The active domain of a database d over $\mathbf{R}$, denoted by ADOM(d), is defined by

$$\text{ADOM}(d) = \bigcup \{\text{ADOM}(r) \mid r \in d\}.$$ ■

3.2 Query and Update Languages for the Relational Model

In the previous section we have concerned ourselves with the structural part of the relational model. Here we elaborate on the manipulative part of the relational model, in the form of query and update languages. In Subsection 3.2.1 we present the relational algebra which is a procedural query language for the relational model. In Subsection 3.2.2 we present the domain relational calculus which is the declarative counterpart of the relational algebra and is based on the first-order predicate calculus. The domain relational calculus is the query language which provides the theoretical underpinning of SQL which is the commercial relational query language used in most DBMSs; in fact, many people go further and equate SQL with relational databases. In Subsection 3.2.3 we present Datalog which is a rule-based query language for the relational model and views a relational database in a logical way. In Subsection 3.2.4 we consider an update language for the relational model that takes into account the dynamic aspects of updating a relational database; it complements the query languages we present in Subsections 3.2.1, 3.2.2 and 3.2.3, which can only be used to retrieve information from a relational database.

3.2.1 The Relational Algebra

The relational algebra is a collection of operators; each operator takes as input either a single relation or a pair of relations and outputs a single relation as its result. A relational query is a composition of a finite number of relational operators. In that sense a query is procedural, since it specifies the order in which the operators comprising the query are to be evaluated. The declarative counterpart of the relational algebra, i.e. the domain relational calculus, is defined in Subsection 3.2.2. The relational algebra was first presented in Codd's seminal 1970 paper [Cod70] and a variant of the domain relational calculus was first presented in 1972 in another of Codd's fundamental papers [Cod72b]. Since then the relational algebra has become a yardstick for measuring the expressiveness of any relational query language.

Our style of presentation of the relational algebra operators is: for each operator we give an informal definition of the operator, then we give its formal definition and finally we give an example of its use over the database presented in Section 3.1.

The set-theoretic operators, union, difference and intersection, defined below, are all binary operators which take two relations over a common relation schema, say R, and return a relation over R. It is customary to call two relations *union-compatible* if their corresponding relation schemas have the same attribute set, and are thus effectively the same.

The union of two relations, r_1 and r_2 over relation schema R, is the set of tuples that are either in r_1 or in r_2.

Definition 3.10 (Union) The union, $\cup$, of two relations r_1 and r_2 over R is defined by

$$r_1 \cup r_2 = \{t \mid t \in r_1 \text{ or } t \in r_2\}.$$ ■

Let s_1 be the relation over SHORT_STUD with schema(SHORT_STUD) = {SNAME, ADDRESS, DEPT}, shown in Table 3.4, representing students having computing accounts, and let s_2 be the relation over SHORT_STUD, shown in Table 3.5, representing students receiving a grant. The query, "Retrieve the students who either have a computing account or are receiving a grant", can be expressed as the union $s_1 \cup s_2$. The result of this query is shown in Table 3.6.

Table 3.4 The relation s_1 over SHORT_STUD

SNAME	ADDRESS	DEPT
Iris	Malet St	Computing
Reuven	Harold Rd	Maths
Hanna	Harold Rd	Linguistics
Brian	Alexandra Rd	Sociology

Table 3.5 The relation s_2 over SHORT_STUD

SNAME	ADDRESS	DEPT
Iris	Malet St	Computing
Reuven	Harold Rd	Maths
Annette	Harold Rd	Linguistics
Cyril	Oakley Gdns	Medicine

Table 3.6 The result of the query $s_1 \cup s_2$ over SHORT_STUD

SNAME	ADDRESS	DEPT
Iris	Malet St	Computing
Reuven	Harold Rd	Maths
Hanna	Harold Rd	Linguistics
Annette	Harold Rd	Linguistics
Brian	Alexandra Rd	Sociology
Cyril	Oakley Gdns	Medicine

The difference between two relations, r_1 and r_2 over relation schema R, is the set of tuples that are in r_1 but not in r_2.

Definition 3.11 (Difference) The difference, $-$, of two relations r_1 and r_2 over R is defined by

$$r_1 - r_2 = \{t \mid t \in r_1 \text{ and } t \notin r_2\}.$$ ■

The query, "Retrieve the students who have a computing account but do not receive a grant", can be expressed as the difference $s_1 - s_2$, where s_1 and s_2 are shown in Tables 3.4 and 3.5, respectively. The result of this query is shown in Table 3.7.

We note that the intersection, $\cap$, of two relations r_1 and r_2 over relation schema R, i.e. the set of tuples that are included in both r_1 and r_2, can be defined in terms of the difference operator by

$$r_1 \cap r_2 = r_1 - (r_1 - r_2).$$

Table 3.7 The result of the query $s_1 - s_2$ over SHORT_STUD

SNAME	ADDRESS	DEPT
Hanna	Harold Rd	Linguistics
Brian	Alexandra Rd	Sociology

The query, "Retrieve the students who have a computing account and are also receiving a grant", can be expressed as the intersection $s_1 \cap s_2$, where s_1 and s_2 are shown in Tables 3.4 and 3.5, respectively. The result of this query is shown in Table 3.8.

Table 3.8 The result of the query $s_1 \cap s_2$ over SHORT_STUD

SNAME	ADDRESS	DEPT
Iris	Malet St	Computing
Reuven	Harold Rd	Maths

The projection of a relation r over relation schema R onto a set of attributes Y included in schema(R) is the set of tuples resulting from projecting each of the tuples in r onto Y.

Definition 3.12 (Projection) The projection, π, of a relation r over relation schema R onto a set of attributes $Y \subseteq$ schema(R) is defined by

$$\pi_{Y}(r) = \{t[Y] \mid t \in r\},$$

where $t[Y]$ is the restriction of t to Y given in Definition 3.8. ■

The query, "Retrieve the departments, degrees and years of students", can be expressed as the projection $\pi_{\{DEPT, DEGREE, YEAR\}}(r_1)$, where r_1 is shown in Table 3.1. The result of this query is shown in Table 3.9.

Table 3.9 The result of the query $\pi_{\{DEPT, DEGREE, YEAR\}}(r_1)$

DEPT	DEGREE	YEAR
Computing	BSC	first
Computing	BSC	third
Maths	BSC	second
Maths	BA	fourth
Linguistics	BA	second
Economics	BCOM	third

We note that the cardinality of $\pi_Y(r)$ is less than or equal to the cardinality of r, since two or more tuples in r may have the same projection onto Y. For example, in the above query the tuple <Linguistics, BA, second> is the {DEPT, DEGREE, YEAR}-value of both the tuples, whose SNAME-values are Hanna and Dan in r_1 of Table 3.1. We further note that projection captures the semantics of existential quantification. For example, in the above query we retrieved the {DEPT, DEGREE, YEAR}-values of tuples such that there exist {SNAME, AGE, ADDRESS}-values for these tuples.

Selection of tuples from a relation r with respect to a selection formula F is the subset of tuples from r that satisfy the formula F.

Definition 3.13 (Selection formula) A *simple selection formula* over a schema R is either an expression of the form A = a or an expression of the form A = B, where A, B $\in$ schema(R) and $a \in$ DOM(A).

A *selection formula* (or simply a formula whenever no ambiguity arises) over R is a well-formed expression composed of one or more simple selection formulae over R together with the Boolean logical connectives: $\wedge$ (and), $\vee$ (or), $\neg$ (not) and parentheses. A selection formula is called *positive* if it does not have any occurrence of $\neg$. We abbreviate $\neg(A = a)$ by $A \neq a$ and $\neg(A = B)$ by $A \neq B$. ■

A simple selection formula of the type A = B is sometimes referred to as *restriction*. For simplicity we have only included equality (=) as a comparison operator but, in general, we can also expect $\leq$ (less than or equal to) and $<$ (less than) to be available in simple selection formulae.

Informally a tuple, t, logically implies a formula, F, if the tuple satisfies F. In the next definition F_1 and F_2 are also formulae.

Definition 3.14 (Logical implication, $\models$) Let r be a relation over relation schema R, t be a tuple in r and, in addition, let F be a selection formula over R. Then t *logically implies* F, written $t \models F$, is defined recursively, as follows:

1) $t \models A = a$, if $t[A] = a$ evaluates to true.

2) $t \models A = B$, if $t[A] = t[B]$ evaluates to true.

3) $t \models F_1 \wedge F_2$, if $t \models F_1$ evaluates to true and $t \models F_2$ evaluates to true.

4) $t \models F_1 \vee F_2$, if $t \models F_1$ evaluates to true or $t \models F_2$ evaluates to true.

5) $t \models \neg F$, if $t \models F$ does not evaluate to true, i.e. $t \not\models F$.

6) $t \models (F)$, if $t \models F$. ■

We are now ready to formalise selection which, when applied to a relation, r, with respect to a formula, F, returns all the tuples in r that logically imply F.

Definition 3.15 (Selection) The selection, σ, applied to a relation r over relation schema R with respect to a selection formula F over R is defined by

$$\sigma_F(r) = \{t \mid t \in r \text{ and } t \models F\}.$$ ■

The following theorem, which easily follows from the definitions of selection and logical implication, shows that the Boolean logical connectives, $\neg$, $\vee$ and $\wedge$, present in selection formulae can be expressed in terms of the set operators $-$, $\cup$ and $\cap$, respectively.

Theorem 3.1 The following equalities are all satisfied, where r is a relation over relation schema R, and F, F_1 and F_2 are selection formulae over R.

1) $\sigma_{\neg F}(r) = r - \sigma_F(r)$.

2) $\sigma_{F_1 \vee F_2}(r) = \sigma_{F_1}(r) \cup \sigma_{F_2}(r)$.

3) $\sigma_{F_1 \wedge F_2}(r) = \sigma_{F_1}(r) \cap \sigma_{F_2}(r)$. □

The query, "Retrieve the students who are either studying in the Linguistics department or whose address is Oxford St", can be expressed as the selection $\sigma_{F_1}(r_1)$, where F_1 is the formula DEPT = 'Linguistics' $\vee$ ADDRESS = 'Oxford St', and r_1 is shown in Table 3.1. The result of this query is shown in Table 3.10.

Table 3.10 The result of the query $\sigma_{F_1}(r_1)$

SNAME	AGE	ADDRESS	DEPT	DEGREE	YEAR
Hanna	31	Harold Rd	Linguistics	BA	second
Dan	34	Gower St	Linguistics	BA	second
Eli	38	Oxford St	Economics	BCOM	third
Naomi	39	Oxford St	Maths	BA	fourth

The query, "Retrieve the students who are not studying Computing and are not in their second year", can be expressed as the selection $\sigma_{F_2}(r_1)$, where F_2 is the formula DEPT $\neq$ 'Computing' $\wedge$ YEAR $\neq$ 'second', and r_1 is shown in Table 3.1. The result of this query is shown in Table 3.11.

Table 3.11 The result of the query $\sigma_{F_2}(r_1)$

SNAME	AGE	ADDRESS	DEPT	DEGREE	YEAR
Eli	38	Oxford St	Economics	BCOM	third
Naomi	39	Oxford St	Maths	BA	fourth

The query, "Retrieve the students who did the same number of courses in their first and second years", can be expressed as the selection $\sigma_{F_3}(s_3)$, where F_3 is the formula FIRST = SECOND, and s_3 is the relation over FST_SND shown in Table 3.12. The result of this query is shown in Table 3.13.

Table 3.12 The relation s_3 over FST_SND

SNAME	FIRST	SECOND
Reuven	5	5
Hanna	4	5
Dan	5	4
Hillary	4	4
Eli	3	6
Naomi	6	5

Informally, the natural join of two relations r_1 over relation schema R_1 and r_2 over relation schema R_2, with schema(R_1) $\cap$ schema(R_2) being the set of attributes X, is the relation

Table 3.13 The result of the query $\sigma_{F_3}(s_3)$

SNAME	FIRST	SECOND
Reuven	5	5
Hillary	4	4

containing tuples that result from concatenating every tuple of r_1 with every tuple of r_2 both of which have the same X-values. The attributes in X are called the *join attributes* of R_1 and R_2.

Definition 3.16 (Natural join) The natural join (or simply the join), $\bowtie$, of two relations r_1 over relation schema R_1 and r_2 over relation schema R_2 is a relation over relation schema R defined by

$$r_1 \bowtie r_2 = \{t \mid \exists t_1 \in r_1 \text{ and } \exists t_2 \in r_2 \text{ such that } t[\text{schema}(R_1)] = t_1 \text{ and } t[\text{schema}(R_2)] = t_2\},$$

where schema(R) = schema(R_1) ∪ schema(R_2). ■

Let $s_1 = \pi_{\{\text{CNAME,DEPT,TNAME}\}}(r_2)$ be the relation shown in Table 3.14 and $s_2 = \pi_{\{\text{DEPT,TNAME,SALARY}\}}(r_3)$ be the relation shown in Table 3.15. The query, "Retrieve the courses given in departments and the salaries of tutors for these courses", can be expressed as the natural join $s_1 \bowtie s_2$.

The result of this query is shown in Table 3.16 We observe that the tuples <Philosophy, Martine, 1600> and <Linguistics, Ruth, 1100> in s_2 did not participate in the join, since their {DEPT, TNAME}-values do not match any corresponding values in s_1. Furthermore, the tuple <algorithms, Computing, Ada> in s_1 did not participate in the join, since its {DEPT, TNAME}-value does not match any corresponding values in s_2. Such tuples are known as *dangling* tuples.

