Lecture 3

SQL

Outline

- Create/Drop/Alter Tables
- Basic Query Structure
- Set Operations
- Aggregation Functions
- Recursive Queries
- Complex Queries
- Views

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL
  - SQL-92
  - SQL:1999 (language name became Y2K compliant!)
  - SQL:2003
- 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.

Data Definition Language

- **Allows the specification of not only a set of relations but also information about each relation, including:**
  - The schema for each relation.
  - The domain of values associated with each attribute.
  - Integrity constraints
  - The set of indices to be maintained for each relations.
  - Security and authorization information for each relation.
  - The physical storage structure of each relation on disk.

Create Table Construct

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

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

  - *r* is the name of the relation
  - each A_i is an attribute name in the schema of relation *r*
  - D_i is the data type of values in the domain of attribute A_i
- Example:

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

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.*
- **numeric(p,d).** Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point.
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **real.** 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.
- ……

Integrity Constraints in Create Table

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

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);*

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

And a Few More Relation Definitions

- **create table** *student* (
  ID **varchar**(5),
  name **varchar**(20) not null,
  dept_name **varchar**(20),
  tot_cred **numeric**(3,0),
  primary key *(ID),*
  foreign key *(dept_name*) **references** *department*);

- **create table** *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*);

And still more…

- **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

Drop and Alter Table Constructs

- **drop table: deletes all information about the dropped relation from the database, incl. all tuples, constraints, relation itself**

  drop table r

- **alter table:**
  - **Add attributes to an existing relation:**

    alter table *r* add *A D*

    - **All tuples in the relation are assigned *null* as the value for the new attribute.**
  - **Drop attributes of a relation:**

    alter table *r* drop *A*

    - **Dropping of attributes not supported by some databases**
    - **Syntax may differ among different implementations**
  - **Modify constraints**

    alter table *r* add primary key (*A, B*)

    alter table *r* drop primary key

Data Manipulation Language

- **Supports querying data from a database**
  - **Getting raw data from tables and views**
  - **Statistics (aggregation)**
  - **Adding, deleting, and modifying data**

  - **Query Language does more than just “query”**

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.
- Equivalent relational algebra expression:

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

- 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 instructors:

  select *name*

  from *instructor*;
- Relational algebra expression:

  Π_{name} (*instructor*)

NOTE: SQL names are case insensitive
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**.**
- Example: 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 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’

  – 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.

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

- 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 the names of 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 names of 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.

The where Clause (Cont.)

- SQL includes a between comparison operator
- Example: Find the names of those instructors with salary between \$70,000 and \$100,000 (that is, ≥ \$70,000 and ≤ \$100,000)

  select *name*
  from *instructor*
  where *salary* **between** 70000 **and** 100000

- Tuple comparison

select *name*, *course_id*
from *instructor*, *teaches*
where (*instructor*.*ID*, *dept_name*) = (*teaches*.*ID*, '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*)

Note: Result of Cartesian product can be large.

- Typically combined with where-clause condition

Cartesian Product

instructor

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

teaches

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

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

- 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 who have a higher salary than at least one instructor in the Computer Science department.
  - **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

- 字符串匹配运算符：like
  - 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 name “100%”

    like ‘100\%’ **escape** ‘\’
- SQL supports a variety of string operations (functions)
  - 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*

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

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 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 (Cont.)

- Find all the salaries of all instructors
  - **select distinct** *salary*
    from *instructor*

- Find the salaries of all instructors that are less than the largest salary.
  - **select distinct** *T.salary*
    from *instructor* **as** *T*, *instructor* **as** *S*
    where *T.salary* < *S.salary*

- Find the largest salary of all instructors.
  - <QUERY1>
    except
    <QUERY2>

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 *courses*
  - **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

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

- 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 loan number which appear in the *loan* relation with null values for *amount.*

    select *loan_number*
    from *loan*
    where *amount* **is null**
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- However, aggregate functions simply ignore nulls
  - More on next slide

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*

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.
  – For example, when calculating avg(score), those who do not have a score (absent from the exam) are not taken into account.
  – But count(score) will return total number of students, including those absent from the exam.

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:

- A_i can be replaced be a subquery that generates a single value.
- r_i can be replaced by any valid subquery
- P can be replaced with an expression of the form:

  B <operation> (subquery)

  Where B is an attribute and <operation> to be defined later.

Subqueries in the Where Clause

- A common use of subqueries is to perform tests:
  - For set membership
  - For set comparisons
  - For set cardinality.

Set Membership

- Find courses offered in Fall 2009 and in Spring 2010

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

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

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

Set Membership (Cont.)

- 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 – “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 |) = 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.

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 |) = 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

- 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

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

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

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

Use of “not exists” Clause

- Find the IDs and names of 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 $X - Y = \emptyset \Leftrightarrow X \subseteq Y$

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- The **unique** construct evaluates to “true” if a given subquery contains no duplicates .
- Find all courses that were offered at most once in 2008

```
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 = 2008);
```

Subqueries in the “from” Clause

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

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

- Note that we do not need to use the **having** clause, since we compute the temporary (view) relation in the **from** clause, and the attributes of *avg_salary* can be used directly in the **where** clause.

