

Chapter-3

Relational Database Model

Asst. Prof. Roshan Tandukar



RDBMS



- **RDBMS** stands for Relational Database Management Systems.
- It is a collection of programs that provides a way to create, update, administer and otherwise interact with a relational database.
- During 1970 to 1972, E.F. Codd published a paper to propose the use of relational database model.
- All modern database management systems like MS SQL Server, IBM DB2, ORACLE, My-SQL and Microsoft Access are based on RDBMS.

RDBMS

- The RDBMS database uses tables to store data.
- A table is a collection of related data entries and contains rows and columns to store data.
 - Rows/Tuples/Records
 - Columns/Attributes/field

<i>customer_id</i>	<i>customer_name</i>	<i>customer_street</i>	<i>customer_city</i>	<i>account_number</i>
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-101
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-201
677-89-9011	Hayes	3 Main St.	Harrison	A-102
182-73-6091	Turner	123 Putnam St.	Stamford	A-305
321-12-3123	Jones	100 Main St.	Harrison	A-217
336-66-9999	Lindsay	175 Park Ave.	Pittsfield	A-222
019-28-3746	Smith	72 North St.	Rye	A-201

Tabular Data in a relational model

Relational DBMS

<i>customer-id</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
192-83-7465	Johnson	12 Alma St.	Palo Alto
019-28-3746	Smith	4 North St.	Rye
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

<i>account-number</i>	<i>balance</i>
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

<i>customer-id</i>	<i>account-number</i>
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

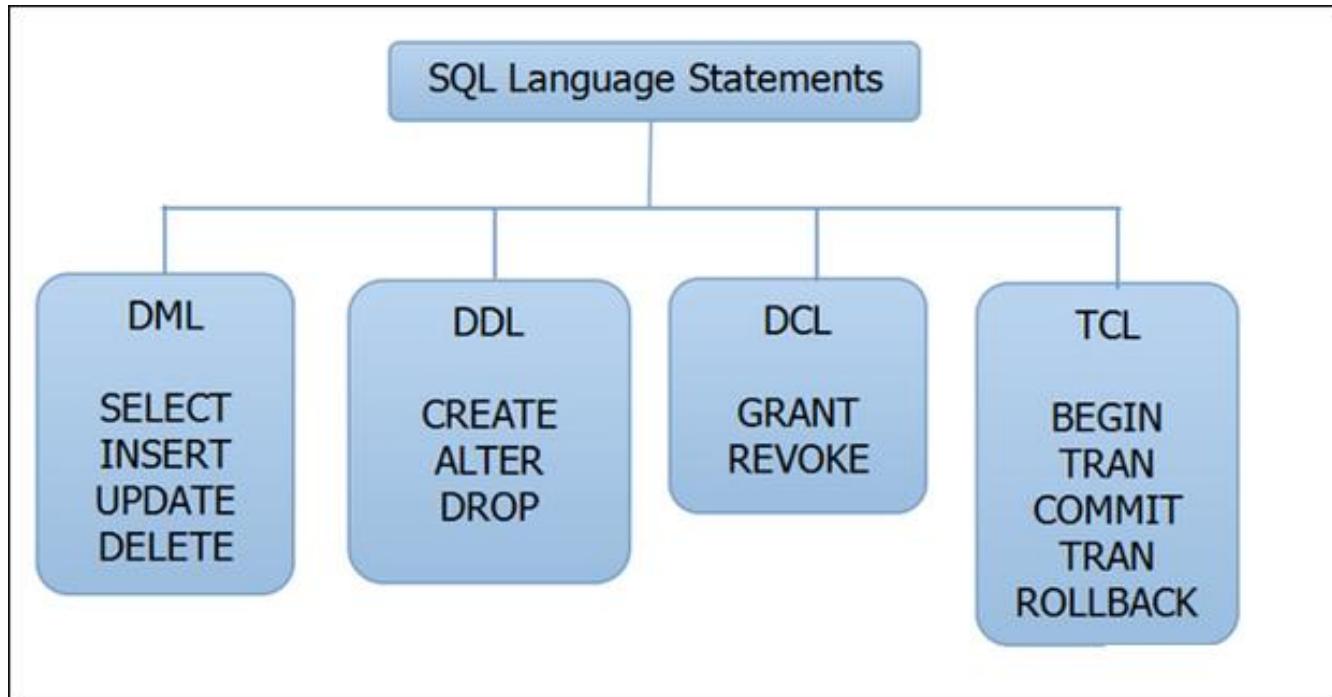
(c) The *depositor* table

SQL



- SQL (**Structured Query Language**) is a standardized programming language that's used to manage relational databases and perform various operations on the data in them
- An official SQL standard was adopted by the American National Standards Institute (ANSI) in 1986 and then by the International Organization for Standardization, known as ISO, in 1987.
- The four main categories of SQL statements are as follows:
 1. DML (Data Manipulation Language)
 2. DDL (Data Definition Language)
 3. DCL (Data Control Language)
 4. TCL (Transaction Control Language)

SQL



DDL (Data Definition Language)

- DDL statements are used to alter/modify a database or table structure and schema.
- These statements handle the design and storage of database objects.

CREATE – create a new Table, database, schema

ALTER – alter existing table, column description

DROP – delete existing objects from database

TRUNCATE - delete the table completely

DML (Data Manipulation Language)

- DML statements affect records in a table.
- These are basic operations we perform on data such as selecting a few records from a table, inserting new records, deleting unnecessary records, and updating/modifying existing records.
- DML statements include the following:

SELECT – select records from a table

INSERT – insert new records

UPDATE – update/Modify existing records

DELETE – delete existing records

DCL (Data Control Language)

- DCL statements control the level of access that users have on database objects.

