UNIT 1 RELATIONAL MODEL

Structure

1.0 INTRODUCTION

One of the main advantage of the relational model is that it is conceptually simple and more importantly based on mathematical theory of relation. It also frees the users from details of storage structure and access methods.

The relational model like all other models consists of three basic components:

- a set of domains and a set of relations
- operation on relations
- integrity rules

In this unit, we first provide the formal definition of a relational data model. Then we define basic operations of relational algebra and finally discuss the integrity rules.

1.1 OBJECTIVES

After completing this unit, you will be able to:

- define the concepts of relational model
- discuss the basic operations of the relational algebra
- state the integrity rules

1.2 CONCEPTS OF A RELATIONAL MODEL

The relational model was propounded by E.F. Codd of the IBM in 1972. The basic concept in the relational model is that of a relation.

A relation can be viewed as a table which has the following properties :

Property 1: it is column homogeneous. In other words, in any given column of a table, all items are of the same kind.

Property 2: each item is a simple number or a character string. That is, a table must be in 1NF. (First Normal Form) which will be introduced in the second unit.

Property 3: all rows of a table are distinct.

Property 4: the ordering of rows within a table is immaterial.

Property 5: the columns of a table are assigned distinct names and the ordering of these columns is immaterial.

Example of a valid relation

S#	P#	SCITY
10	1	BANGALORE
10	2	BANGALORE
11	1	BANGALORE
11	2	BANGALORE

1.3 FORMAL DEFINITION OF A RELATION

Formally, a relation is defined as the subset of the expanded cartesian product of domains. In order to do so, first we define the cartesian product of two sets and then the expanded cartesian product.

The cartesian product of two sets A and B, denoted by $A \times B$ is

$$A \times B = \{(a,b) : a \in A \text{ and } b \in B\}$$

The expanded cartesian product of n sets $A_1, A_2, \ldots, A_n$ is defined by

$$X(A_1, A_2, \ldots, A_n) = \{(a_1, a_2, \ldots a_n): a_j \in Aj \quad 1 <= j <= n)\}$$

The element $(a_1, a_2, \ldots, a_n)$ is called an n-tuple.

Given domains $D_1, D_2, \ldots D_n$ we define a relation, R, as a subset of the expanded cartesian product of these domains as follows:

$$R(D_1, D_2, \ldots D_n) \leq X(D_1, D_2, \ldots D_n)$$

In general we say that a relation defined over n domains has a degree n or is n-ary. The elements of this set are n-tuples.

We shall distinguish between the definition of a relation and the relation itself. We shall say that the definition of a relation gives a name to the relation and specifies the components over which it is defined. These components are referred to as relation attributes or attributes for short. An attribute has a domain associated with it from which it takes on values. The relation itself, on the other hand, is the set of tuples which constitute it at a given instance of time. For example, a statement which says that a relation Supplier is built over attributes S#, P#, SCITY having domains integer, character string respectively is the definition of the relation Supplier. The relation itself is shown below. It must be noted that at the time the definition of a relation is just given, a relation with no tuples in it, i.e. a null relation, is created.

Supplier

S#	P#	SCITY
10	1	BANGALORE
10	2	BANGALORE
10	3	BANGALORE
11	1	BOMBAY
11	2	BOMBAY

A **relational schema** is defined to be a collection of relation definitions.

We can now define the notion of a relational database or database for short. A database is a collection of relations of assorted degrees such that these relations are in accordance with their definitions in the relational schema. Since a relation is time varying, by this definition we can infer that a database is also time varying.

1.4 THE CODD COMMANDMENTS

In the most basic of definitions a DBMS can be regarded as relational only if it obeys the following three rules:

- All information must be held in tables
- Retrieval of the data must be possible using the following types of operations:

  SELECT, JOIN and PROJECT
- All relationships between data must be represented explicitly in that data itself.

This really is the minimum requirement, but it is surprising to see just how some well-known database products fail according to these simple rules to be in fact, relational, no matter what their vendors claim.

To define the requirements more rigorously, compliance with the 12 rules stated below must be demonstrable, within a single product, for it to be termed **relational**. In reality it's true to say that they don't all carry the same degree of importance, and indeed some very good products exist today supporting major large-scale production systems that cannot, hand on heart, claim to obey any more than eight or so of these rules. It's likely however, that it is only when all 12 rules can be satisfied, by facilities that coexist together, that the full benefits of the relational database can be realised.

The Twelve Rules

Just as in the 12 rules that define the distributed product, there is a single overall rule which in some ways covers all others and is commonly called Rule 0. It states that:

Any truly relational database must be manageable entirely through its own relational capabilities

Having stated this rule, we will not delve deeper except to say that its meaning can be interpreted by stating that a relational database must be **relational, wholly relational and nothing but relational**. If a DBMS depends on record-by-record data manipulation tools, it is not truly relational.

Rule 1: The information rule

All information is explicitly and logically represented in exactly one way — by data values in tables.

