CS 4092 Database Design and Development (DDD)

04: SQL

Seokki Lee

Slides are adapted from:

Database System Concepts, 6th & 7th Ed. ©Silberschatz, Korth and Sudarshan

Outline

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

What is SQL?

- Structured Query Language (SQL)
  - Most widely used relational database language
  - Defining the structure of the data
  - Modifying data in the database
  - Retrieving data from the database
  - Specifying constraints

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory.
  - First system to demonstrate that a relational database management system could provide good transaction processing performance
- Renamed to **Structured Query Language (SQL)** as evolved
- ANSI and ISO standard SQL:
  - SQL-86, SQL-89, SQL-92
  - SQL:1999, SQL:2003, SQL:2008, SQL:2011, SQL:2016, SQL:2019
- 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.

Several Aspects of SQL

- Data Definition Language (DDL)
  - Integrity
  - View definition
- Data Manipulation Language (DML)
- Transaction Control
- Embedded SQL and Dynamic SQL
- Authorization

Data Definition Language

- The SQL **data-definition language (DDL)** allows the specification of information about relations, including:
  - The schema for each relation
  - The type of values associated with each attribute
  - The Integrity constraints
  - The set of indicies to be maintained for each relation
  - …

Domain Types in SQL

- **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 *d* digits to the right of decimal point.
  - e.g., numeric(3,1), allows 44.5 to be stored
- **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 $\boldsymbol{A_i}$ is an attribute name in the schema of relation ***r***
  - $\boldsymbol{D_i}$ is the data type of values in the domain of attribute $\boldsymbol{A_i}$

Create Table Construct

- Example:

```
create table instructor (
        ID              char(5),
        name            varchar(20),
        dept_name       varchar(20),
        salary          numeric(8,2))
```

Integrity Constraints in Create Table

- Types of integrity constraints
  - **primary key** (A_1, ..., A_n)
    - Automatically ensures **not null**
  - **foreign key** (A_m, ..., A_n) **references** *r*
  - **not null**
- SQL prevents any update to the database that violates an integrity constraint.
- Example:

```
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);
```

Example: Integrity Constraints in Create Table

- **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);*

- **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*);

And a Few More Relation Definitions

- **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*);
  - A **problem** with the primary key constraint?

And a Few More Relation Definitions

- **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*);

Updates to Tables

- **Insert**
  - **insert into** *instructor* **values** ('10211', 'Smith', 'Biology', 66000);
  - Data manipulation language (DML) but placed here for convenience

- **Copy**
  - **copy** *table_name* **from** *'path/to/csv'* **delimeter** ',' csv header;
  - **copy** *table_name* **to** *'path/to/csv'* **delimeter** ',' csv header;
  - Useful when inserting/exporting large data into a table/csv file

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
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Updates to Tables

- **Delete**
  - Remove all tuples from the *instructor* relation
  - **delete from** *instructor*
  - Data manipulation language (DML) but placed here for convenience
- **Drop Table**
  - **drop table** *r*

Updates to Tables

- **Alter**
  - Renaming tables and columns
  - Adding and dropping columns
  - Adding and dropping constraints
  - Changing schema
  - …

Updates to Tables

- **Alter**
  - **alter table** *r* **rename to** *s*
    - where *s* is the new name of the relation *r*
  - **alter table** *r* **add** *A D*
    - where *A* is the name of the attribute to be added to relation *r* and *D* is the type of *A.*
    - All exiting 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.

Outline

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

Basic Query Structure

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

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

  - $\boldsymbol{A_i}$ represents an attribute
  - $\boldsymbol{R_i}$ represents a relation
  - $\boldsymbol{P}$ is a predicate.
- What is the result of an SQL query?

Basic Query Structure

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

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

  - $\boldsymbol{A_i}$ represents an attribute
  - $\boldsymbol{R_i}$ represents a relation
  - $\boldsymbol{P}$ is a predicate.
- What is the result of an SQL query?
  - The result of an SQL query is a relation.

The select Clause

- The **select** clause lists 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.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword **distinct** after select**.**
- Find the department names of all instructors, and remove duplicates

  select distinct *dept_name*
  from *instructor*

- The (redundant) keyword **all** specifies that duplicates should not be removed.

  select all *dept_name*
  from *instructor*

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

The select Clause (Cont.)

- An asterisk(*) in the select clause denotes …

The select Clause (Cont.)

