The SQL Standard

- SQL – Structured Query Language

  An international standard (ANSI, ISO) that specifies how
  - a relational schema is created
  - data is inserted / updated in the relations
  - data is queried
  - transactions are started and stopped
  - programs access data in the relations
  - and a host of other things are done

- Every relational database management system (RDBMS) is required to support / implement the SQL standard.
  - RDBMS vendors may give additional features
  - Downside of using vendor-specific features - portability

History of SQL

SEQUEL

- developed by IBM in early 70's
- relational query language as part of System-R project at IBM San Jose Research Lab.
- the earliest version of SQL

SQL evolution

- SQL- 86/89
- SQL- 92 - SQL2
- SQL- 99/03 - SQL3
  (includes object relational features)

And the evolution continues .....

Disclaimer: This module covers only important principles of SQL

Components of SQL Standard(1/2)

- *Data Definition Language* (DDL)
  Specifies constructs for schema definition, relation definition, integrity constraints, views and schema modification.

- *Data Manipulation Language* (DML)
  Specifies constructs for inserting, updating and querying the data in the relational instances (or tables).

- *Embedded SQL and Dynamic* SQL
  Specifies how SQL commands can be embedded in a high-level host language such as C, C++ or Java for programmatic access to the data.

Components of SQL Standard(2/2)

- *Transaction Control*

  Specifies how transactions can be started / stopped, how a set of concurrently executing transactions can be managed.

- *Authorization*

  Specifies how to restrict a user / set of users to access only certain parts of data, perform only certain types of queries etc.

Data Definition in SQL

Defining the schema of a relation

create table *r* (attributeDefinition-1, attributeDefinition-2,…,

name of the relation

attributeDefinition-n, [integrityConstraints-1],

[integrityConstraints-2],…,[integrityConstraints-m])

Attribute Definition –

attribute-name domain-type [NOT NULL] [DEFAULT v]

E.g.:

create table *example1* (A char(6) not null default "000000",

B int, C char(1) default "F");

Domain Types in SQL-92 (1/2)

- *Numeric data types*
  - integers of various sizes – INT, SMALLINT
  - real numbers of various precision – REAL, FLOAT, DOUBLE PRECISION
  - formatted numbers – DECIMAL (i, j) or NUMERIC (i, j)

    i – total number of digits (precision)

    j – number of digits after the decimal point (scale)
- *Character string data types*
  - fixed length – CHAR(n) – n: no. of characters
  - varying length – VARCHAR(n) – n: max.no. of characters
- *Bit string data types*
  - fixed length – BIT(n)
  - varying length – BIT VARYING(n)

Domain Types in SQL-92 (2/2)

- *Date data type*

  DATE type has 10 position format – YYYY-MM-DD

- *Time data type*

  TIME type has 8 position format – HH : MM : SS

- *Others*

  There are several more data types whose details are available in SQL reference books

Specifying Integrity Constraints in SQL

Also called Table Constraints

Included in the definition of a table

Key constraints

PRIMARY KEY (A_1,A_2,…,A_k)

specifies that {A_1,A_2,…,A_k} is the primary key of the table

UNIQUE (B_1,B_2,…,B_k)

specifies that {B_1,B_2,…,B_k} is a candidate key for the table

There can be more than one UNIQUE constraint but only one PRIMARY KEY constraint for a table.

Specifying Referential Integrity Constraints

FOREIGN KEY (A_1) REFERENCES r_2 (B_1)

- specifies that attribute A_1 of the table being defined, say r_1, is a *foreign key* referring to attribute B_1 of table r_2
- recall that this means:
  each value of column A_1 is either null or is one of the values appearing in column B_1 of r_2

Specifying What to Do if RIC Violation Occurs

RIC violation

- can occur if a referenced tuple is deleted or modified
- action can be specified for each case using qualifiers

  ON DELETE or ON UPDATE

Actions

- three possibilities can be specified

  SET NULL, SET DEFAULT, CASCADE
- these are actions to be taken on the referencing tuple
- SET NULL – foreign key attribute value to be set null
- SET DEFAULT – foreign key attribute value to be set to its default value
- CASCADE – delete the referencing tuple if the referenced tuple is deleted or update the FK attribute if the referenced tuple is updated

Table Definition Example

```
create table students (
       rollNo char(8) not null,
       name varchar(15) not null,
       degree char(5),
       year smallint,
       sex char not null,
       deptNo smallint,
       advisor char(6),
       primary key(rollNo),
       foreign key(deptNo) references
                                 department(deptId)
              on delete set null on update cascade,
       foreign key(advisor) references
                                 professor(empId)
              on delete set null on update cascade
 );
```

Modifying a Defined Schema

ALTER TABLE command can be used to modify a schema

Adding a new attribute

```
ALTER table student ADD address varchar(30);
```

Deleting an attribute

- need to specify what needs to be done about views or constraints that refer to the attribute being dropped
- two possibilities

  CASCADE – delete the views/constraints also

  RESTRICT – do not delete the attributes if there are some views/constraints that refer to it.
- ALTER TABLE student DROP degree RESTRICT

  Similarly, an entire table definition can be deleted

Data Manipulation in SQL

Basic query syntax

select $A_1, A_2, \ldots, A_m$ ← a set of attributes from relations $R_1, \ldots, R_p$ that are required in the output table.

from $R_1, R_2, \ldots, R_p$ ← the set of tables that contain the relevant tuples to answer the query.

where θ ← a boolean predicate that specifies when a combined tuple of $R_1, \ldots, R_p$ contributes to the output.