There is a connection between the concept of dangling tuples and referential integrity, which was introduced in Subsection 1.7.1 of Chapter 1. For example, suppose that the set of attributes {DEPT, TNAME} forms the primary key for the schema of relation s_2. In this case the attributes {DEPT, TNAME} of the schema of relation s_1 form a foreign key which references these attributes in the schema of s_2. It follows that the tuple <algorithms, Computing, Ada> is dangling as a result of referential integrity being violated. More specifically, if referential integrity is satisfied then there must exist a tuple t in s_2 such that t[DEPT] = Computing and t[TNAME] = Ada. If, in addition, the set of attributes {DEPT, TNAME} were the primary key for the schema of s_1, then the tuples in s_2 that did not participate in the join are also dangling as a result of referential integrity being violated. (It is unlikely that {DEPT, TNAME} is the primary key for the schema of s_1, since one would expect a teacher to teach more than one course in a given department.)

The query, "Retrieve the courses that students can do in the department they are studying in", can be expressed as the natural join $\pi_{\{\text{CNAME,DEPT}\}}(s_1) \bowtie \pi_{\{\text{DEPT,SNAME}\}}(r_1)$, where s_1 and r_1 are shown in Tables 3.14 and 3.1, respectively. The result of this query is shown in Table 3.17.

It may be easier to understand the semantics of the natural join algorithmically rather than by the above declarative definition. The pseudo-code of an algorithm, designated JOIN(r_1, r_2), which given the input relations r_1 over R_1 and r_2 over R_2, with X = schema(R_1) ∩ schema(R_2), returns $r_1 \bowtie r_2$ over R, is presented as the ensuing algorithm.

Table 3.14 The relation $s_1 = \pi_{\{CNAME, DEPT, TNAME\}}(r_2)$

CNAME	DEPT	TNAME
databases	Computing	Robert
programming	Computing	Hanna
programming	Computing	Richard
algorithms	Computing	Ada
logic	Maths	Reuven
graph-theory	Maths	Martine
Hebrew	Linguistics	Dan

Table 3.15 The relation $s_2 = \pi_{\{DEPT, TNAME, SALARY\}}(r_3)$

DEPT	TNAME	SALARY
Computing	Robert	2000
Computing	Hanna	1400
Computing	Richard	1000
Maths	Martine	1600
Philosophy	Martine	1600
Maths	Reuven	1500
Linguistics	Dan	1000
Linguistics	Ruth	1100

Table 3.16 The result of the query $s_1 \bowtie s_2$

CNAME	DEPT	TNAME	SALARY
databases	Computing	Robert	2000
programming	Computing	Hanna	1400
programming	Computing	Richard	1000
graph-theory	Maths	Martine	1600
logic	Maths	Reuven	1500
Hebrew	Linguistics	Dan	1000

Algorithm 3.1 (JOIN(r_1, r_2))

1. **begin**
2. Result := $\emptyset$;
3. **for each** tuple $t_1 \in r_1$ **do**
4. **for each** tuple $t_2 \in r_2$ **do**
5. **if** $t_1[X] = t_2[X]$ **then**
6. Joined_tuple := a tuple over R such that Joined_tuple[schema(R_1)] = t_1 and Joined_tuple[schema(R_2)] = t_2;
7. Result := Result $\cup$ Joined_tuple;
8. **end if**
9. **end for**
10. **end for**
11. **return** Result;
12. **end.**

We note that if schema(R_1) = schema(R_2), then the natural join operator reduces to the intersection operator, i.e.

$$r_1 \bowtie r_2 = r_1 \cap r_2.$$

Table 3.17 The result of the query $\pi_{\{CNAME,DEPT\}}(s_1) \bowtie \pi_{\{DEPT,SNAME\}}(r_1)$

CNAME	DEPT	SNAME
databases	Computing	Iris
databases	Computing	Hillary
databases	Computing	David
programming	Computing	Iris
programming	Computing	Hillary
programming	Computing	David
algorithms	Computing	Iris
algorithms	Computing	Hillary
algorithms	Computing	David
logic	Maths	Reuven
logic	Maths	Naomi
graph-theory	Maths	Reuven
graph-theory	Maths	Naomi
Hebrew	Linguistics	Hanna
Hebrew	Linguistics	Dan

Furthermore, if schema(R_1) $\cap$ schema(R_2) = $\emptyset$, then the natural join operator reduces to the *Cartesian product* operator, denoted by $\times$. Therefore, in this case,

$$r_1 \bowtie r_2 = r_1 \times r_2.$$

Informally, the Cartesian product of r_1 over R_1 and r_2 over R_2, with schema(R_1) $\cap$ schema(R_2) = $\emptyset$, is the result of concatenating every tuple of r_1 with every tuple of r_2.

The renaming operator allows us to change the name of an attribute in a schema of a relation. Renaming is useful when we want to take the union, difference or intersection of relations over different schemas, and when we want to take the natural join of two relations over a set of attributes other than the set of common ones.

Definition 3.17 (Renaming) Let r be a relation over relation schema R, A be an attribute of schema(R) and B be an attribute in $\mathcal{U}$, which is not in schema(R). The renaming, ρ, of A to B in r, is a relation over relation schema S, where schema(S) = (schema(R) $-$ {A}) $\cup$ {B}, defined by

$$\rho_{A \to B}(r) = \{t \mid \exists u \in r \text{ such that } t[\text{schema(R)}-\{B\}] = u[\text{schema(R)}-\{A\}] \text{ and } t[B] = u[A]\}.$$

■

The query, "Rename SNAME to STUDENT_NAME, ADDRESS to STUDENT_ADDRESS and DEPT to DEPARTMENT in s_1", can be expressed as the renaming

$$\rho_{SNAME \to STUDENT_NAME}(\rho_{ADDRESS \to STUDENT_ADDRESS}(\rho_{DEPT \to DEPARTMENT}(s_1))),$$

where s_1 is the relation shown in Table 3.4. The result of this query is shown in Table 3.18. We observe that the only effect of renaming is to change attribute names.

We next define the division operator, which captures the semantics of universal quantification (i.e. for all). Informally, the division of two relations, r over a schema having attributes XY and s over a schema having attributes Y, is the set of X-values, say t[X], of tuples $t \in r$ such that for all tuples $u \in s$, u is included in the set of Y-values of tuples t in r having X-value t[X].

Table 3.18 The result of the query $\rho_{SNAME \rightarrow STUDENT_NAME}(\rho_{ADDRESS \rightarrow STUDENT_ADDRESS}(\rho_{DEPT \rightarrow DEPARTMENT}(s_1)))$

STUDENT_NAME	STUDENT_ADDRESS	DEPARTMENT
Iris	Malet St	Computing
Reuven	Harold Rd	Maths
Hanna	Harold Rd	Linguistics
Brian	Alexandra Rd	Sociology

Definition 3.18 (Division) Let r be a relation over relation schema R, with schema(R) = XY, and s be a relation over relation schema S, with schema(S) = Y. The division, $\div$, of r by s, is a relation over relation schema R_1, where schema(R_1) = X, defined by

$$r \div s = \{t[X] \mid t \in r \text{ and } s \subseteq \pi_Y(\sigma_F(r)), \text{ where } X = \{A_1, A_2, \ldots, A_q\} \text{ and } F \text{ is the formula } A_1 = t[A_1] \wedge A_2 = t[A_2] \wedge \ldots \wedge A_q = t[A_q]\}. \quad \blacksquare$$

Let s_4 be the relation over TOPICS, shown in Table 3.19, with schema(TOPICS) = TOPIC representing the research topics of the Computing department and let s_5 be the relation over INTERESTS, shown in Table 3.20, with schema(INTERESTS) = {LECTURER, TOPIC} representing the particular topics academic staff are interested in. The query, "Retrieve the lecturers who are interested in all the topics of the Computing department", can be expressed as the division $s_5 \div s_4$. The result of this query is shown in Table 3.21.

Table 3.19 The relation s_4 over TOPICS

TOPIC
databases
software-engineering
distributed-computing

Table 3.20 The relation s_5 over INTERESTS

LECTURER	TOPIC
Jack	databases
Jack	software-engineering
Jack	distributed-computing
Jeffrey	databases
Jeffrey	distributed-computing
Jeffrey	automata-theory
John	expert-systems
John	software-engineering
Jill	databases
Jill	software-engineering
Jill	distributed-computing
Jill	algorithms

Table 3.21 The result of the query $s_5 \div s_4$

LECTURER
Jack
Jill

The following proposition shows that the division operator can be expressed by the relational algebra operators, projection, difference and Cartesian product.

Proposition 3.2 Let r be a relation over relation schema R, with schema(R) = XY, and s be a relation over relation schema S, with schema(S) = Y. Then

$$r \div s = \pi_X(r) - \pi_X((\pi_X(r) \times s) - r). \qquad \square$$

A relational algebra expression is an expression resulting from composing a finite number of relational algebra operators together, where the operands of the expression are relation schemas.

Definition 3.19 (Relational algebra expressions) A *relational algebra expression* (or alternatively a relational algebra query, or just simply a query whenever no ambiguity arises) is a well-formed expression composed of a finite number of relational algebra operators whose operands are relation schemas which can be treated as input variables to the query. A query Q having as operands the relation schemas $R_1, R_2, \ldots, R_n$ is denoted by $Q(R_1, R_2, \ldots, R_n)$ or simply by Q if $R_1, R_2, \ldots, R_n$ are understood from context. ■

Definition 3.20 (An answer to a query) An *answer to a query* $Q(R_1, R_2, \ldots, R_n)$ is obtained by replacing every occurrence of R_i in Q by a relation r_i over R_i and computing the result by invoking the algebra operators present in Q; such an answer to Q will be denoted by $Q(r_1, r_2, \ldots, r_n)$ or simply by $Q(d)$ if d is a database over **R** and for all $i \in \{1, 2, \ldots, n\}$, $R_i \in$ **R** and $r_i \in d$. ■

We will assume that parentheses are present in Q to indicate the priority of evaluation of subexpressions of Q in order to avoid ambiguity when computing $Q(r_1, r_2, \ldots, r_n)$. At times, when no ambiguity arises, we will also refer to an answer to a query simply as a query.

The relational algebra operators defined above are considered to be the core operators of any relational query language. Therefore, in the following we will refer to this set of operators (or any minimal subset which is of the same expressiveness) as *the relational algebra*. For example, it is not hard to show that union, difference, projection, selection and Cartesian product are such a minimal subset of the relational algebra [Bec78].

The set of queries, which are expressible in the relational algebra, is considered to be the minimal set of queries that any query language for the relational model should possess. Thus the relational algebra provides a yardstick for measuring the expressive power of a query language for the relational model independently of any implementation.

Definition 3.21 (Relational completeness of a query language) A query language is said to be *relationally complete* if it is at least as expressive as the relational algebra. ■

It is interesting to investigate the independence of the operators comprising the relational algebra. For instance, in Proposition 3.2 we have shown that division is not independent, since it can be expressed with projection, difference and Cartesian product. Furthermore, we have also shown that intersection can be expressed with difference or with join, Cartesian

product can be expressed with join and in Theorem 3.1 we have shown that simple selection together with difference, union and intersection can express any selection formula. Another independence result is that join can be expressed with selection, renaming, Cartesian product and projection. It can be shown that projection, union and difference are independent operators of the relational algebra. Projection is the only operator that removes columns from a relation and union is the only operator that adds rows to a relation. It may seem that difference can be expressed with selection formulae that allow negation but this is not the case as evidenced by the following argument.

An operator, τ, from relations to relations is *monotonic*, if whenever r_1 and r_2 are relations over R, with $r_1 \subseteq r_2$, it is also the case that $\tau(r_1) \subseteq \tau(r_2)$; otherwise τ is *nonmonotonic*. We leave it to the reader to verify that selection is a monotonic operator. Now, let r be a relation over R, with schema(R) = $\{A_1, A_2, \ldots, A_m\}$. The *complement* of r, denoted by $\bar{r}$, is given by

$$\bar{r} = (\pi_{A_1}(r) \times \pi_{A_2}(r) \times \cdots \times \pi_{A_m}(r)) - r.$$

We leave it to the reader to show that, in general, the complement operator is nonmonotonic. We therefore conclude that difference cannot be expressed with selection.

Although the relational algebra provides minimal relational capability, in practice, there arises a need for extending the relational algebra in order to enhance its expressive power. We will now introduce two extensions in the form of two additional algebraic operators, which we will not consider as an integral part of the relational algebra.

Table 3.22 A family relation

PARENT	CHILD
p1	p3
p1	p4
p2	p4
p2	p5
p3	p6
p3	p7
p4	p8
p5	p9
p5	p10
p5	p11
p7	p12
p9	p12

The first extension deals with a new operator, called the *transitive closure* operator, which solves the "parts explosion problem" also known as the "bill of materials problem". Consider the relation, which we call family, over the relation schema FAMILY with schema(FAMILY) = {PARENT, CHILD} describing the descendants of the two parents p1 and p2. The acyclic structure of the family relation shown in Table 3.22 is described pictorially as a family tree shown in Figure 3.1. We note that another common example considers a relation, called *parts*, which has attributes SUPER_PART corresponding to PARENT and SUB_PART corresponding to CHILD. In this case the family tree tells us which parts are immediate subparts of a given part and which subparts are indirect subparts of a given part.