GRANT – allows users to read/write on certain database objects

REVOKE – keeps users from read/write permission on database objects

TCL (Transaction Control Language)

- TCL statements allow you to control and manage transactions to maintain the integrity of data within SQL statements.

BEGIN Transaction – opens a transaction

COMMIT Transaction – commits a transaction

ROLLBACK Transaction – ROLLBACK a transaction in case of any error

Creating a Database

- To initialize a new database:

Syntax:

```
CREATE DATABASE database_name
```

- There are numerous arguments that go along with this command but are database specific
- Only some databases require database to be created and space to be allocated prior to creation of tables.
- Some databases provide graphical user interfaces to create databases and allocate space.

Creating a Table

Syntax

```
CREATE TABLE table_name
(
    Column_name datatype[(size)] [constraints1],
    Column_name datatype[(size)] [constraints2],
    ....
)
```

Example

```
CREATE TABLE books
(
    ISBN char(20),
    Title char(50),
    AuthorID Integer,
    Price float
)
```

Data Types



- Following broad categories of data types exist in most databases:
 - String Data
 - Numeric Data
 - Temporal Data(Date/Time Datatypes)
 - Large Objects

String Datatypes

Data Type Syntax	Maximum Size	Explanation
CHAR(<i>size</i>)	Maximum size of 8,000 characters.	Where size is the number of characters to store. Fixed-length. Space padded on right to equal size characters. Non-Unicode data.
VARCHAR(<i>size</i>) or VARCHAR(max)	Maximum size of 8,000 or max characters.	Where size is the number of characters to store. Variable-length. If <i>max</i> is specified, the maximum number of characters is 2GB. Non-Unicode data.
TEXT	Maximum size of 2GB.	Variable-length. Non-Unicode data.
NCHAR(<i>size</i>)	Maximum size of 4,000 characters.	Fixed-length. Unicode data.
NVARCHAR(<i>size</i>) or NVARCHAR(max)	Maximum size of 4,000 or max characters.	Where size is the number of characters to store. Variable-length. If <i>max</i> is specified, the maximum number of characters is 2GB. Unicode data.
NTEXT	Maximum size of 1,073,741,823 bytes.	Variable length. Unicode data.
BINARY(<i>size</i>)	Maximum size of 8,000 characters.	Where size is the number of characters to store. Fixed-length. Space padded on right to equal size characters. Binary data.
VARBINARY(<i>size</i>) or VARBINARY(max)	Maximum size of 8,000 or max characters.	Where size is the number of characters to store. Variable-length. If <i>max</i> is specified, the maximum number of characters is 2GB. Non-Binary data.
IMAGE	Maximum size of 2GB.	Variable length . Binary data.

Numeric Datatypes

Data Type Syntax	Maximum Size	Explanation
BIT	Integer that can be 0, 1, or NULL.	
TINYINT	0 to 255	
SMALLINT	-32768 to 32767	
INT	-2,147,483,648 to 2,147,483,647	
BIGINT	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	
DECIMAL(<i>m</i> , <i>d</i>)	<i>m</i> defaults to 18, if not specified. <i>d</i> defaults to 0, if not specified.	Where <i>m</i> is the total digits and <i>d</i> is the number of digits after the decimal.
DEC(<i>m</i> , <i>d</i>)	<i>m</i> defaults to 18, if not specified. <i>d</i> defaults to 0, if not specified.	Where <i>m</i> is the total digits and <i>d</i> is the number of digits after the decimal. This is a synonym for the DECIMAL datatype.
NUMERIC(<i>m</i> , <i>d</i>)	<i>m</i> defaults to 18, if not specified. <i>d</i> defaults to 0, if not specified.	Where <i>m</i> is the total digits and <i>d</i> is the number of digits after the decimal. This is a synonym for the DECIMAL datatype.
FLOAT(<i>n</i>)	Floating point number. <i>n</i> defaults to 53, if not specified.	Where <i>n</i> is the number of number of bits to store in scientific notation.
REAL	Equivalent to FLOAT(24)	
SMALLMONEY	- 214,748.3648 to 214,748.3647	
MONEY	-922,337,203,685,477.5808 to 922,337,203,685,477.5807	

Date/Time Datatypes

Data Type Syntax	Maximum Size	Explanation (if applicable)
DATE	Values range from '0001-01-01' to '9999-12-31'.	Displayed as 'YYYY-MM-DD'
DATETIME	Date values range from '1753-01-01 00:00:00' to '9999-12-31 23:59:59'. Time values range from '00:00:00' to '23:59:59:997'	Displayed as 'YYYY-MM-DD hh:mm:ss[.mmm]'
DATETIME2(<i>fractional seconds precision</i>)	Date values range from '0001-01-01' to '9999-12-31'. Time values range from '00:00:00' to '23:59:59:9999999'.	Displayed as 'YYYY-MM-DD hh:mm:ss[.fractional seconds]'
SMALLDATETIME	Date values range from '1900-01-01' to '2079-06-06'. Time values range from '00:00:00' to '23:59:59'.	Displayed as 'YYYY-MM-DD hh:mm:ss'
TIME	Values range from '00:00:00.0000000' to '23:59:59.9999999'	Displayed as 'YYYY-MM-DD hh:mm:ss[.nnnnnnnn]'
DATETIMEOFFSET(<i>fractional seconds precision</i>)	Date values range from '0001-01-01' to '9999-12-31'. Time values range from '00:00:00' to '23:59:59:9999999'. Time zone offset range from -14:00 to +14:00.	Displayed as 'YYYY-MM-DD hh:mm:ss[.nnnnnnnn] [{+ -}hh:mm]'

