


CSIE30600/CSIEB0290
Database Systems

Lecture 6: More SQL



Outline

- More Complex SQL Retrieval Queries
- Specifying Constraints as Assertions and Actions as Triggers
- Views (Virtual Tables) in SQL
- Schema Change Statements in SQL
- ...

More Complex SQL Retrieval Queries

- Additional features allow users to specify more **complex retrievals** from database:
 - Nested queries
 - Joined tables
 - Outer joins
 - Aggregate functions
 - Grouping

CSIE30600/CSIEB0290 Database Systems

More SQL 3

Comparisons Involving NULL and Three-Valued Logic

- Meanings of **NULL**
 - **Unknown value**
 - **Unavailable or withheld value**
 - **Not applicable attribute**
- Each individual **NULL** value considered to be **different** from every other **NULL** value
- SQL uses a **three-valued logic**:
 - **TRUE, FALSE, and UNKNOWN**

CSIE30600/CSIEB0290 Database Systems

More SQL 4

Comparisons Involving NULL and Three-Valued Logic (cont.)

Table 7.1 Logical Connectives in Three-Valued Logic

(a)	AND	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE
	UNKNOWN	UNKNOWN	FALSE	UNKNOWN
(b)	OR	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	UNKNOWN
	UNKNOWN	TRUE	UNKNOWN	UNKNOWN
(c)	NOT			
	TRUE	FALSE		
	FALSE	TRUE		
	UNKNOWN	UNKNOWN		

CSIE30600/CSIEB0290 Database Systems

More SQL 5

Three Valued Logic

- **Trick:** TRUE = 1; FALSE = 0; UNKNOWN=1/2
 - X and Y = min(X,Y)
 - X or Y = max(X,Y)
 - not X = 1 - X
- The result of any **arithmetic expression** involving **null** is **null**
 - Example: 5 + **null** returns **null**
- Tuples for which the condition evaluates to UNKNOWN are **not** included in the result

CSIE30600/CSIEB0290 Database Systems

More SQL 6

Comparisons Involving NULL and Three-Valued Logic (cont.)

- SQL allows queries that check whether an attribute value is `NULL`
 - **IS** or **IS NOT NULL**

Query 18. Retrieve the names of all employees who do not have supervisors.

```
Q18:  SELECT  Fname, Lname
      FROM    EMPLOYEE
      WHERE   Super_ssn IS NULL;
```

Nested Queries, Tuples, and Set/Multiset Comparisons

- **Nested queries**
 - Complete select-from-where blocks (the *nested query*) within WHERE clause of another query (the *outer query*).
- Comparison operator **IN**
 - Compares value v with a set (or multiset) of values V
 - Evaluates to `TRUE` if v is **one** of the elements in V

Nesting of Queries

- Query: Retrieve the name and address of all employees who work for the 'Research' or 'Sales' department.

Q: `SELECT FNAME, LNAME, ADDRESS`
`FROM EMPLOYEE`
`WHERE DNO IN`
`(SELECT DNUMBER`
`FROM DEPARTMENT`
`WHERE DNAME='Research' OR`
`DNAME='Sales');`

CSIE30600/CSIEB0290 Database Systems

More SQL 9

Nesting of Queries(cont.)

- The **nested query** selects the number of the 'Research' department
- The **outer query** select an EMPLOYEE tuple if its DNO value is in the result of the nested query
- The comparison operator **IN** compares a value **v** with a set (or multi-set) of values **V**, and evaluates to TRUE if **v** is one of the elements in **V**
- In general, we can have several levels of nesting
- A reference to an **unqualified attribute** refers to the relation declared in the **innermost nested query**
- In this example, the nested query is **not correlated** with the outer query

CSIE30600/CSIEB0290 Database Systems

More SQL 10

IN and NOT IN

```
SELECT C1.Number, C1.Name
FROM Customer C1
WHERE C1.CRating IN
      (SELECT C2.CRating
       FROM Customer C2
       WHERE Ccity='Hualien');
```

- <attribute-name A> **IN** (subquery S): tests set membership
 - A is equal to **one** of the values in S
- <attribute-name A> **NOT IN** (subquery S)
 - A is equal to **no** value in S

Nested Queries (cont.)

```
Q4A:  SELECT DISTINCT Pnumber
      FROM PROJECT
      WHERE Pnumber IN
          ( SELECT Pnumber
            FROM PROJECT, DEPARTMENT, EMPLOYEE
            WHERE Dnum=Dnumber AND
                  Mgr_ssn=Ssn AND Lname='Smith' )
      OR
      Pnumber IN
          ( SELECT Pno
            FROM WORKS_ON, EMPLOYEE
            WHERE Essn=Ssn AND Lname='Smith' );
```


Nested Queries (cont.)