Equivalent to:

$$\pi_{A_1, A_2, \ldots, A_n}(\sigma_\theta(R_1 \times R_2 \times \ldots \times R_p))$$

Assuming that each attribute name appears exactly once in the table.

Meaning of the Basic Query Block

- The *cross product M* of the tables in the from clause would be considered.
  Tuples in *M* that satisfy the condition θ are *selected*.
  For each such tuple, values for the attributes $A_1, A_2, \ldots, A_m$ (mentioned in the select clause) are *projected*.

- This is a conceptual description
  - in practice more efficient methods are employed for evaluation.

- The word *select* in SQL should not be confused with select operation of relational algebra.

SQL Query Result

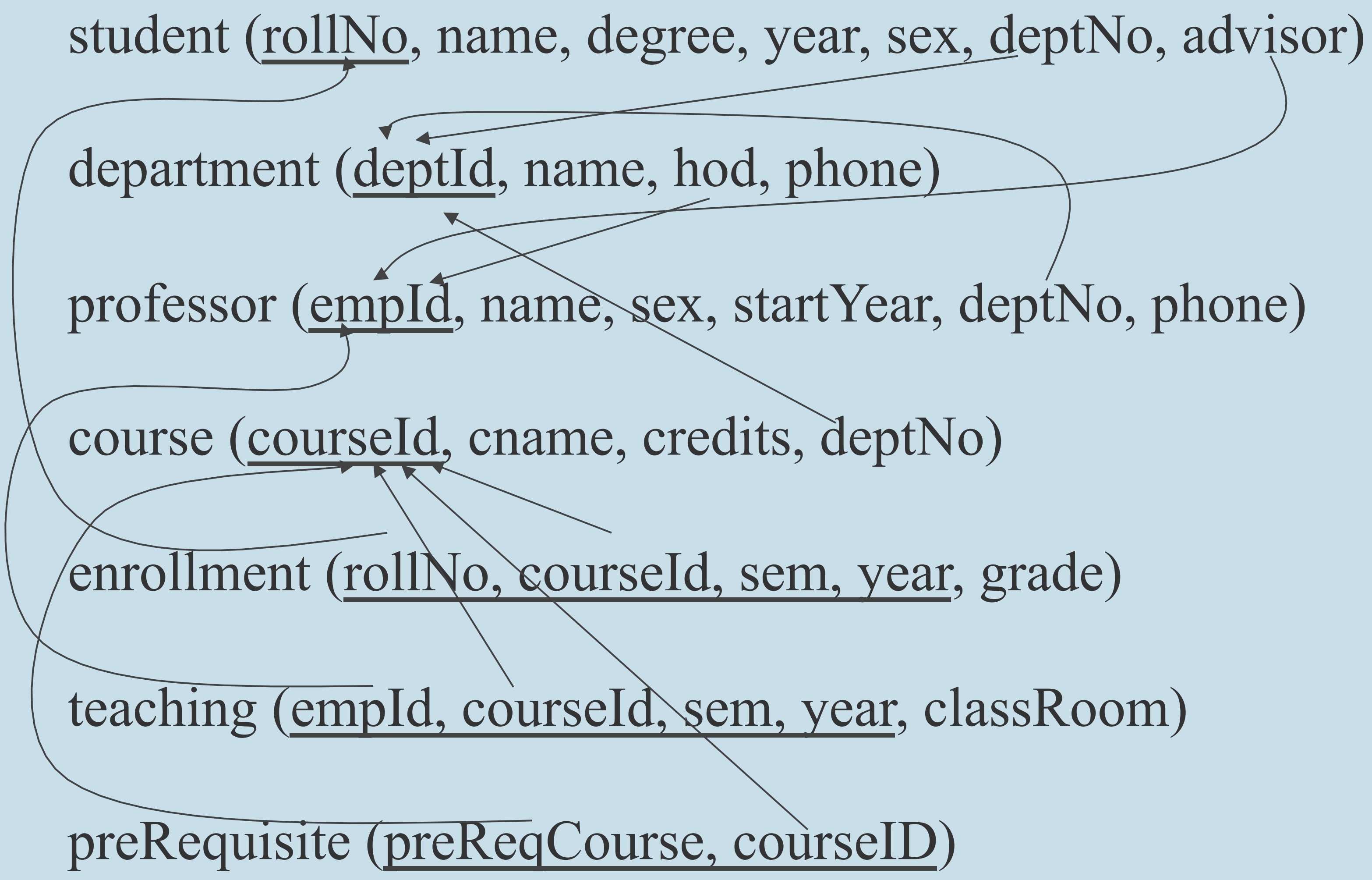
The result of any SQL query

- a table with *select* clause attributes as column names.
- duplicate rows may be present.
  - differs from the definition of a relation.
- duplicate rows can be eliminated by specifying DISTINCT keyword in the *select* clause, if necessary.

```
SELECT DISTINCT name
FROM student WHERE …
```

