CHAPTER 5. Relational Database Management Systems and SQL

5.1 Brief History of SQL in Relational Database Systems

- The ***relational model*** was first proposed ***by E.F. Codd*** in ***1970***.

- A ***language*** now called ***SQL***, originally spelled SEQUEL, was presented in a series of papers ***starting in 1974***.

- An early ***commercial relational database management system***, ***ORACLE***, was developed in the ***late 1970s*** using SQL as its language.

- IBM's first commercially available relational database management system, ***SQL/DS*** was announced in ***1981***.

- IBM's ***DB2***, also using SQL as its language, was released in ***1983***.

- Both American National Standards Institute (***ANSI***) and the International Standards Organization (***ISO***) adopted ***SQL as a standard language for relational databases*** and published ***specifications for the SQL language***, which is usually called ***SQL1 in 1986***.

- A major revision, ***SQL2***, was adopted by both ANSI and ISO in ***1992***.

- The ***current SQL3*** standard was developed over the time, with major parts published in ***1999***, ***2003, 2006, and 2008***.

- This chapter will ***focus*** on the most widely used strictly relational features that are ***available in most relational DBMSs***.

- ***Different implementations of SQL vary slightly from the syntax presented here, but the basic notions are the same. Commands given in this chapter generally use the Oracle syntax, and may need slight modifications to run on other DBMSs***.

5.2 Architecture of a Relational Database Management System

- Relational database management systems support the standard ***three-level architecture*** for database. (see Figure 5.1 on p155 of the textbook)

- The logical level for relational database consists of ***base tables*** that are physically stored. These tables are created using a ***CREATE TABLE*** command.

- A base table can have ***any number of indexes*** either created ***by the system itself*** or created using the ***CREATE INDEX*** command. ***An index is used to speed up retrieval of records based on the value in one or more columns***. Most relational database management systems use ***B trees*** or ***B+ trees*** for indexes.

- ***On the physical level, the base tables and their indexes are represented in files***. The physical representation of the tables may not correspond exactly to our notion of a base table as a two-dimensional object. The DBMS, not the OS, controls the internal structure of both the data files and the indexes.

- The user is generally ***unaware of what indexes exist***, and has ***no control over which index Will be used*** in locating a record.

- Once the base tables have been created, ***"views" for users*** can be created using the ***CREATE VIEW*** command. Relational views can be either ***"windows" into base tables*** or ***"virtual tables"***, ***not permanently stored***, but created when the user needs to access them. Users are unaware of the fact that their views are not physically stored in table form.

- One of the most useful features of a relational database is that it permits ***dynamic database definitions***: can create new tables, add columns to old ones, create new indexes, define views, and drop any of these objects at any time.

5.3 Defining the Database: SQL DDL

5.3.2 CREATE TABLE

- (Form)

CREATE TABLE [schema-name.] base-table-name (colname datatype [column constraints]
[, colname datatype [column constraints]]
. . .
[table constaraints]
[storage specifications]);

- Examples

(ex) **CREATE TABLE** Customer (cno CHAR(3), balance NUMBER(5));

CREATE TABLE Employee (

```
        SSN     CHAR(9)          NOT NULL,
        NAME    VARCHAR2(30)     NOT NULL,
        AGE     INT,
        PRIMARY KEY(SSN)
);

CREATE TABLE Works_On (

        ESSN    CHAR(9)          NOT NULL,
        PNO     INT              NOT NULL,
        HOURS   DECIMAL(3,1)     NOT NULL,

        PRIMARY KEY(ESSN, PNO),
        FOREIGN KEY(ESSN) REFERENCES EMPLOYEE(SSN),
        FOREIGN KEY(PNO) REFERENCES PROJECT(PNUMBER)

);
```

(ex) Figure 5.2 on p159 of the textbook for the following schema:

Student (stuId, lastName, firstName, major, credits)
Faculty (facId, name, department, rank)
Class (classNumber, facId, schedule, room)
Enroll (classNumber, stuId, grad)

- base-table name is a ***user-supplied name (an identifier)*** for the table. ***No SQL key words*** may be used, and ***the table name must be unique within the database***.

