The Relational Data Model and Relational Database Constraints

In the formal relational model terminology, a row is called a *tuple,* a column header is called an *attribute,* and the table is called a *relation.* The data type describing the types of values that can appear in each column is represented by a *domain.*

Domains, Attributes, Tuples, and Relations

A **domain** *D* is a set of atomic values. By **atomic** we mean that each value in the domain is indivisible as far as the formal relational model is concerned.

A **relation schema** *R*, denoted by *R*(*A*1, *A*2, ...,A*n*), is made up of a relation name *R* and a list of attributes, *A*1, *A*2, ..., *An*.

Each **attribute** *Ai* is the name of a role played by some domain *D* in the relation schema *R*. *D* is called the **domain** of *Ai* and is denoted by **dom**(**A**i*).*

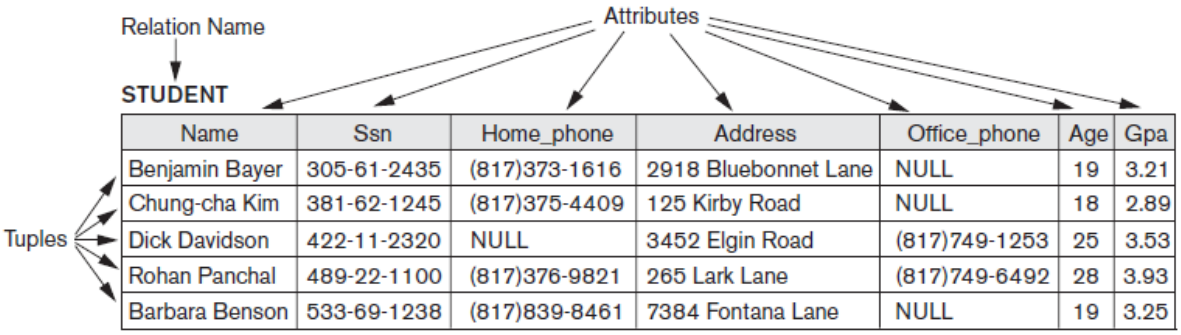
A relation schema is used to *describe* a relation, *R* is called the **name** of this relation.

The **degree** (or **arity**) of a relation is the number of attributes *n* of its relation schema.

A relation of degree seven, which stores information about university students, would contain seven attributes describing each student. as follows:

The terms **relation intension** for the schema *R* .

Relation extension for a relation state *r*(*R*) are also commonly used. STUDENT(Name, Ssn, Home_phone, Address, Office_phone, Age, Gpa).



Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Benjamin Bayer	305-61-2435	(817)373-1616	2918 Bluebonnet Lane	NULL	19	3.21
Chung-cha Kim	381-62-1245	(817)375-4409	125 Kirby Road	NULL	18	2.89
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	(817)749-1253	25	3.53
Rohan Panchal	489-22-1100	(817)376-9821	265 Lark Lane	(817)749-6492	28	3.93
Barbara Benson	533-69-1238	(817)839-8461	7384 Fontana Lane	NULL	19	3.25

A relation (or relation state) $r(R)$ is a **mathematical relation** of degree n on the domains dom($A1$), dom($A2$), ..., dom(An), which is a **subset** of the **Cartesian product** (denoted by x) of the domains that define R:

$$r(R) \subseteq (\text{dom}(A_1) \times \text{dom}(A_2) \times \ldots \times \text{dom}(A_n))$$

Informal Terms	**Formal Terms**
Table	Relation
Column	Attribute/Domain
Row	Tuple
Values in a column	Domain
Table Definition	Schema of a Relation
Populated Table	Extension

Characteristics of Relations

Ordering of Tuples in a Relation.

A relation is defined as a *set* of tuples. Mathematically, elements of a set have *no order* among them; hence, tuples in a relation do not have any particular order. In other words, a relation is not sensitive to the ordering of tuples. Tuple ordering is not part of a relation definition because a relation attempts to represent facts at a logical or abstract level.

Ordering of Values within a Tuple and an Alternative Definition of a Relation.

At a more abstract level, the order of attributes and their values is *not* that important as long as the correspondence between attributes and values is

maintained. An **alternative definition** of a relation can be given, making the ordering of values in a tuple *unnecessary.*

Values and NULLs in the Tuples.

Each value in a tuple is an **atomic** value hence, composite and multivalued attributes are not allowed. This model is sometimes called the **flat relational model**.

An important concept is that of NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple. A special value, called NULL, is used in these cases.

Interpretation (Meaning) of a Relation.

The relation schema can be interpreted as a declaration or a type of **assertion**. For example, the schema of the **STUDENT** relation- in general, a student entity has a Name, USN,Home_phone, Address, Office_phone, Age, and Gpa.

Each tuple in the relation can then be interpreted as a **fact** or a particular instance of the assertion.

Notice that some relations may represent facts about *entities,* whereas other relations may represent facts about *relationships.*

Relational Model Notation:

R(A1, A2, ..., An) is a relational schema of degree *n* denoting that there is a relation *R* having as its attributes *A1, A2, ..., An*. By convention, *Q*, *R*, and *S* denote relation names.

By convention, *q*, *r*, and *s* denote relation states. For example, *r(R)* denotes one possible state of relation *R*. If *R* is understood from context, this could be written, more simply, as *r*.

By convention, *t*, *u*, and *v* denote tuples.

The "dot notation" *R.A* (e.g., STUDENT.Name) is used to qualify an attribute name, usually for the purpose of distinguishing it from a same-named attribute in a different relation (e.g., DEPARTMENT.Name).

Relational Model Constraints and Relational Database Schemas

There are generally many restrictions or **constraints** on the actual values in a database state. These constraints are derived from the rules in the miniworld that the database represents.

Domain Constraints

Domain constraints specify that within each tuple, the value of each attribute *A* must be an atomic value from the domain dom(*A*). The data types associated with domains typically include standard numeric data types for integers (such as short integer, integer, and long integer) and real numbers (float and double precision float). Characters, Booleans, fixed-length strings, and variable-length strings are also available, as are date, time, timestamp, and money, or other special data types. Other possible domains may be described by a subrange of values from a data type or as an enumerated data type in which all possible values are explicitly listed.

