CHAPTER - 3

The Relational Model

3.1 Introduction

In 1970, Dr. Edgar F. Codd of IBM presented a paper entitled as "*A Relational Model of Data for Large Scaled Data Bank*". He gave certain principles of database management which is now called as *Relational model*. After that in 1974, Chamberlain and Boyce from IBM introduce the data language SEQUEL. First implementation of Codd's relational model was system R by IBM. At present there are many implementation of relational technology for both mainframe and microcomputers. IBM's DB2, Oracle, Sybase, MS-SQL server, MS-Access, Ingress etc. are some RDBMS.

Relation model was attempted to specify the database structure in term of matrix i.e. the database should contain tables. The table is in form of set of Columns and Rows. Tables in the database is known as relation and Columns in the table is called as attributes of an tables and rows in the table is called records or tuples. In the relational database model consists of set of tables having the unique name.

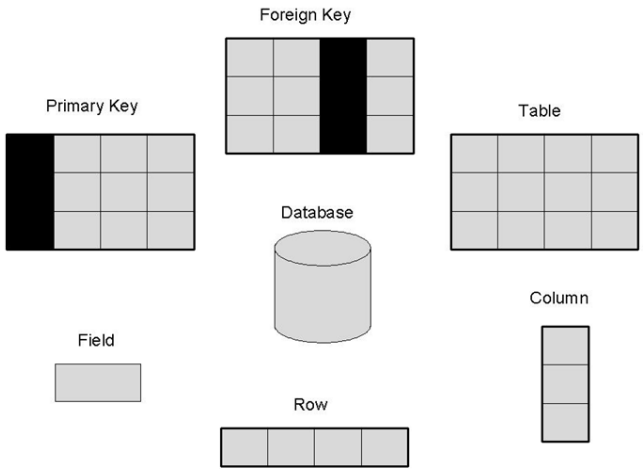
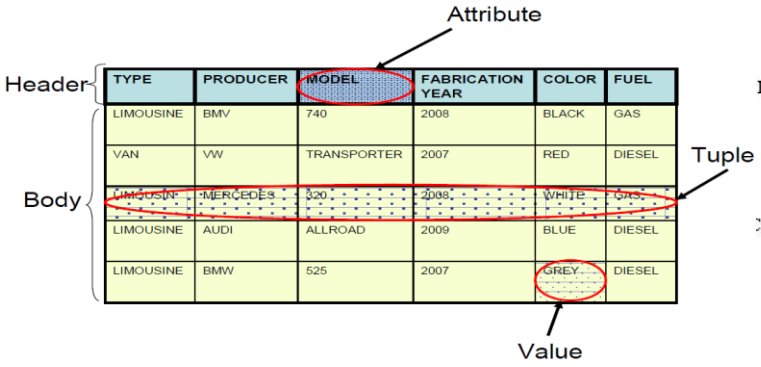
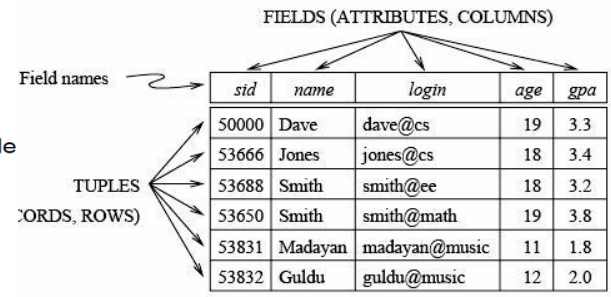


Figure. Relational database components

3.2. Relational model terminology

- **Relation:** A relation is a table. A *table* is a basic storage structure of an RDBMS and consists of columns and rows. A table represents an entity. Only applies to logical structure of the database, not the physical structure.





- **Attribute:** Attribute is a named column of a relation. A *column* is a collection of one type of data in a table. Each column has a column name and contains values that are bound by the same type and size.
- **Attribute domain:** Every attribute has some pre-defined value scope, known as attribute domain.
- **Tuple:** A single row of a table, which contains a single record for that relation is called a tuple. A *row* is a combination of column values in a table and is identified by a primary key. Rows are also known as records.
- **Field:** A field is an intersection of a row and a column. A field contains one data value. If there is no data in the field, the field is said to contain a NULL value.
- **Relation instance:** A finite set of tuples in the relational database system represents relation instance. Relation instances do not have duplicate tuples.
- **Relation schema:** This describes the relation name (table name), attributes and their names.

- **Relation key:** Each row has one or more attributes which can identify the row in the relation (table) uniquely, is called the relation key.
- **Degree:** Degree is a number of attributes in a relation.
- **Cardinality:** Cardinality is a number of tuples in a relation.
- **Relational Database:** Relational Database is a collection of normalized relations.
- **Constraints:** Every relation has some conditions that must hold for it to be a valid relation. These conditions are called *Relational Integrity Constraints*. There are three main integrity constraints.

