Structured Query Language

1 Introduction to SQL

What is SQL?

– When a user wants to get some information from a database file, he can issue a ***query***.

– A query is a user–request to retrieve data or information with a certain condition.

– SQL is a query language that allows user to specify the conditions. (instead of algorithms)

1 Introduction to SQL

Concept of SQL

- The user specifies a certain condition.
- The program will go through all the records in the database file and select those records that satisfy the condition.(searching).
- Statistical information of the data.
- The result of the query will then be stored in form of a table.

Data Definition Language

The SQL data-definition language (DDL) allows the specification of information about relations, including:

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

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.

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))

Integrity Constraints in Create Table

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

Example: Declare *branch_name* as the primary key for *branch*

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**

Not Null and Unique Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**

    name **varchar**(20) **not null**
    budget **numeric**(12,2) **not null**
- **unique** (A_1, A_2, …, A_m)
  - The unique specification states that the attributes *A*1, *A*2, … *A*m
    form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If “Biology” is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for “Biology”.
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.

The check clause

- **check** (P)

  where P is a predicate

  Example: ensure that semester is one of fall, winter, spring or summer:

```
create table section (
    course_id varchar (8),
    sec_id varchar (8),
    semester varchar (6),
    year numeric (4,0),
    building varchar (15),
    room_number varchar (7),
    time slot id varchar (4),
    primary key (course_id, sec_id, semester, year),
    check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))
);
```

CREATE TABLE

- In SQL, we can use the CREATE TABLE command for specifying the primary key attributes, secondary keys, and referential integrity constraints (foreign keys).
- Key attributes can be specified via the PRIMARY KEY and UNIQUE phrases

```
CREATE TABLE DEPT (
    DNAME                   VARCHAR(10)      NOT
    NULL,
    DNUMBER                 INTEGER          NOT
    NULL,
    MGRSSN       CHAR(9),
    MGRSTARTDATECHAR(9),
    PRIMARY KEY (DNUMBER),
    UNIQUE (DNAME),
    FOREIGN KEY (MGRSSN) REFERENCES EMP);
```

DROP TABLE

- Used to remove a relation (base table) and its definition
- The relation can no longer be used in queries, updates, or any other commands since its description no longer exists
- Example:

```
DROP TABLE  DEPENDENT;
```

ALTER TABLE

- Used to add an attribute to one of the base relations
  - The new attribute will have NULLs in all the tuples of the relation right after the command is executed; hence, the NOT NULL constraint is not allowed for such an attribute
    - Example:

```
ALTER TABLE EMPLOYEE ADD
JOB  VARCHAR(12);
```

- The database users must still enter a value for the new attribute JOB for each EMPLOYEE tuple.
  - This can be done using the UPDATE command.

Updates to tables

- **Alter**
  - **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.*
    - All tuples in the relation are assigned *null* as the value for the new attribute.
  - **alter table** *r* **drop** *A*
    - where *A* is the name of an attribute of relation *r*
    - Dropping of attributes not supported by many databases.

DML

- **Insert**
  - **insert into** *instructor* **values** (‘10211’, ’Smith’, ’Biology’, 66000);
- **Delete**
  - **delete from** *student*
- **Update**
  - **update** *student* **set** *sname* **=** *‘abc’* **where** *rollno = 5;*

Insertion

- Add a new tuple to *course*

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

- or equivalently

  insert into *course* (*course_id*, *title*, *dept_name*, *credits*)
  values (’CS-437’, ’Database Systems’, ’Comp. Sci.’, 4);

- Add a new tuple to *student* with *tot_creds* set to null

  insert into *student*
  values (’3003’, ’Green’, ’Finance’, *null*);

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 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 – Updates

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two **update** statements:

update *account*
set *balance* = *balance* ∗ 1.06
where *balance* > 10000

update *account*
set *balance* = *balance* ∗ 1.05
where *balance* ≤ 10000

  - The order is important
  - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

```
update account
set balance =  case
                  when balance <= 10000 then balance *1.05
                  else   balance * 1.06
               end
```

RETRIEVAL QUERIES IN SQL (CONTD.)

- Basic form of the SQL SELECT statement is called a *mapping* or a SELECT-FROM-WHERE *block*

SELECT <attribute list>
FROM <table list>
WHERE <condition>

- <attribute list> is a list of attribute names whose values are to be retrieved by the query
- <table list> is a list of the relation names required to process the query
- <condition> is a conditional expression that identifies the tuples to be retrieved by the 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.
- The result of an SQL query is a relation.

General Structure

SELECT ..... FROM ..... WHERE .....

