CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Querying in SQL

LECTURE 8

Dr. Philipp Leitner

philipp.leitner@chalmers.se

@xLeitix

Last lecture

We started with SQL, and discussed how to create, drop, and alter tables, as well as how to insert and modify data.

Creating Tables

```
CREATE TABLE ENROLLMENT (
    id INTEGER PRIMARY KEY,
    coursename VARCHAR,
    student CHAR(10),

    FOREIGN KEY (student)
      REFERENCES STUDENT(personnr)
      ON DELETE CASCADE
  );
```

Auto-Increment

(Most) relational databases have a feature to support **automatically generating** a primary key

- Usually through an automatically increasing ID counter

```
CREATE TABLE ENROLLMENT (
    id SERIAL,
    coursename VARCHAR,
    student CHAR(10),

    FOREIGN KEY (student)
      REFERENCES STUDENT(personnr)
      ON DELETE CASCADE
 );
```

In PQSQL: special data type "SERIAL"

Constraints

```
UNIQUE(ATTS)
PRIMARY KEY(ATTS)
FOREIGN KEY(ATT3) REFERENCES TABLE2(ATTF)
DEFAULT <value>
NOT NULL
CHECK <expression>
```

Referential Integrity Triggers

For `FOREIGN KEY` we can tell the database what it should do with referential integrity violations

- A primary key in a **referenced table** changes, what should happen with the **referencing foreign key**?
  - Can happen through updates or deletes

- Four possibilities:
  - Reject update (default, `NO ACTION, RESTRICT`)
  - Cascade change (e.g., delete referencing rows as well, `CASCADE`)
  - Set to NULL or a different default (`SET NULL, SET DEFAULT`)

INSERT

```
INSERT INTO EMPLOYEE VALUES (
   ‘Hugo’, ‘Chavez’, ’324355423’, ‘1962-12-30’,
   ‘Some Address’, ‘M’, 37000, ‘325234523’, 4
 );
```

Or:

```
INSERT INTO EMPLOYEE(Fname, Lname, Ssn) VALUES (
   ‘Hugo’, ‘Chavez’, ’324323’);
```

DELETING data

```
DELETE FROM EMPLOYEE
   WHERE Lname = 'Chavez';

DELETE FROM EMPLOYEE
   WHERE Fname = 'Hugo';

DELETE FROM EMPLOYEE;
```

UPDATING data

```
UPDATE EMPLOYEE
SET Fname = 'Hugo', Lname = 'Chavez'
   WHERE Ssn = '324323345';

UPDATE EMPLOYEE
SET Fname = 'Hugo', Lname = 'Chavez';
```

Everybody becomes Hugo Chavez - again, no questions asked.