Key Constraints and Constraints on NULL Values

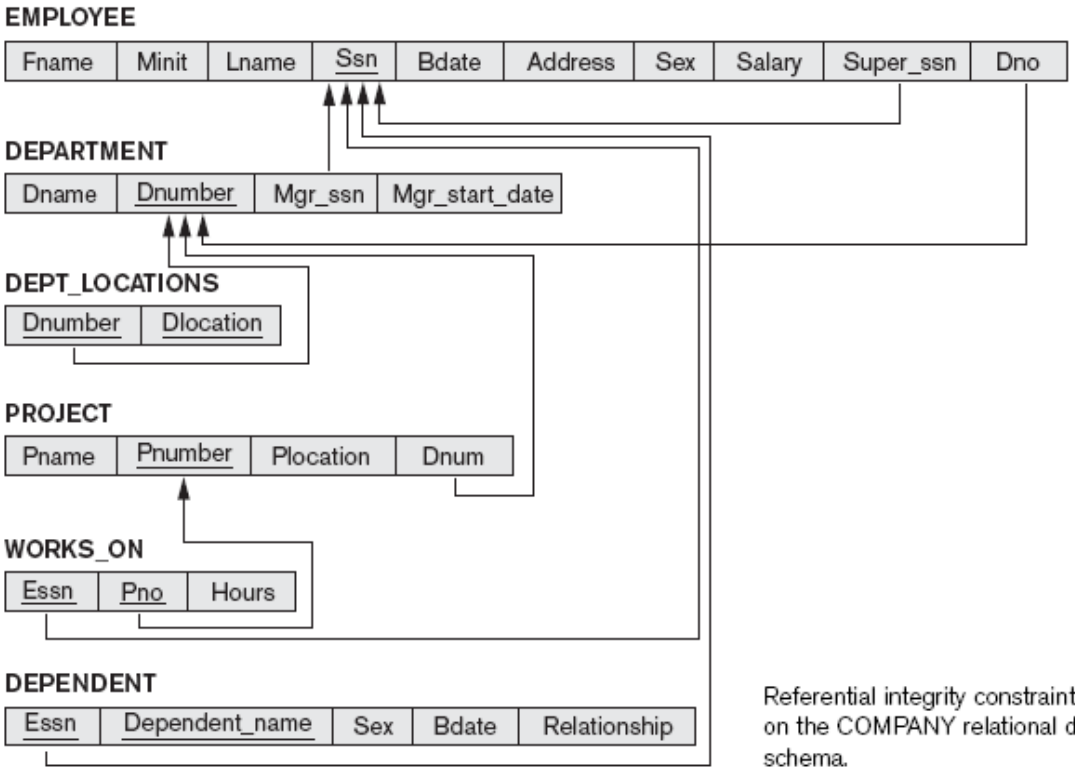
A *relation* is defined as a *set of tuples.* By definition,all elements of a set are distinct, hence- all tuples in a relation must also be distinct. *There are other* ***subsets of attributes*** *of a relation schema R with the property that no two tuples in any relation state r of R should have the same combination of values for these attributes.* Suppose that we denote one such subset of attributes by SK,then for any two *distinct* tuples *t*1 and *t2* in a relation state *r* of *R*, we have the constraint that: *t*1[SK]≠*t2*[SK]

Integrity, Referential Integrity, and Foreign Keys

The **entity integrity constraint** states that no primary key value can be NULL. This is because the primary key value is used to identify individual tuples in a relation

Having NULL values for the primary key implies that we cannot identify some tuples. For example, if two or more tuples had NULL for their primary keys, we may not be able to distinguish them if we try to reference them from other relations.

The **referential integrity constraint** is specified between two relations and is used to maintain the consistency among tuples in the two relations. the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an *existing tuple* in that relation. *The attribute Dno of EMPLOYEE gives the department number for which each employee works; hence, its value in every EMPLOYEE tuple must match the Dnumber value of some tuple in the DEPARTMENT relation.*



Referential integrity constraints displayed on the COMPANY relational database schema.

We can *diagrammatically display referential integrity constraints* by drawing a **directed arc** from each foreign key to the relation it references. For clarity, the arrowhead may point to the primary key of the referenced relation.

The types of constraints we discussed so far may be called **state constraints** because they define the constraints that a *valid state* of the database must satisfy.

Update Operations, Transactions, and Dealing with Constraint Violations

The operations of the relational model can be categorized into *retrievals* and *updates.* There are three basic operations that can change the states of relations in the database: ***Insert, Delete, and Update (or Modify).***

They insert new data, delete old data, or modify existing data records.
Insert is used to insert one or more new tuples in a relation, **Delete** is used to delete tuples.

Update (or **Modify**) is used to change the values of some attributes in existing tuples.
Whenever these operations are applied, the integrity constraints specified on the relational database schema should not be violated.

The Insert Operation

The **Insert** operation provides a list of attribute values for a new tuple *t* that is to be inserted into a relation *R. Domain constraints can be violated if an attribute value is given that does not appear in the corresponding domain or is not of the appropriate data type. Key constraints can be violated if a key value in the new tuple t already exists in another tuple in the relation r(R). Entity integrity can be violated if any part of the primary key of the new tuple t is NULL. Referential integrity can be violated if the value of any foreign key in t refers to a tuple that does not exist in the referenced relation.*

The Delete Operation

The **Delete** operation can violate only referential integrity. This occurs if the tuple being deleted is referenced by foreign keys from other tuples in the database.

The Update Operation

The **Update** (or **Modify**) operation is used to change the values of one or more attributes in a tuple (or tuples) of some relation *R*. It is necessary to specify a condition on the attributes of the relation to select the tuple (or tuples) to be modified.

The relational algebra is very important for several reasons. First, it provides a formal foundation for relational model operations. Second, and perhaps more important, it is used as a basis for implementing and optimizing queries in the query processing and optimization modules that are integral parts of relational database management systems (RDBMSs), Third, some of its concepts are incorporated into the SQL standard query language for RDBMSs.

Unary Operations that operate on single relations.

JOIN and other complex **binary operations**, which operate on two tables by combining related tuples (records) based on *join conditions*.

Unary Relational Operations:

SELECT and PROJECT

The SELECT Operation:

The SELECT operation is used to choose a *subset* of the tuples from a relation that satisfies a **selection condition.** In general, the SELECT operation is denoted by σ

σ<selection condition>(*R*)

where the symbol σ (sigma) is used to denote the SELECT operator and the selection condition is a Boolean expression (condition) specified on the attributes of relation *R*.

The Boolean expression specified in <selection condition> is made up of a number of **clauses** of the form

<attribute name> <comparison op> <constant value>

or

<attribute name> <comparison op> <attribute name>

where <attribute name> is the name of an attribute of *R*, <comparison op> is normally one of the operators {=, <, >, ≥,≤,≠ }, and <constant value> is a constant value from the attribute domain.

