Appendix A
Overview of Relational DBMS

The aim of this appendix is to define the relational database terminology and concepts used in the book, since most of the distributed database technology has been developed using the relational model. Our focus here is on the language and operators. All of the material covered in this appendix is the subject of standard undergraduate database courses and available in any good database book.

A.1 Basic Concepts

A *database* is a structured collection of data related to some real-life phenomena that we are trying to model. A *relational database* is one where the database structure is in the form of tables. Formally, a relation R defined over n sets $D_1, D_2, \ldots, D_n$ (not necessarily distinct) is a set of *n-tuples* (or simply *tuples*) $\langle d_1, d_2, \ldots, d_n \rangle$ such that $d_1 \in D_1, d_2 \in D_2, \ldots, d_n \in D_n$. In a relational database D_is are the domains and a relation R consists of a set of *attributes* $\mathtt{A}_1, \mathtt{A}_2, \ldots, \mathtt{A}_n$ each of which has one of the D_is as its domain that restrict the values the attribute can assume. This is written as $\mathtt{R}(\mathtt{A}_1 : D_1, \mathtt{A}_2 : D_2, \ldots, \mathtt{A}_n : D_n)$; the domain specifications are usually omitted unless necessary.

Example A.1. As an example we use a database that models an engineering company. The entities to be modeled are the *employees* (EMP) and *project*s (PROJ). For each employee, we would like to keep track of the employee number (ENO), name (ENAME), title in the company (TITLE), salary (SAL), identification number of the project(s) the employee is working on (PNO), responsibility within the project (RESP), and duration of the assignment to the project (DUR) in months. Similarly, for each project we would like to store the project number (PNO), the project name (PNAME), and the project budget (BUDGET).

The *relation schemas* for this database can be defined as follows:

EMP(ENO, ENAME, TITLE, SAL, PNO, RESP, DUR)

PROJ(PNO, PNAME, BUDGET)

EMP

ENO	ENAME	TITLE	SAL	PNO	RESP	DUR

PROJ

PNO	PNAME	BUDGET

Fig. A.1: Sample Database Scheme

In relation scheme EMP, there are seven *attributes*: ENO, ENAME, TITLE, SAL, PNO, RESP, DUR. The values of ENO come from the *domain* of all valid employee numbers, say D_1, the values of ENAME come from the domain of all valid names, say D_2, and so on. Note that each attribute of each relation does not have to come from a distinct domain. Various attributes within a relation or from a number of relations may be defined over the same domain. ♦

The *key* of a relation scheme is the minimum non-empty subset of its attributes such that the values of the attributes comprising the key uniquely identify each tuple of the relation. The attributes that make up key are called *prime* attributes. The superset of a key is usually called a *superkey*. We are interested in the minimal set of attributes that uniquely identify each tuple. Each relation has at least one key. Sometimes, there may be more than one possibility for the key. In such cases, each alternative is considered a *candidate key*, and one of the candidate keys is chosen as the *primary key*, which we denote by underlining. Thus in our example the key of PROJ is PNO, and that of EMP is the set (ENO, PNO). The number of attributes of a relation defines its *degree*, whereas the number of tuples of the relation defines its *cardinality*.

In tabular form, the example database consists of two tables, as shown in Figure A.1. The columns of the tables correspond to the attributes of the relations; if there were any information entered as the rows, they would correspond to the tuples. The empty table, showing the structure of the table, corresponds to the *relation schema*; when the table is filled with rows, it corresponds to a *relation instance*. Since the information within a table varies over time, many instances can be generated from one relation scheme. Note that from now on, the term *relation* refers to a relation instance. In Figure A.2 we depict instances of the two relations that are defined in Figure A.1.

An attribute value may be undefined. This lack of definition may have various interpretations, the most common being “unknown” or “not applicable”. This special value of the attribute is generally referred to as the *null value*. The representation of a null value must be different from any other domain value, and special care should be given to differentiate it from zero. For example, value “0” for attribute DUR is *known information* (e.g., in the case of a newly hired employee), while value “null” for DUR means unknown. Supporting null values is an important feature necessary to deal with *maybe* queries [Codd, 1979].

