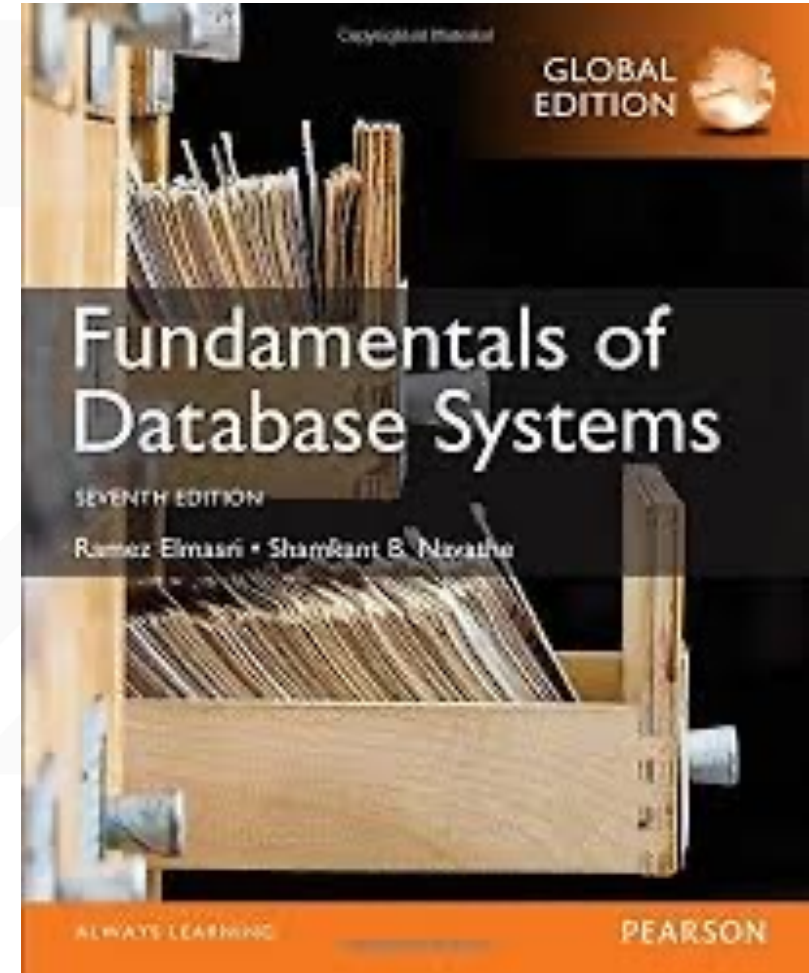# Database Systems

Program in Computer Engineering
Faculty of Engineering

King Mongkut's Institute of Technology Ladkrabang

Text

- Ramez Elmasri and Shamkant B. Navathe. "Fundamentals of Database Systems" 7th Edition., Pearson, 2017

Chapter 7

More SQL: Complex Queries, Views, and Schema
                     Modification

Outline

- More Complex SQL Retrieval Queries

- Views (Virtual Tables) in SQL

- Schema Modification in SQL

More Complex SQL Retrieval Queries

- Additional features allow users to specify more complex retrievals from database:

  - Nested queries,
  - joined tables, and outer joins (in the FROM clause),
  - aggregate functions, and
  - grouping

Comparisons Involving NULL and Three-Valued Logic

- Meanings of NULL
  - Unknown value
  - Unavailable or withheld value
  - Not applicable attribute

- Each individual NULL value considered to be different from every other NULL value

- SQL uses a three-valued logic:
  - TRUE, FALSE, and UNKNOWN (like Maybe)
- NULL = NULL comparison is avoided

Table 7.1 Logical Connectives in Three-Valued Logic

(a)	AND	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE
	UNKNOWN	UNKNOWN	FALSE	UNKNOWN
(b)	OR	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	UNKNOWN
	UNKNOWN	TRUE	UNKNOWN	UNKNOWN
(c)	NOT			
	TRUE	FALSE		
	FALSE	TRUE		
	UNKNOWN	UNKNOWN		

- SQL allows queries that check whether an attribute value is `NULL`
  - **IS** or **IS NOT NULL**

Query 18. Retrieve the names of all employees who do not have supervisors.

