Chapter 3: Introduction to SQL

Database System Concepts, 6th Ed.

Chapter 3: Introduction to SQL

- Overview of the SQL Query Language
- Data Definition
- Basic Query Structure
- Additional Basic Operations
- Set Operations
- Null Values
- Aggregate Functions
- Nested Subqueries
- Modification of the Database

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86, SQL-89, SQL-92
  - SQL:1999, SQL:2003, SQL:2008
- 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.

Overview of SQL Query Language

- The SQL Language has several parts:
- 1. **Data-definition Language (DDL):** Provides commands for
  - Defining relation schemas
  - Deleting relations schemas and
  - Modifying relation schemas
- 2. **Data-manipulation Language (DML):** Provides the ability to
  - Query information from the database and
  - Insert tuple into
  - Delete tuples from
  - Modify tuples in the database
- 3. **Integrity:** The SQL DDL includes commands for specifying integrity constraints the data stored in the database must satisfy

Overview of SQL Query Language

- Domain constraints (data types and lengths)
- Null
- Keys (Primary key, Unique key)
- Referential integrity constraints (Foreign key)
- Check Constraint

- 4. **View definition:** SQL DDL includes commands for defining views
- 5. **Transaction Control:** SQL includes commands for specifying beginning and ending of transactions
- 6. **Embedded SQL and dynamic SQL:** define how SQL statements can be embedded within general-purpose programming languages, such as C, C++ and Java
- 7. **Authorization:** SQL DDL includes commands (**grant**) for specifying access rights to relations and views.

Data Definition Language

The set of relations in a database must be specified to the system by means of a **data-definition language (DDL).** The SQL DDL allows the specification of not only a set of relations , but also information about each relation, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- And as we will see later, also other information such as
  - The set of indices to be maintained for each relations.
  - Security and authorization information for each relation.
  - The physical storage structure of each relation on disk.

Domain Types in SQL

- The SQL standard supports a variety of built-in types:
- **char(n).** Fixed length character string, with user-specified length *n.*
- **varchar(n).** Variable length character strings, with user-specified maximum length *n.*
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.
- **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.

Create Table Construct

- An SQL relation is defined using the **create table** command:

  create table *r* (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - *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
- Example:

  create table *instructor* (
  ID **char**(5),
  name **varchar**(20) **not null,**
  dept_name **varchar**(20),
  salary **numeric**(8,2))
- **insert into** *instructor* **values** (‘10211’, ’Smith’, ’Biology’, 66000);
- **insert into** *instructor* **values** (‘10211’, null, ’Biology’, 66000);

Integrity Constraints in Create Table

- **not null**
- **primary key** (A_1, ..., A_n)
- **foreign key** (A_m, ..., A_n) **references** *r*

Example: Declare *dept_name* as the primary key for *department*

.

create table *instructor* (
ID **char**(5),
name **varchar**(20) **not null,**
dept_name **varchar**(20),
salary **numeric**(8,2),
primary key (*ID*),
foreign key *(dept_name*) **references** *department)*

primary key declaration on an attribute automatically ensures **not null**

And a Few More Relation Definitions

- **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)*);
- **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*);
  - Note: *sec_id* can be dropped from primary key above, to ensure a student cannot be registered for two sections of the same course in the same semester

And more still

- **create table** *course* (
  course_id **varchar**(8) **primary key**,
  title **varchar(**50),
  dept_name **varchar**(20),
  credits **numeric**(2,0),
  foreign key *(dept_name*) **references** *department)*);
  - Primary key declaration can be combined with attribute declaration as shown above

Drop and Alter Table Constructs

- **drop table** *student*
  - Deletes the table and its contents
- **delete from** *student*
  - Deletes all contents of table, but retains table
- **alter table**
  - **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 tuples in the relation are assigned *null* as the value for the new attribute.
  - **alter table** *r* **drop** *A*
    - where *A* is the name of an attribute of relation *r*
    - Dropping of attributes not supported by many databases

Basic Query Structure

- The SQL **data-manipulation language (DML)** provides the ability to query information, and insert, delete and update tuples
- A typical SQL query has the form:

  select A_1, A_2, ..., A_n
  from r_1, r_2, ..., r_m
  where P

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.
- The query takes as its input the relations listed in the **from** clause, operates on them as specified in the **where** and **select** clauses, and then produces a relation as the result.
- So, the result of an SQL query is a relation.

