Database Design

SQL

Database Design

1. *Requirements Analysis:* in this step, we must point out

2. *Conceptual Database Design:* develop a high-level description of the data to be stored in the database, along with the constraints that are known to hold on this data.

3. *Logical Database Design*: convert the conceptual database design into a database schema within the data model of the chosen DBMS.

4. ***Schema Refinement*:** the schemas developed in step 3 are analyzed for potential problems, then are ***normalized***.

5. ***Physical Database Design****:* potential workloads and access patterns are simulated to identify potential weaknesses in the conceptual database. This will often cause the creation of additional indices and/or clustering relations.

6. *Security Design:* Different user groups are identified and their different roles are analyzed so that access patterns to the data can be defined.

SQL

- SQL is a complete relational database language in the sense that it contains both a data definition language (DDL) and a data manipulation language (DML).
- We'll examine components of both parts of SQL.
- If you use Microsoft Access, for example, you'll need to know less about the DDL side of SQL than you will if you use Oracle 9i or MySQL.
- The table on the following pages summarize the commands in the DDL portion of SQL. The entries in the table do not correspond to the order in which you will use the commands, but simply give a quick summary of those available. The table does not contain a complete listing of the commands in the DDL portion of SQL.

SQL Syntax Notation

Notation	Description
CAPITALS	Required SQL command keyword
italics	An end-user provided parameter – normally required
{a | b | ... }	A mandatory parameter, use one from option list
[...]	An optional parameter – everything in brackets is optional
tablename	The name of a table
column	The name of an attribute in a table
data type	A valid data type definition
constraint	A valid constraint definition
condition	A valid conditional expression – evaluates to true or false
columnlist	One or more column names or expressions separated by commas
tablelist	One or more table names separated by commas
conditionlist	One or more conditional expressions separated by logical operators
expression	A simple value (e.g., 76 or ‘married’) or a formula (e.g., price-10)

Some SQL statements

- CREATE TABLE
- DROP TABLE ...
- ALTER TABLE ... ADD/REMOVE ...
- INSERT INTO ... VALUES ...
- DELETE FROM ... WHERE ...
- UPDATE ... SET ... WHERE ...
- SELECT ... FROM ... WHERE ...

Summary of SQL DDL Commands

Command or Option	Description
CREATE SCHEMA AUTHORIZATION	Creates a database schema
CREATE TABLE	Creates a new table in the user's DB schema
NOT NULL	Constraint that ensures a column will not have null values
UNIQUE	Constraint that ensures a column will not have duplicate values
PRIMARY KEY	Defines a primary key for a table
FOREIGN KEY	Defines a foreign key for a table
DEFAULT	Defines a default value for a column (when no value is given)
CHECK	Constraint used to validate data in a column
CREATE INDEX	Creates an index for a table
CREATE VIEW	Creates a dynamic subset of rows/columns from 1 or more tables
ALTER TABLE	Modifies a table's definition: adds/deletes/updates attributes or constraints
DROP TABLE	Permanently deletes a table (and thus its data) from the DB schema
DROP INDEX	Permanently deletes an index
DROP VIEW	Permanently deletes a view

The DDL Component Of SQL

Before you can use a Relational Database System, two tasks must be completed:

(1) Create the database structure,

(2) Create the tables that will hold the end-user data.

Completion of the first task involves the construction of the physical files that hold the database. The RDBMS will automatically create the data dictionary tables and create a default database administrator (DBA).

Creating the physical files requires interaction between the host OS and the RDBMS. Therefore, creating the database structure is the one feature that tends to differ substantially from one RDBMS to another.

Most RDBMS vendors use SQL that deviates very little from ANSI standard SQL. Nevertheless, you might occasionally encounter minor syntactic differences. For example, most RDBMSs require that any SQL command be ended with a semicolon. However, some SQL implementations do not use a semicolon.

Creating Table Structures Using SQL

The CREATE TABLE syntax is:

```
CREATE TABLE tablename (
    column1     data type     [constraint] [,
    column2     data type     [constraint] ]  [,
   PRIMARY KEY (column1 [,column2] )]  [,
   FOREIGN KEY (column1 [,column2] ) REFERENCES tablename ] [,
  CONSTRAINT constraint ] ) ;
```

Data types

INT or INTEGER.

REAL or FLOAT.

CHAR(n) = string of characters with fixed length.

VARCHAR(n) = string of characters with variable and at most n characters of length.

NUMERIC(precision, decimal)

DATE = format 'yyyy-mm-dd'

TIME = format 'hh:mm:ss[.ss…]'.

DATETIME or TIMESTAMP. SQL with format TIMESTAMP 'yyyy-mm-dd hh:mm:ss[.ss…]'.

Declaring Keys

PRIMARY KEY or UNIQUE

SQL only allow to index by using PRIMARY KEY.

SQL does not allow PRIMARY KEY to be null. UNIQUE is different.