In simple terms this means that if an item of data doesn't reside somewhere in a table in the database then it doesn't exist and this should be extended to the point where even such information as table, view and column names to mention just a few, should be contained somewhere in table form. This necessitates the provision of an active data dictionary, that is itself relational, and it is the provision of such facilities that allow the relatively easy additions to RDBMS's of programming and CASE tools for example. This rule serves on its own to invalidate the claims of several databases to be relational simply because of their lack of ability to store dictionary items (or indeed **metadata**) in an integrated, relational form. Commonly such products implement their dictionary information systems in some native file structure, and thus set themselves up for failing at the first hurdle.

Rule 2 : The rule of guaranteed access

Every item of data must be logically addressable by resorting to a combination of table name, primary key value and column name.

Whilst it is possible to retrieve individual items of data in many different ways, especially in a relational/SQL environment, it must be true that any item can be retrieved by supplying the table name, the primary key value of the row holding the item and the column name in which it is to be found. If you think back to the table like storage structure, this rule is saying that at the intersection of a column and a row you will necessarily find one value of a data item (or null).

Rule 3 : The systematic treatment of null values

It may surprise you to see this subject on the list of properties, but it is fundamental to the DBMS that null values are supported in the representation of missing and inapplicable information. This support for null values must be consistent throughout the DBMS, and independent of data type (a null value in a CHAR field must mean the same as null in an INTEGER field for example).

It has often been the case in other product types, that a character to represent **missing or inapplicable data** has been allocated from the domain of characters pertinent to a particular

item. We may for example define four permissible values for a column SEX as:

M	Male
F	Female
X	No data available
Y	Not applicable

Such a solution requires careful design, and must decrease productivity at the very least. This situation is particularly undesirable when very high-level languages such as SQL are used to manipulate such data, and if such a solution is used for numeric columns all sorts of problems can arise during aggregate functions such as SUM and AVERAGE etc.

Rule 4 : The database description rule

A description of the database is held and maintained using the same logical structures used to define the data, thus allowing users with appropriate authority to query such information in the same ways and using the same languages as they would any other data in the database.

Put into easy terms, Rule 4 means that there must be a data dictionary within the RDBMS that is constructed of tables and/or views that can be examined using SQL. This rule states therefore that a dictionary is mandatory, and if taken in conjunction with Rule 1, there can be no doubt that the dictionary must also consist of combinations of tables and views.

Rule 5 : The comprehensive sub-language rule

There must be at least one language whose statements can be expressed as character strings conforming to some well defined syntax, that is comprehensive in supporting the following :

- Data definition
- View definition
- Data manipulation
- Integrity constraints
- Authorisation
- Transaction boundaries

Again in real terms, this means that the RDBMS must be completely manageable through its own dialect of SQL, although some products still support SQL-like languages (Ingress support of Quel for example). This rule also sets out to scope the functionality of SQL – you will detect an implicit requirement to support access control, integrity constraints and transaction management facilities for example.

Rule 6 : The view updating rule

All views that can be updated in theory, can also be updated by the system. This is quite a difficult rule to interpret, and so a word of explanation is required whilst it is possible to create views in all sorts of illogical ways, and with all sorts of aggregates and virtual columns, it is obviously not possible to update through some of them. As a very simple example, if you define a **virtual column** in a view as A*B where A and B are columns in a base table, then how can you perform an update on that virtual column directly? The database cannot possible break down any number supplied, into its two component parts, without more information being supplied. To delve a little deeper, we should consider that the possible complexity of a view is almost infinite in logical terms, simply because a view can be defined in terms of both tables and other views. Particular vendors restrict the complexity of their own implementations, in some cases quite drastically.

Even in logical terms it is often incredibly difficult to tell whether a view is theoretically updatable, let alone delve into the practicalities of actually doing so. In fact there exists another set of rules that, when applied to a view, can be used to determine its level of logical complexity, and it is only realistic to apply Rule 6 to those views that are defined as **simple** by such criteria.

Rule 7 : The insert and update rule

An RDBMS must do more than just be able to retrieve relational data sets. It has to be

capable of inserting, updating and deleting data as a relational set. Many RDBMSes that fail the grade fall back to a single-record-at-time procedural technique when it comes time to manipulate data.

Rule 8 : The physical independence rule

User access to the database, via monitors or application programs, must remain logically consistent whenever changes to the storage representation, or access methods to the data, are changed.

Therefore, and by way of an example, if an index is built or destroyed by the DBA on a table, any user should still retrieve the same data from that table, albeit a little more slowly. It is largely this rule that demands the clear distinction between the logical and physical layers of the database. Applications must be limited to interfacing with the logical layer to enable the enforcement of this rule, and it is this rule that sorts out the men from the boys in the relational market place. Looking at other architectures already discussed, one can imagine the consequences of changing the physical structure of a network or hierarchical system.

However there are plenty of traps awaiting even in the relational world. Consider the application designer who depends on the presence of a B-tree type index to ensure retrieval of data is in a predefined order, only to find that the DBA dynamically drops the index! What about the programmer who doesn't check for prime key uniqueness in his application, because he knows it is enforced by a unique index. The removal of such an index might be catastrophic. I point out these two issues because although they are serious factors, I am not convinced that they constitute the breaking of this rule; it is for the individual to make up his own mind.

Rule 9 : The logical data independence rule

Application programs must be independent of changes made to the base tables.

