3. User Interfaces and SQL Language*

User interface of DBMS

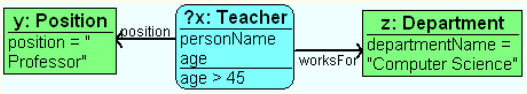
- A DBMS must offer some interfaces to support user to access database, including:
  - Query Languages
  - Interface and maintaining tools (GUI)
  - APIs
  - Class Library
- Query Languages
  - Formal Query Language
  - Tabular Query Language
  - Graphic Query Language
  - Limited Natural Language Query Language

Example of TQL & GQL

Find the names of all students in the department of Info. Science

Student	Sno	Sname	Ssex	Sage	Sdept
		P.T			IS

操作符，表示打印（print） 示例元素，域变量 条件
实际是显示



Find all Teachers, which have position="Professor" and which have age>"45" and which work for department="Computer Science"

Relational Query Languages

- Query languages: Allow manipulation and retrieval of data from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on logic.
  - Allows for much optimization.
- Query Languages != programming languages!
  - QLs not expected to be "Turing complete".
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.

Formal Relational Query Languages

- Two mathematical Query Languages form the basis for "real" languages (e.g. SQL), and for implementation:
  - *Relational Algebra*: More operational, very useful for representing execution plans.
  - *Relational Calculus*: Lets users describe what they want, rather than how to compute it. (Non-operational, *declarative*.)
- The most successful relational database language --- SQL (Structured Query Language, Standard Query Language(1986))

SQL Language

- It can be divided into four parts according to functions.
  - Data Definition Language (DDL), used to define, delete, or alter data schema.
  - Query Language (QL), used to retrieve data
  - Data Manipulation Language (DML), used to insert, delete, or update data.
  - Data Control Language (DCL), used to control user's access authority to data.
- QL and DML are introduced in detail in this chapter.

Important terms and concepts

- Base table
- View
- Data type supported
- NULL
- UNIQUE
- DEFAULT
- PRIMARY KEY
- FOREIGN KEY
- CHECK (Integration Constraint)

Example Instances

- We will use these instances of the Sailors, Reserves and Boats relations in our examples.

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

B1

bid	bname	color
101	tiger	red
103	lion	green
105	hero	blue

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Basic SQL Query

```
SELECT    [DISTINCT] target-list
FROM      relation-list
WHERE     qualification
```

- *relation-list* A list of relation names (possibly with a *range-variable* after each name).
- *target-list* A list of attributes of relations in *relation-list*
- *qualification* Comparisons combined using AND, OR and NOT.
- DISTINCT is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are *not* eliminated!

Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of *relation-list*.
  - Discard resulting tuples if they fail *qualifications*.
  - Delete attributes that are not in *target-list*.
  - If DISTINCT is specified, eliminate duplicate rows.
- This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers.*

Simple Example

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND R.bid=103
```

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	**rusty**	10	35.0	58	103	11/12/96

result

A Note on Range Variables

- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND bid=103
```

OR

```
SELECT  sname
FROM    Sailors, Reserves
WHERE   Sailors.sid=Reserves.sid
        AND bid=103
```

It is good style, however, to use range variables always!

Find sailors who've reserved at least one boat

```
SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
```

- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing *S.sid* by *S.sname* in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

Expressions and Strings

```
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM Sailors S
WHERE S.sname LIKE 'B_%B'
```

- Illustrates use of arithmetic expressions and string pattern matching: *Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names begin and end with B and contain at least three characters.*
- AS and = are two ways to name fields in result.
- LIKE is used for string matching. '_' stands for any one character and '%' stands for 0 or more arbitrary characters.

Find sid's of sailors who've reserved a red or a green boat

- UNION: Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).
- If we replace OR by AND in the first version, what do we get?
- Also available: EXCEPT (What do we get if we replace UNION by EXCEPT?)

