# SQL Queries

CS 377: Database Systems

# Today and Next Lecture

1. Basic Single Table Queries

    1. Select-From-Where Query

    2. Useful Operators

    3. Exercise: Company Database

2. Multi-table Queries

    1. Join

    2. Aliasing

# SQL Query

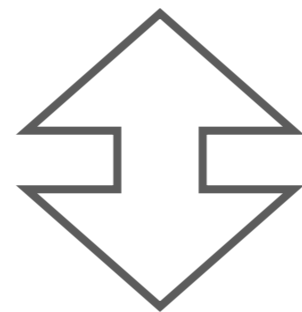Basic form is called a *mapping* or a *SELECT-FROM-WHERE block*
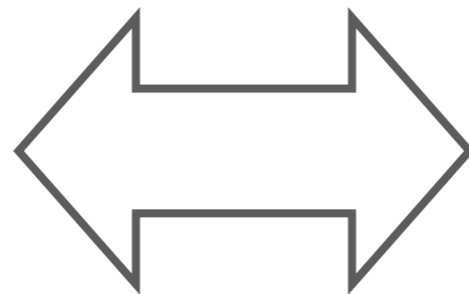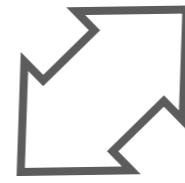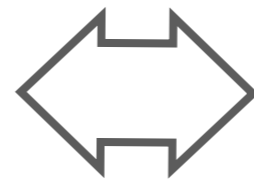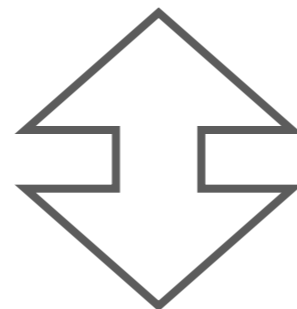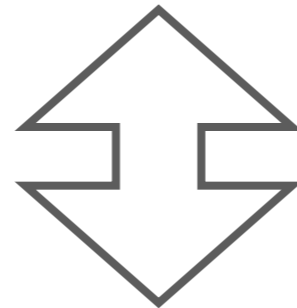
SELECT  <attribute list>
FROM    <table list>
WHERE   <condition on the tables>

# SQL Query <—> Relational Algebra

**SELECT** **<attribute list>**
**FROM** **<table list>**
**WHERE** **<condition on the tables>**

cartesian product of relations is formed

conditions of the form attr1 op constant/attr2

Does not remove duplicates as SELECT in relational algebra

$$\pi_{<\text{attribute list}>}\sigma_{<\text{condition}>}(R_1 \times R_2 \times \cdots \times R_n)$$

# Example: Product Database

## PRODUCT

| Name | Category | Price | Manufacter |
|------|----------|-------|------------|
| iPad | Tablet | $399.00 | Apple |
| Surface | Tablet | $299.00 | Microsoft |
| Kindle | eReader | $79.00 | Amazon |
| Macbook Air | Laptop | $999.99 | Apple |

# Simple SQL Query: * SELECTOR

- Selects all the values of the selected tuples for *all the attributes*

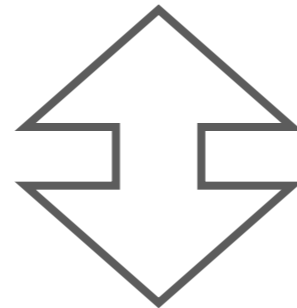- Example:
**SELECT** *
**FROM    Product;**    $\Longleftrightarrow$    $\sigma(\text{Product})$

| Name | Category | Price | Manufacter |
|------|----------|-------|------------|
| iPad | Tablet | $399.00 | Apple |
| Surface | Tablet | $299.00 | Microsoft |
| Kindle | eReader | $79.00 | Amazon |
| Macbook Air | Laptop | $999.99 | Apple |

# Selection Query Using *

SELECT *
FROM    Product
WHERE  Category = 'Tablet';

$$\sigma_{\text{Category}='\text{Tablet'}}(\text{Product})$$

| Name | Category | Price | Manufacter |
|------|----------|-------|------------|
| iPad | Tablet | $399.00 | Apple |
| Surface | Tablet | $299.00 | Microsoft |