The select Clause

- **Queries on Single Relation**
- The **select** clause list the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all instructors:

  select *name*
  from *instructor*

- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g. *Name* ≡ *NAME* ≡ *name*
  - Some people use upper case wherever we use bold font.

The select Clause (Cont.)

- In mathematical definition, a relation is a set. Thus, duplicate tuples would never appear in relations. In practice, duplicate elimination is time-consuming. Therefore, SQL allows duplicates in relations as well as in the results of SQL expressions.
- To force the elimination of duplicates, insert the keyword **distinct** after select**.**
- Find the names of all departments with instructor, and remove duplicates

  select distinct *dept_name*
  from *instructor*

- The keyword **all** specifies that duplicates not be removed.

  select all *dept_name*
  from *instructor*

The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”

  select *
  from *instructor*

- The **select** clause can contain arithmetic expressions involving the operation, +, –, ∗, and /, and operating on constants or attributes of tuples.
- The query:

  select *ID, name, dept_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.
- However, that it does not result in any change to the *instructor* relation.

The where Clause

- The **where** clause specifies conditions that the result must satisfy (Corresponds to the selection predicate of the relational algebra).
- To find all instructors in Comp. Sci. dept with salary > 80000

  select *name*
  from *instructor*
  where *dept_name* = *'*Comp. Sci.' **and** *salary* > 80000
- SQL allows the use of the logical connectives **and, or,** and **not** in the **where** clause.
- The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>.
- SQL allows us to use the comparison operators to compare strings and arithmetic expressions, as well as special types, such as date types.

The from Clause

- **Queries on Multiple Relations**
- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *instructor X teaches*

  select ∗
  from *instructor, teaches*

  - generates every possible instructor – teaches pair, with all attributes from both relations
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra)

Cartesian Product: *instructor X teaches*

instructor

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000

teaches

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

inst.ID	name	dept_name	salary	teaches.ID	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2009
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2009
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2010
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2010
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

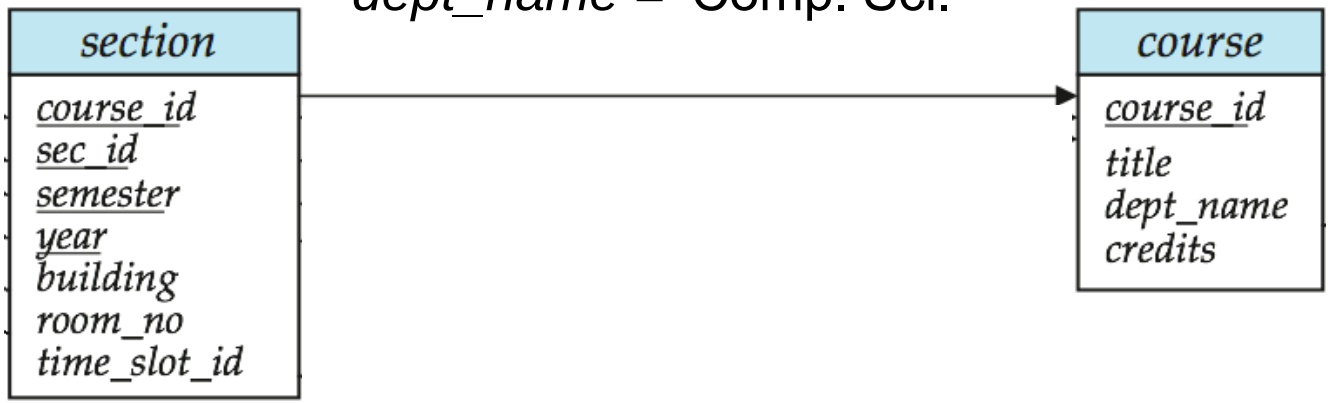
Joins

- For all instructors who have taught some course, find their names and the course ID of the courses they taught.

```
select name, course_id
 from instructor, teaches
 where   instructor.ID = teaches.ID
```