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
  AND (B.color='red' OR B.color='green')
```

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
      AND B.color='red'
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
      AND B.color='green'
```


Find sid's of sailors who've reserved a red and a green boat

- INTERSECT: Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- Included in the SQL/92 standard, but some systems don't support it.
- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

```
SELECT S.sid
FROM Sailors S, Boats B1, Reserves R1,
     Boats B2, Reserves R2
WHERE S.sid=R1.sid AND R1.bid=B1.bid
  AND S.sid=R2.sid AND R2.bid=B2.bid
  AND (B1.color='red' AND
       B2.color='green')
```

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
      AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
      AND B.color='green'
```


Nested Queries

Find names of sailors who've reserved boat #103:

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R
                WHERE R.bid=103)
```

- A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)
- To find sailors who've *not* reserved #103, use NOT IN.
- To understand semantics of nested queries, think of a *nested loops* evaluation: *For each Sailors tuple, check the qualification by computing the subquery.*

Nested Queries with Correlation

Find names of sailors who've reserved boat #103:

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
              FROM Reserves R
              WHERE R.bid=103 AND S.sid=R.sid)
```

- EXISTS is another set comparison operator, like IN.
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple.
- How to *find names of sailors who've reserved boat #103 and reserved only one time*?

Nested Queries with Correlation

- *Find IDs of boats which are reserved by only one sailor.*

```
SELECT bid
FROM Reserves R1
WHERE bid NOT IN (
            SELECT bid
            FROM Reserves R2
            WHERE R2.sid ¬= R1.sid)
```


More on Set-Comparison Operators

- We've already seen IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: *op* ANY, *op* ALL, *op* IN
  <,>,=,≤,≥,≠
- *Find sailors whose rating is greater than that of some sailor called Horatio*:

```
SELECT *
FROM Sailors S
WHERE S.rating > ANY (SELECT S2.rating
                      FROM Sailors S2
                      WHERE S2.sname='Horatio')
```


Rewriting INTERSECT Queries Using IN

Find sid's of sailors who've reserved both a red and a green boat:

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid AND B.color='red'
      AND S.sid IN (SELECT S2.sid
                    FROM Sailors S2, Boats B2, Reserves R2
                    WHERE S2.sid=R2.sid AND R2.bid=B2.bid
                          AND B2.color='green')
```

- Similarly, EXCEPT queries re-written using NOT IN.
- To find *names* (not *sid*'s) of Sailors who've reserved both red and green boats, just replace *S.sid* by *S.sname* in SELECT clause. (What about INTERSECT query?)

Division in SQL

Find sailors who've reserved all boats.

Solution 1:

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS
        ((SELECT B.bid
          FROM Boats B)
         EXCEPT
         (SELECT R.bid
          FROM Reserves R
          WHERE R.sid=S.sid))
```


Division in SQL

Solution 2:

Let's do it the hard way, without EXCEPT:

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS (SELECT B.bid
                  FROM Boats B
                  WHERE NOT EXISTS (SELECT R.bid
                                    FROM Reserves R
                                    WHERE R.bid=B.bid
                                          AND R.sid=S.sid))
```

Sailors S such that ...

there is no boat B without ...

a Reserves tuple showing S reserved B

Aggregate Operators

- Significant extension of relational algebra.
  - COUNT (*)
  - COUNT ([DISTINCT] A)
  - SUM ([DISTINCT] A)
  - AVG ([DISTINCT] A)
  - MAX (A)
  - MIN (A)
- *A* is single column

Examples of Aggregate Operators

```
SELECT COUNT (*)
FROM Sailors S
```

```
SELECT COUNT (DISTINCT S.rating)
FROM Sailors S
WHERE S.sname='Bob'
```

```
SELECT AVG (S.age)
FROM Sailors S
WHERE S.rating=10
```

```
SELECT AVG (DISTINCT S.age)
FROM Sailors S
WHERE S.rating=10
```

```
SELECT S.sname
FROM Sailors S
WHERE S.rating= (SELECT MAX(S2.rating)
                 FROM Sailors S2)
```


Find name and age of the oldest sailor(s)

- The first query is illegal! (We'll look into the reason a bit later, when we discuss GROUP BY.)
- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is not supported in some systems.

```
SELECT S.sname, MAX (S.age)
FROM Sailors S
```

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age =
      (SELECT MAX (S2.age)
       FROM Sailors S2)
```

```
SELECT S.sname, S.age
FROM Sailors S
WHERE (SELECT MAX (S2.age)
       FROM Sailors S2)
       = S.age
```


Motivation for Grouping

- So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.
- Consider: *Find the age of the youngest sailor for each rating level.*
  - In general, we don't know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

For i = 1, 2, ... , 10:

```
SELECT MIN (S.age)
FROM Sailors S
WHERE S.rating = i
```


Queries With GROUP BY and HAVING

```
SELECT      [DISTINCT] target-list
FROM        relation-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
```

- The *target-list* contains
  - (i) attribute names
  - (ii) terms with aggregate operations (e.g., MIN (*S.age*)).
- The attribute list (i) must be a subset of *grouping-list*. Intuitively, each answer tuple corresponds to a *group*, and these attributes must have a single value per group. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.)

Conceptual Evaluation

- The cross-product of *relation-list* is computed, tuples that fail *qualification* are discarded, '*unnecessary*' fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.
- The *group-qualification* is then applied to eliminate some groups. Expressions in *group-qualification* must have a *single value per group!*
  - In fact, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

Find age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors

```
SELECT S.rating, MIN (S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```

Answer relation:

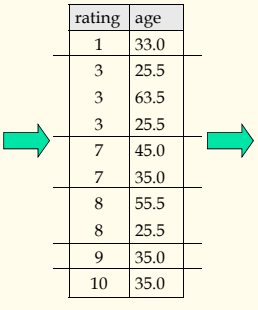
rating	minage
3	25.5
7	35.0
8	25.5

Sailors instance:

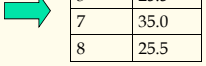
sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

Find age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors.

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5



rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

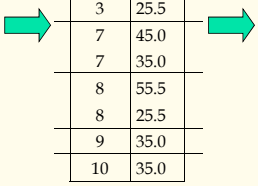


rating	minage
3	25.5
7	35.0
8	25.5

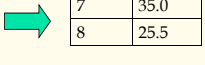
Find age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors and with every sailor under 60.

HAVING COUNT (*) > 1 AND EVERY (S.age <=60)

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5



rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0



rating	minage
7	35.0
8	25.5

What is the result of changing EVERY to ANY?

For each red boat, find the number of reservations for this boat

```
SELECT  B.bid,  COUNT (*) AS scount
FROM  Boats B, Reserves R
WHERE  R.bid=B.bid AND B.color='red'
GROUP BY  B.bid
```

- Grouping over a join of two relations.
- What do we get if we remove *B.color='red'* from the WHERE clause and add a HAVING clause with this condition?

Find age of the youngest sailor with age > 18, for each rating with at least 2 sailors (of any age)

```
SELECT  S.rating,  MIN (S.age)
FROM  Sailors S
WHERE  S.age > 18
GROUP BY  S.rating
HAVING  1 < (SELECT  COUNT (*)
             FROM  Sailors S2
             WHERE  S2.rating = S.rating)
```

rating	minage
3	25.5
7	35.0
8	25.5
10	35.5

- Shows HAVING clause can also contain a sub-query.
- Compare this with the query where we considered only ratings with 2 sailors over 18!
- What if HAVING clause is replaced by:
  - HAVING COUNT(*) >1

Find those ratings for which the average age is the minimum over all ratings

- Aggregate operations cannot be nested! WRONG:

```
SELECT  S.rating
FROM  Sailors S
WHERE  S.age = (SELECT  MIN (AVG (S2.age))
                FROM Sailors S2)
```

- Correct solution (in SQL/92):

```
SELECT  Temp.rating
FROM  (SELECT  S.rating, AVG (S.age) AS avgage
       FROM  Sailors S
       GROUP BY  S.rating) AS Temp
WHERE  Temp.avgage = (SELECT  MIN (Temp.avgage)
                      FROM  Temp)
```

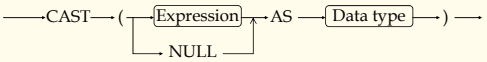

Null Values

- Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *inapplicable* (e.g., no spouse's name).
  - SQL provides a special value *null* for such situations.
- The presence of *null* complicates many issues. E.g.:
  - Special operators needed to check if value is/is not *null*.
  - Is *rating>8* true or false when *rating* is equal to *null*? What about AND, OR and NOT connectives?
  - We need a 3-valued logic (true, false and *unknown*).
  - Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don't evaluate to true.)
  - New operators (in particular, *outer joins*) possible/needed.