- duplicate rows are essential while computing aggregate functions (average, sum etc).
- removing duplicate rows involves additional effort and is done only when necessary.

Example Relational Scheme with RIC's shown



student (rollNo, name, degree, year, sex, deptNo, advisor)

department (deptId, name, hod, phone)

professor (empId, name, sex, startYear, deptNo, phone)

course (courseId, cname, credits, deptNo)

enrollment (rollNo, courseId, sem, year, grade)

teaching (empId, courseId, sem, year, classRoom)

preRequisite (preReqCourse, courseID)

Example Queries Involving a Single Table

Get the rollNo, name of all women students in the dept no. 5.

```
select rollNo, name
from student
where sex = ‘F’ and deptNo = 5;
```

Get the employee Id, name and phone number of professors in the CS dept (deptNo = 3) who have joined after 1999.

```
select empId, name, phone
from professor
where deptNo = 3 and startYear > 1999;
```

Examples Involving Two or More Relations (1/2)

Get the rollNo, name of students in the CSE dept (deptNo = 3)along with their advisor’s name and phone number.

```
select rollNo, s.name, f.name as advisorName,
               phone as advisorPhone
from student as s, professor as f
where s.advisor = f.empId and
             s.deptNo = 3;
```

attribute renaming in the output

table aliases are required if an attribute name appears in more than one table.
Also when *same* relation appears twice in the from clause.

table aliases are used to disambiguate the common attributes

Examples Involving Two or More Relations (2/2)

Get the names, employee ID's, phone numbers of professors in CSE dept who joined before 1995.

```
select empId, f.name, f.phone
from professor as f, department as d
where f.deptNo = d.deptId and
      d.name = ‘CSE’ and
      f.startYear < 1995
```

Nested Queries or Subqueries

While dealing with certain complex queries

- beneficial to specify part of the computation as a separate query & make use of its result to formulate the main query.
- such queries – nested / subqueries.

Using subqueries

- makes the main query easy to understand / formulate
- sometimes makes it more efficient also
  - sub query result can be computed once and used many times.
    - not the case with all subqueries.

Nested Query Example

Get the rollNo, name of students who have a lady professor as their advisor.

IN Operator: One of the ways of making use of the subquery result

Subquery computes the empId’s of lady professors

```
select s.rollNo, s.name
from student s
where s.advisor IN
        (select empId
         from professor
         where sex = ‘F’);
```

NOT IN can be used in the above query to get details of students who don’t have a lady professor as their advisor.

Set Comparison Operators

SQL supports several operators to deal with subquery results or in general with collection of tuples.

Combination of { =, <, ≤, ≥, >, < > } with keywords { ANY, ALL } can be used as set comparison operators.

```
Get the empId, name of the senior-most Professor(s):
```

```
select p.empId, p.name
from professors p
where p.startYear <= ALL ( select distinct startYear
                           from professor );
```

Semantics of Set Comparison Operators

op is one of $<, \leq, >, \geq, =, <>$

- *v op* ANY *S*
  true if for some member *x* of *S, v* op *x* is true
  false if for no member *x* of *S, v* op *x* is true

S is a subquery

- *v op* ALL *S*
  true if for every member *x* of *S, v* op *x* is true
  false if for some member *x* of *S, v* op *x* is not true

- IN is equivalent to = ANY
  NOT IN is equivalent to
  < > ALL

- *v* is normally a single attribute, but while using IN or NOT IN it can be a tuple of attributes

Correlated and Uncorrelated Nested Queries

If the nested query result is <u>independent</u> of the current tuple being examined in the outer query,
nested query is called *uncorrelated,*
otherwise, nested query is called *correlated.*

Uncorrelated nested query

- nested query needs to be computed only once.

Correlated nested query

- nested query needs to be re-computed for each row examined in the outer query.

Example of a Correlated Subquery

Get the roll number and name of students whose gender is same as their advisor's.

```
select s.rollNo, s.name
from student s
where s.sex = ALL ( select f.sex
                    from professor f
                    where f.empId = s.advisor );
```

The *EXISTS* Operator

Using *EXISTS*, we can check if a subquery result is non-empty

EXISTS(*S*) is *true* if *S* has at least one tuple / member
is *false* if *S* contain no tuples

```
Get the employee Id and name of professors
who advise at least one women student.
```

a correlated subquery

```
select f.empId, f.name
from professors f
where EXISTS ( select s.rollNo
               from student s
               where s.advisor = f.empId and
                     s.sex = 'F' );
```

SQL does not have an operator for universal quantification.