EMP

ENO	ENAME	TITLE	SAL	PNO	RESP	DUR
E1	J. Doe	Elect. Eng.	40000	P1	Manager	12
E2	M. Smith	Analyst	34000	P1	Analyst	24
E2	M. Smith	Analyst	34000	P2	Analyst	6
E3	A. Lee	Mech. Eng.	27000	P3	Consultant	10
E3	A. Lee	Mech. Eng.	27000	P4	Engineer	48
E4	J. Miller	Programmer	24000	P2	Programmer	18
E5	B. Casey	Syst. Anal.	34000	P2	Manager	24
E6	L. Chu	Elect. Eng.	40000	P4	Manager	48
E7	R. Davis	Mech. Eng.	27000	P3	Engineer	36
E8	J. Jones	Syst. Anal.	34000	P3	Manager	40

PROJ

PNO	PNAME	BUDGET
P1	Instrumentation	150000
P2	Database Develop.	135000
P3	CAD/CAM	250000
P4	Maintenance	310000

Fig. A.2: Sample Database Instance

A.2 Normalization

The aim of normalization is to eliminate various anomalies (or undesirable aspects) of a relation in order to obtain "better" relations. The following four problems might exist in a relation scheme:

1. *Repetition anomaly*. Certain information may be repeated unnecessarily. Consider, for example, the `EMP` relation in Figure A.2. The name, title, and salary of an employee are repeated for each project on which this person serves. This is obviously a waste of storage and is contrary to the spirit of databases.
2. *Update anomaly*. As a consequence of the repetition of data, performing updates may be troublesome. For example, if the salary of an employee changes, multiple tuples have to be updated to reflect this change.
3. *Insertion anomaly*. It may not be possible to add new information to the database. For example, when a new employee joins the company, we cannot add personal information (name, title, salary) to the `EMP` relation unless an

appointment to a project is made. This is because the key of EMP includes the attribute PNO, and null values cannot be part of the key.

4. *Deletion anomaly*. This is the converse of the insertion anomaly. If an employee works on only one project, and that project is terminated, it is not possible to delete the project information from the EMP relation. To do so would result in deleting the only tuple about the employee, thereby resulting in the loss of personal information we might want to retain.

Normalization transforms arbitrary relation schemes into ones without these problems. A relation with one or more of the above mentioned anomalies is split into two or more relations of a higher *normal form*. A relation is said to be in a normal form if it satisfies the conditions associated with that normal form. Codd initially defined the *first*, *second*, and *third* normal forms (1NF, 2NF, and 3NF, respectively). Boyce and Codd [Codd, 1974] later defined a modified version of the third normal form, commonly known as the *Boyce-Codd normal form* (BCNF). This was followed by the definition of the *fourth* (4NF) [Fagin, 1977] and *fifth* normal forms (5NF) [Fagin, 1979].

There is a hierarchical relationship among these normal forms. Every normalized relation is in 1NF; some of the relations in 1NF are also in 2NF, some of which are in 3NF, and so on. The higher normal forms have better properties than others with respect to the four anomalies discussed above.

One of the requirements of a normalization process is that the decomposition be lossless. This means that the replacement of a relation by several others should not result in loss of information. If it is possible to join the decomposed relations to obtain the original relation, the process is said to be a *lossless decomposition*.

The join operation is defined formally in Section A.3.1. Intuitively, it is an operation that takes two relations and concatenates tuples from both that satisfy a specified condition. The condition is defined over the attributes of the two relations. For example, it might be specified that the value of an attribute of the first relation should be equal to the value of an attribute of the second relation.

Another requirement of the normalization process is *dependency preservation*. A decomposition is said to be dependency preserving if the union of the dependencies in the decomposed relations is equivalent to the closure (with respect to a set of inference rules) of the dependencies of the original relation.

A.2.1 Dependency Structures

The normal forms are based on certain dependency structures. BCNF and lower normal forms are based on *functional dependencies* (FDs), 4NF is based on *multivalued dependencies*, and 5NF is based on *projection-join dependencies*. We only introduce functional dependency, since that is the only relevant one for the example we are considering.

Let R be a relation defined over the set of attributes $A = \{A_1, A_2, \ldots, A_n\}$ and let $X \subset A$, $Y \subset A$. If for each value of X in R , there is only one associated Y value, we say that "X *functionally determines* Y" or that "Y is *functionally dependent* on X." Notationally, this is shown as $X \rightarrow Y$. The key of a relation functionally determines the non-key attributes of the same relation.