Some New Features of SQL

- **CAST expression**
- CASE expression
- Sub-query
- Outer Join
- Recursion

CAST Expression



- Change the expression to the target data type
- Valid target type
- Use
  - Match function parameters

    substr(string1, CAST(x AS Integer), CAST(y AS Integer))
  - Change precision while calculating

    CAST (elevation AS Decimal (5,0))
  - Assign a data type to NULL value

CAST Expression

- Example:

  Students (name, school)

  Soldiers (name, service)

```
CREATE VIEW prospects (name, school, service) AS
      SELECT name, school, CAST(NULL AS Varchar(20))
      FROM Students
UNION
      SELECT name, CAST(NULL AS Varchar(20)), service
      FROM Soldiers ;
```


Some New Features of SQL

- CAST expression
- **CASE expression**
- Sub-query
- Outer Join
- Recursion

CASE Expression

- Simple form :

  Officers (name, status, rank, title)

```
SELECT name, CASE status
                    WHEN 1 THEN 'Active Duty'
                    WHEN 2 THEN 'Reserve'
                    WHEN 3 THEN 'Special Assignment'
                    WHEN 4 THEN 'Retired'
                    ELSE 'Unknown'
               END AS status
FROM Officers ;
```


CASE Expression

- General form (use searching condition):

  Machines (serialno, type, year, hours_used, accidents)
- *Find the rate of the accidents of "chain saw" in the whole accidents :*

```
SELECT sum (CASE
                    WHEN type='chain saw' THEN accidents
                    ELSE 0e0
               END) / sum (accidents)
FROM Machines;
```

CASE Expression

- *Find the average accident rate of every kind of equipment :*

```
SELECT type, CASE
                    WHEN sum(hours_used)>0 THEN
                          sum(accidents)/sum(hours_used)
                    ELSE NULL
               END AS accident_rate
FROM Machines
GROUP BY type;
```

(Because some equipments maybe not in use at all, their hours_used is 0. Use CASE can prevent the expression divided by 0.)

CASE Expression

- Compared with

```
SELECT type, sum(accidents)/sum(hours_used)
FROM Machines
GROUP BY type
HAVING sum(hours_used)>0;
```


Some New Features of SQL

- CAST expression
- CASE expression
- **Sub-query**
- Outer Join
- Recursion

Sub-query

- Embedded query & embedded query with correlation
- The functions of sub-queries have been enhanced in new SQL standard. Now they can be used in SELECT and FROM clause
  - Scalar sub-query
  - Table expression
  - Common table expression

Scalar Sub-query

- The result of a sub-query is a single value. It can be used in the place where a value can occur.
- *Find the departments whose average bonus is higher than average salary :*

```
SELECT d.deptname, d.location
FROM dept AS d
WHERE (SELECT avg(bonus)
          FORM emp
          WHERE deptno=d.deptno)
       > (SELECT avg(salary)
          FORM emp
          WHERE deptno=d.deptno)
```


Scalar Sub-query

- *List the deptno, deptname, and the max salary of all departments located in New York :*

```
SELECT d.deptno, d.deptname, (SELECT MAX (salary)
                                FROM emp
                                WHERE deptno=d.deptno) AS maxpay
FROM dept AS d
WHERE d.location = 'New York' ;
```

Table Expression

- The result of a sub-query is a table. It can be used in the place where a table can occur.

```
SELECT startyear, avg(pay)
FROM (SELECT name, salay+bonus AS pay,
             year(startdate) AS startyear
      FROM emp) AS emp2
GROUP BY startyear;
```

- *Find departments whose total payment is greater than 200000*

```
SELECT deptno, totalpay
FROM (SELECT deptno, sum(salay)+sum(bonus) AS totalpay
      FROM emp
      GROUP BY deptno) AS payroll
WHERE totalpay>200000;
```

- Table expressions are temporary views in fact.

Common Table Expression

- In some complex query, a table expression may need occurring more than one time in the same SQL statements. Although it is permitted, the efficiency is low and there maybe inconsistency problem.
- WITH clause can be used to define a common table expression. In fact, it defines a temporary view.
- *Find the department who has the highest total payment :*

Common Table Expression

- *Find the department who has the highest total payment :*

```
WITH payroll (deptno, totalpay) AS
     (SELECT deptno, sum(salary)+sum(bonus)
     FROM emp
     GROUP BY deptno)
SELECT deptno
FROM payroll
WHERE totalpay = (SELECT max(totalpay)
                  FROM payroll);
```

- Common table expression mainly used in queries which need multi level focuses.

