UNIT-II

Relational operations & Basic SQL: Relational Algebra, Relational Operations, Relational Calculus, Tuple, And Domain Relational Calculus.

PL/ SQL: Database Languages, Data Types, Integrity Constraints, Simple And Nested Queries, Implementation Of Different Types Of Joins, Stored Procedures

Relational Algebra

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either unary or binary. They accept relations as their input and yield relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

The fundamental operations of relational algebra are as follows −

- **Select**
- **Project**
- **Union**
- **Set different**
- **Cartesian product**
- **Rename**

Select Operation (σ)

It selects tuples that satisfy the given predicate from a relation.

Notation − σp(r)

Where σ stands for selection predicate and r stands for relation. p is a prepositional logic formula that uses connectors like and, or, and not. These terms may use relational operators like − =, ≠, ≥, <, >, ≤.

For example −

σsubject = "database"(Books)

Output − Selects tuples from books where the subject is 'database'.

σsubject = "database" and price = "450"(Books)

Output − Selects tuples from books where the subject is 'database' and 'price' is 450.

Project Operation (∏)

It projects column(s) that satisfy a given predicate.

Notation − ∏A1, A2, An (r)

Where A1, A2 , An are attribute names of relation r. Duplicate rows are automatically eliminated, as relation is a set.

For example − ∏subject, author (Books)

Selects and projects columns named as subject and author from the relation Books.

Union Operation (∪)

It performs binary union between two given relations and is defined as −

r ∪ s = { t | t ∈ r or t ∈ s}

Notation − r U s

Where r and s are either database relations or relation result set (temporary relation). For a union operation to be valid, the following conditions must hold −

r, and s must have the same number of attributes.

Attribute domains must be compatible.

Duplicate tuples are automatically eliminated.

∏ author (Books) ∪ ∏ author (Articles)

Output − Projects the names of the authors who have either written a book or an article or both.

Set Difference (−)

The result of set difference operation is tuples, which are present in one relation but are not in the second relation.

Notation − r − s

Finds all the tuples that are present in r but not in s.

∏ author (Books) − ∏ author (Articles)

Output − Provides the name of authors who have written books but not articles.

Cartesian Product (X)

Combines information of two different relations into one.

Notation − r X s

Where r and s are relations and their output will be defined as −

r X s = { q t | q ∈ r and t ∈ s}

σauthor = 'RaghuramaKrishna'(Books X Articles)

Output − Yields a relation, which shows all the books and articles written by RaghuramaKrishna.

Rename Operation (ρ)

The results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. 'rename' operation is denoted with small Greek letter rho ρ.

Notation − ρ x (E)

Where the result of expression E is saved with name of x.

Additional operations are −

Set intersection

Assignment

Natural join

Set Operations

The following standard operations on sets are also available in relational algebra: union (U), intersection (∩), set-difference (−), and cross-product (×).

• **Union:** R u S returns a relation instance containing all tuples that occur in either relation instance R or relation instance S (or both). R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

• **Intersection:** R ∩ S returns a relation instance containing all tuples that occur in both R and S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

• **Set-difference:** R − S returns a relation instance containing all tuples that occur in R but not in S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

• **Cross-product:** R × S returns a relation instance whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of S (in the same order as they appear in S). The result of R × S contains one tuple ⟨r, s⟩ (the concatenation of tuples r and s) for each pair of tuples r ∈ R, s ∈ S. The cross-product operation is sometimes called Cartesian product.

sid	*sname*	*rating*	*age*
31	*Lubbe*	*8*	*55.5*
58	*Rusty*	*10*	*35.0*

Figure 4.9 S1 ∩ S2

sid	sname	rating	age
22	Dustin	7	45.0

Figure 4.10 S1 − S2

The result of the cross-product S1 × R1 is shown in Figure 4.11 The fields in S1 × R1 have the same domains as the corresponding fields in R1 and S1. In Figure 4.11 sid is listedin parentheses to emphasize that it is not an inherited field name; only the corresponding domain is inherited.

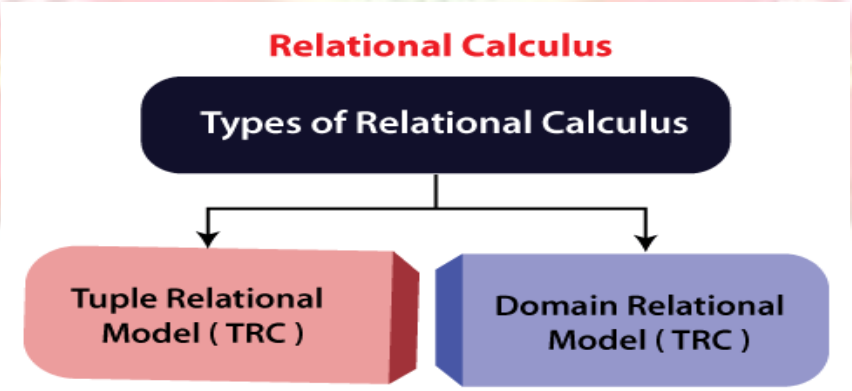
(sid)	sname	rating	age	(sid)	bid	day
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

Figure 4.11 S1 × R1

RELATIONAL CALCULUS

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or declarative, in that it allows us to describe the set of answers without being explicit about how they should be computed.