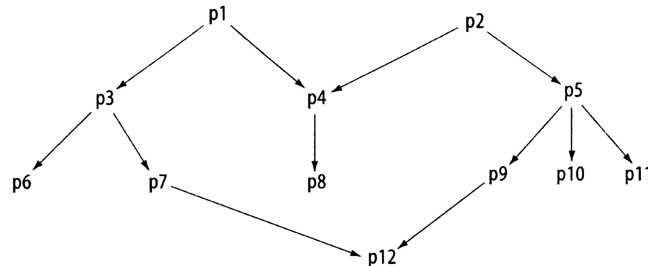

Fig 3.1 The family tree of the family relation in Table 3.22

A query such as "output the parents and their children" is answered easily by just displaying the family relation, and the query "output the parents and their grandchildren" is also easily answered by the query:

$$\pi_{\{GPARENT,GCHILD\}}(\rho_{CHILD \rightarrow Jatt}(\rho_{PARENT \rightarrow GPARENT}(FAMILY)) \bowtie \rho_{PARENT \rightarrow Jatt}(\rho_{CHILD \rightarrow GCHILD}(FAMILY))).$$

Let us denote the above query by $Q^g p$(FAMILY). Using this query we can easily answer queries such as who are the grandchildren of p1 by

$$\pi_{GCHILD}(\sigma_F(Q^g p(FAMILY))),$$

where F is the formula GPARENT = p1, and who are the grandparents of p12 by

$$\pi_{GPARENT}(\sigma_F(Q^g p(FAMILY))),$$

where F is the formula GCHILD = p12.

On the other hand, it can be shown that a query such as "Who are the descendants of a particular parent at all levels?" cannot be expressed as a relational algebra query, in such a way that, for every input relation over FAMILY, we obtain the correct answer. A similar query that, in general, cannot be expressed as a relational algebra query is "Who are the ancestors of a particular child at all levels?" (A formal proof can be found in [AU79] showing that the transitive closure cannot be expressed by using the relational algebra operators we have defined so far.)

In order to understand why the relational algebra is unable to answer such queries, let us first examine the bill of materials for the family relation, shown in Table 3.23. This relation shows the structure of the family tree by indicating the level of a parent-child relationship relative to a given parent at level 1. For example, the tuple <1, p1, p3> indicates that p1 is a parent of p3 at level 1, the tuple <2, p3, p7> indicates that p3 is the parent of p7 at level 2 relative to p1 implying that p1 is a grandparent of p7, and the tuple <3, p7, p12> indicates that p7 is the parent of p12 at level 3 implying that p1 is a great grandparent of p12. Now, informally, the reason that the relational algebra is not powerful enough to express such queries is that we cannot know *a priori* how many levels the bill of materials relation will contain. As we have seen it is easy to answer queries involving parents, grandparents, great grandparents, etc., by joining the family relation as many times as is necessary. However, this technique will not work in general, since if the relational algebra query is to give the correct answer for all

possible relations over FAMILY then it would contain an unbounded number of joins. This leads to a contradiction of the definition of a relational algebra query, which states that a query must be composed of a finite number of relational algebra operators.

Table 3.23 The bill of materials for the family relation

LEVEL	PARENT	CHILD
1	p1	p3
2	p3	p6
2	p3	p7
3	p7	p12
1	p1	p4
1	p2	p4
2	p4	p8
1	p2	p5
2	p5	p9
3	p9	p12
2	p5	p10
2	p5	p11
1	p3	p6
1	p3	p7
2	p7	p12
1	p4	p8
1	p5	p9
2	p9	p12
1	p5	p10
1	p5	p11
1	p7	p12
1	p9	p12

We observe that the family tree is actually a directed graph (recall the formal definition of a directed graph given in Section 2.1 of Chapter 2). We now define the transitive closure operation on directed graphs.

Definition 3.22 (Transitive closure of a directed graph) The *transitive closure* of a directed graph (N, E) is a directed graph (N, E^+) defined by

1) if $(u, v) \in$ E, then $(u, v) \in E^+$,

2) if $(u, v) \in E^+$ and $(v, w) \in$ E, then $(u, w) \in E^+$, and

3) nothing is in E^+ unless it follows from (1) and (2). ■

The transitive closure of the family tree of Figure 3.1 is shown in Figure 3.2; the new arcs, which were added to the original family tree in order to obtain the transitive closure, are shown as squiggled lines. We are now in a position to define the transitive closure of a relation.

Definition 3.23 (Transitive closure of a relation) Let R be a relation schema with schema(R) = {A, B} such that att(1) = A and att(2) = B, and where DOM(A) = DOM(B). The transitive closure of a relation r over R is the relation r^+, where r^+ is the set of arcs of the transitive

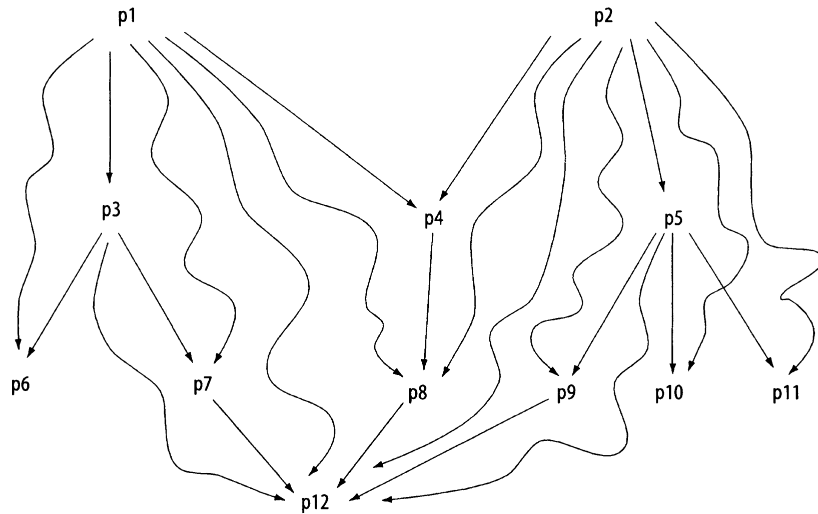

Fig 3.2 The transitive closure of the family tree in Figure 3.1

closure of the directed graph (V, r), with V being the set of values present in the tuples of r, i.e. the active domain of r. ■

The transitive closure *family*$^+$ of the family relation, which was shown in Table 3.22, is shown in Table 3.24. We observe that the transitive closure of a relation expresses exactly the *same information* as the bill of materials of the relation (see the bill of materials for the family relation shown in Table 3.23).

It may be easier to understand the semantics of the transitive closure algorithmically rather than by the above declarative definition, namely Definition 3.22. The pseudo-code of an algorithm, designated TR_CL(r), which given the input relation r over R returns r^+ over R, is presented as the algorithm that follows. In the algorithm we let $Q^{join}(\mathrm{R})$ denote the query

$$\pi_{\{A,B\}}(\rho_{B \rightarrow Jatt}(R) \bowtie \rho_{A \rightarrow Jatt}(R)).$$

Algorithm 3.2 (TR_CL(r))

1. **begin**
2. Result := r;
3. Tmp := $\emptyset$;
4. **while** Tmp $\neq$ Result **do**
5. Tmp := Result;
6. Result := Result $\cup$ Q^{join}(Result);
7. **end while**
8. **return** Result;
9. **end.**

Table 3.24 The transitive closure of the family relation

PARENT	DESCENDANT
p1	p3
p1	p6
p1	p7
p1	p12
p1	p4
p1	p8
p2	p4
p2	p8
p2	p5
p2	p9
p2	p12
p2	p10
p2	p11
p3	p6
p3	p7
p3	p12
p4	p8
p5	p9
p5	p12
p5	p10
p5	p11
p7	p12
p9	p12

The second extension of the relational algebra we deal with is that of allowing aggregate functions in queries. Aggregate functions allow us to answer queries such as:

Q1 How many tutors work in the college?

Q2 How many tutors are employed by the college in each department?

Q3 What is the overall average salary of tutors?

Q4 What is the average salary of tutors per department?

Q5 What is the maximum, respectively minimum, salary of any tutor?

Q6 What is the sum of money that the college spends on its tutors per department?

The above queries cannot be expressed as relational algebra queries, since the relational algebra treats domain values as uninterpreted objects and does not provide for computations that involve iterating over the tuples in a relation. Informally, an aggregate function computes an operation over the A-values of a set of tuples over relation schema R, where A $\in$ schema(R).

Definition 3.24 (Aggregate functions) An aggregate function f_A over R is a Turing-computable function, with A $\in$ schema(R), which given a finite set of tuples over R returns a natural number. ■

The most common aggregate functions are:

1) COUNT, which returns the number of tuples in its input set of tuples (in this case the attribute A is irrelevant);

2) MIN, which returns the minimum A-value of its input set of tuples;

3) MAX, which returns the maximum A-value of its input set of tuples;

4) SUM, which returns the sum of the A-values of its input set of tuples; and

5) AVG, which returns the average A-value of its input set of tuples.

The above list of aggregate functions is by no means exhaustive and various relational DBMSs support additional aggregate functions of a statistical nature. For simplicity, we assume that if an aggregate function is not defined over one of the A-values of a tuple in its input set (which may be empty), then it returns the natural number zero.

Definition 3.25 (Aggregate functions in queries) Let f_A be an aggregate function over R, X $\subseteq$ schema(R) and assume that $A_f \notin$ schema(R) is an attribute in $\mathcal{U}$. The result of applying f_A to a relation r, over R, with the partitioning attribute set X, denoted by $f_A^X(r)$ (or simply f_A if X = $\emptyset$), is a relation over relation schema S, where schema(S) = X $\cup$ $\{A_f\}$, defined by

$$f_A^X(r) = \{t \mid \exists t_1 \in r \text{ such that } t[X] = t_1[X] \text{ and } t[A_f] = f_A(\{t_2 \mid t_2 \in r \text{ and } t_2[X] = t[X]\})\}.$$

■

Consider the relation r_3 over TUTOR with schema(R_3) = {TNAME, DEPT, SALARY, DAY}, shown in Table 3.3.

- The answer to Q1, i.e. COUNT($\pi_{TNAME}(r_3)$), is shown in Table 3.25.
- The answer to Q2, i.e. $COUNT^{DEPT}(\pi_{\{TNAME,DEPT\}}(r_3))$, is shown in Table 3.26.
- The answer to Q3, i.e. $AVG_{SALARY}(\pi_{\{TNAME,SALARY\}}(r_3))$, is shown in Table 3.27.
- The answer to Q4, i.e. $AVG^{DEPT}_{SALARY}(\pi_{\{TNAME,DEPT,SALARY\}}(r_3))$, is shown in Table 3.28.
- The answer to Q5, i.e. $MAX_{SALARY}(r_3) \times MIN_{SALARY}(r_3)$, is shown in Table 3.29.
- The answer to Q6, i.e. $SUM^{DEPT}_{SALARY}(\pi_{\{TNAME,DEPT,SALARY\}}(r_3))$, is shown in Table 3.30.

Table 3.25 The answer to COUNT($\pi_{TNAME}(r_3)$)

COUNT
7

Table 3.26 The answer to $COUNT^{DEPT}(\pi_{\{TNAME,DEPT\}}(r_3))$

DEPT	COUNT
Computing	3
Maths	2
Philosophy	1
Linguistics	2

Table 3.27 The answer to $AVG_{SALARY}(\pi_{\{TNAME,SALARY\}}(r_3))$

AVG_SALARY
1371

Table 3.28 The answer to $AVG^{DEPT}_{SALARY}(\pi_{\{TNAME,DEPT,SALARY\}}(r_3))$

DEPT	AVG_SALARY
Computing	1467
Maths	1550
Philosophy	1600
Linguistics	1050

Table 3.29 The answer to $MAX_{SALARY}(r_3) \times MIN_{SALARY}(r_3)$

MAX_SALARY	MIN_SALARY
2000	1000

Table 3.30 The answer to $SUM^{DEPT}_{SALARY}(\pi_{\{TNAME,DEPT,SALARY\}}(r_3))$

DEPT	SUM_SALARY
Computing	4400
Maths	3100
Philosophy	1600
Linguistics	2100

It may be easier to understand the semantics of the aggregate functions in queries algorithmically rather than by the above declarative definition, namely Definition 3.25. The pseudo-code of an algorithm, designated AGG(f_A, X, r), which given the aggregate function f_A and a relation r over R with A, X $\subseteq$ schema(R), returns $f_A^X(r)$, is presented as the following algorithm. (A *partition* of a set P is a disjoint collection of nonempty subsets of P whose union is P; each subset B_i in a partition P is called a *block*.)

Algorithm 3.3 (AGG(f_A, X, r))

1. **begin**
2. Result := $\emptyset$;
3. P := a partition of r such t_1, t_2 are in the same block in P if and only if $t_1[X] = t_2[X]$;
4. **for each** block B_i in the partition P **do**
5. Agg_tuple := a tuple over S such that Agg_tuple[X] = t[X] with $t \in B_i$ and Agg_tuple[A_f] = $f_A(\{t \mid t \in B_i\})$;
6. Result := Result $\cup$ {Agg_tuple};
7. **end for**
8. **return** Result;
9. **end.**

3.2.2 The Domain Relational Calculus

The domain relational calculus (or simply the relational calculus or the domain calculus or even the calculus) is the declarative counterpart of the relational algebra. It is based on the

first-order predicate calculus (see Subsection 1.9.3). In this logical approach a relational database is considered to be an *interpretation* of a first-order theory. In particular, the domain of the interpretation is a superset of the active database domain and the relations in the database are the extensions of the relation symbols of the database schema. A query in the domain relational calculus is essentially checking whether the database is a *model* of the first-order formula represented in the query. The importance of the relational calculus is that it is more suitable as a basis of user-oriented query languages due to its high-levelness and closeness to natural language. The equivalence between the relational algebra and the domain relational calculus (suitably restricted so that it only yields finite answers to queries) was first shown in [Cod72b] and is discussed in detail in Section 3.3. We proceed to formalise the relational calculus.