# Projection Query

**SELECT Name, Category**
**FROM     Product;**

$$\pi_{\text{Name,Category}}(\text{Product})$$

| Name | Category |
|------|----------|
| iPad | Tablet |
| Surface | Tablet |
| Kindle | eReader |
| Macbook Air | Laptop |

# Select-Project Query

SELECT Name, Category
FROM    Product
WHERE  Manufactor = 'Apple'

| Name | Category |
|------|----------|
| iPad | Tablet |
| Macbook Air | Laptop |

# SQL Details

- SQL commands are case **insensitive**

  - SELECT = select = Select

- Values are case **sensitive**

  - Apple =/= apple

- Single quotes should be used for constants

  - 'Apple' instead of "Apple"

# Exercise: Company Database

- Retrieve the birthdate and address of the employee whose name is 'John B. Smith"

- List the SSN, last name, and department number of all employees

- List the department number and name of all departments

- List the projects under department number 5

# DISTINCT: Eliminate Duplicates

- SQL outputs duplicate values by default

  - Relation is a **multi-set (bag) of tuples** vs a set of tuples

  - Favored for database efficiency

- Syntax:
  Removes duplicate values

  **SELECT DISTINCT <attr list>**
  **FROM    <table>;**

# Example Query: DISTINCT

- SELECT Category
  FROM    Product;

| Category |
|----------|
| Tablet |
| Tablet |
| eReader |
| Laptop |

- SELECT **DISTINCT** Category
  FROM    Product;

| Category |
|----------|
| Tablet |
| eReader |
| Laptop |

# SQL Details: WHERE Conditions

SELECT <attribute list>
FROM    <table list>
WHERE  <condition on the tables>

What can go in here?

- Attribute names of the relation(s) used in the **FROM** clause

- Comparison operators: =, <>, <, >, <=, >=

- Arithmetic operations: +, -, *, /

# SQL Details: WHERE Conditions

- Logical operators to combine conditions: **AND**, **OR**, **NOT**

- Operations on strings (e.g., concatenation)

- Membership test

- Pattern matching

# IN: Member of Set Test

- Tests whether a value is contained in a set

  - True if attribute value is a member of the set of values

  - False otherwise

- Syntax:
  **SELECT** **\<attr list\>**
  **FROM** **\<table\>**
  **WHERE attr IN ( set of values);**

# Example Queries: IN

- Find the name and prices of products made by Amazon or Microsoft:
  **SELECT name, price**
  **FROM    Product**
  **WHERE  Manufacter IN ('Amazon', 'Microsoft');**

- Find the name of products whose made by Amazon or Microsoft and are tablets:
  **SELECT name**
  **FROM    Product**
  **WHERE  (Manufacter, Category) IN (('Amazon', 'Tablet'), ('Microsoft', 'Tablet'));**

# LIKE: Simple String Pattern Matching

- Syntax:
  **SELECT** *
  **FROM** Products
  **WHERE** Name LIKE '%Air';

Substring comparison for partial strings

- Supports 2 wildcard characters

  - Underscore (_) matches exactly one character (equivalent to ? in the UNIX shell)

  - Percent (%) matches 0 or more characters (equivalent to * in the UNIX shell)

# Example Queries: LIKE

- Find names of products whose company start with 'A':
  **SELECT name**
  **FROM    Product**
  **WHERE  Company LIKE 'A%';**

- Find the name and price of products with the word Air in them:
  **SELECT name, price**
  **FROM    Product**
  **WHERE  name LIKE '%Air%;**

# SQL: IS NULL

- Test if an attribute contains the NULL value

- Syntax:
  **attr IS NULL**

- Example: Find employees that have NULL value in the salary attribute

  **SELECT** *
  **FROM** employee
  **WHERE** salary **IS NULL**

# SQL: NOT IN and IS NOT NULL

- Tests whether a value is not contained in a set or not a null value respectively

- Syntax looks similar to the IN and IS NULL operators:
  **attr NOT IN ( set of values)**
  **attr IS NOT NULL**

# SQL: Three-Value Logic

**AND**

| AND | TRUE | FALSE | UNKNOWN |
|---|---|---|---|
| TRUE | TRUE | FALSE | UNKNOWN |
| FALSE | FALSE | FALSE | FALSE |
| UNKNOWN | UNKNOWN | FALSE | UNKNOWN |