  A. Key Constraints: There must be at least one minimal subset of attributes in the relation, which can identify a tuple uniquely. This minimal subset of attributes is called **key** for that relation. If there are more than one such minimal subsets, these are called ***candidate keys***. Key constrains are also referred to as Entity Constraints. Key constraints forces that:
  - In a relation with a key attribute, no two tuples can have identical value for key attributes.
  - Key attribute cannot have NULL values.

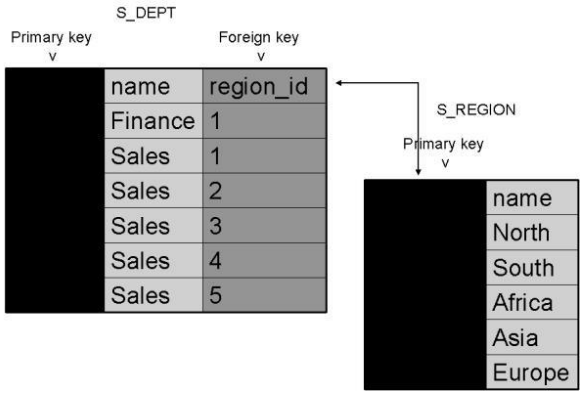
  B. Domain constraints: Attributes have specific values in real-world scenario. For example, age can only be positive integer. The same constraints has been tried to employ on the attributes of a relation. Every attribute is bound to have a specific range of values. For example, age cannot be less than zero and telephone number cannot be an outside 0-9.

  C. Referential integrity constraints: This integrity constraints works on the concept of Foreign Key. A key attribute of a relation can be referred in other relation, where it is called ***foreign key***. Referential integrity constraint states that if a relation refers to an key attribute of a different or same relation, that key element must exists.

A. Keys

An attribute or the set of attribute in the table that uniquely identifies each record in the entity set is called a key for that entity set. The following are the various types of keys available in the DBMS system. The various types of keys in RDBMS are:

- **Simple Key**: A key which has the single attribute is known as a simple key
- **Composite key**: A key which consist two or more attributes is called a Composite Key.
- **Super key:** A **super key** is any set of attributes that uniquely identifies a row. A super key differs from a candidate key in that it does not require the non-redundancy property.
- **Candidate Key:** A **candidate key** is an attribute (or set of attributes) that uniquely identifies a row. A candidate key must possess the properties of *unique identification* - For every row the value of the key must uniquely identify that row.
- **Primary key:** A primary key is a column or a combination of columns that is used to uniquely identify each row in a table. This attribute values must be "UNIQUE" and "NOT NULL" for all tuples from all relation instances. Every relation must contain a primary key.



- **Foreign Key:** A ***foreign key*** is an attribute or column (or attribute combination) in one relation R2 whose values are required to match those of the **primary key** of some relation R1 (R1 and R2 not necessarily distinct). Note that a foreign key and the corresponding **primary key** should be defined on the same underlying domain.

Example: In *Customer*, `c_id` is a primary key. In *Depositor*, `acc_no` is a foreign key referencing *Customer*.

Customer		
c_id	**c_name**	**c_city**
C01	X	A
C02	Y	B
C03	Z	A
C04	X	A

Account	
acc_no	**balance**
A1	200
A2	300
A3	500
A4	500

Depositor	
c_id	**acc_no**
C01	A1
C02	A2
C03	A3
C04	A4

Figure: Sample Relational database

B. Query Languages

Query language is a language through which user request information from database. Query languages can be categories in *procedural* and *non-procedural*. In procedural query language, user required to specify sequence of operations to system to compute desired information where as in nonprocedural query language, user need to specify required information without specifying special procedure for obtaining that information.

Most commercial relational database system offers query language, both procedural and non-procedural. SQL is most popular nonprocedural query language. The pure query language are: relational algebra, tuple relational calculus and domain relational calculus. These query languages cannot commercially use by people but it describes fundamental techniques for extracting data from database and provides basis for commercial query language.

3.2 The relational algebra

Relational algebras received little attention until the publication of E.F. Codd's relational model of data in 1970. Codd proposed such an algebra as a basis for database query languages. The first query language to be based on Codd's algebra was ISBL, and this pioneering work has been acclaimed by many authorities. Even the query language of SQL is loosely based on a 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. Each operator of the relational algebra takes either one or two relations as its input and produces a new relation as its output. ***Codd*** defined 8 such operators as given below:

Unary operations	**Binary operations**
▪ Projection (π) ▪ Selection (σ) ▪ Rename (ρ)	▪ JOIN (⋈) ▪ Division (÷) ▪ Union (∪) ▪ Difference (-) ▪ Intersection (∩) ▪ Cartesian Product (x)

Unary operations

A. Selection (σ)

This operator is used for selection of rows of whole records. The select operation selects a subset of tuples from a relation. It is a unary operator, that is, it applies on a single relation. The tuples subset must satisfy a selection condition or predicate. The formal notation for a select operation is:

$$\sigma_{\text{<select condition>}}\ (\text{<relation>})$$

Where <*select condition*> *is:* attribute, comparison operator, constant value, AND/OR/NOT operation etc. The comparison operator can be **<, >, <=, >=, =, <>** and it depends on attribute domain or data type constant value.

