SQL – Lecture 1

- Introduction
- Data Definition Language
- Data Manipulation Language (1)
  - Basic Structure
  - Set Operations
  - Aggregate Functions
  - Null Values
  - Join expressions

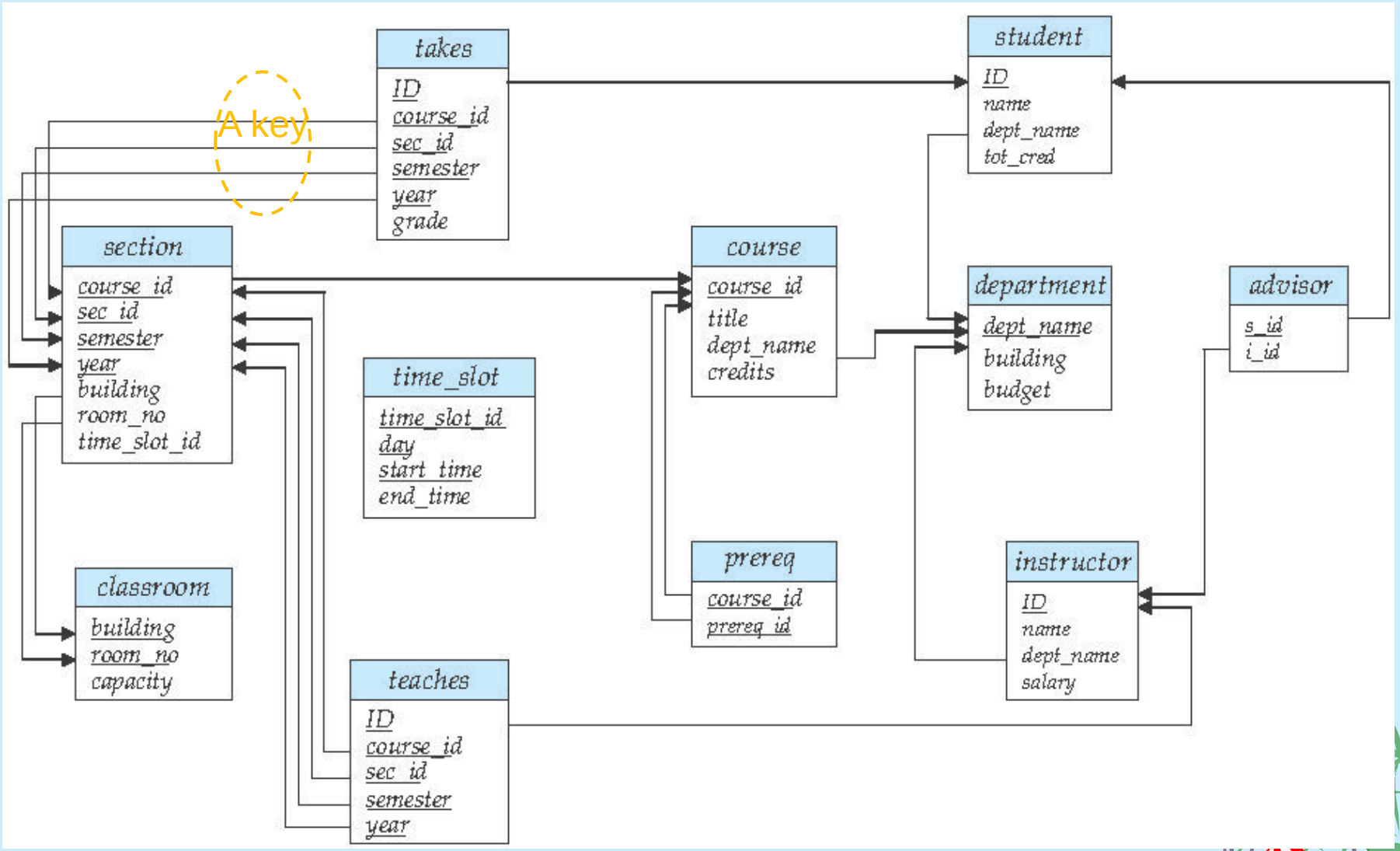
Introduction

- SQL: Structured Query Language
  - Most widely used database language today.
  - Although it is named a “Query Language”, It can do much more than just query in database, including data definition, data manipulate, specify security constraints, etc.
- History of SQL
  - 1975～1979， IBM System R, Original named Sequel
  - 1986， SQL-86
  - 1989， SQL-89
  - 1992， SQL-92 (Most widely implemented)
  - 1999， SQL-99（SQL3）
  - 2003， SQL:2003
  - SQL:2006, SQL:2008
- Commercial systems offer most, if not all, SQL-92 features, plus varying features from later standards and special proprietary features.
  - Not all examples here may work on your particular system.

SQL Parts

- The SQL language has several parts
  - Data-definition language (DDL)
    - View definition
    - Integrity
  - Interactive data-manipulation languages (DML)
  - Database Control Language (DCL)
    - Transaction control
    - Authorization
  - Embedded SQL and dynamic SQL

Schema Diagram for University Database



Data Definition Language (DDL)

DLL provides the definition of relations, it allows the specification of not only a set of relations but also the following information:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints

Basic structure…

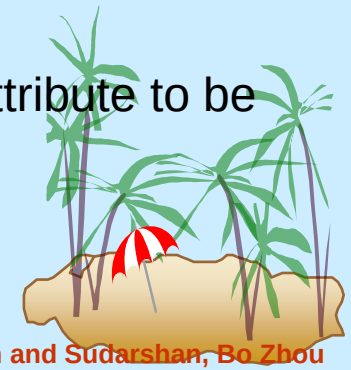
- Security and authorization information for each relation.
- The set of indices to be maintained for each relations.
- The physical storage structure of each relation on disk.

A little of physical structure…

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. . (ex., numeric(3,1), allows 44.5 to be stores exactly, but not 444.5 or 0.32)
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least *n* digits.

- Null values are allowed in all the domain types. Declaring an attribute to be **not null** prohibits null values for that attribute.



Date/Time Types in SQL (Cont.)

- **date.** Dates, containing a (4 digit) year, month and date
  - E.g. **date** '2007-9-23'
- **time.** Time of day, in hours, minutes and seconds.
  - E.g. **time** '09:00:30' **time** '09:00:30.75'