SELECT [ALL / DISTINCT] *expr1* [AS *col1*], *expr2* [AS *col2*] ;

FROM *tablename* WHERE *condition*

General Structure

SELECT [ALL / DISTINCT] *expr1* [AS *col1*], *expr2* [AS *col2*] ;
FROM *tablename* WHERE *condition*

- The query will select rows from the source ***tablename*** and output the result in table form.
- Expressions ***expr1***, ***expr2*** can be :
  - (1) a column, or
  - (2) an expression of functions and fields.
- And ***col1***, ***col2*** are their corresponding column names in the output table.

General Structure

SELECT [ALL / DISTINCT] *expr1* [AS *col1*], *expr2* [AS *col2*] ;
FROM *tablename* WHERE *condition*

- DISTINCT will eliminate duplication in the output while ALL will keep all duplicated rows.
- ***condition*** can be :
  - (1) an inequality, or
  - (2) a string comparison
  - using logical operators AND, OR, NOT.

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*
- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g., *Name* ≡ *NAME* ≡ *name*
  - Some people use upper case wherever we use bold font.

The select Clause (Cont.)

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

  select distinct *dept_name*
  from *instructor*

- The 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*

- 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.

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 with salary > 80000

  select *name*
  from *instructor*
  where *dept_name* = 'Comp. Sci.' **and** *salary* > 80000
- Comparison results can be combined using the logical connectives **and, or,** and **not.**
- Comparisons can be applied to results of arithmetic expressions.

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.
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra).

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');

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*

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**

The Rename Operation

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

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

- Find the name, loan number and loan amount of all customers; rename the column name *loan-number* as *loan-id.*

  select *customer-name, borrower.loan-number* **as** *loan-id, amount*
  from *borrower, loan*
  where *borrower.loan-number = loan.loan-number*

String Operations

- SQL includes a string-matching operator for comparisons on character strings. Patterns are described using two special characters:
  - percent (%). The % character matches any substring.
  - underscore (_). The _ character matches any character.
- Find the names of all customers whose street includes the substring "Main".

  select *customer-name*
  from *customer*
  where *customer-street* **like** '%Main%'

- 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.

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  '\'
```

in that above we use backslash (\) as the escape character.

String Operations (Cont.)

- Patters 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.

General Structure

eg. 1 List all the student records.

```
SELECT * FROM student
```

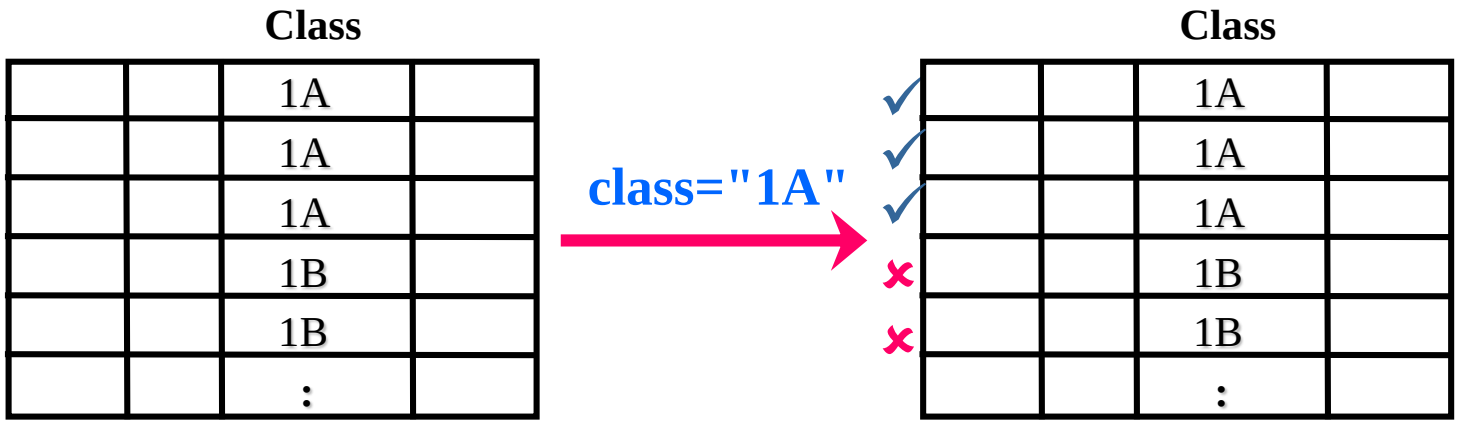
Result

id	name	dob	sex	class	mtest	hcode	dcode	remission
9801	Peter	06/04/86	M	1A	70	R	SSP	.F.
9802	Mary	01/10/86	F	1A	92	Y	HHM	.F.
9803	Johnny	03/16/86	M	1A	91	G	SSP	.T.
9804	Wendy	07/09/86	F	1B	84	B	YMT	.F.
9805	Tobe	10/17/86	M	1B	88	R	YMT	.F.
:	:	:	:	:	:	:	:	:

General Structure

eg. 2 List the names and house code of 1A students.