SQL constraints



- SQL constraints are a set of rules implemented on tables to control what data can be inserted, updated or deleted.
- So, SQL constraints are used to specify rules for the data in a table.
- Constraints are used to limit the type of data that can go into a table.
- This ensures the accuracy and reliability of the data in the table.
- If there is any violation between the constraint and the data action, the action is aborted.
- Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

SQL constraints

- The following constraints are commonly used in SQL:
 1. NOT NULL - Ensures that a column cannot have a NULL value
 2. UNIQUE - Ensures that all values in a column are different
 3. PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table

```
CREATE TABLE Student
(  
    sid int NOT NULL,  
    name varchar(60),  
    age int  
);
```

```
CREATE TABLE Student
(  
    sid int NOT NULL,  
    name varchar(60),  
    email varchar(50) NOT NULL UNIQUE,  
    age int  
);
```

```
CREATE table Student
(  
    sid int PRIMARY KEY,  
    Name varchar(60) NOT NULL  
    Age int  
);
```

SQL constraints

- The following constraints are commonly used in SQL:
 1. NOT NULL - Ensures that a column cannot have a NULL value
 2. UNIQUE - Ensures that all values in a column are different
 3. PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
 4. FOREIGN KEY - Prevents actions that would destroy links between tables
 5. CHECK - Ensures that the values in a column satisfies a specific condition

```
CREATE table BookIssue
(
    Issueid int PRIMARY KEY,
    bookname varchar(60) NOT NULL,
    sid int FOREIGN KEY REFERENCES Student(sid)
);
```

```
CREATE table Student
(
    sid int PRIMARY KEY,
    Name varchar(60) NOT NULL,
    Age int NOT NULL CHECK(age > 12)
);
```

SQL constraints

- The following constraints are commonly used in SQL:
 1. NOT NULL - Ensures that a column cannot have a NULL value
 2. UNIQUE - Ensures that all values in a column are different
 3. PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
 4. FOREIGN KEY - Prevents actions that would destroy links between tables
 5. CHECK - Ensures that the values in a column satisfies a specific condition
 6. DEFAULT - Sets a default value for a column if no value is specified

```
CREATE TABLE Person
(  
  ID int NOT NULL,  
  LastName varchar(25) NOT NULL,  
  FirstName varchar(25),  
  Age int,  
  Country varchar(25) DEFAULT 'Nepal'  
);
```

```
CREATE TABLE Orders  
(  
  ID int NOT NULL,  
  OrderNumber int NOT NULL,  
  OrderDate date DEFAULT GETDATE()  
);
```

Keys



Super Key

- Super key is a set of one or more than one attributes that can be used to uniquely identify a record uniquely in a table.

Candidate Key

- A Candidate Key is a set of one or more fields/columns that can identify a record uniquely in a table and is a minimal super key.
- There can be multiple Candidate Keys in one table.

Primary Key

- Primary key is a set of one or more fields/columns of a table from a candidate key which is chosen to uniquely identify a record in a database table.
- It can not accept null, duplicate values.
- Only one Candidate Key can be Primary Key.

Keys



Alternate key

- An Alternate key is a key that can work as a primary key.
- Basically, it is a candidate key that currently is not a primary key.


Foreign Key

- A foreign key is a column which is known as Primary Key in the other table i.e. A Primary Key in a table can be referred to as a Foreign Key in another table.

Composite/Compound Key

- A composite key is a combination of more than one column to identify the particular row in the table.

CREATE table



```
CREATE table Student
(
    sid int PRIMARY KEY,
    Name varchar(60) NOT NULL,
    Age int
);
```

```
CREATE table Student
(
    sid int,
    Name varchar(60) NOT NULL,
    Age int,
    PRIMARY KEY(sid)
);
```

Modifying Tables: **ALTER** statement

Syntax: ALTER TABLE table_name clause

- Clauses: – some are DBMS specific!
 1. ADD COLUMN column_name column_type [constraints]
 2. DROP COLUMN column_name

Example:

- ALTER COLUMN / MODIFY
ADD CONSTRAINT constraint
- ALTER COLUMN / MODIFY
DROP CONSTRAINT constraint_name

Removing Tables: **DROP** statement



Syntax: DROP TABLE table_name

Example: DROP TABLE Departments;

If there are constraints dependent on table:

- Remove constraints
- Drop table

Example:

- ALTER TABLE Students
DROP CONSTRAINT FK_Department;
DROP TABLE Departments;

Inserting data into a table: **INSERT** statement



Syntax:

```
INSERT INTO table_name [ (column_list) ] VALUES (data_values)
```

Or,

```
INSERT INTO table_name [ (column_list) ] select_statement
```

Example:

```
INSERT INTO Students (StudentNumber, StudentLastName, StudentFirstName)  
VALUES (190, 'Smith', 'John');
```

```
INSERT INTO Students VALUES(190, 'Smith', 'John', 'jsmith@usna.edu', '410-431-  
3456')
```

Updating records: UPDATE Statement



Syntax:

UPDATE table_name

SET column_name1 = expression1 [,column_name2 = expression2,...]