Q18:   SELECT   Fname, Lname
         FROM   EMPLOYEE
         WHERE   Super_ssn **IS** NULL;

Nested Queries, Tuples, and Set/Multiset Comparisons

- Nested queries
  - Complete select-from-where blocks within WHERE clause of another query
  - **Outer query and nested subqueries**

- Comparison operator IN
  - Compares value *v* with a set (or multiset) of values *V*
  - Evaluates to TRUE if *v* is one of the elements in *V*

```
Q4A:    SELECT    DISTINCT Pnumber
        FROM      PROJECT
        WHERE     Pnumber IN
                  ( SELECT      Pnumber
                    FROM        PROJECT, DEPARTMENT, EMPLOYEE
                    WHERE       Dnum=Dnumber AND
                                Mgr_ssn=Ssn AND Lname='Smith' )

                  OR
                  Pnumber IN
                  ( SELECT      Pno
                    FROM        WORKS_ON, EMPLOYEE
                    WHERE       Essn=Ssn AND Lname='Smith' );
```

Select the project number of projects that have an employee with last name 'Smith' involved as **manager**.

Select the project number of projects that have an employee with last name 'Smith' involved as **worker**.

- Use tuples of values in comparisons
  - Place them within parentheses

```
SELECT      DISTINCT Essn
FROM        WORKS_ON
WHERE       (Pno, Hours) IN ( SELECT      Pno, Hours
                              FROM        WORKS_ON
                              WHERE       Essn='123456789' );
```

Select the ESSN of all employees who work the same
(project, hours) combination on some project that
employee 'John Smit' (whose SNN = '123456789') works
on.

- Use other comparison operators to compare a single value *v*
  - `=` `ANY` (or `=` `SOME`) operator
    - Returns `TRUE` if the value *v* is equal to some value in the set *V* and is hence equivalent to `IN`
  - Other operators that can be combined with `ANY` (or `SOME`): >, >=, <, <=, and <>
  - `ALL`: value must exceed all values from nested query

```
SELECT      Lname, Fname
FROM        EMPLOYEE
WHERE       Salary > ALL    ( SELECT    Salary
                              FROM      EMPLOYEE
                              WHERE     Dno=5 );
```

Return the name of employees whose salary is greater than the
salary of all the employees in department 5.

- Avoid potential errors and ambiguities
  - Create tuple variables (aliases) for all tables referenced in SQL query

Query 16. Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
Q16:   SELECT      E.Fname, E.Lname
       FROM        EMPLOYEE AS E
       WHERE       E.Ssn IN    ( SELECT      Essn
                                 FROM        DEPENDENT AS D
                                 WHERE       E.Fname=D.Dependent_name
                                 AND E.Sex=D.Sex );
```

Correlated Nested Queries

- **Queries that are nested using the = or IN comparison operator** can be collapsed into one single block: E.g., Q16 can be written as:

Q16A: **SELECT**   E.Fname, E.Lname
                FROM   EMPLOYEE **AS** E , DEPENDENT **AS** D
                WHERE   E.Ssn = D.Essn
                                AND   E.Sex = D.Sex
                                AND   E.Fname = D.Dependent_name;

- **Correlated** nested query
  - Evaluated once for each tuple in the outer query

The EXISTS and UNIQUE Functions in SQL for correlating queries

- **EXISTS** function
  - Check whether the result of a correlated nested query is empty or not. They are Boolean functions that return a TRUE or FALSE result.

- **EXISTS** and **NOT EXISTS**
  - Typically used in conjunction with a correlated nested query

- SQL function **UNIQUE(Q)**
  - Returns TRUE if there are no duplicate tuples in the result of query Q

Q16B: Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
SELECT   Fname, Lname
FROM     Employee  AS  E
WHERE    EXISTS    (   SELECT  *
                       FROM     DEPENDENT  AS  D
                       WHERE   E.Ssn = D.Essn   AND
                               E.Sex = D.Sex    AND
                               E.Firstname = D.Dependent_name    );
```

Q6: Retrieve the names of employees who have no dependents.