Types of Relational Calculus



Tuple Relational Calculus

A tuple variable is a variable that takes on tuples of a particular relation schema as values. That

is, every value assigned to a given tuple variable has the same number and type of fields.

It is a non-procedural query language which is based on finding a number of tuple variables also known as range variable for which predicate holds true. It describes the desired information without giving a specific procedure for obtaining that information. The tuple relational calculus is specified to select the tuples in a relation. In TRC, filtering variable uses the tuples of a relation. The result of the relation can have one or more tuples.

Notation: A Query in the tuple relational calculus is expressed as following notation

{T | P (T)} or {T | Condition (T)}

Where T is the resulting tuples

P(T) is the condition used to fetch T.

For example: { T.name | Author(T) AND T.article = 'database' }

Output: This query selects the tuples from the AUTHOR relation. It returns a tuple with 'name' from Author who has written an article on 'database'.

TRC (tuple relation calculus) can be quantified. In TRC, we can use Existential (∃) and Universal Quantifiers (∀).

For example: { R| ∃T ∈ Authors(T.article='database' AND R.name=T.name)}

Output: This query will yield the same result as the previous one.

Domain Relational Calculus

A domain variable is a variable that ranges over the values in the domain of some attribute (e.g., the variable can be assigned an integer if it appears in an attribute whose domain is the set of integers).

The second form of relation is known as Domain relational calculus. In domain relational calculus, filtering variable uses the domain of attributes. Domain relational calculus uses the same operators as tuple calculus. It uses logical connectives ∧ (and), ∨ (or) and ¬ (not). It uses Existential (∃) and Universal Quantifiers (∀) to bind the variable. The QBE or Query by example is a query language related to domain relational calculus.

Notation: { a1, a2, a3, ..., an | P (a1, a2, a3, ... ,an)}

Where a1, a2 are attributes, P stands for formula built by inner attributes

For example: {< article, page, subject > | ∈ javatpoint ∧ subject = 'database'}

Output: This query will yield the article, page, and subject from the relational javatpoint, where the subject is a database.

PL/ SQL : Database Languages, Data Types, Integrity Constraints, Simple And Nested Queries, Implementation Of Different Types Of Joins, Stored Procedures

SQL : SQL stands for Structured Query Language. It is used for storing and managing data in relational database management system (RDMS).

- It is a standard language for Relational Database System. It enables a user to create, read, update and delete relational databases and tables.
- All the RDBMS like MySQL, Informix, Oracle, MS Access and SQL Server use SQL as their standard database language.
- SQL allows users to query the database in a number of ways, using English-like statements.

SQL Commands

The standard SQL commands to interact with relational databases are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP. These commands can be classified into the following groups based on their nature –

DDL - Data Definition Language

Sr.No.	Command & Description
1	**CREATE :** Creates a new table, a view of a table, or other object in the database.
2	**ALTER :**Modifies an existing database object, such as a table.
3	**DROP:** Deletes an entire table, a view of a table or other objects in the database.

DML - Data Manipulation Language

Sr.No.	Command & Description
1	**SELECT :** Retrieves certain records from one or more tables.
2	**INSERT :**Creates a record.
3	**UPDATE :**Modifies records.
4	**DELETE :**Deletes records.

DCL - Data Control Language

Sr.No.	Command & Description
1	**GRANT :** Gives a privilege to user.
2	**REVOKE :** Takes back privileges granted from user.

TCL - Transaction Control Language

Sr.No.	Command & Description
1	**COMMIT :** Gives a privilege to user.
2	**ROLLBACK :** Takes back privileges granted from user.
3	**SAVEPOINT :** Creates points within the groups of transactions in which to ROLLBACK

1**. DATA DEFINITION LANGUAGE (DDL):** The Data Definition Language (DDL) is used to create and destroy databases and database objects. These commands will primarily be used by database administrators during the setup and removal phases of a database project.

Let's take a look at the structure and usage of four basic DDL commands:

1. CREATE 2. ALTER 3. DROP 4. RENAME

1. CREATE:

(a)CREATE TABLE: This is used to create a new relation (table)

Syntax: CREATE TABLE <relation_name/table_name > (field_1 data_type(size),field_2 data_type(size), .. .);

Example:

SQL> CREATE TABLE Student (sno NUMBER (3), sname CHAR (10), class CHAR (5));

2. ALTER:

(a)ALTER TABLE ...ADD...: This is used to add some extra fields into existing relation.

Syntax: ALTER TABLE relation_name ADD (new field_1 data_type(size), new field_2 data_type(size),..);

Example: SQL>ALTER TABLE student ADD (Address CHAR(10));

(b)ALTER TABLE...MODIFY...: This is used to change the width as well as data type of fields of existing relations.

Syntax: ALTER TABLE relation_name MODIFY (field_1 newdata_type(Size), field_2 newdata_type(Size),....field_newdata_type(Size));

Example: SQL>ALTER TABLE student MODIFY(sname VARCHAR(10), class VARCHAR(5));

c) ALTER TABLE..DROP...: This is used to remove any field of existing relations.

Syntax: ALTER TABLE relation_name DROP COLUMN (field_name);

Example: SQL>ALTER TABLE student DROP column (sname);

d)ALTER TABLE..RENAME...: This is used to change the name of fields in existing relations.