**OR**

| OR | TRUE | FALSE | UNKNOWN |
|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE |
| FALSE | TRUE | FALSE | UNKNOWN |
| UNKNOWN | TRUE | UNKNOWN | UNKNOWN |

**NOT**

| TRUE | FALSE | UNKNOWN |
|---|---|---|
| FALSE | TRUE | UNKNOWN |

# SQL: ORDER BY

- Sort the tuples in a query based on the values of some attributes

- Default order is in ascending order of the values (ASC)

- Syntax:
  **SELECT <attribute list>**
  **FROM    <table list>**
  **WHERE  <condition on the tables>**
  **ORDER BY <attribute-list> ASC | DESC;**

sorting by multiple columns is just separated with a comma

# Example Query: ORDER BY

- Sort employees by their salary value in descending order
  **SELECT**      **fname, lname, salary**
  **FROM**        **employee**
  **ORDER BY**   **salary DESC;**

- Sort employees by their salary figures and within the same salary figure, by their last name
  **SELECT**      **fname, lname, salary**
  **FROM**        **employee**
  **ORDER BY**   **salary, lname;**

# SQL: LIMIT

- Limit the output to be only the specified number of tuples

  - Useful if your table has many relations and you just want to sanity check your work

  - Can be used with ORDER BY to get a maximum or minimum value

- Syntax:
  **SELECT** **\<attribute list\>**
  **FROM**    **\<table list\>**
  **WHERE**  **\<condition on the tables\>**
  **LIMIT**    **\<number of tuples\>;**

# Exercise: Company Database (2)

- What are the first and last names of employees who live in Houston?

- What are the SSNs of the top 5 employees who worked the most hours on project number 30? List them in descending order

- Which department number(s) are ProductX, ProductY, and ProductZ controlled by?

# Multi-table Queries

# Recap: SQL Query

SELECT &lt;attribute list&gt;
FROM    &lt;table list&gt;
WHERE  &lt;condition on the tables&gt;

$$\pi_{<\text{attribute list}>}\sigma_{<\text{condition}>}(R_1 \times R_2 \times \cdots \times R_n)$$

# Example Query: Cartesian Product

SELECT ssn, lname, dno, dnumber, dname
FROM     employee, department;

| ssn | lname | dno | dnumber | dname |
|------|--------|------|----------|--------|
| 111-12-2345 | Kirk | 5 | 5 | Research |
| 111-12-2345 | Kirk | 5 | 4 | Administration |
| 111-12-2345 | Kirk | 5 | 1 | Headquarters |
| 222-23-2222 | McCoy | 4 | 5 | Research |
| 222-23-2222 | McCoy | 4 | 4 | Administration |
| 222-23-2222 | McCoy | 4 | 1 | Headquarters |
| … | … | … | … | … |
| 134-52-2340 | Scott | 5 | 5 | Research |
| 134-52-2340 | Scott | 5 | 4 | Administration |
| 134-52-2340 | Scott | 5 | 1 | Headquarters |

# SQL: Join Operation

- Relational algebra expression

$$R_1 \bowtie_{\text{condition}} R_2 = \sigma_{\text{condition}}(R_1 \times R_2)$$

  - Cartesian product followed by a selection operation

- SQL command

  - **FROM** clause specifies Cartesian product operation

  - **WHERE** clause specifies condition of the selection operation

# Example Query: Join

SELECT ssn, lname, dno, dnumber, dname
FROM    employee, department
WHERE  dno = dnumber;

| ssn | lname | dno | dnumber | dname |
|---|---|---|---|---|
| 111-12-2345 | Kirk | 5 | 5 | Research |
| 222-23-2222 | McCoy | 4 | 4 | Administration |
| 134-23-2345 | Sulu | 4 | 4 | Administration |
| 234-13-3840 | Chapel | 1 | 1 | Headquarters |
| 134-52-2340 | Scott | 5 | 5 | Research |

# SQL: Join Part II

- Several equivalent ways to write a basic join

- Method 1 is to just use Cartesian Product on the FROM clause

- Method 2 syntax:
  **SELECT <attribute list>**
  **FROM    <table1>**
  **JOIN    <table2> ON <join condition>**
  **WHERE  <condition on the tables>**

