Chapter 3

SQL

SQL

- Background
- Basic Structure__query
- DDL
- DML(Insert ,Delete,Update)
- Embedded SQL
- Dynamic SQL
- ODBC,JDBC
- Transact_SQL

Background

Structured query language (SQL)

1970s , IBM Sequel(System R)

1986, ANSI and ISO , SQL_86 ;

1987, IBM, SAA_SQL;

1989, ANSI, SQL_89, an extended standard for SQL;

SQL_92;

1999 SQL;

SQL:2003

Background

Data definition language (DDL);

Data manipulation language (DML);

Data control language (DCL);

Integrity;

Authorization;

Embedded SQL and dynamic SQL;

SQL is based on set and relational operations with certain modifications and enhancements

Basic Structure__query

- 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 represent attributes

  r_i represent relations

  P is a predicate.

- This query is equivalent to the relational algebra expression.

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

The result of an SQL query is a relation

The select Clause

- The **select** clause corresponds to the projection operation of the relational algebra. It is used to list the attributes desired in the result of a query.

- Find the names of all instructors:

  select name
  from instructor

- In the "pure" relational algebra syntax, the query would be:

$$\Pi_{name}(instructor)$$

- An asterisk in the select clause denotes "all attributes"

  select *
  from *instructor*

The select Clause

- 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 "salary/12" using the as clause:

select ID, name, salary/12 as monthly_salary

The select Clause

•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 should not be removed.**

  select all dept_name
  from instructor

The where Clause

- The **where** clause corresponds to the selection predicate of the relational algebra.

- **Find all instructors in Comp. Sci. dept**

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

The where Clause

- **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.

Between..and

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between $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');

The from Clause

- The **from** clause corresponds to the Cartesian product operation of the relational algebra. It lists the relations to be scanned in the evaluation of the expression.

- Find the Cartesian product instructor X teaches

```
select *
from instructor, teaches
```

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).

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

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'
```

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

Alias

String Operations

- 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

- Match the name "Main%"

  like 'Main\%' **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*

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. $\boldsymbol{\sigma_\theta(r_1)}$**:** 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. $\boldsymbol{\Pi_A(r\ )}$**:** 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. $\boldsymbol{r_1 \ \mathrm{x}\ r_2}$ **:** 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

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

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

- 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:

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.**

Set Operations_Example

- 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)

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_Example

- 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*) **as** *avg_salary*
    from *instructor*
    group by *dept_name*;

Note: 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 – Group By

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

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 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

- Total all salaries

  select sum (*salary*)
  from *instructor*

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

Null Values

- 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*

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

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* ⇔ ∃ *t* ∈ *r* such that (F <comp> *t*)
  Where <comp> can be: <, ≤, >, =, ≠

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

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

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

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

(= **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

- $F \ \text{<comp>} \ \textbf{all} \ r \Leftrightarrow \forall \ t \in r \ \ (F \ \text{<comp>} \ t)$

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

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

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

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

($\neq$ **all**) $\equiv$ **not in**

However, (= **all**) $\not\equiv$ **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

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

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

- ***Note that $X - Y = \emptyset \;\Leftrightarrow\; X \subseteq Y$***
- ***Note: Cannot write this query using = all and its variants***

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.
- The **unique** construct evaluates to “true” if a given subquery contains no duplicates .
- 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);
```

Subqueries in the From Clause

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;

Subqueries in the From Clause

- 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)
    where avg_salary > 42000;
```

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

- 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;
```

Subqueries in the Select Clause

Scalar Subquery

- 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

Select customer_name

From borrower

Where **exists** (select *

from depositor

where depositor.customer_name=borrower.customer_name)

Set operation:

in, not in, >some/all, <=some/all, =some/all, <>some/all,

exists, not exists,

unique,not unique

The select Clause

SELECT [DISTINCT] <Attributes list>

FROM <relations list>

[*WHERE* < predicate >]

[*GROUP BY* < Attributes list >

[*HAVING* < Attributes list >]]

[*ORDER BY* < Attribute > [ASC|DESC]， …， < Attribute > [ASC|DESC]]；

Data Definition Language (DDL)

Allows the specification of not only a set of relations but also information about each relation, including:

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

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))

Integrity Constraints

- **not null**
- **primary key** (A_1, ..., A_n)
- *Foreign Key (A1,…,An) reference table r on delete restrict/cascade/set NULL*
- **check** *(P),* where *P* is a predicate

```
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);
```

And a Few More Relation Definitions

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

- **create table** *takes* (
  ID **varchar**(5),
  course_id **varchar**(8),
  sec_id **varchar**(8),
  semester **varchar**(6),
  year **numeric**(4,0),
  grade **varchar**(2),
  primary key *(ID, course_id, sec_id, semester, year)* ,
  foreign key (*ID*) **references** *student,*
  foreign key (*course_id, sec_id, semester, year*) **references** *section*);
  - Note: *sec_id* can be dropped from primary key above, to ensure a student cannot be registered for two sections of the same cour[illegible]e same semester

- **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*);

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 *n* digits to the right of decimal point.
- **real, double precision.** Floating point and double_precision floating point numbers, with machine_dependent precision.
- **float(n).** Floating point number, with user_specified precision of at least *n* digits.

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.

  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*.

Views

- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a *view*.

- A view provides a mechanism to hide certain data from the view of certain users.

- the command:

  create view *v* **as** <query expression>

  where:

  - <query expression> is any legal expression
  - The view name is represented by *v*

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.

- When a view is created, the query expression is stored in the database; the expression is substituted into queries using the view.

Views _Example

- A view consisting of branches and their customers

```
create view all_customer as
  (select branch_name, customer_name
   from depositor, account
   where depositor.account_number =
account.account_number)
    union
 (select branch_name, customer_name
  from borrower, loan
  where borrower.loan_number = loan.loan_number)
```

Find all customers of the Perryridge branch

```
select customer_name
from all_customer
where branch_name = 'Perryridge'
```

View Expansion

- One view may be used in the expression defining another view.A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

repeat
Find any view relation v_i in e_1
Replace the view relation v_i by the expression defining v_i
until no more view relations are present in e_1

- As long as the view definitions are not recursive, this loop will terminate

Modification of the Database -Deletion

- The command :

  delete from

- 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');

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

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

- Problem: as we delete tuples from deposit, the average salary changes
- Solution used in SQL:

  1. First, compute avg (salary) and find all tuples to delete
  2. Next, delete all tuples found above (without recomputing avg or retesting the tuples)

Modification of the Database -Insertion

- The command :

  Insert into Values()

- Add a new tuple to *course*

insert into *course*
  values *('CS-437', 'Database Systems', 'Comp. Sci.', 4);*

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);

- 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** table1 **select** * **from** table1*

    would cause problem

Modification of the Database -Update

- The command :

Update Set

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

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

- The order is important
- Can be done better using the case statement

Case Statement for Conditional Updates

- Same query as before but with case statement

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

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_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
```

Index

CREATE INDEX H_INDEX ON STUDENT(HEIGHT)

CREATE UNIQUE INDEX SC_INDEX ON

SC(SNO ASC,CNO DESC)

DROP INDEX H_INDEX

SUMMARY

Query

SELECT..

Modification

Delete, Insert , update

Definition

Table,View