- For ***each column***, specify a ***name*** that is ***unique*** within the table, and a ***data type***.

- In ***Oracle***, ***identifiers*** must be ***at most 30 characters long***, ***begin with an alphabetic character***, and ***contain only alphanumeric characters*** (but _, $, and # are permitted).

- Either uppercase or lowercase letters may be used, but ***Oracle will always display them as uppercase***.

- The ***maximum number of columns*** for an Oracle table is ***1000***.

- Each line ends with a ***comma***, except the last, which ends with a ***semicolon***.

- If the optional storage specification is not specified, the database management system will create a default space for the table.

- ***The available data types vary from DBMS to DBMS***.

- ***VARCHAR2 (n)*** stores ***varying length strings of maximum size n bytes***. A size ***n*** up to 4000 bytes ***must be specified***.

- ***CHAR (n)*** can be used ***for fixed-length strings*** with the maximum allowable size of 2000 bytes.

- For ***fixed-point numbers*** the data type ***NUMBER (p, s)*** is used. ***p*** is the total number of digits and ***s*** is the number of digit to the right of the decimal point, if any. ***For integers, the s is omitted***. Floating-Point numbers can also specified as NUMBER, with no precision (p) or scale (s) specified, or as FLOAT (p).

- Values of ***DATE*** type are entered using the ***default format 'dd-mon-yy' as in '02-DEC-11'***, where the ***month is represented using a three-letter*** abbreviation.

- The database management system has facilities to ***enforce data correctness***. The relational model uses ***integrity constraints*** to protect the correctness of the database, ***allowing only legal instances to be created***.

- In a ***CREATE TABLE*** command, ***optional constraints*** can and should be added, both at the ***column level (a.k.a. in-line constraints)*** and at the ***table level (a.k.a. out-of-line constraints)***.

- The ***column (or in-line) constraints*** include options to specify ***NULL/NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, REF, CHECK, and DEFAULT*** for any column, immediately after the specification of the column name and data type.

- If the ***primary key is not composite***, it is also possible to specify PRIMARY KEY as a column constraint, simply by adding the words ***PRIMARY KEY after the data type for column***.
(ex) stuId VARCHAR2 (6) PRIMARY KEY,

- The specification of PRIMARY KEY in SQL carries an ***implicit NOT NULL constraint as well as a UNIQUE constraint.*** It is also desirable to specify ***NOT NULL and/or UNIQUE for candidate keys***.

- The ***CHECK*** constraint can be used to specify a condition that the rows of the table are not permitted to violate, in order to verify that values provided for attributes are appropriate.

- We can also specify a ***default value*** for a column ***if we wish to do so***.

- We ***can optionally provide a name for any constraint***. If we do not, the system will automatically assign a name. However, ***a user-defined name is preferable***, since it gives us an opportunity to choose a meaningful name, which is ***useful if we wish to modify it later***.

  (ex) credits ***NUMBER(3)*** ***DEFAULT*** 0 ***CONSTRAINT*** Student_credits_cc ***CHECK***
  ((credits >= 0) AND (credits < 150));

- ***Table constraints appear after all the columns have been declared***, and can include the specification of a primary key, foreign key, uniqueness, references, checks, and general constraints that can be expressed as conditions to be checked, but not ***NOT NULL, which is always a column constraint***.

- If the primary key is a ***composite***, it ***must be identified using a table constraint*** rather than a column constraint.

- The ***FOREIGN KEY*** constraint requires that we identify the referenced table where the column or column combination appears.

  (ex) **CONSTRAINT** Class_facId_fk **FOREIGN KEY**(facId) **REFERENCES** Faculty (facId)

- In an out-of-line constraint we must use the keyword ***CONSTRAINT***, which can be optionally followed by an identifier.

- The SQL standard allow us to specify what is to be done with records containing the foreign key values when the records they relate to are deleted in their home table.