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

select *
from *instructor*

- Can an attribute be a literal?

The select Clause (Cont.)

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

  select *
  from *instructor*

- An attribute can be a literal with no **from** clause

  select '437'

  - Result is …

The select Clause (Cont.)

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

  select *
  from *instructor*

- An attribute can be a literal with no **from** clause

  select '437'

  - Result is a table with one column and a single row with value “437”
  - Can give the column a name using:

    select '437' **as** *FOO*

The select Clause (Cont.)

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

  select *
  from *instructor*

- An attribute can be a literal with no **from** clause

  select '437'

  - Result is a table with one column and a single row with value “437”
  - Can give the column a name using:

    select '437' **as** *FOO*

- Can an attribute be a literal with **from** clause?

The select Clause (Cont.)

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

  select *
  from *instructor*

- An attribute can be a literal with no **from** clause

  select '437'

  - Result is a table with one column and a single row with value “437”
  - Can give the column a name using:

    select '437' **as** *FOO*

- An attribute can be a literal with **from** clause

  select 'A'
  from *instructor*

  - Result is a table with…

The select Clause (Cont.)

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

  select *
  from *instructor*

- An attribute can be a literal with no **from** clause

  select '437'

  - Result is a table with one column and a single row with value “437”
  - Can give the column a name using:

    select '437' **as** *FOO*

- An attribute can be a literal with **from** clause

  select 'A'
  from *instructor*

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

The select Clause (Cont.)

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

    select *ID, name, salary/12*
    from *instructor*

    would return …

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
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

The select Clause (Cont.)

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

    select *ID, name, salary/12*
    from *instructor*

    would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.
  - Can rename "s*alary/12"* using the **as** clause:

    select *ID, name, salary/12* **as** *monthly_salary*

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
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

The select Clause (Cont.)

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

    select *ID, name, salary/12*
    from *instructor*

    would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.
  - Can rename "s*alary/12"* using the **as** clause:

    select *ID, name, salary/12* **as** *monthly_salary*
- Most systems support additional functions, e.g., substring, and allow user defined functions (UDFs).

The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the RA
- Find the Cartesian product *instructor X teaches*

select ∗
from *instructor, teaches*

The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the RA
- Find the Cartesian product *instructor X teaches*

  select ∗
  from *instructor, teaches*

  - generates every possible instructor – teaches pair, with all attributes from both relations.
  - For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)

The from Clause

- The **from** clause lists the relations involved in the query
  - Corresponds to the Cartesian product operation of the RA
- Find the Cartesian product *instructor X teaches*

  select ∗
  from *instructor, teaches*

  - generates every possible instructor – teaches pair, with all attributes from both relations.
  - For common attributes (e.g., *ID*), the attributes in the resulting table are renamed using the relation name (e.g., *instructor.ID*)
- Cartesian product not very useful directly, but useful combined with where-clause condition

The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the RA

The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the RA
- To find all instructors in Comp. Sci. dept

select *name*
from *instructor*
where *dept_name* = ‘Comp. Sci.’

The where Clause

- SQL allows the use of the logical connectives **and**, **or**, and **not**
- The operands of the logical connectives can be expressions involving the comparison operators <, <=, >, >=, =, and <>.
- Comparisons can be applied to results of arithmetic expressions.
- To find all instructors in Comp. Sci. dept with salary > 70000

  select *name*
  from *instructor*
  where *dept_name* = ‘Comp. Sci.' **and** *salary* > 70000

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, $\geq$ $90,000 and $\leq$ $100,000)

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
  - Find the names of instructors and the course's id they are teaching

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

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
  - Find the names of instructors and the course's id they are teaching
  - **select** *name*, *course_id*
    from *instructor*, *teaches*
    where (*instructor.ID*, *dept_name*) = (*teaches.ID*, 'Biology');

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

Examples

- Find the names of all instructors who have taught some course and the course_id

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

Examples

- Find the names of all instructors who have taught some course and the course_id
  - **select** *name, course_id*
    from *instructor , teaches*
    where *instructor.ID = teaches.ID*

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

Examples

- Find the names of all instructors in the Art department who have taught some course and the course_id

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

Examples

- Find the names of all instructors in the Art department who have taught some course and the course_id
  - **select** *name, course_id*
    from *instructor , teaches*
    where *instructor.ID = teaches.ID*
    and *instructor. dept_name =* 'Art'

