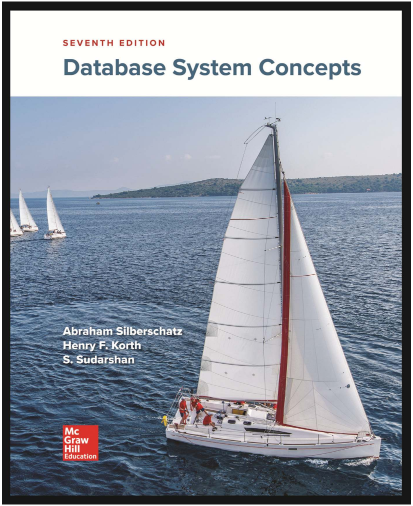
Database System Concepts, 7^{th} Edition
Chapter 3: Introduction to SQL

Silberschatz, Korth and Sudarshan

August 20, 2025

Database System Concepts



Content has been extracted from *Database System Concepts*, Seventh Edition, by Silberschatz, Korth and Sudarshan. Mc Graw Hill Education. 2019.
Visit https://db-book.com/.

Plan

History

- IBM Sequel language developed as part of System R project at IBM San Jose Research Laboratory.
- Renamed Structured Query Language (SQL).
- ANSI and ISO standard SQL:
  - SQL-86
  - SQL-89
  - SQL-92
  - SQL:1999[1]
  - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
  - Not all examples here may work on your particular system.

[1] language name became Y2K compliant!.

SQL Parts

- CRUD – the DML provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.
- Integrity – the DDL includes commands for specifying integrity constraints.
- View definition – the DDL includes commands for defining views.

SQL Parts

- Transaction control – includes commands for specifying the beginning and ending of transactions.
- Embedded SQL and dynamic SQL – define how SQL statements can be embedded within general-purpose programing languages.
- Authorization – includes commands for specifying access rights to relations and views.

Plan

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

- The schema for each relation.
- The type of values associated with each attribute.
- The integrity constraints.
- The set of indices to be maintained for each relation.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk.

Domain Types in SQL

- `char(n)`. Fixed length character string, with user-specified length `n`.
- `varchar(n)`. Variable length character string, with user-specified maximum length `n`.
- `int`. Integer (a finite subset of integers that is machine-dependent).
- `smallint`. Small integer (a machine-dependent subset of integer domain type).

Domain Types in SQL

- `numeric(p, d)`. Fixed point number, with user-specified precision of `p` digits, with `d` digits to the right of decimal point. (ex., `numeric(3,1)`, allows 44.5 to be stores exactly, but not 444.5 or 0.32).
- `real, double precision`. Floating point and double-precision floating point numbers, with machine-dependent precision.
- `float(n)`. Floating point number, with user-specified precision of at least `n` digits.
- More are covered in Chapter 4.

Create Table Construct

- An SQL relation is defiend using the `CREATE TABLE` clause:

CREATE TABLE r (

$$\begin{array}{l} A_1 D_1, A_2 D_2, \ldots, A_n D_n, \\ (integrity\_constraints_1), \\ (integrity\_constraints_2), \\ \quad\vdots \\ (integrity\_constraints_k) \end{array}$$

);

  - r is the name of the relation.
  - each A_i is an attribute name in the schema of relation r.
  - D_i is the data type of values in the domain of attribute A_i.

Create Table Construct

Example:

```
CREATE TABLE instructor (
    ID char(5),
    name varchar(20),
    dept_name varchar(20),
    salary numeric(8,2)
);
```

Integrity Constraints in Create Table

- Types of integrity constraints:
  - `primary key`$(A_{j_1}, \ldots, A_{j_m})$
  - `foreing key`$(A_{k_1}, \ldots, A_{k_n})$
  - `not null`
- SQL prevents any update to the database that violates an integrity constraint.

Integrity Constraints in Create Table

Example:

```
    CREATE TABLE instructor (
        ID          char(5),
        name        varchar(20),
        dept_name   varchar(20),
        salary      numeric(8,2),
        primary key (ID),
        foreign key (dept_name) references deparment
    );
```

Few more Relation Definitions

Example:

```sql
CREATE TABLE student (
    ID          varchar(5),
    name        varchar(20) not null,
    dept_name   varchar(20),
    tot_cred    numeric(3,0),
    primary key (ID),
    foreign key (dept_name) references department
);
```

Few more Relation Definitions

Example:

```
CREATE TABLE takes (
    ID          varchar(5),
    course_id   varchar(8),
    sec_id      varchar(8),
    semester    varchar(6),
    year        numeric(4,0),
    grade       varchar(2),
    primary key (ID, course_id, sec_id, semester, year),
    foreign key (ID) references student,
    foreign key (course_id, sec_id, semester, year)
        references section
);
```