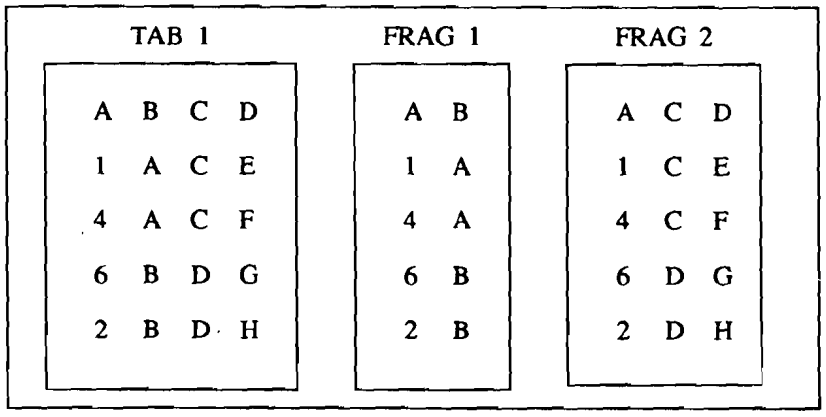

Figure 1 : TAB 1 Split into two fragments

This rule allows many types of database design change to be made dynamically, without users being aware of them. To illustrate the meaning of the rule the examples on the next page show two types of activity, described in more detail later, that should be possible if this rule is enforced.

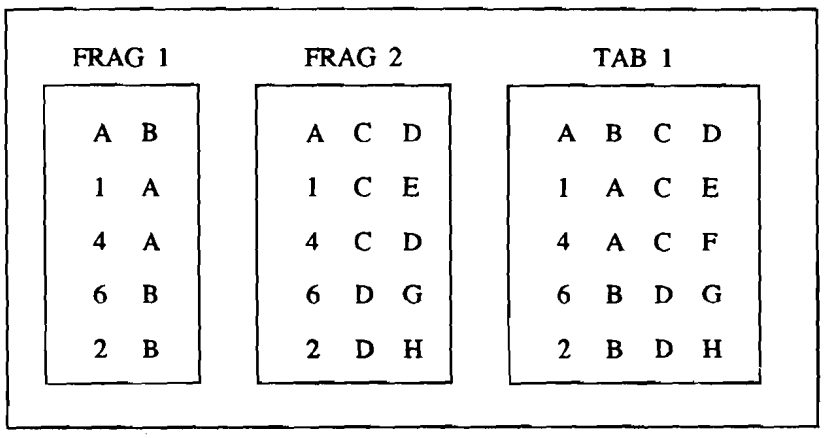

Figure 2 : Two fragments Combined into One Table

Firstly, it should be possible to split a table vertically into more than one **fragment**, as long as such splitting preserves all the original data (is **non-loss**), and maintain the primary key in each and every fragment. This means in simple terms that a single table should be divisible into one or more other tables.

Secondly it should be possible to combine base tables into one by way of **a non-loss join.**

Note that if such changes are made, then views will be required so that users and applications are unaffected by them.

Rule 10 : Integrity rules

The relational model includes two general integrity rules. These integrity rules implicitly or explicitly define the set of consistent database states, or changes of state, or both. Other integrity constraints can be specified, for example, in terms of dependencies during database design. In this section we define the integrity rules formulated by Codd.

Integrity Rule 1

Integrity rule 1 is concerned with primary key values. Before we formally state the rule, let us look at the effect of null values in prime attributes. A null value for an attribute is a value that is either not known at the time or does not apply to a given instance of the object. It may also be possible that a particular tuple does not have a value for an attribute; this fact could be represented by a null value.

If any attribute of a primary key (prime attribute) were permitted to have null values, then, because the attributes in the key must be nonredundant, the key cannot be used for unique identification of tuples. This contradicts the requirements for a primary key. Consider the relation P in figure 3. The attribute Id is the primary key for P. If null values (represented as @ were permitted, as in figure 3, then the two tuples @, Smith are indistinguishable, even though they may represent two different instances of the entity type employee. Similarly, the tuples < @, Lalonde > and 10^4, Lalonde >, for all intents and purposes, are also indistinguishable and may be referring to the same person. As instances of entities are distinguishable, so must be their surrogates in the model.

P:

Id	Name
101	Jones
103	Smith
104	Lalonde
107	Evan
110	Drew
112	Smith

(a)

P:

Id	Name
101	Jones
@	Smith
104	Lalonde
107	Evan
110	Drew
@	Lalonde
@	Smith

(b)

Figure 3 : (a) Relation without null values and (b) relation with null values

Integrity rule 1 specifies that instances of the entities are distinguishable and thus no prime attribute (component of a primary key) value may be null. This rule is also referred to as the entity rule. We could state this rule formally as:

Definition: Integrity Rule 1 (Entity Integrity):

If the attribute A of relation R is a prime attribute of R, then A cannot accept null values.

Integrity Rule 2 (Referential Integrity) :

Integrity rule 2 is concerned with foreign keys, i.e., with attributes of a relation having domains that are those of the primary key of another relation.