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

Examples

- 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.'
```

Joined Relations

- **Join** operations take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join.
- The join operations are typically used as subquery expressions in the **from** clause.

Join operations – Example

- Relation *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe that
  - *prereq information is missing for CS-315*
  - *course information is missing for CS-347*

Natural Join

- Matches tuples with the same values for **all common attributes**, and **retains only one copy** of each common column
  - This is the natural join from relational algebra
- **select** *
  from *course* **natural join** *prereq*;

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101

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*;

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
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

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
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

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 titles of courses that they teach
  - **select** *name*, *title*
    from *instructor* **natural join** *teaches* **natural join** *course*;

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
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

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
32343	HIS-351	1	Spring	2010
45565	CS-101	1	Spring	2010
45565	CS-319	1	Spring	2010
76766	BIO-101	1	Summer	2009
76766	BIO-301	1	Summer	2010
83821	CS-190	1	Spring	2009
83821	CS-190	2	Spring	2009
83821	CS-319	2	Spring	2010
98345	EE-181	1	Spring	2009

course

course_id	*title*	*dept_name*	*credits*
BIO-101	Intro. to Biology	Biology	4
BIO-301	Genetics	Biology	4
BIO-399	Computational Biology	Biology	3
CS-101	Intro. to Computer Science	Comp. Sci.	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3
CS-319	Image Processing	Comp. Sci.	3
CS-347	Database System Concepts	Comp. Sci.	3
EE-181	Intro. to Digital Systems	Elec. Eng.	3
FIN-201	Investment Banking	Finance	3
HIS-351	World History	History	3
MU-199	Music Video Production	Music	3
PHY-101	Physical Principles	Physics	4

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 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?

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 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*;

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 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?

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 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*);

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
- Uses ***null*** values.

Left Outer Join

- course **natural left outer join** prereq

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

course

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

prereq

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Right Outer Join

- course **natural right outer join** prereq

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

course

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

prereq

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Full Outer Join

- course **natural full outer join** prereq

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

course

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

prereq

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

Joined Relations

- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_1, \ldots, A_n)$

Joined Relations – Examples

- *course* **inner join** *prereq* **on** *course.course_id = prereq.course_id*
  - What is the difference between the below and a natural join?

course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- *course* **left outer join** *prereq* **on** *course.course_id = prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	*null*	*null*

Joined Relations – Examples

- *course* **right outer join** *prereq* **on** *course.course_id = prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-347	*null*	*null*	*null*	CS-101	CS-347

- *course* **full outer join** *prereq* **using** (*course_id*)

course_id	*title*	*dept_name*	*credits*	*prere_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	*null*	CS-315
CS-347	*null*	*null*	*null*	CS-101	CS-347

Self Join Example

- Relation ***emp-super***

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

- Find the supervisor of “Bob”
- Find the supervisor of the supervisor of “Bob”

Self Join Example

- Relation ***emp-super***

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

- Find the supervisor of “Bob”
- Find the supervisor of the supervisor of “Bob”
- Can we find ALL the supervisors (direct and indirect) of “Bob”?

The Rename Operation

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

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

- 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.
  - Both options work in Postgres.

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.

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%'
```

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%'
```

- Same result?

```
select name
from instructor
where name like ‘_dar_'
```

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.

- Match the string “100%”

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.

- Match the string “100%”

```
like '100 \%'  escape  '\'
```

String Operations

- Patterns are case sensitive.
- Pattern matching examples:
  - 'Intro%'
  - '%Comp%'
  - '_ _ _'
  - '_ _ _ %'

String Operations

- Patterns are case sensitive.
- Pattern matching examples:
  - 'Intro%' matches any string **beginning with** “Intro”.
  - '%Comp%' matches **any string containing** “Comp” as a substring.
  - '_ _ _' matches any string of **exactly three** characters.
  - '_ _ _ %' matches any string of at **least three** characters.

String Operations

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

Case Construct

- Like case, if, and ? Operators in programming languages

case

when c_1 **then** e_1

when c_2 **then** e_2

…

[**else** e_n]

end

- Each $\mathbf{c_i}$ is a condition
- Each $\mathbf{e_i}$ is an expression
- Returns the first $\mathbf{e_i}$ for which $\mathbf{c_i}$ evaluates to *true*
  - If none of the $\mathbf{c_i}$ is true, then return $\mathbf{e_n}$ (**else**)
  - If there is no else, return *null*

Case Construct Example

- Return customer name and their group defined based on the salary, e.g., salary greater than 1M is premium and standard otherwise.
  - Input schema: *customer(name,salary)*
  - Output schema: *result(name,customer_group)*

Case Construct Example

- Return customer name and their group defined based on the salary, e.g., salary greater than 1M is premium and standard otherwise.