Definition 3.26 (Domain calculus expressions) A domain calculus expression (or alternatively a domain calculus query or just simply a query whenever no ambiguity arises) has the form:

$$\{x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \mid F(x_1, x_2, \ldots, x_n)\},$$

where F is a well-formed formula, $A_1, A_2, \ldots, A_n$ are distinct attributes in $\mathcal{U}$ and $x_1, x_2, \ldots, x_n$ are domain variables which occur freely in F, with $n \geq 0$. When $n = 0$ the above query becomes a Boolean query. ■

We will assume that all the relation symbols mentioned in the well-formed formula F stand for relation schemas which are members of a database schema **R**. (The concepts of a well-formed formula and free occurrence of a variable in a well-formed formula are formalised subsequently; see also Subsection 1.9.3.)

Informally, the answer to a domain calculus query

$$\{x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \mid F(x_1, x_2, \ldots, x_n)\}$$

with respect to a database $d = \{r_1, r_2, \ldots, r_m\}$ over the database schema $\mathbf{R} = \{R_1, R_2, \ldots, R_m\}$ is a relation r over relation schema R with schema(R) $= \{A_1, A_2, \ldots, A_n\}$ such that a tuple $<v_1, v_2, \ldots, v_n> \in r$ if and only if

1) for all $i \in \{1, 2, \ldots, n\}$, $v_i \in \text{DOM}(A_i)$; and
2) if for all $i \in \{1, 2, \ldots, n\}$, we substitute v_i for x_i in F, then $<v_1, v_2, \ldots, v_n>$ satisfies the formula F with respect to the database d. (The exact meaning of satisfaction of a formula will be explained subsequently.)

If there no free variables in F, i.e. $n = 0$, then the answer to the query is either true if $\{<>\}$ is returned or false if $\emptyset$ is returned.

If Q is a domain calculus query, then we denote the *answer* to Q with respect to a database d by Q(d), or simply as Q whenever d is understood from context.

We now give some example queries so that the reader can get a feel of the style of the relational calculus as opposed to the relational algebra.

Let $d = \{r_1, r_2, r_3\}$ denote the example database given in Section 3.1, where r_1 is shown in Table 3.1, r_2 is shown in Table 3.2 and r_3 is shown in Table 3.3.

The query, "Retrieve all the tuples in the relation over STUDENT", can be expressed as the domain calculus query:

$$\{x_1 : \text{SNAME}, x_2 : \text{AGE}, x_3 : \text{ADDRESS}, x_4 : \text{DEPT}, x_5 : \text{DEGREE}, x_6 : \text{YEAR} \mid \text{STUDENT}(x_1, x_2, x_3, x_4, x_5, x_6)\}(d).$$

The query, "Retrieve the departments, degrees and years of students", can be expressed as the domain calculus query:

$$\{x_4 : \text{DEPT}, x_5 : \text{DEGREE}, x_6 : \text{YEAR} \mid \exists x_1 : \text{SNAME}(\exists x_2 : \text{AGE}(\exists x_3 : \text{ADDRESS}(\text{STUDENT}(x_1, x_2, x_3, x_4, x_5, x_6))))\}(d).$$

Note the use of the existential quantifier ($\exists$) in this query, which is the way that the domain calculus simulates the relational algebra projection operation.

The query, "Retrieve the names and ages of students who are either studying in the Linguistics department or whose address is Oxford St", can be expressed as the domain calculus query:

$$\{x_1 : \text{SNAME}, x_2 : \text{AGE} \mid \exists x_3 : \text{ADDRESS}(\exists x_4 : \text{DEPT}(\exists x_5 : \text{DEGREE}(\exists x_6 : \text{YEAR} (\text{STUDENT}(x_1, x_2, x_3, x_4, x_5, x_6) \wedge (x_4 = \text{`Linguistics'} \vee x_3 = \text{`Oxford St'})))))\}(d).$$

This query is an example of how the calculus simulates the relational algebra selection operation.

In general, there are many ways of posing the same query. An alternative formulation of the above query is:

$$\{x_1 : \text{SNAME}, x_2 : \text{AGE} \mid (\exists x_3 : \text{ADDRESS}(\exists x_5 : \text{DEGREE}(\exists x_6 : \text{YEAR}(\text{STUDENT}(x_1, x_2, x_3, \text{Linguistics}, x_5, x_6))))) \vee (\exists x_4 : \text{DEPT}(\exists x_5 : \text{DEGREE}(\exists x_6 : \text{YEAR}(\text{STUDENT}(x_1, x_2, \text{Oxford St}, x_4, x_5, x_6)))))\}(d).$$

The query, "Retrieve the names, degrees and departments of students who are not studying in the Computing department and are also not in their second year", can be expressed as the domain calculus query:

$$\{x_1 : \text{SNAME}, x_5 : \text{DEGREE}, x_4 : \text{DEPT} \mid \exists x_3 : \text{ADDRESS}(\exists x_2 : \text{AGE}(\exists x_6 : \text{YEAR} (\text{STUDENT}(x_1, x_2, x_3, x_4, x_5, x_6) \wedge (x_4 \neq \text{`Computing'} \wedge x_6 \neq \text{`second'}))))\}(d).$$

This query is another example of how the calculus simulates the relational algebra selection operation.

For the following query let s_3 be the relation over FST_SND from Subsection 3.2.1, which is shown in Table 3.12. The query, "Retrieve the names of students who did the same number of courses in their first and second years", can be expressed as the domain calculus query:

$$\{x_1 : \text{SNAME} \mid \exists x_2 : \text{FIRST}(\exists x_3 : \text{SECOND} (\text{FST_SND}(x_1, x_2, x_3) \wedge x_2 = x_3))\}(\{s_3\}).$$

This query is an example of how the calculus simulates the relational algebra restriction operation, which is a special case of selection.

Let SHORT_STUD1 and SHORT_STUD2 be relation schemas with schema(SHORT_STUD1) = schema(SHORT_STUD2) = {SNAME, ADDRESS, DEPT} and let s_1 over SHORT_STUD1 and s_2 over SHORT_STUD2 be the relations from Subsection 3.2.1, shown in Tables 3.4 and 3.5, respectively. The query, "Retrieve the students who either have a computing account or are receiving a grant", can be expressed as the domain calculus query:

$$\{x_1 : \text{SNAME}, x_2 : \text{ADDRESS}, x_3 : \text{DEPT} \mid \\ \text{SHORT_STUD1}(x_1, x_2, x_3) \vee \text{SHORT_STUD2}(x_1, x_2, x_3)\}(\{s_1, s_2\}).$$

This query is an example of how the calculus simulates the relational algebra union operation.

The query, "Retrieve the students who have a computing account but do not receive a grant", can be expressed as the domain calculus query:

$$\{x_1 : \text{SNAME}, x_2 : \text{ADDRESS}, x_3 : \text{DEPT} \mid \\ \text{SHORT_STUD1}(x_1, x_2, x_3) \wedge (\neg \text{SHORT_STUD2}(x_1, x_2, x_3))\}(\{s_1, s_2\}).$$

This query is an example of how the calculus simulates the relational algebra difference operation.

The query, "Retrieve the students who have a computing account and are receiving a grant", can be expressed as the domain calculus query:

$$\{x_1 : \text{SNAME}, x_2 : \text{ADDRESS}, x_3 : \text{DEPT} \mid \\ \text{SHORT_STUD1}(x_1, x_2, x_3) \wedge \text{SHORT_STUD2}(x_1, x_2, x_3)\}(\{s_1, s_2\}).$$

This query is an example of how the calculus simulates the relational algebra intersection operation.

The query, "Retrieve the names of courses and the tutoring days", can be expressed as the domain calculus query:

$$\{x_1 : \text{CNAME}, x_2 : \text{DAY} \mid \\ \exists x_3 : \text{DEPT}(\exists x_4 : \text{TNAME}(\exists x_5 : \text{TEXT}(\exists x_6 : \text{SALARY} \\ \text{COURSE}(x_3, x_1, x_4, x_5) \wedge \text{TUTOR}(x_4, x_3, x_6, x_2))))\}(d).$$

This query is an example of how the calculus simulates the relational algebra natural join operation.

An alternative formulation of the above query is:

$$\{x_1 : \text{CNAME}, x_2 : \text{DAY} \mid \\ \exists x_3^1 : \text{DEPT}(\exists x_3^2 : \text{DEPT}(\exists x_4^1 : \text{TNAME}(\exists x_4^2 : \text{TNAME}(\exists x_5 : \text{TEXT}(\exists x_6 : \text{SALARY} \\ (\text{COURSE}(x_3^1, x_1, x_4^1, x_5) \wedge \text{TUTOR}(x_4^2, x_3^2, x_6, x_2) \wedge x_3^1 = x_3^2 \wedge x_4^1 = x_4^2)))))))\}(d).$$

Assume that s_1 over SHORT_STUD is the relation of Subsection 3.2.1, shown in Table 3.4 and recall that r_2 is a relation over COURSE, shown in Table 3.2. The query, "Retrieve the

courses that students can do in the department they are studying in", can be expressed as the domain calculus query:

$$\{x_1 : \text{CNAME}, x_2 : \text{DEPT}, x_3 : \text{SNAME} \mid \exists x_4 : \text{ADDRESS}(\exists x_5 : \text{TNAME}(\exists x_6 : \text{TEXT} \\ (\text{SHORT_STUD}(x_3, x_4, x_2) \wedge \text{COURSE}(x_2, x_1, x_5, x_6))))\}(\{s_1, r_2\}).$$

This query is another example of how the calculus simulates the relational algebra natural join operation.

Let s_4 be the relation over TOPICS, shown in Table 3.19, and let s_5 be the relation over INTERESTS, shown in Table 3.20. The query, "Retrieve the lecturers who are interested in all the topics of the Computing department", can be expressed as the domain calculus query:

$$\{x_1 : \text{LECTURER} \mid \forall x_2 : \text{TOPIC}(\exists x_3 : \text{TOPIC} \\ (\text{TOPICS}(x_2) \wedge \text{INTERESTS}(x_1, x_3) \wedge x_2 = x_3))\}(\{s_4, s_5\}).$$

This query is an example of how the calculus simulates the relational algebra division operation.

We now formally define the components - which are the symbols allowed in formulae and well-formed formulae built from atomic formulae by using the logical connectives - of domain calculus expressions.

The following symbols are allowed to appear in formulae:

- Constant values (or simply constants), $v, v_1, v_2, \ldots$, which are elements of the set $\mathcal{D}$.
- Domain variables (or simply variables), $x, x_1, x_2, \ldots$, which are members of a countably infinite set of variables $\mathcal{V}$ disjoint from $\mathcal{D}$.
- Relation symbols, R, R_1, R_2, ..., which are drawn from a countably infinite set of symbols disjoint from $\mathcal{V}$ and $\mathcal{D}$; each relation symbol corresponds to the relation schema associated with that symbol.
- The equality operator, =.
- The quantifiers and logical connectives, $\exists$ (there exists), $\forall$ (for all), $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implication) and $\neg$ (not).
- Delimiters, () (parentheses), and , (comma).

As in the relational algebra selection formulae we have only included equality (=) as a comparison operator; in general, however, we can expect also to have at least $\leq$ (less than or equal to) and $<$ (less than) available.

Atomic formulae are defined as follows:

1) $R(y_1, y_2, \ldots, y_n)$, where R is a relation symbol with type(R) = n and for all $i \in \{1, 2, \ldots, n\}$, y_i is either a constant or a variable.
2) $x = y$, where x is a variable and y is either a variable or a constant.

Well-formed formulae (or simply formulae) are now defined recursively as follows:

1) An atomic formula is a formula.
2) If F is a formula, then so are $\neg F$ and (F).
3) If F_1 and F_2 are formulae, then so are $F_1 \wedge F_2$, $F_1 \vee F_2$ and $F_1 \Rightarrow F_2$.
4) If F is a formula then $\exists x : \mathrm{A}(F)$ and $\forall x : \mathrm{A}(F)$ are formulae, where x is a variable and A is an attribute.
5) No other formulae are well-formed formulae.

A *subformula* of a formula F is a substring of F that is also a formula. We omit parentheses in formulae if no ambiguity arises as to the meaning of a formula. In addition, we write $x \neq y$ as an abbreviation for $\neg(x = y)$.

From now on we will assume that all the relation schemas corresponding to the relation symbols that are mentioned in F are included in a database schema **R**.

The *free* occurrences of variables in a formula are defined as follows:

1) All the variables occurring in an atomic formula are free.
2) The free variables occurring in $\neg F$ and F are the same as the free variables occurring in the formula F.
3) The free variables occurring in $F_1 \wedge F_2$, $F_1 \vee F_2$ and $F_1 \Rightarrow F_2$ are the free variables occurring in the formula F_1 together with the free variables occurring in the formula F_2.
4) The free variables occurring in $\exists x : \mathrm{A}(F)$ and $\forall x : \mathrm{A}(F)$ are the free variables occurring in the formula F except for occurrences of x in F.

We write $F(x_1, x_2, \dots, x_n)$ for a formula F to indicate that $x_1, x_2, \dots, x_n$ are all the free variables occurring in F.