SELECT Fname, Lname
FROM Employee **AS** E
WHERE **NOT EXISTS** (**SELECT** *
                                      FROM DEPENDENT
                                      WHERE Ssn = Essn);

Q7: List the names of managers who have at least one dependent.

SELECT Fname, Lname
FROM Employee
WHERE **EXISTS** (SELECT * *Select all DEPENDENT*
FROM DEPENDENT *tuples related to an*
WHERE Ssn = Essn) *EMPLOYEE*

AND
EXISTS
(SELECT *
FROM Department *Select all DEPARTMENT*
WHERE Ssn = Mgr_Ssn) *tuples managed by the*
EMPLOYEE

USE OF NOT EXISTS

- To achieve the "for all" (universal quantifier- see Ch.8) effect, we use double negation this way in SQL:

  Query: List first and last name of employees who work on ALL projects controlled by Dno=5.

```
SELECT    Fname, Lname
FROM      Employee
WHERE     NOT EXISTS (   (  SELECT    Pnumber
                            FROM      PROJECT
                            WHERE     Dno=5          )
                        EXCEPT  (  SELECT    Pno
                                   FROM      WORKS_ON
                                   WHERE     Ssn= Essn    )  );
```

The above is equivalent to double negation: List names of those employees for whom there does NOT exist a project managed by department no. 5 that they do NOT work on.

Double Negation to accomplish "for all" in SQL

Q3B: **SELECT** Lname, Fname
FROM EMPLOYEE
WHERE **NOT EXISTS**
(**SELECT** *
FROM WORKS_ON B
WHERE (B.Pno **IN** (**SELECT** Pnumber
FROM PROJECT
WHERE Dnum=5 **AND**
NOT EXISTS
(**SELECT** *
FROM WORKS_ON C
WHERE C.Essn=Ssn
AND
C.Pno=B.Pno   )));

The above is a direct rendering of:

List names of those employees for whom there does NOT exist a
project managed by department no. 5 that they do NOT work on.

Explicit Sets and Renaming of Attributes in SQL

- Can use explicit set of values in WHERE clause

  Q17:   SELECT   DISTINCT Essn

                 FROM   WORKS_ON

                 WHERE   Pno IN (1, 2, 3);

- Use qualifier AS followed by desired new name
  - Rename any attribute that appears in the result of a query

  Q8A: SELECT   E.Lname **AS** Employee_name, S.Lname **AS** Supervisor_name

             FROM   EMPLOYEE **AS** E, EMPLOYEE **AS** S

            WHERE   E.Super_ssn=S.Ssn;

Specifying Joined Tables in the FROM Clause of SQL

- **Joined table**
  - Permits users to specify a table resulting from a join operation in the FROM clause of a query

- The FROM clause in Q1A
  - Contains a single joined table.
    JOIN may also be called INNER JOIN

Q1A:	SELECT	Fname, Lname, Address
	FROM	(EMPLOYEE **JOIN** DEPARTMENT **ON** Dno=Dnumber)
	WHERE	Dname='Research';

Different Types of JOINed Tables in SQL

- Specify different types of join
  - NATURAL JOIN
  - Various types of OUTER JOIN (LEFT, RIGHT, FULL)

- NATURAL JOIN on two relations R and S
  - No join condition specified
  - Is equivalent to an implicit EQUIJOIN condition for each pair of attributes with same name from R and S

NATURAL JOIN

- Rename attributes of one relation so it can be joined with another using NATURAL JOIN:

Q1B: **SELECT**   Fname, Lname, Address

           FROM   (EMPLOYEE **NATURAL JOIN**

                        (DEPARTMENT **AS**
                           DEPT (Dname, Dno, Mssn, Msdate)))

           WHERE   Dname='Research';

The above works with EMPLOYEE.Dno = DEPT.Dno
as an implicit join condition

INNER and OUTER Joins

- INNER JOIN **(versus** OUTER JOIN**)**
  - Default type of join in a joined table
  - Tuple is included in the result only if a matching tuple exists in the other relation

- LEFT OUTER JOIN
  - Every tuple in left table must appear in result
  - If no matching tuple
    - Padded with NULL values for attributes of right table