The resulting relation degree is equal with the initial relation degree on which the operator is applied. The resulting relation cardinality is less or equal with the initial relation cardinality. If the cardinality is low we say that the select condition selectivity is high and if the cardinality is high we say that the select condition selectivity is low.

R

Name	Age	Sex
A	20	M
M	21	F
B	20	F
F	19	M
A	20	F
R	21	F
C	21	M

R1= σ(Age=20)(R)

Name	Age	Sex
A	20	M
B	20	F
A	20	F

R2= σ(Sex=M AND Age>19)(R)

Name	Age	Sex
A	20	M
C	21	M

Fig. Example of a SELECT

Relational algebra	**Output or Meaning**
$\sigma_{subject="database"}(Books)$	Selects tuples from books where subject is 'database'.
$\sigma_{subject="database" \text{ and } price="450"}(Books)$	Selects tuples from books where subject is 'database' and 'price' is 450.
$\sigma_{subject="database" \text{ and } price < "450" \text{ or } year > "2010"}(Books)$	Selects tuples from books where subject is 'database' and 'price' is 450 or the publication year is greater than 2010, that is published after 2010.
NOTE:	
Relational algebra	**Equivalent SQL Statement**
$\sigma_{Dno=4 \textbf{ AND } Salary>25000}(EMPLOYEE)$	**SELECT** * **FROM** EMPLOYEE **WHERE** Dno=4 **AND** Salary>25000;

Notice that the SELECT operation is **commutative**; that is,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond2>}(\sigma_{<cond1>}(R))$$

B. Projection (π)

The project operator select the unique column of a relation and builds another relation. Duplicate tuples from the resulting relation are eliminated. It is also a unary operator. Projection is not commutative. The formal notation for a project operation is:

$$\pi_{<attribute\ list>}(<relation>)$$

Where <attribute list> is the subset attributes of an existing relation.

The resulting relation degree is equal with the number of attributes from <attribute list> because only those attributes will appear in the resulting relation. The resulting relation cardinality is less or equal with the initial relation cardinality. If the list of attributes contains a relation candidate key, then the cardinality is equal with the initial relation cardinality. If it does not contain a candidate key, then the cardinality could be less because of the possibility to have duplicate tuples, which are eliminated from the resulting relation.

R

Name	Age	Sex
A	20	M
M	21	F
B	20	F
F	19	M
A	20	F

Figure – Example of a PROJECT operation

R1= π(Name, Sex)(R)

Name	Sex
A	M
M	F
B	F
F	M
A	F

R2= π(Age, Sex)(R)

Age	Sex
20	M
21	F
20	F
19	M

For example: Selects and projects columns named as subject and author from relation Books.

$$\Pi_{subject,\ author}(Books)$$

NOTE:	
Relational algebra	**Equivalent SQL Statement**
$\Pi_{Sex, Salary}$(EMPLOYEE)	**SELECT DISTINCT** Sex, Salary **FROM** EMPLOYEE

C. Rename Operator (ρ)

Rename operation gave alternate name to the given column or to any table by using the operator called Rename operator This operator is used for selecting some specific column from multiple table(set of two or more tables) containing multiple columns having same column name Rename operator is denoted by the greek letter rho(ρ).

Syntax: ρ<New Name for Column>(<Input_Table_Name>)

Example: Rename Name in the relation Student to SName

ρ Student(Rollno, SName, Address)(Student)

Binary operation

A. Join (⋈)

The join operation concatenates two relations based on a joining condition or predicate. The relations must have at least one common attribute with the same underlying domain, and on such attributes a joining condition can be specified. The formal notation for a join operation is:

R <join condition> ⋈ S

Where R and S are relations and the <join condition> is: attribute from R, comparison operator, and attribute from S. The comparison operator can be **<, >, <=, >=, =, <>** and it depends on attributes domain. The ***equijoin*** has comparison operator is =.