[WHERE search_condition]

Example:

UPDATE Students

SET PhoneNumber = '410-123-4567'

WHERE StudentNumber = 673;

DELETE Statement



Syntax:

```
DELETE FROM table_name  
[ WHERE search_condition ]
```

Example:


```
DELETE FROM Students  
WHERE StudentNumber = 190;
```

If you omit the WHERE clause, you will delete every row in the table!!!

Another example:


```
DELETE FROM Departments  
WHERE DepartmentName = 'ComScience';
```

SELECT Statement: Simple query



```
SELECT [DISTINCT] column_name(s) | *  
FROM table_name(s)  
[WHERE conditions]  
[ORDER BY column_name(s) [ASC/DESC]]
```

Sorting the Results



```
SELECT [DISTINCT] column_name(s) | *  
FROM table_name(s)  
[WHERE conditions]  
[ORDER BY column_name(s) [ASC/DESC]]
```

Example:

Students(SN, SName, Email, Major)

```
SELECT SN, SName
```

```
FROM Students
```

```
ORDER BY SName ASC, SN DESC
```

WHERE Clause Options

- AND, OR
- IN, NOT IN, BETWEEN
- LIKE

Wild card characters: Pattern matching

SQL-92 Standard (SQL Server, Oracle, etc.):

- _ = Exactly one character
- % = Any set of one or more characters

MS Access

- ? = Exactly one character
- * = Any set of one or more characters

```
SELECT *  
FROM EMPLOYEE  
WHERE Salary BETWEEN 30000 AND 40000;
```

```
SELECT SID, SName  
FROM Students  
WHERE SName LIKE 'R%' AND  
Major IN ('SIT', 'SCS')
```

Calculations in SQL: Aggregate functions

Five SQL Built-in Functions:

1. COUNT: to count the number of records or data
2. SUM: to compute the sum
3. AVG: to compute the average
4. MIN: to identify the maximum value
5. MAX: to identify the minimum value

```
SELECT COUNT(*)  
FROM Students
```

Example

```
SELECT AVG(Salary)  
FROM Employee  
WHERE department='Sales';
```

```
SELECT COUNT(DISTINCT SName)  
FROM Students  
WHERE SID > 700
```

```
SELECT AVG(Age)  
FROM Students  
WHERE name LIKE 'R_ _ _'
```


Aggregate Operators Limitations

- Returns only one row
- Not in WHERE clause

Example:


```
SELECT S.SName, MAX (Age)
FROM Students
```

----- > Illegal

```
SELECT S.SName, S.Age
FROM Students
WHERE AGE=MAX(Age)
```

----- > Illegal

GROUP-BY Clause




```
SELECT [DISTINCT] column_name(s) | aggregate_expr  
FROM table_name(s)  
[WHERE conditions]  
GROUP BY grouping_columns
```

Example:

```
SELECT ClassYear, MIN(Age)  
FROM Students  
GROUP BY ClassYear
```

ClassYear	(no column name)
2009	21
2012	17
2011	18
2010	20

HAVING Clause



```
SELECT [DISTINCT] column_name(s) | aggregate_expr  
FROM table_name(s)  
[WHERE conditions]  
GROUP BY grouping_columns  
HAVING group_conditions
```

- GROUP BY groups the rows
- HAVING restricts the groups presented in the result

Example:

```
SELECT ClassYear, MIN(Age)  
FROM Students  
WHERE Major = 'ComSci'  
GROUP BY ClassYear  
HAVING COUNT(*) > 20
```


```
SELECT Class, MIN(Age)  
FROM Students  
WHERE Major = 'ComSci'  
GROUP BY Class  
HAVING COUNT(*) > 2
```

SELECT Statement



```
SELECT [DISTINCT] column_name(s) | aggregate_expr  
FROM table_name(s)  
WHERE conditions  
GROUP BY grouping_columns  
HAVING group_conditions  
ORDER BY column_name(s) [ASC/DESC]
```

Queries on Multiple Relations



<i>customer-id</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
192-83-7465	Johnson	12 Alma St.	Palo Alto
019-28-3746	Smith	4 North St.	Rye
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

The *customer* table

<i>customer-id</i>	<i>account-number</i>
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

The *depositor* table

```
SELECT customer_id, customer_name, account_number
FROM customer, depositor
WHERE customer.customer_id = depositor.customer_id;
```

Queries on Multiple Relations

<i>customer-id</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
192-83-7465	Johnson	12 Alma St.	Palo Alto
019-28-3746	Smith	4 North St.	Rye
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

<i>account-number</i>	<i>balance</i>
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

```
SELECT customer_id, customer_name, balance
FROM customer, depositor, account
WHERE customer.customer_id=depositor.customer_id AND
      depositor.customer_id=account.customer_id;
```

<i>customer-id</i>	<i>account-number</i>
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The *depositor* table

Aliases

- Sql aliases are used to give a temporary name for a table or a column of a table.
- Mostly used to make a column name more readable
- Aliases only exists for the duration of a particular query.
- To create an alias, 'AS' keyword is used

Syntax:

```
SELECT column_name AS alias  
FROM table_name;
```

```
SELECT column_names  
FROM table_name AS alias;
```

Example:

```
SELECT SUM(salary) AS Total_Salary  
FROM Employee;
```

```
SELECT customer_id,customer_name, balance  
FROM customer AS c, depositor AS d, account AS a  
WHERE c.customer_id=d.customer_id AND  
      d.customer_id=a.customer_id;
```

Task



Given a relation defined by the following schema,

Students(SN, SName, Email)

Courses(Cid,CName, Dept)

Enrolled(SN,Cid, Semester)

Write the query for the following:

1. List the names of the students and their semesters
2. List the names of the students with their enrolled course names.
3. Find the names of the students who are in fourth semester.
4. List the names of the students who have taken the course Computer Science.
5. List the course names which are taught in fourth semester.
6. Find the students who are studying under the Humanities department.