Common Table Expression

- *Find department pairs, in which the first department's average salary is more than two times of the second one's :*

```
WITH deptavg (deptno, avgsal) AS
     (SELECT deptno, avg(salary)
     FROM emp
     GROUP BY deptno)
SELECT d1.deptno, d1.avgsal, d2.deptno, d2.avgsal
FROM deptavg AS d1, deptavg AS d2
WHERE d1.avgsal>2*d2.avgsal;
```


Some New Features of SQL

- CAST expression
- CASE expression
- Sub-query
- **Outer Join**
- Recursion

Outer Join

Teacher (name, rank)
Course (subject, enrollment, quarter, teacher)

```
WITH
   innerjoin(name, rank, subject, enrollment) AS
         (SELECT t.name, t.rank, c.subject, c.enrollment
          FROM teachers AS t, courses AS c
          WHERE t.name=c.teacher AND c.quarter='Fall 96') ,
   teacher-only(name, rank) AS
         (SELECT name, rank
          FROM teachers
          EXCEPT ALL
          SELECT name, rank
          FROM innerjoin) ,
   course-only(subject, enrollment) AS
         (SELECT subject, enrollment
          FROM courses
          EXCEPT ALL
          SELECT subject, enrollment
          FROM innerjoin)
```

Outer Join

```
SELECT name, rank, subject, enrollment
FROM innerjoin
UNION ALL
SELECT name, rank,
        CAST (NULL AS Varchar(20)) AS subject,
        CAST (NULL AS Integer) AS enrollment
FROM teacher-only
UNION ALL
SELECT CAST (NULL AS Varchar(20)) AS name,
        CAST (NULL AS Varchar(20)) AS rank,
        subject, enrollment
FROM course-only ;
```


Some New Features of SQL

- CAST expression
- CASE expression
- Sub-query
- Outer Join
- **Recursion**

Recursion

- If a common table expression uses itself in its definition, this is called recursion. It can calculate a complex recursive inference in one SQL statement.
  FedEmp (name, salary, manager)
- *Find all employees under the management of Hoover and whose salary is more than 100000*

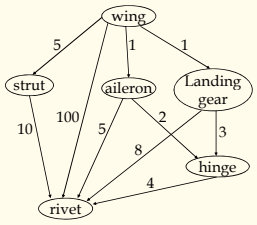
```
WITH agents (name, salary) AS
      ((SELECT name, salary                  --- initial query
        FROM FedEmp
        WHERE manager='Hoover')
       UNION ALL
       (SELECT f.name, f.salary              --- recursive query
        FROM agents AS a, FedEmp AS f
        WHERE f.manager = a.name))
SELECT name                                  --- final query
FROM agents
WHERE salary>100000 ;
```


Recursive Calculation

- A classical "parts searching problem"

Components

Part	Subpart	QTY
wing	strut	5
wing	aileron	1
wing	landing gear	1
wing	rivet	100
strut	rivet	10
aileron	hinge	2
aileron	rivet	5
landing gear	hinge	3
landing gear	rivet	8
hinge	rivet	4



Directed acyclic graph, which assures the recursion can be stopped

Recursive Calculation

- *Find how much rivets are used in one wing?*
- A temporary view is defined to show the list of each subpart's quantity used in a specified part :

```
WITH wingpart (subpart, qty) AS
   ((SELECT subpart, qty              ---initial query
     FROM components
     WHERE part='wing')
    UNION ALL
    (SELECT c.subpart, w.qty*c.qty   ---recursive qry
     FROM wingpart w, components c
     WHERE w.subpart=c.part))
```

wingpart

Subpart	QTY	
strut	5	Used directly
aileron	1	Used directly
landing gear	1	Used directly
rivet	100	Used directly
rivet	50	Used on strut
hinge	2	Used on aileron
rivet	5	Used on aileron
hinge	3	on landing gear
rivet	8	on landing gear
rivet	8	on aileron hinges
rivet	12	on L G hinges