Syntax: ALTER TABLE relation_name RENAME COLUMN (OLD field_name) to (NEW field_name);

Example: SQL>ALTER TABLE student RENAME COLUMN sname to stu_name;

3. DROP TABLE: This is used to delete the structure of a relation. It permanently deletes the records in the table.

Syntax: DROP TABLE relation_name;

Example: SQL>DROP TABLE std;

4. RENAME: It is used to modify the name of the existing database object.

Syntax: RENAME TABLE old_relation_name TO new_relation_name;

Example: SQL>RENAME TABLE std TO std1;

DATA MANIPULATION LANGUAGE (DML): The Data Manipulation Language (DML) is used to retrieve, insert and modify database information. These commands will be used by all database users during the routine operation of the database. Let's take a brief look at the basic DML commands:

1. INSERT 2. UPDATE 3. DELETE 4. TRUNCATE 5. SELECT

1. INSERT INTO: This is used to add records into a relation. These are three type of INSERT INTO queries which are as

a) Inserting a single record

Syntax: INSERT INTO < relation/table name> (field_1,field_2……field_n)VALUES (data_1,data_2,.....data_n);

Example: SQL>INSERT INTO student(sno,sname,class,address)VALUES (1,'Ravi','M.Tech','Palakol');

b) Inserting a single record

Syntax: INSERT INTO < relation/table name>VALUES (data_1,data_2,.....data_n);

Example: SQL>INSERT INTO student VALUES (1,'Ravi','M.Tech','Palakol');

c) Inserting all records from another relation

Syntax: INSERT INTO relation_name_1 SELECT Field_1,field_2,field_n FROM relation_name_2 WHERE field_x=data;

Example: SQL>INSERT INTO std SELECT sno,sname FROM student WHERE name = 'Ramu';

d) Inserting multiple records

Syntax: INSERT INTO relation_name field_1,field_2,.....field_n) VALUES (&data_1,&data_2,.....&data_n);

Example: SQL>INSERT INTO student (sno, sname, class,address) VALUES (&sno,'&sname','&class','&address');

Enter value for sno: 101

Enter value for name: Ravi

Enter value for class: B.Tech

Enter value for name: Palakol

2. UPDATE-SET-WHERE: This is used to update the content of a record in a relation.

Syntax: SQL>UPDATE relation name SET Field_name1=data,field_name2=data,

WHERE field_name=data;

Example: SQL>UPDATE student SET sname = 'kumar' WHERE sno=1;

3. DELETE-FROM: This is used to delete all the records of a relation but it will retain the structure of that relation.

a) DELETE-FROM: This is used to delete all the records of relation.

Syntax: SQL>DELETE FROM relation_name;

Example: SQL>DELETE FROM std;

b) DELETE -FROM-WHERE: This is used to delete a selected record from a relation.

Syntax: SQL>DELETE FROM relation_name WHERE condition;

Example: SQL>DELETE FROM student WHERE sno = 2;

4. TRUNCATE: This command will remove the data permanently. But structure will not be removed.

Difference between Truncate & Delete:-

- By using truncate command data will be removed permanently & will not get back where as by using delete command data will be removed temporally & get back by using roll back command.
- By using delete command data will be removed based on the condition whereas by using truncate command there is no condition.
- Truncate is a DDL command & delete is a DML command.

Syntax: TRUNCATE TABLE <Table name>

Example TRUNCATE TABLE student;

- To Retrieve data from one or more tables.

5. SELECT

1. SELECT FROM: To display all fields for all records.

Syntax : SELECT * FROM relation_name;

Example : SQL> select * from dept;

DEPTNO DNAME LOC

10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON

2. SELECT FROM: To display a set of fields for all records of relation.

Syntax: SELECT a set of fields FROM relation_name;

Example: SQL> select deptno, dname from dept;

DEPTNO	DNAME
10	ACCOUNTING
20	RESEARCH
30	SALES

3. SELECT - FROM -WHERE: This query is used to display a selected set of fields for a selected set of records of a relation.

Syntax: SELECT a set of fields FROM relation_name WHERE condition;

Example: SQL> select * FROM dept WHERE deptno<=20;

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS

TCL : Transaction Control Language

1. COMMIT: This command is used to end a transaction only with the help of the commit command transaction changes can be made permanent to the database.

Syntax: SQL> COMMIT;

Example: SQL> COMMIT;

2. SAVE POINT: Save points are like marks to divide a very lengthy transaction to smaller once. They are used to identify a point in a transaction to which we can latter role back. Thus, save point is used in conjunction with role back.

Syntax: SQL> SAVE POINT ID; Example: SQL> SAVE POINT xyz;

3. ROLLBACK: A role back command is used to undo the current transactions. We can role back the entire transaction so that all changes made by SQL statements are undo (or) role back a transaction to a save point so that the SQL statements after the save point are role back.

Syntax: ROLLBACK (current transaction can be role back)

ROLLBACK to save point ID;

Example: SQL> ROLLBACK;

SQL> ROLLBACK TO SAVE POINT xyz;

SQL Server offers six categories of data types which are listed below –

Exact Numeric Data Types

DATA TYPE	FROM	TO
bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
int	-2,147,483,648	2,147,483,647
smallint	-32,768	32,767
tinyint	0	255
bit	0	1
decimal	-10^38 +1	10^38 -1
numeric	-10^38 +1	10^38 -1
money	-922,337,203,685,477.5808	+922,337,203,685,477.5807
smallmoney	-214,748.3648	+214,748.3647

Approximate Numeric Data Types

DATA TYPE	FROM	TO
Float	-1.79E + 308	1.79E + 308
real	-3.40E + 38	3.40E + 38

Date and Time Data Types

DATA TYPE	FROM	TO
datetime	Jan 1, 1753	Dec 31, 9999
smalldatetime	Jan 1, 1900	Jun 6, 2079
date	Stores a date like June 30, 1991	
time	Stores a time of day like 12:30 P.M.	

