Views

View - a pseudo table where data is derived from one or more base tables.

- Do not actually store data. Just display it.
- Formed with a defining query and given a name. The resulting view-table can be used like any other table for querying.
- can also be used for insertion/update, but there are some subtleties involved.

Syntax:

1. `CREATE VIEW <view name> [(colHeading1, ..., colHeadingN)] AS <query>` - creates a new view.

2. `CREATE OR REPLACE VIEW <view name> [(colHeading1, ..., colHeadingN)] AS <query>` - replaces the view with the `<view name>` if it exists, with the new one.

Examples:

1. Create a phone directory showing each employee’s name, department and phone number.

```
CREATE OR REPLACE VIEW PhoneDirectory (Department, Employee, Phone) AS
SELECT dept_name, RTRIM(emp_lname||','||emp_fname), emp_phone
FROM htopi.dept, htopi.employee;
```

View used in a single-table query: Display the directory ordered alphabetically by the department name, then employee name - uses the PhoneDirectory view.
SELECT *
FROM PhoneDirectory
ORDER BY Department, Employee;

View used in a multi-table query Display the part of the phone directory that lists only employees working in building 'H'.

SELECT Department, Employee, Phone
FROM PhoneDirectory, htopi.dept
WHERE Department = dept_name and dept_building = 'H'
ORDER BY Department, Employee;

Purposes of Views:

- A view can greatly simplify user perception of a complex database.
- A view can restrict access to data.
- A view can be used as a table storing intermediate results of some computation:

  Example: Display the names and number of dependents of all employees over 25 years of age who have more than average number of dependents in their department.

Solution 1 - uses a correlated subquery:

```
SELECT e1.emp_lname, e1.emp_dept,
MONTHS_BETWEEN(SysDate, e1.emp_dateofbirth)/12 as Age,
e1.emp_nbr_of_dependents
FROM htopi.employee e1
WHERE MONTHS_BETWEEN(SysDate, e1.emp_dateofbirth)/12 < 25 and
  e1.emp_nbr_of_dependents > (SELECT AVG(e2.emp_nbr_of_dependents)
                              FROM htopi.employee e2
                              WHERE e2.emp_dept = e1.emp_dept)
```

Note, we cannot display the average number of dependents in the same table.

Solution 1 - uses a view to store intermediate results: list of all departments with the average number of dependents of employees in the department.

```
CREATE OR REPLACE VIEW AvgDependents (DepartmentNumber, AvgNumDependents) AS
SELECT emp_dept, avg(emp_nbr_of_dependents)
FROM htopi.employee
GROUP BY emp_dept;
```
SELECT e1.emp_lname, e1.emp_dept,
MONTHS_BETWEEN(SysDate, e1.emp_dateofbirth)/12 as Age,
e1.emp_nbr_of_dependents, AvgNumDependents
FROM htopi.employee e1, AvgDependents
WHERE MONTHS_BETWEEN(SysDate, e1.emp_dateofbirth)/12 < 25 and
    DepartmentNumber = e1.emp_dept and
    e1.emp_nbr_of_dependents > AvgNumDependents;

Storing intermediate results in a view simplifies the query by eliminating the correlated subquery. Also, the resulting table includes a row that displays the average number of dependents in the department.

**Maintaining Data Integrity**

Data integrity (i.e. validity) constraints:

1. **Domain constraints**: the values in each of the columns of a relation have to come from a certain domain.

2. **Entity integrity**: no primary key value can be null. Primary key value should be unique.

3. Fundamental question: If the same data is specified in two different places in the database, are the values equal?

For example, consider the specification of a country *America* vs *USA* vs *U.S.A*.

Most importantly, **Referential integrity**: a foreign key value has to be either null or a valid primary key value in the relation to which the foreign key refers.

SQL Support for enforcing data integrity constraints: **Domain constraints** are enforced via

1. data types/length of columns,
2. CHECK operator that every inserted value belongs to an appropriate range.

**Syntax**: within CREATE TABLE, when defining a column, CHECK (<test>)

**Example:**
CREATE TABLE Student (  
  StNo CHAR(4),  
  StLName CHAR(20),  
  StGender CHAR(1)  
    CHECK (StGender in ('F','M'))  
  StAdvisor CHAR(4),  
)  

alternative syntax: add a named constraint in the end of CREATE TABLE.