- **timestamp**: date plus time of day
  - E.g. **timestamp** '2007-9-23 09:00:30.75'
- **Interval**: period of time
  - E.g. Interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values
- Can extract values of individual fields from date/time/timestamp
  - E.g. **extract** (**year from** r.starttime)
- Can cast string types to date/time/timestamp
  - E.g. **cast** <string-valued-expression> **as date**
  - E.g. **cast** <string-valued-expression> **as time**

Create Table Construct

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

  create table *r* (

  $A_1\ D_1, A_2\ D_2, \ldots, A_n\ D_n,$
  (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),
        dept_name   varchar(20),
        salary      numeric(8,2))
```

Integrity Constraints in Create Table

- Types of integrity constraints
  - **primary key** (A_1, ..., A_n) : Any Primary key is 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);
```

Few More Relation Examples

- **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*);

- **create table** *course* (
  course_id **varchar**(8),
  title **varchar(**50),
  dept_name **varchar**(20),
  credits **numeric**(2,0),
  primary key *(course_id),*
  foreign key *(dept_name*) **references** *department*);

Drop and Alter Table Constructs

- The **drop table** command deletes all information about the dropped relation from the database.
- The **alter table** command is used to add attributes to an existing relation. All tuples in the relation are assigned *null* as the value for the new attribute. The form of the **alter table** command is

  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.*
- The **alter table** command can also be used to drop attributes of a relation

  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 Structure of SQL Queries

- A typical SQL query has the form:

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

  - A_is represent attributes: that desired in the result of the query
  - r_is represent relations : to be scanned in the evaluation of the query
  - P is a predicate : selection criteria to be satisfied.

- This query is equivalent to the relational algebra expression.

$$\Pi_{A1, A2, ..., An}(\sigma_P (r_1 \times r_2 \times ... \times r_m))$$

- The result of an SQL query is a relation (can be duplicated!).