- Find the course ID, semester, year and title of each course offered by the Comp. Sci. department

```
select section.course_id, semester, year, title
 from section, course
 where   section.course_id = course.course_id  and
          dept_name = 'Comp. Sci.'
```



Typical SQL Query Form

- We now consider the general case of SQL queries involving multiple relations. As we have seen earlier, an SQL query can contain three types of clauses, the **select** clause, the **from** clause, and the **where** clause**.**
- The role of each clause is as follows:
- The **select** clause is used to list the attributes desired in the result of a query.
- The **from** clause is a list of the relations to be accessed in the evaluation of the query.
- The **where** clause is a predicate involving attributes of the relation in the from clause.

Typical SQL Query Form

- 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**;**

- Each A_i represents an attribute, and each r_i a relation. P is a predicate. If the **where** clause is omitted, the predicate P is ***true.***

Natural Join

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column
- **select** *
  from *instructor* **natural join** *teaches*;

ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010

Natural Join Example

- List the names of instructors along with the course ID of the courses that they taught.
  - **select** *name*, *course_id*
    from *instructor, teaches*
    where *instructor.ID* = *teaches.ID*;
  - **select** *name*, *course_id*
    from *instructor* **natural join** *teaches*;

Natural Join (Cont.)

- Danger in natural join: beware of unrelated attributes with same name which get equated incorrectly
- List the names of instructors along with the the titles of courses that they teach
  - Incorrect version (makes course.dept_name = instructor.dept_name)
    - **select** *name*, *title*
      from *instructor* **natural join** *teaches* **natural join** *course*;
  - Correct version
    - **select** *name*, *title*
      from *instructor* **natural join** *teaches*, *course*
      where *teaches.course_id = course.course_id*;
  - Another correct version
    - **select** *name*, *title*
      from (*instructor* **natural join** *teaches*)
      join *course* **using**(*course_id*);

The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:

  old-name **as** *new-name*

- E.g.
  - **select** *ID, name, salary/12* **as** *monthly_salary*
    from *instructor*

- Find the names of all instructors who have a higher salary than some instructor in 'Comp. Sci'.
  - **select distinct** *T. name*
    from *instructor* **as** *T*, *instructor* **as** *S*
    where *T.salary* > *S.salary* **and** *S.dept_name* = *'Comp. Sci.'*
- Keyword **as** is optional and may be omitted

  instructor **as** *T* ≡ *instructor T*
  - Keyword **as** must be omitted in Oracle

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 matches any character.
- Find the names of all instructors whose name includes the substring “dar”.

```
select name
from instructor
where name like '%dar%'
```

- Match the string “100 %”

```
like ‘100 \%'  escape  '\'
```

String Operations (Cont.)

- Patters 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 vice versa)
  - finding string length, extracting substrings, etc.

Ordering the Display of Tuples

- List in alphabetic order the names of all instructors

  select distinct *name*
  from *instructor*
  order by *name*

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by** *name* **desc**
- Can sort on multiple attributes
  - Example: **order by** *dept_name, name*

Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between $90,000 and $100,000 (that is, ≥ $90,000 and ≤ $100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000
- Tuple comparison
  - **select** *name*, *course_id*
    from *instructor*, *teaches*
    where (*instructor*.*ID*, *dept_name*) = (*teaches*.*ID*, 'Biology');

Set Operations

- Find courses that ran in Fall 2009 or in Spring 2010

(**select** *course_id* **from** *section* **where** *sem* = ‘Fall’ **and** *year* = 2009)
union
(**select** *course_id* **from** *section* **where** *sem* = ‘Spring’ **and** *year* = 2010)

- Find courses that ran in Fall 2009 and in Spring 2010

(**select** *course_id* **from** *section* **where** *sem* = ‘Fall’ **and** *year* = 2009)
intersect
(**select** *course_id* **from** *section* **where** *sem* = ‘Spring’ **and** *year* = 2010)