Relation (R), may contain references to another relation (S). Relations R and S need not be distinct. Suppose the reference in r is via a set of attributes that forms a primary key of the relation S. This set of attributes in R is a foreign key. A valid relationship between a tuple in R to one in S requires that the values of the attributes in the foreign key of R correspond to the primary key of a tuple in S. This ensures that the reference from a tuple of the relation R

is made unambiguously to an existing tuple in the S relation. The referencing attribute(s) in the R relation can have null value(s); in this case, it is not referencing any tuple in the S relation. However, if the value is not null, it must exist as the primary attribute of a tuple of the S relation. If the referencing attribute in R has a value that is nonexistent in S, R is attempting to refer a nonexistent tuple and hence a nonexistent instance of the corresponding entity. This cannot be allowed. We illustrate this point in the following example:

Example

Consider the example of employees and their has a manager and as managers are also employees, we may represent managers by their employee numbers, if the employee number is a key of the relation employee. Figure 4 illustrates an example of such an employee relation. The Manager attribute represents the employee number of the manager. Manager is a foreign key; note that it is referring to the primary key of the same relation. An employee can only have a manager who is also an employee. The chief executive officer (CEO) of the company can have himself or herself as the manager or may take null values. Some employees may also be temporarily without manager, and this can be represented by the Manager taking null values.

Emp#	Name	Manager
101	Jones	@
103	Smith	110
104	Lalonde	107
107	Evan	110
110	Drew	112
112	Smith	112

Figure 4 : Foreign Keys

Definition : Integrity Rule 2 (Referential Integrity)

Given two relations R and S, suppose R refers to the relation S via a set of attributes that forms the primary key of S and this set of attributes forms a foreign key in R. Then the value of the foreign key in a tuple in R must either be equal to the primary key of a tuple of S or be entirely null.

If we have the attribute A of relation R defined on domain D and the primary key of relation S also defined on domain D, then the values of A in tuples of R must be either null or equal to the value, let us say v, where v is the primary key value for a tuple in S. Note that R and S may be the same relation. The tuple in S is called the **target** of the foreign key. The primary key of the referenced relation and the attributes in the foreign key of the referencing relation could be composite.

Referential integrity is very important. Because the foreign key is used as a surrogate for another entity, the rule enforces the existence of a tuple for the relation corresponding to the instance of the referred entity. In example, we do not want a nonexisting employee to be manager. The integrity rule also implicitly defines the possible actions that could be taken whenever updates, insertions, and deletions are made.

If we delete a tuple that is a target of a foreign key reference, then three explicit possibilities exist to maintain database integrity:

- All tuples that contain references to the deleted tuple should also be deleted. This may cause, in turn, the deletion of other tuples. This option is referred to as a **domino** or **cascading deletion**, since one deletion leads to another.
- Only tuples that are not referenced by any other tuple can be deleted. A tuple referred by other tuples in the database cannot be deleted.
- The tuple is deleted. However, to avoid the domino effect, the pertinent foreign key attributes of all referencing tuples are set to null.

Similar actions are required when the primary key of a referenced relation is updated. An update of a primary key can be considered as a deletion followed by an insertion.

The choice of the option to use during a tuple deletion depends on the application. For example, in most cases it would be inappropriate to delete all employees under a given manager on the manager's departure; it would be more appropriate to replace it by null.

Another example is when a department is closed. If employees were assigned to departments, then the employee tuples would contain the department key too. Deletion of a department tuples should be disallowed until the employees have either been reassigned or their appropriate attribute values have been set to null. The insertion of a tuple with a foreign key reference or the update of the foreign key attributes of a relation require a check that the referenced relation exists.

Although the definition of the relational model specifies the two integrity rules, it is unfortunate that these concepts are not fully implemented in all commercial relational DBMSs. The concept of referential integrity enforcement would require an explicit statement as to what should be done when the primary key of a target tuple is updated or the target tuple is deleted.

Rule 11 : Distribution rule :

A RDBMS must have distribution independence.

This is one of the more attractive aspects of RDBMSes.Database system built on the relational framework are well suited to today's client/server database design.

Rule 12 : No subversion rule :

If an RDBMS supports a lower level language that permits for example, row-at-a-time processing, then this language must not be able to bypass any integrity rules or constraints of the relational language.

Thus, not only must a RDBMS be governed by relational rules, but those rules must be its primary laws.

The practical importance of these rules is difficult to estimate, and depends largely on the RDBMS in question, its proposed use and individual view points, but the theoretical importance is undeniable. It is interesting to see how some of the rules relate to others, and to some of the more important advantages of the relational model. It is unlikely at the present time that any RDBMS can claim full logical data independence because of their generally poor ability to handle updating through views. Even token adherence to this rule however, when combined with facilities enabling physical data independence, potentially yield advantages to applications developers, unheard of with any other type of database system. Coupling these two rules with the data independence and distribution independence rules can take the protection of customer investment to new heights.

The beauty of the relational database is that the concepts that define it are few, easy to understand and explicit. The 12 rules explained can be used as the basic relational design criteria, and as such are clear indications of the purity of the relational concept. Whilst you do not find these rules being quoted so often these days as in the recent past, it does not mean that they are any less important. Rather it can be interpreted as reflecting a reduced importance as propaganda. Other factors, of which performance is the most obvious, have now taken precedence.

