}(y)$. Let C be the set of MCDs obtained as the output of FORM-MCDs.

```
COMBINE-MCDs($C$)
 1  $Answer \leftarrow \emptyset$
 2  for $\{C_1, \dots, C_n\} \subseteq C$ such that $G_{C_1}, \dots, G_{C_n}$ is a partition of the subgoals of $Q$
 3     do Define a mapping $\Psi_i$ on $\tilde{Y}_i$ as follows:
 4        if there exists a variable $x$ in $Q$ such that $\varphi_i(x) = y$
 5           then $\Psi_i(y) = x$
 6           else $\Psi_i(y)$ is a fresh copy of $y$
 7     Let $S$ be the transitive closure of $S_{C_1} \cup \dots \cup S_{C_n}$
 8                                         ▷ $S$ is an equivalence relation of variables of $Q$.
 9     Choose a representative for each equivalence class of $S$.
10     Define mapping $EC$ as follows:
11     if $x \in Var(Q)$
12        then $EC(x)$ is the representative of the equivalence class of $x$ under $S$
13        else $EC(x) = x$
14     Let $Q'$ be given as $Q'(EC(\tilde{X})) \leftarrow V_{C_1}(EC(\Psi_1(\tilde{Y}_1))), \dots, V_{C_n}(EC(\Psi_n(\tilde{Y}_n)))$
15     $Answer \leftarrow Answer \cup \{Q'\}$
16  return $Answer$
```

The following theorem summarises the properties of the MiniCon Algorithm.

Theorem 30.37 *Given a conjunctive query Q and conjunctive views $\mathcal{V}$, both without comparison predicates and constants, the MiniCon Algorithm produces the union of conjunctive queries that is a maximally contained rewriting of Q using $\mathcal{V}$.*

The complete proof of Theorem 30.37 exceeds the limitations of the present chapter. However, in Problem *30-1.* the reader is asked to prove that union of the conjunctive queries obtained as output of Combine-MCDs is contained in Q.

It must be noted that the running times of the Bucket Algorithm, the Inverse-rules Algorithm and the MiniCon Algorithm are the same in the worst case: $O(nmM^n)$, where n is the number of subgoals in the query, m is the maximal number of subgoals in a view, and M is the number of views. However, practical test runs show that in case of large number of views (3–400 views) the MiniCon Algorithm is significantly faster than the other two.

Exercises

30.3-1 Prove Theorem 30.25 using Proposition 30.24 and Theorem 30.20.

30.3-2 Prove the two statements of Lemma 30.26. *Hint.* For the first statement, write in their definitions in place of views $v_i(\tilde{Y}_i)$ into Q'. Minimise the obtained query Q'' using Theorem 30.19. For the second statement use Proposition 30.24 to prove that there exists a homomorphism h_i from the body of the conjunctive query defining view $v_i(\tilde{Y}_i)$ to the body of Q. Show that $\tilde{Y'}_i = h_i(\tilde{Y}_i)$ is a good choice.

30.3-3 Prove Theorem 30.31 using that datalog programs have unique minimal fixpoint.

Problems

30-1. MiniCon is correct

Prove that the output of the MiniCon Algorithm is correct. *Hint.* It is enough to show that for each conjunctive query Q' given in line 14 of Combine-MCDs $Q' \sqsubseteq Q$ holds. For the latter, construct a homomorphism from Q to Q'.

30-2. $(\mathcal{P}^-, \mathcal{V}^{-1})\downarrow$ ***is correct***

Prove that each tuple produced by logic program $(\mathcal{P}^-, \mathcal{V}^{-1})\downarrow$ is contained in the output of $\mathcal{P}$ (part of the proof of Theorem 30.32). *Hint.* Let t be a tuple in the output of $(\mathcal{P}^-, \mathcal{V}^{-1})$ that does not contain function symbols. Consider the derivation tree of t. Its leaves are literals, since they are extensional relations of program $(\mathcal{P}^-, \mathcal{V}^{-1})$. If these leaves are removed from the tree, then the leaves of the remaining tree are *edb* relations of $\mathcal{P}$. Prove that the tree obtained is the derivation tree of t in datalog program $\mathcal{P}$.