# Example Query: Join Part II

SELECT ssn, lname, dno, dnumber, dname
FROM    employee, department
WHERE  dno = dnumber;

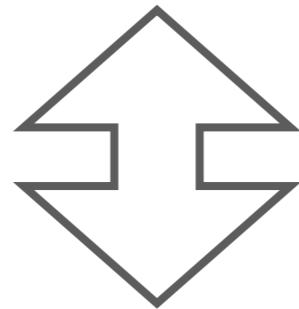SELECT ssn, lname, dno, dnumber, dname
FROM    employee
JOIN     department ON dno = dnumber;

# Example Query: Join in RA

Query: Find the name and address of employees working in the 'Research' department

RA expression:

$$RD = \sigma_{Dname='Research'}(DEPARTMENT)$$
$$RE = RD \bowtie_{Dnumber \ = \ Dno} EMPLOYEE$$
$$Answer = \pi_{fname,lname,Address}(RE)$$

# Example Query: Join in SQL

Q: Find the name and address of employees working in the 'Research' department

- SQL expression 1
  **SELECT** **fname, lname, address**
  **FROM** **employee, department**
  **WHERE** **dname='Research'** **AND** **dno = dnumber**;

- SQL expression 2
  **SELECT** **fname, lname, address**
  **FROM** **employee**
  **JOIN** **department ON** **dno = dnumber**
  **WHERE** **dname='Research'**;

join condition

selection condition

# Exercises: Company Database (3)

- Find the name of employees in the 'Research' department who earn over $30,000

- Find the name of employees who work on the project 'ProductX'

- For the projects located in 'Stafford', find the name of the project, the name of the controlling department, the last name of the department's manager, his address, and birthdate

# SQL: Ambiguity

What happens if we wanted a list of employees with the name of their dependents and the projects the employee works on?

**SELECT essn, pno, dep_name**
**FROM Project, Dependent**
**WHERE essn = essn;**

Ambiguous attribute names: the same name for two (or more) attributes in different relations

# SQL: Qualifying Attribute Names

- Ambiguous attributed names that appear in the same query need to be made explicit (otherwise cannot tell which relation it is from)

- Qualify (prefix) the attribute name with the source relation name

  - Can be done in **SELECT** or **WHERE** clause

- SQL syntax: **&lt;relation&gt;.&lt;attr&gt;**

# Example Query: Qualifying Attribute Names

Find project numbers of projects worked on by employees who have a daughter named 'Alice'

```
SELECT   pno
FROM     works_on, dependent
WHERE    works_on.essn = dependent.essn
   AND   name = 'Alice';
```

# SQL: Ambiguity Take 2

What if we wanted a list of each employees first name, last name, and their manager's first name and last name?

**SELECT fname, lname, fname, lname
FROM     employee, employee
WHERE   superssn = ssn;**

Ambiguous attribute names that won't be solved
by qualifying their relation

# SQL: Aliasing

- Need to use the same relation multiple times in a **SELECT** command and every attribute name of that relation will be ambiguous

- Use an alias or identifier that follows a relation name in the **FROM** clause of a **SELECT** command

# SQL: Aliasing Syntax

- SQL Syntax:
  **SELECT <alias1>.<attr1>, <alias2>.<attr2>
  FROM <relation1> <alias1>, <relation2> <alias 2>
  WHERE <alias1>.<A> = <alias2>.A;**

- No comma between alias and relation name!

- Refer to the relation using the given alias in other parts of query

# Example Query: Aliasing

List each employees first name, last name, and their manager's first name and last name

**SELECT** e.fname, e.lname, m.fname, m.lname
**FROM** employee e, employee m
**WHERE** e.superssn = m.ssn;

e and m are called aliases or tuple variables
for employee relation

# SQL: Arithmetic Operations

- Any arithmetic expression (that makes sense) can be used in the **SELECT** clause

- Example: Show the effect of giving all employees who work on the 'ProductX' project a 10% raise

```
SELECT    fname, lname, 1.1*salary
FROM      employee, works_on, project
WHERE     ssn = essn
   AND    pno = pnumber
   AND    pname = 'ProductX'
```

# Exercises: Company Database (4)

- What are the name of the departments that are located in Houston?

- Find the name of the managers who are in charge of the departments located in Houston

- What are the names of the children whose parents work on ProductX?