1.5 RELATIONAL ALGEBRA

Relational algebra is a procedural language. It specifies the operations to be performed on existing relations to derive result relations. Furthermore, it defines the complete scheme for each of the result relations. The relational algebraic operations can be divided into basic set-oriented operations and relational- oriented operations. The former are the traditional set operations, the latter, those for performing joins, selection, projection, and division.

Basic Operations

Basic operations are the traditional set operations: union, difference, intersection and cartesian product. Three of these four basic operations – union, intersection, and difference – require that operand relations be **union compatible**. Two relations are union compatible if they have the same arity and one-to-one correspondence of the attributes with the corresponding attributes defined over the same domain. The cartesian product can be defined on any two relations. Two relations P and Q are said to be union compatible if both P and Q are of the same degree n and the domain of the corresponding n attributes are identical, i.e. if $P = P\{P_1, ... _n\}$ and $Q = \{Q_1, ... Q_n\}$ then

$$Dom(P_i) = Dom(Q_i) \text{ for } i = \{1,2, ...n\}$$

where $Dom(P_i)$ represents the domain of the attribute P_i.

Example 1

In the examples to follow, we utilise two relations P and Q given in Figure 5. R is a computed result relation. We assume that the relations P and Q in Figure 5 represent employees working on the development of software application packages J_1 and J_2 respectively.

P :

Id	Name
101	Jones
103	Smith
104	Lalonde
107	Even
110	Drew
112	Smith

Q :

Id	Name
103	Smith
104	Lalonde
106	Byron
110	Drew

Figure 5 : Union compatible relations

If we assume that P and Q are two union-compatible relations, then the union of P and Q is the set-theoretic union of P and Q. The resultant relation, $R = P \cup Q$, has tuples drawn from P and Q such that

$$R = \{t \mid \in P \vee t \in Q\}$$

The result relation R contains tuples that are in either P or Q or in both of them. The duplicate tuples are eliminated.

Remember that from our definition of union compatibility the degree of the relations P and R is the same. The cardinality of the resultant relation depends on the duplication of tuples in P and Q. From the above expression, we can see that if all the tuples in Q were contained in P, then $|R| = |P|$ and R = P, while if the tuples in P and Q were disjoint, then $|R| = |P| + |Q|$.

Example 2

R, the union of P and Q given in Figure 5 in the above example 1 is shown in Figure 6(a). R represents employees working on the packages J_1 or J_2, or both of these packages. Since a relation does not have duplicate tuples, an employee working on both J_1 and J_2 will appear in the relation R only once.

R :

Id	Name
101	Jones
103	Smith
104	Lalonde
107	Even
110	Drew
112	Smith

(a) $P \cup Q$

R :

Id	Name
101	Jones
107	Evan
112	Smith

(b) $P - Q$

R:

Id	Name
103	Smith
104	Lalonde
110	Draw

(c) $P \cap Q$

Figure 6 : Results of (a) union (b) difference and (c) intersection operations

Difference (–)

The difference operation removes common tuples from the first relation.

$R = P - Q$ such that

$R = \{t \mid t \in P \wedge t \notin Q\}$

Example 3

R, the result of P – Q, gives employees working only on package J_1. (figure 6(b) in example 2). Employees working on both packages J_1 and J_2 have been removed.

Intersection ($\cap$)

The intersection operation selects the common tuples from the two relations.

$$R = P \cap Q \text{ where}$$

$$R = \{t \mid t \in P \wedge t \in Q\}$$

Example 4

The resultant relation of $P \cap Q$ is the set of all employees working on both the packages. (figure 5(c) of example 2).

The intersection operation is really unnecessary. It can be very simply expressed as:

$$P \cap Q = P - (P - Q)$$

It is, however, more convenient to write an expression with a single intersection operation than one involving a pair of difference operations.

Note that in these examples the operand and the result relation schemes, including the attribute names, are identical i.e. P = Q = R. If the attribute names of compatible relations are not identical, the naming of the attributes of the result relation will have to be resolved.

Cartesian Product ($\times$)

The extended cartesian or simply the cartesian product of two relations is the concatenation of tuples belonging to the two relations. A new resultant relation scheme is created consisting of all possible combinations of the tuples.

$$R = P \times Q$$

where a tuple $r \in R$ is given by $\{t_1 \,||\, t_2 \mid t_1 \in P \wedge t_2 \in Q\}$, i.e. the result relation is obtained by concatenating each tuple in relation P with each tuple in relation Q. Here, represents the concatenation operation.

The scheme of the result relation is given by:

$$R = P \,||\, Q$$

The degree of the result relation is given by:

$$|R| = |P| + |Q|$$

The cardinality of the result relation is given by:

$$|R| = |P| * |Q|$$

Example 5

The cartesian product of the PERSONNEL relation and SOFTWARE_PACKAGE relations of figure 7(a) is shown in figure 7(b). Note that the relations P and Q from figure 5 of example 1 are a subset of the PERSONNEL relation.