30-3. Datalog views

This problem tries to justify why only conjunctive views were considered. Let $\mathcal{V}$ be a set of views, Q be a query. For a given instance $\mathcal{I}$ of the views the tuple t is a ***certain answer*** of query Q, if for any database instance $\mathcal{D}$ such that $\mathcal{I} \subseteq \mathcal{V}(\mathcal{D})$, $t \in Q(\mathcal{D})$ holds, as well.

a. Prove that if the views of $\mathcal{V}$ are given by datalog programs, query Q is conjunctive and may contain non-equality ($\neq$) predicates, then the question whether for a given instance $\mathcal{I}$ of the views tuple t is a certain answer of Q is algorithmically undecidable. *Hint.* Reduce to this question the ***Post Correspondence Problem,*** which is the following: Given two sets of words $\{w_1, w_2, \ldots, w_n\}$ and $\{w'_1, w'_2, \ldots, w'_n\}$ over the alphabet $\{a, b\}$. The question is whether there exists a sequence of indices $i_1, i_2, \ldots, i_k$ (repetition allowed) such that

$$w_{i_1} w_{i_2} \cdots w_{i_k} = w'_{i_1} w'_{i_2} \cdots w'_{i_k} \ . \qquad (30.86)$$

The Post Correspondence Problem is well known algorithmically undecidable problem.

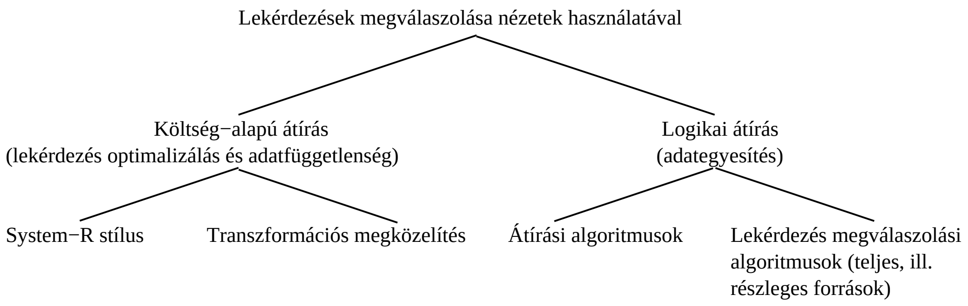
Lekérdezések megválaszolása nézetek használatával
Költség−alapú átírás
(lekérdezés optimalizálás és adatfüggetlenség)
Logikai átírás
(adategyesítés)
System−R stílus
Transzformációs megközelítés
Átírási algoritmusok
Lekérdezés megválaszolási algoritmusok (teljes, ill. részleges források)

Figure 30.6. A taxonomy of work on answering queries using views.

Let the view V be given by the following datalog program:

$$\begin{array}{lcl} V(0,0) & \leftarrow & S(e,e,e) \\ V(X,Y) & \leftarrow & V(X_0,Y_0), S(X_0,X_1,\alpha_1),\ldots,S(X_{g-1},Y,\alpha_g), \\ & & S(Y_0,Y_1,\beta_1),\ldots,S(Y_{h-1},Y,\beta_h) \\ & & \text{where } w_i=\alpha_1\ldots\alpha_g \text{ and } w'_i=\beta_1\ldots\beta_h \\ & & \text{is a rule for all } i\in\{1,2,\ldots,n\} \\ S(X,Y,Z) & \leftarrow & P(X,X,Y), P(X,Y,Z)\ . \end{array} \qquad (30.87)$$

Furthermore, let Q be the following conjunctive query.

$$Q(c) \leftarrow P(X,Y,Z), P(X,Y,Z'), Z \neq Z'\ . \qquad (30.88)$$

Show that for the instance $\mathcal{I}$ of V that is given by $\mathcal{I}(V)=\{\langle e,e\rangle\}$ and $\mathcal{I}(S)=\{\}$, the tuple $\langle c\rangle$ is a certain answer of query Q if and only if the Post Correspondence Problem with sets $\{w_1,w_2,\ldots,w_n\}$ and $\{w'_1,w'_2,\ldots,w'_n\}$ has *no* solution.

b. In contrast to the undecidability result of *a.*, if $\mathcal{V}$ is a set of conjunctive views and query Q is given by datalog program $\mathcal{P}$, then it is easy to decide about an arbitrary tuple t whether it is a certain answer of Q for a given view instance $\mathcal{I}$. Prove that the datalog program $(\mathcal{P}^-,\mathcal{V}^{-1})^{datalog}$ gives exactly the tuples of the certain answer of Q as output.