```
select
    name,
    case
        when salary > 1000000 then ‘premium’
        else ‘standard’
    end as customer_group
from customer
```

Ordering the Display of Tuples

- List in alphabetic order the names of all instructors (without duplicates)

  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**

Ordering the Display of Tuples

- List in alphabetic order the names of all instructors (without duplicates)

  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*

Ordering the Display of Tuples

- List in alphabetic order the names of all instructors (without duplicates)

  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*
- Order is not expressible in standard relational algebra.
  - What should be careful?

Outline

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

Set Operations

- Set operations **union**, **intersect**, and **except**
  - Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the
  - **union all**
  - **intersect all**
  - **except all**

Set Operations (Cont.)

- Find courses that ran in Fall 2017 or in Spring 2018

  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2017)
  union
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2018)

- Find courses that ran in Fall 2017 and in Spring 2018

  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2017)
  intersect
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2018)

- Find courses that ran in Fall 2017 but not in Spring 2018

  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2017)
  except
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2018)

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 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**
- The predicate **is not null** succeeds if the value on which it is applied is not null.

Null Values (Cont.)

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

- The result of any arithmetic expression involving **null** is **null**
  - 5 + **null** returns **null**

Null Values (Cont.)

- The predicate in a **where** clause can involve Boolean operations (**and**, **or**, **not**); thus the definitions of the Boolean operations need to be extended to deal with the **null**.
  - **and** : *(true* **and** *null) = null,*
    (false **and** *null) = false,*
    (null **and** *null) = null*
  - **or:** (*null* **or** *true*) *= true*,
    (*null* **or** *false*) *= null*
    (*null* **or** *unknown) = null*
  - **not:** *(**not** null) = null*

- Result of **where** clause predicate is treated as ***false*** if it evaluates to ***null***.

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a single value

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

- Most DBMS support user - defined aggregation functions
  - Covered in CS6051

Aggregate Functions Examples

- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*)
    from *instructor*
    where *dept_name*= 'Comp. Sci.';

- Find the number of tuples in the *course* relation
  - **select count** (*)
    from *course*;

Aggregate Functions Examples

- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*)
    from *instructor*
    where *dept_name*= 'Comp. Sci.';

- Find the number of tuples in the *course* relation
  - **select count** (*)
    from *course*;

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

Aggregate Functions Examples

- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*)
    from *instructor*
    where *dept_name*= 'Comp. Sci.';

- Find the number of tuples in the *course* relation
  - **select count** (*)
    from *course*;

- Find the total number of instructors who teach a course in the Spring 2018 semester
  - **select count** (**distinct** *ID*)
    from *teaches*
    where *semester* = 'Spring' **and** *year* = 2018;

Aggregate Functions – Group By

- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*) **as** *avg_salary*
    from *instructor*
    group by *dept_name*;

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

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

Aggregate Functions – Group By

- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*) **as** *avg_salary*
    from *instructor*
    group by *dept_name*;
  - What about departments with no instructor?

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

Aggregate Functions – Group By

- Find the average salary of instructors in each department
  - **select** *dept_name*, **avg** (*salary*) **as** *avg_salary*
    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 (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list
  - **select** *dept_name*, *ID*, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;

Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list
  - **select** *dept_name*, ***ID***, **avg** (*salary*)
    from *instructor*
    group by *dept_name*;

Aggregate Functions – Having Clause

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

  select *dept_name*, **avg** (*salary*) **as** *avg_salary*
  from *instructor*
  group by *dept_name*
  having avg (*salary*) > 42000;

- Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups
- Equivalent without **having**?

Aggregate Functions – Having Clause

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

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name
having avg (salary) > 42000;
```

- Equivalent

```
select dept_name, avg_salary
from
    (select dept_name, avg (salary) as avg_salary
     from instructor
     group by dept_name)
Where avg_salary > 42000;
```

Aggregate Functions – Having Clause

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

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name
having avg_salary > 42000;
```

Aggregate Functions – Having Clause

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

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name
having avg_salary > 42000;
```