PERSONNEL :

Id	Name
101	Jones
103	Smith
104	Lalonde
106	Byron
107	Evan
110	Drew
112	Smith

Software Packages :

S
J_1
J_2

7(a)

PERSONNEL :

Id	P.Name	S
101	Jones	J_1
101	Jones	J_2
103	Smith	J_1
103	Smith	J_2
104	Lalonde	J_1
104	Lalonde	J_2
106	Byron	J_1
106	Byron	J_2
107	Evan	J_1
107	Evan	J_2
110	Drew	J_1
110	Drew	J_2
112	Smith	J_1
112	Smith	J_2

(b)

Figure 7 : (a) PERSONNEL (Emp#, Name) and SOFTWARE_PACKAGE(S) represent employees and software packages respectively; (b) the Cartesian product of PERSONNEL and SOFTWARE_PACKAGES

The union and intersection operations are associative and commutative; therefore, given relations R, S, T:

$$R \cup (S \cup T) = (R \cup S) \cup T = (S \cup R) \cup T = T \cup (S \cup R) = \ldots \; R \cap (S \cap T) = (R \cap S) \cap T = \ldots$$

The difference operation, in general, is noncommutative and nonassociative.

$$R - S \neq S - R \qquad \text{noncommutative}$$

$$R - (S - T) \neq (R - S) - T \qquad \text{nonassociative}$$

Additional Relational Algebraic Operations

The basic set operations, which provide a very limited data manipulation facility, have been supplemented by the definition of the following operations: projection, selection, join, and division. These operations are represented by symbols π, σ, ∞ and + respectively. Projection and selection are unary operations; join and division are binary.

Projection (π)

The projection of a relation is defined as a projection of all its tuples over some set of attributes, i.e., it yields a **vertical subset** of the relation. The projection operation is used to either reduce the number of attributes in the resultant relation or to reorder attributes. In the first case, the arity (or degree) of the relation is reduced. The projection operation is shown graphically in figure 8. Figure 8 shows the projection of the relation PERSONNEL on the attribute Name. The cardinality of the result relation is also reduced due to the deletion of duplicate tuples.

We defined the projection of a tuple t_i over the attribute A, denoted $t_i[A]$ or $\pi_A(t_i)$, as (a), where a is the value

PERSONAL :

Id	Name
101	Jones
103	Smith
104	Lalonde
106	Byron
107	Evan
110	Drew
112	Smith

→

Name
Jones
Smith
Lalonde
Byron
Evan
Drew

Figure 8 : Projection of relation PERSONNEL over attribute Name

of tuple t_i over the attribute A. Similarly, we define the projection of a relation T, denoted by T[A] or $\pi_A(T)$, on the attribute A. This is defined in terms of the projection for each tuple in t_i belonging to T on the attribute A as:

$$T[A] = \{a_i \mid t_i[A] = a_i \wedge t_i \in T\}$$

where T[A] is a single attribute relation and $|T[A]| \leq T$. The cardinality T[A] may be less than the cardinality |T| because of the deletion of any duplicates in the result. A case in point is illustrated in figure 8.

Similarly, we can define the projection of a relation on a set of attribute names, X, as a concatenation of the projections for each attribute A in X for every tuple in the relation.

$$T[X] = \{ t_i[A] \mid t_i \in T\}$$

A belongs to X

where $t_i[A]$ represents the concatenation of all $t_i[A]$ for all $A \in X$.

A belongs to X

Simply stated, the projection of a relation P on the set of attribute names Y belong to P is the projection of each tuple of the relation P on the set of attribute names Y.

Note that the projection operation reduces the arity if the number of attributes in X is less than the arity of the relation. The projection operation may also reduce the cardinality of the result relation since duplicate tuples are removed. (Note that the projection operation produces a relation as the result. By definition, a relation cannot have duplicate tuples. In most commercial implementations of the relational model, however, the duplicates would still be present in the result).

Selection (σ)

Suppose we want to find those employees in the relation PERSONNEL of figure 7(a) of example 5 with an Id less than 105. This is an operation that selects only some of the tuples the relation. Such an operation is known as a **selection operation**. The projection operation yields a vertical subset of a relation. The action is defined over a subset of the attribute names but over all the tuples in the relation. The selection operation, however, yields a horizontal subset of a given relation, i.e., the action is defined over the complete set of attribute names but only a subset of the tuples are included in the result. To have a tuple included in the result relation, the specified selection conditions or predicates must be satisfied by it. The selection operation, is sometimes known as the **restriction operation.**

PERSONNEL :

Id	Name
101	Jones
103	Smith
104	Lalonde
106	Byron
107	Evan
110	Drew
112	Smith

Results of Selection

Id	Name
101	Jones
103	Smith
114	Lalonde

Figure 9 : Result of Selection over PERSONNEL for Id < 105.

Any finite number of predicates connected by Boolean operators may be specified in the selection operation. The predicates may define a comparison between two domain-compatible attributes or between an attribute and a constant value; if the comparison is between attribute A_1 and constant c_1, then c_1 belong to $Dom(A_1)$.

Given a relation P and a predicate expression B, the selections of those tuples of relation P that satisfy the predicate B is a relation R written as:

$$R = \sigma_B(P)$$

The above expression could be read as "select those tuples t from P in which the predicate B(t) is true." The set of tuples in relation R are in this case defined as follows:

$$R = \{t \mid t \in to P \wedge B(t)\}$$

JOIN (⋈)

The join operator, as the name suggests, allows the combining of two relations to form a single new relation. The tuples from the operand relations that participate in the operation and contribute to the result are related. The join operation allows the processing of relationships existing between the operand relations.