Definition 3.27 (Satisfaction of a formula by a tuple) Let $d = \{r_1, r_2, \dots, r_m\}$ be a database over the database schema $\mathbf{R} = \{R_1, R_2, \dots, R_m\}$ and consider the query

$$\{x_1 : A_1, x_2 : A_2, \dots, x_n : A_n \mid F(x_1, x_2, \dots, x_n)\}.$$

A tuple $<v_1, v_2, \dots, v_n>$ *satisfies* the formula F with respect to d, if for all $i \in \{1, 2, \dots, n\}$, $v_i \in \mathrm{DOM}(A_i)$, and one of the following conditions is satisfied:

1) If F is the atomic formula $\mathrm{R}(y_1, y_2, \dots, y_k)$, then $\mathrm{R} \in \mathbf{R}$ and the tuple t, resulting from substituting v_i for each variable $y_i \in \{y_1, y_2, \dots, y_k\}$, satisfies $t \in r$, where r is the relation over R in d.
2) If F is the atomic formula $x_i = y_j$, then $v_i = v_j$ is satisfied, where v_i is substituted for x_i, and either y_j is a variable and v_j is substituted for y_j or y_j is a constant and $v_j = y_j$.
3) If F is the formula (G), then $<v_1, v_2, \dots, v_n>$ satisfies the formula F if $<v_1, v_2, \dots, v_n>$ satisfies G.

4) If F takes one of the forms: $\neg F$, $F_1 \wedge F_2$, $F_1 \vee F_2$ or $F_1 \Rightarrow F_2$, then $<v_1, v_2, \ldots, v_n>$ satisfies F is defined according to the semantics of the corresponding logical connectives. As an example, $<v_1, v_2, \ldots, v_n>$ satisfies $F_1 \Rightarrow F_2$ if either $<v_1, v_2, \ldots, v_n>$ does not satisfy F_1 or $<v_1, v_2, \ldots, v_n>$ satisfies F_2. (See Definition 3.14 of logical implication for the semantics of the rest of the connectives.)

5) If F is the formula $\exists x_i : \mathrm{A}\ (\mathrm{G}(x_1, x_2, \ldots, x_i, \ldots, x_n))$, then $<v_1, v_2, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n>$ satisfies F if there exists a constant $v_i \in \mathrm{DOM(A)}$ such that when v_i is substituted for x_i, $<v_1, v_2, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n>$ satisfies G.

6) If F is the formula $\forall x_i : \mathrm{A}\ (\mathrm{G}(x_1, x_2, \ldots, x_i, \ldots, x_n))$, then $<v_1, v_2, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n>$ satisfies F if for all constants $v_i \in \mathrm{DOM(A)}$, when v_i is substituted for x_i, $<v_1, v_2, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n>$ satisfies G. ■

Informally, an answer to a query

$$\{x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \mid F(x_1, x_2, \ldots, x_n)\}$$

with respect to a database d is the set of all tuples satisfying F.

Definition 3.28 (An answer to a domain calculus query) An *answer to a query*

$$\{x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \mid F(x_1, x_2, \ldots, x_n)\}$$

with respect to a database d over **R**, denoted by

$$\{x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \mid F(x_1, x_2, \ldots, x_n)\}(d),$$

is a relation r over relation schema R, with schema(R) $= \{A_1, A_2, \ldots, A_n\}$, defined by

$$r = \{t \mid t \text{ satisfies } F\}.$$ ■

At times we will also refer to an answer to a query simply as a query when no ambiguity arises.

We do not deal with the extension of the domain calculus to incorporate the transitive closure operator or aggregate functions. We mention that an extension of the relational calculus which deals with aggregate functions was given in [Klu82]. On the other hand, we will see that the transitive closure operator can be expressed naturally in Datalog, which is presented in the next section.

We now briefly introduce SQL (Structured Query Language) [AC75, Cha80], which is a relational database query language based on the domain calculus. SQL is in fact more than just a query language, since it also supports updates, data definition of relation schemas, transaction processing and recovery, security of relations, definition of integrity constraints and definition of views. SQL was developed during the 1970's at IBM as part of the System R project. During the 1980's SQL was standardised by ISO and the current ISO SQL standard is its second version, called SQL2 [DD93]. Currently most relational DBMSs support SQL and there is a growing demand from users that these systems support the standard.

In the following we will only cover a small subset of the data manipulation part of SQL. A simple SQL query is a statement, called a *SELECT* statement, having the form:

SELECT $A_1, A_2, \ldots, A_q$
FROM $R_1, R_2, \ldots, R_k$
WHERE F

In the above SELECT statement, $R_1, R_2, \ldots, R_k$ are relation schemas, $A_1, A_2, \ldots, A_q$ are attributes in those relation schemas and F is a selection formula over a relation schema whose attributes comprise the union of the attributes in each schema(R_j), $j \in \{1, 2, \ldots, k\}$.

The semantics of the above SQL query can be best explained in terms of the following relational algebra query:

$$\pi_{\{A_1, A_2, \ldots, A_q\}}(\sigma_F(R_1 \times R_2 \times \cdots \times R_k)).$$

That is, in order to answer a simple SQL query we take the Cartesian product of the relations r_j over R_j in the database which we are querying, then select the tuples that logically imply F and finally project the result onto attributes specified in the SELECT clause. If the WHERE clause is omitted then all the tuples in the Cartesian product are projected onto the specified attributes.

The SQL query

SELECT DEPT, DEGREE, YEAR
FROM STUDENT

is equivalent to the relational algebra query, $\pi_{\{\text{DEPT}, \text{DEGREE}, \text{YEAR}\}}(\text{STUDENT})$.

The SQL query

SELECT *
FROM STUDENT
WHERE DEPT = 'Linguistics' OR ADDRESS = 'Oxford St'

is equivalent to the relational algebra query $\sigma_{F_1}(\text{STUDENT})$, where F_1 is the formula: DEPT = 'Linguistics' $\vee$ ADDRESS = 'Oxford St'. Note that "*" is used to denote the set of all attributes in schema(STUDENT).

The SQL query

SELECT *
FROM STUDENT
WHERE (NOT (DEPT = 'Computing')) AND (NOT (YEAR = 'second'))

is equivalent to the relational algebra query $\sigma_{F_2}(\text{STUDENT})$, where F_2 is the formula: DEPT $\neq$ 'Computing' $\wedge$ YEAR $\neq$ 'second'.

The SQL query

```
SELECT SNAME, STUDENT.DEPT, CNAME
FROM STUDENT, COURSE
WHERE STUDENT.DEPT = COURSE.DEPT
```

is equivalent to the relational algebra query $\pi_{\{SNAME, DEPT, CNAME\}}(STUDENT \bowtie COURSE)$. Note that whenever an attribute appears in two or more schemas we use the dot notation R.A to indicate that we are referring to the attribute A in schema(R).

The SQL query

```
SELECT DEPT, MAX(SALARY)
FROM TUTOR
GROUP BY DEPT
```

is equivalent to the relational algebra query $MAX^{DEPT}_{SALARY}(TUTOR)$. Note that the GROUP BY clause has the effect of partitioning an input relation to the query, over the schema TUTOR, according to the attribute DEPT.

The final example shows an SQL query in which the formula in the WHERE clause of the query is an SQL query itself. Such a query in the WHERE clause is called a *subquery*. The syntax of a subquery, which is nested within the WHERE clause of an SQL query, is the same as a general SQL query and thus multiple subqueries are allowed.

The SQL query

```
SELECT TNAME
FROM TUTOR
WHERE EXISTS
  SELECT *
  FROM COURSE
  WHERE TNAME.TUTOR = TNAME.COURSE
```

is equivalent to the relational algebra query $\pi_{TNAME}(TUTOR \bowtie COURSE)$. Note that the subquery is connected to the main query by using the keyword **EXISTS**; informally, this leads to the selection of only those tuples such that the result of applying the subquery is a nonempty relation.

3.2.3 Datalog

Datalog is an abbreviation for *Data Logic*. As mentioned in Subsection 1.7.5 of Chapter 1 Datalog is a rule-based declarative query language. The syntax of Datalog is essentially a subset of the syntax of Prolog [MW88a, SS94]. In Datalog function symbols in predicates are not allowed and in its purest form there are no extra-logical predicates that operate by "side-effect" such as input and output predicates and, in addition, there are no procedural predicates such as the infamous *cut*, which, in general, cannot be interpreted declaratively. The semantics of Datalog are purely logical as opposed to the semantics of Prolog which

are procedural. Thus, for example, in Prolog the order of the rules in a program and the order of the literals in the body of rules can have an effect on the semantics of a program. Furthermore, the order of facts in a Prolog database can also have an effect on the semantics of a program. In Datalog the order of rules in a program and the order of literals in the body of rules have no effect whatsoever on the semantics of the program. Another important difference between Prolog and Datalog is that Prolog processes one fact at a time while Datalog processes sets of facts at a time. The importance of Datalog is that it adds deductive or inference capabilities to the relational calculus, thus transforming a relational database into a logical database.

We now formally define the syntax of Datalog programs.

The atomic formulae of Datalog are the same as the atomic formulae of domain calculus expressions. As with domain calculus expressions, we write $x \neq y$ as an abbreviation for $\neg(x = y)$. We will distinguish between the following types of atomic formula:

- An atomic formula of the form $R(y_1, y_2, \ldots, y_k)$ is called a *predicate formula* (or simply a predicate); recall that the relation symbol R of the atomic formula corresponds to the relation schema having that symbol (when no ambiguity arises we use the terms relation symbol, predicate and relation schema interchangeably).
- An atomic formula of the form $x = y$ is called an *equality formula* (or simply an equality).
- A predicate formula of the form $R(v_1, v_2, \ldots, v_k)$, where the v_i are constants in DOM(att(i)) for each $i \in \{1, 2, \ldots, k\}$, is called a *ground atomic formula* over R.

A *literal* is either an atomic formula, say L, or the negation of L, namely $\neg L$; L is called a *positive literal* and $\neg L$ is called a *negative literal*. A literal which is a ground atomic formula or the negation of a ground atomic formula is called a *ground literal*.

A *clause* (or alternatively a *rule*) is an expression of the form:

$$L :- L_1, L_2, \ldots, L_n.$$

In a rule such as above $n \geq 0$ is a natural number, for all $i \in \{1, 2, \ldots, n\}$, L_i is a literal and L is a predicate. The sequence of literals, $L_1, L_2, \ldots, L_n$, is called the *body* of the clause and L is called the *head* of the clause. If $n = 0$ then we abbreviate $L :-$ simply to L. In the special case where $n = 0$ and L is a ground atomic formula over R, we call L a *fact* over R (or simply a fact if R is understood from context). A clause which is not a fact is called a *nontrivial rule* (or when there is no confusion, simply a rule). A *Datalog program* P (or simply a program P) is a finite set $C_1, C_2, \ldots, C_m$ of clauses. (When it is convenient then, without any loss of generality, we view P as a sequence of clauses.)

We observe that a relation r over a schema R induces a set of facts all having the relation symbol R. For example, the first two tuples in the relation r_1 over STUDENT, shown in Table 3.1, induce the following two facts:

STUDENT(Iris, 21, Malet St, Computing, BSC, first).
STUDENT(Reuven, 32, Harold Rd, Maths, BSC, second).

Prior to the ensuing definition, we recall the definition of an acyclic and cyclic directed graph from Section 2.1 of Chapter 2.

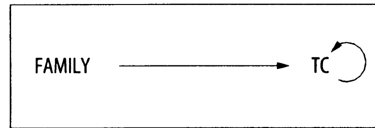

Fig 3.3 The dependency graph of TC

Definition 3.29 (Recursive and nonrecursive Datalog programs) The *dependency graph* of a Datalog program P is a digraph (N, E), where the set of nodes N is the set of relation symbols that appear in the literals of P, and there is an arc from R_1 to R_2 in E if there is a rule in P whose body contains either the positive literal $R_1(y_1, y_2, \ldots, y_k)$ or the negative literal $\neg R_1(y_1, y_2, \ldots, y_k)$ and whose head is the literal $R_2(z_1, z_2, \ldots, z_q)$.

A Datalog program is said to be *nonrecursive* if its dependency graph is acyclic, otherwise if its dependency graph is cyclic, it is said to be a *recursive* Datalog program. ■

An example of a recursive Datalog program, say TC, is:

$$\text{TC}(x_1, x_2) :- \text{FAMILY}(x_1, x_2).$$
$$\text{TC}(x_1, x_3) :- \text{FAMILY}(x_1, x_2), \text{TC}(x_2, x_3).$$

The cyclic dependency graph of the above simple Datalog program is shown in Figure 3.3.

After we define the meaning of a Datalog program it will be evident that the Datalog program given above computes the transitive closure of FAMILY, assuming that the program contains some facts over FAMILY (see Definition 3.34 below).

For the rest of this section we will assume that Datalog programs are nonrecursive and refer to them simply as Datalog programs. Although we restrict Datalog programs to be nonrecursive, Definition 3.34, giving the semantics of Datalog programs, does obtain for recursive Datalog programs also. (The definition for recursive Datalog programs will be needed in Chapter 9 on deductive databases).

We now give some examples of Datalog programs so that the reader can get a feel of the programming style of Datalog. In all of the programs we have assumed a relation symbol, RESULT, whose set of facts will contain the result of the query when the Datalog program is evaluated.