And more still

Example:

```
CREATE TABLE course (
    course_id   varchar(8),
    title       varchar(50),
    dept_name   varchar(20),
    credits     numeric(2,0),
    primary key (course_id),
    foreign key (dept_name) references department
);
```

Updates to Tables

- Insert:

```
INSERT INTO instructor
    VALUES ('10211','Smith','Biology', 66000);
```

- Delete:

```
DELETE FROM studentes;
```

- Drop a table:

```
DROP TABLE r;
```

Updates to Tables

- `ALTER` Clause:
  - ADD

    ALTER TABLE r **ADD** A D ;

    - where A is the name of the attribute to be added to relation r and D is the domain of A.
    - all existing tuples in the relation are assigned *null* as the value for the new attribute.
  - DROP

    ALTER TABLE r **DROP** A ;

    - where A is the name of an attribute of relation r.
    - dropping of attributes not supported by many databases.

Plan

Basic Query Structure

- A typical SQL query has the form:

  SELECT

  $A_1, A_2, \ldots, A_n$

  FROM

  $r_1, r_2, \ldots, r_m$

  WHERE

  P

  - A_i represents an attribute.
  - r_i represents a relation.
  - P is a predicate.
- The result of an SQL is a relation.

The SELECT Clause

- The `SELECT` clause lists the attributes desired in the result of a query.
  - It corresponds to the projection (Π) operation of the relational algebra.

Example:

"Find the names of all instructors."

```
SELECT name FROM instructor;
```

The SELECT Clause

Note

SQL names are case sensitive (i.e., you may use upper- or lower-case letters.)

- E.g., Name $\equiv$ NAME $\equiv$ name.
- Some people use upper case wherever we use bold font.

The SELECT Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword `DISTINCT` after the `SELECT` clause.

Example:

"Find the deparment names of all instructors, and remove duplicates."

```
SELECT DISTINCT dept_name FROM instructor;
```

- The keyword `ALL` specifies that duplicates should not be remove.

```
SELECT ALL dept_name FROM instructor;
```

The SELECT Clause (Cont.)

- An asterisk ($*$) in the `SELECT` clause denotes "all attributes".

```
SELECT * FROM instructor;
```

- An attribute can be a literal with `FROM` clause.

```
SELECT 'A' FROM instructor;
```

  - Result is a table with one column and N rows (number of tuples in the *instructor* table), each row with value "A".

The SELECT Clause (Cont.)

- An attribute can be a literal with no `FROM` clause.

```
SELECT '42';
```

  - Results is a table with one column and a single row with value “42”.
  - Can give the column a name using:

```
SELECT '42' AS foo;
```

The SELECT Clause (Cont.)

- The SELECT clause can contain arithmetic expression involving the operation, $+$, $-$, $*$, and $/$, and operating on constants or attributes of tuples.
  - The query:

```
SELECT ID, name, salary/12 FROM instructor;
```

    would return a relation that is the same as the *instructor* relation, except that the value of the attribute `salary` is divided by 12.
  - Can rename ``salary/12'' using the `AS` clause

```
SELECT ID, name, salary/12 AS monthly_salary
FROM instructor;
```

The WHERE Clause

- The `WHERE` clause specifies conditions that the result must satisfy
  - Corresponds to the selection (σ) predicate of the relational algebra.

Example:

"Find all instructors in Comp. Sci. department."

```
SELECT name
FROM instructor
WHERE dept_name = 'Comp. Sci.';
```

The WHERE Clause

- SQL allows the use of the logical connectives **and**, **or**, and **not**.
- The operands of the logical connectives can be expressions involving the comparison operators $<$, $<=$, $>$, $>=$, $=$, and $<>$.
- Comparisons can be applied to results of arithmetic expressions.

Example:

"Find all instructors in Comp. Sci. with salary $>$ 80000."

```sql
SELECT name
FROM instructor
WHERE dept_name = 'Comp. Sci.' AND salary > 80000;
```

The FROM Clause

- The `FROM` clause lists the relations involved in the query.
  - Corresponds to the Cartesian product ($\times$) operation of the relational algebra.

Example:

"Find the Cartesian producto *instructor* $\times$ *teaches*."

```
SELECT
    *
FROM
    instructor, teaches;
```

The FROM Clause

  - generates every possible instructor – teaches pair, with all attributes from both relations.
  - for common attributes (e.g., `ID`), the attributes in the resulting table are renamed using the relation name (e.g., `instructor.ID`)
- Cartesian product is not very useful directly, but it is useful combined with the `WHERE` clause condition (σ operation in relational algebra).

Examples

"Find the names of all instructors who have taught some course and the `course_id`."