- Use **tuples** of values in comparisons
 - Place them within **parentheses**

```
SELECT    DISTINCT Essn
FROM      WORKS_ON
WHERE     (Pno, Hours) IN ( SELECT    Pno, Hours
                           FROM      WORKS_ON
                           WHERE     Essn='123456789' );
```

CSIE30600/CSIEB0290 Database Systems

More SQL 13

Correlated Nested Queries

- If a condition in the WHERE-clause of a *nested query* references an attribute of a relation declared in the *outer query*, the two queries are said to be **correlated**
 - The result of a correlated nested query is different for each tuple (or combination of tuples) of the relation(s) of the outer query
- Query 12: Retrieve the name of each employee who has a dependent with the same first name as the employee.

```
Q12:  SELECT    E.FNAME, E.LNAME
      FROM      EMPLOYEE AS E
      WHERE     E.SSN IN
              (SELECT    ESSN
               FROM      DEPENDENT
               WHERE     ESSN=E.SSN AND
                        E.FNAME=DEPENDENT_NAME);
```

CSIE30600/CSIEB0290 Database Systems

More SQL 14

Correlated Nested Queries (cont.)

- In Q₁₂, the nested query has a different result in the outer query
- A query written with nested SELECT... FROM... WHERE... blocks and using the = or IN comparison operators can **always** be expressed as a single block query. For example, Q₁₂ may be written as in Q_{12A}

```
Q12A: SELECT E.FNAME, E.LNAME
       FROM EMPLOYEE E, DEPENDENT D
       WHERE E.SSN=D.ESSN AND
             E.FNAME=D.DEPENDENT_NAME;
```

Correlated Subqueries: Scoping

- An attribute in a subquery belongs to one of the tuple variables corresponding to the **closest** relation
 - In general, an attribute in a subquery belongs to one of the tuple variables in that subquery's FROM clause
 - If not, look at the **immediately surrounding** subquery, then to the one surrounding that, and so on.

Nested Queries

- The FROM clause takes a relation, but *results from SQL queries are themselves relations, so we can use them in the FROM clause, too!*

```
SELECT  (N.CRating+1) AS CIncrRating
FROM    (SELECT * FROM Customer
        WHERE CRating = 0) AS N
WHERE   N.CBalance = 0;
```

- This can often be a more elegant way to write a query, but will be slower. Why?

CSIE30600/CSIEB0290 Database Systems

More SQL 17

The EXISTS and UNIQUE Functions in SQL

- **EXISTS** function
 - Check whether the result of a correlated nested query is empty or not
- **EXISTS** and **NOT EXISTS**
 - Typically used in conjunction with a correlated nested query
- SQL function **UNIQUE (Q)**
 - Returns TRUE if there are no duplicate tuples in the result of query Q

CSIE30600/CSIEB0290 Database Systems

More SQL 18

The EXISTS Function

- **EXISTS** is used to check whether the result of a correlated nested query is empty (contains no tuples) or not
- We can formulate Query 12 in an alternative form that uses EXISTS as Q12B (next slide)

CSIE30600/CSIEB0290 Database Systems

More SQL 19

The EXISTS Function(cont.)

- Query 12: Retrieve the name of each employee who has a dependent with the same first name as the employee.

```
Q12B: SELECT      FNAME, LNAME
        FROM        EMPLOYEE
        WHERE       EXISTS
                   ( SELECT *
                     FROM  DEPENDENT
                     WHERE SSN=ESSN AND
                           FNAME=DEPENDENT_NAME);
```

CSIE30600/CSIEB0290 Database Systems

More SQL 20

NOT EXISTS

- Query 6: Retrieve the names of employees who have no dependents.

```
Q6:  SELECT  FNAME, LNAME
      FROM    EMPLOYEE
      WHERE   NOT EXISTS
            ( SELECT  *
              FROM    DEPENDENT
              WHERE   SSN=ESSN );
```

- In Q6, the correlated nested query retrieves all DEPENDENT tuples related to an EMPLOYEE tuple. If *none exist*, the EMPLOYEE tuple is selected
 - EXISTS is necessary for the expressive power of SQL