The *NOT EXISTS* Operator

```
Obtain the department Id and name of
departments that do not offer any 4 credit courses.
```

a correlated subquery

```
select d.deptId, d.name
from department d
where NOT EXISTS ( select courseId
                   from course c
                   where c.deptNo = d.deptId and
                         c.credits = 4 );
```

Queries with *existentially* quantified predicates can be easily specified using *EXISTS* operator.

Queries with *universally* quantified predicates can only be specified after translating them to use *existential* quantifiers.

Example Involving Universal Quantifier

`Determine the students who are enrolled for every course taught by Prof Ramanujam. Assume that Prof Ramanujam teaches at least one course.`

As SQL does not have universal quantifier, we will rewrite the query this way:

`Determine the student(s) who are st there` ***`does not`*** `exist a course taught by Prof Ramanujam which is` ***`not`*** `enrolled by the student.`

Same query expressed in TRC

Determine the students who are enrolled for ***every*** course taught by Prof Ramanujam. Assume that Prof Ramanujam teaches at least one course.

```
1.  {s.rollNo | student (s) ∧
2.    (∀c)(course (c) ∧
3.         ((∃t),(∃p)( teaching(t) ∧ professor(p) ∧
4.              t.courseId = c.courseId ∧
5.              p.name = "Ramanujam" ∧
6.              p.empId = t.empId )) →
7.                (∃e) (enrollment(e) ∧
8.                      e.courseId = c.courseId ∧
9.                      e.rollNo = s.rollNo)
10.   )
11.  }
```

An Example Involving the Universal Quantifier

Determine the students who are enrolled for every course taught by Prof Ramanujam. Assume that Prof Ramanujam teaches at least one course.

```
select s.rollNo, s.name
from student s
where NOT EXISTS ( select t.*
                   from teaching t, professor p
                   where t.empId = p.empId and
                     p.name = “Ramanujam” and
                     NOT EXISTS
                     (select e.*
                      from enrollment e
                      where e.courseId = t.courseId
                      and e.rollNo = s.rollNo)
                   );
```

Another Example Involving the Universal Quantifier

Determine the students who have obtained either S or A grade in all the pre-requisite courses of the course CS7890. It is known that CS7890 has at least one pre-requisite.

```
 select s.rollNo, s.name
from student s
where NOT EXISTS(select *
                 from preRequisite p
                 where p.courseId = “CS7890” and
                 NOT EXISTS
                  (select *
                   from enrollment e
                   where e.courseId = p.preReqcourse
                     and e.rollNo = s.rollNo and
                       e.grade = “S” or e.grade = “A”)
                   );
```

Missing *where* Clause

If the *where* clause in an SQL query is not specified, it is treated as - the where condition is true for all tuple combinations.

- Essentially no filtering is done on the cross product of from clause tables.

```
Get the name and contact phone of all Departments.
```

```
    select name, phone
    from department
```

Union, Intersection and Difference Operations

- In SQL, using operators *UNION, INTERSECT* and *EXCEPT*, one can perform set *union, intersection* and *difference* respectively.

- Results of these operators are ***sets*** –
  i.e duplicates are automatically removed.

- Operands need to be union compatible and also have *same* attribute names in the *same* order.

Example using UNION

Obtain the roll numbers of students who are currently enrolled for either CS2300 or CS2320 courses.

```
(SELECT rollNo
 FROM enrollment
 WHERE courseId = ‘CS2300’ and
           sem = ‘odd’ and year = ‘2019’ ) UNION
(SELECT rollNo
 FROM enrollment
 WHERE courseId = ‘CS2320’ and
          sem = ‘odd’ and year = ‘2019’ );
```

Equivalent to:

```
(SELECT rollNo
FROM enrollment
WHERE (courseId = ‘CS2300’ or courseID = ‘CS2320’)
   and sem = ‘odd’ and year = ‘2019’ )
```

Example using INTERSECTION

Obtain the roll numbers of students who are currently enrolled for both CS230 and CS232 Courses.

```
(select rollNo
from enrollment
where courseId = ‘CS2300’ and
      sem = ‘odd’ and year = ‘2019’ )

INTERSECT

(select rollNo
from enrollment
where courseId = ‘CS2320’ and
      sem = ‘odd’ and year = ‘2019’;
```

Example using EXCEPT

Obtain the roll numbers of students who are currently not enrolled for CS2300 course.

```
(SELECT rollNo
FROM enrollment
WHERE sem = ‘odd’ and year = ‘2019’)
EXCEPT
      (SELECT rollNo
      FROM enrollment
      WHERE courseId = ‘CS2300’ and
      sem = ‘odd’ and year = ‘2019’);
```