```
SELECT name, hcode, class FROM student ;
WHERE class="1A"
```



General Structure

eg. 2 List the names and house code of 1A students.

Result

name	hcode	class
Peter	R	1A
Mary	Y	1A
Johnny	G	1A
Luke	G	1A
Bobby	B	1A
Aaron	R	1A
:	:	:

General Structure

eg. 3 List the residential district of the Red House members.

```
SELECT DISTINCT dcode FROM student ;
WHERE hcode="R"
```

Result

dcode
HHM
KWC
MKK
SSP
TST
YMT

General Structure

eg. 5 List the names, id of 1A students with no fee remission.

```
SELECT name, id, class FROM student ;
WHERE class="1A" AND NOT remission
```

Result

name	id	class
Peter	9801	1A
Mary	9802	1A
Luke	9810	1A
Bobby	9811	1A
Aaron	9812	1A
Ron	9813	1A
Gigi	9824	1A
:	:	:

Comparison

expr IN (*value1*, *value2*, *value3*)

expr BETWEEN *value1* AND *value2*

expr LIKE "%_"

Comparison

eg. 7 List the students who were not born in January, March, June, September.

```
SELECT name, class, dob FROM student ;
WHERE MONTH(dob) NOT IN (1,3,6,9)
```

Result

name	class	dob
Wendy	1B	07/09/86
Tobe	1B	10/17/86
Eric	1C	05/05/87
Patty	1C	08/13/87
Kevin	1C	11/21/87
Bobby	1A	02/16/86
Aaron	1A	08/02/86
:	:	:

Comparison

eg. 8 List the 1A students whose Math test score is between 80 and 90 (incl.)

```
SELECT name, mtest FROM student ;
WHERE class="1A" AND ;
mtest BETWEEN 80 AND 90
```

Result

name	mtest
Luke	86
Aaron	83
Gigi	84

Comparison

eg. 9 List the students whose names start with "T".

```
SELECT name, class FROM student ;
WHERE name LIKE "T%"
```

Result

name	class
Tobe	1B
Teddy	1B
Tim	2A

Comparison

eg. 10 List the Red house members whose names contain "a" as the 2nd letter.

```
SELECT name, class, hcode FROM student ;
WHERE name LIKE "_a%" AND hcode="R"
```

Result

name	class	hcode
Aaron	1A	R
Janet	1B	R
Paula	2A	R

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 account balance at the Perryridge branch.

  select avg *(balance)*
  from *account*
  where *branch-name* = ‘Perryridge’

- Find the number of tuples in the *customer* relation.

  select count (*)
  from *customer*

- Find the number of depositors in the bank.

  select count (distinct *customer-name)*
  from *depositor*

Aggregate Functions – Group By

- Find the number of depositors for each branch.

  select *branch-name,* **count (distinct** *customer-name)*
  from *depositor, account*
  where *depositor.account-number = account.account-number*
  group by *branch-name*

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

Aggregate Functions – Having Clause

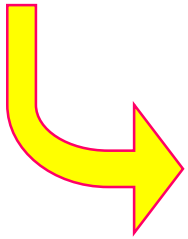
- Find the names of all branches where the average account balance is more than $1,200.

  select *branch-name,* **avg** *(balance)*
  from *account*
  group by *branch-name*
  having avg *(balance)* > 1200

  Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Grouping

SELECT ..... FROM ..... WHERE *condition* ;
GROUP BY *groupexpr* [HAVING *requirement*]



Group functions:

COUNT( ), SUM( ), AVG( ), MAX( ), MIN( )

- ***groupexpr*** specifies the related rows to be grouped as one entry. Usually it is a column.
- WHERE ***condition*** specifies the condition of individual rows before the rows are group. HAVING ***requirement*** specifies the condition involving the whole group.

Grouping

eg. 11 List the number of students of each class.

```
SELECT class, COUNT(*) FROM student ;
GROUP BY class
```

Result

class	cnt
1A	10
1B	9
1C	9
2A	8
2B	8
2C	6

Grouping

eg. 12 List the average Math test score of each class.

```
SELECT class, AVG(mtest) FROM student ;
GROUP BY class
```

Result

class	avg_mtest
1A	85.90
1B	70.33
1C	37.89
2A	89.38
2B	53.13
2C	32.67

Grouping

eg. 13 List the number of girls of each district.

```
SELECT dcode, COUNT(*) FROM student ;
WHERE sex="F" GROUP BY dcode
```

Result

dcode	cnt
HHM	6
KWC	1
MKK	1
SSP	5
TST	4
YMT	8

Grouping

eg. 14 List the max. and min. test score of Form 1 students of each district.