Given a relation defined by the following schema,

Students(SNb, SName, Email)

Courses(Cid,CName, Dept)

Enrolled(SNb,Cid, Semester)

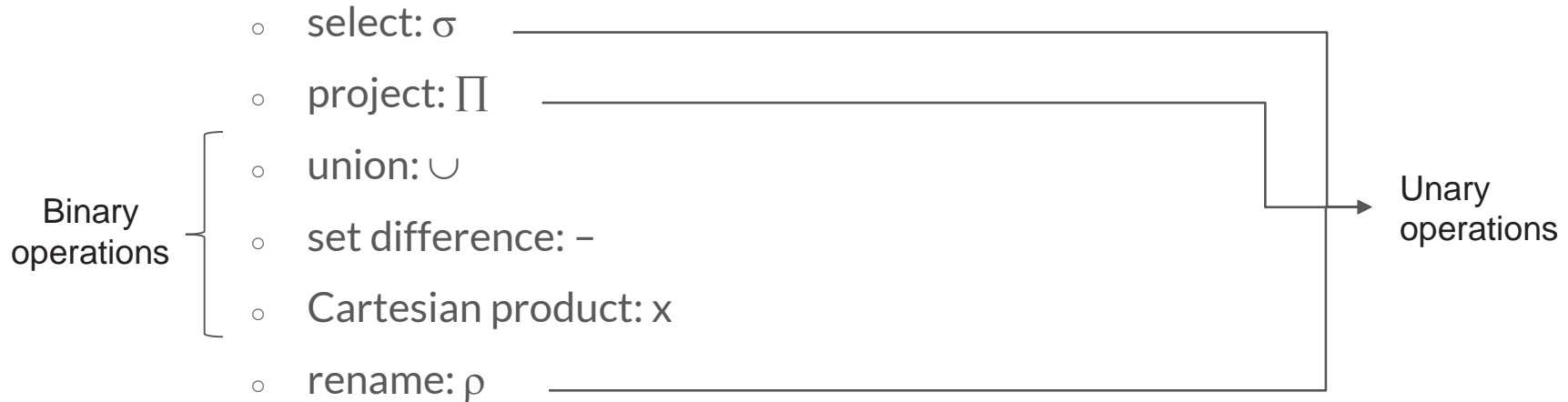
List all course names, and the number of students enrolled in the course

Formal Relational Query Languages

- These provide the mathematical foundation for the commercial relational query languages
- Either may be Procedural or non-procedural/declarative
 1. Relational algebra
 2. Tuple relational calculus
 3. Domain relational calculus

Relational Algebra

- A procedural language consisting of a set of operations that take one or two relations as input and produce a new relation as their result.
- RA mainly provides theoretical foundation for relational databases and SQL.
- Six basic operators



Select Operation

- The **select** operation selects tuples that satisfy a given predicate.
- **Notation:** $\sigma_p(r)$
- p is called the **selection predicate**
- Example: Select those tuples of the instructor relation where the instructor is in the “Physics” department.

- Query

$\sigma_{\text{dept_name}=\text{“Physics”}}(\text{instructor})$

- Result

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

Select Operation

- Comparisons can be done in the selection predicate using:

$=, \neq, >, \geq, <, \leq$

- Several predicates can be combined into a larger predicate by using the connectives:

\wedge (**and**), \vee (**or**), \neg (**not**)

- Example: Find the instructors in Physics with a salary greater \$90,000, we write:

$\sigma_{\text{dept_name}=\text{"Physics"} \wedge \text{salary} > 90,000}(\text{instructor})$

- The select predicate may include comparisons between two attributes.
 - Example, find all departments whose name is the same as their building name:

$\sigma_{\text{dept_name}=\text{building}}(\text{department})$

Project Operation

- A unary operation that returns its argument relation, with certain attributes left out.
- Notation:

$\Pi_{A_1, A_2, A_3, \dots, A_k}(r)$
where A_1, A_2, \dots, A_k are attribute names and r is a relation name.

- The result is defined as the relation of k columns obtained by excluding the columns that are not mentioned
- Duplicate rows removed from result, since relations are sets

Project Operation

- Example: Display ID, name and salary attributes of instructor
- Query:

$\Pi_{ID, name, salary}(\text{instructor})$

- Result:

<i>ID</i>	<i>name</i>	<i>salary</i>
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

Composition of Relational Operations

- The result of a relational-algebra operation is relation and therefore of relational-algebra operations can be composed together into a **relational-algebra expression**.
- Consider the query: Find the names of all instructors in the Physics department.

$$\Pi_{\text{name}}(\sigma_{\text{dept_name} = \text{"Physics"}}(\text{instructor}))$$

- Instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.