Aggregation of Data

Data analysis

- to get info on summary and trends in certain attributes
- need for computing aggregate values for data
- total value, average value etc

Aggregate functions in SQL

- five aggregate function are provided in SQL
- AVG, SUM, COUNT, MAX, MIN
- can be applied to any column of a table
- can be used in the *select* clause of SQL queries

Aggregate functions

Optional keyword

- AVG ([DISTINCT]A):
  computes the average of (distinct) values in column A
- SUM ([DISTINCT]A):
  computes the sum of (distinct) values in column A
- COUNT ([DISTINCT]A):
  computes the number of (distinct) values in column A or no. of tuples in result
- MAX (A): computes the maximum of values in column A
- MIN (A): computes the minimum of values in column A

Examples involving aggregate functions (1/2)

Suppose data about GATE exam in a particular year is available as a table with schema

gateMarks(regNo,name,sex,branch,city,state,marks)

Obtain the total number of students who have taken GATE in CS and their average marks

```
Select count(regNo) as CsTotal, avg(marks) as CsAvg
from gateMarks
where branch = 'CS'
```

Output

CStotal	CSavg

Get the maximum, minimum and average marks obtained by Students from the city of Hyderabad

```
Select max(marks), min(marks), avg(marks)
from gateMarks
where city = 'Hyderabad';
```

Examples involving aggregate functions (2/2)

Get the names of students who obtained the maximum marks in the branch of EC

```
Select name, max(marks)
from gateMarks
where branch = 'EC'
```

Will not work

Only aggregate functions can be specified here. It does not make sense to include normal attributes ! (unless they are grouping attributes – to be seen later)

```
Select regNo, name, marks
from gateMarks
where branch = 'EC' and marks = ANY
                    (select max(marks)
                     from gateMarks
                     where branch = 'EC');
```

Correct way of specifying the query

Date Aggregation and Grouping

Grouping

- Partition the set of tuples in a relation into groups based on certain criteria and compute aggregate functions for each group
- All tuples that agree on a set of attributes (i.e have the same value for each of these attributes) are put into a group

Called the grouping attributes

- The specified aggregate functions are computed for each group
- Each group contributes one tuple to the output
- All the grouping attributes *must* also appear in the select clause
  - the result tuple of the group is listed along with the values of the grouping attributes of the group

Examples involving grouping(1/2)

Determine the maximum of the GATE CS marks obtained by students in each city, for all Cities. Assume 4 cities exist - Hyderabad, Chennai, Mysore and Bangalore.

```
Select city, max(marks) as maxMarks
from gateMarks
where branch = 'CS'
group by city;
```

Grouping attributes must appear in the select clause

Grouping attribute

Result:

City	maxMarks
Hyderabad	87
Chennai	88
Mysore	90
Bangalore	86

Examples involving grouping(2/2)

In the University database, for each department, obtain the name, deptId and the total number of four credit courses offered by the department

```
Select deptId, name, count(*) as totalCourses
from department, course
where deptId = deptNo and credits = 4
group by deptId, name;
```

Having clause

After performing grouping, is it possible to report information about only a subset of the groups ?

- Yes, with the help of *having clause* which is always used in conjunction with Group By clause

```
Report the total enrollment in each course in the
even semester of 2014; include only the courses with
a minimum enrollment of 10.
```

```
Select courseId, count(rollNo) as Enrollment
from enrollment
where sem = even and year = 2014
group by courseId
having count(rollNo) ≥ 10;
```

Where clause versus Having clause

- Where clause
  - Performs tests on rows and eliminates rows not satisfying the specified condition
  - Performed *before* any grouping of rows is done
- Having clause
  - Always performed *after* grouping
  - Performs tests on groups and eliminates groups not satisfying the specified condition
  - Tests can only involve grouping attributes and aggregate functions

```
Select courseId, count(rollNo) as Enrollment
from enrollment
where sem = 2 and year = 2014
group by courseId
having count(rollNo) ≥ 10;
```

String Operators in SQL

- Specify strings by enclosing them in single quotes
  e.g., 'Chennai'

Common operations on strings –

- Pattern matching – using 'LIKE' comparison operator
  - specify patterns using special characters –
- Character '%' (percent) matches any Substring
  e.g., 'Ram%' matches any string starting with "Ram"
- Character '_' (underscore) matches any single character
  e.g., (a) '_ _ _ nagar' matches with any string ending
  with "nagar", with any 3 characters before that.
  (b) '_ _ _ _' matches any string with exactly four
  characters

Using the ‘LIKE’ operator

```
Obtain roll numbers and names of all students
whose names end with ‘Mohan’
```

```
    Select rollNo, name
    from student
    where name like ‘%Mohan’;
```

- Patterns are case sensitive.
- Special characters (percent, underscore) can be included in patterns using an escape character ‘\’ (backslash)

