RETRIEVAL USING SIMPLE QUERIES

SELECT < ***select list***>
FROM <***table list***>
WHERE <***search condition***>
GROUP BY <***grouping column list***>
HAVING <***search condition***>
ORDER BY<***column list***> **[ASC|DESC]**

LOGICAL PROCESSING

Have a look at page 48, and you are advised to read examples of logical processing for various queries in Block 3.

FROM	Mandatory. An intermediate table resulting in simply a single table or the Cartesian products of tables.
WHERE	Optional. Restricting the rows using predicates in a search condition. Single predicates can be linked into a search condition using the logical operators AND, OR and NOT. The BETWEEN, IN, LIKE and IS NULL operators can also be used in formulating predicates.
GROUP BY	Optional. Collect rows of a table into groups based on the values in one or more columns.
HAVING	Optional. Restricts the groups which have been formed by the GROUP BY clause.
SELECT	Mandatory. Produce the columns for the final table from the intermediate table it receives. The columns are expressed in terms of value expressions which comprise column expressions and column functions.
ORDER BY	Optional. Reorders the rows on the basis of the one or more columns in the table specified in the ORDER BY clause.

The SELECT / FROM / WHERE Clauses

Example 1: List the country with population growth rate > 0.5.
SELECT NAME, ((BIRTHS – DEATHS) / POPULATION) * 100
FROM COUNTRY
WHERE ((BIRTHS – DEATHS) / POPULATION) * 100 > 0.5

The GROUP BY clauses.

Example 2: How many students are registered in each region?
SELECT REGION, **COUNT(*)**
FROM STUDENT
GROUP BY REGION

The HAVING Clauses

Example 3: How many students are enrolled for each of courses c2 and c3?
SELECT COURSE_CODE, COUNT(STUDENT_ID)
FROM ENROLMENT
GROUP BY COURSE_CODE
HAVING COURSE_CODE **IN** ('c2', 'c3')

The ORDER BY Clause

Example 4: List enrolment details of students identified other than S01 and S04 with ordering.
SELECT * **FROM** ENROLMENT **WHERE** STUDENT_ID **NOT IN** ('s01', 's04')
ORDER BY STUDENT_ID, COURSE_CODE **DESC** (**ORDER BY ASC** is the default)

Notes [1]

✍ Column expressions (**Column name,Numeric/String expression)** can be used in the predicates of a WHERE clause. Column functions (**SUM / MAX /MIN/COUNT…**, *see page 20*), however, are not allowed:

SELECT STUDENT_ID, COURSE_CODE, TMA_NO
FROM TMA_GRADE **WHERE** GRADE > AVG(GRADE)

- With GROUP BY, the final table produced by the SELECT clause must reduce each group to a single row, this SELECT clause can only reference columns which are either Group Columns or having a Column Function

SELECT COUNTRY, **MAX**(PRODUCT)
FROM PRODUCTION
GROUP BY CLASSIFICATION

- For HAVING, columns referenced in the search condition must either be Grouping Columns or have a Column Function applied to them.

SELECT COUNTRY, **COUNT**(PRODUCT), **SUM**(QUANTITY)
FROM PRODUCTION
GROUP BY COUNTRY
HAVING SUM(QUANTITY) > 15000 **AND** CLASSIFICATION <> 'Chemicals'

USING JOINS

Aliases and Self-joins

Example 5:
SELECT P.STAFF_NO, P.NAME, Q.STAFF_NO
FROM STAFF **P**, STAFF **Q**
WHERE P.STAFF_NO <> Q.STAFF_NO **AND** P.NAME = Q.NAME

Outer Joins

Example 6:
SELECT student.student_id, name, phone_no
FROM student **LEFT OUTER JOIN** telephone
ON student.student_id = telephone.student_id

If LEFT OUTER is not specified, and there are some students who are not in the telephone table, the result shows no entries for these students at all, but we normally prefer to have display even it contains **NULL** value under the telephone column.