Clauses can be connected by the standard Boolean operators ***and, or, not***

$\sigma_{Dno=4}(EMPLOYEE)$

$\sigma_{Salary>30000}(EMPLOYEE)$

Similarly,

$\sigma_{Dno=4\ \mathbf{AND}\ Salary>25000}(EMPLOYEE)$

```
SELECT    *
FROM      EMPLOYEE
WHERE     Dno=4 AND Salary>25000;
```

The **PROJECT** operation, on the other hand, selects certain *columns* from the table and discards the other columns. If we are interested in only certain attributes of a relation, we use the PROJECT operation to *project* the relation over these attributes only.

π Lname, Fname, Salary(EMPLOYEE)

The general form of the PROJECT operation is

π<attribute list>(R)

where π (pi) is the symbol used to represent the PROJECT operation, and <attribute list> is the desired sublist of attributes from the attributes of relation *R*.

The result of the PROJECT operation has only the attributes specified in <attribute list> *in the same order as they appear in the list.* Hence, its **degree**

is equal to the number of attributes in <attribute list>. The PROJECT operation *removes any duplicate tuples*, so the result of the PROJECT operation is a set of distinct tuples, and hence a valid relation. This is known as **duplicate elimination**.

$\pi_{Sex, Salary}(EMPLOYEE)$

```
SELECT    DISTINCT Sex, Salary
FROM      EMPLOYEE
```

Sequences of Operations and the RENAME Operation

A single relational algebra expression, also known as an **in-line expression**, as follows:

$$\pi_{Fname, Lname, Salary}(\sigma_{Dno=5}(EMPLOYEE))$$

shows the result of this in-line relational algebra expression.

Alternatively, we can explicitly show the sequence of operations, giving a name to each intermediate relation, as follows:

$$DEP5\_EMPS \leftarrow \sigma_{Dno=5}(EMPLOYEE)$$
$$RESULT \leftarrow \pi_{Fname, Lname, Salary}(DEP5\_EMPS)$$

It is sometimes simpler to break down a complex sequence of operations by specifying intermediate result relations than to write a single relational algebra expression.

We can also use this technique to **rename** the attributes.

We can also define a formal **RENAME** operation—which can rename either the relation name or the attribute names, or both—as a unary operator. The general RENAME operation when applied to a relation R of degree n is denoted by any of the following three forms:

$$\rho_{S(B1, B2, \ldots, Bn)}(R) \quad \text{or} \quad \rho_{S}(R) \quad \text{or} \quad \rho_{(B1, B2, \ldots, Bn)}(R)$$

Relational Algebra Operations from Set Theory

The UNION, INTERSECTION, and MINUS Operations

Several set theoretic operations are used to merge the elements of two sets in various ways, including **UNION**, **INTERSECTION**, and **SET DIFFERENCE** (also called **MINUS** or **EXCEPT**)

The three operations UNION, INTERSECTION, and SET DIFFERENCE on two union-compatible relations *R* and *S* as follows:

■ **UNION:** The result of this operation, denoted by *R* U*S*, is a relation that includes all tuples that are either in *R* or in *S* or in both *R* and *S*. Duplicate tuples are eliminated.

■ **INTERSECTION:** The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both *R* and *S*.

■ **SET DIFFERENCE** (or **MINUS**): The result of this operation, denoted by *R* – *S*, is a relation that includes all tuples that are in *R* but not in *S*.

Notice that both UNION and INTERSECTION are *commutative operations*; that is, R **U** S = S **U** R and $R \cap S = S \cap R$

Both **UNION** and **INTERSECTION** can be treated as *n*-ary operations applicable to any number of relations because both are also *associative operations;* that is, $R \mathbf{U} (S \mathbf{U} T) = (R \mathbf{U} S) \mathbf{U} T$ and $(R \cap S) \cap T = R \cap (S \cap T)$

1

The **MINUS** operation is *not commutative;* that is, in general, $R - S \neq S - R$

INTERSECTION can be expressed in terms of union and set difference as follows: $R \cap S = ((R \mathbf{U} S) - (R - S)) - (S - R)$

The set operations UNION, INTERSECTION, and MINUS.

(a) **STUDENT**

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

(a) Two union-compatible relations.

(b)

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

(b) STUDENT ∪ INSTRUCTOR.

(c)

Fn	Ln
Susan	Yao
Ramesh	Shah

(c) STUDENT ∩ INSTRUCTOR.

(d)

Fn	Ln
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(d) STUDENT – INSTRUCTOR.

(e)

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

(e) INSTRUCTOR – STUDENT.

The CARTESIAN PRODUCT (CROSS PRODUCT) Operation

CARTESIAN PRODUCT operation—also known as **CROSS PRODUCT** or **CROSS JOIN**—which is denoted by ×. This is also a binary set operation, but the relations on which it is applied do *not* have to be union compatible.

In general, the result of $R(A1, A2, \ldots, An) \times S(B1, B2, \ldots, Bm)$ is a relation Q with degree $n + m$ attributes $Q(A1, A2, \ldots, An, B1, B2, \ldots, Bm)$, in that order. The resulting relation Q has one tuple for each combination of tuples—one from R and one from S.

Retrieve a list of names of each female employee's dependents

$$\text{FEMALE\_EMPS} \leftarrow \sigma_{\text{Sex}='F'}(\text{EMPLOYEE})$$
$$\text{EMPNAMES} \leftarrow \pi_{\text{Fname, Lname, Ssn}}(\text{FEMALE\_EMPS})$$
$$\text{EMP\_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$$
$$\text{ACTUAL\_DEPENDENTS} \leftarrow \sigma_{\text{Ssn}=\text{Essn}}(\text{EMP\_DEPENDENTS})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname, Lname, Dependent\_name}}(\text{ACTUAL\_DEPENDENTS})$$

Binary Relational Operations: JOIN and DIVISION

The JOIN Operation

The **JOIN** operation, denoted by ⋈ , is used to combine *related tuples* from two relations into single ―longer‖ tuples. To illustrate JOIN, suppose that we want to retrieve the name of the manager of each department.