  - Not valid as having clause is processed before select clause

Aggregate Functions – Having Clause

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

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name
having avg (salary) > 42000
order by avg_salary;
```

Aggregate Functions – Having Clause

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

```
select dept_name, avg (salary) as avg_salary
from instructor
group by dept_name
having avg (salary) > 42000
order by avg_salary;
```

  - Order By clause is processed after select clause.

Null Values and Aggregates

- Total all salaries

  select sum (*salary*)
  from *instructor*

  - If **null** values exist, what is the result?
  - If all the salary values are **null**, what is the result?

Null Values and Aggregates

- Total all salaries

  select sum (*salary*)
  from *instructor*

  - Above statement ignores null amounts
  - Result is null if there is no non-null amount
- Above holds for all aggregate functions?

Null Values and Aggregates

- 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

Null Values and Aggregates

- 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

Empty Relations and Aggregates

- What if the input relation is empty
- Conventions:
  - **sum**: returns *null*
  - **avg**: returns *null*
  - **min**: returns *null*
  - **max**: returns *null*
  - **count**: returns 0

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- Multiset **(bag semantics)** versions of some of the relational algebra operators

Multiset Relational Algebra

- Pure relational algebra operates on **set-semantics** (no duplicates)
- Multiset (**bag-semantics**) relational algebra retains duplicates, to match SQL semantics
  - SQL duplicate retention was initially for efficiency, but is now a feature
- Multiset relational algebra defined as follows
  - **selection**: has as many duplicates of a tuple as in the input, if the tuple satisfies the selection
  - **projection**: has as many duplicates of a tuple as in the input
  - **cross-product**: If there are m copies of t_1 in r, and n copies of t_2 in s, there are m x n copies of $t_1.t_2$ in r x s
  - Other operators similarly defined
    - **union**: $m + n$ *copies*
    - **intersection**: min(m, n) copies
    - **difference**: max(0, $m - n$) copies

Duplicates

- Example: Suppose multiset relations r_1 (A, B) and r_2 (C) are as follows:

  $r_1 = \{(1,a)\ (2,a)\}$ $\quad$ $r_2 = \{(2), (3), (3)\}$

- Then $\Pi_B(r_1)$ would be {(a), (a)}, while $\Pi_B(r_1)$ x r_2 would be

  {(a,2), (a,2), (a,3), (a,3), (a,3), (a,3)}

- SQL duplicate semantics:

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

  is equivalent to the *multiset* version of the expression:

$$\prod_{A_1,A_2,\ldots,A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

SQL and Relational Algebra

- **select** A_1, A_2, **sum**(A_3)
  from r_1, r_2, …, r_m
  where P
  group by A_1, A_2

  is equivalent to the following expression in multiset relational algebra

SQL and Relational Algebra

- **select** A_1, A_2, **sum**(A_3)
  from r_1, r_2, …, r_m
  where P
  group by A_1, A_2

  is equivalent to the following expression in multiset relational algebra

$$_{A1,\ A2}\mathcal{G}_{\textbf{sum}(A3)}\ (\sigma_{P}\ (r_1 \times r_2 \times .. \times r_m)))$$

SQL and Relational Algebra

- More generally, the non-aggregated attributes in the **select** clause may be a subset of the **group by** attributes, in which case the equivalence is as follows:

  select A_1, **sum**(A_3) *AS sumA3*
  from $r_1, r_2, \ldots, r_m$
  where P
  group by A_1, A_2

  is equivalent to the following expression in multiset relational algebra

SQL and Relational Algebra

- More generally, the non-aggregated attributes in the **select** clause may be a subset of the **group by** attributes, in which case the equivalence is as follows:

  select A_1, **sum**(A_3) *AS sumA3*
  from $r_1, r_2, \ldots, r_m$
  where P
  group by A_1, A_2

  is equivalent to the following expression in multiset relational algebra

$$\Pi_{A1,sumA3}\left(\ _{A1,A2}\mathcal{G}\ \textbf{sum}(A3)\ \textbf{as}\ \text{sumA3}(\sigma_P(r_1 \times r_2 \times .. \times r_m))\right)$$

Outline

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

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.
- The nesting can be done in the following SQL query

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

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.
- The nesting can be done in the following SQL query

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

  as follows:

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

    B <operation> (subquery)

    B is an attribute and <operation> to be defined later.
  - **Select** clause**:**

    A_i can be replaced with a subquery that generates a single value.

Subqueries in the From Clause

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

Subqueries 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

Subqueries 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

- Another way to write the above query (Hint: renaming)

Subqueries 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

- Another way to write the above query (Hint: renaming)

```
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

- Find the maximum across all departments of the total salary at each department

Subqueries in the From Clause

- Find the maximum across all departments of the total salary at each department

```
select max(tot_salary)
from (select dept_name, sum(salary)
      from instructor
      group by dept_name) as dept_total(dept_name, tot_salary);
```

Correlated Subqueries in the From Clause

- Nested subqueries in the from clause **can't use** correlation variables.