Example A.2. For example, in the PROJ relation of Example A.1 (one can observe these in Figure A.2 as well), the valid FD is

PNO → (PNAME, BUDGET)

In the EMP relation we have

(ENO, PNO) → (ENAME,TITLE,SAL,RESP,DUR)

This last FD is not the only FD in EMP, however. If each employee is given unique employee numbers, we can write

ENO→ (ENAME, TITLE, SAL)

(ENO, PNO) → (RESP, DUR)

If the salary for a given position is fixed, it would give rise to one more FD

TITLE→ SAL

♦

A.2.2 Normal Forms

The first normal form (1NF) states simply that the attributes of the relation contain atomic values only. In other words, the tables should be flat with no repeating groups. The relations EMP and PROJ in Figure A.2 satisfy this condition, so they both are in 1NF.

Relations in 1NF still suffer from the anomalies discussed earlier. To eliminate some of these anomalies, they should be decomposed into relations in higher normal forms. We are not particularly interested in the second normal form. In fact, it is only of historical importance, since there are algorithms that take a 1NF relation and directly normalize it to third normal form (3NF) or higher.

A relation R is in 3NF if for each FD $X \rightarrow Y$ where Y is not in X, either X is a superkey of R or Y is a prime attribute. There are algorithms that provide a lossless and dependency-preserving decomposition of a 1NF relation into a 3NF relation.

Boyce-Codd normal form (BCNF) is a stronger form of 3NF. The definitions are identical except for the last part. For a relation to be in BCNF, for every FD $X \rightarrow Y$, X has to be a superkey. Notice that the clause "or Y is a prime attribute" is deleted from the definition. The final form of relation EMP, as well as the relations PAY, PROJ, and ASG, are in BCNF.

It is possible to decompose a 1NF relation directly into a set of relations in BCNF. These algorithms are guaranteed to generate lossless decompositions; however, they cannot be guaranteed to preserve dependencies.

The following example shows the result of normalization on the sample database that we introduced in Example A.1.

Example A.3. The following set of relation schemes are normalized into BCNF with respect to the functional dependencies defined over the relations.

EMP(ENO, ENAME, TITLE)

PAY(TITLE, SAL)

PROJ(PNO, PNAME, BUDGET)

ASG(ENO, PNO, RESP, DUR)

The normalized instances of these relations are shown in Figure A.3. ♦

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

ASG

ENO	PNO	RESP	DUR
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E3	P3	Consultant	10
E3	P4	Engineer	48
E4	P2	Programmer	18
E5	P2	Manager	24
E6	P4	Manager	48
E7	P3	Engineer	36
E8	P3	Manager	40

PROJ

PNO	PNAME	BUDGET
P1	Instrumentation	150000
P2	Database Develop.	135000
P3	CAD/CAM	250000
P4	Maintenance	310000

PAY

TITLE	SAL
Elect. Eng.	40000
Syst. Anal.	34000
Mech. Eng.	27000
Programmer	24000

Fig. A.3: Normalized Relations

A.3 Relational Data Languages

Data manipulation languages developed for the relational model (commonly called *query languages*) fall into two fundamental groups: *relational algebra* languages and *relational calculus* languages. The difference between them is based on how the user query is formulated. The relational algebra is procedural in that the user is expected to specify, using certain high-level operators, how the result is to be obtained. The relational calculus, on the other hand, is non-procedural; the user only specifies the relationships that should hold in the result. Both of these languages were originally proposed by Codd [1970], who also proved that they were equivalent in terms of expressive power [Codd, 1972].

A.3.1 Relational Algebra

Relational algebra consists of a set of operators that operate on relations. Each operator takes one or two relations as operands and produces a result relation, which, in turn, may be an operand to another operator. These operations permit the querying and updating of a relational database.

There are five fundamental relational algebra operators and five others that can be defined in terms of these. The fundamental operators are *selection*, *projection*, *union*, *set difference*, and *Cartesian product*. The first two of these operators are unary operators, and the last three are binary operators. The additional operators that can be defined in terms of these fundamental operators are *intersection*, $\theta - join$, *natural join*, *semijoin* and *division*. In practice, relational algebra is extended with operators for grouping or sorting the results, and for performing arithmetic and aggregate functions. Other operators, such as *outer join* and *transitive closure*, are sometimes used as well to provide additional functionality. We only discuss the more common operators.