  (ex) **CONSTRAINT** Class_facId_fk **FOREIGN KEY**(facId) **REFERENCES** Faculty (facId) **ON DELETE CASCADE**;

  ON DELETE CASCADE: delete all class records for that faculty member
  ON DELETE SET NULL: set the facId in the class record to a null value
  No Specification: not allow the deletion of a Faculty record

- The table uniqueness constraint mechanism can be used to specify that the values in a combination of columns must be unique.

(ex) **CONSTRAINT** Class_Schedule_room_uk **UNIQUE** (schedule, room)

(NOTE) 5.3.3 CREATE INDEX
5.3.4 ALTER TABLE, RENAME TABLE
5.3.5 DROP Statements **will be discussed later.**

5.4 Manipulating the Database: SQL DML

- The SQL ***DML*** statements are ***SELECT***, ***UPDATE***, ***INSERT***, and ***DELETE***.

5.4.1 Introduction to the SELCT Statement

- The SELECT statement is used for retrieval of data.

(Form)
```
SELECT [DISTICNT] col-name [AS newname], [, col-name …] …
FROM    table-name [alias] [, table-name] …
[WHERE predicate]
[GROUP BY col-name [, col-name] … [HAVING predicate] ]
or,
[ORDER BY col-name [, col-name] …];
```

(NOTE) Consider The University Database (FIGURE 5.4 on p171 of the textbook) for Examples.

Student (StuId, lastName, firstName, major, credits)
Faculty (facId, name, department, rank)
Class (classNumber, facId, schedule, room)
Enroll (stuID, classNumber, grade)

- Example 1. Simple Retrieval with Condition

Question: Get names, IDs, and number of credits of all Math majors.

SQL Query →
```
SELECT lastName, firstName, stuId, credits
FROM   Student
WHERE  major = 'Math';
```

- Example 2. Use of * for "all columns"

Question: Get all information about CSC Faculty.

SQL Query ➔ SELECT *
FROM Faculty
WHERE department = 'CSC';

or

SELECT facId, name, department, rank
FROM Faculty
WHERE department = 'CSC';

- Example 3. Retrieval without Condition, Use of "Distinct," Use of Qualified Names

Question: Get the course number of all courses in which students are enrolled.

SQL Query ➔ SELECT classNumber
FROM Enroll;

To eliminate the duplicates, we need to use "DISTICT" option.
➔ SELECT DISTINCT classNumber
FROM Enroll;

In any retrieval, especially if there is a possibility of confusion because of the same column name appears on two different tables
➔ SELECT DISTINCT Enroll.classNumber
FROM Enroll;

- Example 4. Retrieving an Entire Table

Question: Get all information about all students.

SQL Query ➔ SELECT *
FROM Students;

- Example 5. Use of "ORDER BY" and AS

Question: Get names and IDs of all Faculty members, arranged in alphabetical order by name.
Call the resulting columns FacultyName and FacultyNumber

SQL Query ➔ SELECT name AS FacultyName, facId AS FacultyNumber
FROM Faculty

```
ORDER BY name;
```

We could break the "tie" by giving a minor order

```
➔ SELECT    name AS FacultyName, facId AS FacultyNumber
  FROM      Faculty
  ORDER BY  name, department;
```

(Note) ASC (: default) or DESC

- Example 6. Use of Multiple Conditions

Question: Get names of all math majors who have more than 30 credits

```
SQL Query ➔ SELECT   lastName, firstName
            FROM     Student
            WHERE    major = 'Math' AND credits > 30;
```

5.4.2 SELECT Using Multiple Tables

- Example 7. Natural Join

Question: Find IDs and names of all students taking ART103A.

```
SQL Query ➔ SELECT   Enroll.stuId, lastName, firstName
            FROM     Student, Enroll
            WHERE    classNumber = 'ART103A' AND Enroll.stuId = Student.stuId;
```

• We could have written "**Student.stuId**" instead of "**Enroll.stuId**".

• We did not need to use the qualified name for classNumber because it does not appear on the Student table.

• Without the condition, Enroll.stuId = Student.stuId the result will be a Cartesian product.

