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)

Correct way of specifying the query

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


Data 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

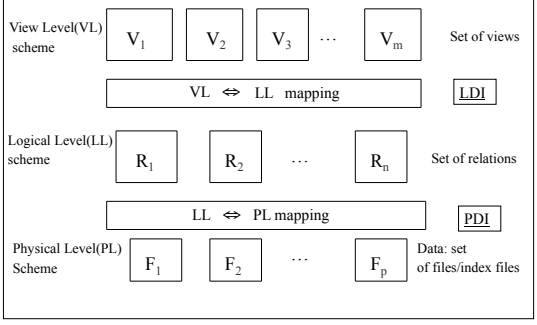
REMARKS

- 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

Abstraction Levels in a **DBMS**: Three-Schema Architecture



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 r1 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(rollNo, 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 must 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