Mapping EER to RM / SQL

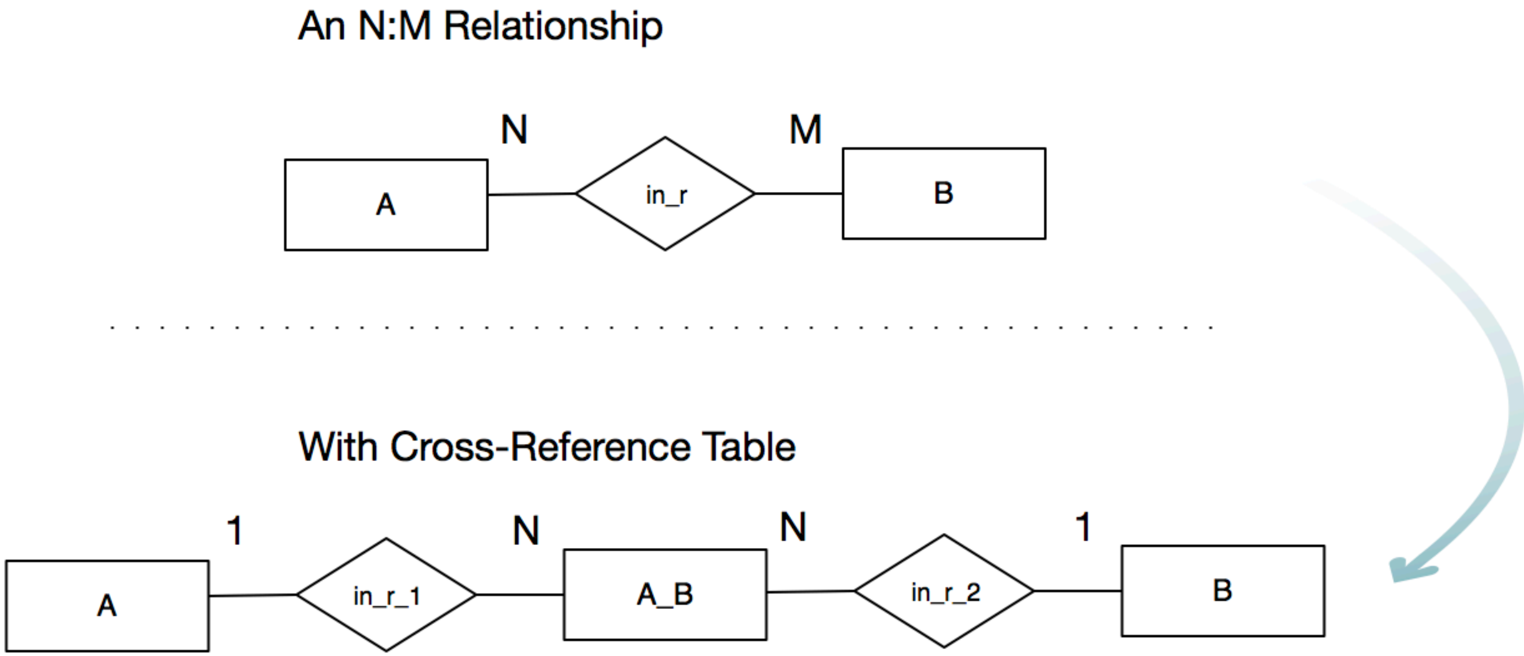
General:

- Entity types become tables, attributes become columns
- One key becomes the primary key, all others `UNIQUE`
- Data types need to be introduced
- 1:1 and 1:N relationships get mapped through PK/Oks

For N:M relationships and many special cases:

- Use cross-reference table

Cross-reference table



Cross-reference table

E.g., many EMPLOYEEs WORKS_ON many PROJECTs

```
CREATE TABLE EMPLOYEE(…);
CREATE TABLE PROJECT(…);
CREATE TABLE EMP_PROJ(
    Employee CHAR(9), Project CHAR(9),
    PRIMARY KEY(Employee, Project),
    FOREIGN KEY(Employee) REFERENCES EMPLOYEE(Ssn),
    FOREIGN KEY(Project) REFERENCES PROJECT(Number)
);
```

Mapping inheritance

Disjoint / partial: one table for each entity

Disjoint / total:

one table per concrete entity OR one table for each entity

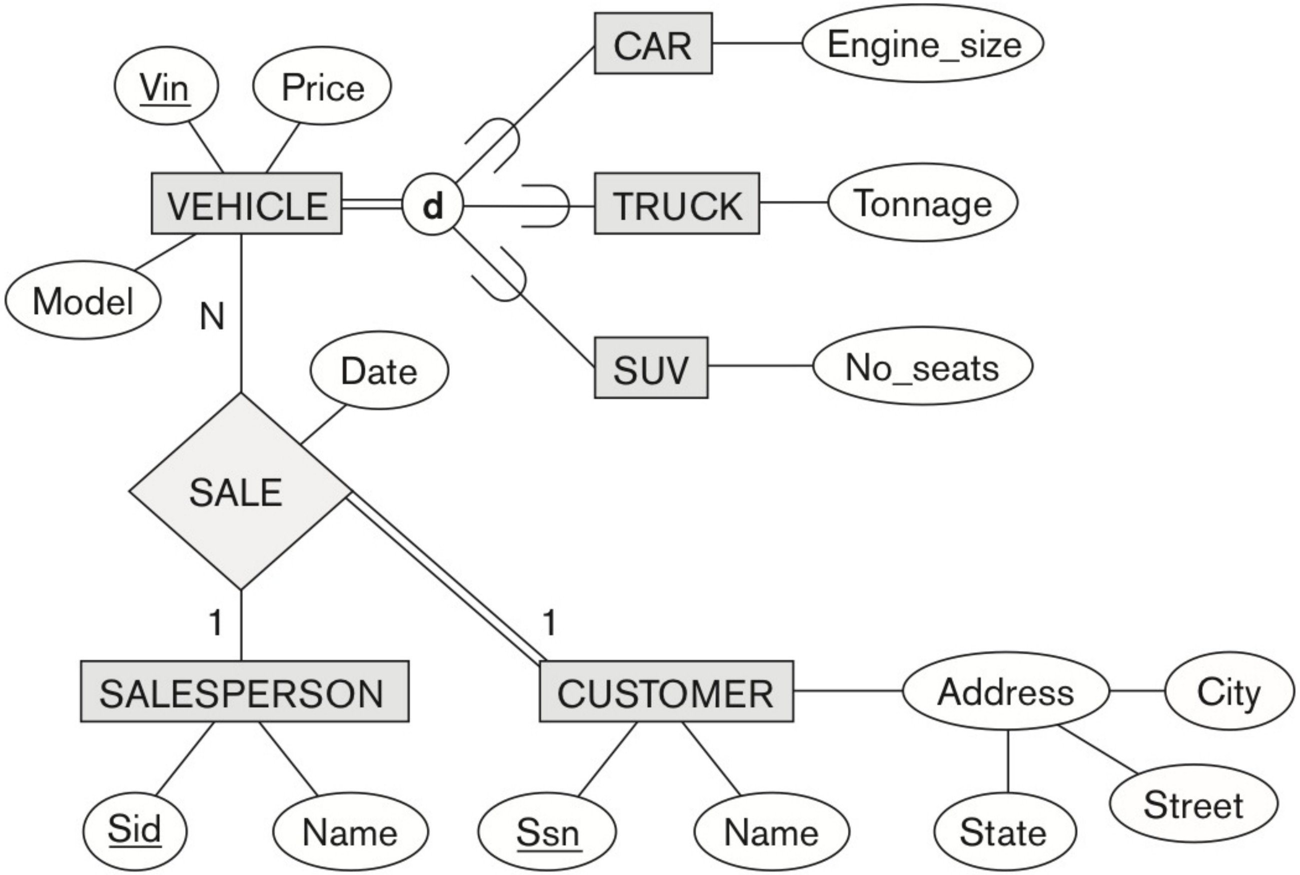
Overlapping / total: single table with all attributes

Overlapping / partial: single table with all attributes

In-Class Exercise

(1) Produce a relational model for this EER diagram.

(2) What SQL code would you use to create the tables?

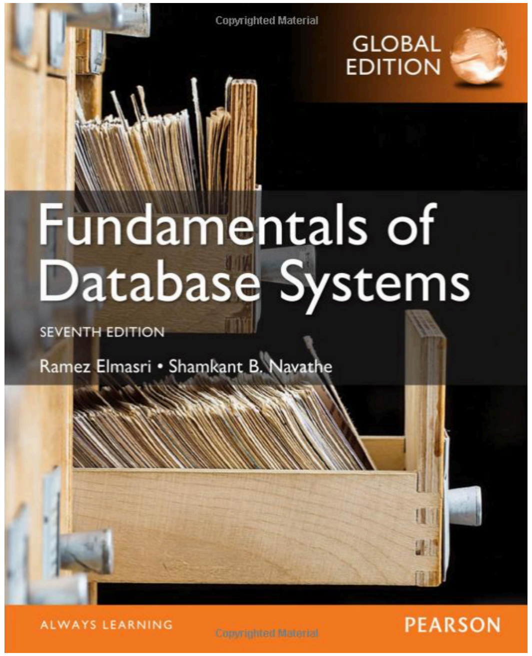


LECTURE 8

Covers …

Most of the remaining parts of Chapter 6 and 7

Please read this up until next lecture!