• Note that some relational DBMSs allow the phrase, "**FROM Enroll NATURAL JOIN Student**"

- Example 8. Natural Join with Ordering

Question: Find stuId and grade of all students taking any course taught by the Faculty member whose facId is F110. Arrange in order by stuId.

```
SQL Query ➔ SELECT     stuId, grade
```

```
FROM        Class, Enroll
WHERE       facId = 'F110' AND Class.classNumber = Enroll.classNumber
ORDER BY    stuId ASC;
```

- Example 9. Natural Join of Three Tables

Question: Find course numbers and the names and majors of all students enrolled in the courses taught by Faculty member F110.

SQL Query →
```
SELECT      Enroll.classNumber, lastName, firstName, major
FROM        Class, Enroll, Student
WHERE       facId = 'F110' AND Class.classNumber = Enroll.classNumber
              AND Enroll.stuId = Student.stuId;
```

• SQL ***ignores the order in which the tables are named*** in the FROM line.

• Most sophisticated relational database management systems choose ***which table to use first and which condition to check first***, using an ***optimizer to identify the most efficient method*** of accomplishing any retrieval before choosing a plan.

- Example 10. Use of Aliases

Question: Get a list of all courses that meet in the same room, with their schedules and room numbers.

SQL Query →
```
SELECT      A.classNumber, A.schedule, A.room, B.classNumber, B.schedule
FROM        Class A, Class B
WHERE       A.room = B.room AND A.classNumber < B.classNumber;
```

• We added the second condition "A.classNumber < B.classNumber" to keep every classfrom being included, since every class obviously satisfies the requirement that it meets in the same room as itself. It also keeps records with the two classes reserved from appearing.

• Incidentally, we can introduce aliases in any SELECT, even when they are not required.

- Example 11. Other Joins

Question: Find all combinations of students and faculty where the student's major is different from the faculty member's department.

SQL Query →
```
SELECT      stuId, S.lastName, S.firstName, major, facId, F.name, department
```

```
              FROM       Student S, Faculty F
              WHERE      S.major <> F.department;
```

• We might use any type of predicate as the condition for the join. If we want to compare two columns, however, they must have the same domains. (note) "major" and "department" have the same domain.

- Example 12. Using a Subquery with Equality

Question: Find the numbers of all the courses taught by Byrne of the Math department.

```
SQL Query ➔ SELECT      classNumber
              FROM       Class
              WHERE      facId = ( SELECT    facId
                                   FROM      Faculty
                                   WHERE     name = 'Byrne' AND department = 'Math' );
```

• This can also be done by using a natural join

```
➔ SELECT      classNumber
   FROM       Class, Faculty
   WHERE      Class.facId = Faculty.facId AND name = 'Byrne' AND department = 'Math';
```

• When you write a subquery involving two tables, you name only ***one table in each select***. ***The query to be done first, the subquery, is the one in parentheses***, following the first WHERE line. The main query is performed using the result of the subquery.

- Example 13. Subquery Using 'IN'

Question: Find the names and IDs of all Faculty members who teach a class in Room H221.

```
SQL Query ➔ SELECT      name, facId
              FROM       Faculty
              WHERE      facId IN ( SELECT    facId
                                    FROM      Class
                                    WHERE     room = 'H221');
```

• This can also be done by using a natural join

```
➔ SELECT      name, Faculty.facId
```

```
FROM       Class, Faculty
WHERE      Class.facId = Faculty.facId AND room = 'H221';
```

- Example 14. Nested Subqueries

Question: Get an alphabetical list of names and IDs of all students in any class taught by F110.

SQL Query ➔
```
SELECT      lastName, firstName, stuId
FROM        Student
WHERE       stuId IN ( SELECT    stuId
                       FROM      Enroll
                       WHERE     classNumber    IN
                     ( SELECT    classNumber
                       FROM      class
                       WHERE     facId = 'F110' ) );
ORDER BY    lastName, firstName ASC;
```

• Note that the ordering refers to the final result, not to any intermediate steps.

• We could have performed either part of the operation as a natural join and the other part as a subquery, mixing both methods.

- Example 15. Query Using EXISTS

Question: Find the names of all students enrolled in CSC201A.