Correlated Subqueries in the From Clause

- Nested subqueries in the from clause **can't use** correlation variables.
- **Lateral** clause permits later part of the **from** clause (after the lateral keyword) to access correlation variables from the earlier part.

Correlated Subqueries in the From Clause

- Nested subqueries in the from clause **can't use** correlation variables.
- **Lateral** clause permits later part of the **from** clause (after the lateral keyword) to access correlation variables from the earlier part.
- And yet another way to write it: **lateral** clause

```
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; some databases such as SQL Server offer alternative syntax

With Clause

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

With Clause

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

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

Complex Queries using With Clause

- Useful for writing **complex queries**
- Find all departments where the total salary is greater than the average of the total salary at all departments

instructor

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

Complex Queries using With Clause

- Useful for writing **complex queries**
- 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;
```

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

Set Membership

- SQL allows testing tuples for membership in a relation.
- The set is a collection of values (produced by a select clause)
  - **in**: test for set membership
  - **not in**: test for absence of set membership
- Name all instructors whose name is neither “Mozart” nor Einstein”

```
select distinct name
from instructor
where  name not in ('Mozart', 'Einstein')
```

Subqueries in the Where Clause

- Find courses offered in Fall 2017 and in Spring 2018

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
        course_id  in (select course_id
                        from section
                        where semester = 'Spring' and year= 2018);
```

Section

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2009	Painter	514	B
BIO-301	1	Summer	2010	Painter	514	A
CS-101	1	Fall	2009	Packard	101	H
CS-101	1	Spring	2010	Packard	101	F
CS-190	1	Spring	2009	Taylor	3128	E
CS-190	2	Spring	2009	Taylor	3128	A
CS-315	1	Spring	2010	Watson	120	D
CS-319	1	Spring	2010	Watson	100	B
CS-319	2	Spring	2010	Taylor	3128	C
CS-347	1	Fall	2009	Taylor	3128	A
EE-181	1	Spring	2009	Taylor	3128	C
FIN-201	1	Spring	2010	Packard	101	B
HIS-351	1	Spring	2010	Painter	514	C
MU-199	1	Spring	2010	Packard	101	D
PHY-101	1	Fall	2009	Watson	100	A

Subqueries in the Where Clause

- Find courses offered in Fall 2017 and in Spring 2018

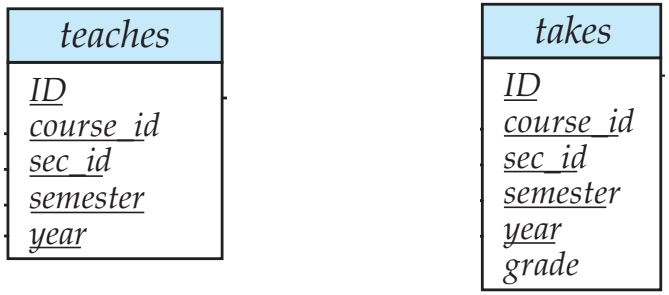
```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
        course_id  in (select course_id
                        from section
                        where semester = 'Spring' and year= 2018);
```

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

```
select distinct course_id
from section
where semester = 'Fall' and year= 2017 and
        course_id  not in (select course_id
                           from section
                           where semester = 'Spring' and year= 2018);
```

Subqueries in the Where Clause

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

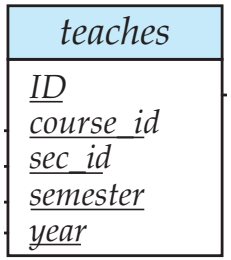


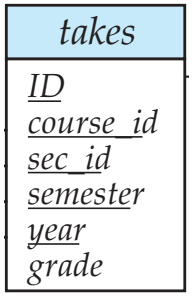
Subqueries in the Where Clause

- 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);
```

Possible to do it over multiple attributes and any relations





Subqueries in the Where Clause

- 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
  - How?