# Nested Queries

# Review: Basic SQL Retrieval Query

A SQL query can consist of several clauses, but only **SELECT** and **FROM** are mandatory

SELECT <attribute list>
FROM    <table list>
[WHERE  <condition on the tables (join or selection)>]
[ORDER BY <attribute list>]
[LIMIT <number of tuples>]

# Subquery

- Subquery: A parenthesized **SELECT-FROM-WHERE** statement which results in a relation of tuples

- Syntax:
  (**SELECT**-command)

- Usage

  - Inside **WHERE** clause (nested query)

  - Inside **FROM** clause (temporal relation)

# Nested Query

- Nested query is when a subquery is specified within the **WHERE** clause of another query, called the outer query

- Syntax:
  **SELECT ...**
  **FROM ...**
  **WHERE ...** **(SELECT ...**      <span style="color:blue">Nested Query</span>
  **FROM ...**
  **WHERE ...)**

# Nested Query Forms

- Forms of nested query:

  - Set membership: **IN** and **NOT IN**

  - Set comparison:
    **compareOp ANY** or **compareOp ALL**

  - Test for empty relation: **EXIST**

- In theory, nesting can be arbitrarily deep but in practice the number of levels is limited

# Example Query: Nested Query

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

- Soln #1:  **SELECT fname, lname**
  **FROM    employee, department**
  **WHERE dno = dnumber**
  **AND dname = 'Research';**

- Soln #2:  **SELECT fname, lname**
  **FROM EMPLOYEE**
  **WHERE dno IN (SELECT dnumber**
  **FROM department**
  **WHERE dname = 'Research')**

# Example Query: Nested Query (2)

Find fname, lname of employees that do not have any dependents

**SELECT fname, lname**
**FROM    employee**
**WHERE  ssn NOT IN (SELECT essn**
**                              FROM   dependent);**

# Correlated Nested Queries

- Correlated: inner query (query in the **WHERE** clause) uses one or more attributes from relation(s) specified in the outer query

- Uncorrelated: inner query is a stand-alone query that can be executed independently from the outer query

- Example Syntax:
  **SELECT …**
  **FROM    R1**
  **WHERE  attr1 IN (SELECT attr2**
                        **FROM    R2**
                        **WHERE R2.attr3 = R1.attr4)**

# Example Query: Correlated Nested Query

Retrieve the name of each employee who has a dependent with the same name as the employee

SELECT    e.fname, e.lname
FROM      employee AS e
WHERE     e.ssn IN (SELECT essn
                    FROM    dependent
                    WHERE   essn = e.ssn
                    AND     e.fname = name);

# Correlated Nested Query Execution

- FOR (each tuple X in the outer query) DO {
  Execute inner query using attribute value of tuple X
  }

- Example:
  **SELECT fname, lname, salary, uno**
  **FROM    employee a**
  **WHERE salary >= ALL (SELECT salary**
                                **FROM    employee a**
                                **WHERE  b.dno = a.dno)**

# Correlated Nested Query Execution (2)

| FName | LName | DNo | Salary |
|---|---|---|---|
| John | Smith | 4 | 50,000 |
| James | Bond | 4 | 80,000 |
| Jane | Brown | 3 | 60,000 |
| Jennifer | Wallace | 5 | 30,000 |
| James | Borg | 1 | 55,000 |
| Joyce | English | 5 | 25,000 |
| Alicia | Wong | 4 | 70,000 |

- Outer tuple a =

| John | Smith | 4 | 50,000 |
|---|---|---|---|

WHERE 50,000 >= ALL (SELECT salary FROM employee b where b.dno = 4)
=> FALSE

# Correlated Nested Query Execution (2)

| FName | LName | DNo | Salary |
|---------|---------|-----|--------|
| John | Smith | 4 | 50,000 |
| James | Bond | 4 | 80,000 |
| Jane | Brown | 3 | 60,000 |
| Jennifer | Wallace | 5 | 30,000 |
| James | Borg | 1 | 55,000 |
| Joyce | English | 5 | 25,000 |
| Alicia | Wong | 4 | 70,000 |

- Outer tuple a =

| James | Bond | 4 | 80,000 |
|-------|------|---|--------|

  WHERE 80,000 >= ALL (SELECT salary FROM employee
  b where b.dno = 4)
  => TRUE (select tuple)