- Find courses that ran in Fall 2009 but not in Spring 2010

(**select** *course_id* **from** *section* **where** *sem* = ‘Fall’ **and** *year* = 2009)
except
(**select** *course_id* **from** *section* **where** *sem* = ‘Spring’ **and** *year* = 2010)

Set Operations

- Set operations **union**, **intersect**, and **except**
  - Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the corresponding multiset versions **union all, intersect all** (Oracle dose not support) and **except all** (Oracle dose not support)**.**

  Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs:
  - *m* + *n* times in *r* **union all** *s*
  - min(*m,n)* times in *r* **intersect all** *s*
  - max(0, *m* – *n)* times in *r* **except all** *s*

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 result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- The predicate **is null** can be used to check for null values.
  - Example: Find all instructors whose salary is null.

select *name*
from *instructor*
where *salary* **is null**

Null Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
  - Example*: 5 < null or null <> null or null = null*
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*,
    (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - AND: *(true* **and** *unknown)* = *unknown,*
    (false **and** *unknown) = false,*
    (unknown **and** *unknown) = unknown*
  - NOT*: (***not** *unknown) = unknown*
  - "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Aggregate Functions

- *Aggregate* functions are functions that take a collection (a set or multiset) of values as input and return a single value.
- SQL offers five built-in aggregate functions:

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

- The input to **sum** and **avg** must be a collection of numbers, but the other operators can operate on collections of nonnumeric data types, such as strings, as well.

Basic Aggregate Functions

- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*)
    from *instructor*
    where *dept_name*= ’Comp. Sci.’;
- Retaining duplicates is important in computing an average. There are cases where we must eliminate duplicates before computing an aggregate function.
- Find the total number of instructors who teach a course in the Spring 2010 semester
  - **select count** (**distinct** *ID*)
    from *teaches*
    where *semester* = ’Spring’ **and** *year* = 2010
- Because of the keyword **distinct** preceding ID, even if an instructor teaches more than one course, she is counted only once in the result.

Basic Aggregate Functions (Cont.)

- We use the aggregate function **count** frequently to count the number of tuples in a relation. The notation for this function in SQL is **count (*)**
- Find the number of tuples in the *course* relation
  - **select count** (*)
    from *course*;
- SQL does not allow the use of **distinct** with **count (*).**
- It is legal to use distinct with **max** and **min**, even though the result does not change.
- We can use the keyword **all** in place of **distinct** to specify duplicate retention, but, since all is the default, there is no need to do so.

Aggregation with Grouping

- There are circumstances where we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this wish in SQL using the **group by** clause.
- The attribute or attributes given in the **group by** clause are used to form groups.
- Tuples with the same value on all attributes in the **group by clause** are placed in one group.

Aggregation with Grouping (Cont.)

- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*) as *avg_salary_dept_wise*
    from *instructor*
    group by *dept_name*;
  - Note: departments with no instructor will not appear in result

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 with Grouping (Cont.)

- In contrast, consider the query “Find the average salary of all instructors.”
  - **select avg** (*salary*) as *avg_salary*
    from *instructor*
- In this case the **group by** clause has been omitted, so the entire relation is treated as a single group.
- Find the number of instructors in each department who teach a course in the Spring 2010 semester.
  - **select** *dept_name,* **count** (**distinct** *ID*) *as instr_count*
    from *instructor* **natural join** *teaches*
    where *semester = ‘Spring’* **and** *year = 2010*
    group by *dept_name;*

Aggregation with Grouping (Cont.)

- When an SQL query uses grouping, it is important to ensure that the only attributes that appear in the select statement without being aggregated are those that are present in the **group by** clause.
- In other words, any attribute that is not present in the **group by** clause must appear only inside an aggregate function if it appears in the **select** clause, otherwise the query is treated as erroneous.
  - /* erroneous query */
    select *dept_name*, *ID*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;
- Each instructor in a particular group (defined by *dept_name)* can have a different *ID*, and since only one tuple is output for each group, there is no unique way of choosing which *ID* value to output.