The operands of some of the binary relations should be *union compatible*. Two relations `R` and `S` are union compatible if and only if they are of the same degree and the *i*-th attribute of each is defined over the same domain. The second part of the definition holds, obviously, only when the attributes of a relation are identified by their relative positions within the relation and not by their names. If relative ordering of attributes is not important, it is necessary to replace the second part of the definition by the phrase "the corresponding attributes of the two relations should be defined over the same domain." The correspondence is defined rather loosely here.

Many operator definitions refer to "formula", which also appears in relational calculus expressions we discuss later. Thus, let us define precisely, at this point, what we mean by a formula. We define a formula within the context of first-order predicate calculus (since we use that formalism later), and follow the notation of Gallaire et al. [1984]. First-order predicate calculus is based on a *symbol alphabet* that consists of (1) variables, constants, functions, and predicate symbols; (2) parentheses; (3) the logical connectors $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implication), and $\leftrightarrow$ (equivalence); and (4)

quantifiers $\forall$ (for all) and $\exists$ (there exists). A *term* is either a constant or a variable. Recursively, if f is an n-ary function and $t_1, \ldots, t_n$ are terms, $f(t_1, \ldots, t_n)$ is also a term. An *atomic formula* is of the form $P(t_1, \ldots, t_n)$, where P is an n-ary predicate symbol and the t_i's are terms. A *well-formed formula* (*wff*) can be defined recursively as follows: If w_i and w_j are wffs, then (w_i), $\neg(w_i)$, $(w_i) \wedge (w_j)$, $(w_i) \vee (w_j)$, $(w_i) \rightarrow (w_j)$, and $(w_i) \leftrightarrow (w_j)$ are all wffs. Variables in a wff may be *free* or they may be *bound* by one of the two quantifiers.

Selection.

Selection produces a horizontal subset of a given relation. The subset consists of all the tuples that satisfy a formula (condition). The selection from a relation R is

$$\sigma_F(\mathrm{R})$$

where R is the relation and F is a formula.

The formula in the selection operation is called a *selection predicate* and is an atomic formula whose terms are of the form $\mathtt{A}\theta c$, where A is an attribute of R and θ is one of the arithmetic comparison operators $<$, $>$, $=$, $\neq$, $\leq$, and $\geq$. The terms can be connected by the logical connectors $\wedge$, $\vee$, and $\neg$. Furthermore, the selection predicate does not contain any quantifiers.

Example A.4. Consider the relation EMP shown in Figure A.3. The result of selecting those tuples for electrical engineers is shown in Figure A.4. ♦

$\sigma_{\text{TITLE="Elect. Eng."}}(\text{EMP})$

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E6	L. Chu	Elect. Eng.

Fig. A.4: Result of Selection

Projection.

Projection produces a vertical subset of a relation. The result relation contains only those attributes of the original relation over which projection is performed. Thus the degree of the result is less than or equal to the degree of the original relation.

The projection of relation R over attributes A and B is denoted as

$$\Pi_{A,B}(R)$$

Note that the result of a projection might contain tuples that are identical. In that case the duplicate tuples may be deleted from the result relation. It is possible to specify projection with or without duplicate elimination.

Example A.5. The projection of relation PROJ shown in Figure A.3 over attributes PNO and BUDGET is depicted in Figure A.5. ♦

$\Pi_{PNO,BUDGET}$ (PROJ)

PNO	BUDGET
P1	150000
P2	135000
P3	250000
P4	310000

Fig. A.5: Result of Projection

Union.

The union of two relations R and S (denoted as $R \cup S$) is the set of all tuples that are in R, or in S, or in both. We should note that R and S should be union compatible. As in the case of projection, the duplicate tuples are normally eliminated. Union may be used to insert new tuples into an existing relation, where these tuples form one of the operand relations.

Intersection.

Intersection of two relations R and S ($R \cap S$) consists of the set of all tuples that are in both R and S. In terms of the basic operators, it can be specified as follows:

$$R \cap S = R - (R - S)$$

Set Difference.