Recursive Calculation

- *Find how much rivets are used in one wing?*

```
WITH wingpart (subpart, qty) AS
      ((SELECT subpart, qty              ---initial query
        FROM components
        WHERE part='wing')
       UNION ALL
       (SELECT c.subpart, w.qty*c.qty   ---recursive qry
        FROM wingpart w, components c
        WHERE w.subpart=c.part))
SELECT sum(qty) AS qty
FROM wingpart
WHERE subpart='rivet' ;
```

- The result is :

qty
183

Recursive Calculation

- *Find all subparts and their total quantity needed to assemble a wing :*

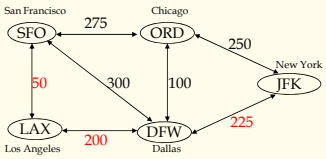
```
WITH wingpart (subpart, qty) AS
      ((SELECT subpart, qty              ---initial query
        FROM components
        WHERE part='wing')
       UNION ALL
        (SELECT c.subpart, w.qty*c.qty ---recursive qry
        FROM wingpart w, components c
        WHERE w.subpart=c.part))
SELECT subpart, sum(qty) AS qty
FROM wingpart
Group BY  subpart ;
```

- The result is :

subpart	qty
strut	5
aileron	1
landing gear	1
hinge	5
rivet	183

Recursive Search

- Typical airline route searching problem
- *Find the lowest total cost route from SFO to JFK*



Flights

FlightNo	Origin	Destination	Cost
HY 120	DFW	JFK	225
HY 130	DFW	LAX	200
HY 140	DFW	ORD	100
HY 150	DFW	SFO	300
HY 210	JFK	DFW	225
HY 240	JFK	ORD	250
HY 310	LAX	DFW	200
HY 350	LAX	SFO	50
HY 410	ORD	DFW	100
HY 420	ORD	JFK	250
HY 450	ORD	SFO	275
HY 510	SFO	DFW	300
HY 530	SFO	LAX	50
HY 540	SFO	ORD	275

Recursive Search

```
WITH trips (destination, route, nsegs, totalcost) AS
    ((SELECT destination, CAST(destination AS varchar(20)), 1, cost
      FROM flights                                  --- initial query
      WHERE origin='SFO')
     UNION ALL
      (SELECT f.destination,                        --- recursive query
              CAST(t.route||','||f.destination AS varchar(20)),
              t.nsegs+1, t.totalcost+f.cost
      FROM trips t, flights f
      WHERE t.destination=f.origin
              AND f.destination<>'SFO'              --- stopping rule 1
              AND f.origin<>'JFK'                   --- stopping rule 2
              AND t.nsegs<=3))                      --- stopping rule 3
SELECT route,  totalcost                            --- final query
FROM trips
WHERE destination='JFK' AND totalcost=              --- lowest cost rule
                          (SELECT min(totalcost)
                           FROM trips
                           WHERE destination='JFK') ;
```


Result

Trips

Destination	Route	Nsegs	Totalcost
DFW	DFW	1	300
ORD	ORD	1	275
LAX	LAX	1	50
JFK	DFW, JFK	2	525
LAX	DFW, LAX	2	500
ORD	DFW, ORD	2	400
DFW	LAX, DFW	2	250
DFW	ORD, DFW	2	375
JFK	ORD, JFK	2	525
DFW	DFW, LAX, DFW	3	700
DFW	DFW, ORD, DFW	3	500
JFK	DFW, ORD, JFK	3	650
LAX	LAX, DFW, LAX	3	450
JFK	LAX, DFW, JFK	3	475
ORD	LAX, DFW, ORD	3	350
LAX	ORD, DFW, LAX	3	575
JFK	ORD, DFW, JFK	3	600
ORD	ORD, DFW, ORD	3	475

Final result

route	totalcost
LAX, DFW, JFK	475

Recursive Search

- *Only change the final query slightly, the least transfer time routes can be found :*

```
… …
SELECT route,  totalcost                            --- final query
FROM trips
WHERE destination='JFK' AND nsegs=                  --- least stop rule
                          (SELECT min(nsegs)
                           FROM trips
                           WHERE destination='JFK') ;
```

Final result

route	totalcost
DFW, JFK	525
ORD, JFK	525

Data Manipulation Language

- Insert
  - Insert a tuple into a table
    - *INSERT INTO EMPLOYEES VALUES ('Smith', 'John', '1980-06-10', 'Los Angles', 16, 45000);*
- Delete
  - Delete tuples fulfill qualifications
    - *DELETE FROM Person WHERE LastName = 'Rasmussen' ;*
- Update
  - Update the attributes' value of tuples fulfill qualifications
    - *UPDATE Person SET Address = 'Zhongshan 23', City = 'Nanjing' WHERE LastName = 'Wilson';*