Natural Joins

Natural join includes two identical attributes of same names from two table. Assume R1 and R2 have attributes A is common. Natural join is formed by concatenating all tuples from R1 and R2 with same values for A, and dropping the occurrences of A in R2:

Course

CourseId	Title	Eid
CS51T	DBMS	123
CS52S	OS	345
CS52T	TOC	345
CS51S	SE	456

Course ⋈ Instructor

CourseId	Title	Eid	Ename
CS51T	DBMS	123	Ram
CS52S	OS	345	Anil
CS52T	TOC	345	Anil
CS51S	SE	456	Hari

Instructor

Eid	Ename
123	Ram
345	Anil
456	Hari

$\Pi_{CourseId, ename}$Course ⋈ Instructor

CourseId	Ename
CS51T	Ram
CS52S	Anil
CS52T	Anil
CS51S	Hari

Natural Joins Example

$R1 \bowtie R2 = \Pi_{A'}(\sigma_C(R1 \times R2))$

where C is the condition that the values for R1 and R2 are the same for all attributes in A and A' is all attributes in R1 and R2 apart from the occurrences of A in R2.

Inner Join

In Inner join, tables are joined together where there is the match (=) of the primary and foreign keys.

$R \bowtie_{<R.primary\_key = S.foreign\_key>} S$

Inner joins return rows only when there is at least one row from both tables that matches the join condition. Inner joins eliminate the rows that do not match with a row from the other table.

R

ColA	*ColB*
A	1
B	2
D	3
F	4
E	5

R JOIN R.ColA = S.SColA S

A	1	A	1
D	3	D	3
E	5	E	4

S

SColA	*SColB*
A	1
C	2
D	3
E	4

R JOIN R.ColB = S.SColB S

A	1	A	1
B	2	C	2
D	3	D	3
F	4	E	4

Student

StudentId	Name	Course
100	Ram	PH
200	Sunil	CM
300	Anil	CM

Courses

Course#	name
PH	Pharmacy
CM	Computing

Students ⋈ course = Course# Courses

StudentId	Name	Course	Course#	Courses.name
100	Ram	PH	PH	Pharmacy
200	Sunil	CM	CM	Computing
300	Anil	CM	CM	Computing

Outer Join

Inner join eliminates the rows that do not match the conditions. In some cases this lost data might hold useful information. An outer join retains the information that would have been lost from the tables, replacing missing data with nulls. So the outer join =Inner join + rows of one table which do not satisfy the condition. There are three forms of the outer join, depending on which data is to be kept.

- LEFT OUTER JOIN - keep data from the left-hand table
- RIGHT OUTER JOIN - keep data from the right-hand table
- FULL OUTER JOIN - keep data from both tables

R

ColA	*ColB*
A	1
B	2
D	3
F	4
E	5

R LEFT OUTER JOIN R.ColA = S.SColA S

A	1	A	1
D	3	D	3
E	5	E	4
B	2	-	-
F	4	-	-

R FULL OUTER JOIN R.ColA = S.SColA S

A	1	A	1
D	3	D	3
E	5	E	4
B	2	-	-
F	4	-	-
-	-	C	2

S

SColA	*SColB*
A	1
C	2
D	3
E	4

R RIGHT OUTER JOIN R.ColA = S.SColA S

A	1	A	1
D	3	D	3
E	5	E	4
-	-	C	2

Example:

R1

First Name	Last Name
A	Mary
B	John
C	Ann

R2

Last Name	Sex
Ann	F
John	M
Mary	F
Bill	M

Natural Join

First Name	Last Name	Sex
A	Mary	F
B	John	M
C	Ann	F

R3=R1(Last Name=Last name) R2

First Name	Last Name	Last Name	Sex
A	Mary	Mary	F
B	John	John	M
C	Ann	Ann	F

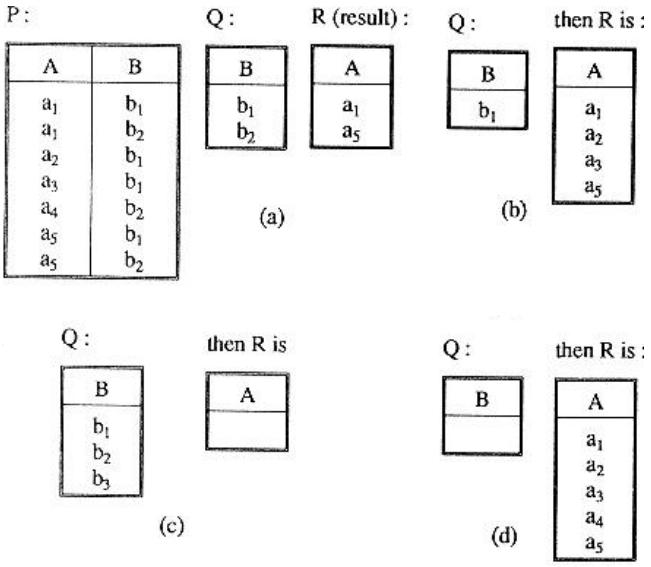
Right Outer Join

First Name	Last Name	Last Name	Sex
A	Mary	Mary	F
B	John	John	M
C	Ann	Ann	F
NULL	NULL	Bill	M

Figure – Example of a JOIN operation

B. Division (÷)

The ***division*** operator divides a relation **R1** of degree **(n+m)** by a relation **R2** of degree **m** and produces a relation of degree **n**. The **(n+i)**th attribute of **R1** and the **i**th attribute from **R2** should be defined on the same domain. The result of a division operation between **R1** and **R2** is another relation, which contains all the tuples that concatenated with all **R2** tuples are belonging to **R1** relation. The formal notation for a division operation is ÷.