RIGHT OUTER JOIN is similar to **LEFT OUTER JOIN** except that it adds **NULL** entries to preserve rows originating from the right (in the JOIN) table. **FULL OUTER JOIN** simply combines LEFT and RIGHT.

Natural Joins

Instead of having to give a join condition, using keywords NATURAL JOIN implicitly represents the joining condition using matching on all columns(name) from the tables.
Example 7: **SELECT** student.student_id, name, phone_no **FROM** student **NATURAL JOIN** telephone

USING NULL

- There is a value but it is not known at the moment;
- There is no value because the column is not applicable for that row.

Effect of NULLs in different situation

Study the points listed on page 57.

SQL'S UPDATE FACILITIES

DELETE	Example 8:	**DELETE** FROM MOD_COURSE **WHERE** QUOTA **IS NULL**
INSERT	Example 9:	**INSERT INTO** MOD_COURSE **VALUES** ('c9', 'Psychology', 0.5, 50)
UPDATE	Example 10:	**UPDATE** MOD_COUNTRY **SET** YR = 2001, POPULATION = POPULATION * 2

RETRIEVAL USING COMPOSITE QUERIES

UNION

- To combine the results of separate compatible queries
- **UNION Compatible** the separate query specifications which are UNIONed together must each result in intermediate tables having the same number of columns which also correspond in data types.
- The use of **UNION** automatically removes duplicates. **UNION ALL** can be used to keep duplicates.

SUBQUERIES

One query specification is nested within another, so that the result of the inner query specification becomes part of the processing of the other.

Example 11: List all those countries whose production of oats is less than the average.

```
SELECT COUNTRY
FROM PRODUCTION
WHERE PRODUCT = 'Oats' AND QUANTITY < (SELECT AVG(QUANTITY)
                                       FROM PRODUCTION
                                       WHERE PRODUCT = 'Oats')
```

See the last paragraph of page 65 to study the logical processing for such a query..

Subqueries versus Join

1. A subquery cannot always be used as an alternative to a join; for instances, selecting data which is in more than one table in order to present this data in the final table.
   Example 12:
   SELECT S.STUDENT_ID, GRADE **FROM** STUDENT S, ASSIGNMENT A
   WHERE S.STUDENT_ID = A.STUDENT_ID **AND** COURSE_CODE = 'C4' **AND** ASSIGNMENT_NO = '1' **AND** REGION ='3'
   Exercise: There could be two equivant ways using subqueries (hints: student_id can come from Student or Assignment)
2. It is necessary to use a subquery when one wants to compare a single row value with a function of some values drawn from the whole table, or when one wants to compare a single group value with an aggregate involving some or all of the group values. *See Example 11 above.*
3. When data is selected from only one table, but depends on a comparison with data in one or more other tables, it is conceptually simpler and more elegant to formulate the query using a subquery rather than a join. However, using a join still results in greater efficiency on execution.

Outer reference and Correlated subqueries

- A correlated subquery has a search condition containing one or more references to a column or columns in the main query specification.

✍ A correlated subquery has to be evaluated for every row of the appropriate intermediate table in main query specification whereas an ordinary subquery us evaluated once – at the start of processing.

Example 13: List all countries which, in any year had over 20% of the total population of all countries in that year:

```
SELECT COUNTRY, YR
FROM POPULATION P
WHERE P.POPULATION > (SELECT 0.2 * SUM(Q.POPULATION)
                      FROM POPULATION Q
                      WHERE P.YR = Q.YR)
```

If the data from the other tables depends on values drawn from the main query specification then a correlated subquery must be used. *See the 1st paragraph of page 72 how correlated query is logically processed.*

Exercise: Give a SQL query: for each country, give any census(10) year where its population had not increased by 5% since the previous census.

The EXISTS operator

An **EXISTS** predicate involving a subquery is extremely useful when retrieving data based on the existence or non-existence of other data. The **SELECT** clause of a subquery used in an **EXISTS** predicate does not have to refer to only one column.