CREATE TABLE Student (  
  StNo CHAR(4),  
  StLName CHAR(20),  
  ...  
  CONSTRAINT checkGender CHECK (StGender in ('F','M'))  
)  

Entity constraints are enforced via the PRIMARY KEY constraint that guarantees that the values of primary key are unique and not NULL.

1. PRIMARY KEY constraint
   Example:

   CREATE TABLE Student (  
   StNo CHAR(4) PRIMARY KEY,  
   ...  
   )  

2. PRIMARY KEY constraint, alternative syntax
   Syntax: CONSTRAINT <constraint-name> PRIMARY KEY (<column(s)>),
   Example:

   CREATE TABLE Student (  
   StNo CHAR(4),  
   ...  
   CONSTRAINT PK_Student PRIMARY KEY (StNo)  
   )  

Referential integrity constraints are enforced via the FOREIGN KEY constraint.
1. FOREIGN KEY constraint

**Syntax:**

`CONSTRAINT <constraint-name> FOREIGN KEY (<column>) REFERENCES <table>(column-other-table)`

**Example:**

```sql
CREATE TABLE Student (  
  StNo CHAR(4)  
  StLName CHAR(20),  
  ....  
  StAdvisor CHAR(4),  
  CONSTRAINT FK_StudentFaculty FOREIGN KEY (StAdvisor)  
  REFERENCES Faculty(FacNo)  
)
```

2. What happens if the primary key values change (`Advisor's ID is changed`) or the row that is referenced is deleted (`Advisor leaves the college and his/her record is deleted`)?

SQL provides an option of automatically resetting the foreign key to NULL when the row that is referenced by it is deleted.

**Example:** The value of StAdvisor will be set to NULL when the record about the faculty with the FacNo=StAdvisor is deleted.

```sql
CREATE TABLE Student (  
  ....  
  StAdvisor CHAR(4),  
  CONSTRAINT FK_StudentFaculty FOREIGN KEY StAdvisor  
  REFERENCES Faculty(FacNo) ON DELETE SET NULL  
)
```

To check whether referential integrity has been violated, run a query verifying that for every value of foreign key in one table, there is a row with the same value of the primary key in the referred table.

**Example:** Check if every Student record refers to an existing Faculty record. Display those student ID's and their StAdvisor value for which referential integrity is violated, i.e. no record with such StAdvisor value exists in the Faculty table.

```sql
SELECT StNo, StAdvisor  
FROM Student  
WHERE NOT EXISTS (  
  SELECT *  
  FROM Faculty  
  WHERE FacNo = StAdvisor  
)
```
Here’s an alternative way to get the same result:

```sql
SELECT StNo, StAdvisor
FROM Student
WHERE StAdvisor NOT IN (
    SELECT FacNo
    FROM Faculty
)
```

DROP TABLE <table> deletes a table.

**INSERT, UPDATE, DELETE**

**INSERT**

**Syntax:** INSERT INTO <table> VALUES (<column1 value>, <column2 value>, ...)

The second form of the INSERT command is used to insert rows where some of the column data is unknown (NULL). This form of the INSERT command requires that you specify the names of the columns for which data are being stored.

```sql
INSERT INTO Emp (emp_ssn, emp_lname, emp_fname)
    VALUES ('999111111', 'Bock', 'Douglas')
```

**DELETE**

The DELETE command is perhaps the simplest of the SQL statements. It removes one or more rows from a table. Multiple table delete operations are not allowed in SQL.

**Syntax:**

DELETE FROM <table_name>
[WHERE <condition>];

Since the WHERE clause is optional, you can easily delete all rows from a table by omitting a WHERE clause since the WHERE clause limits the scope of the DELETE operation.

For example, the DELETE FROM command shown here removes all rows in the Student table. DELETE FROM Student;

**UPDATE**
Values stored in individual columns of selected rows can be modified (updated) with the UPDATE command. The UPDATE command changes data in the table (not the table structure).

Syntax:

UPDATE table
SET column = expression [,column = expression]... 
[WHERE condition];

COMMIT and ROLLBACK

COMMIT

INSERT, UPDATE, and DELETE commands are not committed to the database until the COMMIT statement is executed. COMMIT is a transaction managing command that confirms operations to the database on the server (closing Oracle also acts as a confirmation of the commands entered).

ROLLBACK

ROLLBACK (ROLL) can be issued to cancel any database operations since the most recent COMMIT. Like COMMIT, ROLLBACK is also a transaction managing command; however, it cancels operations instead of confirming them.

Example: constrExample.sql