The query, "Retrieve all the tuples in the relation over STUDENT", can be expressed as the result of the Datalog program:

$$\text{RESULT}(x_1, x_2, x_3, x_4, x_5, x_6) :- \text{STUDENT}(x_1, x_2, x_3, x_4, x_5, x_6).$$

The query, "Retrieve the departments, degrees and years of students", can be expressed as the result of the Datalog program:

$$\text{RESULT}(x_4, x_5, x_6) :- \text{STUDENT}(x_1, x_2, x_3, x_4, x_5, x_6).$$

Note how the absence of variables in the head of a rule simulates the use of the

existential quantifier (∃) in a domain calculus query and thus the relational algebra projection operation.

The query, "Retrieve the names and ages of students who are either studying in the Linguistics department or whose address is Oxford St", can be expressed as the result of the Datalog program:

$$\text{RESULT}(x_1, x_2) :- \text{STUDENT}(x_1, x_2, x_3, x_4, x_5, x_6), x_4 = \text{'Linguistics'}.$$
$$\text{RESULT}(x_1, x_2) :- \text{STUDENT}(x_1, x_2, x_3, x_4, x_5, x_6), x_3 = \text{'Oxford St'}.$$

Note how two rules in a Datalog program simulate the use of disjunction (∨) in a domain calculus query. In addition, each rule simulates a relational algebra selection operation.

The query, "Retrieve the names, degrees and departments of students who are not studying in the Computing department and are also not in their second year", can be expressed as the result of the Datalog program:

$$\text{RESULT}(x_1, x_5, x_4) :- \text{STUDENT}(x_1, x_2, x_3, x_4, x_5, x_6), x_4 \neq \text{'Computing'}, x_6 \neq \text{'second'}.$$

The above program is an example of how Datalog simulates the relational algebra selection operation.

For the following query let s_3 be the relation over FST_SND from Subsection 3.2.1, which is shown in Table 3.12. The query, "Retrieve the names of students who did the same number of courses in their first and second years", can be expressed as the result of the Datalog program:

$$\text{RESULT}(x_1) :- \text{FST_SND}(x_1, x_2, x_3), x_2 = x_3.$$

The above program is an example of how Datalog simulates the relational algebra restriction operation, which is a special case of selection.

Let SHORT_STUD1 and SHORT_STUD2 be relation schemas with schema(SHORT_STUD1) = schema(SHORT_STUD2) = {SNAME, ADDRESS, DEPT} and let s_1 over SHORT_STUD1 and s_2 over SHORT_STUD2 be the relations from Subsection 3.2.1, shown in Tables 3.4 and 3.5, respectively. The query, "Retrieve the names of students who either have a computing account or are receiving a grant", can be expressed as the result of the Datalog program:

$$\text{RESULT}(x_1, x_2, x_3) :- \text{SHORT_STUD1}(x_1, x_2, x_3).$$
$$\text{RESULT}(x_1, x_2, x_3) :- \text{SHORT_STUD2}(x_1, x_2, x_3).$$

The above program is an example of how Datalog simulates the relational algebra union operation.

The query, "Retrieve the names of students who have a computing account but do not receive a grant", can be expressed as the result of the Datalog program:

$$\text{RESULT}(x_1, x_2, x_3) :- \text{SHORT_STUD1}(x_1, x_2, x_3), \neg\text{SHORT_STUD2}(x_1, x_2, x_3).$$

The above program is an example of how Datalog simulates the relational algebra difference operation.

The query, "Retrieve the names of students who have a computing account and are receiving a grant", can be expressed as the result of the Datalog program:

$$\text{RESULT}(x_1, x_2, x_3) :- \text{SHORT_STUD1}(x_1, x_2, x_3), \text{SHORT_STUD2}(x_1, x_2, x_3).$$

The above program is an example of how Datalog simulates the relational algebra intersection operation.

The query, "Retrieve the names of courses and the tutoring days", can be expressed as the result of the Datalog program:

$$\text{RESULT}(x_1, x_2) :- \text{COURSE}(x_3, x_1, x_4, x_5), \text{TUTOR}(x_4, x_3, x_6, x_2).$$

The above program is an example of how Datalog simulates the relational algebra natural join operation.

An alternative formulation of the above query is:

$$\text{RESULT}(x_1, x_2) :- \text{COURSE}(x_3^1, x_1, x_4^1, x_5), \text{TUTOR}(x_4^2, x_3^2, x_6, x_2), x_3^1 = x_3^2, x_4^1 = x_4^2.$$

Assume that s_1 over SHORT_STUD is the relation of Subsection 3.2.1, shown in Table 3.4, and recall that r_2 is a relation over COURSE. The query, "Retrieve the courses that students can do in the department they are studying in", can be expressed as the result of the Datalog program:

$$\text{RESULT}(x_1, x_2, x_3) :- \text{SHORT_STUD}(x_3, x_4, x_2), \text{COURSE}(x_2, x_1, x_5, x_6).$$

The above program is another example of how Datalog simulates the relational algebra natural join operation.

Let s_4 be the relation over TOPICS, shown in Table 3.19, and let s_5 be the relation over INTERESTS, shown in Table 3.20. The query, "Retrieve the lecturers who are interested in all the topics of the Computing department", can be expressed as the result of the Datalog program:

$$\text{RESULT}(x_1) :- \text{INTERESTS}(x_1, x_3), \neg\text{DIFF}(x_1).$$
$$\text{DIFF}(x_1) :- \text{PROD}(x_1, x_2), \neg\text{INTERESTS}(x_1, x_2).$$
$$\text{PROD}(x_1, x_2) :- \text{INTERESTS}(x_1, x_3), \text{TOPICS}(x_2).$$

The above program is an example of how Datalog simulates the relational algebra division operation. The acyclic dependency graph of the above program is shown in Figure 3.4.

A Datalog program makes sense only if the relations that can be derived from executing these programs are finite. The *safety* restriction, defined next, provides a syntactic restriction of programs which enforces the finiteness of derived predicates (or relations).

Definition 3.30 (Safe Datalog program) A variable x occurring in one of the literals in the head or the body of a rule, say C, *occurs positively* in C if and only if either

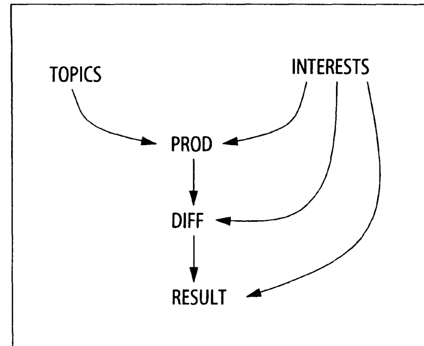

Fig 3.4 The dependency graph of RESULT

1) The variable x appears in a predicate formula $R(y_1, y_2, \ldots, x, \ldots, y_k)$, which is a positive literal in the body of C, or
2) The variable x appears in an equality formula $x = v$, which is a positive literal in the body of C, where v is a constant, or
3) The variable x appears in the equality $x = y$ or $y = x$, which is a positive literal in the body of C, where y is a variable that appears positively in C.

A Datalog rule C is said to be *safe* if all the variables appearing in the literals of C (including the head of C) occur positively in C. A Datalog program P is said to be *safe* if all the rules of P are safe. ■

We observe that in a safe rule all the variables in the head of a rule must appear in one or more literals in its body. Furthermore, all the variables appearing in negative literals in the body of a safe rule must occur positively in one or more atomic formulae in its body.

Example 3.1 The following rules are not safe:

1) $\text{RESULT1}(x) :- \text{COURSE}(x_1, x_2, x_3, x_4)$
2) $\text{RESULT2}(x_1, x_2, x_3, x_4) :- \neg\text{COURSE}(x_1, x_2, x_3, x_4)$
3) $\text{RESULT3}(x_1) :- \text{COURSE}(x_1, x_2, x_3, x_4), x_5 = x_6$

In the first rule the variable x does not appear in the body of the rule. In the second rule the variables x_1, x_2, x_3, x_4 appear only in the negative literal in the body of the rule. In the third case x_5 and x_6 appear only in the equality $x_5 = x_6$. ■

The reader can verify that all the examples of Datalog programs given prior to Example 3.1 are safe.

As we will show in Subsection 3.3.2 the class of nonrecursive safe Datalog programs can express exactly the same set of queries that the relational algebra can express. Moreover, we also show in Subsection 3.3.2 that nonrecursive safe Datalog can be viewed as a restricted

version of the domain calculus, that is, nonrecursive safe Datalog is exactly as expressive as this restricted version of the domain calculus.

We proceed to define the semantics of general Datalog programs, which may be recursive and unsafe.

Definition 3.31 (The schema of Datalog program) The database schema of a Datalog program P, denoted by SCHEMA(P), is a set of relation schemas defined by

SCHEMA(P) = {R | R is a relation symbol that appears in a literal of a rule in P}. ■

Definition 3.32 (Substituting the variables in a clause) Let C be a clause and $\{x_1, x_2, \ldots, x_q\}$ be the variables appearing in the literals of the body of C. A *substitution* θ for C is a set of assignments $\{x_1/v_1, x_2/v_2, \ldots, x_q/v_q\}$, where for all $i \in \{1, 2, \ldots, q\}$, v_i is in the domain of constants, $\mathcal{D}$. We denote by θ(C) the clause resulting from applying the substitution θ to the literals in C, i.e. the result of substituting, for each $i \in \{1, 2, \ldots, q\}$, the constant v_i for the variable x_i in each of the literals in C. ■

We note that all the literals of the clause θ(C) are ground literals.

Definition 3.33 (Truth of a clause with respect to a database) A literal L in the body of a clause C in a Datalog program P is true with respect to a substitution θ for C and a database d over SCHEMA(P) if one of the following conditions is satisfied:

1) $\theta(L)$ is a ground atomic formula of the form $R(v_1, v_2, \ldots, v_k)$ and $<v_1, v_2, \ldots, v_k> \in r$, where $r \in d$ is the relation over R.
2) $\theta(L)$ is an equality, $v = v$, where v is a constant.
3) $\theta(L)$ is a ground literal of the form $\neg R(v_1, v_2, \ldots, v_k)$ and $<v_1, v_2, \ldots, v_k> \notin r$, where $r \in d$ is the relation over R.
4) $\theta(L)$ is a negative literal of the form, $\neg(v_i = v_j)$, where v_i and v_j are distinct constants, i.e. such that $v_i \neq v_j$.

A clause C in a program P is true with respect to a substitution θ for C and a database d over SCHEMA(P) if each of the literals in the body of C is true with respect to θ and d. ■

We observe that the truth of a negative literal with respect to a substitution and a database is consistent with the CWA (closed world assumption) [Rei78], since $\neg R(v_1, v_2, \ldots, v_k)$ is assumed to be true if the tuple $<v_1, v_2, \ldots, v_k>$ is absent from the database. This causes a problem if the Datalog program is unsafe, since infinite relations may be derived due to the fact that the underlying domain is infinite. Thus RESULT2 of Example 3.1 is an infinite relation.

We further note that if the body of the clause is empty, i.e. the clause is a predicate, then C is trivially true with respect to any substitution θ for C and any database d over SCHEMA(P). In particular, if C is a fact then C is trivially true with respect to θ and d.

The meaning of a Datalog program P, denoted by MEANING(P), is informally the database resulting from adding to the initial set of facts recorded in P as many new facts of the form

$\theta(L)$ as possible, where θ is a substitution that makes a rule C in P true and L is the head of C. (A set of facts whose relation symbols are in SCHEMA(P) is naturally associated with a database d over SCHEMA(P), since a fact having the relation symbol R can be viewed as a set of tuples over R.)

Definition 3.34 (The meaning of a Datalog program) The pseudo-code of an algorithm, which realises MEANING(P), is next presented. (The variable Im is called the *immediate consequence* of the current state of MEANING(P).) ■

Algorithm 3.4 (MEANING(P))

1. **begin**
2. Result := $\emptyset$;
3. Tmp := $\{<>\}$;
4. **while** Tmp $\neq$ Result **do**
5. Tmp := Result;
6. Im := $\emptyset$;
7. **for all** clauses C in P and substitutions θ for C such that C is true with respect to θ and Result **do**
8. Im := Im $\cup$ $\{\theta(L)\}$ where L is the head of C;
9. **end for**
10. Result := Result $\cup$ Im;
11. **end while**
12. **return** Result;
13. **end.**

We observe that the current state of Result strictly increases after each iteration of the while loop beginning at line 4, provided the relations in the immediate consequence, Im, of Result are not already included in the respective relations in Result. Thus Im induces an *immediate consequence* operator, say $\mathcal{T}$, such that $\mathcal{T}$(Result) = Result $\cup$ Im. Such an increasing operator is called *inflationary*, and the final output database returned from MEANING(P) is called the *inflationary fixpoint* of the Datalog program P [GS86, KP91]. (We also refer to MEANING(P) as the inflationary meaning of P.)

We will now show how we can optimise Algorithm 3.4.

Let CONST(P) denote the set of all constants appearing in the literals of the clauses in a Datalog program P, and call a substitution $\theta = \{x_1/v_1, x_2/v_2, \ldots, x_q/v_q\}$ for a clause C in P a *safe* substitution if $\{v_1, v_2, \ldots, v_q\} \subseteq$ CONST(P).

The following proposition, which follows immediately from the definition of a clause being true with respect to a substitution and a database, states that when computing MEANING(P) for a safe Datalog program, P, it is sufficient to consider only safe substitutions.

Proposition 3.3 Let d over SCHEMA(P) be the current state of Result at line 10 of the algorithm MEANING(P) after one or more executions of the while loop beginning at line 4 of the algorithm. Then a clause C in a safe Datalog program P is true with respect to a substitution θ for C and d if and only if θ is a safe substitution for C. □

The next proposition states that when considering only safe substitutions, then safe and unsafe Datalog are equivalent, since only finite relations can be generated in both cases.