Example 14:

```
SELECT STUDENT_ID, NAME
FROM STUDENT S
WHERE EXISTS (SELECT *
              FROM ENROLMENT
              WHERE S.STUDENT_ID = STUDENT_ID)
```

Two Alternatives:

```
SELECT DISTINCT S.STUDENT_ID, NAME
FROM STUDENT S, ENROLMENT E
WHERE S.STUDENT_ID = E.STUDENT_ID

SELECT STUDENT_ID, NAME
FROM STUDENT
WHERE STUDENT_ID IN (SELECT STUDENT_ID FROM ENROLMENT)
```

DATA DEFINITION

TABLES

CREATE TABLE <table name> (<column definitions>)

Example 15:

```
CREATE TABLE RECENT_STUDENT
 ( NAME CHAR(16) NOT NULL,
   GPD DECIMAL(4,1),
   CARS INTEGER,
   POPULATION DECIMAL(6,1) )
```

DROP TABLE <table name>

Example 16:

```
DROP TABLE DROP_ME1
```

ALTER TABLE <table name> **ADD** <column name> <data type>

Example 17:

```
ALTER TABLE STAFF ADD TEL_NO INTEGER
```

VIEWS

CREATE VIEW <view name> (column list>) **AS** <view specification>

Example 18:

```
CREATE VIEW COUNSELLING2 (S_NAME, S_NO, C_NAME,
C_NO, REGION) AS
    SELECT S.NAME, STUDENT_ID, C_NAME,
    COUNSELLOR_NO, S.REGION
    FROM STUDENT S, STAFF C
    WHERE COUNSELLOR_NO = STAFF_NO
```

DROP VIEW <view name>

Example 19:

```
DROP VIEW V_DROP_ME2
```

Using WITH CHECK OPTION

To prevent an update resulting in data outside the view

Consider the following two INSERT statements to this VIEW :
CREATE VIEW mod_reg4_student (name, id, counselor, region) **AS**
SELECT name, student_id, counselor_no, region **FROM** mod_student **WHERE** region = '4'

i) INSERT INTO mod_reg4_student VALUES ('Hancock', 's21', '1941', '4') and
ii) INSERT INTO mod_reg4_student VALUES ('Watson', 's50', '4219', '2')

What is the difference when mod_reg4_student is defined with 'WITH CHECK OPTION'?

Related schema tables

- **VIEWS**

VIEW_NAME	The name of the view
VIEW_TEXT	The query specification used to define the view

- **VIEW_USAGE**

BASED_NAME	The name(s) of the table(s) (a view of a table) on which the view is based
BASED_TYPE	Indicates whether each table on which the view is based is a base table® or a view(V)
DERIVED_NAME	The name of the view

Logically processing queries containing views *(see page 104)*

- **Materializing model**

  Materialize the view as a real table which is the result of executing the query specification defining the view (and relabelling the columns if necessary). This table is then considered to be a base table which may be processed by the FROM clause.

```
SELECT *
FROM REG4_STUDENT
WHERE NAME LIKE 'R%'
```

```
SELECT *
FROM (SELECT NAME, STUDENT_ID, COUNSELLOR_NO, REGION
    FROM STUDENT
    WHERE REGION = '4')
WHERE NAME LIKE 'R%'
```

- **Reformulating model**

Reformulates the query in terms of actual base tables and combines the logic of the view definition with the logic of the query involving the view.

SELECT COURSE_CODE
FROM COURSE_SIZE
WHERE SIZE > 3

SELECT COURSE_CODE
FROM ENROLMENT
GROUP BY COURSE_CODE
HAVING COUNT(STUDENT_ID) > 3

Updating Views

Rules which a view has to satisfy for it to be accepted as updateable:

- one-to-one correspondences are guaranteed
- the mappings to the base table(s) can be established

1. The SELECT clause can only include column names (i.e. there must be no numeric or string expressions or column functions) and cannot use the DISTINCT operator.
2. The FROM clause can only include one table (i.e. there must be no joins).
3. The WHERE clause cannot include a subquery that references itself.
4. There can be no GROUP BY clause and no HAVING clause.
5. There can be no UNION.