To get the manager‘s name, we need to combine each department tuple with the employee tuple whose Ssn value matches the Mgr_ssn value in the department tuple.

$$DEPT\_MGR \leftarrow DEPARTMENT \bowtie_{Mgr\_ssn=Ssn} EMPLOYEE$$
$$RESULT \leftarrow \pi_{Dname,\ Lname,\ Fname}(DEPT\_MGR)$$

Note that Mgr_ssn is a foreign key of the DEPARTMENT relation that references Ssn, the primary key of the EMPLOYEE relation. This referential integrity constraint plays a role in having matching tuples in the referenced relation EMPLOYEE.

DEPT_MGR

Dname	Dnumber	Mgr_ssn	···	Fname	Minit	Lname	Ssn	···
Research	5	333445555	···	Franklin	T	Wong	333445555	···
Administration	4	987654321	···	Jennifer	S	Wallace	987654321	···
Headquarters	1	888665555	···	James	E	Borg	888665555	···

Result of the JOIN operation $DEPT\_MGR \leftarrow DEPARTMENT \bowtie_{Mgr\_ssn=Ssn} EMPLOYEE$.

The general form of a JOIN operation on two relations5 $R(A1, A2, \ldots, An)$ and $S(B1, B2, \ldots, Bm)$ is

$$R \bowtie_{\text{<join condition>}} S$$

A general join condition is of the form <condition> **AND** <condition> **AND**...**AND** <condition> where each <condition> is of the form $Ai\ \Theta Bj$, Ai is an attribute of R, Bj is an attribute of S, Ai and Bj have the same domain, and Θ(theta) is one of the comparison operators {=, <, <=, >, >=, ≠}. A JOIN operation with such a general join condition is called a **THETA JOIN**.

The DIVISION Operation

The DIVISION operation, denoted by ÷, is useful for a special kind of query that sometimes occurs in database applications.

Q: Retrieve the names of employees who work on all the projects that 'John Smith' works on.

To express this query using the DIVISION operation, proceed as follows. First, retrieve the list of project numbers that _John Smith' works on in the intermediate relation SMITH_PNOS:

using the DIVISION operation retrieve the list of project numbers that _John Smith' works on in the intermediate relation SMITH_PNOS:

$$SMITH \leftarrow \sigma_{Fname='John' \textbf{ AND } Lname='Smith'}(EMPLOYEE)$$
$$SMITH\_PNOS \leftarrow \pi_{Pno}(WORKS\_ON \bowtie_{Essn=Ssn} SMITH)$$

create a relation that includes a tuple <Pno, Essn> whenever the employee whose Ssn is Essn works on the project whose number is Pno in the intermediate relation SSN_PNOS:

$$SSN\_PNOS \leftarrow \pi_{Essn,\ Pno}(WORKS\_ON)$$

Finally, apply the DIVISION operation to the two relations, which gives the desired employees' Social Security numbers:

$$SSNS(Ssn) \leftarrow SSN\_PNOS \div SMITH\_PNOS$$
$$RESULT \leftarrow \pi_{Fname,\ Lname}(SSNS * EMPLOYEE)$$

The DIVISION operation. (a) Dividing SSN_PNOS by SMITH_PNOS. (b) $T \leftarrow R \div S$.

(a)

SSN_PNOS

Essn	Pno
123456789	1
123456789	2
666884444	3
453453453	1
453453453	2
333445555	2
333445555	3
333445555	10
333445555	20
999887777	30
999887777	10
987987987	10
987987987	30
987654321	30
987654321	20
888665555	20

SMITH_PNOS

Pno
1
2

SSNS

Ssn
123456789
453453453

(b)

R

A	B
a1	b1
a2	b1
a3	b1
a4	b1
a1	b2
a3	b2
a2	b3
a3	b3
a4	b3
a1	b4
a2	b4
a3	b4

S

A
a1
a2
a3

T

B
b1
b4

ER-to-Relational Mapping Algorithm

Step 1: Mapping of Regular Entity Types

Step 2: Mapping of Weak Entity Types

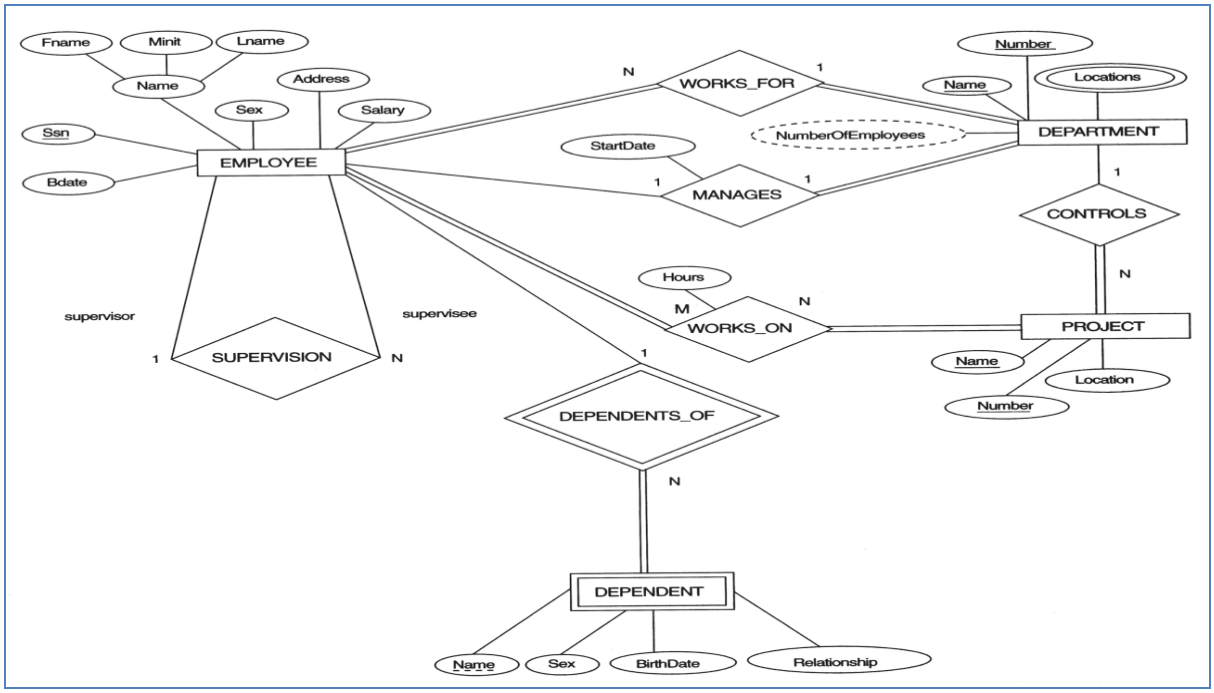
Step 3: Mapping of Binary 1:1 Relation Types

Step 4: Mapping of Binary 1:N Relationship Types.