```
SELECT MAX(mtest), MIN(mtest), dcode ;
FROM student ;
WHERE class LIKE "1_" GROUP BY dcode
```

Result

max_mtest	min_mtest	dcode
92	36	HHM
91	19	MKK
91	31	SSP
92	36	TST
75	75	TSW
88	38	YMT

Grouping

eg. 15 List the average Math test score of the boys in each class. The list should not contain class with less than 3 boys.

```
SELECT AVG(mtest), class FROM student ;
WHERE sex="M" GROUP BY class ;
HAVING COUNT(*) >= 3
```

Result

avg_mtest	class
86.00	1A
77.75	1B
35.60	1C
86.50	2A
56.50	2B

IV *Display Order*

SELECT ..... FROM ..... WHERE .....
GROUP BY ..... ;
ORDER BY *colname* ASC / DESC

IV Display Order

eg. 18 List the number of students of each district (in desc. order).

```
SELECT COUNT(*) AS cnt, dcode FROM student ;
GROUP BY dcode ORDER BY cnt DESC
```

Result

cnt	docode
11	YMT
10	HHM
10	SSP
9	MKK
5	TST
2	TSW
1	KWC
1	MMK
1	SHT

V Output

eg. 21 Print the Red House members by their classes, sex and name.

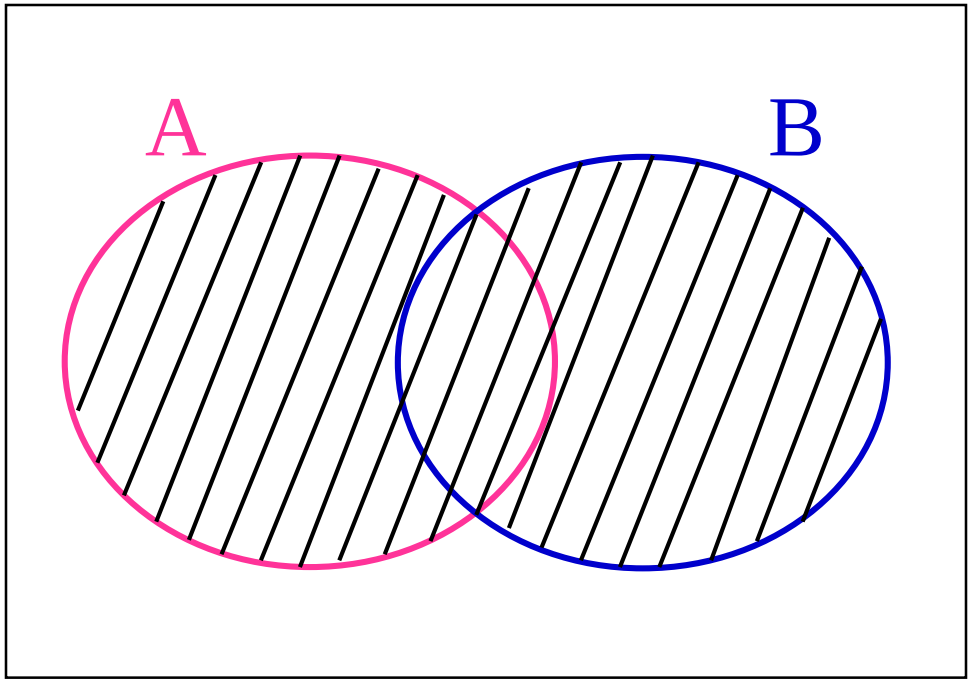
```
SELECT class, name, sex FROM student ;
WHERE hcode="R" ;
ORDER BY class, sex DESC, name TO PRINTER
```

Result

class	name	sex
1A	Aaron	M
1A	Peter	M
1A	Ron	M
1B	Tobe	M
1B	Janet	F
1B	Kitty	F
1B	Mimi	F
:	:	:

3 Union, Intersection and Difference of Tables

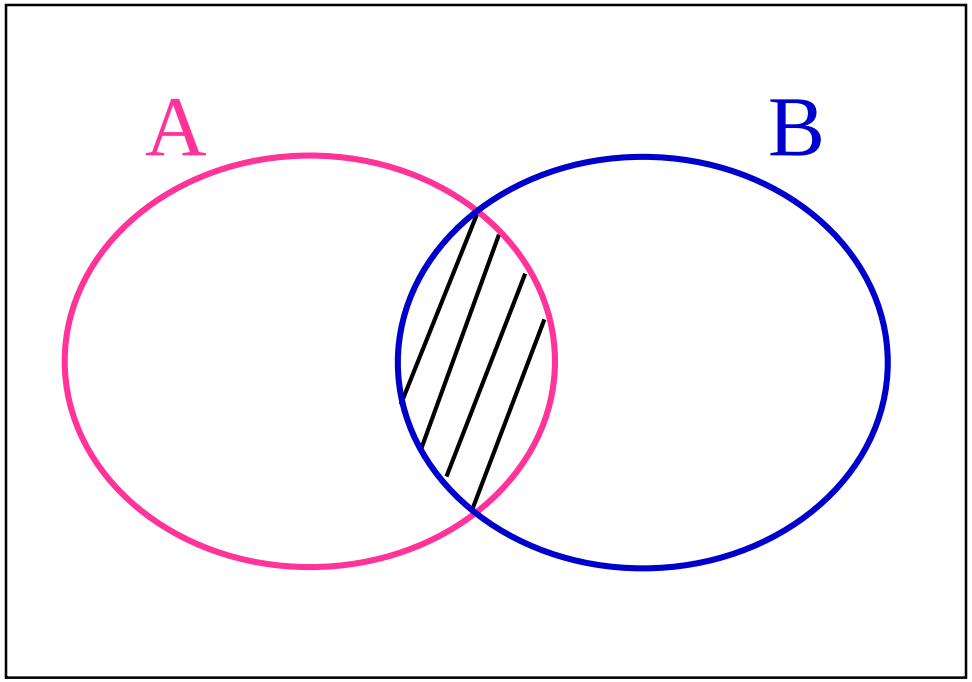