In general, an updateable view is a subset of a single base table.

CONSTRAINTS

- **PRIMARY KEY Constraint**
- **UNIQUE Constraint (Alternate Key)**
- **Not NULL Constraint**
- **DOMAIN Constraint**
- **Referential Constraint**
  - SET NULL: sets the value of the foreign key to NULL
  - SET DEFAULT: sets the value of the foreign key to its default value.
  - CASCADE: deletes the dependent row.
- **CHECK Constraint**
  - A general expression for a search condition that must not be FALSE; to take account of NULL value
  - Example 20: On student table, students and their counselors must be in the same region
    CHECK (REGION = **SELECT** REGION **FROM** STAFF **WHERE** COUNSELLOR_NO = STAFF_NO)

THE USE OF SQL FACILITIES FOR ACCESS CONTROL

Multiuser environment is based on an authorization identifier for each user, who may own many tables and who has a schema in which the user's tables, views and privileges are defined. A database consists of the total collection of all users' tables. A user may refer to any other user's table by **qualifying** the table name with the name of its schema.

Example 21:

SELECT S.STUDENT_ID, NAME, REGION
FROM dba.STUDENT S, **dba**.ENROLMENT E
WHERE S.STUDENT_ID = E.STUDENT_ID **AND** COUNSELLOR_NO = TUTOR_NO

Access control is necessary in such a shared data environment.

GRANT <PRIVILEGES> **ON** <TABLE> **TO** <USERS>

Example 22:
GRANT SELECT, UPDATE(QUOTA) **ON** COURSE **TO** PUBLIC

One or more of the following privileges can be specified:

SELECT	allows any query statement to retrieve data from the named table
INSERT	allows any INSERT statement to add rows to the named table
DELETE	allows any DELETE statement to remove rows from the named table
UPDATE	allows any UPDATE statement to change the values in any column of the named table
UPDATE (<column list>)	allows any UPDATE statement to change the values in only the specified columns of the named table
REFERENCES	Allows a user to create foreign key references to the named table
ALL PRIVILEGERS	gives all of the above privileges on the named table

Owner can remove privileges using a REVOKE statement:

REVOKE <privileges> **ON** <table> **FROM** <users>

Example 23:
REVOKE SELECT, UPDATE(QUOTA) **ON** COURSE **FROM** PUBLIC

WITH GRANT OPTION
Allows a user to be given certain privileges for a table and to grant the same privileges to other users.

GRANT <privileges> **ON** <table> **TO** <users>
WITH GRANT OPTION

Example 24:
(As M357)
GRANT SELECT **ON** STUDENT **TO** ADMIN **WITH GRANT OPTION**
(As ADMIN)
GRANT SELECT **ON M357.**STUDENT **TO** FACULTY

Using VIEWS to limit access to a database system

There is a need for controlling access to parts of tables. The solution is to grant privileges for views of the base table rather than for the base table itself.

Exercise: Give three distinct reasons for using VIEWS.

PROGRAMMING WITH SQL

EMBEDDED SQL

An application program, written in conventional language like C, C++, PASCAL, COBAL, PL/1 or SMALLTALK, can have SQL code embedded in it to allow database access. The programming language is referred to as the **host language**.

Interaction between the SQL DBCS and an application program

As a program buffer, **host variables** can be used both in host-language statements and in embedded SQL statement.

One special host variable – SQLSTATE, whose value is set by DBCS to indicate exceptions and errors in the execution of SQL statement.