Step 5: Mapping of Binary M:N Relationship Types.

Step 6: Mapping of Multivalued attributes.

Step 7: Mapping of N-ary Relationship Types.



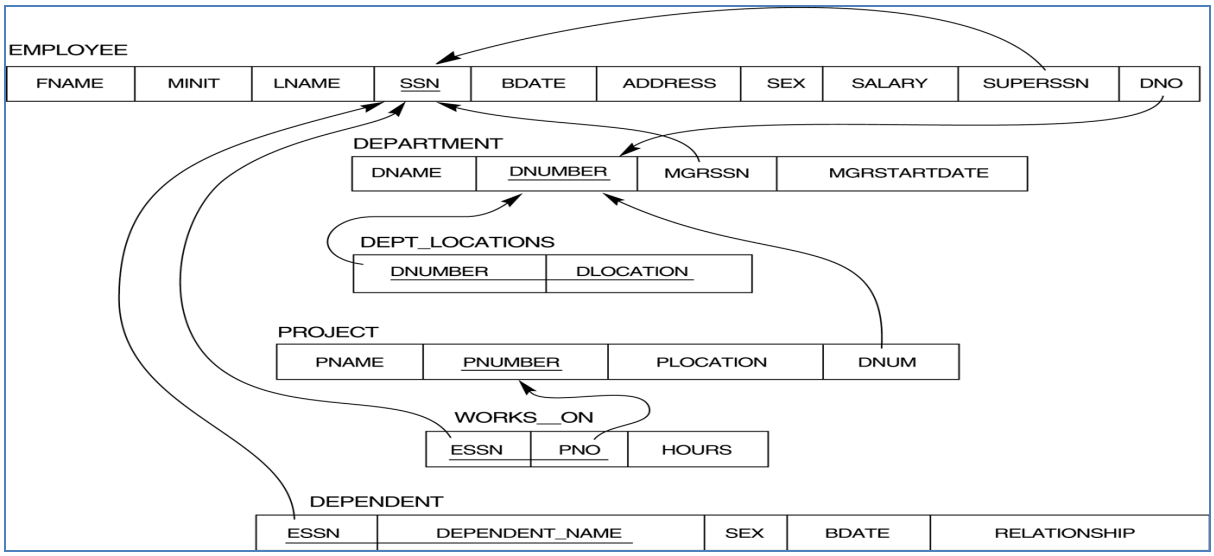


FIGURE : Result of mapping the COMPANY ER schema into a relational schema

Step 1: Mapping of Regular Entity Types.

For each regular (strong) entity type E in the ER schema, create a relation R that includes all the simple attributes of E. Choose one of the key attributes of E as the primary key for R. If the chosen key of E is composite, the set of simple attributes that form it will together form the primary key of R. Example: EMPLOYEE, DEPARTMENT, and PROJECT are the regular entities in the ER diagram. SSN, DNUMBER, and PNUMBER are the primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT respectively.

Step 2: Mapping of Weak Entity Types For each weak entity type W in the ER schema with owner entity type E, create a relation R and include all simple attributes of W as attributes of R.

Example: Create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT. Include the primary key SSN of the EMPLOYEE relation as a foreign key attribute of DEPENDENT (renamed to ESSN). The primary key of the DEPENDENT relation is the combination {ESSN, DEPENDENT_NAME} because DEPENDENT_NAME is the partial key of DEPENDENT.

Step 3: Mapping of Binary 1:1 Relation Types For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R.

Example: 1:1 relation MANAGES is mapped by choosing the participating entity type DEPARTMENT to serve in the role of S, because its participation in the MANAGES relationship type is total.

Step 4: Mapping of Binary 1:N Relationship Types. For each regular binary 1:N relationship type R, identify the relation S that represent the participating entity type at the N-side of the relationship type.

Example: 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION in the figure. For WORKS_FOR we include the primary key DNUMBER of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it DNO.

Step 5: Mapping of Binary M:N Relationship Types. For each regular binary M:N relationship type R, *create a new relation* S to represent R. Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; *their combination will form the primary key* of S.

Example: The M:N relationship type WORKS_ON from the ER diagram is mapped by creating a relation WORKS_ON in the relational database schema. The primary keys of the PROJECT and EMPLOYEE relations are included as foreign keys in WORKS_ON and renamed PNO and ESSN, respectively. Attribute HOURS in WORKS_ON represents the HOURS attribute of the relation type. The primary key of the WORKS_ON relation is the combination of the foreign key attributes {ESSN, PNO}.

Step 6: Mapping of Multivalued attributes. For each multivalued attribute A, create a new relation R. This relation R will include an attribute corresponding to A, plus the primary key attribute K-as a foreign key in R-of the relation that represents the entity type of relationship type that has A as an attribute.

Example: The relation DEPT_LOCATIONS is created. The attribute DLOCATION represents the multivalued attribute LOCATIONS of DEPARTMENT, while DNUMBER-as foreign key-represents the primary key of the DEPARTMENT relation. The primary key of R is the combination of {DNUMBER, DLOCATION}.

Step 7: Mapping of N-ary Relationship Types. For each n-ary relationship type R, where n>2, create a new relationship S to represent R. **Example:** The relationship type SUPPY in the ER below. This can be mapped to the relation SUPPLY shown in the relational schema, whose primary key is the combination of the three foreign keys {SNAME, PARTNO, PROJNAME}

Consider the following schema: Suppliers(*sid:* integer, *sname:* string, *address:* string) **Parts**(*pid:* integer, *pname:* string, *color:* string) **Catalog**(*sid:* integer, *pid:* integer, *cost:* real)

1. Find the *name*s of suppliers who supply some red part.

$$\pi_{sname}(\pi_{sid}((\pi_{pid}\sigma_{color='red'}Parts) \bowtie Catalog) \bowtie Suppliers)$$

SQL: SELECT S.sname
FROM Suppliers S, Parts P, Catalog C
WHERE P.color='red' AND C.pid=P.pid AND C.sid=S.sid

2. Find the *sid*s of suppliers who supply some red or green part.

$$\pi_{sid}(\pi_{pid}(\sigma_{color='red' \vee color='green'}Parts) \bowtie catalog)$$

SQL:
SELECT C.sid
FROM Catalog C, Parts P
WHERE (P.color = _red' OR P.color = _green') AND P.pid = C.pid