Note − Here, datetime has 3.33 milliseconds accuracy whereas smalldatetime has 1 minute accuracy.

Character Strings Data Types

Sr.No.	DATA TYPE & Description
1	**Char :**Maximum length of 8,000 characters.(Fixed length non-Unicode characters)
2	**varchar :**Maximum of 8,000 characters.(Variable-length non-Unicode data).
3	**varchar(max) :**Maximum length of 2E + 31 characters, Variable-length non-Unicode data (SQL Server 2005 only).
4	**Text :**Variable-length non-Unicode data with a maximum length of 2,147,483,647 characters.

Unicode Character Strings Data Types

Sr.No.	DATA TYPE & Description
1	**nchar :**Maximum length of 4,000 characters.(Fixed length Unicode)
2	**nvarchar :**Maximum length of 4,000 characters.(Variable length Unicode)
3	**nvarchar(max) :**Maximum length of 2E + 31 characters (SQL Server 2005 only).(Variable length Unicode)
4	**Ntext :**Maximum length of 1,073,741,823 characters. (Variable length Unicode)

Binary Data Types

Sr.No.	DATA TYPE & Description
1	**Binary :** Maximum length of 8,000 bytes(Fixed-length binary data)
2	**Varbinary :** Maximum length of 8,000 bytes.(Variable length binary data)
3	**varbinary(max) :**Maximum length of 2E + 31 bytes (SQL Server 2005 only). (Variable length Binary data)
4	**Image :**Maximum length of 2,147,483,647 bytes. (Variable length Binary Data)

Misc Data Types

Sr.No.	DATA TYPE & Description
1	**sql_variant :**Stores values of various SQL Server-supported data types, except text, ntext, and timestamp.
2	**Timestamp :**Stores a database-wide unique number that gets updated every time a row gets updated
3	**Uniqueidentifier :**Stores a globally unique identifier (GUID)
4	**Xml :**Stores XML data. You can store xml instances in a column or a variable (SQL Server 2005 only).
5	**Cursor :**Reference to a cursor object
6	**Table :**Stores a result set for later processing

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:

- Key constraints
- Domain constraints
- Referential integrity constraints

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 constraints force that:

- in a relation with a key attribute, no two tuples can have identical values for key attributes.
- a key attribute cannot have NULL values.

Key constraints are also referred to as Entity Constraints.

Domain Constraints

Attributes have specific values in real-world scenario. For example, age can only be a positive integer. The same constraints have 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 numbers cannot contain a digit outside 0-9.

Referential Integrity Constraints

Referential integrity constraints work on the concept of Foreign Keys. A foreign key is a key attribute of a relation that can be referred in other relation.

Referential integrity constraint states that if a relation refers to a key attribute of a different or same relation, then that key element must exist.

CONSTRAINTS:

Constraints are used to specify rules for the data in a table. If there is any violation between the constraint and the data action, the action is aborted by the constraint. It can be specified when the table is created (using CREATE TABLE statement) or after the table is created (using ALTER TABLE statement).

INTEGRITY CONSTRAINTS

- NOT NULL - Ensures that a column cannot have a NULL value
- UNIQUE - Ensures that all values in a column are different
- PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- FOREIGN KEY - Uniquely identifies a row/record in another table
- CHECK - Ensures that all values in a column satisfies a specific condition

1. NOT NULL: When a column is defined as NOTNULL, then that column becomes a mandatory column. It implies that a value must be entered into the column if the record is to be accepted for storage in the table.

Syntax: CREATE TABLE Table_Name (column_name data_type (size) NOT NULL,);

Example: CREATE TABLE student (sno NUMBER(3)NOT NULL, name CHAR(10));

2. UNIQUE: The purpose of a unique key is to ensure that information in the column(s) is unique i.e. a value entered in column(s) defined in the unique constraint must not be repeated across the column(s). A table may have many unique keys.

Syntax: CREATE TABLE Table_Name(column_name data_type(size) UNIQUE, ….);

Example: CREATE TABLE student (sno NUMBER(3) UNIQUE, name CHAR(10));

3. CHECK: Specifies a condition that each row in the table must satisfy. To satisfy the constraint, each row in the table must make the condition either TRUE or unknown (due to a null).

Syntax: CREATE TABLE Table_Name(column_name data_type(size) CHECK(logical expression), ….);

Example: CREATE TABLE student (sno NUMBER (3), name CHAR(10),class CHAR(5),CHECK(class IN('CSE','CAD','VLSI'));

4. PRIMARY KEY: A field which is used to identify a record uniquely. A column or combination of columns can be created as primary key, which can be used as a reference from other tables. A table contains primary key is known as Master Table.

- It must uniquely identify each record in a table.
- It must contain unique values.
- It cannot be a null field.
- It cannot be multi port field.
- It should contain a minimum no. of fields necessary to be called unique.