- RIGHT OUTER JOIN
  - Every tuple in right table must appear in result
  - If no matching tuple
    - Padded with NULL values for attributes of left table

Aggregate Functions in SQL

- Used to summarize information from multiple tuples into a single-tuple summary
- Built-in aggregate functions
  - COUNT, SUM, MAX, MIN, and AVG
- **Grouping**
  - Create subgroups of tuples before summarizing
- To select entire groups, HAVING clause is used
- Aggregate functions can be used in the SELECT clause or in a HAVING clause

Renaming Results of Aggregation

- Following query returns a single row of computed values from EMPLOYEE table:

Q19: **SELECT** **SUM** (Salary), **MAX** (Salary), **MIN** (Salary), **AVG** (Salary)
FROM EMPLOYEE;

- The result can be presented with new names:

Q19A: **SELECT** **SUM** (Salary) **AS** Total_Sal, **MAX** (Salary) **AS** Highest_Sal,
MIN (Salary) **AS** Lowest_Sal, **AVG** (Salary) **AS** Average_Sal
FROM EMPLOYEE;

- NULL values are discarded when aggregate functions are applied to a particular column

Query 20. Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

Q20: SELECT SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
       FROM   (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
       WHERE   Dname='Research';

Queries 21 and 22. Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

Q21: SELECT COUNT (*)
       FROM   EMPLOYEE;

Q22: SELECT COUNT (*)
       FROM   EMPLOYEE, DEPARTMENT
       WHERE   DNO=DNUMBER AND DNAME='Research';

Aggregate Functions on Booleans

- SOME and ALL may be applied as functions on Boolean Values.
  - SOME returns true if at least one element in the collection is TRUE (similar to OR)
  - ALL returns true if all of the elements in the collection are TRUE (similar to AND)

Grouping: The GROUP BY Clause

- **Partition** relation into subsets of tuples
  - Based on **grouping attribute(s)**
  - Apply function to each such group independently

- GROUP BY clause
  - Specifies grouping attributes

- COUNT (*) counts the number of rows in the group

Examples of GROUP BY

- The grouping attribute must appear in the SELECT clause:

  Q24:   **SELECT**   Dno, **COUNT** (*), **AVG** (Salary)
                  FROM   EMPLOYEE
                  GROUP BY   Dno;

- If the grouping attribute has NULL as a possible value, then a separate group is created for the null value (e.g., null Dno in the above query)

- GROUP BY may be applied to the result of a JOIN:

  Q25:   **SELECT**   Pnumber, Pname, **COUNT** (*)
                  FROM   PROJECT, WORKS_ON
                  WHERE   Pnumber=Pno
                  GROUP BY   Pnumber, Pname;

- **HAVING** clause
  - Provides a condition to select or reject an entire group:

- **Query 26.** For each project *on which more than two employees work,* retrieve the project number, the project name, and the number of employees who work on the project.

Q26:	**SELECT**	Pnumber, Pname, **COUNT** (*)
	FROM	PROJECT, WORKS_ON
	WHERE	Pnumber=Pno
	GROUP BY	Pnumber, Pname
	HAVING	**COUNT** (*) > 2;

Pname	Pnumber	· · ·	Essn	Pno	Hours
ProductX	1		123456789	1	32.5
ProductX	1		453453453	1	20.0
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
ProductZ	3		666884444	3	40.0
ProductZ	3		333445555	3	10.0
Computerization	10	· · ·	333445555	10	10.0
Computerization	10		999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

These groups are not selected by the HAVING condition of Q26.

After applying the WHERE clause but before applying HAVING

Pname	Pnumber	...	Essn	Pno	Hours
ProductY	2		123456789	2	7.5
ProductY	2		453453453	2	20.0
ProductY	2		333445555	2	10.0
Computerization	10		333445555	10	10.0
Computerization	10	...	999887777	10	10.0
Computerization	10		987987987	10	35.0
Reorganization	20		333445555	20	10.0
Reorganization	20		987654321	20	15.0
Reorganization	20		888665555	20	NULL
Newbenefits	30		987987987	30	5.0
Newbenefits	30		987654321	30	20.0
Newbenefits	30		999887777	30	30.0

After applying the HAVING clause condition

Pname	Count (*)
ProductY	3
Computerization	3
Reorganization	3
Newbenefits	3

Result of Q26
(Pnumber not shown)

Combining the WHERE and the HAVING Clause

- Consider the query: we want to count the *total* number of employees whose salaries exceed $40,000 in each department, but only for departments where more than five employees work.