Example 6

In figure10we encounter the following relations: ASSIGNMENT (**Emp#, Prod#, Job#**)

JOB_FUNCTION (**Job#,** Title)

EMPLOYEE :

Emp#	Name	Profession
101	Jones	Analyst
103	Smith	Programmer
104	Lalonde	Receptionist
106	Byron	Receptionist
107	Evan	VPR & D
110	Drew	VP operations
112	Smith	Manager

PRODUCT :

Prod#	Prod-Name	Prod-Details
HEAP1	HEAP-SORT	ISS Module
BINS9	BINARY-SEARCH	ISS/R Module
FM6	FILE-MANAGER	ISS/R-PC Subsys
B++1	B++_TREE	ISS/R Turbo Sys
B++2	B++_TREE	ISS/R-PC Turbo

(a)

JOB-FUNCTION

Job#	Title
1000	CEO
900	President
800	Manager
700	Chief Programmer
600	Analyst

ASSIGNMENT

Emp#	Prod#	Job#
107	HEAP 1	800
101	HEAP 1	600
110	BINS9	800
103	HEAP1	700
101	BINS9	700
110	FM6	800
107	B++1	800

(b)

Figure 10 (a) Relation schemes for employee role in development teams (b) Sample relations

Suppose we want to respond to the query **Get product number of assignments whose development teams have a chief programmer.** This requires first computing the cartesian product of the ASSIGNMENT and JOB_FUNCTION relations. Let us name this product relation TEMP. This is followed by selecting those tuples of TEMP where the attribute Title has the value chief programmer and the value of the attribute Job# in ASSIGNMENT and JOB_FUNCTION are the same. The required result, shown below is obtained by projecting these tuples on the attribute Prod#. The operations are specified below.

TEMP = (ASSIGNMENT X JOB_FUNCTION)

$\pi_{Prod\#}$ (σ Title = 'chief programmer' $\wedge$ ASSIGNMENT.Job# (TEMP))

Prod #
HEAP 1
BINS 9

In another method of responding to this query, we can first select those tuples from the JOB_FUNCTION relation so that the value of the attribute Title is chief programmer. Let us call this set of tuples the relation TEMP1. We then compute the cartesian product of TEMP1 and ASSIGNMENT, calling the product TEMP2. This is followed by a projection on Prod# over TEMP2 to give us the required response. These operations are specified below:

TEMP1 = (σ Title = 'chief programmer'(JOB_FUNCTION))

TEMP2 = (σ ASSIGNMENT.Job# = JOB_FUNCTION.Job# (ASSIGNMENT X TEMP1))

$\pi_{Prod\#}$ (TEMP2) gives the required result.

Notice that in the selection operation that follows the cartesian product we take only those tuples where the value of the attributes ASSIGNMENT.Job# and JOB_FUNCTION.Job# are the same. These combined operations of cartesian product followed by selection are the join operation. Note that we have qualified the identically named attributes by the name of the corresponding relation to distinguish them.

In case of the join of a relation with itself, we would need to rename either the attributes of one of the copies of the relation or the relation name itself. We illustrate this in example 7.

In general the join condition may have more than one term, necessitating the use of the subscript in the comparison operator. Now we shall define the different types of join operations.

In these discussions we use P, Q, R and so on to represent both the relation scheme and the collection or bag of underlying domains of the attributes. We call it a bag of domains because more than one attribute may be defined on the same domain.

Typically, $P \cap Q$ may be null and this guarantees the uniqueness of attribute names in the result relation. When the same attribute name occurs in the two schemes we use qualified names.

Two common and very useful variants of the join are the **equi-join** and the **natural join**. In the equi-join the comparison operator $theta_i$(i = 1,2,....., n) is always the equality operator (=). Similarly, in the natural join the comparison operator is always the equality operator. However, only one of the two sets of domain compatible attributes involved in the natural join are A_i from P and B_i from Q, for i = 1, ..., n, the natural join predicate is a conjunction of terms of the following form:

$$(t_1[A_i] = t_2[B_i]) \text{ for } i = 1,2, ..., n$$

Domain compatibility requires that the domains of A_i and B_i be compatible, and for this reason relation schemes P and Q have attributes defined on common domains, i.e., $P \cap Q \neq \phi$. Therefore, join attributes have common domains in the relation schemes P and Q. Consequently, only one set of the join attributes on these common domains needs to be preserved in the result relation. This is achieved by taking a projection after the join operation, thereby eliminating the duplicate attributes. If the relation P and Q have attributes with the same domains but different attribute names, then renaming or projection may be specified.