Chapter notes

There are several dimensions along which the treatments of the problem "answering queries using views" can be classified. Figure 30.6 shows the taxonomy of the work.

The most significant distinction between the different work s is whether their goal is data integration or whether it is query optimisation and maintenance of physical data independence. The key difference between these two classes of works is the output of the the algorithm for answering queries using views. In the former case, given a query Q and a set of views $\mathcal{V}$, the goal of the algorithm is to produce an expression Q' that references the views and is either equivalent to or contained in Q. In the latter case, the algorithm must go further and produce a (hopefully optimal) query execution plan for answering Q using the views (and possibly the database relations). Here the rewriting must be an equivalent to Q in order to ensure the correctness of the plan.

The similarity between these two bodies of work is that they are concerned with the core issue of whether a rewriting of a query is equivalent or contained in the query. However, while logical correctness suffices for the data integration context, it does not in the query optimisation context where we also need to find the *cheapest* plan using the views. The complication arises because the optimisation algorithms need to consider views that do not contribute to the *logical* correctness of the rewriting, but do reduce the cost of the resulting plan. Hence, while the reasoning underlying the algorithms in the data integration context is mostly logical, in the query optimisation case it is both logical and cost-based. On the other hand, an aspect stressed in data integration context is the importance of dealing with a large number of views, which correspond to data sources. In the context of query optimisation it is generally assumed (not always!) that the number of views is roughly comparable to the size of the schema.

The works on query optimisation can be classified further into System-R style optimisers and transformational optimisers. Examples of the former are works of Chaudhuri, Krishnamurty, Potomianos and Shim [6]; Tsatalos, Solomon, and Ioannidis [26]. Papers of Florescu, Raschid, and Valduriez [12]; Bello et. al. [3]; Deutsch, Popa and Tannen [8], Zaharioudakis et. al. [30], furthermore Goldstein és Larson[15] belong to the latter.

Rewriting algorithms in the data integration context are studied in works of Yang and Larson [28]; Levy, Mendelzon, Sagiv and Srivastava [20]; Qian [25]; furthermore Lambrecht, Kambhampati and Gnanaprakasam [22]. The Bucket Algorithm was introduced by Levy, Rajaraman and Ordille [17]. The Inverse-rules Algorithm is invented by Duschka and Genesereth [9, 10]. The MiniCon Algorithm was developed by Pottinger and Halevy [24, 23].

Query answering algorithms and the complexity of the problem is studied in papers of Abiteboul and Duschka [2]; Grahne and Mendelzon [16]; furthermore Calvanese, De Giacomo, Lenzerini and Vardi [5].

The STORED system was developed by Deutsch, Fernandez and Suciu [7]. Semantic caching is discussed in the paper of Yang, Karlapalem and Li [29]. Extensions of the rewriting problem are studied in [4, 13, 14, 21, 29].

Surveys of the area can be found in works of Abiteboul [1], Florescu, Levy and Mendelzon [11], Halevy [18, 19], furthermore Ullman[27].

Research of the authors was (partially) supported by Hungarian National research Fund (OTKA) grants Nos. T034702, T037846T and T042706.

Bibliography

[1] S. Abiteboul. Querying semi-structured data. In F. Afrati, P. Kolaitis (szerkesztők), *Proceedings of ICDT'97*, *Lecture Notes in Computer Science* 1186. kötete, 1–18. o. Springer-Verlag, 1997. 1467

[2] S. Abiteboul, O. Duschka. Complexity of answering queries using materialized views. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 254–263. o. ACM-Press, 1998. 1467

[3] R. Bello, K. Dias, A. Downing, J. Freenan, T. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, M. Ziauddin. Materialized views in Oracle. In *Proceedings of Very Large Data Bases'98*, 659–664. o., 1998. 1467

[4] P. Buneman. Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 117–121. o. ACM-Press, 1997. 1467

[5] D. Calvanese, D. De Giacomo, M. Lenzerini, M. Varni. Answering regular path queries using views. In *Proceedings of the Sixteenth International Conference on Data Engineering*, 190–200. o., 2000. 1467