What we will be covering

Writing SQL Queries

From the simply run-of-the-mill selects …

… over joins and aggregates …

… to subselects

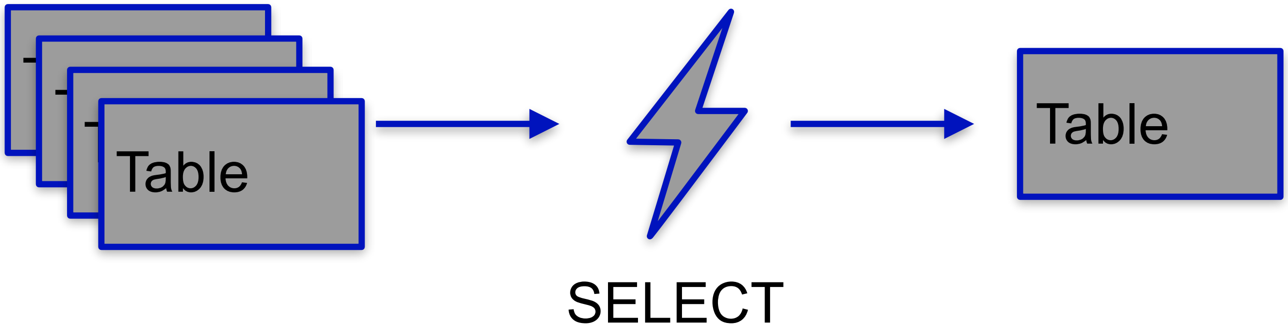
Basic Retrieval Queries (“READ”)

SELECT statement

One basic statement for retrieving information from a database

Idea follows relational algebra:

Note that we are now speaking about tables, not relations



Basic Structure

```
SELECT   <attribute list>
FROM     <table list>
WHERE    <condition>;
```

where

- <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- <table list> is a list of the relation names required to process the query.
- <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

Basic Structure

PROJECT and RENAME

```
SELECT   <attribute list>
FROM     <table list>
WHERE    <condition>;
```

where

- <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- <table list> is a list of the relation names required to process the query.
- <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

Basic Structure

```
SELECT    <attribute list>
FROM      <table list>
WHERE     <condition>;
```

SELECT

where

- <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- <table list> is a list of the relation names required to process the query.
- <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

Simple Example

Query 0. Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

```
Q0:  SELECT  Bdate, Address
     FROM    EMPLOYEE
     WHERE   Fname='John' AND Minit='B' AND Lname='Smith';
```

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

Simple Example

Query 0. Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

```
Q0:   SELECT   Bdate, Address
      FROM     EMPLOYEE
      WHERE    Fname='John' AND Minit='B' AND Lname='Smith';
```

Bdate	Address
1965-01-09	731 Fondren, Houston, TX

Projection Wildcards

SQL supports the use of **wildcards** as part of projection (unlike RA)

```
SELECT * FROM EMPLOYEE;
```

(means “all attributes” from `EMPLOYEE`)

Aliasing Tables

Tables can be given an **alias** during selection

Particularly important for joins

Mandatory if same table is selected more than one

```
SELECT E.Bdate FROM EMPLOYEE as E;
```

Renaming Attributes

In some implementations you can drop the "as"

Two ways to rename attributes:

(1) in the attribute list:

```
SELECT Fname as Firstname FROM  EMPLOYEE;
```

(2) as part of table selection

```
SELECT * FROM EMPLOYEE
  as E(Fn, Mi, Ln, Ssn, Bd, Addr, Sex, Sal, Sssn, Dno);
```

Arithmetics

Arithmetic operations can be used in the SELECT and WHERE clause

```
SELECT Fname, Lname, 1.1 * Salary AS Increased_sal
FROM EMPLOYEE;
```

(show what happened if everybody got a 10% salary increase)

SELECTing Rows

Basic syntax: `WHERE <boolean expression>`

```
SELECT Fname as Firstname FROM EMPLOYEE
  WHERE Fname = 'John';

SELECT Fname FROM EMPLOYEE as E
  WHERE E.Fname = 'John';

SELECT Fname, Lname FROM EMPLOYEE
  WHERE Fname = 'John' AND Lname = 'Smith';
```

Basic Operators for WHERE Clauses

Standard logical operators:

`=, <, <=, >, >=, <>` **(note: =, not ==)**

Boolean operators:

`AND, OR`

Negation:

`NOT` (as in: `NOT(a = 0)`)

Is / Is Not Null:

`IS NULL, IS NOT NULL`

Three-Value Logic

SQL uses a special kind of logic:

TRUE, FALSE, UNKNOWN (NULL)