00 000	Statement successful
01 000	Warning

01 003	Null value eliminated in set function
02 000	No data (no row satisfied the conditions of a WHERE clause for either retrieval or update)
22 000	Data exception
22 012	Division by zero
23 000	Integrity constraint violation

Direct Statement

Example 25:

```
EXEC SQL {identify the SQL statement}
INSERT INTO STUDENT (STUDENT_ID, STUDENT_NAME, REGISTERED, REGION)
VALUES (:StudentId, :StudentName, :YearRegistered, :Region);
```

Statement using Cursors

Example 26:

```
EXEC SQL
DECLARE REGIONAL_STUDENT CURSOR FOR {cursor definition}
        SELECT STUDENT_ID, NAME, REGISTERED
        FROM STUDENT
        WHERE REGION = :Region
        ORDER BY NAME;
.
.
read (Region);
.
EXEC SQL
OPEN REGIONAL_STUDENT; {makes the cursor active}
. . .
EXEC SQL
FETCH REGIONAL_STUDENT {process one at a row}
INTO :StudentId, :StudentName, :YearRegistered;

WHILE SQLSTATE <> '02000' DO
BEGIN
        … {process data from one row}
        FETCH REGIONAL_STUDENT
        INTO :StudentId, :StudentName, :YearRegistered;
END
EXEC SQL

CLOSE REGIONAL_STUDENT; {makes the cursor inactive}
```

SQL ROUTINES

Functions and Procedures

Example 27:

```
CREATE FUNCTION greater (par1 integer, par2 integer)
        RETURNS INTEGER
BEGIN
        IF par1 > par2
                THEN RETURN par1;
                ELSE RETURN par2;
          END IF;
END
SELECT greater (582,947)
```

Example 28:

```
CREATE PROCEDURE proc_pop (IN my_city VARCHAR(24), OUT my_pop DECIMAL(6,1))
BEGIN
        SELECT population INTO  my_pop
        FROM city WHERE name = my_city;
END
CALL proc_pop ('Paris', Paris_pop)
```

Quiz: Do you know the use of routines may address the needs of performance, sharing and also access control?
Exercise: How to justify the choice of a function or a procedure?

Triggers

Triggers are similar to routines. The only difference is that it is implicitly invoked as a consequence of some actions.

Example 29:

```
CREATE TRIGGER add_enrolment
        AFTER INSERT ON mod_enrolment
        REFERENCING NEW AS new_enrolment
        FOR EACH ROW
BEGIN
        UPDATE mod_course
        SET student_count = student_count + 1
        WHERE mod_course.course_code = new_enrolment.course_code;
END
```

Exercise: Give four difference in the use of CHECK constraints and triggers.

TRANSACTION MANAGEMENT

SQL transaction needs to be defined explicitly by a user whenever a unit work involves more than one SQL statement and updates a database.

COMMIT WORK	The transaction succeeds and all the changes are made during the course of the transaction are **committed** to the database
ROLLBACK WORK	The transaction fails and all the changes made during the course of the transaction are cancelled by **rolling back** the database to its previous state

Concurrent Transaction Problems

- Lost update
- Reading uncommitted data
- Non-repeatable read
- Phantom rows

Please have a look at the visual explanation (Page 150-152) for each case.

Serializable execution

A serializable execution is some sequence of the steps of all transactions in which each transaction can execute to completion as if it had sole use of the database.

Needs Explanation?

Given: Record1{50}; Record2{100}; Record3{150}; Record5{200}
Transaction A [Record1 + 4; Record2 * 5; Record3 - 7]
Transaction B [Record5 + 3; Record1 – 1; Record2 + 2]
There could be many possible sequences of execution of concurrent transaction A and B, for one instance:
A: Record1 + 4 ✍ Record 1{54};
B: Record5 + 3 ✍ Record5{203};
A: Record2 * 5 ✍ Record2{500};
B: Record1 – 1 ✍ …….
A: Record3 – 7 ✍ Record3{143};
B: Record1 – 1 ✍ Record1{53};
B: Record2 + 2 ✍ Record2{502}
The outcome equals one of the serial execution of transactions; A then B
A: (Record1{54}; Record2{500}; Record3{143}); then
B: (Record1{53}; Record2{502}; Record3{143}; Record5{203})

Exercise: Can you draft other possible instances and outcome of execution?

Locking and isolation levels

Implementing serializable transactions involves **locking**. There are several levels of isolation to balance the accessibility and integrity.

SET TRANSACTION ISOLATION LEVEL < SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED >
Please have a look at the table in page 154.