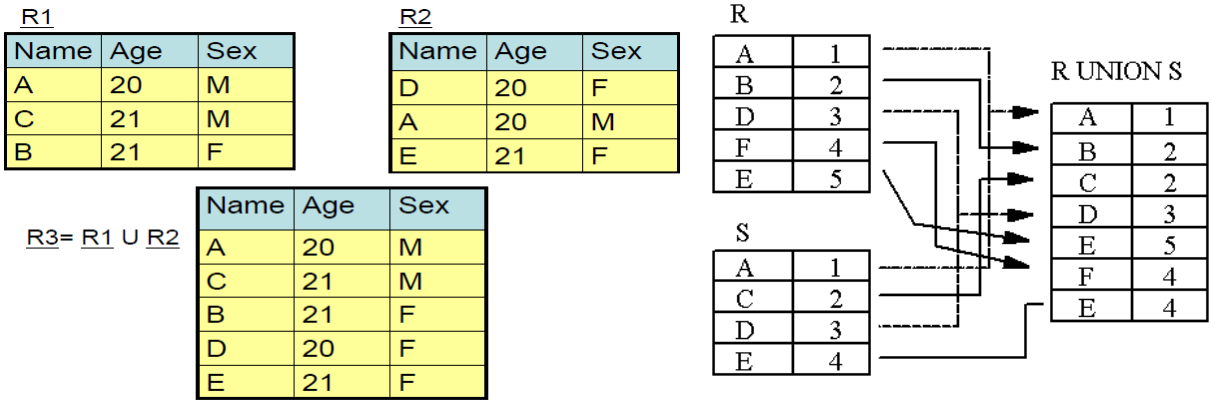
C. Union (∪)

The ***union*** of two union-compatible relations **R1** and **R2**, **R1 UNION R2**, defined as: **R1**∪ **R2** = {t | t ∈ **R1** or t ∈ **R2**}

For a union operation to be valid, the following conditions must hold:

- R1, R2 must have same number of attributes.
- Attribute domains must be compatible.

The formal notation for a union operation is U. **UNION** operation is associative and commutative. Duplicate tuples are automatically eliminated.



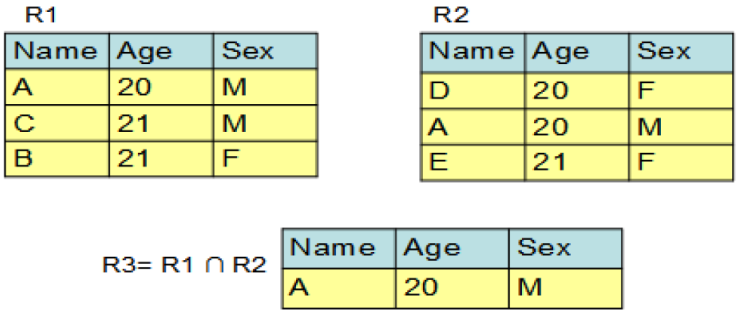
Fig– Example of a UNION operation on two relations: R1 and R2

For example: Projects the name of author who has either written a book or an article or both.

```
Π author (Books) ∪ Π author (Articles)
```

D. Intersection (∩)

The intersection of two intersection-compatible relations **R1** and **R2**, **R1 INTERSECT R2**, is the set of all tuples **t** belonging to both **R1** and **R2**. The formal notation for an intersect operation is ∩. **INTERSECT** operation is associative and commutative.



Fig– Example of an INTERSECT operation

E. Difference (-)

The ***difference*** between two union-compatible relations **R1** and **R2**, **R1 MINUS R2**, is the set of all tuples **t** belonging to **R1** and not to **R2**. Thus, the result of set difference operation is tuples which present in one relation but are not in the second relation. The formal notation for a difference operation is -. **DIFFERENCE** operation is not associative and commutative.

R1

Name	Age	Sex
A	20	M
C	21	M
B	21	F

R2

Name	Age	Sex
D	20	F
A	20	M
E	21	F

R3= R1 - R2

Name	Age	Sex
C	21	M
B	21	F

R3= R2 – R1

Name	Age	Sex
D	20	F
E	21	F

Fig– Example of a DIFFERENCE operation

For example: Results the name of authors who has written books but not articles.