- **INCORRECT QUERY:**

      SELECT   Dno, **COUNT** (*)
      FROM   EMPLOYEE
      WHERE   Salary > 40000
      GROUP BY Dno
      HAVING   **COUNT** (*) > 5;

Correct Specification of the Query:

- Note: the WHERE clause applies tuple by tuple whereas HAVING applies to entire group of tuples

Query 28. For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

```
Q28:   SELECT      Dnumber, COUNT (*)
       FROM        DEPARTMENT, EMPLOYEE
       WHERE       Dnumber=Dno AND Salary>40000 AND
                   ( SELECT      Dno
                     FROM        EMPLOYEE
                     GROUP BY Dno
                     HAVING      COUNT (*) > 5)
```

EXPANDED Block Structure of SQL Queries

```
SELECT <attribute and function list>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <grouping attribute(s)> ]
[ HAVING <group condition> ]
[ ORDER BY <attribute list> ];
```

Views (Virtual Tables) in SQL

- Concept of a view in SQL
  - Single table derived from other tables called the **defining tables**
  - Considered to be a virtual table that is not necessarily populated

Specification of Views in SQL

- **CREATE VIEW** command
  - Give table name, list of attribute names, and a query to specify the contents of the view
  - In V1, attributes retain the names from base tables. In V2, attributes are assigned names

```
V1:   CREATE VIEW    WORKS_ON1
      AS SELECT      Fname, Lname, Pname, Hours
           FROM      EMPLOYEE, PROJECT, WORKS_ON
          WHERE      Ssn=Essn AND Pno=Pnumber;

V2:   CREATE VIEW    DEPT_INFO(Dept_name, No_of_emps, Total_sal)
      AS SELECT      Dname, COUNT (*), SUM (Salary)
           FROM      DEPARTMENT, EMPLOYEE
          WHERE      Dnumber=Dno
       GROUP BY      Dname;
```

- Once a View is defined, SQL queries can use the View relation in the FROM clause
- View is always up-to-date
  - Responsibility of the DBMS and not the user
- `DROP VIEW` command
  - Dispose of a view

View Implementation, View Update, and Inline Views

- Complex problem of efficiently implementing a view for querying
- **Strategy1: Query modification** approach
  - Compute the view as and when needed. Do not store permanently
  - Modify view query into a query on underlying base tables
  - Disadvantage: inefficient for views defined via complex queries that are time-consuming to execute

- **Strategy 2: View materialization**
  - Physically create a temporary view table when the view is first queried
  - Keep that table on the assumption that other queries on the view will follow
  - Requires efficient strategy for automatically updating the view table when the base tables are updated

- **Incremental update strategy for materialized views**
  - DBMS determines what new tuples must be inserted, deleted, or modified in a materialized view table

- Multiple ways to handle materialization:
    - **immediate update** strategy updates a view as soon as the base tables are changed
    - **lazy update** strategy updates the view when needed by a view query
    - **periodic update** strategy updates the view periodically (in the latter strategy, a view query may get a result that is not up-to-date). This is commonly used in Banks, Retail store operations, etc.

View Update

- Update on a view defined on a single table without any aggregate functions
    - Can be mapped to an update on underlying base table?
    - possible if the primary key is preserved in the view

- Update not permitted on aggregate views. E.g.,

UV2:	**UPDATE**	DEPT_INFO
	SET	Total_sal=100000
	WHERE	Dname='Research';

cannot be processed because Total_sal is a computed value in the view definition

- View involving joins
  - Often not possible for DBMS to determine which of the updates is intended

- Clause `WITH CHECK OPTION`
  - Must be added at the end of the view definition if a view is to be updated to make sure that tuples being updated stay in the view