3. Find the *sid*s of suppliers who supply some red part or are at 221 Packer Street.

$$\rho(R1, \pi_{sid}((\pi_{pid}\sigma_{color='red'}Parts) \bowtie Catalog))$$
$$\rho(R2, \pi_{sid}\sigma_{address='221PackerStreet'}Suppliers)$$
$$R1 \cup R2$$

SQL:
SELECT S.sid
FROM Suppliers S
WHERE S.address = _221 Packer street' OR S.sid IN
(SELECT C.sid
FROM Parts P, Catalog C
WHERE P.color='red' AND P.pid = C.pid
)

4. **Find the *sid*s of suppliers who supply some red part and some green part**.

$$\rho(R1, \pi_{sid}((\pi_{pid}\sigma_{color='red'}Parts) \bowtie Catalog))$$
$$\rho(R2, \pi_{sid}((\pi_{pid}\sigma_{color='green'}Parts) \bowtie Catalog))$$
$$R1 \cap R2$$

SQL:
SELECT C.sid
FROM Parts P, Catalog C
WHERE P.color = _red' AND P.pid = C.pid AND
EXISTS (SELECT P2.pid
FROM Parts P2, Catalog C2
WHERE P2.color = _green' AND C2.sid = C.sid AND P2.pid = C2.pid)

5. Find the *sid*s of suppliers who supply every part.

$$(\pi_{sid,pid}Catalog)/(\pi_{pid}Parts)$$

SQL:

```
SELECT C.sid
FROM Catalog C
WHERE NOT EXISTS (SELECT P.pid FROM Parts P WHERE NOT EXISTS
(SELECT C1.sid FROM Catalog C1 WHERE C1.sid = C.sid AND C1.pid = P.pid))
```

6. Find the *sid*s of suppliers who supply every red part.

$$(\pi_{sid,pid}Catalog)/(\pi_{pid}\sigma_{color='red'}Parts)$$

SQL:

```
SELECT C.sid
FROM Catalog C
WHERE NOT EXISTS (SELECT P.pid FROM Parts P WHERE P.color = _red‘
AND (NOT EXISTS (SELECT C1.sid FROM Catalog C1 WHERE C1.sid = C.sid
AND C1.pid = P.pid)))
```

7. Find the *sid*s of suppliers who supply every red or green part.

$$(\pi_{sid,pid}Catalog)/(\pi_{pid}\sigma_{color='red'\vee color='green'}Parts)$$

SQL:

```
SELECT C.sid
FROM Catalog C
WHERE NOT EXISTS (SELECT P.pid FROM Parts P WHERE (P.color = _red‘ OR
P.color = _green‘) AND (NOT EXISTS (SELECT C1.sid FROM Catalog C1 WHERE
C1.sid = C.sid AND C1.pid = P.pid)))
```

8. **Find the *sid*s of suppliers who supply every red part or supply every green part**.

$$\rho(R1, ((\pi_{sid,pid}Catalog)/(\pi_{pid}\sigma_{color='red'}Parts)))$$
$$\rho(R2, ((\pi_{sid,pid}Catalog)/(\pi_{pid}\sigma_{color='green'}Parts)))$$
$$R1 \cup R2$$

SQL:

```
SELECT C.sid FROM Catalog C WHERE (NOT EXISTS (SELECT P.pid FROM
Parts P WHERE P.color = _red‘ AND (NOT EXISTS (SELECT C1.sid FROM
Catalog C1 WHERE C1.sid = C.sid AND C1.pid = P.pid)))) OR ( NOT EXISTS
(SELECT P1.pid FROM Parts P1 WHERE P1.color = _green‘ AND (NOT EXISTS
(SELECT C2.sid FROM Catalog C2 WHERE C2.sid = C.sid AND C2.pid =
P1.pid))))
```

9. Find pairs of *sid*s such that the supplier with the first *sid* charges more for some part than the supplier with the second *sid*.

$$\rho(R1, Catalog)$$
$$\rho(R2, Catalog)$$
$$\pi_{R1.sid,R2.sid}(\sigma_{R1.pid=R2.pid \wedge R1.sid \neq R2.sid \wedge R1.cost > R2.cost}(R1 \times R2))$$

SQL:
SELECT C1.sid, C2.sid FROM Catalog C1, Catalog C2 WHERE C1.pid = C2.pid AND C1.sid = C2.sid AND C1.cost > C2.cost

10. Find the *pid*s of parts supplied by at least two different suppliers.

$$\rho(R1, Catalog)$$
$$\rho(R2, Catalog)$$
$$\pi_{R1.pid}\sigma_{R1.pid=R2.pid \wedge R1.sid \neq R2.sid}(R1 \times R2)$$

SQL:
SELECT C.pid FROM Catalog C WHERE EXISTS (SELECT C1.sid FROM Catalog C1 WHERE C1.pid = C.pid AND C1.sid != C.sid)

11. Find the Supplier names of the suppliers who supply a red part that costs less than 100 dollars.

$$\pi_{sname}(\pi_{sid}((\sigma_{color='red'} Parts) \bowtie (\sigma_{cost<100} Catalog)) \bowtie Suppliers)$$

12. Find the Supplier names of the suppliers who supply a red part that costs less than 100 dollars and a green part that costs less than 100 dollars.

$$(\pi_{sname}((\sigma_{color='red'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers)) \cap (\pi_{sname}((\sigma_{color='green'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers))$$

13. Find the Supplier ids of the suppliers who supply a red part that costs less than 100 dollars and a green part that costs less than 100 dollars.

$$(\pi_{sid}((\sigma_{color='red'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers)) \cap (\pi_{sid}((\sigma_{color='green'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers))$$

14. Find the Supplier names of the suppliers who supply a red part that costs less than 100 dollars and a green part that costs less than 100 dollars.

$$\pi_{sname}((\pi_{sid,sname}((\sigma_{color='red'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers)) \cap (\pi_{sid,sname}((\sigma_{color='green'} Parts) \bowtie (\sigma_{cost<100} Catalog) \bowtie Suppliers)))$$

SQL – 1: *SQL Data Definition and Data Types; Specifying basic constraints in SQL; Schema change statements in SQL; Basic queries in SQL; More complex SQL Queries.*

The name **SQL** is presently expanded as Structured Query Language. Originally, SQL was called SEQUEL (Structured English QUEry Language) and was designed and implemented at IBM Research as the interface for an experimental relational database system called SYSTEM R.