Aggregate Functions – Having Clause

- At times, it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those departments where the average salary of the instructors is more than $42,000. This condition does not apply to a single tuple; rather, it applies to each group constructed by the group by clause.
- Find the names and average salaries of all departments whose average salary is greater than 42000

```
select dept_name, 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

Aggregate Functions – Having Clause

- As was the case for the **select** clause, any attribute that is present in the having clause without being aggregated must appear in the **group by** clause, otherwise the query is treated as erroneous.
- The meaning of a query containing aggregation, **group by**, or **having** clauses is defined by the following sequence of operations:
- 1. As was the case for queries without aggregation, the **from** clause is first evaluated to get a relation.
- 2. If a **where** clause is present, the predicate in the **where** clause is applied on the result relation of the **from** clause**.**
- 3. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause if it is present. If the **group by** clause is absent, the entire set of tuples satisfying the **where** predicate is treated as being in one group.

Aggregate Functions – Having Clause

- 4. The **having** clause, if it is present, is applied to each group; the groups that do not satisfy the **having** clause predicate are removed.
- 5. The **select** clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.
- For each course section offered in 2009, find the average total credits (*tot_cred)* of all students enrolled in the section, if the section had at least 2 students.

select *course_id, semester, year, sec_id,* ***avg*** *(tot_cred)*
from *takes* **natural join** *student*
where *year = 2009*
group by *course_id, semester, year, sec_id*
having count (*ID*) >= *2;*

Null Values and Aggregates

- Null values, when they exist, complicate the processing of aggregate operators.
- Assume that some tuples in the *instructor* relation have a null value for *salary.*
- Total all salaries

  select sum (*salary*)
  from *instructor*

  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes
- What if collection has only null values?
  - count returns 0
  - all other aggregates return null

Null Values and Aggregates

- In general, aggregate functions treat nulls according to the following rule:
- All aggregate functions except **count (*)** ignore null values in their input collection.
- As a result of null values being ignored, the collection of values may be empty.
- The count of an empty collection is defined to be 0, and all other aggregate operations return a value of null when applied on an empty collection.

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for **set membership**, make **set comparisons**, and determine **set cardinality**, by nesting subqueries in the **where** clause.
- Subqueries can also be nested in the **from** clause.
- We will also see how a class of subqueries called **scalar subqueries** can appear wherever an expression returning a value can occur (even in the select clause).
- SQL allows testing tuples for membership in a relation. The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause. The **not in** connective tests for the absence of set membership.

Set Membership: Example Query

- Find courses offered in Fall 2009 and in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
        course_id in (select course_id
                      from section
                      where semester = 'Spring' and year= 2010);
```

- Find courses offered in Fall 2009 but not in Spring 2010

```
select distinct course_id
from section
where semester = 'Fall' and year= 2009 and
        course_id  not in (select course_id
                           from section
                           where semester = 'Spring' and year= 2010);
```

Set Membership: Example Query

- The **in** and **not in** operators can also be used on enumerated sets.
- Find the names of instructors whose names are neither “Mozart” nor “Einstein”.

  Select distinct *name*
  from *instructor*
  Where *name* **not in** (‘Mozart’, ’Einstein’);

- In the preceding examples, we tested membership in a one-attribute relation.
- It is also possible to test for membership in an arbitrary relation in SQL.

Set Membership: Example Query

- 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 : Example Query

- 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';
```

- Same query using > **some** clause

```
select name
from instructor
where salary > some (select salary
                     from instructor
                     where dept_name = 'Biology');
```

Definition of Some Clause

- F <comp> **some** $r \Leftrightarrow \exists\, t \in r$ such that (F <comp> t)
  Where <comp> can be: $<,\ \leq,\ >,\ =,\ \neq$

(5 < **some** | 0 / 5 / 6 |) = true (read: 5 < some tuple in the relation)

(5 < **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (since 0 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≢ **not in**

Set Comparison : Example Query

- 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');
```

- Find the departments that have the highest average salary.

```
select dept_name
from instructor
group by dept_name
having avg (salary) >= all (select avg (salary)
                            from instructor
                            group by dept_name);
```