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		

Additional Operators

LIKE (for textual comparison):

`Fname LIKE '%ohn%'` (matches 'John', 'Ohn', 'Johns', …)

The % is a wildcard for 0 or more other characters

_ can be used to match exactly 1 other character

BETWEEN (to check if value is between two other values):

`Salary BETWEEN 30000 AND 40000`

IN (to check if value is in list):

`Fname IN('John', 'Herbert')`

Basic Operators for WHERE Clauses

First order logical operators:

```
IN     e.g., Ssn IN (123456, 123457, 123458)
ALL    e.g., Salary > ALL (<subquery>)
ANY    e.g., Salary < ANY (<subquery>)
SOME        (same as ANY)
EXISTS e.g., EXISTS(<subquery>)
UNIQUE e.g., UNIQUE(<subquery>)
```

We get to subqueries later

Tables versus Relations

Remember: relations are mathematical sets

    No duplicates, no ordering

Tables are ***not*** sets

    Duplicates ok, ordered (typically in order of insertion)

Duplicates in Tables

All SQL tables are allowed to have duplicates

Multiple rows where all attributes have the same values

In practice no large concern for “normal” tables because of keys

However, the result of a query (which is also a table!) can and will often have duplicates.

Duplicates in Tables

Special SQL keyword to eliminate duplicates:

Query 11. Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

```
Q11:    SELECT   ALL Salary
        FROM     EMPLOYEE;

Q11A:   SELECT   DISTINCT Salary
        FROM     EMPLOYEE;
```

You will virtually never type "all", it's the assumed default

Ordering Query Results

Use **ORDER BY** clause: ORDER BY <list_of_attributes>

- Keyword **DESC** to see result in a descending order of values
- Keyword **ASC** (default) to specify ascending order explicitly

Placed at the **end** of a query

Multiple order attributes can be given, then **first has priority**, second used as tie breaker and so on

```
ORDER BY D.Dname DESC, E.Lname ASC, E.Fname ASC
```

Joining Tables

Most queries are run against multiple tables (i.e., **join queries**)

Query 1. Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1:  SELECT  Fname, Lname, Address
     FROM    EMPLOYEE, DEPARTMENT
     WHERE   Dname='Research' AND Dnumber=Dno;
```

Query 1. Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1:   SELECT   Fname, Lname, Address
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    Dname='Research' AND Dnumber=Dno;
```

Fname	Lname	Address
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

Basic Syntax of a JOIN

Most simple syntax is just listing multiple tables in FROM clause:

```
FROM Employee, Department
```

Note that this is equivalent to the **Carthesian Product A x B**, which we said is usually not very useful

Usually we combine the JOIN with **explicitly linking** the tables through their PK/FKs

```
SELECT E.Fname, E.Lname, D.Dname
FROM Employee as E, Department as D
WHERE E.Dno = D.Dnumber;
```

Explicit Linking via On Clauses

Instead of establishing links in the WHERE clause we can also use an explicit "ON"

```
SELECT E.Fname, E.Lname, D.Dname
FROM Employee as E INNER JOIN Department as D
  ON E.Dno = D.Dnumber;
```

A bit more verbose, but makes clearer what is going on

Also more consistent with OUTER JOIN syntax (see later)

Self-Joins

Counter-intuitively, a table can also be joined with itself:

```
SELECT  E.Fname, E.Lname, S.Fname, S.Lname
  FROM  EMPLOYEE AS E, EMPLOYEE AS S
  WHERE E.Super_ssn=S.Ssn;
```

In this case aliasing the tables is **necessary.**

Basic Syntax of a JOIN

Complex join queries can get fairly verbose

```
SELECT *
FROM A as a, B as b, C as c, D as d
WHERE a.pk = b.fk AND b.pk = c.fk AND c.pk = d.fk;
```

Natural Joins

Given the verbosity of straight-up JOINS, there is also a **syntax for natural joins in SQL**

```
SELECT *
FROM A NATURAL JOIN B NATURAL JOIN C;
```

No explicit joining in the WHERE clause necessary

Requires that **primary key and foreign key attributes are named the same**

Natural Joins

The following **does not work:**

```
SELECT E.Fname, E.Lname, D.Dname
FROM (Employee as E) NATURAL JOIN (Department as D);
```