The ***union*** of A and B (A∪B)



A table containing all the rows from **A** and **B**.

3 Union, Intersection and Difference of Tables

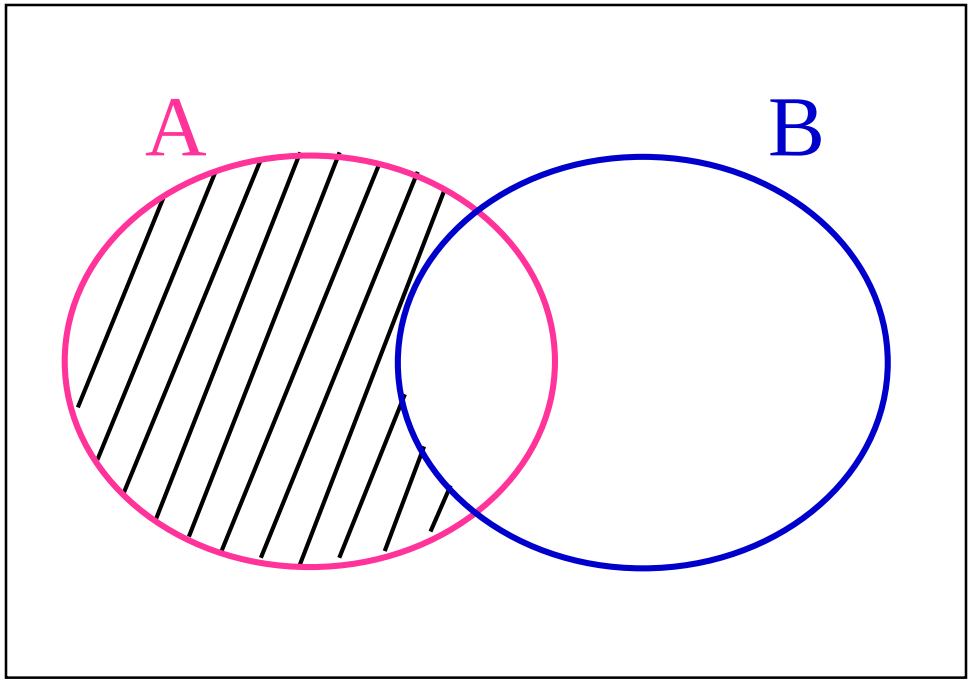
The ***intersection*** of A and B (A∩B)



A table containing only rows that appear in both **A** and **B**.

3 Union, Intersection and Difference of Tables

The ***difference*** of A and B (A–B)



A table containing rows that appear in **A** but not in **B**.

Set Operations

- Find all customers who have a loan, an account, or both:

  (select *customer-name* **from** *depositor*)
  union
  (select *customer-name* **from** *borrower)*

- Find all customers who have both a loan and an account.

  (**select** *customer-name* **from** *depositor*)
  intersect
  (select *customer-name* **from** *borrower)*

- Find all customers who have an account but no loan.

  (**select** *customer-name* **from** *depositor*)
  except
  (select *customer-name* **from** *borrower)*

Joined Relations

- Join operations take two relations and return as a result another relation.
- These additional 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)

Joined Relations – Datasets for Examples

- Relation *loan*

loan-number	*branch-name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