Example:

```
CREATE TABLE Store(
   Name CHAR(20),
   Beer VARCHAR(20),
   Price REAL,
   PRIMARY KEY(Name, Beer) );
```

```
CREATE TABLE Store(
   Name CHAR(20),
   Beer VARCHAR(20),
   Price REAL,
   UNIQUE (Name, beer));
```

create table …

```
CREATE TABLE Company(
   name VARCHAR(20) PRIMARY KEY,
   country VARCHAR(20),
   employees INT,
   for_profit CHAR(1));
```

Multi-column Keys

- This makes name a key:

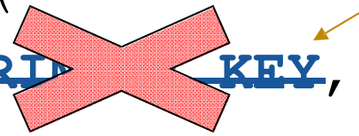
```
CREATE TABLE Company(
   name VARCHAR(20) PRIMARY KEY,
   country VARCHAR(20),
   employees INT,
   for_profit BOOLEAN);
```

- How can we make a key on name & country?

Multi-column Keys

- Syntax change if a primary key has multiple columns:

```
CREATE TABLE Company(
   name VARCHAR(20) PRIMARY KEY,
   country VARCHAR(20),
   employees INT,
   for_profit BOOLEAN,
   PRIMARY KEY (name, country));
```



goes away

added

Multi-column Keys (2)

- Likewise for secondary keys:

```
CREATE TABLE Company(
   name VARCHAR(20) UNIQUE,
   country VARCHAR(20),
   employees INT,
   for_profit BOOLEAN,
   UNIQUE (name, country));
```

goes away

added

Multi-column Keys (3)

- This makes manufacturer a foreign key:

```
CREATE TABLE Product(
   name VARCHAR(20),
   price DECIMAL(10,2),
   manufacturer VARCHAR(20)
     REFERENCES Company(name));
```

good idea to include target column name

Multi-column Keys (3)

- Similar syntax for foreign keys:

```
CREATE TABLE Product(
   name VARCHAR(20),
   price DECIMAL(10,2),
   manu_name VARCHAR(20),
   manu_co VARCHAR(20),
   FOREIGN KEY (manu_name, manu_co)
     REFERENCES Company(name, country));
```

now need both name & country

added

Declaring Keys using UNIQUE

What's the difference?

```
CREATE TABLE Store(
        Name CHAR(20)  UNIQUE,
        Beer VARCHAR(20)  UNIQUE,
        Price REAL );
```

```
CREATE TABLE Store(
        Name CHAR(20),
        Beer VARCHAR(20),
        Price REAL,
        UNIQUE (Name, beer));
```

Foreign Keys

```
CREATE TABLE Beer (
        Name CHAR(20) PRIMARY KEY,
        Manufacturer  CHAR(20) );
CREATE TABLE Store(
        Name CHAR(20),
        Beer CHAR(20) REFERENCES Beer(Name),
        Price REAL );
```

Or:

```
CREATE TABLE Store(
        Name CHAR(20),
        Beer CHAR(20),
        Price REAL,
        FOREIGN KEY Beer REFERENCES Beer(Name) );
```

Connection Rules

Add ON [DELETE, UPDATE] [CASCADE, SET NULL] during foreign key declaration.

Example:

```
CREATE TABLE Store(
        Name CHAR(20),
        Beer CHAR(20),
        Price REAL,
        FOREIGN KEY Beer REFERENCES Beer(Name)
            ON DELETE SET NULL
            ON UPDATE CASCADE );
```

Constraints

CHECK (condition): condition is checked when the values are changed (INSERT, UPDATE).

Example:

```
CREATE TABLE Store (
        Name CHAR(20),
        Beer CHAR(20) CHECK(
                        Beer IN (“Hanoi”, “Saigon”, “Huda”, “Laos”) ),
        Price REAL CHECK (Price<= $5.00)
);
```

Constraints

```
CREATE TABLE Store(
        Name CHAR(20),
        Beer CHAR(20),
        Price REAL,
        CHECK (Name = ‘Lan Chín' OR Price <= $5.00)
);
```

Only Lan Chín is allowed to sell beer having prices more than $5.00

Some other constraints

NOT NULL = attributes are not allowed to have null status.

DEFAULT value = set a defaulted value for the attributes if you do not input anything.

Example:

```
CREATE TABLE Customer (
        Name CHAR(30) PRIMARY KEY,
        Address CHAR(50)      DEFAULT '123 Sesame St',
        Phone CHAR(16) );
```

Key attributes are set default NOT NULL.

Some Notes On Table Creation

Support for referential constraints varies widely from RDBMS to RDBMS.

- MS Access, SQL Server, and Oracle support ON DELETE CASCADE.
- MS Access and SQL Server, support ON UPDATE CASCADE.
- Oracle does not support ON UPDATE CASCADE.
- Oracle supports SET NULL.
- MS Access and SQL Server do not support SET NULL.
- MS Access does not support ON DELETE CASCADE or ON UPDATE CASCADE at the SQL line level, however, it does support it through the relationship window interface (see Day 16 notes).