Examples

"Find the names of all instructors who have taught some course and the `course_id`."

```
SELECT
    name, course_id
FROM
    instructor, teaches
WHERE
    instructor.ID = teaches.ID;
```

Examples

"Find the names of all instructors in the Art deparment who have taught some course and the `course_id`."

Examples

"Find the names of all instructors in the Art deparment who have taught some course and the `course_id`."

```
SELECT
    name, course_id
FROM
    instructor, teaches
WHERE
    instructor.ID = teaches.ID AND
    instructor.dept_name = 'Art';
```

Plan

The Rename Operation

- The SQL allows renaming relations and attributes using the `AS` clause:

$$\textit{old\_name}\ \texttt{AS}\ \textit{new\_name}$$

"Find the names of all instructors who have a higher salary than some instructor in Comp. Sci. deparment."

```sql
SELECT DISTINCT
    T.name
FROM
    instructor AS T, instructor AS S
WHERE
    T.salary > S.salary AND
    S.dept_name = 'Comp. Sci.';
```

The Rename Operation

- keyword `AS` is optional and may be omitted.

$$\textit{instructor}\ \texttt{AS}\ \equiv \textit{instructor}\ T$$

Self Join Example

- Relation *employee − supervisor*:

person	**supervisor**
Bob	Alice
Mary	Susan
Alice	David
David	Mary

- Exercises:
  - "Find the supervisor of Bob"
  - "Find the supervisor of the supervisor of Bob"
  - "Could you find ALL the supervisors (direct and indirect) of Bob?"

Homework

- Read the Note 3.1 in the page 80 of the textbook.
- Find **ALL** the supervisors (direct and indirect) of Bob?[2]

[2] Read about *Recursion in SQL* in section 5.4.2 of the textbook.

String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator `LIKE` uses patterns that are described using two special characters:
  - percent (%) – The % character matches any substring.
  - underscore(_) – The _ character any character.

Example:

"Find the names of all instructors whose name includes the substring 'dar'."

String Operations

```
SELECT
    name
FROM
    instructor
WHERE
    name LIKE '%dar%';
```

- Match the string '100%'

```
    LIKE '100\%';
```

in that above we use backslash (\) as the escape character.

String Operations (Cont.)

- Patterns are case sensitive.
- Pattern matching examples:
  - `'Intro%'` matches any string beginning with “Intro”.
  - ‘%Comp%’ matches any string containing “Comp” as a substring.
  - ‘___’ matches any string of exactly three characters.
  - ‘___%’ matches any string of at least three characters.
- SQL supports a variety of string operations such as:
  - concatenation (using “||”).
  - converting from upper to lower case (and viceversa).
  - finding string length, extracting substrings, etc.

Ordering the Display of Tuples

Example:

"List in alphabetic order the names of all instructors."

```
SELECT DISTINCT
    name
FROM
    instructor
ORDER BY
    name;
```

Ordering the Display of Tuples

- We may specify `DESC` for descending order of `ASC` for ascending order, for each attribute; `ASC` is the default.

```
ORDER BY name DESC
```

- Can sort in multiple attributes.

```
ORDER BY dept_name, name
```

Where Clause Predicates

- SQL includes a `BETWEEN` comparison operator.

Example:

"Find the names of all instructors with salary between \$90000 and \$100000 (that is, $\geq$ \$90000 and $\leq$ \$100000).

```sql
SELECT
    name
FROM
    instructor
WHERE
    salary BETWEEN 90000 AND 100000;
```

Where Clause Predicates (Cont.)

- Tuple comparison:

```sql
SELECT
    name, course_id
FROM
    instructor, teaches
WHERE
    (instructor.ID, dept_name) = (teaches.ID, 'Biology');
```

Plan

Set Operations

“Find courses that ran in Fall 2017 or in Spring 2018.”

```sql
(SELECT course_id FROM section WHERE sem = 'Fall' AND year = 2017)
UNION
(SELECT course_id FROM section WHERE sem = 'Spring' AND year = 2018)
```

“Find courses that ran in Fall 2017 and in Spring 2018.”

```sql
(SELECT course_id FROM section WHERE sem = 'Fall' AND year = 2017)
INTERSECT
(SELECT course_id FROM section WHERE sem = 'Spring' AND year = 2018)
```

Set Operations (Cont.)

"Find courses that ran in Fall 2017 but not in Spring 2018."

```sql
(SELECT course_id FROM section WHERE sem = 'Fall' AND year = 2017)
EXCEPT
(SELECT course_id FROM section WHERE sem = 'Spring' AND year = 2018)
```

- Set operations `UNION`, `INTERSECT`, and `EXCEPT` automatically eliminate duplicates.
- To retain all duplicate use `UNION ALL`, `INTERSECT ALL`, and `EXCEPT ALL`.