Example 7

Given the EMPLOYEE and SALARY relations of figure 11(i), if we have to find the salary of employees by name, we join the tuples in the relation EMPLOYEE with those in SALARY such that the value of the attribute Id in EMPLOYEE is the same as that in SALARY. The natural join takes the predicate expression to be EMPLOYEE.Id = SALARY.Id. The result of the natural join is shown in figure 11(ii). When using the natural join, we do not need to specify this predicate. The expression to specify the operation of finding the salary of employees by name is given as follows. Here we project the result of the natural join operation on the attributes Name and Salary:

π (Name.Salary) (EMPLOYEE $\bowtie$ SALARY)

EMPLOYEE :

Id	Name
101	Jones
103	Smith
104	Lalonde
107	Evan

SALARY :

Id	Salary
101	67
103	55
104	75
107	80

EMPLOYEE ⋈ SALARY

Id	Name	Salary
101	Jones	67
103	Smith	55
104	Lalonde	75
107	Evan	80

ASSIGNMENT.Emp#	COASSIGN.Emp #
107	107
107	101
107	103
101	107
101	101
101	103
110	110
110	101
103	107
103	101
103	103
101	110

Figure 11 : (i) The natural join of EMPLOYEE and SALARY relations; (ii) The joint of ASSIGNMENT with the renamed copy

Division (÷)

Before we define the division operation, let us consider an example.

Example 8

Given the relations P and Q as shown in figure 12 (a), the result of dividing P by Q is the relation R and it has two tuples. For each tuple in R, its product with the tuples of Q must be in P. In our example (a_1,b_1) and (a_1,b_2) must both be tuples in P; the same is true for (a_5, b_1) and (a_5,b_2).

Simply stated, the cartesian product of Q and R is a subset of P. In figure 12 (b), the result relation R has four tuples; the cartesian product of R and Q gives a resulting relation which is

P :

A	B
a_1	b_1
a_1	b_2
a_2	b_1
a_3	b_1
a_4	b_2
a_5	b_1
a_5	b_2

Q :

B
b_1
b_2

R (result) :

A
a_1
a_5

(a)

Q :

B
b_1

then R is :

A
a_1
a_2
a_3
a_5

(b)

Q :

B
b_1
b_2
b_3

then R is

A

(c)

Q :

B

then R is :

A
a_1
a_2
a_3
a_4
a_5

(d)

Figure 12 : Examples of the division operation. (a) R = P ÷ Q; (b) R = P ÷ Q (P is the same as in part i); (c) R = P ÷ Q (P is the same as in part i); (d) R = P ÷ Q (P is the same as in part i)

again a subset of P. In figure 12 (c), since there are no tuples in P with a value b_3 for the attribute B (i.e., $selection_{B = b3}(P) = 0$), we have an empty relation R, which has a cardinality of zero.

In figure 12(d), the relation Q is empty. The result relation can be defined as the projection of P on the attributes in P – Q. However, it is usual to disallow division by a empty relation.

Finally, if relation P is an empty relation, then relation R is also an empty relation.

Let us treat the Q as representing one set of properties (the properties are defined on the Q, each tuple in Q representing an instance of these properties) and the relation r as representing entities with these properties (entities are defined on P – Q, and the properties are, as before, defined on Q); note that $P \cup Q$ must be equal to P. Each tuple in P represents an object with some given property. The resultant relation R, then, is the set of entities that possesses all the properties specified in Q. The two entities a_1 and a_5 possess all the properties, i.e., b_1 and b_2. The other entities in P, a_2, a_3, and a_4, only possess one, not both, of the properties. The division operation is useful when a query involves the phrase "for all objects having all the specified properties." Note that both P – Q and Q in general represent a set of attributes. It should be clear that Q not a subset of P.

1.6 RELATIONAL COMPLETENESS

The notion of relational completeness was propounded by Codd in 1972 as a basis for evaluating the power of different query languages.

A language is relationally complete if the basic relational algebra operations can be performed. The basic relational algebra operations are

- Union
- Difference
- Gross product
- Projection
- Selection

Query languages that are actually used in practice provide features in addition to the one mentioned above. For example, they provide facilities for

1. modification, storage and deletion of information
2. printing relations
3. assigning relations to some relation names
4. computing aggregate functions like SUM and MAX
5. performing arithmetic, for example, like retrieving the Salary + commission.

Check Your Progress

1. Define the following terms:
   (a) Intention of a set
   (b) Extension of a set
2. What is union compatible?

.....

.....

.....

.....

1.7 SUMMARY

The relational model has evoked a wide amount of interest in the database community. This model has a very sound mathematical basis to it. It exhibits a high degree of data independence.

However, it has its share of difficulties. These are:

- The relational model does not deal with issues like **semantic integrity, concurrency** and **database security**. These issues are left to be solved by the implementors of database management systems based on the relational model. The most serious consequence of the foregoing was the absence of the concept of semantic integrity in relational systems.
- Traditionally, implementations of the relational model have suffered from the drawback that they are relatively poor on response time. The biggest problem is in the realization of the **join operator**. Whereas, a DBMS based on the relational model can handle small databases, as the sizes of databases reach the region of billions of bytes the performance of these systems falls rather drastically. Consequently, these systems are able to support databases of relatively small sizes.

1.8 MODEL ANSWERS

1. (a) The **intention** of a set defines the permissible occurrences by specifying a membership condition.

   (b) The **extension** of the set specifies one of numerous possible occurrences by explicitly listing the set members. These two methods of defining a set are illustrated by the following example:

   Intention of set G = {g | g is an odd positive integer less than 10}

   Extension of set G = {1,3,5,7,9}

2. Two relations are union compatible if they have some arity and one to one correspondence of the attributes with the corresponding attributes defined over the same domain.

1.9 FURTHER READING

Bipin C.Desai, *An Introduction to Database System*, Galgotia Publication, New Delhi.