Syntax: CREATE TABLE Table_Name(column_name data_type(size) PRIMARY KEY,….);

Example: CREATE TABLE faculty (fcode NUMBER(3) PRIMARY KEY, fname CHAR(10));

5. FOREIGN KEY: It is a table level constraint. We cannot add this at column level. To reference any primary key column from other table this constraint can be used. The table in

which the foreign key is defined is called a detail table. The table that defines the primary key and is referenced by the foreign key is called the master table.

Syntax: CREATE TABLE Table_Name(column_name data_type(size)
FOREIGN KEY(column_name) REFERENCES table_name);

Example: CREATE TABLE subject (scode NUMBER (3) PRIMARY KEY, subname CHAR(10),fcode NUMBER(3), FOREIGN KEY(fcode) REFERENCE faculty);

Defining integrity constraints in the alter table command:

Syntax: ALTER TABLE Table_Name ADD PRIMARY KEY (column_name);

Example: ALTER TABLE student ADD PRIMARY KEY (sno);

(Or)

Syntax: ALTER TABLE table_name ADD CONSTRAINT constraint_name
PRIMARY KEY(colname)

Example: ALTER TABLE student ADD CONSTRAINT SN PRIMARY KEY(SNO)

Dropping integrity constraints in the alter table command:

Syntax: ALTER TABLE Table_Name DROP constraint_name;

Example: ALTER TABLE student DROP PRIMARY KEY;

(or)

Syntax: ALTER TABLE student DROP CONSTRAINT constraint_name;

Example: ALTER TABLE student DROP CONSTRAINT SN;

6. DEFAULT : The DEFAULT constraint is used to insert a default value into a column. The default value will be added to all new records, if no other value is specified.

Syntax: CREATE TABLE Table_Name(col_name1,col_name2,col_name3 DEFAULT '<value>');

Example: CREATE TABLE student (sno NUMBER(3) UNIQUE, name CHAR(10),address VARCHAR(20) DEFAULT 'Visakhapatnam');

THE FORM OF A BASIC SQL QUERY

This section presents the syntax of a simple SQL query and explains its meaning through a conceptual evaluation strategy. A conceptual evaluation strategy is a way to evaluate the query that is intended to be easy to understand, rather than efficient. A DBMS would typically execute a query in a different and more efficient way.

We present a number of sample queries using the following schema:

Sailors(sid: integer, sname: string, rating: integer, age: real)

Boats(bid: integer, bname: string, color: string)

Reserves(sid: integer, bid: integer, day: date)

Sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

bid	*bname*	*color*
101	*Interlake*	*blue*
102	*Interlake*	*red*
103	*Clipper*	*green*
104	*Marine*	*red*

Figure 1An Instance S 3 of Sailors Figure 2 An Instance R2 of Reserves

(Q1) Find the names of sailors who have reserved boat 103.

This query can be written as follows:

$$\pi_{sname}((\sigma_{bid=103} Reserves) \bowtie Sailors)$$

$$\{\langle N\rangle \mid \exists I,T,A(\langle I,N,T,A\rangle \in Sailors \wedge \exists Ir,Br,D(\langle Ir,Br,D\rangle \in Reserves \wedge Ir = I \wedge Br = 103))\}$$