Explicit Sets

- It is also possible to use an **explicit (enumerated) set of values** in the WHERE-clause rather than a nested query
- Query 13: Retrieve the social security numbers of all employees who work on project number 1, 2, or 3.

```
Q13:  SELECT  DISTINCT ESSN
      FROM    WORKS_ON
      WHERE   PNO IN (1, 2, 3);
```

Set Comparison

- Find all branches that have greater assets than **some** branch located in Brooklyn.

```
select distinct T.branch_name
from branch as T, branch as S
where T.assets > S.assets and
      S.branch_city = 'Brooklyn';
```

- Same query using **> SOME (ANY)** clause.

```
select branch_name
from branch
where assets > SOME
      (select assets
       from branch
       where branch_city = 'Brooklyn');
```

CSIE30600/CSIEB0290 Database Systems

More SQL 23

Definition of SOME Clause

- $F \langle \text{comp} \rangle \text{SOME } r \Leftrightarrow \exists t \in r \text{ such that } (F \langle \text{comp} \rangle t)$ where $\langle \text{comp} \rangle$ can be: $<$, \leq , $>$, $=$, \neq

$(5 < \text{some } \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{true}$ (read: 5 < some tuple in the relation)

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

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

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

$(= \text{some}) \equiv \text{in}$
However, $(\neq \text{some}) \neq \text{not in}$

CSIE30600/CSIEB0290 Database Systems

More SQL 24

Query with ALL

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

```
select branch_name
from branch
where assets > ALL
(select assets
from branch
where branch_city = 'Brooklyn');
```

CSIE30600/CSIEB0290 Database Systems

More SQL 25

Definition of ALL Clause

- $F \langle \text{comp} \rangle \text{ALL } r \Leftrightarrow \forall t \in r (F \langle \text{comp} \rangle t)$

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

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

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

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

$(\neq \text{all}) \equiv \text{not in}$
However, $(= \text{all}) \neq \text{in}$

CSIE30600/CSIEB0290 Database Systems

More SQL 26

Joined Relations

- Can specify a "**joined relation**" in the FROM-clause
 - Looks like any other relation but is the result of a join
 - Allows the user to specify different types of joins (regular "theta" **JOIN**, **NATURAL JOIN**, **LEFT OUTER JOIN**, **RIGHT OUTER JOIN**, **CROSS JOIN**, etc)

```
Q1A:  SELECT  Fname, Lname, Address
      FROM    (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
      WHERE   Dname='Research';
```

Inner JOIN

- Default type of join in a joined table
- Tuple is included in the result only if a matching tuple exists in the other relation
- If we want to keep those tuples that **do not match** the condition, we need to use **outer join**.

Why Outer Join?

- Consider the following tables and query

`Student(sid, name, address)`

`Spouse(sid, name)`, sid references Student.sid

List the names of ALL students and their spouses, if they have one.

```
SELECT Student.name, Spouse.name
```

```
FROM Student, Spouse
```

```
WHERE Student.sid=Spouse.sid
```

- Does this SQL query do the job?
 - No! Students without spouses will ***not*** be listed.

Outer Join

- An extension of the join operation that **avoids loss of information**.
- Computes the join and then adds tuples from one relation that do not match tuples in the other relation to the result of the join.
- Uses **null** values to pad dangling tuples

LEFT OUTER JOIN

- **INNER JOIN** on C.SalespersonNum = S.Number gives us: “smith” with “johnson” and “jones” with “johnson”
- **LEFT OUTER JOIN** on C.SalespersonNum = S.Number gives us:

INNER JOIN plus “wei” with “<null>” salesperson

- Lists all customers, and their salesperson if any

Customer						
Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	<null>

Salesperson				
Number	Name	Address	Office	
101	johnson	aaa	23	
102	miller	bbb	26	

CSIE30600/CSIEB0290 Database Systems

More SQL 31

LEFT OUTER JOIN: Example

- Examples:
 - Q8:

```
SELECT  E.FNAME, E.LNAME, S.FNAME, S.LNAME
FROM    EMPLOYEE E S
WHERE   E.SUPERSSN=S.SSN
```
- Compare the result with the following query:
 - Q8a:

```
SELECT  E.FNAME, E.LNAME, S.FNAME, S.LNAME
FROM    (EMPLOYEE E LEFT OUTER JOIN
EMPLOYEE S ON E.SUPERSSN=S.SSN)
```