Plan

Null Values

- It is possible for tuples to have a null value, denoted by `null`, for some of their attributes.
- `null` signifies an unknown value or that a value does not exist.
- The results of any arithmetic expression involving `null` is `null`.

Example

5 + `null` returns `null`.

Null Values (Cont.)

- The predicate `IS NULL` can be used to check for null values.
- The predicate `IS NOT NULL` succeeds if the value on which it is applied is not null.

Example

"Find all instructors whose salary is null."

```
SELECT name FROM instructor WHERE salary IS NULL
```

Null Values (Cont.)

- SQL treats as **unknown** the result of any comparison involving a null value (other than predicates `IS NULL` and `IS NOT NULL`).
  - Example: $5 <$ `null` or `null` $<>$ `null` or `null` $=$ `null`

Null Values (Cont.)

- The predicate in a WHERE clause can involve Boolean operations (`AND`, `OR`, `NOT`); thus the definitions of the Boolean operations need to be extended to deal with the value **unknown**.

Null Values (Cont.)

- The predicate in a WHERE clause can involve Boolean operations (`AND`, `OR`, `NOT`); thus the definitions of the Boolean operations need to be extended to deal with the value **unknown**.
  - `AND`:
    (`true` `AND` **unknown**) = **unknown**
    (`false` `AND` **unknown**) = `false`
    (**unknown** `AND` **unknown**) = **unknown**

Null Values (Cont.)

- The predicate in a WHERE clause can involve Boolean operations (`AND`, `OR`, `NOT`); thus the definitions of the Boolean operations need to be extended to deal with the value **unknown**.
  - `AND`:
    (`true` `AND` **unknown**) = **unknown**
    (`false` `AND` **unknown**) = `false`
    (**unknown** `AND` **unknown**) = **unknown**
  - `OR`:
    (**unknown** `OR` `true`) = `true`
    (**unknown** `OR` `false`) = **unknown**
    (**unknown** `OR` **unknown**) = **unknown**

Null Values (Cont.)

- The predicate in a WHERE clause can involve Boolean operations (`AND`, `OR`, `NOT`); thus the definitions of the Boolean operations need to be extended to deal with the value **unknown**.
  - `AND`:
    (`true` `AND` **unknown**) = **unknown**
    (`false` `AND` **unknown**) = `false`
    (**unknown** `AND` **unknown**) = **unknown**
  - `OR`:
    (**unknown** `OR` `true`) = `true`
    (**unknown** `OR` `false`) = **unknown**
    (**unknown** `OR` **unknown**) = **unknown**
- Result of `WHERE` clause predicate is treated as `false` if it evaluates to **unknown**.

Plan

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value:
  - `AVG`: average value.
  - `MIN`: minimum value.
  - `MAX`: maximum value.
  - `SUM`: sum of values.
  - `COUNT`: number of values.

Aggregate Functions Examples

"Find the average salary of instructors in the Computer Science department."

```sql
    SELECT
        AVG(salary)
    FROM
        instructor
    WHERE
        dept_name = 'Comp. Sci.';
```

Aggregate Functions Examples

"Find the total number of instructors who teach a course in the Spring 2018 semester."

```sql
SELECT
    COUNT(DISTINCT ID)
FROM
    teaches
WHERE
    semester = 'Spring' AND year = 2018
```

Aggregate Functions Examples

"Find the number of tuples in the course relation."

```
SELECT
    COUNT(*)
FROM
    course;
```

Aggregate Functions – Group By

"Find the average salary of instructors in each department."

```sql
SELECT dept_name, AVG(salary) AS avg_salary
FROM instructor
GROUP BY dept_name;
```

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Aggregation (Cont.)

- Attributes in `SELECT` clause outside of aggregate functions must appear in `GROUP BY` list.

```sql
/* erroneous query */
SELECT
    dept_name, ID, AVG(salary)
FROM
    instructor
GROUP BY
    dept_name;
```

Aggregate Functions – Having Clause

"Find the names and average salaries of all departments whose average salary is greater than 42000".

```
SELECT dept_name, AVG(salary) as avg_salary
FROM instructor
GROUP BY dept_name
HAVING AVG(salary) > 42000;
```

Aggregate Functions – Having Clause

"Find the names and average salaries of all departments whose average salary is greater than 42000".

```
SELECT dept_name, AVG(salary) as avg_salary
FROM instructor
GROUP BY dept_name
HAVING AVG(salary) > 42000;
```

Note:
Predicates in the `HAVING` clause are applied after the formation of groups whereas predicates in the `WHERE` clause are applied before forming groups.