Definition of all Clause

- F <comp> **all** $r \Leftrightarrow \forall\ t \in r$ (F <comp> *t)*

(5 < **all** {0, 5, 6}) = false

(5 < **all** {6, 10}) = true

(5 = **all** {4, 5}) = false

(5 ≠ **all** {4, 6}) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**

However, (= **all**) ≢ **in**

Test for Empty Relations

- SQL includes a feature for testing whether a subquery has any tuples in its result.
- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- Yet another way of specifying the query “Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester”

```
select course_id
from section as S
where semester = 'Fall' and year= 2009 and
        exists (select *
                from section as T
                where semester = 'Spring' and year= 2010
                        and S.course_id= T.course_id);
```

Correlation Variables

- The above query also illustrates a feature of SQL where a correlation name from an outer query (*S* in the above query), can be used in a subquery in the **where** clause. A subquery that uses a correlation name from an outer query is called a **correlated subquery.**
- **Correlation name** or **correlation variable**
- In the above query, *T* and *S* can be thought of as copies of the relation *section*, but more precisely, they are declared as aliases, that is as alternative names, for the relation *section.*
- An identifier, such as *T* and *S*, that is used to rename a relation is referred to as a **correlation name** in the SQL standard, but is also commonly referred to as a **table alias**, or a **correlation variable,** or a **tuple variable.**

Not Exists

- 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));
```

  - 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 returns the value true if the argument subquery contains no duplicate tuples.
  - (Evaluates to “true” on an empty set)
- Find all courses that were offered at most once in 2009

```
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 = 2009);
```

-
```
select T.course id
from course as T
where 1 = (select count(R.course id)
             from section as R
             where T.course id= R.course id and
             R.year = 2009);
```

Test for Presence of Duplicate Tuples

- We can test for the existence of duplicate tuples in a subquery by using the **not unique** construct.
- Find all courses that were offered at least twice in 2009
- **select** *T.course_id*
  from *course* **as** *T*
  where not unique (**select** *R.course_id*
  from *section* **as** *R*
  where *T.course_id*= *R.course_id*
  and *R.year* = 2009);
- Formally, the **unique** test on a relation is defined to fail if and only if the relation contains two tuples *t1* and *t2* such that *t1 = t2.*
- Since the test *t1 = t2* fails if any of the fields of *t1 or t2* are null, it is possible for **unique** to be true even if there are multiple copies of a tuple, as long as at least one of the attributes of the tuple is null.

Subqueries in the From Clause

- SQL allows a subquery expression to be used in the **from** clause
- 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;
```

- Note that we do not need to use the **having** clause
- We can give the subquery result relation a name, and rename the attributes, using the **as** clause

```
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;
```

Subqueries in the From Clause (Cont.)

- Nested subqueries in the **from** clause are supported by most but not all SQL implementations. However, some SQL implementations, notably Oracle, do not support renaming of the result relation in the **from** clause.
- As another example, suppose we wish to find the maximum across all departments of the total salary at each department. The **having** clause does not help us in this task, but we can write this query easily by using a subquery in the **from** clause, as follows:
- **select max** (*tot_salary*)
  from (**select** *dept_name*, **sum**(*salary*)
  from *instructor*
  group by *dept_name*)
  as *dept_total* (*dept_name*, *tot_salary*)

Subqueries in the From Clause (Cont.)

- Nested subqueries in the **from** clause cannot use correlation variables from other relations in the **from** clause. However, SQL:2003 allows a subquery in the **from** clause that is prefixed by the **lateral** keyword to access attributes of preceding tables or subqueries in the **from** clause.
- Find the names of each instructor, along with their salary and the average salary in their department
- **select** *name*, *salary*, *avg_salary*
  from *instructor I1*,
  lateral (**select avg**(*salary*) as *avg_salary*
  from *instructor I2*
  where *I2.dept_name= I1.dept_name*);
- Note: lateral is part of the SQL standard, but is not supported on many database systems (IBM DB2 supports it); some databases such as SQL Server offer alternative syntax

With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.
- Find all departments with the maximum budget

```
with max_budget (value) as
    (select max(budget)
     from department)
select budget
from department, max_budget
where department.budget = max_budget.value;
```

- We could have written the above query by using a nested subquery in either the **from** clause or the **where** clause. However, using nested subqueries would have made the query harder to read and understand. The **with** clause makes the query logic clearer; it also permits a view definition to be used in multiple places within a query.