$\Pi_{\text{author}}(\text{Books}) - \Pi_{\text{author}}(\text{Articles})$

F. Cartesian product (X)

The Cartesian product combines information of two different relations into one. The Cartesian product between two relations **R1** and **R2**, **R1 TIMES R2**, is the set of all tuples **t** such that **t** is the concatenation of a tuple **r** belonging to **R1** and a tuple **s** belonging to **R2**. The concatenation of a tuple $\mathbf{r = (r_1, r_2, \ldots, r_m)}$ and a tuple $\mathbf{s = (s_{m+1}, s_{m+2}, \ldots, s_{m+n})}$ is the tuple t = $\mathbf{(r_1, r_2, \ldots, r_m, s_{m+1}, s_{m+2}, \ldots, s_{m+n})}$. **R1** and **R2** don't have to be union-compatible. The formal notation for a Cartesian product operation is ×.

If **R1** has degree **n** and cardinality **N1** and **R2** has degree **m** and cardinality **N2** then the resulting relation **R3** has degree **(n+m)** and cardinality **(N1*N2)**, as shown in figure.

R1

Name	Age	Sex
A	20	M
C	21	M

R2

Name	Age	Sex
D	20	F
E	21	F

R3= R1 X R2

Name	Age	Sex	Name	Age	Sex
A	20	M	D	20	F
C	21	M	D	20	F
A	20	M	E	21	F
C	21	M	E	21	F

Fig- Example of a CARTESIAN PRODUCT operation

Example: yields a relation as result which shows all books and articles written by William.

$\Pi_{\text{author = 'William'}}(\text{Books X Articles})$

Modification of the Database

A. Delete operation

This operation is used to delete the selected tuples from the relations. We cannot use such operation to delete a value on only particular attributes. In relational algebra a deletion is expressed by $r \leftarrow r - E$ where r is a relation and E is a relational-algebra query.

Example: Delete all the record from instructor whose number is 3412

$\text{Instructor} \leftarrow \text{Instructor} - \sigma_{\text{number=3412}}(\text{Instructor})$

Example: Delete all the records from books whose docId>100 and publication date before 1990

$\text{Books} \leftarrow \text{Books} - \sigma_{\text{DocId>100 ^ Year<1990}}(\text{Books})$

B. Insertion

This operation is used to insert a single tuple or set of tuples in a relation. The relational algebra expresses an insertion by $r \leftarrow r \cup E$ where r is a relation and E is a relational-algebra expression.

Example: Insert a single tuple in a relation books

Books ← Books ∪ {(12, "DBMS", "MC-Graw Hill", 2012)}

C. Update

In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple.

For example: Account (account_number, branch_Name, Balance)

- Update the balance by 6% whose balance is greater than 100000
- Update the balance by 6 % whose balance is greater than 10000 and by 5% whose balance is less than 10000

Solution:

- Account ← Π accno,branchname,balance*1.06 (σbalance > 100000(Account))
- *Account* ← Παccno, *branchname*, *balance* *1.06 (σ*balance* > 10000 (*Account*)) ∪ Παccon, *branchname*, *balance**1.05 (σ*balance*≤10000 (A*ccount*))

D. Aggregation Operators

Operators that summarize or aggregate the values in a single attribute of a relation. Operators are the same in relational algebra and SQL. Some of the important aggregation operators are:

- **SUM:** computes the sum of a column with numerical values.
- **AVG:** computes the average of a column with numerical values.
- **MIN and MAX:** for a column with numerical values, computes the smallest or largest value, respectively. For a column with string or character values, computes the lexicographically smallest or largest values, respectively.
- **COUNT:** computes the number of non-NULL tuples in a column.

For example: ΠMax(Salary), MIN(Salary), SUM(salary), AVG(Salary)(Employee) ΠCOUNT(Salary) (σsalary>1000)

Example Queries:

Assume the following relations:

- BOOKS(DocId, Title, Publisher, Year)
- STUDENTS(StId, StName, Major, Age)
- AUTHORS(AName, Address)
- borrows(DocId, StId, Date)
- has-written(DocId, AName)
- describes(DocId, Keyword)

Statements	***Relational algebra***
List the year and title of each book.	Π Year, Title(BOOKS)
List all information about students whose major is CS.	σ Major = 'CS'(STUDENTS)
List all students with the books they can borrow.	STUDENTS X BOOKS
List all books published by McGraw-Hill before 1990.	σ Publisher = 'McGrawHill'^Year<1990(BOOKS)

List the name of those authors who are living in Davis.	ΠAName(σ Address like '%Davis%' (AUTHORS))
List the name of students who are older than 30 and who are not studying CS.	ΠStName(σ Age>30(STUDENTS)) – ΠStName(σ Major='CS'(STUDENTS))
Rename AName in the relation AUTHORS to Name.	ρAUTHORS(Name, Address)(AUTHORS)

3.3 Schema and Views

Schema

In a relational database, the schema defines the tables, the fields in each table, and the relationships between fields and tables. Schemas are generally stored in a *data dictionary* or *catalog*. Although a schema is defined in text database language, the term is often used to refer to a graphical depiction of the database structure. A relation schema is a list of attributes and their corresponding domains. Values or data contain in relation change when it is updated. Relation instance is a snapshot of data in relation at particular time.