SQL Query ➔
```
SELECT      lastName, firstName
FROM        Student
WHERE       EXISTS  ( SELECT    *
                      FROM      Enroll
                      WHERE     Enroll.stuId = Student.stuId AND
                                classNumber = 'CSC201A' );
```

• This can also be done by using a join or a subquery with IN.

• Notice we needed to ***use the name of the main query table ("Student") in the subquery*** to express the condition "Student.stuId = Enroll.stuId". ***In general, we avoid mentioning a table not listed in the FROM for that particular query***, but it is necessary and permissible to do so in this case. This form is called ***correlated*** subquery.

- Example 16. Query Using NOT EXISTS

Question: Find the names of all students who are not enrolled in CSC201A.

```
SQL Query → SELECT    lastName, firstName
            FROM      Student
            WHERE     NOT EXISTS  ( SELECT    *
                                    FROM      Enroll
                                    WHERE     Enroll.stuId = Student.stuId AND
                                              classNumber = 'CSC201A' );
```

• Unlike the previous example, we cannot readily express this using a join or an IN subquery.

5.4.3 SELECT with Other Operators

- Example 17. Query using UNION

Question: Get IDs of all Faculty who are assigned to the History department or who teach in Room H221.

```
SQL Query → SELECT    facId
            FROM      Faculty
            WHERE     department = 'History'

            UNION

            SELECT    facId
            FROM      Class
            WHERE     room = 'H221';
```

• In addition to UNION, SQL supports the operations INTERSECT (for set intersection), MINUS (for set difference), and UNION ALL(for UNION allowing duplicate rows).

- Example 18(a). Using Aggregate Functions

Question: Find the total number of students enrolled in ART103A.

```
SQL Query → SELECT    COUNT ( DISTINCT stuId)
            FROM      Enroll
            WHERE     classNumber = 'ART103A';
```

- SQL has five commonly used built-in aggregate functions:
  COUNT : returns the ***number of values*** in the column,
  SUM : returns the ***sum of the values*** in the column,
  AVG : returns the ***mean of the values*** in the column,
  MAX : returns the ***largest value*** in the column,
  MIN : returns the ***smallest value*** in the column.

- The built-in functions ***operate on a single column of a table***. Each of them ***eliminates null values first***, and ***operates only on the remaining non-null values***.

  (Note) ***COUNT (*)*** is a special use of the COUNT. It counts all the rows of a table, ***regardless of whether null values or duplicate values occur***.

- Additional Function Examples:

Example 18(b) Find the number of departments that have faculty in them.

```
SQL Query ➔ SELECT     COUNT ( DISTINCT department )
            FROM       Faculty;
```

Example 18(c) Find the sum of all the credits that history majors have.

```
SQL Query ➔ SELECT     SUM ( credits )
            FROM       Student
            WHERE      major = 'History';
```

Example 18(d) Find the average number of credits students have.

```
SQL Query ➔ SELECT     AVG ( credits )
            FROM       Student;
```

Example 18(e) Find the student with the largest number of credits.

```
SQL Query ➔ SELECT     stuId, lastName, firstName
            FROM       Student
            WHERE      credits = ( SELECT     MAX ( credits )
                                   FROM       Student );
```

Example 18(f) Find the ID of the student(s) with the highest grade in any course.

```
SQL Query → SELECT    stuId
            FROM      Enroll
            WHERE     grade = ( SELECT    MIN ( grade )
                                FROM      Enroll );
```

Example 18(g) Find names and the IDs of students who have less than the average number of credits.

```
SQL Query → SELECT    lastName, firstName, stuId
            FROM      Student
            WHERE     credits < ( SELECT    AVG ( credits )
                                  FROM      Student );
```

- Example 19(a). Using an Expression and a String Constant

Question: Assuming each class is three credits list, for each student, the number of classes he or she has completed.

```
SQL Query → SELECT    stuId, 'Number of classes =', credits/3
            FROM      Student;
```

- Example 20(b). Use of GROUP BY

Question: For each class, show the number of students enrolled.

```
SQL Query → SELECT    classNumber, COUNT (*)
            FROM      Enroll
            GROUP BY  classNumber;
```

• Note that we could have used COUNT (DISTINCT stuId) in place of COUNT (*) in this query.