Proposition 3.4 Assume that only safe substitutions θ for C are considered in line 7 of MEANING(P). Then safe and unsafe Datalog are equivalent in the sense that for every unsafe Datalog program P^u, there exists a safe Datalog program P^s such that MEANING(P^u) = MEANING(P^s).

Proof. Let P^u be an unsafe Datalog program. We can then construct a safe Datalog program, denoted by P^c, having a distinguished unary predicate, CONSTANT, which is the head of all the rules in P^c, and such that the set of facts in MEANING(P^c) over CONSTANT is exactly CONST(P^u). First for every rule R^u in P^u whose head is $R(x_1, x_2, \ldots, x_i, \ldots, x_k)$ such that x_i does not appear in the literals of the body of R^u add CONSTANT(x_i) to the body of R^u. Then for every rule R^u in P^u, and every negative literal in the body of R^u of the form $\neg R(x_1, x_2, \ldots, x_k)$, which causes R^u to be unsafe, add the positive literals, CONSTANT(x_1), CONSTANT(x_2),..., CONSTANT(x_k), to the body of R^u. Moreover, for every rule R^u in P^u, and every equality in the body of R^u of the form $x_i = x_j$, which causes R^u to be unsafe, add the positive literals, CONSTANT(x_i) and CONSTANT(x_j) to the body of R^u. Finally, add the rules in P^c to the modified version of P^u, and denote this program by P^s. We leave it to the reader to verify that P^s is indeed a safe Datalog program and that MEANING(P^s) = MEANING(P^u). □

For the rest of the book *we assume that only safe substitutions are considered at line 7 of* MEANING(P). Therefore, due to the above proposition, we need not distinguish between safe and unsafe Datalog, since under this assumption all Datalog programs can be considered as being *safe*. (At times for clarity we will highlight the fact that a Datalog program is safe.)

Definition 3.35 (The initial database of a Datalog program) The initial (relational) database of a Datalog program P, over SCHEMA(P) = $\{R_1, R_2, \ldots, R_n\}$, denoted by DB(P), is the set of relations $\{r_1, r_2, \ldots, r_n\}$ such that for all $i \in \{1, 2, \ldots, n\}$, r_i is defined by

$$r_i = \{<v_1, v_2, \ldots, v_k> \mid R_i(v_1, v_2, \ldots, v_k) \text{ is a fact over } R_i \text{ that appears in P}\}. \quad \blacksquare$$

We say that a database d_1 over a database schema **R** is *included* in a database d_2 over **R**, written $d_1 \subseteq d_2$, if $\forall\, r_i^1 \in d_1$ over $R_i \in \mathbf{R}$, with $r_i^1 \neq \emptyset$, $\exists\, r_i^2 \in d_2$ over R_i such that $r_i^1 \subseteq r_i^2$.

The following proposition follows immediately from inspecting Algorithm 3.4 noting that if a clause C is actually a fact, say $R(v_1, v_2, \ldots, v_k)$, then C is true with respect to any database and the empty substitution $\theta = \emptyset$.

Proposition 3.5 The initial database of a Datalog program P, DB(P), is included in the meaning of P, MEANING(P). □

Let P be a Datalog program and SCHEMA(P) = $\{R_1, R_2, \ldots, R_n\}$. Now, due to the fact that in this section P is assumed to be nonrecursive, we can order the relation schemas in SCHEMA(P) in such a way that for any $R_i, R_j \in$ SCHEMA(P) if there is a path from R_i to R_j in the dependency graph of P then $i < j$. (We note that such an ordering can be obtained by a topological sort of the dependency graph, as defined in Subsection 1.9.2 of Chapter 1.) Let us assume that the relation schemas in SCHEMA(P) are ordered in this manner.

The pseudo-code of an algorithm, which realises NEW_MEANING(P), taking into account Propositions 3.3, 3.4 and 3.5 and the ordering of the relation schemas in SCHEMA(P), is presented as the algorithm that follows; for the purpose of the algorithm, given a database d over SCHEMA(P), $d \cup R_i(v_1, v_2, \ldots, v_k)$ is the database resulting from inserting $<v_1, v_2, \ldots, v_k>$ into the relation $r_i \in d$ over R_i, where d is a database over SCHEMA(P).

Algorithm 3.5 (NEW_MEANING(P))

1. **begin**
2. Result := DB(P);
3. **for** $i := 1$ **to** n **do**
4. **while** there exists a rule C in P such that R_i is the relation symbol of its head and there exists a safe substitution θ for C such that C is true with respect to θ and Result **do**
5. Result := Result $\cup$ $\theta(L)$ where L is the head of C;
6. **end while**
7. **end for**
8. **return** Result;
9. **end.**

We leave the proof of the following theorem to the reader.

Theorem 3.6 Given a Datalog program P (which is assumed to be nonrecursive and safe), MEANING(P) = NEW_MEANING(P). □

We now show how the meaning of a Datalog program can be used to answer queries.

Definition 3.36 (Datalog query) A *Datalog query* with respect to a Datalog program P is an expression of the form $:-_P L$, where L is a predicate (or simply $:-L$ whenever P is understood from context). ■

Definition 3.37 (The answer to a Datalog query) Let L be the predicate $R(y_1, y_2, \ldots, y_k)$ and $\{x_1, x_2, \ldots, x_q\}$ be the variables appearing in L. Furthermore, let us call a substitution $\theta = \{x_1/v_1, x_2/v_2, \ldots, x_q/v_q\}$ *safe* for L with respect to a Datalog program P if $\{v_1, v_2, \ldots, v_q\} \subseteq$ CONST(P). Then the answer to the Datalog query, $:-_P L$, is the relation r over R, defined by

$$\{\theta(L) \mid \theta \text{ is a safe substitution for } L \text{ with respect to P, and } \theta(L) \in r, \text{ where } r \in \text{MEANING(P) is the relation over R} \in \text{SCHEMA(P)}\}.$$ ■

We note that if R $\notin$ SCHEMA(P), then the answer to $:-L$ is the empty set. The following notation will be useful later on when we discuss the equivalence of the relational algebra, the domain calculus and nonrecursive safe Datalog programs.

Definition 3.38 (Datalog query with respect to a database) Let P be a Datalog program and d be a database over **R** such that **R** $\subseteq$ SCHEMA(P). The Datalog program P with respect to d, denoted by P(d), is the Datalog program resulting from removing all the facts in DB(P) from P and then adding to P all the facts contained in the relations of the database d. That is, DB(P(d)) = d holds, i.e. the facts in P are replaced by those in d to obtain P(d).

Let Q be the Datalog query $:-_P L$. Then the Datalog query Q with respect to P and d, denoted by Q(d), is defined as the Datalog query $:-_{P(d)} L$. ∎

3.2.4 An Update Language for the Relational Model

So far we have only considered query languages for the relational model, which can only be used to retrieve information from a relational database. In this section we consider the dynamic aspects of updating a relational database resulting in its transition from one state to another. An update can take one of three forms, namely an insertion, a deletion or a modification. Insertion of a tuple into a relation results in the addition of this tuple to the relation. The deletion of a set of tuples from a relation with respect to a condition, C, results in the removal of the set of tuples satisfying C from the relation. The modification of a set of tuples with respect to two conditions, C_1 and C_2, results in replacing the set of tuples satisfying C_1 by the set of tuples satisfying C_2. A transaction can now be defined as the sequential composition of one or more updates. The aim of this section is two-fold. Firstly, to formalise the notion of an update and a transaction, and secondly to show that the equivalence of two transactions can be tested in polynomial time in the size of the transactions being tested. The test for equivalence is an essential ingredient in optimising a transaction, which intuitively means replacing the transaction by an equivalent one which requires less operations. Prior to defining updates we formalise the notion of a condition and a set of tuples satisfying a condition.

Definition 3.39 (Condition) A *simple condition* over R is either an expression of form A = a or an expression of the form A $\neq a$, where A ∈ schema(R) and a ∈ DOM(A).

A *condition* C over R is a conjunction $c_1 \wedge c_2 \wedge \ldots \wedge c_q$ of simple conditions c_i, $i \in \{1, 2, \ldots, q\}$, such that C does not contain two distinct simple conditions of the form A = a and A = b or of the form A = a and A $\neq a$, for some A ∈ schema(R).

A *positive condition* over R is a condition of the form

$$A_1 = a_1 \wedge A_2 = a_2 \wedge \ldots \wedge A_q = a_q,$$

where $\{A_1, A_2, \ldots, A_q\} \subseteq$ schema(R) and $a_i \in$ DOM(A_i), for $i \in \{1, 2, \ldots, q\}$.

A *complete condition* over R is a positive condition over R, where $\{A_1, A_2, \ldots, A_q\}$ = schema(R) obtains in the definition of a positive condition. ∎

We observe that disallowing distinct simple conditions of the form A = a and A = b or of the form A = a and A $\neq a$, for some A ∈ schema(R), restricts conditions to be *meaningful* by not having mutually exclusive conditions.

Definition 3.40 (Satisfaction of a condition by a tuple) Let r be a relation over R, let t be a tuple in r and, in addition, let $C = c_1 \wedge c_2 \wedge \ldots \wedge c_q$ be a condition over R. Then t *satisfies* C, written $t \models C$, is defined recursively, as follows:

1) $t \models$ A = a, if t[A] = a is true.

2) $t \models$ A $\neq a$, if t[A] $\neq a$ is true.

3) $t \models C$, if $\forall i \in \{1, 2, \ldots, q\}$, $t \models c_i$. ∎

We note that we could extend conditions in a straightforward way to be general Boolean expressions but we prefer to keep the formalism simple. Furthermore, for simplicity we only formalise updates on single relations but we note that the definitions given below can be extended to databases (containing several relations) in a straightforward manner.

Definition 3.41 (An update) Let r be a relation over relation schema R, with schema(R) = $\{A_1, A_2, \ldots, A_m\}$. An update over R is either an *insertion* over R, a *deletion* over R or a *modification* over R.

An insertion over R is an expression of the form insert(C), where C is a complete condition over R. The *effect* of an insertion insert(C) over R on r is defined by

$$[\text{insert}(C)](r) = r \cup \{t \mid t \models C\}.$$

A deletion over R is an expression of the form delete(C), where C is a condition over R. The *effect* of a deletion delete(C) over R on r is defined by

$$[\text{delete}(C)](r) = r - \{t \mid t \in r \text{ and } t \models C\}.$$

The modification of a tuple t over R with respect to a condition C, denoted by [modify(C)](t), is defined by

$$[\text{modify}(C)](t) = u, \text{ where } u \text{ is a tuple over R such that}$$
$$\forall A_i \in \text{schema(R)}, u[A_i] = a_i \text{ if } (A_i = a_i) \in C, \text{ otherwise } u[A_i] = t[A_i].$$

A modification over R is an expression of the form modify(C_1; C_2), where C_1 and C_2 are conditions over R such that for each A $\in$ schema(R), if A $\neq a$ is in C_2 for some $a \in$ DOM(A), then A $\neq a$ is also in C_1. The *effect* of a modification modify(C_1; C_2) over R on r is defined by

$$[\text{modify}(C_1; C_2)](r) = (r - \{t \mid t \in r \text{ and } t \models C_1\}) \cup \{[\text{modify}(C_2)](t) \mid t \in r \text{ and } t \models C_1\}.$$

■

The definition of a modification can be viewed as a deletion followed by a sequence of insertions whereby each inserted tuple is the result of modifying some deleted tuple. We note that we could restrict condition C_2 above to be positive and the effect of the modification would remain unchanged; however, for the purpose of the normal form introduced in Definition 3.44, we find the above definition convenient.

As a running example for this subsection, suppose that we have a relation schema EMPLOYEE having attributes: ENAME (employee name, abbreviated to EN), DNAME (department name, abbreviated to DN) and SALARY (employee salary, abbreviated to SL). A relation r over EMPLOYEE is shown in Table 3.31.

Table 3.31 The relation r over EMPLOYEE

ENAME	DNAME	SALARY
John	Computing	30K
Jack	Computing	35K
Jake	Biology	30K

Consider the following updates, where ":=" denotes assignment:

1) $r_1 := [\text{insert}(\text{EN} = \text{Jill} \wedge \text{DN} = \text{Maths} \wedge \text{SL} = 25\text{K})](r)$ is shown in Table 3.32.

2) $r_2 := [\text{insert}(\text{EN} = \text{Joe} \wedge \text{DN} = \text{Maths} \wedge \text{SL} = 35\text{K})](r_1)$ is shown in Table 3.33.

3) $r := [\text{delete}(\text{DN} = \text{Maths})](r_2)$ is shown in Table 3.31.

4) $r := [\text{delete}(\text{DN} \neq \text{Computing} \wedge \text{DN} \neq \text{Biology})](r_2)$ is shown in Table 3.31.

5) $r_1 := [\text{delete}(\text{EN} = \text{Joe})](r_2)$ is shown in Table 3.32.

6) $r_3 := [\text{modify}(\text{DN} = \text{Computing}; \text{DN} = \text{Maths})](r)$ is shown in Table 3.34.

7) $r_4 := [\text{modify}(\text{DN} \neq \text{Computing}; \text{DN} = \text{Maths})](r)$ is shown in Table 3.35.