Plan

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries. A **subquery** is `SELECT-FROM-WHERE` expression that is nested within another query.
- The nesting can be done in the following SQL query:

SELECT

$A_1, A_2, \ldots, A_n$

FROM

$r_1, r_2, \ldots, r_m$

WHERE

P

as follows:

Nested Subqueries

SELECT

$A_1, A_2, \ldots, A_n$

FROM

$r_1, r_2, \ldots, r_m$

WHERE

P

as follows:

- *From clause*: r_i can be replaced by any valid subquery.
- *Where clause*: P can be replaced with an expression of the form:

  $$B \;\; < operation > \;\; (subquery)$$

  B is an attribute and $< operation >$ will be defined later.
- *Select clause*: A_i can be replaced be a subquery that generates a single value.

Set Membership

Find courses offered in Fall 2017 and in Spring 2018.

```sql
SELECT DISTINCT
    course_id
FROM
    section
WHERE
    semester = 'Fall' AND year = 2017 AND
    course_id IN (SELECT
                        course_id
                    FROM
                        section
                    WHERE
                        semester = 'Spring' AND year = 2018
                    );
```

Set Membership (Cont.)

Find courses offered in Fall 2017 but not in Spring 2018.

```sql
SELECT DISTINCT
    course_id
FROM
    section
WHERE
    semester = 'Fall' AND year = 2017 AND
    course_id NOT IN (SELECT
                        course_id
                    FROM
                        section
                    WHERE
                        semester = 'Spring' AND year = 2018
                    );
```

Set Membership (Cont.)

Name all instructors whose name is neither "Mozart" nor "Einstein".

```sql
SELECT DISTINCT
    name
FROM
    instructor
WHERE
    name NOT IN ('Mozart', 'Einstein');
```

Set Membership (Cont.)

Find the total number of (distinct) students who have taken course sections taught by the instructor with ID 10101.

```
SELECT
    COUNT(DISTINCT ID)
FROM
    takes
WHERE
    (course_id, sec_id, semester, year) IN (
        SELECT
            course_id, sec_id, semester, year
        FROM
            teaches
        WHERE
            teaches.ID = 10101
    );
```

Note

Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

Set Comparison – “some” Clause

Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
SELECT DISTINCT
    T.name
FROM
    instructor AS T, instructor AS S
WHERE
    T.salary > S.salary AND
    S.dept_name = 'Biology';
```

Set Comparison – "some" Clause (Cont.)

Same query using > `SOME` clause.

```sql
SELECT DISTINCT
    name
FROM
    instructor
WHERE
    salary > SOME (
        SELECT
            salary
        FROM
            instructor
        WHERE
            dept_name = 'Biology'
    );
```

Definition of "some" Clause

$F < comp > \texttt{SOME}\ r \Leftrightarrow \exists t \in r$ such that $(F < comp > t)$ where $< comp >$ can be: $<, \leq, >, \geq, =, \neq$.

$$\left(5 < \ \texttt{SOME} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}\right) = true.$$

read: $5 <$ some tuple in the relation.

$$\left(5 < \ \texttt{SOME} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}\right) = false.$$

$$\left(5 = \ \texttt{SOME} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}\right) = true.$$

$$\left(5 \neq \ \texttt{SOME} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline \end{array}\right) = true.$$

Note $(= \texttt{SOME}) \equiv \texttt{IN}$. However, $(\neq \texttt{SOME}) \not\equiv \texttt{NOT IN}$.

Set Comparison – "all" Clause

Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

```
SELECT
    name
FROM
    instructor
WHERE
    salary > ALL (
        SELECT
            salary
        FROM
            instructor
        WHERE
            dept_name = 'Biology'
    );
```

Definition of "all" Clause

$$F < comp > \texttt{ALL}\ r \Leftrightarrow \exists t \in r(F < comp > t).$$

$$\left(5 < \ \texttt{ALL} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline\end{array}\right) = false.$$

$$\left(5 < \ \texttt{ALL} \begin{array}{|c|}\hline 6 \\ \hline 10 \\ \hline\end{array}\right) = true.$$

$$\left(5 = \ \texttt{ALL} \begin{array}{|c|}\hline 4 \\ \hline 5 \\ \hline\end{array}\right) = false.$$

$$\left(5 \neq \ \texttt{ALL} \begin{array}{|c|}\hline 4 \\ \hline 6 \\ \hline\end{array}\right) = true.$$

Note $(\neq$ `ALL`$) \equiv$ `NOT IN`. However, $(=$ `ALL`$) \not\equiv$ `IN`.

Test for Empty Relations

- The `EXISTS` construct returns the value `true` if the argument subquery is nonempty.
- `EXISTS`

$$r \Leftrightarrow r \neq \emptyset$$

- `NOT EXISTS`

$$r \Leftrightarrow r = \emptyset$$

Use of "exists" Clause

- Yet another way of specifying the query "Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester".