ALTER TABLE

All changes in the table structure are made using the ALTER TABLE command, followed by a keyword that produces the specific change you want to make.

There are three options for the keyword: ADD, MODIFY, and DROP.

ADD enables you to add a column to a table.

MODIFY enables you to change a column's characteristics.

DROP allows you to delete a column from a table Most RDBMSs do not allow you to delete a column from a table, unless that column does not contain any values, because such an action may delete crucial data that are used by other tables.

ALTER TABLE

The basic syntax:

```
ALTER TABLE   tablename
 {ADD | MODIFY} ( columnname  datatype
                  [ {ADD | MODIFY} columnname  datatype] );
```

The ALTER TABLE command can also be used to add table constraints. In that case the syntax would be as follows:

```
ALTER TABLE   tablename
 ADD  constraint   [ ADD constraint];
```

ALTER TABLE

You can also use the ALTER TABLE command to remove a column or table constraint. The basic syntax of this form of the ALTER command is:

```
ALTER TABLE  tablename
 DROP { PRIMARY KEY |
        COLUMN  columnname |
        CONSTRAINT  constraintname } ;
```

Notice that when removing a constraint, you need to specify the name given to the constraint. This is one reason why it is always advisable to name the constraints in the CREATE TABLE or ALTER TABLE statements.

Changing a Column's Data Type

ALTER TABLE command can be used to change the data type of a column.

For example, suppose we wanted to change the data type of V_CODE attribute in the PRODUCT table from integer to character. The following SQL command would accomplish this task:

```
ALTER TABLE  PRODUCT

    MODFIY (V_CODE CHAR(5));
```

Most RDBMSs does not allow you to change the data type of an attribute unless the column to be changed is empty.

Changing a Column's Data Characteristics

If the column to be changes already contains data, you can make any changes in the column's characteristics if those changes do not alter the data type.

For example, if we wanted to increase the width of the P_PRICE column from 8 digits to 9 digits, we would need to issue the following command:

```
ALTER TABLE  PRODUCT

    MODFIY (P_PRICE DECIMAL(9,2));
```

Many RDBMSs place restrictions on the types of changes to column characteristics that can occur. For example, Oracle will allow you to widen a column but will not allow you to narrow a column.

Adding a Column to a Table

You can alter an existing table by adding one or more columns.

For example, suppose that we want to add a column to the PRODUCT table called P_SALECODE that will allow us to decide if products that have been in inventory for a certain length of time should be placed on sale. Let's assume that the P_SALECODE entries will be 1, 2, or 3, and we're not going to do arithmetic on the column so we'll make it a character.

```
ALTER TABLE  PRODUCT

    ADD (P_SALECODE CHAR(1));
```

When adding a column, be careful not to include the NOT NULL clause for the new column. Doing so will cause an error message because when adding a new column to a table that already has rows, the existing rows will default to a value of null for the new column. Therefore, it is not possible to add the NOT NULL clause for this new column.

You can add the NOT NULL clause to the table structure after all the data for the new column has been entered and the column no longer contains nulls.

Deleting a Column from a Table

Occasionally, you may want to modify a table by deleting a column.

Suppose that we want to delete the V_ORDER attribute from the VENDOR table. To accomplish this task we would use the following SQL command:

```
ALTER TABLE  VENDOR
    DROP COLUMN V_ORDER;
```

As before, some RDBMSs will impose restrictions on the deletion of an attribute. For example, most RDBMSs will not allow you to delete attributes that are involved in foreign key relationships, nor may you delete an attribute of a table that contains only that one attribute.

Adding Primary and Foreign Key Designations

Although we were able to create a new table based on an existing table in the previous example, the process is not without its problems. Primarily, the new PART table was created without the inclusion of the integrity rules of the table on which it was based. In particular, there is no primary key designated for the table shown on the previous page.

To define the primary key for this table we need to use the ALTER TABLE command as shown below:

```
ALTER TABLE PRODUCT
    ADD PRIMARY KEY (P_CODE);
```

Adding Primary and Foreign Key Designations

Quite aside from the fact that the integrity rules are not automatically transferred to a new table that derives its data from one or more other tables, there are several other scenarios that would leave you without entity and referential integrity enforcement.

For example, you might have simply forgotten to define the primary and foreign keys when you created the tables.

The integrity rules can be reestablished via the ALTER command as shown below:

```
ALTER TABLE PRODUCT
   ADD PRIMARY KEY(P_CODE)
   ADD FOREIGN KEY(V_CODE) REFERENCES VENDOR;
```

Deleting a Table From the Database

- A table can be deleted from the database through the DROP command as shown below:

```
DROP TABLE PRODUCT
```

- A table can only be dropped from a database if it is not participating as the “1” side of any relationships. If you attempt to delete such a table, the RDMS will issue an error message to indicate that a foreign key integrity violation has occurred.