The set difference of two relations R and S ($R - S$) is the set of all tuples that are in R but not in S. In this case, not only should R and S be union compatible, but

EMP x PAY

ENO	ENAME	EMP.TITLE	PAY.TITLE	SAL
E1	J. Doe	Elect. Eng.	Elect. Eng.	40000
E1	J. Doe	Elect. Eng.	Syst. Anal.	34000
E1	J. Doe	Elect. Eng.	Mech. Eng.	27000
E1	J. Doe	Elect. Eng.	Programmer	24000
E2	M. Smith	Syst. Anal.	Elect. Eng.	40000
E2	M. Smith	Syst. Anal.	Syst. Anal.	34000
E2	M. Smith	Syst. Anal.	Mech. Eng.	27000
E2	M. Smith	Syst. Anal.	Programmer	24000
E3	A. Lee	Mech. Eng.	Elect. Eng.	40000
E3	A. Lee	Mech. Eng.	Syst. Anal.	34000
E3	A. Lee	Mech. Eng.	Mech. Eng.	27000
E3	A. Lee	Mech. Eng.	Programmer	24000
≈	≈	≈	≈	≈
E8	J. Jones	Syst. Anal.	Elect. Eng.	40000
E8	J. Jones	Syst. Anal.	Syst. Anal.	34000
E8	J. Jones	Syst. Anal.	Mech. Eng.	27000
E8	J. Jones	Syst. Anal.	Programmer	24000

Fig. A.6: Partial Result of Cartesian Product

the operation is also asymmetric (i.e., $\mathtt{R} - \mathtt{S} \neq \mathtt{S} - \mathtt{R}$). This operation allows the deletion of tuples from a relation. Together with the union operation, we can perform modification of tuples by deletion followed by insertion.

Cartesian Product.

The Cartesian product of two relations R of degree k_1 and S of degree k_2 is the set of $(k_1 + k_2)$-tuples, where each result tuple is a concatenation of one tuple of R with one tuple of S, for all tuples of R and S. The Cartesian product of R and S is denoted as $\mathtt{R} \times \mathtt{S}$.

It is possible that the two relations might have attributes with the same name. In this case the attribute names are prefixed with the relation name so as to maintain the uniqueness of the attribute names within a relation.

Example A.6. Consider relations EMP and PAY in Figure A.3. $\mathtt{EMP} \times \mathtt{PAY}$ is shown in Figure A.6. Note that the attribute TITLE, which is common to both relations, appears twice, prefixed with the relation name. ♦

EMP

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng
E2	M. Smith	Syst. Anal.
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.
E9	A. Hsu	Programmer
E10	T. Wong	Syst. Anal.

(a)

EMP $\bowtie_{\text{EMP.ENO=ASG.ENO}}$ ASG

ENO	ENAME	TITLE	PNO	RESP	DUR
E1	J. Doe	Elect. Eng.	P1	Manager	12
E2	M. Smith	Syst. Anal.	P1	Analyst	12
E2	M. Smith	Syst. Anal.	P2	Analyst	12
E3	A. Lee	Mech. Eng.	P3	Consultant	12
E3	A. Lee	Mech. Eng.	P4	Engineer	12
E4	J. Miller	Programmer	P2	Programmer	12
E5	J. Miller	Syst. Anal.	P2	Manager	12
E6	L. Chu	Elect. Eng.	P4	Manager	12
E7	R. Davis	Mech. Eng.	P3	Engineer	12
E8	J. Jones	Syst. Anal.	P3	Manager	12

(b)

Fig. A.7: The Result of Join

θ-Join.

Join is a derivative of Cartesian product. There are various forms of join; the primary classification is between *inner join* and *outer join*. We first discuss inner join and its variants and then describe outer join.

The most general type of inner join is the θ-join. The θ-join of two relations R and S is denoted as

$$\mathtt{R} \bowtie_F \mathtt{S}$$

where F is a formula specifying the *join predicate*. A join predicate is specified similar to a selection predicate, except that the terms are of the form $\mathtt{R.A}\theta\mathtt{S.B}$, where A and B are attributes of R and S, respectively.

The join of two relations is equivalent to performing a selection, using the join predicate as the selection formula, over the Cartesian product of the two operand relations. Thus

$$\mathtt{R} \bowtie_F \mathtt{S} = \sigma_F(\mathtt{R} \times \mathtt{S})$$

In the equivalence above, we should note that if F involves attributes of the two relations that are common to both of them, a projection is necessary to make sure that those attributes do not appear twice in the result.

Example A.7. Let us consider the EMP relation in Figure A.3 and add two more tuples as depicted in Figure A.7(a). Then Figure A.7(b) shows the θ-join of relations EMP and ASG over the join predicate EMP.ENO=ASG.ENO.