Join Operation

In SQL, usually joining of tuples from different relations is implicitly specified in the 'where' clause

```
Get the names of professors working in CSE dept.
```

```
    Select f.name
    from professor as f, department as d
    where f.deptNo = d.deptId and
          d.name = 'CSE';
```

The above query specifies joining of professor and department relations on condition *f.deptNo = d.deptId* and selection on department relation using *d.name = 'CSE'*

Explicit Specification of Joining in ‘From’ Clause

```
select f.name
 from (professor as f join department as d on
         f.deptNo = d.deptId)
 where d.name = ‘CSE’;
```

Join types:

1. inner join (default):
   from (r_1 inner join r_2 on <predicate>)
   use of just ‘join’ is equivalent to ‘inner join’
2. left outer join:
   from (r_1 left outer join r_2 on <predicate>)
3. right outer join:
   from (r_1 right outer join r_2 on <predicate>)
4. full outer join:
   from (r_1 full outer join r_2 on <predicate>)

Natural join

The adjective ‘natural’ can be used with any of the join types to specify natural join.

FROM (r_1 NATURAL <join type> r_2 [USING <attr. list>])

- natural join by default considers all common attributes
- a subset of common attributes can be specified in an optional USING <attr. list> phrase

REMARKS

- Specifying join operation explicitly goes against the spirit of declarative style of query specification
- But the queries may be easier to understand
- The feature is to be used judiciously

Views (or Virtual Tables)

- Views provide virtual relations which may contain data spread across different tables. Used by applications.
  - simplified query formulations
  - data hiding
  - a view of frequently used data – efficient query answering
- Once created, a view is always kept *up-to-date* by the RDBMS
- View is not part of conceptual schema
  - created to give a user group, concerned with a certain aspect of the information system, their *view* of the system
- View implementation
  - Views need not be stored as permanent tables
  - They can be created on-the-fly whenever needed or
  - They can also be *materialized* and kept up-to-date
- Tables involved in the view definition – base tables

Creating Views

CREATE VIEW v AS <query expr>

creates a view 'v', with structure and data defined by the outcome of the query expression

Create a view which contains name, employee Id and phone number of professors who joined CSE dept in or after the year 2000.

name of the view

```
create view profAft2K as
(Select f.name, empId, phone
 from professor as f, department as d
 where f.depNo = d.deptId and
       d.name = 'CSE' and
       f.startYear >= 2000);
```

If the details of a new CSE professor are entered into *professor* table, the above view gets updated automatically

Queries on Views

Once created a view can be used in queries just like any other table.

e.g. Obtain names of professors in CSE dept, who joined after 2000 and whose name starts with 'Ram'

```
select name
from profAft2K
where name like 'Ram%';
```

The definition of the view is stored in DBMS, and executed to create the temporary table (view), when encountered in query

Operations on Views

- Querying is allowed

- Update operations are usually restricted

  because – to update a view, we may require to modify many base tables

  – there may not be a unique way of updating the base tables to reflect the update on view

  – views may contain some aggregate values

  – ambiguity where primary key of a base table is not included in view definition.

Restrictions on Updating Views

- Updates on views defined on joining of more than one table are not allowed
- For example, updates on the following view are not allowed
- Note that we are not keeping information about when a student has completed the course in the view

create a view StudentGrades with rollNo, name, courseID and grade

```
create view studentGrade(rollNo,name,courseId,grade) as
     (select s.rollNo, s.name, e.courseId, e.grade
      from  student s, enrollment e
      where s.rollNo = e.rollNo);
```

- Suppose we want to update grade in the view from "U" to "D" for one particular course for a student, there will be ambiguity in doing the update on base tables.

Restrictions on Updating Views

- Updates on views defined with 'group by' clause and aggregate functions is not permitted, as a tuple in view will not have a corresponding tuple in base relation.

- For example, updates on the following view are not allowed

```
Create a view deptAvgCredits which contains the
average credits of courses offered by a dept.
```

```
create view deptAvgCredits(deptNo,avgCredits)
  as select deptNo, avg(credits)
     from course
     group by deptNo;
```

Restrictions on Updating Views

- Updates on views which do not include Primary Key of base table, are also not permitted
- For example, updates on the following view are not allowed

```
Create a view StudentPhone with Student name and
phone number.
```

```
  create view StudentPhone (sname,sphone) as
      (select name, phone
       from student);
```

View StudentPhone does not include Primary key of the base table.