The DML Portion of SQL

The DML portion of SQL can be viewed as two separate components which overlap in certain areas. The two components are the non-query DML commands and the query DML commands.

Non-query DML commands allow you to populate tables (INSERT), modify data in tables (UPDATE), delete data from tables (DELETE) as well as make changes permanent (COMMIT) and undo changes (to some extent with ROLLBACK).

The query DML commands essentially consist of a single statement (SELECT) with many different optional clauses.

Summary of SQL DML Commands

Command or Option	Description
INSERT	Inserts row(s) into a table
UPDATE	Modifies attribute values in one or more of a table's rows
DELETE	Deletes one or more rows from a table
COMMIT	Permanently saves data changes
ROLLBACK	Restores data to their original values
SELECT	Selects attributes from rows in one or more tables or views
WHERE	Restricts the selection of rows based on a conditional expression
GROUP BY	Groups the selected rows based on one or more attributes
HAVING	Restricts the selection of grouped rows based on a condition
ORDER BY	Orders the selected rows
Comparison Operators	
=, <, >, <=, >=, <>	Used in conditional expressions
Logical Operators	
AND, OR, NOT	Used in conditional expressions

Summary of SQL DML Commands (cont.)

Command or Option	Description
Special Operators	*used in conditional expressions*
BETWEEN	Checks whether an attributes values is within a range
IS NULL	Checks whether an attribute value is null
LIKE	Checks whether an attribute value matches a given string pattern
IN	Checks whether an attribute value matches any value within a value list
EXISTS	Checks if a subquery returns any rows or not
DISTINCT	Limits values to unique values, i.e., eliminates duplicates
Aggregate Functions	*used with SELECT to return mathematical summaries on columns*
COUNT	Returns the number of rows with non-null values for a given column
MIN	Returns the minimum attribute value found in a given column
MAX	Returns the maximum attribute value found in a given column
SUM	Returns the sum of all values for a given column
AVG	Returns the average of all values for a given column

Adding Rows To Tables

SQL requires the use of the INSERT INTO command to enter data into a table.

The syntax of the INSERT INTO command is:

```
INSERT INTO tablename
        VALUES (value1, value 2, ...value n);
```

Example - Adding Rows To Tables

In order to add the two rows to the VENDOR table shown below, we would need to execute the following two SQL commands:

```
INSERT INTO VENDOR
    VALUES (21225, 'Bryson, Inc.', 'Smithson', '615', '223-3234', 'TN', 'Y');
INSERT INTO VENDOR
    VALUES (21226, 'SuperLoo, Inc.', 'Flushing', '904', '215-8995', 'FL', 'N');
```

If an attribute in a row has no value (i.e., is null) you would use the following syntax to enter the row into the table:

```
INSERT INTO PRODUCT
    VALUES ('23114-AA',  'Sledge hammer, 12 lb.', '02-Jan-02', 8, 5, 14.40, 0.05, NULL);
```

Example - Adding Rows With Optional Values To Tables

There may be occasions on which more than one attribute is optional (i.e., can be null). Rather than declaring each attribute as NULL in the INSERT command, you can just indicate the attributes that have required values.

This is done by listing the attribute names for which values are being inserted inside parentheses after the table name.

For the purposes of example, suppose that only the P_CODE and P_DESCRIPT are required attributes in the PRODUCT table. If this is the case, then either of the following syntactic forms could be used:

```
INSERT INTO PRODUCT
    VALUES ('23114-AA', 'Sledge hammer, 12 lb.', NULL, NULL, NULL, NULL, NULL, NULL);
-or-
INSERT INTO PRODUCT(P_CODE, P_DESCRIPT)
    VALUES('23114-AA', 'Sledge hammer, 12 lb.');
```

Deleting Rows From A Table

It is easy to use SQL to delete a row from a table. This is handled via the DELETE command.

The syntax of the DELETE command is:

```
DELETE FROM tablename
        [WHERE conditionlist ];
```

To delete a row of a table based on a primary key value you would use a command such as:

```
DELETE FROM PRODUCT
    WHERE P_CODE = '23114-AA';
```

Deleting Rows From A Table (cont.)

Deletion also works to remove potentially multiple rows from a table.

For example, suppose that we want to delete every product from the PRODUCT table where the value of the P_MIN attribute is equal to 5. To accomplish this you would issue the following command:

```
DELETE FROM PRODUCT
    WHERE P_MIN = 5;
```

DELETE is a set-oriented command. This means that since the WHERE condition is optional, if it is not specified, all rows from the specified table will be deleted!

Updating the Rows of a Table

To modify the data within a table the UPDATE command is used.

The syntax of the UPDATE command is:

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

Notice that the WHERE condition is optional in the UPDATE command. If the WHERE condition is omitted, then the update is applied to all rows of the specified table.

Updating the Rows of a Table (cont.)