The same result could have been obtained as

$$\mathtt{EMP} \bowtie_{\mathtt{EMP.ENO=ASG.ENO}} \mathtt{ASG} = \Pi_{\mathtt{ENO, ENAME, TITLE, SAL}}(\sigma_{\mathtt{EMP.ENO=PAY'ENO}}(\mathtt{EMP} \times \mathtt{ASG}))$$

EMP $\bowtie_{\text{TITLE}}$ PAY

ENO	ENAME	TITLE	SAL
E1	J. Doe	Elect. Eng.	40000
E2	M. Smith	Analyst	34000
E3	A. Lee	Mech. Eng.	27000
E4	J. Miller	Programmer	24000
E5	B. Casey	Syst. Anal.	34000
E6	L. Chu	Elect. Eng.	40000
E7	R. Davis	Mech. Eng.	27000
E8	J. Jones	Syst. Anal.	34000

Fig. A.8: The Result of Natural Join

Notice that the result does not have tuples E9 and E10 since these employees have not yet been assigned to a project. Furthermore, the information about some employees (e.g., E2 and E3) who have been assigned to multiple projects appear more than once in the result. ♦

This example demonstrates a special case of θ-join which is called the *equijoin*. This is a case where the formula F only contains equality (=) as the arithmetic operator. It should be noted, however, that an equijoin does not have to be specified over a common attribute as the example above might suggest.

A natural join is an equijoin of two relations over a specified attribute, more specifically, over attributes with the same domain. There is a difference, however, in that usually the attributes over which the natural join is performed appear only once in the result. A natural join is denoted as the join without the formula

$$R \bowtie_A S$$

where A is the attribute common to both R and S. We should note here that the natural join attribute may have different names in the two relations; what is required is that they come from the same domain. In this case the join is denoted as

$$R \bowtie_{R.A=S.B} S$$

where Aand B are the join attributes of R and S respectively.

Example A.8. The join of EMP and ASG in Example A.7 is actually a natural join. Here is another example – Figure A.8 shows the natural join of relations EMP and PAY in Figure A.3 over the attribute TITLE.

♦

Inner join requires the joined tuples from the two operand relations to satisfy the join predicate. In contrast, outer join does not have this requirement – tuples exist in

the result relation regardless. Outer join can be of three types: left outer join (⟕), right outer join (⟖) and full outer join (⟗). In the left outer join, the tuples from the left operand relation are always in the result, in the case of right outer join, the tuples from the right operand are always in the result, and in the case of full outer relation, tuples from both relations are always in the result. If there are no matching tuples from the other relation, its attributes have "Null" values. Outer join is useful in those cases where we wish to include information from one or both relations even if the do not satisfy the join predicate.

Example A.9. Consider the left outer join of EMP (as revised in Example A.7) and ASG over attribute ENO(i.e., EMP $⟕_{ENO}$ ASG). The result is given in Figure A.9. Notice that the information about two employees, E9 and E10 are included in the result even thought they have not yet been assigned to a project with "Null" values for the attributes from the ASG relation. ♦

EMP ⟕ ENO ASG

ENO	ENAME	TITLE	PNO	RESP	DUR
E1	J. Doe	Elect. Eng.	P1	Manager	12
E2	M. Smith	Syst. Anal.	P1	Analyst	12
E2	M. Smith	Syst. Anal.	P2	Analyst	12
E3	A. Lee	Mech. Eng.	P3	Consultant	12
E3	A. Lee	Mech. Eng.	P4	Engineer	12
E4	J. Miller	Programmer	P2	Programmer	12
E5	J. Miller	Syst. Anal.	P2	Manager	12
E6	L. Chu	Elect. Eng.	P4	Manager	12
E7	R. Davis	Mech. Eng.	P3	Engineer	12
E8	J. Jones	Syst. Anal.	P3	Manager	12
E9	A. Hsu	Programmer	Null	Null	Null
E10	T. Wong	Syst. Anal.	Null	Null	Null

Fig. A.9: The Result of Left Outer Join

Semijoin.

The semijoin of relation R , defined over the set of attributes A, by relation S, defined over the set of attributes B, is the subset of the tuples of R that participate in the join of R with S. It is denoted as $R ⋉_F S$ (where F is a predicate as defined before) and can be obtained as follows:

$$R ⋉_F S = \Pi_A(R ⋈_F S) = \Pi_A(R) ⋈_F \Pi_{A∩B}(S)$$
$$= R ⋈_F \Pi_{A∩B}(S)$$

The advantage of semijoin is that it decreases the number of tuples that need to be handled to form the join. In centralized database systems, this is important because it usually results in a decreased number of secondary storage accesses by making better use of the memory. It is even more important in distributed databases since it usually reduces the amount of data that needs to be transmitted between sites in order to evaluate a query. Note that the operation is asymmetric (i.e., $R \ltimes_F S \neq S \ltimes_F R$).