The select Clause

- The **select** clause list the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
  - E.g. Find the names of all instructors

    select *name*
    from *instructor*

    $\Pi_{name}(instructor)$
- An asterisk in the select clause denotes “all attributes”

  select *
  from *instructor*
- NOTE: SQL names are case insensitive, meaning you can use upper case or lower case.
  - E.g., *Name* ≡ *NAME* ≡ *name*
  - You may wish to use upper case in places where 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 keyword **all** specifies that duplicates not be removed.

```
select all dept_name
from instructor
```

The keyword 'all' is useless, because it is default

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

The select Clause (Cont.)

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

  select '437'

  - Results 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 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*
    from *instructor*

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*

    select *name*
    from *instructor*
    where *dept_name* = 'Comp. Sci.'

- Comparison results can be combined using the logical connectives **and, or,** and **not**
  - *To find all instructors in Comp. Sci. dept with salary > 80000*

    select *name*
    from *instructor*
    where *dept_name* = 'Comp. Sci.' **and** *salary* > 80000

- Comparisons can be applied to results of arithmetic expressions.

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');
```

Equivalent to:

```
select name, course_id
from instructor, teaches
where  instructor.ID = teaches.ID and dept_name ='Biology';
```

The from Clause

- 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.
  - 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 (selection operation in relational algebra).

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*
- 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'

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

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 in the Art department who have taught some course and the course title
  - **select** *instructor.ame* as *teacher-name*, *course.title* as *course-title*
    from *instructor , teaches, course*
    where *instructor.ID = teaches.ID* ***and***
    teaches.course_id = course.course_id ***and***
    instructor. dept_name = 'Art'

Tuple Variables

- Tuple variables are defined in the **from** clause via the use of the **as** clause.

- Find the names of all instructors in the Art department who have taught some course and the course_id
  - **select** *T.name, C.course_id*
    from *instructor as T , teaches as C*
    where *T.ID = C.ID* ***and*** *T. dept_name* = 'Art'

- 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*

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.)

- 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)
  - finding string length, extracting substrings, etc.

Ordering the Display of Tuples

- SQL query does not assure any order of query result in default.
- 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* *desc* , *name* *asc*

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- *Multiset* versions of some of the relational algebra operators – given multiset relations r_1 and r_2:

1. **$\sigma_\theta(r)$:** If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selections σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
2. **$\Pi_A(t_1)$:** For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.
3. **r_1 x r_2:** If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are c_1 x c_2 copies of the tuple t_1. t_2 in r_1 x r_2

Duplicates (Cont.)

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

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

- 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:

$$\Pi_{A1,, A2, ..., An}(\sigma_P (r_1 \text{ x } r_2 \text{ x } ... \text{ x } r_m))$$

Set Operations

- The set operations **union, intersect,** and **except** operate on relations and correspond to the relational algebra operations $\cup$, $\cap$, $-$.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all, intersect all** and **except all.**

  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*

Set Operations

- Find courses that ran in Fall 2019 or in Spring 2020

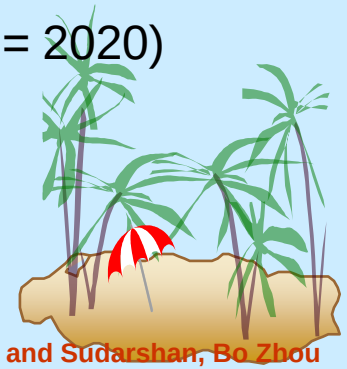
  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2019)
  union
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2020)

- Find courses that ran in Fall 2019 and in Spring 2020

  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2019)
  intersect
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2020)

- Find courses that ran in Fall 2019 but not in Spring 2020

  (**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2019)
  except
  (**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2020)



An example

Find the instructor who has the highest salary

- The query is:

$$\Pi_{id,\ name}(instructor) - \Pi_{instructor.id,\ instructor.name}(\sigma_{instructor.salary < d.salary}(instructor \times \rho_{d}(instructor)))$$

- SQL Query