```sql
SELECT
    course_id
FROM
    section AS S
WHERE
    semester = 'Fall' AND year = 2017 AND
    EXISTS (
        SELECT *
        FROM section AS T
        WHERE semester = 'Spring' AND
            year= 2018 AND
            S.course_id = T.course_id
    );
```

Use of "exists" Clause

```sql
        SELECT
            course_id
        FROM
            section AS S
        WHERE
            semester = 'Fall' AND year = 2017 AND
            EXISTS (
                SELECT *
                FROM section AS T
                WHERE semester = 'Spring' AND
                    year= 2018 AND
                    S.course_id = T.course_id
            );
```

- **Correlation name:** variable `S` in the outer query.
- **Correlated subquery:** the inner query,

Use of "not exists" Clause

- Find all students who have taken all courses offered in the Biology department.

```
SELECT DISTINCT
    S.ID, S.name
FROM
    student AS S
WHERE
    NOT EXISTS (
        ( SELECT course_id
            FROM course
            WHERE dept_name = 'Biology' )
        EXCEPT
        ( SELECT T.course_id
            FROM takes AS T
            WHERE S.ID = T.ID )
    );
```

Use of “not exists” Clause

```sql
SELECT DISTINCT
    S.ID, S.name
FROM
    student AS S
WHERE
    NOT EXISTS (
        ( SELECT course_id
          FROM course
          WHERE dept_name = 'Biology' )
        EXCEPT
        ( SELECT T.course_id
          FROM takes AS T
          WHERE S.ID = T.ID )
    );
```

- First nested query lists all courses offered in Biology.
- Second nested query lists all courses a particular student took.

- Note: That $X - Y = \emptyset \Leftrightarrow X \subseteq Y$.
- Note: Cannot write this query using `= ALL` and its variants.

Test for Absence of Duplicate Tuples

- The `UNIQUE` construct tests whether a subquery has any duplicate tuples in its result.
- The `UNIQUE` construct evaluates to '`true`' if a given subquery contains no duplicates.

Test for Absence of Duplicate Tuples (Cont.)

- Find all courses that were offered at most once in 2017.

```sql
SELECT
    T.course_id
FROM
    course AS T
WHERE
    UNIQUE (
        SELECT
            R.course_id
        FROM
            section AS R
        WHERE
            T.course_id = R.course_id AND
            R.year = 2017
    );
```

Subqueries in the From Clause

- SQL allows a subquery expression to be used in the `FROM` clause.

Example:

Find the average instructors' salaries of those departments where the average salary is greater than $42,000.

```
SELECT
    dept_name, avg_salary
FROM ( SELECT dept_name, avg (salary) as avg_salary
        FROM instructor
        GROUP BY dept_name )
WHERE
    avg_salary > 42000;
```

Subqueries in the From Clause (Cont.)

- Nothe that we do not need to use the `HAVING` clause.
- Another way to write the previous query:

```
SELECT
    dept_name, avg_salary
FROM ( SELECT dept_name, avg (salary)
        FROM instructor
        GROUP BY dept_name
    ) AS dept_avg (dept_name, avg_salary)
WHERE
    avg_salary > 42000;
```

WITH Clause

- The `WITH` clause provides a way of defining a temporary relation whose definition is available only to the query in which the `WITH` clause occurs.

Example:

Find all departments with the maximum budget.

```sql
WITH max_budget (value) AS
( SELECT MAX(budget)
  FROM department )
SELECT
    department.name
FROM
    department, max_budget
WHERE
    department.budget = max_budget.value;
```

Complex Queries using With Clause

Find all departments where the total salary is greater than the average of the total salary at all departments.

```
WITH dept_total(dept_name, value) AS
( SELECT dept_name, sum(salary)
  FROM instructor
  GROUP BY dept_name ),
dept_total_avg(value) AS
( SELECT AVG(value)
  FROM dept_total )
SELECT
    dept_name
FROM
    dept_total, dept_total_avg
WHERE
    dept_total.value > dept_total_avg.value;
```

Scalar Subquery

- Scalar subquery is one which is used where a single value is expected.

Example:

List all departments along with the number of instructors in each department.

```
SELECT
    dept_name,
    ( SELECT count(*)
      FROM instructor
      WHERE department.dept_name = instructor.dept_name )
    AS num_instructors
FROM
    department;
```

- Runtime error if subquery returns more than one result tuple.

Plan

Modification of the Database

- Deletion of tuples from a given relation.
- Insertion of new tuples into a given relation.
- Updating of values in some tuples in a given relation.

Deletion

Delete all instructors.

```sql
DELETE FROM instructor;
```

Deletion

Delete all instructors.

```sql
DELETE FROM instructor;
```

Delete all instructors from the Finance department.

```sql
DELETE FROM instructor
WHERE dept_name= 'Finance';
```

Deletion

Delete all instructors.