CSIE30600/CSIEB0290 Database Systems

More SQL 32

RIGHT OUTER JOIN

- **INNER JOIN** on C.SalespersonNum = S.Number gives us:
“smith” with “johnson” and “jones” with “johnson”
- **RIGHT OUTER JOIN** on C.SalespersonNum = S.Number gives:
INNER JOIN plus “<null>” customer with “miller”
 - Lists customers that have a salesperson, and salespersons that do not have a customer

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	<null>

Salesperson

Number	Name	Address	Office
101	johnson	aaa	23
102	miller	bbb	26

CSIE30600/CSIEB0290 Database Systems

More SQL 33

FULL OUTER JOIN

- **FULL OUTER JOIN = LEFT OUTER JOIN \cup RIGHT OUTER JOIN**

FULL OUTER JOIN on C.SalespersonNum = S.Number gives us:

INNER JOIN

plus “wei” with “<null>” salesperson

plus “<null>” customer with “miller”

- Lists all customer-salesperson pairs, and customers that do not have a salesperson, and salespersons that do not have a customer
- NOTE: You could also have NATURAL <left, right, full> OUTER JOIN

CSIE30600/CSIEB0290 Database Systems

More SQL 34

CROSS JOIN

- A “**CROSS JOIN**” is simply a **cross product**

```
SELECT *
FROM Customer CROSS JOIN Salesperson;
```
- *How would you write this query without the “**CROSS JOIN**” operator?*

```
SELECT *
FROM Customer, Salesperson;
```

More Join Examples

- Examples:

```
Q1: SELECT      FNAME, LNAME, ADDRESS
      FROM      EMPLOYEE, DEPARTMENT
      WHERE     DNAME='Research' AND DNUMBER=DNO
```
- could be written as:

```
Q1: SELECT      FNAME, LNAME, ADDRESS
      FROM      (EMPLOYEE JOIN DEPARTMENT
                ON DNUMBER=DNO)
      WHERE     DNAME='Research'
```
- or as:

```
Q1: SELECT      FNAME, LNAME, ADDRESS
      FROM      (EMPLOYEE NATURAL JOIN DEPARTMENT
                AS DEPT(DNAME, DNO, MSSN, MSDATE))
      WHERE     DNAME='Research'
```


Multiple Joins

- Another Example: Q₂ could be written as follows; this illustrates multiple joins in the joined tables

```
Q2: SELECT  PNUMBER, DNUM, LNAME,
            BDATE, ADDRESS
        FROM    ((PROJECT JOIN DEPARTMENT
                ON DNUM=DNUMBER)
                JOIN EMPLOYEE ON
                MGRSSN=SSN)
        WHERE   PLOCATION='Stafford'
```

CSIE30600/CSIEB0290 Database Systems

More SQL 37

Aggregate Functions

- Used to summarize information from multiple tuples into a single-tuple summary
- Include **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**
- Query: Find the maximum salary, the minimum salary, and the average salary among all employees.

```
Q: SELECT  MAX(SALARY), MIN(SALARY),
            AVG(SALARY)
        FROM    EMPLOYEE;
```

- Some SQL implementations *may not allow more than one function* in the SELECT-clause

CSIE30600/CSIEB0290 Database Systems

More SQL 38

Aggregate Functions(contd.)

Query 20. Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q20:  SELECT  SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
        FROM    (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
        WHERE   Dname='Research';
```

Queries 21 and 22. Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