- Relation *borrower*

customer-name	*loan-number*
Jones	L-170
Smith	L-230
Hayes	L-155

- Note: borrower information missing for L-260 and loan information missing for L-155

Joined Relations – Examples

- *loan* **inner join** *borrower* **on**
  loan.loan-number = borrower.loan-number

loan-number	*branch-name*	*amount*	*customer-name*	*loan-number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

loan **left outer join** *borrower* **on**
loan.loan-number = borrower.loan-number

loan-number	*branch-name*	*amount*	*customer-name*	*loan-number*
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

Joined Relations – Examples

- *loan* **natural inner join** *borrower*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

loan **right outer join** *borrower*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	null	null	Hayes

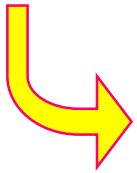
Joined Relations – Examples

- *loan* **full outer join** *borrower* **using** *(loan-number)*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	null	null	Hayes

4 Natural Join

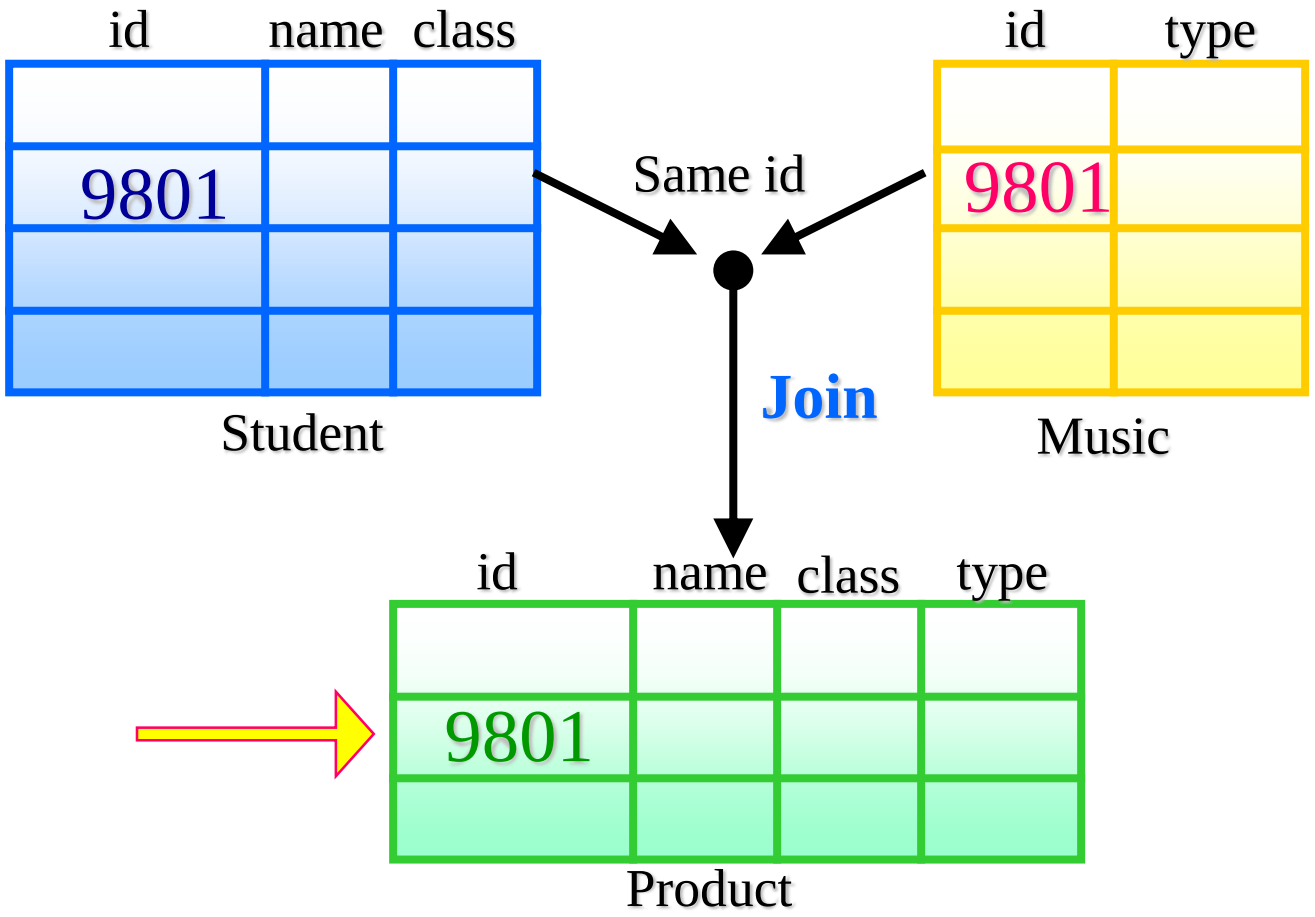
A **Natural Join** is a join operation that joins two tables by their common column. This operation is similar to the setting relation of two tables.