```
DELETE FROM instructor;
```

Delete all instructors from the Finance department.

```
DELETE FROM instructor
WHERE dept_name= 'Finance';
```

Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building.

```
DELETE FROM instructor
WHERE dept_name IN ( SELECT dept_name
    FROM department
    WHERE building = 'Watson' );
```

Deletion (Cont.)

- Delete all instructors whose salary is less than the average salary of instructors.

```
        DELETE FROM instructor
        WHERE salary < ( SELECT AVG(salary)
            FROM instructor );
```

- Problem: as we delete tuples from instructor, the average salary changes
- Solution used in SQL:
  1. First, compute `AVG(salary)` and find all tuples to delete.
  2. Next, delete all tuples found above (without recomputing average or retesting the tuples).

Insertion

Add a new tuple to course.

```
INSERT INTO course
VALUES('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

Insertion

Add a new tuple to course.

```
INSERT INTO course
VALUES('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

or equivalently.

```
INSERT INTO course (course_id, title, dept_name, credits)
VALUES('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

Insertion

Add a new tuple to course.

```
INSERT INTO course
VALUES('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

or equivalently.

```
INSERT INTO course (course_id, title, dept_name, credits)
VALUES('CS-437', 'Database Systems', 'Comp. Sci.', 4);
```

Add a new tuple to student with `credits` set to **null**.

```
INSERT INTO student
VALUES('3003', 'Green', 'Finance', null);
```

Insertion (Cont.)

Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000.

```sql
INSERT INTO instructor
    SELECT ID, name, dept_name, 18000
    FROM student
    WHERE dept_name = 'Music' AND total_cred > 144;
```

Insertion (Cont.)

The `SELECT FROM WHERE` statement is evaluated fully before any of its results are inserted into the relation.
Otherwise queries like:

```
INSERT INTO table1 SELECT * FROM table1;
```

would cause problem.

Updates

Give a 5% salary raise to all instructors.

```
UPDATE instructor
SET salary = salary * 1.05;
```

Updates

Give a 5% salary raise to all instructors.

```sql
UPDATE instructor
SET salary = salary * 1.05;
```

Give a 5% salary raise to those instructors who earn less than 70000.

```sql
UPDATE instructor
SET salary = salary * 1.05
WHERE salary < 70000;
```

Updates

Give a 5% salary raise to all instructors.

```
UPDATE instructor
SET salary = salary * 1.05;
```

Give a 5% salary raise to those instructors who earn less than 70000.

```
UPDATE instructor
SET salary = salary * 1.05
WHERE salary < 70000;
```

Give a 5% salary raise to instructors whose salary is less than average.

```
UPDATE instructor
SET salary = salary * 1.05
WHERE salary < ( SELECT AVG(salary)
    FROM instructor );
```

Updates (Cont.)

Increase salaries of instructors whose salary is over \$100,000 by 3%, and all others by a 5%.

- Write two update statements:

```sql
UPDATE instructor
SET salary = salary * 1.03
WHERE salary > 100000;
UPDATE instructor
SET salary = salary * 1.05
WHERE salary <= 100000;
```

- The order is important.
- Can be done better using the case statement (next slide)...

Case Statement for Conditional Updates

Same query as before but with case statement

```
    UPDATE instructor
    SET salary = CASE
        WHEN salary <= 100000 THEN salary * 1.05
        ELSE salary * 1.03
    END
```

Updates with Scalar Subqueries

Recompute and update `tot_creds` value for all students.

```
UPDATE student S
SET tot_cred = ( SELECT SUM(credits)
    FROM takes, course
    WHERE takes.course_id = course.course_id AND
        S.ID = takes.ID AND
        takes.grade <> 'F' AND
        takes.grade IS NOT NULL );
```

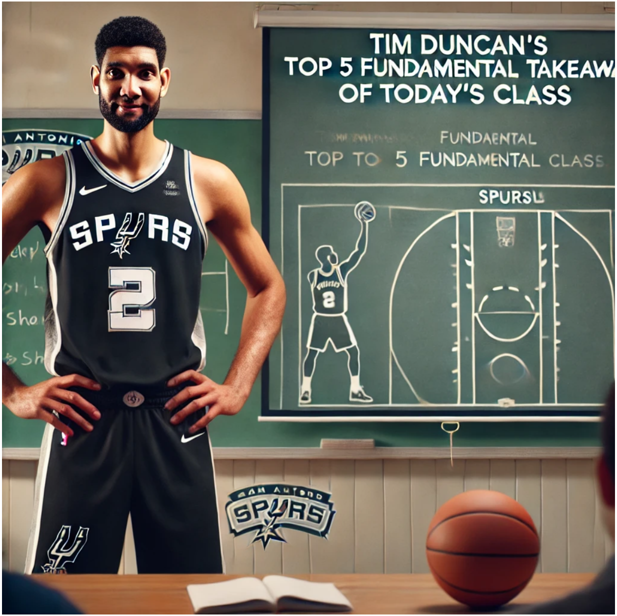
Updates with Scalar Subqueries (Cont.)

- Sets `tot_creds` to **null** for students who have not taken any course.
- Instead of `SUM(credits)`, use:

```
CASE
WHEN SUM(credits) IS NOT NULL THEN SUM(credits)
ELSE 0
END
```

End of Chapter 3.

TDT5FTOTC


TIM DUNCAN'S
TOP 5 FUNDAMENTAL TAKEAW
OF TODAY'S CLASS
FUNDAENTAL
TOP TO 5 FUNDAMENTAL CLASS
SPURSL
SPURS
2
SPURS

Top 5 Fundamental Takeaways

Top 5 Fundamental Takeaways

5. **Handling Nulls and Special Cases:** SQL provides special handling for null values (unknown or missing data) through specific predicates (IS NULL, IS NOT NULL).

Top 5 Fundamental Takeaways

5. **Handling Nulls and Special Cases:** SQL provides special handling for null values (unknown or missing data) through specific predicates (IS NULL, IS NOT NULL).
4. **Data Definition Language (DDL) and Integrity Constraints:** SQL's DDL (CREATE TABLE) defines database schemas, data types, and constraints (primary keys, foreign keys, NOT NULL) to ensure data integrity.

Top 5 Fundamental Takeaways

5 **Handling Nulls and Special Cases:** SQL provides special handling for null values (unknown or missing data) through specific predicates (IS NULL, IS NOT NULL).

4 **Data Definition Language (DDL) and Integrity Constraints:** SQL's DDL (CREATE TABLE) defines database schemas, data types, and constraints (primary keys, foreign keys, NOT NULL) to ensure data integrity.

3 **Database Modifications (CRUD Operations):** SQL supports data manipulation through statements to insert (INSERT), delete (DELETE), and update (UPDATE) database records based on various conditions.

Top 5 Fundamental Takeaways

5 **Handling Nulls and Special Cases:** SQL provides special handling for null values (unknown or missing data) through specific predicates (IS NULL, IS NOT NULL).

4 **Data Definition Language (DDL) and Integrity Constraints:** SQL's DDL (CREATE TABLE) defines database schemas, data types, and constraints (primary keys, foreign keys, NOT NULL) to ensure data integrity.

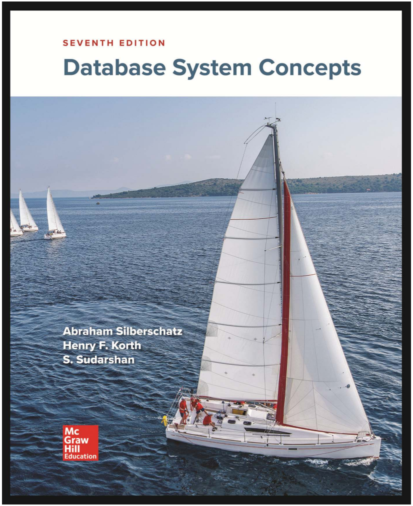
3 **Database Modifications (CRUD Operations):** SQL supports data manipulation through statements to insert (INSERT), delete (DELETE), and update (UPDATE) database records based on various conditions.

2 **Advanced SQL Operations:** SQL includes powerful features like set operations (UNION, INTERSECT, EXCEPT), aggregates (AVG, COUNT, SUM), grouping (GROUP BY, HAVING), and nested subqueries to handle complex queries.

Top 5 Fundamental Takeaways

5. **Handling Nulls and Special Cases:** SQL provides special handling for null values (unknown or missing data) through specific predicates (IS NULL, IS NOT NULL).
4. **Data Definition Language (DDL) and Integrity Constraints:** SQL's DDL (CREATE TABLE) defines database schemas, data types, and constraints (primary keys, foreign keys, NOT NULL) to ensure data integrity.
3. **Database Modifications (CRUD Operations):** SQL supports data manipulation through statements to insert (INSERT), delete (DELETE), and update (UPDATE) database records based on various conditions.
2. **Advanced SQL Operations:** SQL includes powerful features like set operations (UNION, INTERSECT, EXCEPT), aggregates (AVG, COUNT, SUM), grouping (GROUP BY, HAVING), and nested subqueries to handle complex queries.
1. **SQL Basics and Structure:** SQL is a standardized language used for querying and managing relational databases using commands like SELECT, FROM, and WHERE.

Database System Concepts



Content has been extracted from *Database System Concepts*, Seventh Edition, by Silberschatz, Korth and Sudarshan. Mc Graw Hill Education. 2019.
Visit `https://db-book.com/`.