# Correlated Nested Query Execution (2)

| FName | LName | DNo | Salary |
|---------|---------|-----|--------|
| James | Bond | 4 | 80,000 |
| Jane | Brown | 3 | 60,000 |
| Jennifer | Wallace | 5 | 30,000 |
| James | Borg | 1 | 55,000 |

SELECT fname, lname, salary, uno
FROM    employee a
WHERE salary >= ALL (SELECT salary FROM
employee a WHERE  b.dno = a.dno)

# Correlated Nested Query Scope

Scoping rules defines where a name is visible

- Each nesting level constitutes a new inner scope

- Names of relations and their attributes in outer query are visible in the inner query but not the converse

- Attribute name specified inside an inner query is associated with nearest relation

# Example: Scoping Nested Queries

SELECT <attribute list from R1 and/or R2>
FROM      R1, R2
WHERE   <conditions from R1 and/or R2> AND
        (SELECT <attribute list from R1, R2, R3 and/or R4>
         FROM      R3, R4
         WHERE <conditions from R1, R2, R3, and/or R4>)

- Attributes of R1 and R2 are visible in the inner query

- Attributes of R3 and R4 are not visible in the outer query

# Example: Scoping Nested Queries (2)

SELECT &lt;attribute list from R1 and/or R2&gt;
FROM      R1, R2
WHERE  &lt;conditions from R1 and/or R2&gt; AND
    (SELECT  x
     FROM      R3, R4
     WHERE &lt;conditions from R1, R2, R3, and/or R4&gt;)

- If R3 or R4 contains the attribute name x, then x refers to that attribute in R3 or R4

- If R3 and R4 does not contain the attribute name x, then x in the inner query refers to the attribute in R1 or R2

# SQL Query: EXISTS

- Checks whether the result of a correlated nested query is empty (contains no tuples) or not

- Example: Retrieve the names of employees who have no dependents

```
SELECT fname, lname
FROM    employee
WHERE  NOT EXISTS (SELECT *
                   FROM   dependent
                   WHERE ssn = essn);
```

# Exercises: Company Database (5)

- Find the fname, lname of the employee(s) that earn the highest salary in his/her department

- Find the fname, lname of the female employee that earn the highest salary (among the females) in her department

- Find fname, lname of employees that do not have any dependent

- Find fname, lname of employees that do not work on any project controlled by the Research department

# SQL Query: Aggregate Functions

- **COUNT**, **SUM**, **MAX**, **MIN**, **AVG** can be used in the **SELECT** clause

- Example: Find the sum, maximum, minimum, and average salary among all employees in the Research department

SELECT SUM(salary), MAX(salary)
            MIN(salary), AVG(salary)
FROM     employee, department
WHERE   dno = dnumber AND dname = 'Research'

# SQL Query: Aggregate Functions (2)

- Name given to the selected aggregate function attribute is the same as the function call

  - **SELECT MAX(salary), MIN(salary), AVG(salary)
    FROM      employee;**

    | max(salary) | min(salary) | avg(salary) |
    | --- | --- | --- |

- Rename selected attributes with AS alias clause inside the SELECT clause

  - **SELECT MAX(salary) AS max, MIN(salary) AS min,
    AVG(salary) AS average
    FROM      employee;**

# SQL Example: Aggregate Function

Retrieve the names of all employees who have two or more dependents

SELECT lname, fname
FROM    employee
WHERE  ( SELECT COUNT (*)
              FROM    dependent
              WHERE  ssn = essn) >= 2;

# SQL Query: GROUP BY

- Apply aggregate functions to subgroups of tuples in a relation

  - Corresponds to grouping and aggregate function in RA

  - Grouping attributes: attributes used to group the tuples

  - Function is applied to each subgroup independently

- Syntax:
  **SELECT** **\<attribute list>**
  **FROM** **\<table list>**
  **WHERE** **\<condition on the tables>**
  **GROUP BY** **\<grouping attributes>**

# GROUP BY Execution

A query with **GROUP BY** clause is processed as follows:

1. Select the tuples that satisfies the **WHERE** condition

2. Selected tuples from (1) are grouped based on their value in the grouping attributes

3. One or more set functions is applied to the group

# SQL Example: GROUP BY

For each department, retrieve the department number, the number of employees in the department, and their average salary

**SELECT     dno, count(*), avg(salary)**
**FROM        employee**
**GROUP BY dno**

# SQL Query: GROUP BY details

- What happens if we do not include certain grouping attributes in the **SELECT** clause?