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 department.name
from department, max_budget
where department.budget = max_budget.value;
```

Complex Query using With Clause

- 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;
```

Subquery in the “select” Clause

- Scalar subquery is one which is used where a single value is expected
- List all departments along with the number of instructors in each department

```
select dept_name,
        (select count(*)
           from instructor
          where department.dept_name = instructor.dept_name)
        as num_instructors
from department;
```

- 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

Modification of the Database – 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*);

- Problem: as we delete tuples from *instructor*, the average salary changes
- Solution used in SQL:
  1. First, compute **avg**(salary) and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Modification of the Database – Insertion

- Add all instructors to the *student* relation with tot_creds set to 0

  insert into *student*
  select *ID, name, dept_name, 0*
  from *instructor*

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

  Otherwise queries like

  insert into *table*1 **select** * **from** *table*1

  would cause problem

Modification of the Database – Updates

- 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 is important! (Why?)
  - Can be done better using the **case** statement (next slide)

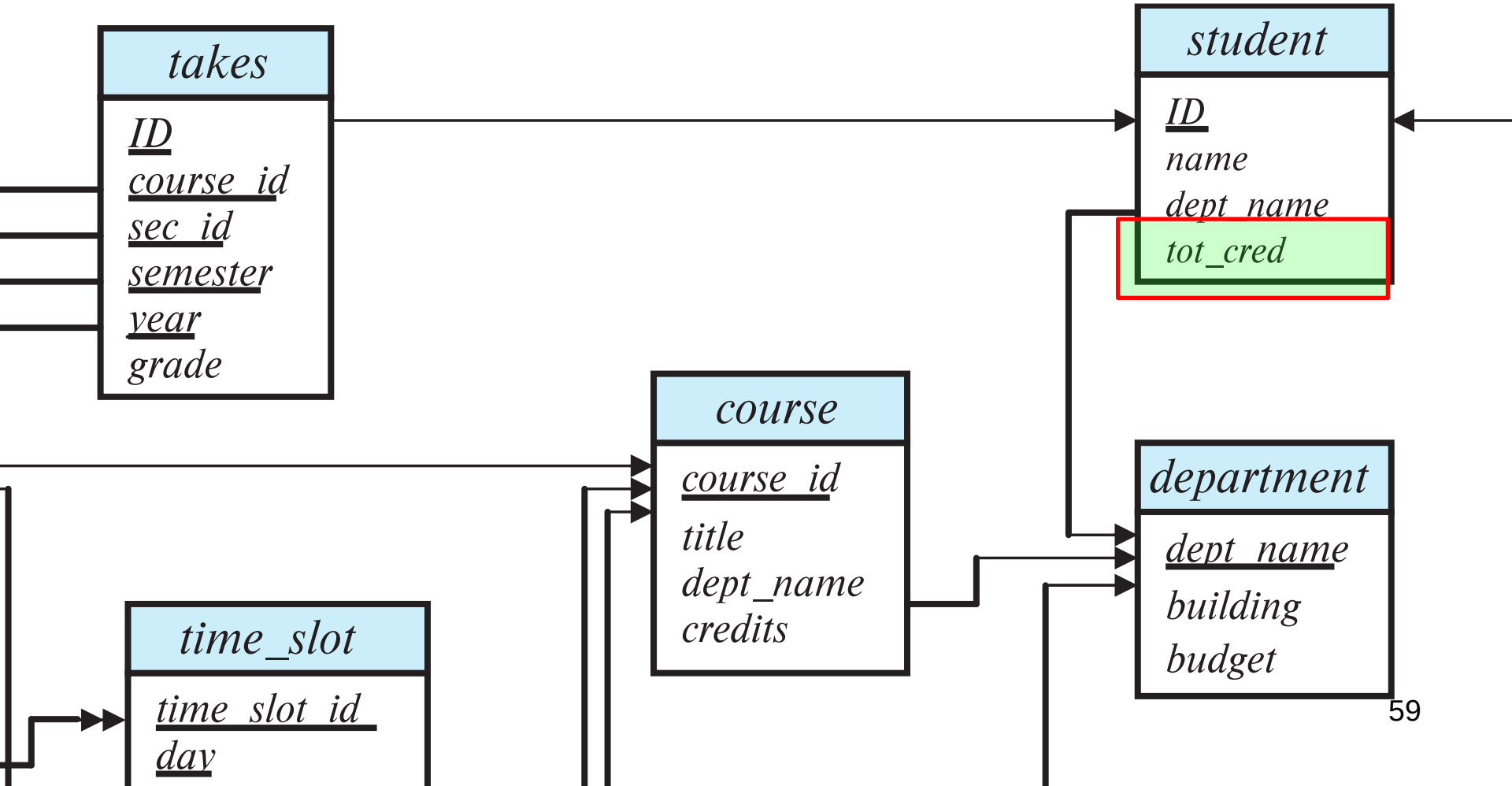
Case Statement for Conditional Updates

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

```
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



Updates with Scalar Subqueries

- Recompute and update tot_creds value for all students

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

- Sets *tot_cred* to 0 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
```

Next Lecture

- SQL to be continued

End of
Lecture 3

SQL

Practice

- Environment: MySql/Oracle
- Try to
  - Create tables in the University example.
    - With scripts provided
  - Drop/Add primary key
  - Drop/Add an attribute
  - View table structure (“desc”)
  - Change an attribute (data type)

Practice (cont.)

- Try to
  - Insert tuples
  - Delete tuples with conditions
  - Update tuples with conditions
  - Commit/Rollback
    - Login a second time to find the effect of Commit/Rollback
  - Execute example queries
- Question
  - Is it hard to insert rows one by one?
    - Lab1