If more than one attribute is to be updated in a row, the updates are separated by commas:

```
UPDATE PRODUCT
SET P_INDATE = ’18-JAN-2004’, P_PRICE = 16.99, P_MIN = 10
    WHERE P_CODE = ’13-Q2/P2’;
```

Saving Changes to a Table

Any changes made to the table contents are not physically saved into the underlying physical table (the file system) until a COMMIT command has been executed.

Depending on the sophistication of the system on which you are working, if the power should fail during the updating of a table (or database in general), before the COMMIT command was executed, your modifications are simply lost. More sophisticated systems will be able to recover from such disasters, but for small PC-based systems you'd better have a UPS installed!

The syntax for the COMMIT command is:

```
COMMIT [ tablename ];

-or-

COMMIT;   //saves all changes made in any modified tables
```

Restoring Table Contents

If you have not yet used the COMMIT command to permanently store the changes in the database, you can restore the database to its previous state (i.e., the one that was the result of the last COMMIT) with the ROLLBACK command.

ROLLBACK undoes any changes made and brings the data back to the values that existed before the changes were made.

The syntax for the ROLLBACK command is:

```
ROLLBACK;
```

MS Access does not support ROLLBACK! Some RDBMSs like Oracle automatically COMMIT data changes when issuing DDL commands, so ROLLBACK won't do anything on these systems.

ROLLBACK rolls back everything since the last COMMIT, which means that even changes that you might not want undone will be if no commit has been issued.

Query Portion of the DML of SQL

The query portion of the DML of SQL consists of a single command called the SELECT command.
The syntax of the SELECT command is:

```
SELECT [ ALL | DISTINCT] columnlist
FROM tablelist
[ WHERE condition ]
[GROUP BY  columnlist ]
[HAVING condition ]
[ORDER BY columnlist ];
```

We'll examine most of the features of the SELECT command, starting with simple queries and working our way toward more complex queries. I'll continue to use the same database that we've developed in this set of notes.

Simple Selection Queries in SQL

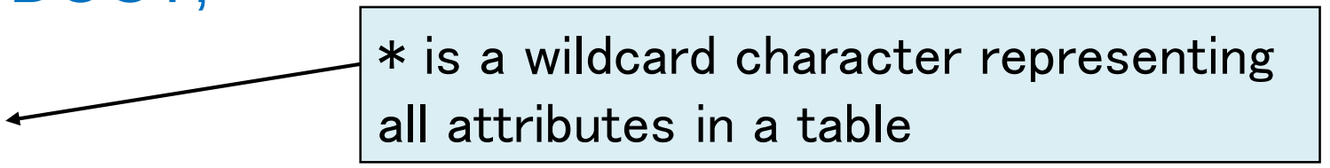
Perhaps the simplest query to form is that which retrieves every row from some specified table.

For example, suppose that we wanted to list every attribute value in every row of the PRODUCT table. In other words, to view this table. The following command will accomplish this task:

```
SELECT    P_CODE, P_DESCRIPT, P_INDATE, P_ONHAND, P_MIN,
          P_PRICE, P_DISCOUNT, V_CODE
FROM PRODUCT;
```

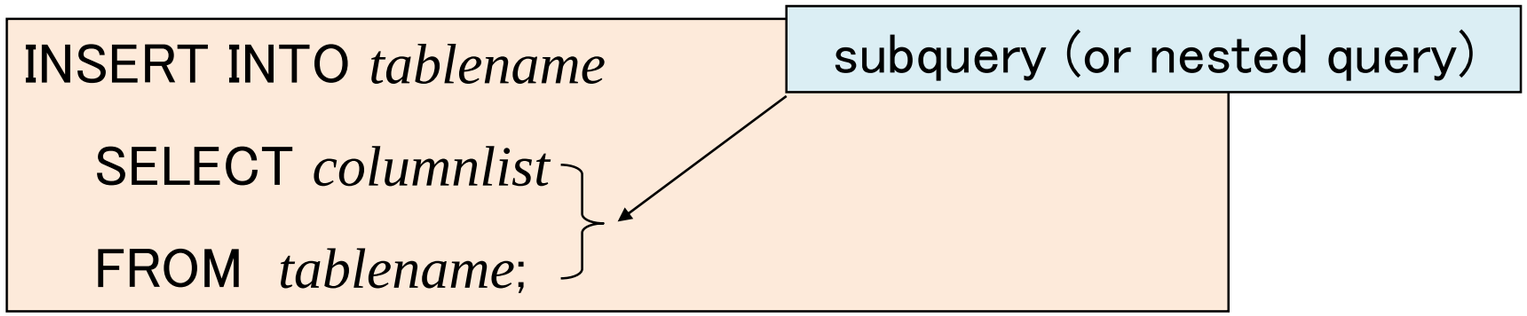
-or-

```
SELECT *
FROM PRODUCT;
```



Inserting Table Rows with a Select Subquery

Although this is technically a non-query DML operation, it also includes a query command, so I've included an example here before we move on to more complex query expressions.
SQL allows you to enter rows into a table using the data from another table as the populating basis. The syntax for this type of insert command is:



The inner query is always executed first by the RDBMS and the values extracted by the inner query will be used as input to the outer query (in this case the INSERT command). The values returned by the inner query must match the attributes and data types of the table in the INSERT statement.

