Introduction to SQL

Introduction to Oracle

- Log onto grace system
- Go into public directory
  cd public/Mondial_dataset
- Start oracle
  tap oraclient
- Your SID is 'dbclass1'
- Start sqlplus

  sqlplus
- Enter user name and password
- To change your password
  alter user <username> identified by <pass>;

Load tables

- Copy Mondial_dataset from public directory to your own
  ```
  cp -r ../../public/Mondial_dataset .
  cd Mondial_dataset
  ```
- Start sqlplus
  ```
  sqlplus
  ```
- Create tables
  ```
  @ create
  ```
- Load data
  ```
  @ data
  ```

- If you need to trash everything
  ```
  @ drop
  ```

Basic Query Structure

- A typical SQL query has the form:

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

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.
- The result of an SQL query is a relation.

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
- Example: find the names of all countries:

  select N*ame*

  from *Country*
- 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 names of all cities that have the headquarters of an organization

  select distinct city
  from *organization*

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

  select all city
  from *organization*

The select Clause (Cont.)

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

```
select *
from organization
```

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

```
select code, name, area/100
from country
```

would return a relation that is the same as the country relation, except that the value of the attribute area is divided by 100.

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 cities in USA with population > 80000

```
select name
from city
where country = ‘USA' and population > 80000
```

- Comparison results can be combined using the logical connectives **and, or,** and **not.**
- Comparisons can be applied to results of arithmetic expressions.

The where clause...

- Find all provinces (states) in the USA that have more than 20 people per square mile
-

```
select name
from province
where country = 'USA'
    and population / area > 20
```

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 *country X province*

  select ∗
  from *country,province*

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

Joins

- For the names of all countries in the UN

```
select country.name, population
   from country, organization
   where   organization.country = code
        and organization.name = 'United Nations'
```

- Note: you need to clarify ambiguous names

Rename variables/relations

```
select c.name, population
 from country [as] c, organization [as] o
 where   o.country = code
      and o.name = 'United Nations'
```

Natural join

- Matches attributes with same name

  ```
  select *
  from country natural join province
  ```

- Caveat: country.name and province.name don't mean the same thing – result is incorrect/unexpected
- But

  ```
  select *
  from economy natural join population
  ```

  works! (economy.country and population.country refer to the same thing)

Natural join cont..

- How do you get the name of the country as well?