- Example 21(a). Use of HAVING

Question: Find all courses in which fewer than three students are enrolled.

```
SQL Query → SELECT    classNumber
            FROM      Enroll
            GROUP BY  classNumber
```

HAVING COUNT (*) < 3;

• ***HAVING*** is used to determine ***which groups have some quality***, just as WHERE is used with tuples to determine which records have some quality. You are ***not permitted to use HAVING without a GROUP BY***, and the predicate in the HAVING line ***must have a single value for each group***.

- Example 22(a). Use of LIKE and NOT LIKE

Question: Get details of all MTH courses.

```
SQL Query → SELECT    *
            FROM      Class
            WHERE     classNumber LIKE 'MTH%';
```

• SQL allows us to use ***LIKE*** in the predicate ***to show a pattern string*** for character fields:

% stands for any sequence of characters of any length >= 0.

_ stands for any single character.

(Examples)

• classNumber LIKE 'MTH%'
• stuId LIKE 'S_ _ _ _'
• schedule LIKE '%9'
• classNumber LIKE '%101%'
• NOT LIKE 'A%'

• **SELECT** sname
FROM Sailors
WHERE Sailors.sname **LIKE** 'B_%B'

→ BoB, B...B, BB, Bob
O O X X

(Note) Although SQL is ***not case sensitive for commands***,
SQL is ***case sensitive for data***.

- Example 23. Use of NULL

Question: Find the stuId and classNumber of all students whose grades in that course are missing.

```
SQL Query → SELECT    classNumber, stuId
```

```
FROM     Enroll
WHERE    grade IS NULL;
```

- A null grade is considered to have "unknown" as a value, so ***it is impossible to judge whether it is equal to or not equal to another grade***. If we put the condition "WHERE grad <> 'A' AND grad <> 'B' AND grad <> 'C' AND grad <> 'D' AND grad <> 'F'" we would get an empty table back. SQL uses the logical expression, ***columnname IS [NOT] NULL***.

- Notice that it is ***illegal to write "WHERE grade = NULL***. Also, the ***WHERE line is the only one on which NULL can appear in a SELECT statement***.

5.4.4 Operators for Updating: UPDATE, INSERT, DELETE

- The ***UPDATE*** operator is used ***to change values in records already stored*** in a table.

```
UPDATE   tablename
SET      columnname = expression [, columnname = expression] . . .
[WHERE   predicate];
```

(Example 1) Updating a Single Field of One Record

Operation: Change the major of S1020 to Music.

```
SQL Command:  UPDATE   Student
              SET      major = 'Music'
              WHERE    stuId = 'S1020';
```

(Example 2) Updating Several Fields of One Record

Operation: Change Tanaka's department to MIS and rank to Assistant.

```
SQL Command:  UPDATE   Faculty
              SET      department = 'MIS', rank = 'Assistant'
              WHERE    name = 'Tanaka';
```

(Example 3) Updating Using NULL

Operation: Change the major of S1013 from Math to NULL.

```
SQL Command: UPDATE    Student
             SET       major = NULL
             WHERE     stuId = 'S1013';
```

(Example 4) Updating Several Records

Operation: Change grades of all students in CSC201A to A.

```
SQL Command: UPDATE    Enroll
             SET       grade = 'A'
             WHERE     classNumber = 'CSC201A';
```

(Example 5) Updating All Records

Operation: Give all students three extra credits.

```
SQL Command: UPDATE    Student
             SET       credits = credits + 3;
```

(Example 6) Updating with a Subquery

Operation: Change the room to B220 for all courses taught by Tanaka.

```
SQL Command: UPDATE    Class
             SET       room = 'B220'
             WHERE     facId = ( SELECT  facId
                                 FROM    Facult
                                 WHERE   name = 'Tanaka');
```

- The ***INSERT*** operator is used ***to put new records into a table***.

```
INSERT
INTO     tablename [(colname [, colname]. . .)]
VALUES   (value [, value]. . .);
```

- Note that the ***column names are optional if we are inserting values for all columns in their proper order***.