Selection Queries With Conditional Restrictions

You can select partial table contents by placing restrictions on the rows to be included in the result. This is accomplished using the WHERE clause:

```
SELECT columnlist
FROM tablelist
WHERE conditionlist ;
```

The SELECT statement will retrieve all rows that match the specified condition(s) specified in the WHERE clause.

For example:

```
SELECT P_DESCRIPT, P_INDATE, P_PRICE, V_CODE
    FROM PRODUCT
    WHERE V_CODE = 21344;
```

Using Computed Columns and Column Aliases

Suppose that your query needs to determine a value which is not physically stored in the database but is calculated from data that is in the database.

For example, let's suppose that we want to determine the total value of each of the products currently held in inventory. Logically, this determination requires the multiplication of each product's quantity on hand by its current price. The SQL query for this is shown below and the resulting output is on the next page.

```
SELECT  P_DESCRIPT, P_ONHAND,  P_PRICE, P_ONHAND * P_PRICE   AS TOTVALUE
FROM PRODUCT;
```

SQL will accept any valid expression in the computed columns that apply to the attributes in any of the tables specified in the FROM clause. Note that Access will automatically add an Expr label to all computed columns. Oracle uses the actual expression to label the computed column.

Standard SQL permits the use of aliases for any column in a SELECT statement. The alias for any column is preceded by the keyword AS.

Using The Logical Operators AND, OR, and NOT

In the real world, a search of data normally involves multiple conditions. SQL allows you to express multiple conditions in a single query through the use of logical operators.

The logical operators supported by SQL are: AND, OR, and NOT.

Suppose you want a list of the table of PRODUCTS for either

V_CODE = 21344 or V_CODE = 24288. The SQL query to accomplish this is:

```
SELECT  P_DESCRIPT,  P_INDATE, P_PRICE, V_CODE
FROM PRODUCT
WHERE
      V_CODE = 21344  OR  V_CODE = 24288;
```

Exercise 1

Given the relation as follows

Product (Name, price, type)

where price is presented in VND

Write SQL commands to list all products where price is presented in thousand VND, and note those beer products with more than 100.000 VND “expensive”, and remaining products as “normal”.

Exercise 2

Given following relations:

Employee (NameE, address, City)

Work (NameE, NameC, salary)

Company (NameC, City, phone)

Write SQL commands to list all employee's names living in the same city with their company.

Exercise 3

Given following relations:

Staff (Staff#, Name, address, phone)

Manage (staff#, manager#)

Write SQL commands to:

- List all names of staffs having the same address.
- Find all IDs of staffs having same names with their managers.
- Find all IDs of managers who manage staffs named “Binh”.

CASE

CASE statement evaluates a list of conditions one-by-one and returns a value based on that condition

Syntax 1:

```
CASE expression
        WHEN expression1 THEN result1
        WHEN expression2 THEN result2
        ...
        ELSE result
END
```

Example:

Student (S#, Name, Class, GPA)

```
SELECT Name,
  CASE GPA
    WHEN >3 THEN “Excellence”
    WHEN <=3 OR >=2 THEN “Good”
    ELSE “Bad”
  END AS Classification
FROM Student;
```

CASE

Syntax 2:

```
CASE
        WHEN ComparsionCondition1 THEN result1
        WHEN ComparsionCondition THEN result2
        ...
        ELSE result
END
```

Special Operators in SQL

ANSI standard SQL allows the use of special operators in conjunction with the WHERE clause. These special operators include:

- BETWEEN – Used to check whether an attribute value is within a range.
- IS NULL – Used to determine if an attribute value is null.
- LIKE – Used to match an attribute value to a string pattern. Many wildcard options are available.
- IN – Used to determine if an attribute value is within a list of values.
- EXISTS – Used to determine if a subquery returns an empty set or not.

The BETWEEN Special Operator

- Suppose that we want to see a listing for all products whose prices are between $50 and $100. The BETWEEN operator can be used for this query expression.

```
SELECT *
        FROM PRODUCT
        WHERE P_PRICE BETWEEN 50.00 AND 100.00;
```

- If your RDBMS does not support BETWEEN you would need to express this query as:

```
SELECT *
        FROM PRODUCT
        WHERE P_PRICE > 50.00 AND P_PRICE < 100.00;
```

The IS NULL Special Operator

Suppose that we want to see a listing for all products that do not currently have a vendor assigned, i.e., V_CODE = null. The null entries could be found with the following query expression.

```
SELECT P_CODE, P_DESCRIPT, V_CODE
    FROM PRODUCT
    WHERE V_CODE IS NULL;
```

NOTE: SQL uses a special operator for testing for nulls. You cannot use a condition such as V_CODE = NULL. The reason is that NULL is technically not a “value”, but a special property of an attribute that represents precisely the absence of any value at all.