[6] S. Chaudhury, R. Krishnamurty, S. Potomianos, K. Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*, 190–200. o., 1995. 1467

[7] A. Deutsch, M. Fernandez, D. Suciu. Storing semistructured data with stored. In *Proceedings of SIGMOD'99*, 431–442. o., 1999. 1467

[8] A. Deutsch, L. Popa, D. Tannen. Physical data independence, constraints and optimization with universal plans. In *Proceedings of VLDB'99*, 459–470. o., 1999. 1467

[9] O. Duschka, M. Genesereth. Answering recursive queries using views. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 109–116. o. ACM-Press, 1997. 1467

[10] O. Duschka, M. Genesereth. Query planning in infomaster. In *Proceedings of ACM Symposium on Applied Computing*, 109–111. o. ACM-Press, 1997. 1467

[11] D. Florescu, A. Halevy, A. O. Mendelzon. Database techniques for the world-wide web: a survey. *SIGMOD Record*, 27(3):59–74, 1998. 1467

[12] D. Florescu, L. Raschid, P. Valduriez. Answering queries using oql view expressions. In *Workshop on Materialized views, in cooperation with ACM SIGMOD*, 627–638. o., 1996. 1467

[13] D. D. Florescu. *Search spaces for object-oriented query optimization*. PhD thesis, University of Paris VI, 1996. 1467

[14] M. Friedman, D. S. Weld. Efficient execution of information gathering plans. In *Proceedings International Joint Conference on Artificial Intelligence*, 785–791. o., 1997. 1467

[15] J. Goldstein, P. A. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *Optimizing queries using materialized views: a practical, scalable solution*, 331–342. o., 2001. 1467

[16] G. Grahne, A. Mendelzon. Tableau techniques for querying information sources through global schemas. In *Proceedings of ICDT'99*, *Lecture Notes in Computer Science* 1540. kötete, 332–347. o. Springer-Verlag, 1999. 1467

[17] A. Halevy, A. Rajaraman, J. J. Ordille, D. Srivastava. Querying heterogeneous information sources using source descriptions. In *Proceedings of Very Large Data Bases*, 251–262. o., 1996. 1467

[18] A. Halevy. Logic based techniques in data integration. In J. Minker (szerkesztő), *Logic-based Artificial Intelligence*, 575–595. o. Kluwer Academic Publishers, 2000. 1467

[19] A. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10:270–294, 2001. 1467

[20] A. Halevy, A. Mendelzon, Y. Sagiv, D. Srivastava. Answering queries using views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 95–104. o. ACM-Press, 1995. 1467

[21] C. T. Kwok, D. Weld. Planning to gather information. In *Proceedings of AAAI 13th National Conference on Artificial Intelligence*, 32–39. o., 1996. 1467

[22] E. T. Lambrecht, S. Kambhampati, S. Gnanaprakasam. Optimizing recursive information gathering plans. In *Proceedings of 16th International Joint Conference on Artificial Intelligence*, 1204–1211. o., 1999. 1467

[23] R. Pottinger. MinCon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2):182–198, 2001. 1467

[24] R. Pottinger, A. Halevy. A scalable algorithm for answering queries using views. In *Proceedings of Very Large Data Bases'00*, 484–495. o., 2000. 1467

[25] X. Qian. Query folding. In *Proceedings of International Conference on Data Engineering*, 48–55. o., 1996. 1467

[26] O. G. Tsatalos, M. C. Solomon, Y. Ioannidis. The GMAP: a versatile tool for physical data independence. *The VLDB Journal*, 5(2):101–118, 1996. 1467

[27] J. D. Ullman. Information integration using logical views. In *Proceedings of ICDT'97*, *Lecture Notes in Computer Science* 1186. kötete, 19–40. o. Springer-Verlag, 1997. 1467

[28] H. Z. Yang, P. A. Larson. Query transformation for PSJ-queries. In *Proceedings of Very Large Data Bases'87*, 245–254. o., 1987. 1467

[29] J. Yang, K., Karlapalem, Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proceedings of Very Large Data Bases'97*, 136–145. o., 1997. 1467

[30] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, M. Urata. Answering complex SQL queries using automatic summary tables. In *Proceedings of SIGMOD'00*, 105–116. o., 2000. 1467

Name index

Subject Index

Contents