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND R.bid=103
```

(Q2) Find the names of sailors who have reserved a red boat.

$$\pi_{sname}((\sigma_{color='red'} Boats) \bowtie Reserves \bowtie Sailors)$$

$$\{\langle N\rangle \mid \exists I,T,A(\langle I,N,T,A\rangle \in Sailors \wedge \exists\langle I,Br,D\rangle \in Reserves \wedge \exists\langle Br,BN,'red'\rangle \in Boats)\}$$

SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’

(Q3) Find the colors of boats reserved by Lubber.

$$\pi_{color}((\sigma_{sname='Lubber'} Sailors) \bowtie Reserves \bowtie Boats)$$

SELECT B.color FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND S.sname = ‘Lubber’

(Q4) Find the names of sailors who have reserved at least one boat.

$$\pi_{sname}(Sailors \bowtie Reserves)$$

SELECT S.sname FROMSailors S, Reserves R WHERES.sid = R.sid

(Q5) Find the names of sailors who have reserved a red or a green boat.

$$\rho(Tempboats, (\sigma_{color='red'} Boats) \cup (\sigma_{color='green'} Boats))$$
$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

$$\rho(Tempboats, (\sigma_{color='red' \vee color='green'} Boats))$$
$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

(Q6) Find the names of sailors who have reserved a red and a green boat. It is tempting to try to do this by simply replacing by in the denition of Tempboats:

$$\rho(Tempboats2, (\sigma_{color='red'} Boats) \cap (\sigma_{color='green'} Boats))$$
$$\pi_{sname}(Tempboats2 \bowtie Reserves \bowtie Sailors)$$

(Q7) Find the names of sailors who have reserved at least two boats.

$$\rho(Reservations, \pi_{sid,sname,bid}(Sailors \bowtie Reserves))$$
$$\rho(Reservationpairs(1 \rightarrow sid1, 2 \rightarrow sname1, 3 \rightarrow bid1, 4 \rightarrow sid2,$$
$$5 \rightarrow sname2, 6 \rightarrow bid2), Reservations \times Reservations)$$
$$\pi_{sname1}\sigma_{(sid1=sid2)\wedge(bid1\neq bid2)} Reservationpairs$$

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge$$
$$\exists Br1, Br2, D1, D2(\langle I, Br1, D1\rangle \in Reserves \wedge \langle I, Br2, D2\rangle \in Reserves \wedge Br1 \neq Br2)$$

(Q8) Find the sids of sailors with age over 20 who have not reserved a red boat.

$$\pi_{sid}(\sigma_{age>20} Sailors) -$$
$$\pi_{sid}((\sigma_{color='red'} Boats) \bowtie Reserves \bowtie Sailors)$$

(Q9) Find the names of sailors who have reserved all boats. The use of the word all (or every) is a good indication that the division operation might be applicable:

$$\rho(Tempsids, (\pi_{sid,bid} Reserves)/(\pi_{bid} Boats))$$
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge$$
$$\forall B, BN, C(\neg(\langle B, BN, C\rangle \in Boats) \vee$$
$$(\exists\langle Ir, Br, D\rangle \in Reserves(I = Ir \wedge Br = B))))\}$$

(Q10) Find the names of sailors who have reserved all boats called Interlake.

$$\rho(Tempsids, (\pi_{sid,bid} Reserves)/(\pi_{bid}(\sigma_{bname='Interlake'} Boats)))$$
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

(Q11) Find all sailors with a rating above 7.

$$\{S \mid S \in Sailors \wedge S.rating > 7\}$$

$$\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in Sailors \wedge T > 7\}$$

SELECT S.sid, S.sname, S.rating, S.age FROM Sailors AS S WHERE S.rating > 7

(Q12) Find the names and ages of sailors with a rating above 7.

$$\{P \mid \exists S \in Sailors(S.rating > 7 \wedge P.name = S.sname \wedge P.age = S.age)\}$$

(Q13) Find the sailor name, boat id, and reservation date for each reservation.

$$\{P \mid \exists R \in Reserves \ \ \exists S \in Sailors$$
$$(R.sid = S.sid \wedge P.bid = R.bid \wedge P.day = R.day \wedge P.sname = S.sname)\}$$

(Q14) Find sailors who have reserved all red boats.

$$\{S \mid S \in Sailors \wedge \forall B \in Boats$$
$$(B.color = 'red' \Rightarrow (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

(Q15) Find the names and ages of all sailors.

SELECT DISTINCT S.sname, S.age

FROM Sailors S

SELECT DISTINCT S.sname, S.age FROM Sailors S

(Q16) Find the sids of sailors who have reserved a red boat.

SELECTR.sid FROMBoats B, Reserves R WHERE B.bid = R.bid AND B.color = ‘red’

The answer to this query with and without the keyword DISTINCT on instance S3 of Sailors is shown in Figures 5.4 and 5.5. The only difference is that the tuple for Horatio appears twice if DISTINCT is omitted; this is because there are two sailors called Horatio and age 35.

(Q5) Compute increments for the ratings of persons who have sailed two different boats on the same day.

SELECT S.sname, S.rating+1 AS rating FROM Sailors S, Reserves R1, Reserves R2

WHERE S.sid = R1.sid AND S.sid = R2.sid AND R1.day = R2.day AND R1.bid <> R2.bid

Also, each item in a qualification can be as general as expression1 = expression2.

SELECT S1.sname AS name1, S2.sname AS name2 FROM Sailors S1, Sailors S2 WHERE 2*S1.rating = S2.rating-1.

(Q6) Find the ages of sailors whose name begins and ends with B and has at least three

characters.

SELECT S.age FROM Sailors S WHERE S.sname LIKE ‘B %B’

The only such sailor is Bob, and his age is 63.5.

UNION, INTERSECT, AND EXCEPT

SQL provides three set-manipulation constructs that extend the basic query form pre-sented earlier. Since the answer to a query is a multiset of rows, it is natural to consider the use of operations such as union, intersection, and difference. SQL supports these operations under the names UNION, INTERSECT, and EXCEPT.

SQL also provides other set operations: IN (to

check if an element is in a given set),op ANY,op ALL(tocom-pare a value with the elements in a given set, using comparison operator op), and EXISTS (to check if a set is empty). IN and EXISTS can be prefixed by NOT, with the obvious modification to their meaning. We cover UNION, INTERSECT, and EXCEPT in this section. Consider the following query:

(Q1) Find the names of sailors who have reserved both a red and a green boat.

SELECT S.sname FROM Sailors S, Reserves R1, Boats B1, Reserves R2, Boats

B2 WHERE S.sid = R1.sid AND R1.bid = B1.bid AND S.sid = R2.sid AND R2.bid

= B2.bid AND B1.color=‘red’ AND B2.color = ‘green’

(Q2) Find the sids of all sailors who have reserved red boats but not green boats.

SELECT S.sid FROMSailors S, Reserves R, Boats B

WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = ‘red’ EXCEPT

SELECT S2.sid FROM Sailors S2, Reserves R2, Boats B2 WHERE

S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = ‘green’

NESTED QUERIES

A nested query is a query that has another query embedded within it; the embedded query is called a subquery.

(Q1) Find the names of sailors who have reserved boat 103.

SELECT S.sname FROM Sailors S WHERE S.sid IN (SELECT R.sid FROM Reserves R

WHERE R.bid = 103)

(Q2) Find the names of sailors who have reserved a red boat.

SELECT

S.sname FROM

Sailors S WHERE S.sid IN (SELECT R.sid FROM

Reserves R WHERE R.bid IN (SELECT B.bid FROM Boats B WHERE B.color =

_red‘)

(Q3) Find the names of sailors who have not reserved a red boat.

SELECT S.sname FROM Sailors S WHERE S.sid NOT IN (SELECT R.sid FROM Reserves R WHERE R.bid IN (SELECT B.bid FROM Boats B WHERE B.color = _red‘).

Correlated Nested Queries

In the nested queries that we have seen thus far, the inner subquery has been completely independent of the outer query:

(Q1) Find the names of sailors who have reserved boat number 103.

SELECT S.sname FROM Sailors S WHERE EXISTS (SELECT * FROM Reserves R WHERE R.bid = 103 AND R.sid = S.sid)

Set-Comparison Operators

(Q1) Find sailors whose rating is better than some sailor called Horatio.

SELECT S.sid FROMSailors S WHERE S.rating > ANY (SELECT S2.rating FROM Sailors S2 WHERE S2.sname = _Horatio‘)

(Q2) Find the sailors with the highest rating .

SELECT S.sid FROM Sailors S WHERE S.rating >= ALL (SELECT S2.rating FROM Sailors S2)

More Examples of Nested Queries

(Q1) Find the names of sailors who have reserved both a red and a green boat.

SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = _red‘ AND S.sid IN (SELECT S2.sid FROM Sailors S2,

Boats B2, Reserves R2 WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = _green‘)

Q9) Find the names of sailors who have reserved all boats.

SELECT S.sname FROM Sailors S WHERE NOT EXISTS ((SELECT B.bid FROM Boats B) EXCEPT (SELECT R.bid FROM

Reserves R WHERE R.sid = S.sid))

AGGREGATE OPERATORS

We now consider a powerful class of constructs for computing aggregate values such as MIN and SUM.

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.

2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.

3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.

4. MAX (A): The maximum value in the A column.

5. MIN (A): The minimum value in the A column.

(Q1) Find the average age of all sailors.

SELECT AVG (S.age) FROM

Sailors S

(Q2) Find the average age of sailors with a rating of 10.

SELECT AVG (S.age) FROM

Sailors S WHERE S.rating = 10

SELECT

S.sname, MAX (S.age) FROMSailors S

Q3) Count the number of sailors.

SELECT COUNT (*) FROM Sailors S

The GROUP BY and HAVING Clauses

we want to apply aggregate operations to each of a number of groups of rows in a relation, where

the number of groups depends on the relation instance (i.e., is not known in advance). (Q31) Find the age of the youngest sailor for each rating level.

SELECT MIN (S.age)

FROM Sailors S

WHERE S.rating = i

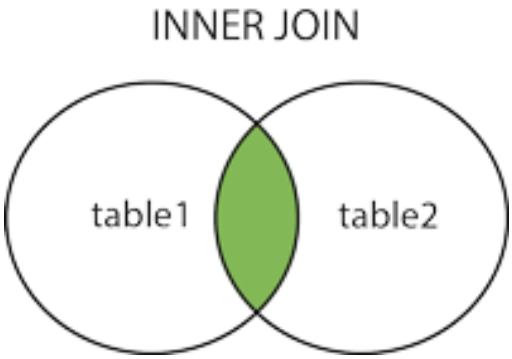
SQL JOIN

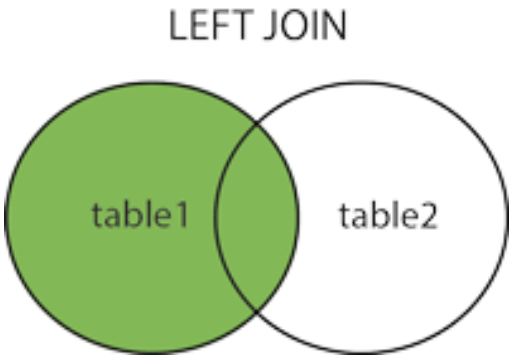
A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

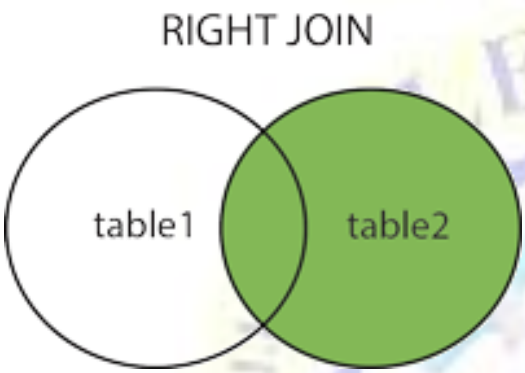
Different Types of SQL JOINs

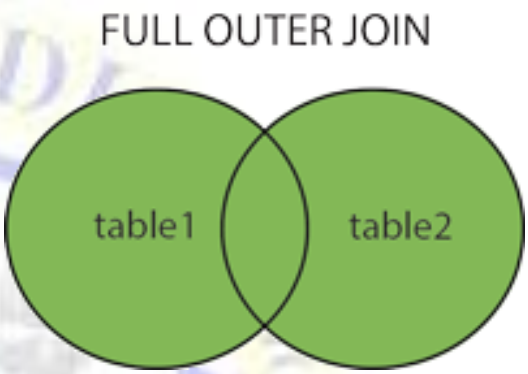
Here are the different types of the JOINs in SQL:

- (INNER) JOIN: Returns records that have matching values in both tables
- LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
- RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
- FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table









Sample Table

EMPLOYEE

EMP_ID	EMP_NAME	CITY	SALARY	AGE
1	Angelina	Chicago	200000	30
2	Robert	Austin	300000	26
3	Christian	Denver	100000	42
4	Kristen	Washington	500000	29
5	Russell	Los angels	200000	36
6	Marry	Canada	600000	48

PROJECT

PROJECT_NO	EMP_ID	DEPARTMENT
101	1	Testing
102	2	Development
103	3	Designing
104	4	Development

1. INNER JOIN

In SQL, INNER JOIN selects records that have matching values in both tables as long as the condition is satisfied. It returns the combination of all rows from both the tables where the condition satisfies.

Syntax