SQL is now the standard language for commercial relational DBMSs. A joint effort by the American National Standards Institute (ANSI) and the International Standards Organization (ISO) has led to a standard version of SQL (ANSI 1986), called SQL-86 or SQL1.

SQL is a comprehensive database language: It has statements for data definitions, queries, and updates. Hence, it is both a DDL *and* a DML.

SQL Data Definition and Data Types SQL uses the terms **table**, **row**, and **column** for the formal relational model terms *relation*, *tuple*, and *attribute*, respectively.We will use the corresponding terms interchangeably.

The main SQL command for data definition is the **CREATE** statement,which can be used to create schemas, tables (relations), and domains (as well as other constructs such as views, assertions, and triggers).

Schema and Catalog Concepts in SQL Early versions of SQL did not include the concept of a relational database schema;

all tables (relations) were considered part of the same schema. An **SQL schema** is identified by a **schema name**, and includes an **authorization identifier** to indicate the user or account who owns the schema, as well as **descriptors** for *each element* in the schema.

Schema **elements** include tables, constraints, views, domains, and other constructs (such as authorization grants) that describe the schema. For example, the following statement creates a schema called COMPANY, owned by the user with authorization identifier 'Jsmith'.

Note that each statement in SQL ends with a semicolon. **CREATE SCHEMA** COMPANY **AUTHORIZATION** 'Jsmith';

In general, not all users are authorized to create schemas and schema elements. The privilege to create schemas, tables, and other constructs must be explicitly granted to the relevant user accounts by the system administrator or DBA.

The CREATE TABLE Command in SQL

The **CREATE TABLE** command is used to specify a new relation by giving it a name and specifying its attributes and initial constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints, such as NOT NULL.

The key, entity integrity, and referential integrity constraints can be specified within the CREATE TABLE statement after the attributes are declared, or they can be added later using the ALTER TABLE command.

For example, by writing

CREATE TABLE *COMPANY.EMPLOYEE* (……);
rather than
CREATE TABLE EMPLOYEE (EID VARCHAR(10),……); as shown above ,
we can explicitly (rather than implicitly) make the EMPLOYEE table part of the COMPANY schema.
The relations declared through CREATE TABLE statements are called ***Base Tables*** (or base relations).

This means that the relation and its tuples are actually created and stored as a file by the DBMS ***Attribute Data Types and Domains in SQL .***

The basic **data types** available for attributes include ***numeric, character string, bit string, Boolean, date, and time.***

Numeric data types include integer numbers of various sizes (*INTEGER or INT, and SMALLINT*) and floating-point (real) numbers of various precision *(FLOAT or REAL, and DOUBLE PRECISION)*. Formatted numbers can be declared by using *DECIMAL(i,j)—or DEC(i,j) or NUMERIC(i,j).*

Character-string data types are either fixed length—*CHAR(n) or CHARACTER(n)*, where *n* is the number of characters—or varying length—*VARCHAR(n) or CHAR VARYING(n) or CHARACTER VARYING(n*), where *n* is the maximum number of characters.

Another variable-length string data type called CHARACTER **LARGE OBJECT** or CLOB is also available to specify columns that have large text values, such as documents. The CLOB maximum length can be specified in kilobytes (K), megabytes (M), or gigabytes (G). For example, CLOB(20M) specifies a maximum length of 20 megabytes.

Bit-string data types are either of fixed length *n*—BIT(*n*)—or varying length—BIT VARYING(*n*), where *n* is the maximum number of bits. The default for *n*, the length of a character string or bit string, is 1.

Boolean data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a Boolean data type is UNKNOWN.

DATE data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD.

The **TIME** data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form **HH:MM:SS**.

A **timestamp** data type (TIMESTAMP) includes the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier. example, TIMESTAMP '2008-09-27 09:12:47.648302'.

Another data type related to **DATE, TIME,** and **TIMESTAMP** is the **INTERVAL** data type.

This specifies an **interval**—a *relative value* that can be used to increment or decrement an absolute value of a date, time, or timestamp.

Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.

Specifying Constraints in SQL These include *key and referential integrity constraints, restrictions on attribute domains and NULLs and constraints* on individual tuples within a relation.

Specifying Attribute Constraints and Attribute Defaults Because SQL allows NULLs as attribute values, a *constraint* NOT NULL may be specified if NULL is not permitted for a particular attribute.

It is also possible to define a *default value* for an attribute by appending the clause **DEFAULT** <value> to an attribute definition. The default value is included in any new tuple if an explicit value is not provided for that attribute.

Ex: **CREATE TABLE** EMPLOYEE (Ssn varchar(10),Ename varchar(10),salary integer,address varchar(10),……. Dno INT **NOT NULL DEFAULT** 1, **CONSTRAINT** EMPPK **PRIMARY KEY** (Ssn), **CONSTRAINT** EMPSUPERFK **FOREIGN KEY** (Super_ssn) **REFERENCES** EMPLOYEE(Ssn) **ON DELETE** SET NULL **ON UPDATE** CASCADE, **CONSTRAINT** EMPDEPTFK **FOREIGN KEY**(Dno) **REFERENCES** DEPARTMENT(Dnumber) **ON DELETE** SET DEFAULT **ON UPDATE** CASCADE);

Another type of constraint can restrict attribute or domain values using the **CHECK** clause following an attribute or domain definition.

For example, suppose that department numbers are restricted to integer numbers between 1 and 20; then, we can change the attribute declaration of Dnumber in the DEPARTMENT table to the following:

Dnumber INT ***NOT NULL CHECK*** *(Dnumber > 0* ***AND*** *Dnumber < 21);*

Specifying Key and Referential Integrity Constraints Because keys and referential integrity constraints are very important, there are special clauses within the CREATE TABLE statement to specify them.

For example, the primary key of DEPARTMENT can be specified as follows (instead of the way it is specified in Figure): *Dnumber INT* ***PRIMARY KEY****; We can specify RESTRICT, CASCADE, SET NULL or SET DEFAULT on referential integrity constraints (foreign keys)*

CREATE TABLE DEPT (DNAME VARCHAR(10) NOT NULL, DNUMBER INTEGER NOT NULL, MGRSSN CHAR(9), MGRSTARTDATE CHAR(9), PRIMARY KEY (DNUMBER), FOREIGN KEY (MGRSSN) REFERENCES EMP ON DELETE SET DEFAULT ON UPDATE CASCADE);

Specifying Constraints on Tuples Using CHECK :

other *table constraints* can be specified through additional CHECK clauses at the end of a CREATE TABLE statement. These can be called **tuple-based** constraints because they apply to each tuple *individually* and are checked whenever a tuple is inserted or modified.