```
SELECT a.comcol, a.col1, b.col2, expr1, expr2 ;
FROM table1 a, table2 b ;
WHERE a.comcol = b.comcol
```

4 Natural Join

eg. 25 Make a list of students and the instruments they learn. (Natural Join)



4 Natural Join

eg. 25 Make a list of students and the instruments they learn. (Natural Join)

```
SELECT s.class, s.name, s.id, m.type ;
FROM student s, music m ;
WHERE s.id=m.id ORDER BY class, name
```

Result

class	name	id	type
1A	Aaron	9812	Piano
1A	Bobby	9811	Flute
1A	Gigi	9824	Recorder
1A	Jill	9820	Piano
1A	Johnny	9803	Violin
1A	Luke	9810	Piano
1A	Mary	9802	Flute
:	:	:	:

4 Natural Join

eg. 26 Find the number of students learning piano in each class.

Three Parts :

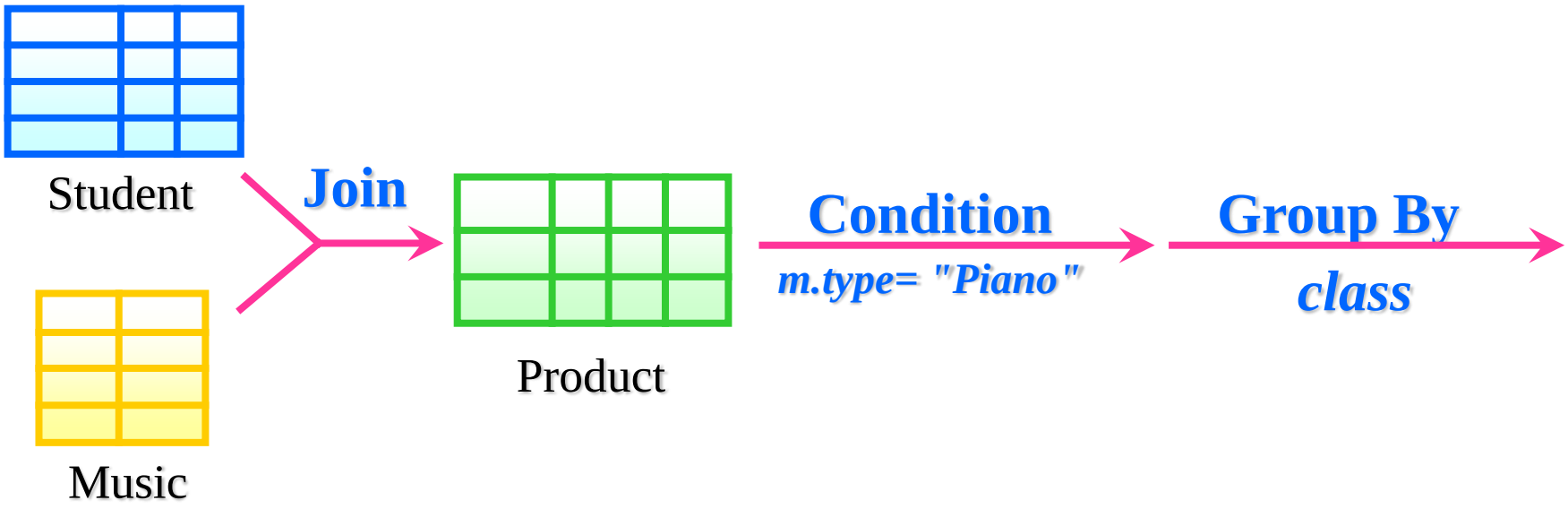
(1) Natural Join.

(2) Condition: *m.type="Piano"*

(3) GROUP BY class

4 Natural Join

eg. 26



4 Natural Join

eg. 26 Find the number of students learning piano in each class.