View in SQL

- General view
  - Virtual tables derived from base tables
  - Logical data independence
  - Security of data
  - Update problems of view
- Temporary view and recursive query
  - WITH
  - RECURSIVE

Update problems of view

- CREATE VIEW YoungSailor AS
  SELECT sid, sname, rating
  FROM Sailors
  WHERE age<26;
- CREATE VIEW Ratingavg AS
  SELECT rating, AVG(age)
  FROM Sailors
  GROUP BY rating;

Embedded SQL

- In order to access database in programs, and take further process to the query results, need to combine SQL and programming language (such as C / C++, etc.)
- Problems should be solved:
  - How to accept SQL statements in programming language
  - How to exchange data and messages between programming language and DBMS
  - The query result of DBMS is a set, how to transfer it to the variables in programming language
  - The data type of DBMS and programming language may not the same exactly.

General Solutions

- Embedded SQL
  - The most basic method. Through pre-compiling, transfer the embedded SQL statements to inner library functions call to access database.
- Programming APIs
  - Offer a set of library functions or DLLs to programmer directly, linking with application program while compiling.
- Class Library
  - Supported after emerging of OOP. Envelope the library functions to access database as a set of class, offering easier way to treat database in programming language.

Usage of Embedded SQL (in C)

- SQL statements can be used in C program directly:
  - Begin with *EXEC SQL*, end with ';'
  - Though *host variables* to transfer information between C and SQL. Host variables should be defined begin with *EXEC SQL*.
  - In SQL statements, should add ':' before host variables to distinguish with SQL's own variable or attributes' name.
  - In host language (such as C), host variables are used as general variables.
  - Can't define host variables as Array or Structure.
  - A special host variable, SQLCA (SQL Communication Area)
    EXEC SQL INCLUDE SQLCA
  - Use SQLCA.SQLCODE to justify the state of result.
  - Use *indicator* (short int) to treat *NULL* in host language.

Example of *host variables* defining

```
EXEC SQL BEGIN DECLARE SECTION;
  char SNO[7];
  char GIVENSNO[7];
  char CNO[6];
  char GIVENCNO[6];
  float GRADE;
  short GRADEI;        /*indicator of GRADE*/
EXEC SQL END DECLARE SECTION;
```

Executable Statements

- CONNECT
  - EXEC SQL CONNECT :uid IDENTIFIED BY :pwd;
- Execute DDL or DML Statements
  - EXEC SQL INSERT INTO SC(SNO,CNO,GRADE)
    VALUES(:SNO, :CNO, :GRADE);
- Execute Query Statements
  - EXEC SQL SELECT GRADE
    INTO :GRADE :GRADEI
    FROM SC
    WHERE SNO=:GIVENSNO AND
    CNO=:GIVENCNO;
- Because {SNO,CNO} is the key of SC, the result of this query has only one tuple. How to treat result if it has a set of tuples?

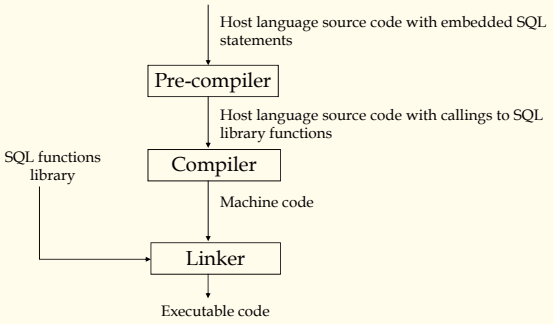
Cursor

1. Define a cursor
   - EXEC SQL DECLARE <*cursor name*> CURSOR FOR
     SELECT …
     FROM …
     WHERE …
2. EXEC SQL OPEN <*cursor name*>
   - Some like open a file
3. Fetch data from cursor
   - EXEC SQL FETCH <*cursor name*>
     INTO :hostvar1, :hostvar2, …;
4. SQLCA.SQLCODE will return 100 when arriving the end of cursor
5. CLOSE CURSOR <*cursor name*>