- What happens if we include an attribute in the **SELECT** clause that is not in the group by attribute list?

# SQL Query: GROUP BY details

- What happens if we do not include certain grouping attributes in the **SELECT** clause?

  Ans: The query still executes but you have no idea what the result means anymore

- What happens if we include an attribute in the **SELECT** clause that is not in the group by attribute list?

# SQL Query: GROUP BY details

- What happens if we do not include certain grouping attributes in the **SELECT** clause?

  Ans: The query still executes but you have no idea what the result means anymore

- What happens if we include an attribute in the **SELECT** clause that is not in the group by attribute list?

  Ans: In theory, this should not be allowed as you can not produce a single value for non-grouping attributes. However, some implementations returns one of the tuples.

# SQL Query: HAVING

- **HAVING** clause specifies a selection condition on groups (rather than individual tuples)

- Filters out groups that do not satisfy the group condition

- Syntax:
  SELECT      <attribute list>
  FROM        <table list>
  WHERE       <condition on the tables>
  GROUP BY  <grouping attributes>
  HAVING      <group condition>

# SQL Query: HAVING Details

- If a SQL query uses **HAVING** clause, then there **MUST** be a **GROUP BY** clause

- Group condition is a condition on a set of tuples

- Can't use non-grouping attribute inside the **HAVING** clause

- Most common form of group condition is: SetFunction(<attr>) RelationalOperator <value>

# SQL Query: HAVING Process Order

1.  Select tuples that satisfy the **WHERE** condition

2.  Selected tuples from (1) are grouped based on their value in the grouping attributes

3.  Filter groups so only those satisfying the condition are left

4.  Set functions in the **SELECT** clause are applied to these groups

# SQL Example: HAVING

For each project on which more than two employees work, retrieve the project number, project name, and the number of employees who work on that project

```
SELECT      pnumber, pname, COUNT(*)
FROM        project, works_on
WHERE       pnumber = pno
GROUP BY  pnumber, pname
HAVING      COUNT(*) > 2;
```

# SQL Example: HAVING (2)

For each department with at least 2 employees, find the department name, and the number of employees in that department that earns greater than $40K

```
SELECT      dname, COUNT(ssn)
FROM        department, employee
WHERE        dnumber = dno
   AND       salary > 40000
GROUP BY  dname
HAVING      COUNT(ssn) > 2;
```

Is this right? What does it return?

# SQL Example: HAVING (2)

- Previous query only counts the number of departments that have at least 2 employees that earn more then $40K.

- SELECT    dname, COUNT(ssn)
  FROM     employee, department
  WHERE    dno = dnumber
      AND    dno IN ( SELECT    dno
                   FROM      employee
                   GROUP BY  dno
                   HAVING    COUNT(ssn) >= 2)
      AND    salary > 40000
  GROUP BY dname

# Summary of SQL Queries

SELECT      [DISTINCT] <attribute list>
FROM        <table list>
[WHERE      <condition on the tables>]
[GROUP BY <grouping attributes>]
[HAVING     <group condition>]
[ORDER BY  <attribute list>   ASC | DESC]
[LIMIT      <number of tuples>]

This has every possible clause of a SQL command included

# Exercises: Company Database (6)

- What are the SSN of the employees who work on at least 2 projects?

- What is the name of the project where the employees have worked the highest number of hours (total)?

- Which department has the highest number of dependents?

- What department(s) has no projects?

# MySQL Workbench

- Open source, integrated development environment for MySQL database system

  - SQL Editor

  - Data modeling

  - Data administration + performance monitoring

- Works on Windows, Linux, Mac OS X

- https://www.mysql.com/products/workbench/

# MySQL Workbench DEMO

# SQL Queries: Recap

- Basic query form

- Useful operators:

  - *, DISTINCT, IN, LIKE, ORDER BY, LIMIT, IS NULL

- Multi-table queries

  - Join

  - Aliasing and qualification

  - Nested queries

- Additional operators and commands

  - Set operations, GROUP BY, HAVING