```
SELECT s.class, COUNT(*) ;
FROM student s, music m ;
WHERE s.id=m.id AND m.type="Piano" ;
GROUP BY class ORDER BY class
```

Result

class	cnt
1A	4
1B	2
1C	1

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  select *ID*, *name*, *dept_name*
  from *instructor*

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

- A view is defined using the **create view** statement which has the form

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

  where <query expression> is any legal SQL 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.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

Example Views

- A view of instructors without their salary

  create view *faculty* **as**
  select *ID*, *name*, *dept_name*
  from *instructor*

- Find all instructors in the Biology department

  select *name*
  from *faculty*
  where *dept_name* = 'Biology'

- Create a view of department salary totals

  create view *departments_total_salary*(*dept_name*, *total_salary*) **as**
  select *dept_name*, **sum** (*salary*)
  from *instructor*
  group by *dept_name*;

Views Defined Using Other Views

- **create view** *physics_fall_2009* **as**
  select *course*.*course_id*, *sec_id*, *building*, *room_number*
  from *course*, *section*
  where *course*.*course_id* = *section*.*course_id*
  and *course*.*dept_name* = 'Physics'
  and *section*.*semester* = 'Fall'
  and *section*.*year* = '2009';
- **create view** *physics_fall_2009_watson* **as**
  select *course_id*, *room_number*
  from *physics_fall_2009*
  where *building*= 'Watson';

Update of a View

- Add a new tuple to *faculty* view which we defined earlier

  insert into *faculty* **values** (’30765’, ’Green’, ’Music’);

  This insertion must be represented by the insertion of the tuple

  (’30765’, ’Green’, ’Music’, null)

  into the *instructor* relation

- Updates on more complex views are difficult or impossible to translate, and hence are disallowed.
- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation

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 all customers who have both an account and a loan at the bank.

  select distinct *customer-name*
  from *borrower*
  where *customer-name* **in (select** *customer-name*
  from depositor)

- Find all customers who have a loan at the bank but do not have an account at the bank

  select distinct *customer-name*
  from *borrower*
  where *customer-name* **not in (select** *customer-name*
  from *depositor)*

Example Query

- Find the names of all branches that have greater assets than all branches located in Brooklyn.

```
select branch-name
from branch
where assets > all
    (select assets
    from branch
    where branch-city = ‘Brooklyn’)
```

Example Query

- Find courses offered in summer 2009 and in winter 2010

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

- Find courses offered in summer 2009 but not in winter 2010

```
select distinct course_id
from section
where semester = 'summer' and year= 2009 and
        course_id  not in (select course_id
                           from section
                           where semester = 'winter' 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');
```

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');
```

Not Exists

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

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

  - First nested query lists all courses offered in Biology
  - Second nested query lists all courses a particular student took

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

Subqueries in the Form 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
- 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;
```

Subqueries in the Form Clause (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);
```

- Note: lateral is part of the SQL standard, but is not supported on many database systems; some databases such as SQL Server offer alternative syntax

With Clause

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

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;
```

Scalar Subquery

- Scalar subquery is one which is used where a single value is expected

```
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

Insertion (Cont.)

- 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

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.

Trigger Example

- Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
  - setting the account balance to zero
  - creating a loan in the amount of the overdraft
  - giving this loan a loan number identical to the account number of the overdrawn account
- The condition for executing the trigger is an update to the *account* relation that results in a negative *balance* value.

Trigger Example in SQL:1999

```
create trigger overdraft-trigger after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
        insert into borrower
           (select customer-name, account-number
            from depositor
            where nrow.account-number =
                        depositor.account-number);
        insert into loan values
           (n.row.account-number, nrow.branch-name,
                                          – nrow.balance);
        update account set balance = 0
        where account.account-number = nrow.account-number
end
```

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - **E.g. create trigger** *overdraft-trigger* **after update of** *balance* **on** *account*
- Values of attributes before and after an update can be referenced
  - **referencing old row as** : for deletes and updates
  - **referencing new row as :** for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. E.g. convert blanks to null.

```
create trigger setnull-trigger before update on r
referencing new row as nrow
for each row
    when nrow.phone-number = ' '
    set nrow.phone-number = null
```

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called *transition tables*) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

External World Actions

- We sometimes require external world actions, such as re-ordering an item whose quantity in a warehouse has become small, or turning on an alarm light, to be triggered on a database update
- Triggers cannot be used to directly implement external-world actions, BUT
  - Triggers can be used to record actions-to-be-taken in a separate table, and we can have an external process that repeatedly scans the table and carries out external-world actions.
- E.g. Suppose a warehouse has the following tables
  - *inventory(item, level):* How much of each item is in the warehouse
  - *minlevel(item, level) :* What is the minimum desired level of each item
  - *reorder(item, amount):* What quantity should we re-order at a time
  - *orders(item, amount) :* Orders to be placed (read by external process)

External World Actions (Cont.)

```
create trigger reorder-trigger after update of amount on inventory
referencing old row as orow, new row as nrow
for each row
        when nrow.level < = (select level
                                    from minlevel
                                    where minlevel.item = orow.item)
                and orow.level > (select level
                                        from minlevel
                                        where minlevel.item = orow.item)
  begin
          insert into orders
                (select item, amount
                  from reorder
                  where reorder.item = orow.item)
  end
```

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - maintaining summary data (e.g. total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of through a trigger