Syntax :

CREATE SCHEMA: Creates a schema in the current database.	CREATE SCHEMA schema_name_clause [<schema_element>[...n]] Where, <schema_name_clause> ::= { schema_name | AUTHORIZATION owner_name | schema_name AUTHORIZATION owner_name} <schema_element> ::= { table_definition | view_definition | grant_statement | revoke_statement | deny_statement } ▪ **Note:** The contents inside the square bracket is optional
For example: CREATE SCHEMA Sprockets AUTHORIZATION Krishna CREATE TABLE NineProngs (source int, cost int, partnumber int) GRANT SELECT TO Anibal DENY SELECT TO Hung-Fu;	

ALTER SCHEMA: Transfers a securable between schemas.	ALTER SCHEMA schema_name TRANSFER [<entity_type> ::] securable_name[;]
	Where
	<entity_type> ::=
	{
	Object | Type | XML Schema Collection
	}

For example: Modifies the schema HumanResources by transferring the table Address from schema Person into the schema. ALTER SCHEMA HumanResources TRANSFER Person.Address;	
DROP SCHEMA: Removes a schema from the database.	DROP SCHEMA schema_name
For example: DROP TABLE Sprockets.NineProngs; DROP SCHEMA Sprockets;	

Schema and Schema Diagram

Schema diagram is a graphical representation of database schema along with primary key and foreign key dependencies. In schema diagram, each relation is represented by box where attributes are listed inside box and relation name is specified above it. Primary key in relation is place above the horizontal line that crosses the box. Foreign key in schema diagram appear as arrow from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

For **example:** The relation schema for relation *customer* is express as

```
Customer-schema = (customer_id, customer_name, customer_city)
```

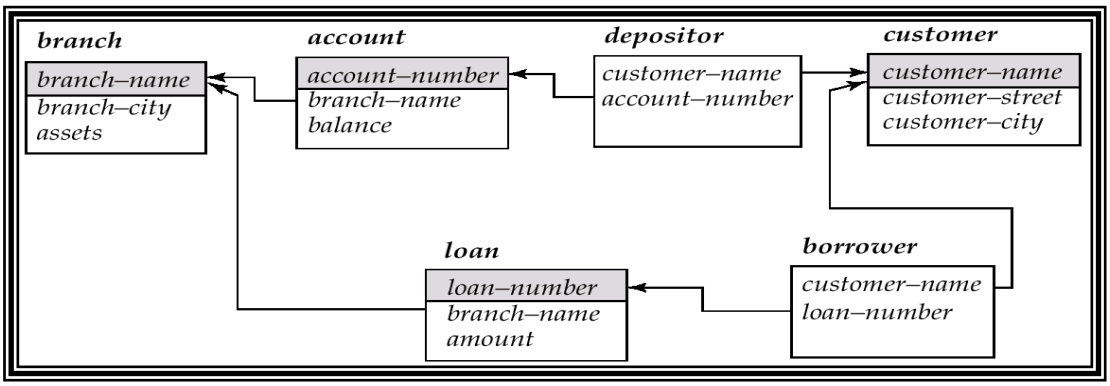
We may also specify domains of attributes as

```
Customer-schema = (customer_id: integer, customer_name: string, customer_city:string)
```

For **example:** Database schemas for banking enterprise

- `Branch-schema = (branch_name, branch_city, assets)`
- `Account-schema = (account_number, branch_name, balance)`
- `Customer-schema = (customer_name, customer_street, customer_city)`
- `Depositor-schema = (customer_name, account_number)`
- `Loan-schema = (loan_number, branch_name, amount)`
- `Borrower-schema = (customer_name, loan_number)`

Now the corresponding schema diagram is given as:



Note: The difference between E-R diagram and schema diagram is, E-R diagram do not shows the foreign key but schema diagram shows it explicitly.

View

A view is a virtual table that consists of columns from one or more tables. In other word, A view is a virtual table derived from one or more tables or other views. It is virtual because it does not contain any data, but a definition of a table based on the result of a SELECT statement. Views allow you to hide data or limit access to a select number of columns; therefore, they can also be used for security purposes. A view can be used for the following purposes:

- Provides user security functions.
- Simplifies the constructions of complex queries.
- Summarize data from multiple tables.