Subqueries in the Where Clause

- 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

```
select count (distinct s.ID)
from takes s join teaches t
          using (course_id, sec_id, semester, year)
where t.ID = 10101;
```

Set Comparison

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

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

Set Comparison

- 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';
```

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

Set Comparison – "some" Clause

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

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

- 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 |) =

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

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

(5 ≠ **some** | 0 | 5 |) =

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 – “all” Clause

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

Set Comparison – "all" Clause

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

```
select name
from instructor
where salary > all (select salary
                    from instructor
                    where dept name = 'Biology');
```

Definition of "all" Clause

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

(5 < **all** | 0 / 5 / 6 |) =

(5 < **all** | 6 / 10 |) =

(5 = **all** | 4 / 5 |) =

(5 ≠ **all** | 4 / 6 |) =

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 if a subquery has any tuples in its result.
- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

Use of "exists" Clause

- Find all courses taught in both the Fall 2017 semester and in the Spring 2018 semester

```
select course_id
from section as S
where semester = 'Fall' and year = 2017 and
        exists  (select *
                 from section as T
                 where semester = 'Spring' and year= 2018
                       and S.course_id = T.course_id);
```

- **Correlation name (variable)** – variable S in the outer query
- **Correlated subquery** – the inner query

Use of "not exists" Clause

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

```
select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
                    from course
                    where dept_name = 'Biology')
                    except
                    (select T.course_id
                     from takes as T
                     where S.ID = T.ID));
```

  - First nested query lists all courses offered in Biology
  - Second nested query lists all courses a particular student took
- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using = **all** and its variants

IN vs EXISTS

- Mostly same
- Difference
  - Syntax (the way writing it in SQL)
  - Query processing

IN vs EXISTS

- Mostly same
- Difference
  - Syntax (the way writing it in SQL)
  - Query processing
    - IN: subquery is evaluated (full scan) → join to the outer table
    - EXISTS: checking the existence in the subquery result while looping through the outer table
      - Outer table is small → much faster than using IN

Not IN vs Not EXISTS

- Find employees who are not manager

emp

empno	mgr	name
1	1	Bob
2		Alice
3	3	Paul

```
SELECT * FROM emp
WHERE empno NOT IN (SELECT mgr FROM emp);
```

```
SELECT * FROM emp e1
WHERE NOT EXISTS (SELECT mgr FROM emp e2
                  WHERE e1.empno = e2.mgr);
```

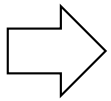
Not IN vs Not EXISTS

- In general, it is not the same!
  - Beaware of NULL (Unknown)
    - IN: comparing 2 to NULL → false (thus empty) “slide 90”

emp

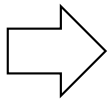
empno	mgr	name
1	1	Bob
2		Alice
3	3	Paul

```
SELECT * FROM emp
WHERE empno NOT IN (SELECT mgr FROM emp);
```



empno	mgr	name

```
SELECT * FROM emp e1
WHERE NOT EXISTS (SELECT mgr FROM emp e2
                  WHERE e1.empno = e2.mgr);
```



empno	mgr	name
2		Alice

Test for Absence of Duplicate Tuples

- The **unique** construct evaluates to **true** if a given subquery contains **no duplicates**.
  - Evaluates to “true” on an empty set

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

```
select T.course_id
 from course as T
 where unique ( select R.course_id
                  from section as R
                  where T.course_id = R.course_id
                        and R.year = 2017);
```

Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- Example queries
  - **select** *dept_name*,
    (**select count**(*)
    from *instructor*
    where *department.dept_name* = *instructor.dept_name*)
    as *num_instructors*
    from *department*;

Scalar Subquery

- Scalar subquery is one which is used where a single value is expected
- Example queries
  - **select** *dept_name*,
    (**select count**(*)
    from *instructor*
    where *department.dept_name = instructor.dept_name*)
    as *num_instructors*
    from *department*;
  - **select** *name*
    from *instructor*
    where *salary * 10 >*
    (**select** *budget* **from** *department*
    where *department.dept_name = instructor.dept_name*)
- Runtime error if subquery returns more than one result tuple

Modification of the Database

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

Deletion

- Delete all instructors

  delete from *instructor*

- Delete all instructors from the Finance department

  delete from *instructor*
  where *dept_name*= 'Finance';

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

  delete from *instructor*
  where *dept name* **in** (**select** *dept name*
  from *department*
  where *building* = 'Watson');

Deletion (Cont.)

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