Cartesian-Product Operation



- The Cartesian-product operation (denoted by \times) allows us to combine information from any two relations.
- Example: The Cartesian product of the relations instructor and teaches is written as: $\text{instructor} \times \text{teaches}$
- We construct a tuple of the result out of each possible pair of tuples: one from the instructor relation and one from the teaches relation
- If multiple tables have the same column name then to distinguish them we append the name of the table before the attribute name and connect with dot operator.
- So, if instructor ID appears in both relations we distinguish between these attribute by attaching to the attribute the name of the relation from which the attribute originally came.
 - instructor.ID
 - teaches.ID

The 'instructor X teaches' table

[illegible]

Cartesian-Product

- The Cartesian-Product
instructor X teaches
associates every tuple of instructor with every tuple of teaches.
 - Most of the resulting rows have information about instructors who did NOT teach a particular course.
- To get only those tuples of “instructor X teaches” that pertain to instructors and the courses that they taught, we write:

$\sigma_{\text{instructor.id} = \text{teaches.id}} (\text{instructor} \times \text{teaches})$

Now, we get only those tuples of “instructor X teaches” that pertain to instructors and the courses that they taught.

Cartesian-Product

- The table corresponding to:

$\sigma_{\text{instructor.id} = \text{teaches.id}}(\text{instructor} \times \text{teaches})$

<i>instructor.ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>	<i>teaches.ID</i>	<i>course_id</i>	<i>sec_id</i>	<i>semester</i>	<i>year</i>
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
32343	El Said	History	60000	32343	HIS-351	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-101	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-319	1	Spring	2018
76766	Crick	Biology	72000	76766	BIO-101	1	Summer	2017
76766	Crick	Biology	72000	76766	BIO-301	1	Summer	2018
83821	Brandt	Comp. Sci.	92000	83821	CS-190	1	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-190	2	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-319	2	Spring	2018
98345	Kim	Elec. Eng.	80000	98345	EE-181	1	Spring	2017

Join Operation

- The **join** operation allows us to combine a select operation and a Cartesian-Product operation into a single operation.
- Consider relations r (R) and s (S)
- Let “theta” be a predicate on attributes in the schema R “union” S . The join operation $r \bowtie_{\theta} s$ is defined as follows:

$$r \bowtie_{\theta} s = \sigma_{\theta} (r \times s)$$

- Thus

$$\sigma_{\text{instructor.id} = \text{teaches.id}} (\text{instructor} \times \text{teaches})$$

- Can equivalently be written as

$$\text{instructor} \bowtie_{\text{Instructor.id} = \text{teaches.id}} \text{teaches.}$$

Union Operation

- The union operation allows us to combine two relations
- Notation: $r \cup s$
- For $r \cup s$ to be valid.
 1. r, s must have the same **arity** (same number of attributes)
 2. The attribute domains must be **compatible** (example: 2nd column of r deals with the same type of values as does the 2nd column of s)
- Example: to find all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or in both

$\Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Fall"} \wedge \text{year}=2017} (\text{section})) \cup$

$\Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Spring"} \wedge \text{year}=2018} (\text{section}))$

Union Operation

- Result of:

$$\Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Fall"} \wedge \text{year}=2017}(\text{section})) \cup \\ \Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Spring"} \wedge \text{year}=2018}(\text{section}))$$

<i>course_id</i>
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Set-Intersection Operation

- The set-intersection operation allows us to find tuples that are in both the input relations.
- Notation: $r \cap s$
- Assume:
 - r, s have the same arity
 - attributes of r and s are compatible
- Example: Find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters.

$$\Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Fall"} \wedge \text{year}=2017}(\text{section})) \cap \Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Spring"} \wedge \text{year}=2018}(\text{section}))$$

- Result

<i>course_id</i>
CS-101

Set-Intersection Operation

- The set-difference operation allows us to find tuples that are in one relation but are not in another.
- Notation: $r - s$
- Set differences must be taken between **compatible** relations.
 - r and s must have the same arity
 - attribute domains of r and s must be compatible
- Example: to find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester

$$\Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Fall"} \wedge \text{year}=2017}(\text{section})) - \Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Spring"} \wedge \text{year}=2018}(\text{section}))$$

<i>course_id</i>
CS-347
PHY-101

The Rename Operation

- The results of relational-algebra expressions do not have a name that we can use to refer to them
- The rename operator, ρ , is provided for that purpose
- The expression:

$$\rho_x(E)$$

returns the result of expression E under the name x

- Another form of the rename operation:

$$\rho_{x(A_1, A_2, \dots, A_n)}(E)$$

The Assignment Operation

- It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables.
- The assignment operation is denoted by \leftarrow and works like assignment in a programming language.
- Example: Find all instructor in the “Physics” and Music department.

$\text{Physics} \leftarrow \sigma_{\text{dept_name}=\text{“Physics”}}(\text{instructor})$

$\text{Music} \leftarrow \sigma_{\text{dept_name}=\text{“Music”}}(\text{instructor})$

$\text{Physics} \cup \text{Music}$

- With the assignment operation, a query can be written as a sequential program consisting of a series of assignments followed by an expression whose value is displayed as the result of the query.

Types of join



- Natural join
- Outer join
 - a. Left outer join
 - b. Right outer join
 - c. Full outer join

Natural join

- Notation: $r \bowtie s$
- Let r and s be relations on schemas R and S respectively.
Then, $r \bowtie s$ is a relation on schema $R \cup S$ obtained as follows:
 - Consider each pair of tuples t_r from r and t_s from s .
 - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
 - t has the same value as t_r on r
 - t has the same value as t_s on s
- Example:
 $R = (A, B, C, D), S = (E, B, D)$
 - Result schema = (A, B, C, D, E)

Natural join



ITEM_ID	ITEM_NAME	ITEM_UNIT	COMPANY_ID
1	Chex Mix	Pcs	16
6	Cheez-It	Pcs	15
2	BN Biscuit	Pcs	15
3	Mighty Munch	Pcs	17
4	Pot Rice	Pcs	15
5	Jaffa Cakes	Pcs	18
7	Salt n Shake	Pcs	-