(Example 1) Inserting a Single Record, with All Fields Specified

Operation: Insert a new Faculty record with ID of F330, name of Jones, department of CSC, and rank of Instructor.

SQL Command:
```
INSERT
INTO      Faculty (facId, name, department, rank)
VALUES    ('F330', 'Jones', 'CSC', 'Instructor');
```

(Example 2) Inserting a Single Record, without Specifying Fields

Operation: Insert a new student record with ID of S1030, name of Alice Hunt, major of art, and 12 credits.

SQL Command:
```
INSERT
INTO      Student
VALUES    ('S1030', 'Hunt', 'Alice', 'Art', 12);
```

(Example 3) Inserting a Record with Null Value in a Field

Operation: Insert a new student record with ID of S1031, name of Maria Bono, zero credits, and ***no major***.

SQL Command:
```
INSERT
INTO      Student (lastName, firstName, stuId, credits)
VALUES    ('Bono', 'Maria', 'S1031', 0);
```

(note) - We rearranged the field names, but there is no confusion.
- ***major will be set to null***, since we excluded it from the field list in the INTO line.

(Example 4) Inserting Multiple Records

Operation: Create and fill a new table that shows each course and the number of students enrolled in it.

SQL Command:
```
CREATE    TABLE   Enrollment (
          classNumber   VARCHAR2 (7)   NOT   NULL,
          Students      NUMBER (3) );
```

```
INSERT
INTO        Enrollment (classNumber, Students)
            SELECT      classNumber, COUNT(*)
            FROM        Enroll
            GROUP BY    classNumber;
```

- ***Enrollment*** table ***can be updated*** as needed, but it ***will not be updated automatically*** when the ***Enroll*** table is updated.

(Example 5) Inserting DATE values and SYSDATE

```
INSERT
INTO     EMPLOYEE (empId, lastName, firstName, birthdate, hireDate)
VALUES  (1001, 'Hynes', 'Susan', '15-OCT-1985', '01-JUN-2010');
```

```
INSERT
INTO     EMPLOYEE (empId, lastName, firstName, birthdate, hireDate)
VALUES  (1001, 'Hynes', 'Susan', '15-OCT-1985', SYSDATE);
```

- TRUNC (SYSDATE) sets the time part 00:00.
- Oracle has several date/time functions including TO_CHAR and TO_DATE (see examples 5(c) (on p201) and 5(d) (on p202)).

- The ***DELETE*** is used ***to erase records***.

```
DELETE
FROM     tablename
WHERE    predicate ;
```

(Example 1) Deleting a Single Record

Operation: Erase the record of student S1020.

```
SQL Command: DELETE
             FROM     Student
             WHERE    stuId = 'S1020';
```

(Example 2) Deleting Several Records

Operation: Erase all enrollment records for student S1020.

SQL Command:
```
DELETE
FROM     Enroll
WHERE    stuId = 'S1020';
```

(Example 3) Deleting All Records from a Table

Operation: Erase all the class records.

SQL Command:
```
DELETE
FROM     Class;
```

- The delete command will not work on the ***Class*** table unless we first delete the ***Enroll*** records for any students registered in the class. ***Why?***

- Assuming that we have deleted the ***Enroll*** records, then this would remove all records from the ***Class*** table, but ***its structure would remain***, so could add new records to it any time.

(Example 4) DELETE with a Subquery

Operation: Erase all enrollment records for Owen McCarthy.

SQL Command:
```
DELETE
FROM     Enroll
WHERE    stuId = ( SELECT   stuId
                   FROM     Student
                   WHERE    lastName = 'McCarthy'
                            AND firstName = 'Owen' );
```

5.4.5 Creating and Using Views

- ***Views*** are an important tool for providing users with ***a simple, customized environment*** and for ***hiding data***.

- A ***relational view*** is either a window into a base table or virtual table derived from one or more underlying base tables. ***It does not exist in storage in the sense that the base tables do***.