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  '\'
```

- SQL supports a variety of string operations such as
  - concatenation (using “||”)
  - converting from upper to lower case (and vice versa)
  - finding string length, extracting substrings, etc.

Ordering the Display of Tuples

- List in alphabetic order the names of all instructors

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

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

Where Clause Predicates

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

Set Operations

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

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

Find courses that ran in Fall 2009 and in Spring 2010

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

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

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

Set Operations

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

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- The predicate **is null** can be used to check for null values.
  - Example: Find all instructors whose salary is null*.*

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

Null Values and Three Valued Logic

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

Aggregate Functions

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

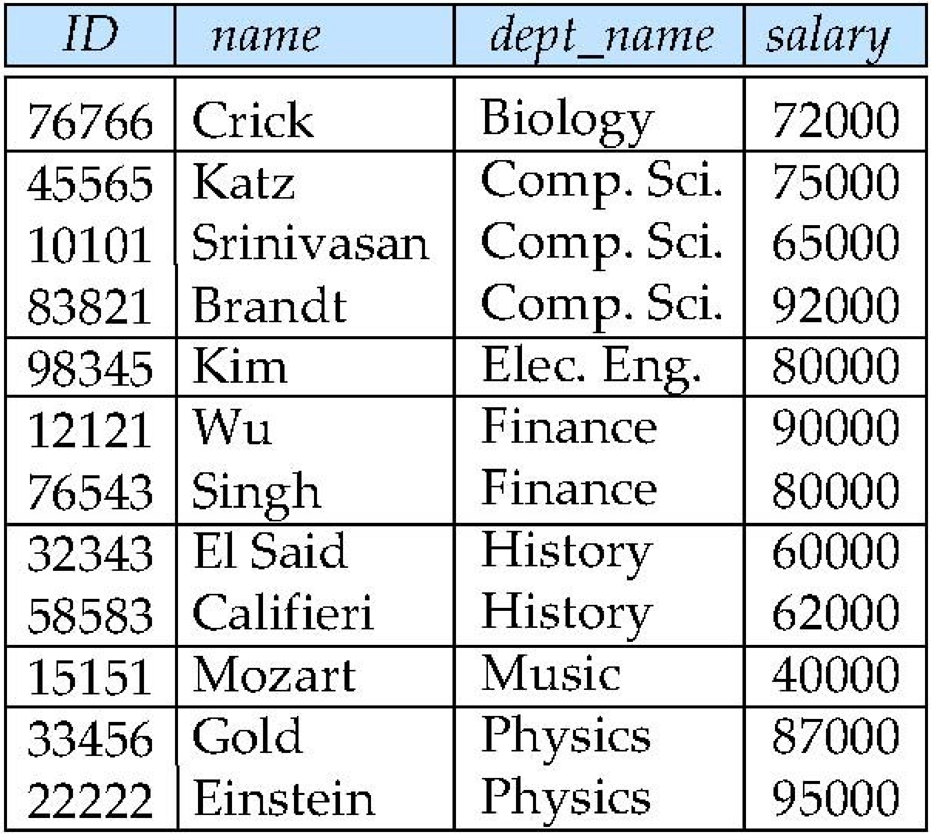
avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

Aggregate Functions (Cont.)

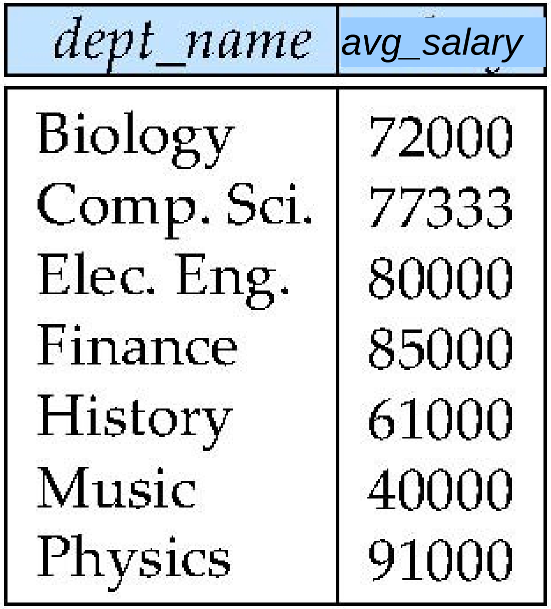
- Find the average salary of instructors in the Computer Science department
  - **select avg** (*salary*)
from *instructor*
where *dept_name*= 'Comp. Sci.';
- Find the total number of instructors who teach a course in the Spring 2010 semester
  - **select count** (**distinct** *ID*)
from *teaches*
where *semester* = 'Spring' **and** *year* = 2010
- 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*)
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

Aggregation (Cont.)

- Attributes in **select** clause outside of aggregate functions must appear in **group by** list
  - /* erroneous query */

    ```
    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)
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

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

Nested Subqueries

- SQL provides a mechanism for the nesting of subqueries.
- A **subquery** is a **select-from-where** expression that is nested within another query.
- A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

Example Query

- Find courses offered in Fall 2009 and in Spring 2010

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

Find courses offered in Fall 2009 but not in Spring 2010

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

Example Query

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

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

Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

Set Comparison

- 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: $<, \; \leq, \; >, \; =, \; \neq$

$(5 < \textbf{some} \; \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline\end{array} \;) = \text{true}$

(read: 5 < some tuple in the relation)

$(5 < \textbf{some} \; \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline\end{array} \;) = \text{false}$

$(5 = \textbf{some} \; \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline\end{array} \;) = \text{true}$

$(5 \neq \textbf{some} \; \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline\end{array} \;) = \text{true (since } 0 \neq 5)$

(= **some**) ≡ **in**

However, ($\neq$ **some**) $\not\equiv$ **not in**

Example Query

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

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

Definition of all Clause

- F <comp> **all** *r* ⇔ ∀ *t* ∈ *r* (F <comp> *t)*

$(5 < \textbf{all} \begin{array}{|c|}\hline 0\\ \hline 5\\ \hline 6\\ \hline\end{array}) = \text{false}$

$(5 < \textbf{all} \begin{array}{|c|}\hline 6\\ \hline 10\\ \hline\end{array}) = \text{true}$

$(5 = \textbf{all} \begin{array}{|c|}\hline 4\\ \hline 5\\ \hline\end{array}) = \text{false}$

$(5 \neq \textbf{all} \begin{array}{|c|}\hline 4\\ \hline 6\\ \hline\end{array}) = \text{true (since } 5 \neq 4 \text{ and } 5 \neq 6)$

(≠ **all**) ≡ **not in**

However, (= **all**) ≡ **i̸n**

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.
- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

Correlation Variables

- Yet another way of specifying the query “Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester”

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

- **Correlated subquery**
- **Correlation name** or **correlation variable**

Not Exists

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

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

Note that $X - Y = \emptyset \quad \Leftrightarrow \quad X \subseteq Y$

Note: Cannot write this query using = **all** and its variants

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- Find all courses that were offered at most once in 2009

```
 select T.course_id
from course as T
where unique (select R.course_id
              from section as R
              where T.course_id= R.course_id
                    and R.year = 2009);
```

Derived Relations

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors’ salaries of those departments where the average salary is greater than $42,000.”

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

- Note that we do not need to use the **having** clause
- Another way to write above query

```
 select dept_name, avg_salary
from (select dept_name, avg (salary)
        from instructor
        group by dept_name) as dept_avg (dept_name,
avg_salary)
```

Derived Relations (Cont.)

- 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);
```

With Clause

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

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