Complex Queries using With Clause

- With clause is very useful for writing complex queries
- Supported by most database systems, with minor syntax variations
- 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

- SQL allows subqueries to occur wherever an expression returning a value is permitted, provided the subquery returns only one tuple containing a single attribute; such subqueries are called **scalar subqueries.**
- Lists all departments along with the number of instructors in each department
- E.g. **select** *dept_name*,
  (**select count**(*)
  from *instructor*
  where *department*.*dept_name* = *instructor*.*dept_name*)
  as *num_instructors*
  from *department*;
- E.g. **select** *name*
  from *instructor*
  where *salary * 10 >*
  (**select** *budget* **from** *department*
  where *department*.*dept_name* = *instructor*.*dept_name*)

Scalar Subquery

- Scalar subqueries can occur in **select, where,** and **having** clauses.
- **Scalar subqueries** may also be defined without aggregates.
- It is not always possible to figure out at compile time if a subquery can return more than one tuple in its result; if the result has more than one tuple when the subquery is executed, a run-time error occurs.
- Note that technically the type of a scalar subquery result is still a relation, even if it contains a single tuple.
- However, when a scalar subquery is used in an expression where a value is expected, SQL implicitly extracts the value from the single attribute of the single tuple in the relation, and returns that value.

Modification of the Database

- Deletion of tuples from a given relation
- Insertion of new tuples into a given relation
- Updating values in some tuples in a given relation
- A delete request is expressed in much the same way as a query. We can delete only whole tuples; we cannot delete values on only particular attributes. SQL expresses a deletion by

  delete from *r*

  where *P;*

  - where *P* represents a predicate and *r* represents a relation.
- The **delete** statement first finds all tuples *t* in *r* for which P(*t*) is true, and then deletes them from *r*. The **where** clause can be omitted, in which case all tuples in *r* are deleted.

Modification of the Database – Deletion

- A **delete** command operates on only one relation. If we want to delete tuples from several relations, we must use one **delete** command for each relation.
- The predicate in the **where** clause may be as complex as a **select** command's **where** clause. At the other extreme, the where clause may be empty.
- 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 with a salary between $13,000 and $15,000.
- **delete from** *instructor*

  where *salary* **between** 13000 **and** 15000**;**
- 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 deposit, 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 **avg** or retesting the tuples)

Modification of the Database – 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 *tot_creds* set to null

  insert into *student*
  values (’3003’, ’Green’, ’Finance’, *null*);

- Most relational database products have special “bulk loader” utilities to insert a large set of tuples into a relation. These utilities allow data to be read from formatted text files, and can execute much faster than an equivalent sequence of insert statements.

Insertion (Cont.)

- More generally, we might want to insert tuples on the basis of the result of a query.
- 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.
- **insert into** *instructor*

  select *ID, name, dept_name,* 18000

  from *student*

  where *dept_name* = ‘Music’ **and** *tot_cred* > 144;
- The **select from where** statement is evaluated fully before any of its results are inserted into the relation (otherwise queries like

  insert into *student* **select** * **from** *student*

  would cause problems, if *student* did not have any primary key defined.