- The view is ***dynamically produced*** as the user works with it. Views allow a dynamic external model to be created for the users easily.

- The reasons for providing views rather than allowing all users to work with base tables:

  • Views allow different users ***to see the data in different forms***.
  • The view mechanism ***provides a simple authorization control device***. View users are unaware of, and cannot access, certain data items.
  • Views can ***free users from complicated DML operations***.
  • If the database is restructured on the logical level, the view can be used ***to keep the user's model constant***.

- (Form) CREATE VIEW *viewname* [(*viewcolname* [, *viewcolname*)] …
  AS SELECT *colname* [, *colname*] …
  FROM *basetablename* [, *basetablename*] …
  WHERE *condition;*

- ***Column names in the view can be different from the corresponding column names in the base tables***, but they must obey the same rules of construction. ***If we choose to make them the same, we need not specify them twice***, so we leave out the *viewcolname* specifications.

(Example 1) Choosing a Vertical and Horizontal Subset of a Table

```
CREATE VIEW HISTMAJ (last, first, StudentId)
AS SELECT lastName, firstName, stuId
     FROM Student
     WHERE major = 'History';
```

(Note) The user of this view need not know the actual column names.

(Example 2) Choosing a Vertical Subset of a Table

```
CREATE VIEW ClassLoc
AS SELECT classNumber, schedule, room
     FROM Class;
```

(Example 3) A View Using Two Tables

```
CREATE VIEW ClassList
```

```
AS  SELECT  Student.stuId, lastName, firstName
     FROM    Enroll, Student
     WHERE   classNumber = 'CSC101' AND Enroll.stuId = Student.stuId;
```

(Example 4) A View of a View

```
CREATE  VIEW  ClassLoc2
AS  SELECT  classNumber, room
     FROM    ClassLoc;
```

(Example 5) A View Using a Function

- In the SELECT statement in the AS line we can include built-in functions and GROUP BY options.

```
CREATE  VIEW  ClassCount  (classNumber, TotCount)
AS  SELECT     classNumber, COUNT (*)
     FROM       Enroll
     GROUP BY  classNumber;
```

or

```
CREATE  VIEW  ClassCount2
AS  SELECT     classNumber, COUNT (*) AS TotCount
     FROM       Enroll
     GROUP BY  classNumber;
```

(Example 6) Operations on View

- Once a view is created, the user can write SELECT statements to retrieve data through the view.
- Users can write SQL queries that refer to joins, ordering, grouping, built-in functions, and so on, of views just as if they were operating on base tables. ***Since the SELECT operation does not change the underlying base tables***, there is ***no restriction on allowing authorized users to perform SELECT with view***.

```
SELECT   *
FROM     ClassLoc
WHERE    room LIKE 'H%';
```

- INSERT, DELETE, and UPDATE can present problems with views.

(ex) Consider a view of student records, ***StudentVw1 (lastName, firstName, major, credits)***. If we were permitted to insert records, ***<u>any records created through this view would actually be Student records</u>***, but would not contain stuId, which is a key of the Student table. ➔ ***<u>have to reject</u>***

However, if we had the following view,
StudentVww2 (stuId, lastName, firstName, credits) we should have no problem to inserting records, since we would be inserting **Student** records with a null major field, which is ***<u>allowable</u>***. ➔

```
INSERT
INTO      StudentVw2
VALUES    ('S1040', 'Levine', 'Adam', 30);
```

(Note) However, the system should actually insert the record into the ***<u>Student</u>*** table.

ClassCount (classNumber, TotCount) was meant to be ***<u>a dynamic summary</u>*** of the ***Enroll*** table rather than being a row and column subset of the table. ***<u>It would not make sense for us to permit new ClassCount records to be inserted</u>***, since these do not correspond to individual rows and columns of a base table.

- The problem we have identified for INSERT apply with minor changes to UPDATE and DELETE as well. As a general rule, ***<u>these three operations can be performed on views that consist of actual rows and columns of underlying base tables, provided the primary key is included in the view, and no other constraints are violated</u>***.