Example A.10. To demonstrate the difference between join and semijoin, let us consider the semijoin of EMP with PAY over the predicate EMP.TITLE= PAY.TITLE, that is,

$$EMP \ltimes_{EMP.TITLE = PAY.TITLE} PAY$$

The result of the operation is shown in Figure A.10. We encourage readers to compare Figures A.7 and A.10 to see the difference between the join and the semijoin operations. Note that the resultant relation does not have the SAL attribute and is therefore smaller. ♦

EMP $\bowtie_{EMP.TITLE=PAY.TITLE}$ PAY

ENO	ENAME	TITLE
E1	J. Doe	Elect. Eng.
E2	M. Smith	Analyst
E3	A. Lee	Mech. Eng.
E4	J. Miller	Programmer
E5	B. Casey	Syst. Anal.
E6	L. Chu	Elect. Eng.
E7	R. Davis	Mech. Eng.
E8	J. Jones	Syst. Anal.

Fig. A.10: The Result of Semijoin

Division.

The division of relation R of degree r with relation S of degree s (where $r > s$ and $s \neq 0$) is the set of $(r - s)$-tuples t such that for all s-tuples u in S, the tuple tu is in R. The division operation is denoted as $R \div S$ and can be specified in terms of the fundamental operators as follows:

$$R \div S = \Pi_{\bar{A}}(R) - \Pi_{\bar{A}}((\Pi_{\bar{A}}(R) \times S) - R)$$

where $\bar{A}$ is the set of attributes of R that are not in S [i.e., the $(r - s)$-tuples].

Example A.11. Assume that we have a modified version of the ASG relation (call it ASG′) depicted in Figure A.11a and defined as follows:

$$\text{ASG}' = \Pi_{\text{ENO,PNO}}\ (\text{ASG}) \bowtie_{\text{PNO}} \text{PROJ}$$

If one wants to find the employee numbers of those employees who are assigned to all the projects that have a budget greater than $200,000, it is necessary to divide ASG′ with a restricted version of PROJ, called PROJ′ (see Figure A.11b). The result of division (ASG′÷ PROJ′) is shown in Figure A.11c.

The keyword in the query above is "*all*." This rules out the possibility of doing a selection over ASG′ to find the necessary tuples, since that would only produce those that correspond to employees working on *some* project with a budget greater than $200,000, not those who work on all projects. Note that the result contains only the tuple ⟨E3⟩ since the tuples ⟨E3, P3, CAD/CAM, 250000⟩ and ⟨E3, P4, Maintenance, 310000⟩ both exist in ASG′. On the other hand, for example, ⟨E7⟩ is not in the result, since even though the tuple ⟨E7, P3, CAD/CAM, 250000⟩ is in ASG′, the tuple ⟨E7, P4, Maintenance, 310000⟩ is not. ♦

Since all operations take relations as input and produce relations as outputs, we can nest operations using a parenthesized notation and represent relational algebra programs. The parentheses indicate the order of execution. The following are a few examples that demonstrate this.

Example A.12. Consider the relations of Figure A.3. The retrieval query

"Find the names of employees working on the CAD/CAM project"

can be answered by the relational algebra program

$$\Pi_{\text{ENAME}}(((\sigma_{\text{PNAME= "CAD/CAM"}}\ \text{PROJ}) \bowtie_{\text{PNO}} \text{ASG}) \bowtie_{\text{ENO}} \text{EMP})$$

The order of execution is: the selection on PROJ, followed by the join with ASG, followed by the join with EMP, and finally the projection on ENAME.