Syntax:

CREATE VIEW: Creates a virtual table whose contents (columns and rows) are defined by a query.	CREATE VIEW [schema_name..] view_name [(column [..n])] [WITH <view_attribute>[..n]] AS select_statement [WITH CHECK OPTION] [;] Where <view_attribute> ::= { [ENCRYPTION] [SCHEMABINDING] [VIEW_METADATA] } ▪ **Note:** The contents inside the square bracket is optional ▪ ENCRYPTION prevents the view from being published as part of SQL Server replication. ▪ The specification of SCHEMABINDING ensure no modification of the base table or tables in a way that would affect the view definition. ▪ For views created with VIEW_METADATA, the browse-mode metadata returns the view name and not the base table names when it describes columns from the view in the result set.
For example: Create a view based on the EMPLOYEE table CREATE VIEW MYVIEW AS SELECT LASTNAME, HIREDATE FROM EMPLOYEE	
For example: The following example creates a view that contains all employees and their hire dates called EmployeeHireDate. Permissions are granted to the view, but requirements are changed to select employees whose hire dates fall before a certain date. CREATE VIEW HumanResources.EmployeeHireDate AS SELECT p.FirstName, p.LastName, e.HireDate FROM HumanResources.Employee AS e JOIN Person.Person AS p ON e.BusinessEntityID = p.BusinessEntityID ;	
ALTER VIEW: Modifies a previously created view. This includes an indexed view.	ALTER VIEW [schema_name .] view_name [(column [..n])][WITH <view_attribute> [..n]] AS select_statement [WITH CHECK OPTION][;] Where, <view_attribute> ::= { [ENCRYPTION] [SCHEMABINDING] [VIEW_METADATA] }
For example: The view must be changed to include only the employees that were hired before 2002. If ALTER VIEW is not used, but instead the view is dropped and re-created, the previously used GRANT statement and any other statements that deal with permissions pertaining to this view must be re-entered. ALTER VIEW HumanResources.EmployeeHireDate AS SELECT p.FirstName, p.LastName, e.HireDate FROM HumanResources.Employee AS e JOIN Person.Person AS p ON e.BusinessEntityID = p.BusinessEntityID WHERE HireDate < CONVERT(DATETIME,'20020101',101) ;	
DROP VIEW: Removes one or more views from the current database. DROP VIEW can be executed against indexed views.	DROP VIEW [schema_name .] view_name [...,n] [;]

Example: creating view from single table.

```
CREATE VIEW   vw_emp   AS
  SELECT empno,  ename, job FROM emp;
```

Example: Creating view from multiple tables.

```
CREATE VIEW vw_emp_info AS
    SELECT e.empno, e.ename, e.ejob, d.dname
         FROM emp e, dept d
    WHERE e.empno = d.dept no AND  e.sal>1000;
```

3.4 Data Dictionary (or **information repositories**)

A *data dictionary* or *metadata repository* is a "centralized repository of information about data such as meaning, relationships to other data, origin, usage, and format. In database management systems, a file that defines the basic organization of a database. A data dictionary contains a list of all files in the database, the number of records in each file, and the names and types of each field. Most database management systems keep the data dictionary hidden from users to prevent them from accidentally destroying its contents. Data dictionaries do not contain any actual data from the database, only bookkeeping information for managing it. Without a data dictionary, however, a database management system cannot access data from the database. A useful data dictionary system should store and manage the following types of information:

a. Descriptions of the schemas of the database system.
b. Detailed information on physical database design, such as storage structures, access paths, and file, record sizes, and default value for column.
c. Descriptions of the types of database users, their responsibilities, and their access rights.
d. High-level descriptions of the database transactions and applications and of the relationships of users to transactions. This is called *Integrity constraint* information
e. The relationship between database transactions and the data items referenced by them. This is useful in determining which transactions are affected when certain data definitions are changed.
f. Usage statistics such as frequencies of queries and transactions and access counts to different portions of the database.
g. The history of any changes made to the database and applications, and documentation that describes the reasons for these changes. This is sometimes referred to as **data provenance**.

The terms *data dictionary* and *data repository* indicate a more general software utility than a catalogue. A *catalogue* is closely coupled with the DBMS software. It provides the information stored in it to the user and the DBA, but it is mainly accessed by the various software modules of the DBMS itself, such as DDL and DML compilers, the query optimizer, the transaction processor, report generators, and the constraint enforcer. On the other hand, a *data dictionary* is a data structure that stores metadata, i.e., (structured) data about data. The software package for a stand-alone data dictionary or data repository may interact with the software modules of the DBMS, but it is mainly used by the designers, users and administrators of a computer system for information resource management. These systems maintain information on system hardware and software configuration, documentation, application and users as well as other information relevant to system administration.

If a data dictionary system is used only by the designers, users, and administrators and not by the DBMS Software, it is called a *passive data dictionary.* Otherwise, it is called an *active data dictionary* or *data dictionary.* When a passive data dictionary is updated, it is done so manually and independently from any changes to a DBMS (database) structure. With an active data dictionary, the dictionary is updated first and changes occur in the DBMS automatically as a result.