Exercise

Giving following relation:

Film (Name, Year, Length, Type, Studio, Director)

Where “Length” is presented in minutes.

What is the difference between these commands:

```
SELECT *
FROM Film;
```

```
SELECT *
FROM Film
WHERE Length <= 120 OR Length > 120;
```

Second command does not output records with Length is NULL.

The LIKE Special Operator

The LIKE special operator is used in conjunction with wildcards to find patterns within string attributes.

Standard SQL allows you to use the percent sign (%) and underscore (_) wildcard characters to make matches when the entire string is not known.

% means any and all following characters are eligible.

Example: 'M%' includes Mark, Marci, M-234x, etc.

_ means any one character may be substituted for the underscore.

Example: '_07-345-887_' includes 407-345-8871, 007-345-8875

Note: Access uses * instead of % and ? instead of _. Oracle searches are case-sensitive, Access searches are not.

The IN Special Operator

Many queries that would seem to require the use of the logical OR operator can be more easily handled with the help of the special operator IN.

For example the query:

```
SELECT *
    FROM PRODUCT
    WHERE V_CODE = 21344 OR V_CODE = 24288;
```

can be handled more efficiently with:

```
SELECT *
    FROM PRODUCT
    WHERE V_CODE IN (21344, 24288);
```

Example with IN

Given the relation as follows:

Store (Name, beer, price)

Using IN to write SQL commands listing all beers sold in both “Hải Xồm” and “Lan Chín”.

```
SELECT beer
FROM Store
WHERE Name= ‘Hải Xồm’
AND beer IN (SELECT beer
             FROM Store
             WHERE Name = ‘Lan Chín’);
```

The EXISTS Special Operator

The EXISTS operator can be used whenever there is a requirement to execute a command based on the result of another query. That is, if a subquery returns any rows, then run the main query, otherwise, don't. We'll see this operator in more detail when we look at subqueries in more depth.

For example, suppose we want a listing of vendors, but only if there are products to order. The following query will accomplish our task.

```
SELECT  *
    FROM VENDOR
    WHERE EXISTS  ( SELECT *
                          FROM PRODUCT
                          WHERE P_ONHAND <= P_MIN);
```

Example 1 using EXISTS

Given relations as follows

Student (StudentID, name, age)

Course (Course#, name, credits)

Take (StudentID, Course#, grade)

Write SQL commands listing names of students taken at least one course.

```
SELECT name
FROM Student
WHERE EXISTS (SELECT *
              FROM Take
              WHERE Student.StudentID = Take.StudentID);
```

Example 2 using EXISTS

Given relations as follows

Student (StudentID, name, age)

Course (Course#, name, credits)

Take (StudentID, Course#, grade)

Write SQL commands to delete all students who do not take any course.

```
DELETE FROM Student
WHERE NOT EXISTS (SELECT *
                  FROM Take
                  WHERE Student.StudentID = Take.StudentID);
```

Exercise 3 using EXISTS

Given a relation as bellow.

Beers (name, manufacturer)

Write an SQL command to list all unique beers of all manufacturers.

```
SELECT name
FROM Beers AS b1
WHERE NOT EXISTS (SELECT *
                  FROM Beers AS b2
                  WHERE b2.manufacturer = b1.manufacturer
                      AND b2.name <> b1.name);
```

Ordering a List

The ORDER BY clause is especially useful if the listing order is important.

The syntax is:

```
SELECT  columnlist
    FROM tablelist
    [ WHERE conditionlist ]
    [ORDER BY columnlist  [ASC | DESC] ]
```

If the ordering column contains nulls, they are either listed first or last depending on the RDBMS.

The ORDER BY clause must always be listed last in the SELECT command sequence.

The default order is ascending.

Cascading Order Sequences

Ordered listings are used frequently. For example, suppose you want to create a phone directory of employees. It would be helpful if you could produce an ordered sequence (last name, first name, middle initial) in three stages:

- ORDER BY last name.
- Within last names, ORDER BY first name.
- Within the order created in Step 2, ORDER BY middle initial.

A multi-level ordered sequence is called a cascading order sequence, and is easily created by listing several attributes, separated by commas, after the ORDER BY clause.