```
delete from instructor
where salary < (select avg (salary)
                from instructor);
```

Deletion (Cont.)

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

```
delete from instructor
where salary < (select avg (salary)
                from instructor);
```

  - Problem: as we delete tuples from *instructor*, the average salary may change

Deletion (Cont.)

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

```
delete from instructor
where salary < (select avg (salary)
                from instructor);
```

  - Problem: as we delete tuples from *instructor*, the average salary may change
  - 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)

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*);

Insertion (Cont.)

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

Insertion (Cont.)

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

```
insert into instructor
    select ID, name, dept_name, 18000
    from   student
    where  dept_name = 'Music' and total_cred > 144;
```

Insertion (Cont.)

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

```
insert into instructor
    select ID, name, dept_name, 18000
    from   student
    where  dept_name = 'Music' and total_cred > 144;
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.
  - **insert into** *table*1 **select** * **from** *table*1

Insertion (Cont.)

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

```
insert into instructor
    select ID, name, dept_name, 18000
    from   student
    where  dept_name = 'Music' and total_cred > 144;
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation.
  - **insert into** *table*1 **select** * **from** *table*1
  - May insert an infinite number of tuples

Updates

- Give a 5% salary raise to all instructors

```
update instructor
   set salary = salary * 1.05
```

- Give a 5% salary raise to those instructors who earn less than 70,000

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

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

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

Updates (Cont.)

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

Updates (Cont.)

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  - Write two **update** statements:

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

Updates (Cont.)

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others by a 5%
  - 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 statements is important
  - Can be done better using the **case** statement

Case Statement for Conditional Updates

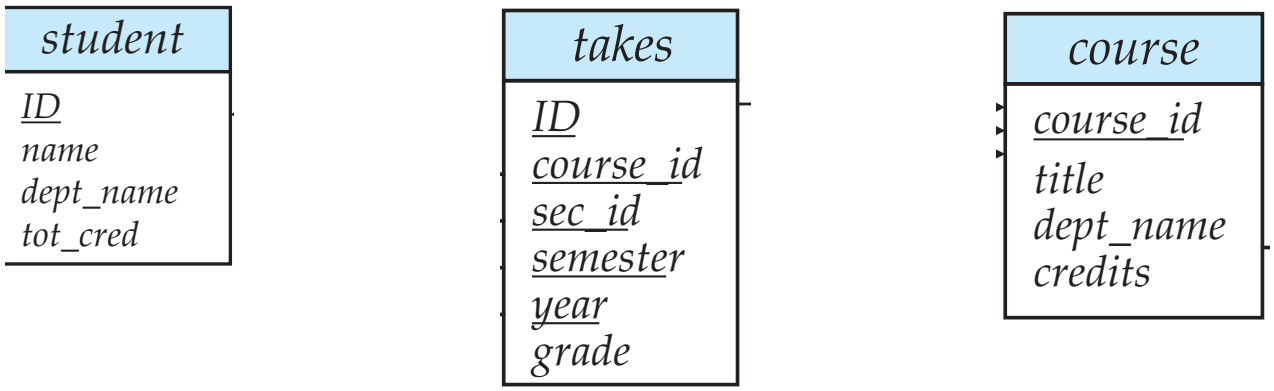
- Can be done better using the **case** statement

```
update instructor
    set salary = case
                    when salary <= 100000 then salary * 1.05
                    else salary * 1.03
                 end
```

Updates with Scalar Subqueries

- Recompute and update tot_creds value for all students

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



Updates with Scalar Subqueries

- Sets *tot_creds* to null for students who have not taken any course

Updates with Scalar Subqueries

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

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

Updates with Scalar Subqueries

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

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

  - Or **COALESCE**(sum(credits),0)
    - **COALESCE** returns first non-null arguments

Recap

- SQL queries
  - Clauses: **SELECT**, **FROM** , **WHERE**, **GROUP BY**, **HAVING**, **ORDER BY**
  - Nested subqueries
  - Equivalence with relational algebra
- SQL update, inserts, deletes
  - Semantics of referencing updated relation in **WHERE**
- SQL DDL
  - Table definition: **CREATE TABLE**

Corresponding Reading Materials

- Database System Concepts 6th & 7th Edition
  - Chapter 3
- Database Systems: The Complete Book 2nd Edition
  - Chapter 6