An equivalent program where the size of the intermediate relations is smaller is

$$\Pi_{\text{ENAME}}\ (\text{EMP} \ltimes_{\text{ENO}} (\Pi_{\text{ENO}} (\text{ASG} \ltimes_{\text{PNO}} (\sigma_{\text{PNAME="CAD/CAM"}}\ \text{PROJ}))))$$

♦

Example A.13. The update query

"Replace the salary of programmers by $25,000"

can be computed by

$$(\text{PAY} - (\sigma_{\text{TITLE= "Programmer"}}\ \text{PAY})) \cup (\langle \text{Programmer, 25000} \rangle)$$

♦

ASG'

ENO	PNO	PNAME	BUDGET
E1	P1	Instrumentation	150000
E2	P1	Instrumentation	150000
E2	P2	Database Develop.	135000
E3	P3	CAD/CAM	250000
E3	P4	Maintenance	310000
E4	P2	Database Develop.	135000
E5	P2	Database Develop.	135000
E6	P4	Maintenance	310000
E7	P3	CAD/CAM	250000
E8	P3	CAD/CAM	250000

(a)

PROJ'

PNO	PNAME	BUDGET
P3	CAD/CAM	250000
P4	Maintenance	310000

(b)

(ASG' ÷ PROJ')

ENO
E3

(c)

Fig. A.11: The Result of Division

A.3.2 Relational Calculus

In relational calculus-based languages, instead of specifying *how* to obtain the result, one specifies *what* the result is by stating the relationship that is supposed to hold for the result. Relational calculus languages fall into two classes: *tuple relational calculus* and *domain relational calculus*. The difference between the two is in terms of the primitive variable used in specifying the queries. We briefly review these two types of languages.

Relational calculus languages have a solid theoretical foundation since they are based on first-order predicate logic as we discussed before. Semantics is given to formulas by interpreting them as assertions on the database. A relational database can be viewed as a collection of tuples or a collection of domains. Tuple relational calculus interprets a variable in a formula as a tuple of a relation, whereas domain relational calculus interprets a variable as the value of a domain.

Tuple relational calculus.

The primitive variable used in tuple relational calculus is a *tuple variable* which specifies a tuple of a relation. In other words, it ranges over the tuples of a relation. Tuple calculus is the original relational calculus developed by Codd [1970].

In tuple relational calculus queries are specified as $\{t|F(t)\}$, where t is a tuple variable and F is a well-formed formula. The atomic formulas are of two forms:

1. *Tuple-variable membership expressions*. If t is a tuple variable ranging over the tuples of relation R (predicate symbol), the expression "tuple t belongs to relation R" is an atomic formula, which is usually specified as R.t or R(t).
2. *Conditions*. These can be defined as follows:
   - **(a)** $s[\mathtt{A}]\theta t[\mathtt{B}]$, where s and t are tuple variables and A and B are components of s and t, respectively. θ is one of the arithmetic comparison operators $<$, $>$, $=$, $\neq$, $\leq$, and $\geq$. This condition specifies that component A of s stands in relation θ to the B component of t: for example, $s[\mathtt{SAL}] > t[\mathtt{SAL}]$.
   - **(b)** $s[\mathtt{A}]\theta c$, where s, A, and θ are as defined above and c is a constant. For example, $s[\mathtt{ENAME}]$ = "Smith".

Note that A is defined as a component of the tuple variable s. Since the range of s is a relation instance, say S, it is obvious that component A of s corresponds to attribute A of relation S. The same thing is obviously true for B.

There are a number of languages that are based on relational tuple calculus, the most popular one being SQL[1] [Date, 1987]. We use SQL as the user language in this book. SQL is now an international standard (actually, the only one) with various versions released: SQL1 was released in 1986, modifications to SQL1 were included in the 1987 and 1989 versions, SQL2 was issued in 1992, and SQL3, with object-oriented language extensions, was released in 1999. There have been minor revisions in 2003, 2006, and 2008. In 2011 another major release was made adding temporal data support, 2016 saw a major release with JSON support, and 2019 another release with multidimensional array data type support. Although the names SQL1-SQL3 are still used, the common convention now is to refer to these as SQLxx where xx is the last two digits of the release year.

Domain relational calculus.

The domain relational calculus was first proposed by Lacroix and Pirotte [1977]. The fundamental difference between a tuple relational language and a domain relational

[1] Sometimes SQL is cited as lying somewhere between relational algebra and relational calculus. Its originators called it a "mapping language." However, it follows the tuple calculus definition quite closely; hence we classify it as such.

language is the use of a *domain variable* in the latter. A domain variable ranges over the values in a domain and specifies a component of a tuple. In other words, the range of a domain variable consists of the domains over which the relation is defined. The wffs are formulated accordingly. The queries are specified in the following form:

$$x_1, x_2, ..., x_n | F(x_1, x_2, ..., x_n)$$

where F is a wff in which $x_1, \ldots, x_n$ are the free variables.