Table 3.32 The relation r_1 over EMPLOYEE

ENAME	DNAME	SALARY
John	Computing	30K
Jack	Computing	35K
Jake	Biology	30K
Jill	Maths	25K

Table 3.33 The relation r_2 over EMPLOYEE

ENAME	DNAME	SALARY
John	Computing	30K
Jack	Computing	35K
Jake	Biology	30K
Jill	Maths	25K
Joe	Maths	35K

Table 3.34 The relation r_3 over EMPLOYEE

ENAME	DNAME	SALARY
John	Maths	30K
Jack	Maths	35K
Jake	Biology	30K

Table 3.35 The relation r_4 over EMPLOYEE

ENAME	DNAME	SALARY
John	Computing	30K
Jack	Computing	35K
Jake	Maths	30K

Informally a transaction is the composition of several updates. In the following an update will be designated by *upd*.

Definition 3.42 (Transaction) A *transaction* T over R is a finite sequence of updates over R. The *effect* of a transaction T = $upd_1, upd_2, \ldots, upd_n$, on a relation r over R, where $n \geq 0$ is a natural number, is defined by

$$[\mathrm{T}](r) = [upd_n](\ldots([upd_2]([upd_1](r)))\ldots). \quad \blacksquare$$

We note that according to Definition 3.42, if $n = 0$ then $[\mathrm{T}](r) = r$. Consider the following transactions on r:

1) T_1 = insert(EN = Jill $\wedge$ DN = Maths $\wedge$ SL = 25K), modify(EN = Jack; DN = Maths).
2) T_2 = insert(EN = Jill $\wedge$ DN = Maths $\wedge$ SL = 25K), insert(EN = Jack $\wedge$ DN = Maths $\wedge$ SL = 35K), delete(EN = Jack $\wedge$ DN = Computing $\wedge$ SL = 35K).
3) T_3 = modify(EN = John; EN = Jill $\wedge$ DN = Maths $\wedge$ SL = 25K), insert(EN = John $\wedge$ DN = Computing $\wedge$ SL = 30K), modify(EN = Jack; DN = Maths).
4) T_4 = delete(EN = Jack), insert(EN = Jack $\wedge$ DN = Maths $\wedge$ SL = 35K), insert(EN = Jill $\wedge$ DN = Maths $\wedge$ SL = 25K).
5) T_5 = modify(DN $\neq$ Biology; DN = Maths), modify(EN = John; DN = Computing), insert(EN = Jill $\wedge$ DN = Maths $\wedge$ SL = 25K).

It can be verified that $[T_1](r) = [T_2](r) = [T_3](r) = [T_4](r) = [T_5](r) = r_5$, where r is shown in Table 3.31 and r_5 is shown in Table 3.36.

Table 3.36 The relation r_5 over EMPLOYEE

ENAME	DNAME	SALARY
John	Computing	30K
Jack	Maths	35K
Jake	Biology	30K
Jill	Maths	25K

The next definition formalises the intuition that two transactions over R are equivalent if they have the same effect on all relations over R.

Definition 3.43 (Equivalent transactions) Two transactions, T_1 and T_2, over a relation schema R are said to be *equivalent* if for all relations, r, over R, $[T_1](r) = [T_2](r)$. $\blacksquare$

We next define a normal form for transactions which will be useful in proving that the equivalence of transactions can be decided in polynomial time in the size of the transactions.

Definition 3.44 (Normal form transaction) Let T be a transaction over R. Then the *active domain* of T with respect to an attribute A $\in$ schema(R), denoted by ADOM(T, A), is the set of all values in DOM(A) that occur in the conditions of the updates of T. The active domain

of T with respect to R (or simply the active domain of T when R is understood from context), denoted by ADOM(T), is given by

$$\mathrm{ADOM(T)} = \bigcup_{\mathrm{A} \in \mathrm{schema(R)}} \mathrm{ADOM(T, A)}.$$

We associate with T and each attribute, A ∈ schema(R), a set of *normal form conditions*, denoted by NF(T, A), given by

$$\mathrm{NF(T, A)} = \{\mathrm{A} = a \mid a \in \mathrm{ADOM(T, A)}\} \bigcup \left\{ \bigwedge_{a \in \mathrm{ADOM(T,A)}} \mathrm{A} \neq a \right\}.$$

The set of normal form conditions for T, denoted by NF(T), is the set of all possible conjunctions of normal form conditions having one normal form condition from NF(T, A) for each attribute A in schema(R) such that A appears in T. Formally NF(T) is given by

$$\mathrm{NF(T)} = \left\{ \bigwedge_{\mathrm{A} \in \mathrm{schema(R)}} C_\mathrm{A} \mid C_\mathrm{A} \in \mathrm{NF(T, A)} \right\}.$$

The transaction T is in *normal form* if *every* condition C occurring in T is in NF(T). ■

It follows that a transaction T is in normal form if for every two conditions C_1 and C_2 occurring in T the set of tuples $\{t \mid t \models C_1\}$ is either disjoint or equal to the set of tuples $\{t \mid t \models C_2\}$. To see this, suppose for example that schema(R) = {A, B}, ADOM(T, A) = {0, 1} and ADOM(T, B) = {1, 2}. Then NF(T) = {A = 0 ∧ B = 1, A = 0 ∧ B = 2, A = 0 ∧ B ≠ 1 ∧ B ≠ 2, A = 1 ∧ B = 1, A = 1 ∧ B = 2, A = 1 ∧ B ≠ 1 ∧ B ≠ 2, A ≠ 0 ∧ A ≠ 1 ∧ B = 1, A ≠ 0 ∧ A ≠ 1 ∧ B = 2, A ≠ 0 ∧ A ≠ 1 ∧ B ≠ 1 ∧ B ≠ 2}.

From Definition 3.44 we have that insertions are always in normal form. On the other hand, transactions consisting of deletions and modifications may not be in normal form. For example, the transaction, delete(DN ≠ Biology ∧ SL = 30K), modify(DN ≠ Maths; DN = Computing), is *not* in normal form, while the transaction, delete(DN ≠ Biology ∧ DN ≠ Maths ∧ DN ≠ Computing ∧ SL = 30K), delete(DN = Maths ∧ SL = 30K), delete(DN = Computing ∧ SL = 30K), modify(DN ≠ Biology ∧ DN ≠ Maths ∧ DN ≠ Computing ∧ SL = 30K; DN = Computing ∧ SL = 35K), modify(DN = Computing ∧ SL = 30K; DN = Computing ∧ SL = 35K), modify(DN = Biology ∧ SL = 35K; DN = Computing ∧ SL = 30K), is in normal form. (We observe that the active domain of EN with respect to both these transactions is empty.)

Given a condition C over R we let the *restriction* of C to the attributes in schema(R)−{A}, denoted by $C\mid_{\neg \mathrm{A}}$, to be the condition C with any simple condition of the form A = a or A ≠ a removed from C.

The following two axioms, called the *split* axioms, allow us by their repeated application to transform any transaction into normal form.

Definition 3.45 (Split axioms) The two split axioms are given by

SPLIT1: delete(C) is transformed into the equivalent transaction:

$$\mathrm{delete}(C \wedge \mathrm{A} \neq a),\ \mathrm{delete}(C\mid_{\neg \mathrm{A}} \wedge \mathrm{A} = a),$$

SPLIT2: modify(C; C') is transformed into the equivalent transaction:

$$\text{modify}(C \wedge \text{A} \neq a; C_1), \text{modify}(C\,|_{\neg\text{A}} \wedge \text{A} = a; C_2),$$

where A $\in$ schema(R), $a \in$ DOM(A), A $\neq a$ is not one of the simple conditions of C, and for each $b \in$ DOM(A), there is no simple condition in C of the form A = b. Moreover, $C_1 = C_2 = C'$ if A = b is a simple condition in C' for some $b \in$ DOM(A), otherwise $C_1 = C' \wedge \text{A} \neq a$ and $C_2 = C'\,|_{\neg\text{A}} \wedge \text{A} = a$. ■

Intuitively, when the split axioms transform a condition C the set of tuples satisfying C is partitioned into two complementary sets: the set of tuples satisfying C and A $\neq a$ and the set of tuples satisfying C and A = a. This allows us to apply the resulting updates to each one of the two sets of tuples independently.

For example, the SPLIT1 axiom transforms delete(DN = Maths $\wedge$ EN $\neq$ Jack) into the transaction: delete(DN = Maths $\wedge$ EN $\neq$ Jack $\wedge$ EN $\neq$ John), delete(DN = Maths $\wedge$ EN = John). As another example, the SPLIT2 axiom transforms modify(DN $\neq$ Biology; DN = Maths) into the transaction: modify(DN $\neq$ Biology $\wedge$ DN $\neq$ Computing; DN = Maths), modify(DN = Computing; DN = Maths). As a final example, the SPLIT2 axiom transforms modify(EN $\neq$ John; DN = Computing) into the transaction: modify(EN $\neq$ John $\wedge$ EN $\neq$ Jack; DN = Computing $\wedge$ EN $\neq$ Jack), modify(EN = Jack; DN = Computing $\wedge$ EN = Jack). It is now evident that applying the SPLIT axioms repeatedly to a transaction results in an equivalent transaction.

Lemma 3.7 Given a transaction T over R, a normal form transaction T′, which is equivalent to T and such that for all A $\in$ schema(R), ADOM(T, A) $\subseteq$ ADOM(T′, A), can be found in polynomial time in any finite set of values ADOM(T′) that includes ADOM(T).

Proof. We assume without loss of generality that T consists of a single update. Otherwise, we can transform each update in T into normal form and then concatenate all of the resulting normal form updates to obtain the desired normal form transaction T′.

In order to transform T into normal form we iteratively apply the relevant split axiom with respect to some value in ADOM(T′) until the current state of T is in normal form. The number of such iterations is bounded by $O(|\text{ADOM}(\text{T}')|^{\text{type}(\text{R})})$, since by Definition 3.45 each update can only be split once with respect to a given domain value. The result follows since this transformation is polynomial in ADOM(T′), although we note that it is exponential in type(R). □

The next theorem establishes the central result of this section, namely that transaction equivalence can be decided in polynomial time in the size of the transactions being tested [AV88].

Theorem 3.8 The problem of whether two transactions over a relation schema R are equivalent can be decided in polynomial time in the number of active domain values in the two transactions.

Proof. Let T $= upd_1, upd_2, \ldots, upd_n$ be a transaction over R which is in normal form. We show how to transform T into a pair (f, s), called a *transition* over R, where f is a partial mapping from the set of all tuples over R to themselves and s is a relation over R. The relation s corresponds to the tuples that are inserted by T and is equal to $[\mathrm{T}](\emptyset)$. The partial mapping f is defined from the updates, upd_i, as the composition $f_n f_{n-1} \cdots f_2 f_1$ (we denote composition of mappings by juxtaposition), where f_i, $i \in \{1, 2, \ldots, n-1, n\}$, is defined as follows:

1) If upd_i is the insertion update, insert(C), then for all tuples t over R, $f(t) = t$, i.e. in this case f is the identity mapping.
2) If upd_i is the deletion update, delete(C), then for all tuples t over R such that $t \models C, f(t)$ is undefined and for all other tuples t' over R such that $t' \not\models C, f(t') = t'$.
3) If upd_i is the modification update, modify(C_1; C_2), then for all tuples t over R such that $t \models C_1, f(t) = [\mathrm{modify}(C_1; C_2)](\{t\})$ and for all other tuples t' over R such that $t' \not\models C_1$, $f(t') = t'$.

We observe that although the mapping f is defined at the individual tuple level, conceptually we can view the tuples as being grouped into sets of tuples according to the condition C that they satisfy, where C is a normal form condition.

The *effect* of a transition (f, s), as constructed above, on a relation r over R is given by

$$[(f, s)](r) = \{f(t) \mid t \in r \cap dom(f)\} \cup s,$$

where $dom(f)$ denotes the domain of the partial mapping f; we note that if $t \notin r \cap dom(f)$ then $f(t)$ is undefined. It is evident that the effect of a transaction T on a relation r is the same as the effect of the transition, which was constructed from T, on r, i.e. $[(f, s)](r) = [\mathrm{T}](r)$.

Next, let T_1 and T_2 be two transactions over R. By Lemma 3.7 we can assume without any loss of generality that T_1 and T_2 are in normal form and that ADOM(T_1) = ADOM(T_2). When T_1 and T_2 are in normal form we can transform them in polynomial time in the sizes of T_1 and T_2 into the transitions (f_1, s_1) and (f_2, s_2), respectively, as described above.

We say that two transitions (f_1, s_1) and (f_2, s_2) over R are *equivalent* if for all relations r over R the effect of (f_1, s_1) on r is the same as the effect of (f_2, s_2) on r, i.e. $[(f_1, s_1)](r) = [(f_2, s_2)](r)$. It, therefore, remains to decide whether (f_1, s_1) and (f_2, s_2) are equivalent.

Now, it can be shown that (f_1, s_1) and (f_2, s_2) are equivalent if and only if the following two conditions obtain:

1) $s_1 = s_2$, i.e. both transitions insert the same tuples, and
2) for each distinct $i, j \in \{1, 2\}$ and for each tuple t over R, if $f_i(t)$ is defined and is not in s_j, then $f_j(t)$ is defined and $f_i(t) = f_j(t)$, i.e. if one transition modifies a tuple t that was not the result of an insertion, then the other transition must also modify this tuple in the same manner.

We observe that if $f_i(t)$ is undefined then $f_j(t)$ must also be undefined, otherwise a contradiction occurs due to the following argument. If $f_j(t)$ is defined and it is not in s_j then by condition (2) $f_i(t) = f_j(t)$, otherwise if $f_j(t) \in s_j$ then $f_j(t) = f_i(t) = t$, since $s_i = s_j$ by condition (1).