Functions in SQL

function_name(*column*, [parameter 1, parameter 2, …])

Functions operating on each record:

LOWER(A) – change characters from Uppercase to Lowercase

UPPER(a) – change characters from Lowercase to Uppercase

Example:
```
SELECT  UPPER(LastName), LOWER(FirstName)
FROM student ;
```

ROUND(a) – round a number a

PI() – get the value of pi

SQRT(a) – square root of a

POWER(a,b) – get the value of a power b

Functions in SQL

Change the attribute type: CONVERT(*column*, new type)

Example 1: convert the number 2011 to a string “2011”:

```
SELECT   CONVERT(2011, CHAR(5));
```

Example 2: output all courses starting on 25th of any month. (Starting date is stored in column StartDate of table course)

```
SELECT *
FROM course
WHERE  CONVERT(StartDate, CHAR(15)) LIKE "25%";
```

Functions in SQL

Function operating on multiple records:

MAX(*column*) – find the maximum value for numeric data

MIN(*column*) – find the minimum value for numeric data

AVG(*column*) – find the average value for numeric data

COUNT(*column*) – count the number of records

SUM(*column*) – find the summation value for numeric data

Example:

```
SELECT AVG(Age) , SUM(Salary)
FROM Staff;
```

Grouping Records

Frequency distributions can be created quickly and easily using the GROUP BY clause within the SELECT statement.

The syntax is:

```
SELECT  column list
    FROM table list
    [WHERE condition list ]
    [GROUP BY column list ]
    [HAVING condition list ];
```

The GROUP BY clause is generally used when you have attribute columns combined with aggregate functions in the SELECT statement.

For example, to determine the minimum price for each sales code, use the following statement shown on the next page.

Exercise 1

Given relations as follows:

Supplier (Supplier#, Name, Address)

Supply (Supplier#, Product#, amount)

Write SQL commands to:

- List all IDs of suppliers and total number of products that they supply to the market.
- List all names of suppliers and total number of products that they supply to the market.

Exercise 2

Given relations as follows:

Supplier (Supplier#, Name, Address)

Supply (Supplier#, Product#, amount)

Write SQL commands to:

- List all names of suppliers that supply at least 3 products
- List all names of suppliers that supply the biggest amount for each product.
- List all names of suppliers that supply all products to the market.

The HAVING Clause

The HAVING clause is a useful extension of the GROUP BY clause.

HAVING operates like the WHERE clause in the SELECT statement. However, the WHERE clause applies to columns and expressions for individual rows, while the HAVING clause is applied to the **output** of a GROUP BY operation.

For example, suppose you want to generate a listing of the number of products in the inventory supplied by each vendor, but you want to limit the listing to the products whose prices average below $10.00. The first part of this requirement is satisfied with the help of the GROUP BY clause, the second part of the requirement will be accomplished with the HAVING clause.

Exercise 1

Given following relations:

Student (S#, name, age)

Take (S#, C#, grade)

Write SQL commands to:

- List all names of students whose GPAs are at least 2.5
- Given the relation Course (C#, name, credits). List all names and GPAs of the students.

Example 2

Given relations as follows:

Supplier (Supplier#, Name, Address)

Supply (Supplier#, Product#, amount)

Write SQL commands to:

- List all names of suppliers that supply at least 3 products

Virtual Tables: CREATE VIEW

The output of a relational operator is another relation (or table).

Using our sample database as an example, suppose that at the end of each business day, we would like to get a list of all products to reorder, which is the set of all products whose quantity on hand is less than some threshold value (minimum quantity).

Rather than typing the same query at the end of every day, wouldn't it be better to permanently save that query in the database?

To do this is the function of a relational view. In SQL a view is a table based on a SELECT query. That query can contain columns, computed columns, aliases, and aggregate functions from one or more tables.

The tables on which the view is based are called base tables.

Views are created in SQL using the CREATE VIEW command.

Virtual Tables: CREATE VIEW

The syntax of the CREATE VIEW command is:

```
CREATE VIEW  viewname
AS SELECT query
```

The CREATE VIEW statement is a DDL command that stores the subquery specification, i.e., the SELECT statement used to generate the virtual table in the data dictionary.

An example:

```
CREATE VIEW PRODUCT_3 AS
   SELECT P_DESCRIPT, P_ONHAND, P_PRICE
   FROM PRODUCT
   WHERE P_PRICE > 50.00;
```

Virtual Tables: CREATE VIEW

A relational view has several special characteristics:

- You can use the name of a view anywhere a table name is expected in an SQL statement.
- Views are dynamically updated. That is, the view is re-created on demand each time it is invoked.
- Views provide a level of security in the database because the view can restrict users to only specified columns and specified rows in a table.
- Views may also be used as the basis for reports.

Delete VIEW

```
CREATE VIEW few_rows_from_t1
AS SELECT *
    FROM t1 LIMIT 10;
DROP VIEW few_rows_from_t1;

CREATE VIEW table_from_other_db AS
      SELECT * FROM db1.foo WHERE A IS NOT NULL;
DROP VIEW table_from_other_db;
```

Joining Database Tables

The ability to combine (join) tables on common attributes is perhaps the most important distinction between a relational database and other types of databases.

In SQL, a join is performed whenever data is retrieved from more than one table at a time.

To join tables, you simply enumerate the tables in the FROM clause of the SELECT statement. The RDBMS will create the Cartesian product of every table specified in the FROM clause.

To effect a natural join, you must specify the linking on the common attributes in the WHERE clause. This is called the join condition.

The join condition is generally composed of an equality comparison between the foreign key and the primary key in the related tables.