```
SELECT table1.column1, table1.column2, table2.column1,....
FROM table1
INNER JOIN table2
ON table1.matching_column = table2.matching_column;
```

Example Query

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT
FROM EMPLOYEE
INNER JOIN PROJECT
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

OUTPUT :

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development

2. LEFT JOIN

The SQL left join returns all the values from left table and the matching values from the right table. If there is no matching join value, it will return NULL.

Syntax

SELECT table1.column1, table1.column2, table2.column1,....

FROM table1

LEFT JOIN table2

ON table1.matching_column = table2.matching_column;

Example Query

SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT

FROM EMPLOYEE

LEFT JOIN PROJECT

ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;

Output

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development
Russell	NULL
Marry	NULL

3. RIGHT JOIN

In SQL, RIGHT JOIN returns all the values from the values from the rows of right table and the matched values from the left table. If there is no matching in both tables, it will return NULL.

Syntax

SELECT table1.column1, table1.column2, table2.column1,....

FROM table1

RIGHT JOIN table2 ON table1.matching_column = table2.matching_column;

Example Query

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT
FROM EMPLOYEE
RIGHT JOIN PROJECT
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

Output

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development
Christian	Designing
Kristen	Development

4. FULL JOIN

In SQL, FULL JOIN is the result of a combination of both left and right outer join. Join tables have all the records from both tables. It puts NULL on the place of matches not found.

Syntax