COMPANY_ID	COMPANY_NAME	COMPANY_CITY
18	Order All	Boston
15	Jack Hill Ltd	London
16	Akas Foods	Delhi
17	Foodies.	London
19	sip-n-Bite.	New York

** Same column came once

COMPANY_ID	ITEM_ID	ITEM_NAME	ITEM_UNIT	COMPANY_NAME	COMPANY_CITY
16	1	Chex Mix	Pcs	Akas Foods	Delhi
15	6	Cheez-It	Pcs	Jack Hill Ltd	London
15	2	BN Biscuit	Pcs	Jack Hill Ltd	London
17	3	Mighty Munch	Pcs	Foodies.	London
15	4	Pot Rice	Pcs	Jack Hill Ltd	London
18	5	Jaffa Cakes	Pcs	Order All	Boston

Natural join

- Find the names of all instructors in the Comp. Sci. department together with the course titles of all the courses that the instructors teach
 - $\Pi_{\text{name, title}} (\sigma_{\text{dept_name}=\text{"Comp. Sci."}} (\text{instructor} \times \text{teaches} \times \text{course}))$

Outer Join



- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- Uses null values:
 - null signifies that the value is unknown or does not exist

Example



Relation *loan*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

Relation *borrower*

<i>customer_name</i>	<i>loan_number</i>
Jones	L-170
Smith	L-230
Hayes	L-155

loan ⋈ *Borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

Example



Relation *loan*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

Relation borrower

<i>customer_name</i>	<i>loan_number</i>
Jones	L-170
Smith	L-230
Hayes	L-155

Left Outer Join

loan  *Borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	<i>null</i>

Example



Relation *loan*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

Relation *borrower*

<i>customer_name</i>	<i>loan_number</i>
Jones	L-170
Smith	L-230
Hayes	L-155

Right Outer Join

loan ⋈_r *borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	<i>null</i>	<i>null</i>	Hayes

Example



Relation *loan*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

Relation *borrower*

<i>customer_name</i>	<i>loan_number</i>
Jones	L-170
Smith	L-230
Hayes	L-155

Full Outer Join

loan ⋈ *borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	<i>null</i>
L-155	<i>null</i>	<i>null</i>	Hayes

Null Values



- It is possible for tuples to have a null value, denoted by null, for some of their attributes
- null signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving null is null.
- Aggregate functions simply ignore null values (as in SQL)
- For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same (as in SQL)

Aggregate Functions and Operations

- **Aggregation function** takes a collection of values and returns a single value as a result.

avg: average value

min: minimum value

max: maximum value

sum: sum of values

count: number of values

- **Aggregate operation** in relational algebra

$$G_1, G_2, \dots, G_n \quad \mathcal{G}_{F_1(A_1), F_2(A_2), \dots, F_n(A_n)}(E)$$

E is any relational-algebra expression

- G_1, G_2, \dots, G_n is a list of attributes on which to group (can be empty)
- Each F_i is an aggregate function
- Each A_i is an attribute name

- Note: Some books/articles use γ instead of \mathcal{G} [Calligraphic G]

Aggregate Functions and Operations

- Relation account grouped by branch-name:

<i>branch_name</i>	<i>account_number</i>	<i>balance</i>
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

branch_name **g** **sum**(*balance*) (*account*)

<i>branch_name</i>	sum (<i>balance</i>)
Perryridge	1300
Brighton	1500
Redwood	700

Aggregate Functions and Operations



- Result of aggregation does not have a name
 - Can use rename operation to give it a name
 - For convenience, we permit renaming as part of aggregate operation

Other RA Operations

- A deletion is expressed in relational algebra by:

$$r \leftarrow r - E$$

where r is a relation and E is a relational algebra query.

$$account \leftarrow account - \sigma_{branch_name = \text{"Perryridge"}}(account)$$

- **Insertion:** in relational algebra, an insertion is expressed by:

$$r \leftarrow r \cup E$$

where r is a relation and E is a relational algebra expression.

$$account \leftarrow account \cup \{(\text{"A-973"}, \text{"Perryridge"}, 1200)\}$$

$$depositor \leftarrow depositor \cup \{(\text{"Smith"}, \text{"A-973"})\}$$

- **Update:** Use the generalized projection operator to do this task

$$r \leftarrow \Pi_{F_1, F_2, \dots, F_l}(r)$$

$$account \leftarrow \Pi_{account_number, branch_name, balance * 1.05}(account)$$

Examples

- Find all loans of over \$1200

$$\sigma_{amount > 1200} (loan)$$

- Find the loan number for each loan of an amount greater than \$1200

$$\Pi_{loan_number} (\sigma_{amount > 1200} (loan))$$

- Find the names of all customers who have a loan, an account, or both, from the bank

$$\Pi_{customer_name} (borrower) \cup \Pi_{customer_name} (depositor)$$

- Find the names of all customers who have a loan and an account at bank.

$$\Pi_{customer_name} (borrower) \cap \Pi_{customer_name} (depositor)$$

Examples

- Find the names of all customers who have a loan at the Perryridge branch.

$$\Pi_{customer_name} (\sigma_{branch_name = "Perryridge"} (\sigma_{borrower.loan_number = loan.loan_number} (borrower \times loan)))$$

- Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.

$$\Pi_{customer_name} (\sigma_{branch_name = "Perryridge"} (\sigma_{borrower.loan_number = loan.loan_number} (borrower \times loan))) - \Pi_{customer_name} (depositor)$$