Modification of the Database – Updates

- In certain situations, we may wish to change a value in a tuple without changing *all* values in the tuple. For this purpose, the **update** statement can be used.
- As we could for **insert** and **delete**, we can choose the tuples to be updated by using a query.
- Increase salaries of all instructor by 5 percent

  update *instructor*
  set *salary* = *salary* * 1.05
- Again, increase the salary of the instructors by 7 percent with salary of less than $70,000,

  update *instructor*
  set *salary* = *salary* * 1.05
  where *salary* <= 70000;

Modification of the Database – Updates

- In general, the **where** clause of the update statement may contain any construct legal in the **where** clause of the select statement (including nested selects).
- As with **insert** and **delete**, a nested **select** within an **update** statement may reference the relation that is being updated.
- Give a 5 percent salary raise to instructors whose salary is less than average

update *instructor*
set *salary* = *salary* * 1.05
where *salary* < (**select avg** (*salary*)
from *instructor*) ;

Modification of the Database – Updates

- Increase salaries of instructors whose salary is over $90,000 by 3%, and all others receive a 5% raise
  - Write two **update** statements:

```
update instructor
    set salary = salary * 1.03
    where salary > 100000;
update instructor
    set salary = salary * 1.05
    where salary <= 100000;
```

  - The order of the two update statements is important (why?)
- SQL provides a **case** construct that we can use to perform both the updates with a single **update** statement, avoiding the problem with the order of updates. (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
```

- The general form of the case statement is as follows.

```
case
when pred1 then result1
when pred2 then result2
 ..........
when predn then resultn
else result0
end
```

Updates with Scalar Subqueries

- Consider an update where we set the *tot_cred* attribute of each student tuple to the sum of the credits of courses successfully completed by the student (scalar subquery used in the **set** clause).
- We assume that a course is successfully completed if the student has a grade that is not 'F' or null.

```
update student S
   set tot_cred = ( select sum(credits)
                    from takes natural join course
                    where S.ID= takes.ID and
                          takes.grade <> 'F' and
                          takes.grade is not null);
```

- Sets *tot_cred* to null for students who have not taken any course
- To replace null with 0, instead of "**select sum**(*credits*)", use:

```
select  case
          when sum(credits) is not null then sum(credits)
          else 0
        end
```

End of Chapter 3

Advanced SQL Features**

- Create a table with the same schema as an existing table:

  create table *temp_account* **like** *account*

  or

  create table *temp_account* **as**

  select *

  from *account*

Figure 3.02

name
Srinivasan
Wu
Mozart
Einstein
El Said
Gold
Katz
Califieri
Singh
Crick
Brandt
Kim

Figure 3.03

dept_name
Comp. Sci.
Finance
Music
Physics
History
Physics
Comp. Sci.
History
Finance
Biology
Comp. Sci.
Elec. Eng.

Figure 3.04

name
Katz
Brandt

Figure 3.05

name	*dept_name*	*building*
Srinivasan	Comp. Sci.	Taylor
Wu	Finance	Painter
Mozart	Music	Packard
Einstein	Physics	Watson
El Said	History	Painter
Gold	Physics	Watson
Katz	Comp. Sci.	Taylor
Califieri	History	Painter
Singh	Finance	Painter
Crick	Biology	Watson
Brandt	Comp. Sci.	Taylor
Kim	Elec. Eng.	Taylor

Figure 3.07

name	*Course_id*
Srinivasan	CS-101
Srinivasan	CS-315
Srinivasan	CS-347
Wu	FIN-201
Mozart	MU-199
Einstein	PHY-101
El Said	HIS-351
Katz	CS-101
Katz	CS-319
Crick	BIO-101
Crick	BIO-301
Brandt	CS-190
Brandt	CS-190
Brandt	CS-319
Kim	EE-181

Figure 3.08

ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010
83821	Brandt	Comp. Sci.	92000	CS-190	1	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-190	2	Spring	2009
83821	Brandt	Comp. Sci.	92000	CS-319	2	Spring	2010
98345	Kim	Elec. Eng.	80000	EE-181	1	Spring	2009

Figure 3.09

course_id
CS-101
CS-347
PHY-101

Figure 3.10

course_id
CS-101
CS-315
CS-319
CS-319
FIN-201
HIS-351
MU-199

Figure 3.11

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

Figure 3.12

course_id
CS-101

Figure 3.13

course_id
CS-347
PHY-101

Figure 3.16

dept_name	*count*
Comp. Sci.	3
Finance	1
History	1
Music	1

Figure 3.17

dept_name	*avg(salary)*
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000