Ex: CREATE TABLE statement for the DEPARTMENT table to make sure that a manager's start date is later than the department creation date. ***CHECK (Dept_create_date <= Mgr_start_date);***

Delete a Table or Database

To delete a table (the table structure, attributes, and indexes will also be deleted):

- **DROP TABLE** table_name

To delete a database:

- **DROP DATABASE** database_name

Truncate a Table

What if we only want to get rid of the data inside a table, and not the table itself? Use the **TRUNCATE TABLE** command (deletes only the data inside the table):

- **TRUNCATE TABLE** table_name

The ALTER Command

The definition of a base table or of other named schema elements can be changed by using the ALTER command.

For base tables, the possible *alter table actions* include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints.
To drop a column, we must choose either CASCADE or RESTRICT for drop behavior. If CASCADE is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If RESTRICT is chosen, the command is successful only if no views or constraints (or other elements) reference the column.
ALTER TABLE COMPANY.EMPLOYEE **DROP ADDRESS CASCADE;**
It is also possible to alter a column definition by dropping an existing default clause or by defining a new default clause. The following examples illustrate this clause:
ALTER TABLE COMPANY.DEPARTMENT **ALTER MGRSSN DROP DEFAULT;**
ALTER TABLE COMPANY.DEPARTMENT ALTER MGRSSN SET DEFAULT "333445555";
One can also change the constraints specified on a table by adding or dropping a constraint. To be dropped, a constraint must have been given a name when it was specified. For example, to drop the constraint named EMPSUPERFK in Figure 8.2 from the EMPLOYEE relation, we write:
ALTER TABLE **EMPLOYEE**
DROP CONSTRAINT EMPSUPERFK CASCADE;
The **ALTER TABLE** statement is used to add or drop columns in an existing table.

- **ALTER TABLE** table_name **ADD** column_name datatype
- **ALTER TABLE** table_name **DROP COLUMN** column_name
- **ALTER TABLE** table_name **MODIFY COLUMN** column_name
- **ALTER TABLE** table_name **RENAME** newtablename

Note: Some database systems don't allow the dropping of a column in a database table (DROP COLUMN column_name).

Person:

LastName	**FirstName**	**Address**
Yadav	Chethan	RR Nagar

Example
To add a column named "City" in the "Person" table:
ALTER TABLE Person **ADD** City varchar(30);
Result:

LastName	**FirstName**	**Address**	**City**
Yadav	Chethan	RR nagar	

Example
To drop the "Address" column in the "Person" table:

ALTER TABLE Person **DROP COLUMN** Address;
Result:

LastName	FirstName	City
Yadav	Chethan	Bangalore

The INSERT INTO Statement
The INSERT INTO statement is used to insert new rows into a table.

Syntax: INSERT INTO table_name **VALUES** (value1, value2,....)

We can also specify the columns for which you want to insert data:

INSERT INTO *table_name (column1, column2,...)***VALUES** *(value1, value2,....)*

Insert a New Row

LastName	FirstName	Address	City
Yadav	Chethan	Rr nagr	bangalore

And this SQL statement:
INSERT INTO *Persons* ***VALUES*** *('Rao', 'Praveen', 'RajajiNagar', 'Bangalore')*
Will give this result:

LastName	FirstName	Address	City
Yadav	Chethan	Rr nagr	bangalore
Rao	Praveen	RajajiNgar	Bangalore

Insert Data in Specified Columns:

INSERT INTO *Persons* ***(LastName, Address)VALUES (' Shetty ', ' banashankari ')***
Will give this result:

LastName	FirstName	Address	City
Yadav	Chethan	Rr nagr	bangalore
Rao	Praveen	RajajiNgar	Bangalore
Shetty		**banashankari**	

The Update Statement
The UPDATE statement is used to modify the data in a table.
Syntax:

UPDATE table_name
SET column_name = new_value
WHERE column_name = some_value;

Person:

LastName	FirstName	Address	City
Yadav	Chethan	Rr nagr	bangalore
Rao	Praveen	RajajiNgar	Bangalore
Shetty		banashankari	

Update one Column in a Row
We want to add a first name to the person with a last name of " Shetty ":

```
UPDATE Person SET FirstName = '  pradeep '
WHERE LastName = '  shetty ';
```

Result:

LastName	FirstName	Address	City
Kumar	Kiran	Rr nagr	bangalore
Rao	Praveen	RajajiNgar	Bangalore
Shetty	Pradeep	banashankari	

Update several Columns in a Row
We want to change the address and add the name of the city:

```
UPDATE Person
SET  City = 'Tumkur'
WHERE LastName = 'shetty';
```

Result:

LastName	FirstName	Address	City
Kumar	Kiran	Rr nagr	bangalore
Rao	Praveen	RajajiNgar	Bangalore
Shetty	Pradeep	banashankari	Tumkur

The Delete Statement
The DELETE statement is used to delete rows in a table.
Syntax:

```
DELETE FROM table_name
WHERE column_name = some_value
```

Person:

LastName	FirstName	Address	City
Yadav	Chethan	Rr nagr	bangalore
Rao	Praveen	RajajiNgar	Bangalore
Shetty	Pradeep	banashankari	Tumkur

Delete a Row
"Pradeep Shetty" is going to be deleted:

DELETE
FROM Person
WHERE LastName = 'shetty'

LastName	FirstName	Address	City
Kumar	Kiran	Rr nagr	bangalore
Rao	Praveen	RajajiNgar	Bangalore

Delete All Rows
It is possible to delete all rows in a table without deleting the table. This means that the tablestructure, attributes, and indexes will be intact:

DELETE FROM table_name;
or
DELETE * FROM table_name;

LastName	FirstName	Address	City

Try this: ***SELECT * from Person;***

Examples:
The employee JONES is transfered to the department 20 as a manager and his salary is increased by 1000:

```
UPDATE EMP set
JOB = 'MANAGER', DEPTNO = 20, SAL = SAL +1000
where ENAME = 'JONES';
```

• All employees working in the departments 10 and 30 get a 15% salary increase.

```
UPDATE EMP set
SAL = SAL * 1.15 where DEPTNO in (10,30);
```

```
UPDATE EMP set
SAL = (select min(SAL) from EMP
where JOB = 'MANAGER')
where JOB = 'SALESMAN' and DEPTNO = 20;
```

Explanation: The query retrieves the minimum salary of all managers. This value then is assigned to all salesmen working in department 20.