```
SELECT table1.column1, table1.column2, table2.column1,....
FROM table1
FULL JOIN table2
ON table1.matching_column = table2.matching_column;
```

Example Query

```
SELECT EMPLOYEE.EMP_NAME, PROJECT.DEPARTMENT
FROM EMPLOYEE
FULL JOIN PROJECT
ON PROJECT.EMP_ID = EMPLOYEE.EMP_ID;
```

Output

EMP_NAME	DEPARTMENT
Angelina	Testing
Robert	Development

Christian	Designing
Kristen	Development
Russell	NULL
Marry	NULL

Procedures:

"A procedures or function is a group or set of SQL and PL/SQL statements that perform a specific task."

A function and procedure is a named PL/SQL Block which is similar . The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.

A procedure is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages. A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure.

The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block. A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.

We can pass parameters to procedures in three ways :

Parameters Description

IN type

These types of parameters are used to send values to stored procedures.

OUT type These types of parameters are used to get values from stored procedures. This is similar to a return type in functions.

IN OUT type These types of parameters are used to send values and get values from stored procedures.

A procedure may or may not return any value.

Syntax:

CREATE [OR REPLACE] PROCEDURE procedure_name (<Argument> {IN, OUT, IN OUT}

<Datatype>,…)

IS

Declaration section<variable, constant> ;

BEGIN

Execution section

EXCEPTION

Exception section

END

IS - marks the beginning of the body of the procedure and is similar to DECLARE in anonymous

PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [ ] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

How to execute a Procedure?

There are two ways to execute a procedure :

From the SQL prompt : EXECUTE [or EXEC] procedure_name;

Within another procedure – simply use the procedure name : procedure_name;

Example:

create table named emp have two column id and salary with number datatype.

```
CREATE OR REPLACE PROCEDURE p1(id IN NUMBER, sal IN NUMBER)
AS
BEGIN
INSERT INTO emp VALUES(id, sal);
DBMD_OUTPUT.PUT_LINE('VALUE INSERTED.');
END;
/
```

What is a Stored Procedure?

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

Stored Procedure Syntax

CREATE PROCEDURE procedure_name

AS

sql_statement

GO;

Execute a Stored Procedure

EXEC procedure_name;

Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

Customer ID	Customer Name	Contact Name	Address	City	Postal Code	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-95822	Sweden

Stored Procedure Example

The following SQL statement creates a stored procedure named "SelectAllCustomers" that selects all records from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT * FROM Customers
GO;
```

Execute the stored procedure above as follows:

Example : EXEC SelectAllCustomers;

Stored Procedure With One Parameter

The following SQL statement creates a stored procedure that selects Customers from a particular City from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30)
AS
SELECT * FROM Customers WHERE City = @City
GO;
```

Execute the stored procedure above as follows:

Example: EXEC SelectAllCustomers @City = 'London';

Stored Procedure With Multiple Parameters

Setting up multiple parameters is very easy. Just list each parameter and the data type separated by a comma as shown below.

The following SQL statement creates a stored procedure that selects Customers from a particular City with a particular PostalCode from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30), @PostalCode nvarchar(10)
AS
SELECT * FROM Customers WHERE City = @City AND PostalCode = @PostalCode
GO;
```

Execute the stored procedure above as follows:

Example: EXEC SelectAllCustomers @City = 'London', @PostalCode = 'WA1 1DP';