```
Q21:  SELECT  COUNT (*)
        FROM    EMPLOYEE;
```

```
Q22:  SELECT  COUNT (*)
        FROM    EMPLOYEE, DEPARTMENT
        WHERE   DNO=DNUMBER AND DNAME='Research';
```

CSIE30600/CSIEB0290 Database Systems

More SQL 39

Challenge Questions

- What is the implication of using DISTINCT when computing the **SUM** or **AVG** of an attribute?

SUM(DISTINCT Balance) or **AVG(DISTINCT Balance)**

- What is the implication of using DISTINCT when computing the **MIN** or **MAX** of an attribute?

MIN(DISTINCT Balance) or **MAX(DISTINCT Balance)**

CSIE30600/CSIEB0290 Database Systems

More SQL 40

Aggregates and NULLs

- General rule: aggregates **ignore** NULL values
 - $\text{Avg}(1,2,3,\text{NULL},4) = \text{Avg}(1,2,3,4)$
 - $\text{Count}(1,2,3,\text{NULL},4) = \text{Count}(1,2,3,4)$
- But...
 - **Count(*)** returns the total number of tuples, regardless whether they contain NULLs or not

CSIE30600/CSIEB0290 Database Systems

More SQL 41

Grouping

- In many cases, we want to apply the aggregate functions to *subgroups of tuples* in a relation
- Each subgroup of tuples consists of the set of tuples that have the **same value** for the **grouping attribute(s)**
- The function is applied to each subgroup **independently**
- SQL has a **GROUP BY-clause** for specifying the grouping attributes, which **must also appear in the SELECT-clause**

CSIE30600/CSIEB0290 Database Systems

More SQL 42

Grouping (cont.)

- Query: For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
Q: SELECT      DNO, COUNT(*), AVG(SALARY)
   FROM        EMPLOYEE
   GROUP BY    DNO;
```

- In here, the EMPLOYEE tuples are divided into groups-
 - Each group having the same value for the grouping attribute **DNO**
- The COUNT and AVG functions are applied to each such group of tuples separately
- The SELECT-clause includes only the grouping attribute and the functions to be applied on each group of tuples
- A join condition can be used in conjunction with grouping

CSIE30600/CSIEB0290 Database Systems

More SQL 43

Grouping (cont.)

- Query: For each project, retrieve the project number, project name, and the number of employees who work on that project.

```
Q: SELECT      PNUMBER, PNAME, COUNT (*)
   FROM        PROJECT, WORKS_ON
   WHERE       PNUMBER=PNO
   GROUP BY    PNUMBER, PNAME;
```

- In this case, the grouping and functions are applied after the joining of the two relations

CSIE30600/CSIEB0290 Database Systems

More SQL 44

The HAVING-Clause

- Sometimes we want to retrieve the values of these functions for only those *groups that satisfy certain conditions*
- The **HAVING-clause** is used for specifying a selection condition **on groups** (rather than on individual tuples)

CSIE30600/CSIEB0290 Database Systems

More SQL 45

The HAVING-Clause (contd.)

- Query: For each project *on which more than two employees work*, retrieve the project number, project name, and the number of employees who work on that project.

```
Q: SELECT  PNUMBER, PNAME, COUNT(*)
   FROM    PROJECT, WORKS_ON
   WHERE   PNUMBER=PNO
   GROUP BY PNUMBER, PNAME
   HAVING  COUNT(*) > 2;
```

CSIE30600/CSIEB0290 Database Systems

More SQL 46

The HAVING-Clause (contd.)

Query 28. For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than \$40,000.

```
Q28:  SELECT  Dnumber, COUNT (*)
      FROM    DEPARTMENT, EMPLOYEE
      WHERE   Dnumber=Dno AND Salary>40000 AND Dnumber IN
            ( SELECT  Dno
              FROM    EMPLOYEE
              GROUP BY Dno
              HAVING  COUNT (*) > 5)
```

CSIE30600/CSIEB0290 Database Systems

More SQL 47

GROUP BY and NULLS (1)

- Aggregates ignore NULLs
- On the other hand, NULL is treated as an ordinary value in a grouped attribute
- If there are NULLs in the Salesperson column (below), a group will be returned for the NULL value (next slide)

Customer

Number	Name	Address	CRating	CAmount	CBalance	SalespersonNum
1	smith	xxx	5	1,000	1,000	101
2	jones	yyy	7	5,000	4,000	101
3	wei	zzz	10	10,000	10,000	NULL

CSIE30600/CSIEB0290 Database Systems

More SQL 48

GROUP BY and NULLS (2)

```
SELECT SalespersonNum, Count(*) AS T
FROM Customer
GROUP BY SalespersonNum;
```

Answer

SalespersonNum	T
NULL	1
101	2

CSIE30600/CSIEB0290 Database Systems

More SQL 49

GROUP BY, HAVING: Note

- The only attributes that can appear in a “grouped” query answer are **aggregate operators** (that are applied to the group) or the **grouping attribute(s)**.

```
SELECT SalespersonNum, COUNT(*)
FROM Customer
GROUP BY SalespersonNum;
```

```
SELECT SalespersonNum
FROM Customer
GROUP BY SalespersonNum
HAVING Count(*) > 10;
```