Allowed Updates on Views

Updates to views are allowed only if

- defined on single base table
- not defined using 'group by' clause and aggregate functions
- view includes Primary Key of base table

Inserting data into a table

- Specify a tuple(or tuples) to be inserted

```
INSERT INTO student VALUES
(‘CS05D014’, ‘Mohan’, ‘PhD’,2005, ‘M’,3, ‘FCS008’),
(‘CS05S031’, ‘Madhav’, ‘MS’,2005, ‘M’,4, ‘FCE009’);
```

- Specify the result of query to be inserted

  INSERT INTO r_1 SELECT … FROM … WHERE …

- Specify that a sub-tuple be inserted

```
INSERT INTO student(rollNo, name, sex)
VALUES (CS05M022, ‘Rajasri’, ‘F’),
       (CS05B033, ‘Kalyan’, ‘M’);
```

  - *the attributes that can be NULL or have declared default values can be left-out to be updated later*

Deleting rows from a table

- Deletion of tuples is possible ; deleting only part of a tuple is not possible
- Deletion of tuples can be done *only from one* relation at a time
- Deleting a tuple might trigger further deletions due to *referentially triggered actions* specified as part of RIC's
- Generic form: `delete from r where <predicate>;`

Delete tuples from professor relation with start year as 1982.

```
delete from professor
 where startYear = 1982;
```

- If 'where' clause is not specified, then all the tuples of that relation are deleted (Be careful !)

A Remark on Deletion

- The where predicate is evaluated for each of the tuples in the relation to mark them as qualified for deletion *before* any tuple is actually deleted from the relation
- Note that the result may be different if tuples are deleted as and when we find that they satisfy the where condition!
- An example:

  Delete all tuples of students that scored the least marks in the CS branch:

```
DELETE
FROM gateMarks
WHERE branch = “CS” and
        marks = ANY ( SELECT MIN(marks)
                      FROM gateMarks
                      WHERE branch = “CS”)
```

Updating tuples in a relation

```
update r
set <<attr = newValue> list>
where <predicates>;
```

Change phone number of all professors working in CSE dept to “94445 22605”

```
update professors
set phone = ‘9444422605’
where deptNo = (select deptId
                from department
                where name = ‘CSE’);
```

If ‘where’ clause is not specified, values for the specified attributes in all tuples is changed.

Miscellaneous features in SQL (1/3)

- Ordering of result tuples can be done using 'order by' clause

  e.g., List the names of professors who joined after 1980, in alphabetic order.

  ```
  select name
  from professor
  where startYear > 1980
  order by name;
  ```

- Use of 'null' to test for a null value, if the attribute can take null

  e.g., Obtain roll numbers of students who don't have phone numbers

  ```
  select rollNo
  from student
  where phoneNumber is null;
  ```

Miscellaneous features in SQL (2/3)

- Use of 'between and' to test the range of a value
  e.g., `Obtain names of professors who have joined between 1980 and 1990`

```
select name
from professor
where startYear between 1980 and 1990;
```

- Change the column name in result relation
  e.g.,

```
select name as studentName, rollNo as studentNo
from student;
```

Miscellaneous features in SQL (3/3)

- Use of ‘distinct’ key word in ‘select’ clause to determine duplicate tuples in result.

Obtain all distinct branches of study for students

```
select distinct d.name
from student as s, department as d
where s.deptNo = d.deptId;
```

- Use of asterisk (*) to retrieve all the attribute values of selected tuples.

Obtain details of professors along with their department details.

```
select *
from professor as f, department as d
where f.deptNo = d.deptId;
```

Database System Architectures

Centralized Architecture – used long ago, before PCs were born

Complete DB functionality – storage, running application programs, transaction processing etc – is on one system - Server

Access systems are just display devices - terminals

Client/Server Architecture – two tier systems

Client – powerful enough to do local processing
- runs graphical user interface and application programs
- sends Database queries/updates to Server

Server – provides rest of the DB System functionality

Three Tier System Architectures – also possible – details left out here

Application Development Process: 2-tier Systems

Host language (HL) – the high-level programming language in which the application is developed (e.g., C, C++, Java etc.)

Managing Database Access – several approaches are available

- Embedded SQL approach − SQL commands are embedded in the HL programs
    - A static approach - SQL commands can't be given at runtime
    - Dynamic SQL
- Call Level Interface SQL/CLI − an API based approach
- JDBC − Java DB connectivity − an API based approach
- Use a Database programming language − Oracle's PL/SQL

Embedded SQL, Dynamic SQL − we will study in some detail
Other approaches − to be studied by students

Impedance Mismatch

- Impedance Mismatch:

  Problems due to difference in HL data model vs DB data model

  - Data types of HL vs those in DB

  - HL languages do not support set-of-records as supported by SQL
- Handling Data types
  - For each SQL attribute data type − corresponding HL data type
  - Specified as language *binding*
  - To be done for each host language
- Handling SQL query results
  - Results are either sets or multi-sets of tuples
  - A data structure to hold results and an iterator are needed
- Does not arise in case of dedicated DB programming languages

Embedded SQL Approach

Host language (HL) – the high-level programming language in which the application is developed (e.g., C, C++, Java etc.)