```
select  id, name
from instructor

   except

select  T.id, T.name
from instructor as T, instructor as D
where T.salary < D.salary
```

Aggregate Functions

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

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

Aggregate Functions (Cont.)

- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*)
    from *instructor*
    where *dept_name*= ’Comp. Sci.’;
- 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;
- Find the number of tuples in the *course* relation
  - **select count** (*)
    from *course*;

Aggregate Functions – Group By

- Find the average salary of instructors in each department
  - **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 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, teaches
where instructor.ID = teaches.ID and
        semester = 'Spring' and year = 2010
group by dept_name
```

Note: Attributes in **select** clause outside of aggregate functions must appear in **group by** list

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*)
  from *instructor*
  group by *dept_name*
  having avg (*salary*) > 42000;

- Having clause vs. Where clause
  - predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups
    - Apply where predicates -> form groups -> apply having predicates
  - predicates in the **having** clause are applied to each group whereas predicates in the **where** clause are applied to each tuple.

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

Null Values and Three Valued Logic

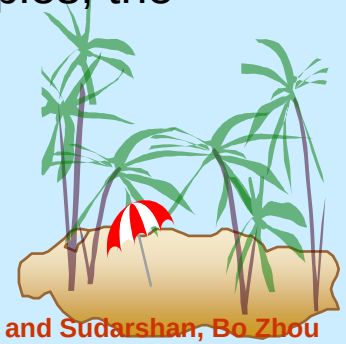
- Three values – *true*, *false*, *unknown*
- Any comparison with *null* returns *unknown* (other than predicates **is null** and **is not null**)
  - *E.g. 5 < null or null <> null or null = null*
- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*, (*unknown* **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*

Special treatment of Null Values

- Null Values and Aggregates

  select sum (*salary*)
  from *instructor*

  - Above statement ignores null amounts, and return the total salary
  - 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 select distinct
  - When a query use select distinct clause, to remove duplicated tuples. When comparing the corresponding attributes from two tuples, the values are treated as identical if one of the following:
    - the both values are not null and equal in value
    - or both are null.



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
- Three types of joins:
  - Natural join
  - Inner join
  - Outer join

Join Expression

- Join expressions are introduced to make some SQL query easier to understand.

  Select *name, course_id*
  from *instructor, teaches*
  where *instructor.ID = teaches.ID*

  Could be simplified as:

  Select *name, course_id*
  from *instructor* **nature join** *teaches*

  Or :

  Select *name, course_id*
  from *instructor* **join** *teaches* **using** *(ID)*

Join Conditions

- The ON condition allows a general predicate over the relations being joined.

Select *
from *student* **join** *takes* **on** *student.ID = takes.ID*

Is equivalent to

Select *

from *student, takes*

where *student.ID = takes.ID*

Student Relation

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Takes Relation

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	*null*

student natural join *takes*

ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2017	A
19991	Brandt	History	80	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	CS-101	1	Fall	2017	F
45678	Levy	Physics	46	CS-101	1	Spring	2018	B+
45678	Levy	Physics	46	CS-319	1	Spring	2018	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2017	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2018	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2017	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2018	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2017	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2018	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2017	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2018	*null*

Dangerous in Natural Join

- Beware of unrelated attributes with same name which get equated incorrectly
- Example -- List the names of students instructors along with the titles of courses that they have taken
  - Correct version

```
select name, title
from student natural join takes, course
where takes.course_id = course.course_id;
```

  - Incorrect version

```
select name, title
from student natural join takes natural join course;
```

    - This query omits all (student name, course title) pairs where the student takes a course in a department other than the student's own department. *(dept_name, course_id)*
    - The correct version (above), correctly outputs such pairs.

Outer Join

- An extension of the join operation that avoids loss of information.
  - E.g. To list all students along with the course they have taken.
  - E.g. To list all instructors along with the course they teach.
- 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.
  - Left Outer Join
  - Right Outer Join
  - Full Outer Join
- Uses null values.

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 and course information is missing for CS-437

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*

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

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

Joined Relations

- Join operations take two relations and return as a result another relation.
- The join operations are typically used as subquery expressions in the **from** clause
- Join condition – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- 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 Types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using (A_1, A_2, ..., A_n)

The end of the lecture