Examples

- loan **inner join** borrower on
loan.loan_number = borrower.loan_number

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>	<i>loan_number</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

- loan **left outer join** borrower on
loan.loan_number = borrower.loan_number

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>	<i>loan_number</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	<i>null</i>	<i>null</i>

Examples

- loan **inner join** borrower on
loan.loan_number = borrower.loan_number
- loan **left outer join** borrower on
loan.loan_number = borrower.loan_number

- Find all customers who have either an account or a loan (but not both) at the bank.

```
select customer_name  
      from (depositor natural full outer join borrower)  
      where account_number is null or loan_number is null
```

Views

- Provide a mechanism to hide certain data from the view of certain users. To create a view we use the command:
- **create view** *v* **as** <query expression>
- where:
- <query expression> is any legal expression
- The view name is represented by *v*

- Example:

create view *all-customer* **as**

(**select** *branch-name, customer-name*

from *depositor, account*

where *depositor.account-number = account.account-number*);

Task



Student(StudentID, Sname, Major, level)

Professor(ProfID, Pname, Department)

Supervise(ProfID, StuID, Year)

- a. Retrieve the names of all senior level students majoring in “Computer science”.
- b. Retrieve the names of all students supervised by Professor Allen King in 2007.
- c. Find the name of the Professors who supervises students from junior level.
- d. Retrieve the list of students name and their supervisor’s name of BCA department.
- e. Retrieve the name list of the professor and the number of students supervised by that professor where the number of supervised students should be more than 5.

Integrity Constraints



- Integrity constraints are a set of rules that ensures that the data insertion, updating, and other operations in database have to be performed in such a way that data integrity is not affected.
 - Each and every time a table insert, update, delete, or alter operation is performed, it is evaluated against the terms or rules mentioned in the integrity constraint.
 - So, it is used to guard against accidental damage to the database.
-
1. Domain Constraint
 2. Referential Integrity Constraint
 3. Triggers
 4. Assertion

Domain Constraint

- A domain of possible values must be associated with every attribute.
- A domain types may be integers, types, character types, and date/time types defined in SQL
- Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take.
- Domain constraints are the most elementary form of integrity constraint and are tested whenever a new data item is entered into the database.

Example:

```
CREATE TABLE Student
(  
    Cid integer,  
    Name varchar(30),  
    Address varchar(40),  
    Date_of_birth date  
)
```

Domain Constraint

- The **create domain** clause can be used to define new domains.
- For example, the statements:

```
create domain Dollars numeric(12,2)  
create domain Pounds numeric(12,2)
```

- It defines the domains Dollars and Pounds to be decimal numbers with a total of 12 digits, two of which are placed after the decimal point.

```
create domain HourlyWage numeric(5,2)  
constraint wage-value-test check(value >= 4.00)
```

Referential Integrity Constraints

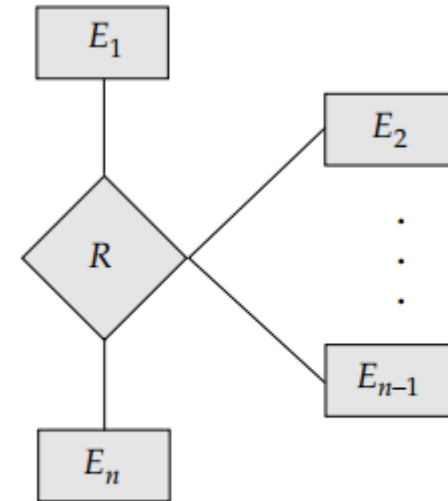
- A value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
- This condition is called **referential integrity**.

Example:

- Suppose there is a tuple t_1 in the account relation with
 $t_1[\text{branch-name}] = \text{"Lunartown,"}$
but there is no tuple in the branch relation for the Lunartown branch.
This situation would be undesirable and we expect the branch relation to list all bank branches.
- The attribute branch-name in Account-schema is a foreign key referencing the primary key of Branch-schema.

Referential Integrity Constraints and ER diagram

- For an n -ary relationship set R , relating entity sets E_1, E_2, \dots, E_n , let K_i be the primary key of E_i .
- The attributes of the relation schema for relationship set R include $K_1 \cup K_2 \cup \dots \cup K_n$.
- Referential integrity: For each i , K_i in the schema for R is a foreign key referencing K_i in the relation schema generated from entity set E_i



An n -ary relationship set.

Assertions



- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- Domain constraints and referential-integrity constraints are special forms of assertions.

Example

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.
- Every loan has at least one customer who maintains an account with a minimum balance of \$1000.00.

Assertions

- An assertion in SQL takes the form

create assertion <assertion-name> **check** <predicate>

```
create assertion sum-constraint check
(not exists (select * from branch
where (select sum(amount) from loan
where loan.branch-name = branch.branch-name)
>= (select sum(balance) from account
where account.branch-name = branch.branch-name)))
```

Triggers



- A **trigger** is a statement that the system executes automatically as a side effect of a modification to the database.
- To design a trigger mechanism, we must meet two requirements:
 1. Specify when a trigger is to be executed. This is broken up into an event that causes the trigger to be checked and a condition that must be satisfied for trigger execution to proceed.
 2. Specify the actions to be taken when the trigger executes.

Triggers



```
create trigger overdraft-trigger after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
insert into borrower
(select customer-name, account-number
from depositor
where nrow.account-number = depositor.account-number);
insert into loan values
(nrow.account-number, nrow.branch-name, - nrow.balance);
update account set balance = 0
where account.account-number = nrow.account-number
end
```