Views as authorization mechanism

- SQL query authorization statements (GRANT and REVOKE) are described in detail in Chapter 30

- Views can be used to hide certain attributes or tuples from unauthorized users

- E.g., For a user who is only allowed to see employee information for those who work for department 5, he may only access the view DEPT5EMP:

```
CREATE VIEW        DEPT5EMP  AS
SELECT             *
FROM               EMPLOYEE
WHERE              Dno = 5;
```

Schema Change Statements in SQL

- **Schema evolution commands**
  - DBA may want to change the schema while the database is operational
  - Does not require recompilation of the database schema

The DROP Command

- **DROP** command
  - Used to drop named schema elements, such as tables, domains, or constraint

- Drop behavior options:
  - `CASCADE` and `RESTRICT`

- Example:
  - DROP SCHEMA COMPANY CASCADE;
  - This removes the schema and all its elements including tables, views, constraints, etc.

The ALTER table command

- **Alter table actions** include:
  - Adding or dropping a column (attribute)
  - Changing a column definition
  - Adding or dropping table constraints
- Example:
  - ```
    ALTER TABLE COMPANY.EMPLOYEE ADD COLUMN Job
    VARCHAR(12);
    ```

Adding and Dropping Constraints

- Change constraints specified on a table
  - Add or drop a named constraint

```
ALTER TABLE COMPANY.EMPLOYEE
DROP CONSTRAINT EMPSUPERFK CASCADE;
```

Dropping Columns, Default Values

- To drop a column
    - Choose either `CASCADE` **or** `RESTRICT`
    - `CASCADE` would drop the column from views etc. `RESTRICT` is possible if no views refer to it.

    ALTER TABLE COMPANY.EMPLOYEE **DROP COLUMN** Address **CASCADE**;

- Default values can be dropped and altered :

    ALTER TABLE COMPANY.DEPARTMENT **ALTER COLUMN** Mgr_ssn **DROP DEFAULT**;
    ALTER TABLE COMPANY.DEPARTMENT **ALTER COLUMN** Mgr_ssn **SET DEFAULT** '333445555';

Table 7.2 Summary of SQL Syntax

Table 7.2 Summary of SQL Syntax

CREATE TABLE <table name> (<column name> <column type> [<attribute constraint>]
                 { , <column name> <column type> [<attribute constraint>] }
                 [<table constraint> { , <table constraint> }])

DROP TABLE <table name>
ALTER TABLE <table name> ADD <column name> <column type>

SELECT [DISTINCT] <attribute list>
FROM (<table name> { <alias> } | <joined table>) { , (<table name> { <alias> } | <joined table>) }
[WHERE <condition>]
[GROUP BY <grouping attributes> [HAVING <group selection condition>]]
[ORDER BY <column name> [<order>] { , <column name> [<order>] }]

<attribute list> ::= (* | (<column name> | <function> (([DISTINCT] <column name> | *)))
                 { , (<column name> | <function> (([DISTINCT] <column name> | *)) }))

<grouping attributes> ::= <column name> { , <column name> }

<order> ::= (ASC | DESC)

INSERT INTO <table name> [(<column name> { , <column name> })]
(VALUES (<constant value> , { <constant value> }) { , (<constant value> { , <constant value> }) }
| <select statement>)

Table 7.2   Summary of SQL Syntax

DELETE FROM <table name>
[WHERE <selection condition>]

UPDATE <table name>
SET <column name> = <value expression> { , <column name> = <value expression> }
[WHERE <selection condition>]

CREATE [UNIQUE] INDEX <index name>
ON <table name> (<column name> [<order>] { , <column name> [<order>] })
[CLUSTER]

DROP INDEX <index name>

CREATE VIEW <view name> [(<column name> { , <column name> })]
AS <select statement>

DROP VIEW <view name>

NOTE: The commands for creating and dropping indexes are not part of standard SQL.

Summary

- Complex SQL:
  - Nested queries, joined tables (in the FROM clause), outer joins, aggregate functions, grouping

- `CREATE VIEW` statement and materialization strategies

- Schema Modification for the DBAs using `ALTER TABLE`, `ADD and DROP COLUMN, ALTER CONSTRAINT` etc.