Example of Query with Cursor

```
        :
EXEC SQL DECLARE C1 CURSOR FOR
        SELECT SNO, GRADE
        FROM SC
        WHERE CNO = :GIVENCNO;
EXEC SQL OPEN C1;
if (SQLCA.SQLCODE<0) exit(1);        /* There is error in query*/
while (1) {
   EXEC SQL FETCH C1 INTO :SNO, :GRADE :GRADEI
   if (SQLCA.SQLCODE==100) break;
   /* treat data fetched from cursor, omitted*/
        :
}
EXEC SQL CLOSE C1;
        :
```


Conceptual Evaluation



Dynamic SQL

- In above embedded SQL, the SQL statements must be written before compiling. But in some applications, the SQL statement can't decided in ahead, they need to be built dynamically while the program running.
- Dynamic SQL is supported in SQL standard and most RDBMS pruducts
  - Dynamic SQL executed directly
  - Dynamic SQL with dynamic parameters
  - Dynamic SQL for query

Dynamic SQL executed directly

- Only used in the execution of non query SQL statements

```
        :
EXEC SQL BEGIN DECLARE SECTION;
char sqlstring[200];
EXEC SQL END DECLARE SECTION;
char cond[150];
strcpy( sqlstring, "DELETE FROM STUDENT WHERE ");
printf(" Enter search condition :");
scanf("%s", cond);
strcat( sqlstring, cond);
EXEC SQL EXECUTE IMMEDIATE :sqlstring;
        :
```

Dynamic SQL with dynamic parameters

- Only used in the execution of non query SQL statements. Use ***place holder*** to realize dynamic parameter in SQL statement. Some like the macro processing method in C.

```
        :
EXEC SQL BEGIN DECLARE SECTION;
char sqlstring[200];
int birth_year;
EXEC SQL END DECLARE SECTION;
strcpy( sqlstring, "DELETE FROM STUDENT WHERE
                    YEAR(BDATE) <= :y; ");
printf(" Enter birth year for delete :");
scanf("%d", &birth_year);
EXEC SQL PREPARE PURGE FROM :sqlstring;
EXEC SQL EXECUTE PURGE USING :birth_year;
        :
```


Dynamic SQL for query

- Used to form query statement dynamically

```
        :
EXEC SQL BEGIN DECLARE SECTION;
char sqlstring[200];
char SNO[7];
float GRADE;
short GRADEI;
char GIVENCNO[6];
EXEC SQL END DECLARE SECTION;
char orderby[150];
strcpy( sqlstring, "SELECT SNO,GRADE FROM SC WHERE CNO= :c ");
printf(" Enter the ORDER BY clause :");
scanf("%s", orderby);
strcat( sqlstring, orderby);
printf(" Enter the course number :");
scanf("%s", GIVENCNO);
EXEC SQL PREPARE query FROM :sqlstring;
EXEC SQL DECLARE grade_cursor CURSOR FOR query;
EXEC SQL OPEN grade_cursor USING :GIVENCNO;
```


Dynamic SQL for query (Cont.)

```
if (SQLCA.SQLCODE<0) exit(1);              /* There is error in query*/
while (1) {
    EXEC SQL FETCH grade_cursor INTO :SNO, :GRADE :GRADEI
    if (SQLCA.SQLCODE==100)      break;
    /* treat data fetched from cursor, omitted*/
        :
}
EXEC SQL CLOSE grade_cursor;
        :
```


Stored Procedure

- Used to improve performance and facilitate users. With it, user can take frequently used database access program as a procedure, and store it in the database after compiling, then call it directly while need.
  - Make user convenient. User can call them directly and don't need code again. They are reusable.
  - Improve performance. The stored procedures have been compiled, so they don't need parsing and query optimization again while being used.
  - Expand function of DBMS. (can write script)

Example of a Stored Procedure

```
EXEC SQL
    CREATE PROCEDURE drop_student
          (IN student_no CHAR(7),
           OUT message CHAR(30))
    BEGIN ATOMIC
          DELETE FROM STUDENT
                WHERE SNO=student_no;
          DELETE FROM SC
                WHERE SNO=student_no;
          SET message=student_no || 'droped';
    END;
EXEC SQL

        :
CALL drop_student(…);        /* call this stored procedure later*/
        :
```