(PK of Department is called DNumber, but foreign key in Employee is called Dno)

Can use **aliasing** to get around this problem:

```
SELECT E.Fname, E.Lname, D.Dname
FROM (Employee as E) NATURAL JOIN
  (Department as D(Dname, Dno, Mssn, Msdate));
```

Outer Joins

SQL also supports the LEFT, RIGHT, and FULL style of outer joins

```
SELECT  E.Fname, E.Lname, S.Fname, S.Lname
FROM  EMPLOYEE AS E LEFT OUTER JOIN
   EMPLOYEE AS S ON E.Super_ssn=S.Ssn;
```

RIGHT and FULL outer joins follow in the same style

In outer joins we are **required to use** the “On” syntax

Unions, Intersect, Except

Finally, SQL also supports the set operators UNION, INTERSECT, and EXCEPT (difference)

Query 4. Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

```
Q4A:  ( SELECT   DISTINCT Pnumber
        FROM     PROJECT, DEPARTMENT, EMPLOYEE
        WHERE    Dnum=Dnumber AND Mgr_ssn=Ssn
                 AND Lname='Smith' )
        UNION
      ( SELECT   DISTINCT Pnumber
        FROM     PROJECT, WORKS_ON, EMPLOYEE
        WHERE    Pnumber=Pno AND Essn=Ssn
                 AND Lname='Smith' );
```

Subqueries

Some queries require us to execute **another nested query** as part of the WHERE clause

Typically in conjunction with **predicate logic operators:**

IN, ALL, ANY (SOME), EXISTS, UNIQUE

Find all employees with higher salary than all employees in department number 5:

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

Subqueries

```
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' );
```

Subqueries can also return tuples for comparison

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

Using tables from the superquery in the subquery

Often is is required to reference something (tables, attributes, etc.) from the superquery in the subquery

Needs aliases

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 );
```

Using tables from the superquery in the subquery

Note that easily leads to **very inefficient queries**

Pattern to look out for:

N+1 Select Problem

(if the DBMS needs to do a **full table scan** for each tuple in the superquery)

"WITH" Subqueries

Sometimes it is easier to just create a temporary table instead of doing a "true" subquery:

```
WITH dno_5_sales as (
    SELECT salary from EMPLOYEE WHERE Dno = 5)

  SELECT Lname, Fname FROM EMPLOYEE
  WHERE Salary > ALL (dno_5_sales);
```

Aggregate Functions

Used to summarize information from multiple tuples into a single-tuple summary

Available aggregate functions:

`COUNT`, `SUM`, `MAX`, `MIN`, and `AVG`

NULL values are discarded when aggregating

Examples

Return a single row with summary statistics of employees:

```
SELECT SUM(Salary), MAX(Salary),
  MIN(Salary), AVG(Salary) FROM  EMPLOYEE;
```

Often used in combination with aliasing:

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

Grouping

Common problem when aggregating:

We don't want to get the aggregate of **all** rows, but somehow group them by a specific attribute

(**partition** relation into subsets of tuples)

Example: find the average salary **per department**

Solution: GROUP BY clause

Grouping

Example: get the number of employees and their average salary **per department**

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

Unlike other aggregate functions, this query **does not** return just one row; it returns one row **per distinct grouping value** (in that case department numbers).

Basic operating principle:

First apply the grouping (i.e., figure out how many rows the result will have)

Then apply the aggregate functions for each group

Grouping

Important restrictions:

- the grouping attribute(s) **must** appear in the SELECT clause
- other non-aggregate attribute(s) **cannot** appear in the SELECT clause

Wrong:

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

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

Grouping and WHERE

Grouping may also be done in combination with a WHERE clause, for instance as part of a JOIN

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

Note that the WHERE clause is evaluated **before** the grouping happens.

Filtering after grouping

Sometimes we need to reject (filter out) an entire group - this cannot be done with WHERE

Use HAVING:

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

Key Takeaway

Basic SQL Query Syntax

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

Key Takeaway

Understanding basic SQL queries (SELECT, FROM, WHERE)

DISTINCT, ORDER BY

Different types of joins and their syntax

Subselects and WITH

Aggregation, Grouping, and the HAVING clause