Embedded SQL approach:

- SQL statements are interspersed in HL program for application development
- Pre-compiler replaces these with suitable library calls
  - Library is supplied by the RDBMS vendor
- SQL commands − identified by special reserved words − EXEC SQL

Data transfer –

takes place through specially declared HL variables

Declaring Variables

Variables that need to be used in SQL statements are declared in a special section. These are called *shared* variables.

```
EXEC SQL BEGIN DECLARE SECTION
      char rollNo[9];                  //HL is C language
      char studName[20], degree[6];
      int year; char sex;
      int deptNo; char advisor[9];
EXEC SQL END DECLARE SECTION
```

Note that schema for student relation is

student(<u>rollNo</u>, name, degree, year, sex, deptNo, advisor)

Use in SQL statements: variable name is prefixed with a colon(:)

e.g., :ROLLNO in an SQL statement refers to rollNo variable

In HL program, shared variables can be used directly w/o colon.

Handling Error Conditions

The HL program needs to know if an SQL statement has executed successfully or otherwise

Special variable called SQLSTATE is used for this purpose
It is a string of 6 characters.

- SQLSTATE is set to appropriate value by the RDBMS run-time system after executing each SQL statement
- Non-zero values indicate errors in execution
    - Different values indicate different types of error situations
- SQLSTATE variable <u>must</u> be declared in the HL program
- HL program needs to check for error situations and handle them appropriately.

Database Connections in Embedded SQL Approach

DB connection

- Needs to be established before accessing the DB in the app pgm
- Specify the particular server and authenticate the application
- Several connections - to access 2 or more DB servers
- Only one connection can be *active* at any time

SQL Commands

```
    CONNECT TO <serverName> AS <connName>
    AUTHORIZATION <uName, passWd>
```

To change to a different server

```
SET CONNECTION <connName>
```

```
DISCONNECT <connName>
```

Embedded SQL Statements – An example

Suppose we collect data through user interface into variables
rollNo, studName, degree, year, sex, deptNo, advisor

A row into the student table can be inserted as follows:

```
 EXEC SQL INSERT INTO STUDENT
   VALUES (:rollNo,:studName,:degree,
          :year,:sex,:deptNo,:advisor);
```

Query result handling and Cursors

- HL languages do not support set-of-records as supported by SQL
- A *cursor* is a mechanism which allows us to retrieve one row at a time from the result of a query
- We can declare a cursor on any SQL query
- Once declared, we use open, fetch, move and close commands to work with cursors
- We usually need a cursor when embedded statement is a SELECT query
- INSERT, DELETE and UPDATE do not need a cursor.

Embedded SQL - Cursors (1/2)

We do not need a cursor if the query results in a single row.

e.g.,
```
EXEC SQL SELECT s.name, s.sex
         INTO :name, :sex
         FROM student s
         WHERE s.rollNo = :rollNo;
```

- Result row values name and phone are assigned to HL variables :name and :phone, using INTO clause
- Cursor is not required as the result always contains only one row (*rollNo* is a key for *student* relation)

Embedded SQL - Cursors (2/2)

- If the result contains more than one row, cursor declaration is needed

e.g.,
```
select s.name, s.degree
from student s
where s.sex = 'F';
```

- Query results in a collection of rows
- HL program has to deal with set of records.
- The use of 'INTO' will not work here
- We can solve this problem by using a *cursor*.

Declaring a cursor on a query

Cursor name

```
declare studInfo cursor for
        select name, degree
        from student
        where sex = 'F';
```

- Command OPEN studInfo; opens the cursor and makes it point to first record
- To read current row of values into HL variables:
  FETCH studInfo INTO :name, :degree;
- After executing FETCH statement cursor is pointed to next row by default
- Cursor movement can be optionally controlled by the programmer
- After reading all records we close the cursor using the CLOSE studInfo command.

Dynamic SQL

- Useful for applications to generate and run SQL statements, based on user inputs
- Queries may not be known in advance

e.g.,
```
char sqlString[ ] = {“select * from student”};
    EXEC SQL PREPARE runQ FROM sqlString;
    EXEC SQL EXECUTE runQ;
```

- *sqlString* is a C variable that holds user submitted query
  - Typically built by previous statements in the HL program using end-user inputs.
- *runQ* is an SQL variable that holds the SQL statement.

Connecting to Database from HL − Other Approaches

ODBC (Open Database Connectivity), SQL/CLI and JDBC

- accessing database and data is through an API
- many DBMSs can be accessed
- no restriction on number of active connections
- appropriate drivers are required
- steps in accessing data from a HL program
    - select the data source
    - load the appropriate driver dynamically
    - establish the connection
    - authenticate the client
    - work with database
    - close the connection.

A comparison of the Approaches

Embedded SQL Approach
+ queries are part of source code, syntax-check at compile time
+ application programs are easy to understand, readable
+ development is eaiser
 - Any changes to queries : recompilation required
 - Complex applications requiring runtime query creation are difficult

API based Approach
+ More flexibility in programming
+ Complex applications can be developed
- Application development is complex, error-prone

DB Language Approach
+ No impedance mismatch
- Programmers need to learn a new language; apps not portable