Incorrect!

```
SELECT C.Name,
SalespersonNum,
COUNT(*)
FROM Customer C
GROUP BY
SalespersonNum;
```

CSIE30600/CSIEB0290 Database Systems

More SQL 50

Summary of SQL Queries

- A query in SQL can consist of up to six clauses, but only the first two, SELECT and FROM, are mandatory. The clauses are specified in the following order:

```
SELECT <attribute and function list>  
FROM <table list>  
[WHERE <condition>  
[GROUP BY <grouping attribute(s)>  
[HAVING <group condition>  
[ORDER BY <attribute list>;
```

Summary of SQL Queries (cont.)

- The SELECT-clause lists the attributes or functions to be retrieved
- The FROM-clause specifies all relations (or aliases) needed in the query but not those needed in nested queries
- The WHERE-clause specifies the conditions for selection and join of tuples from the relations specified in the FROM-clause
- GROUP BY specifies grouping attributes

Summary of SQL Queries (cont.)

- HAVING specifies a condition for selection of groups
- ORDER BY specifies an order for displaying the result of a query
- A query is evaluated by first applying the WHERE-clause, then GROUP BY and HAVING, and finally the SELECT-clause

CSIE30600/CSIEB0290 Database Systems

More SQL 53

Specifying Complex Update

- Example: Give all employees in the 'Research' department a 10% raise in salary.

```

U6: UPDATE EMPLOYEE
     SET    SALARY = SALARY *1.1
     WHERE DNO IN
           ( SELECT DNUMBER
             FROM DEPARTMENT
             WHERE DNAME='Research');

```

- In this request, the modified SALARY value depends on the original SALARY value in each tuple
 - The reference to the SALARY attribute on the right of = refers to the old SALARY value before modification
 - The reference to the SALARY attribute on the left of = refers to the new SALARY value after modification

CSIE30600/CSIEB0290 Database Systems

More SQL 54

CASE Statement for Conditional Updates

- Increase all accounts with balances over \$10,000 by 6%, all other accounts receive 5%.

update *account*

set *balance* = **case**

when *balance* <= 10000 **then** *balance* * 1.05

when *balance* >= 20000 **then** *balance* * 1.07

else *balance* * 1.06

end;

Derived Relations

- SQL allows a subquery expression to be used in **from** clause
- Find the average account balance of those branches where the average account balance is greater than \$1200.

```
select branch_name, avg_balance
from (select branch_name, avg (balance)
      from account
      group by branch_name )
as branch_avg ( branch_name, avg_balance )
where avg_balance > 1200;
```

Note that we do not need to use the **having** clause, since we compute the temporary (view) relation *branch_avg* in the **from** clause, and the attributes of *branch_avg* can be used directly in the **where** clause.

WITH Clause

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

- Find all accounts with the maximum balance

```
with max_balance (value) as  
    select max (balance)  
    from account  
select account_number  
from account, max_balance  
where account.balance = max_balance.value;
```

Complex Query using WITH Clause

- Find all branches where the total account deposit is greater than the average of the total account deposits at all branches.

```
with branch_total (branch_name, value) as  
    select branch_name, sum (balance)  
    from account  
    group by branch_name  
with branch_total_avg (value) as  
    select avg (value)  
    from branch_total  
select branch_name  
from branch_total, branch_total_avg  
where branch_total.value >= branch_total_avg.value;
```

Specifying Constraints as Assertions and Actions as Triggers

- **CREATE ASSERTION**

- Specify additional types of **constraints** outside scope of built-in relational model constraints

- **CREATE TRIGGER**

- Specify automatic **actions** that database system will perform when certain events and conditions occur

Assertions in SQL

- **CREATE ASSERTION**

- Specify a query that selects any tuples that violate the desired condition
- Use only in cases where it is not possible to use CHECK on attributes and domains

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT *
                    FROM   EMPLOYEE E, EMPLOYEE M,
                          DEPARTMENT D
                    WHERE  E.Salary>M.Salary
                          AND E.Dno=D.Dnumber
                          AND D.Mgr_ssn=M.Ssn ) );
```

Triggers in SQL

- **CREATE TRIGGER** statement
 - Used to monitor the database
- Typical trigger has three components:
 - **Event(s)**
 - **Condition**
 - **Action**

Views (Virtual Tables)

- In some cases, it is not desirable for all users to see the entire logical model (ie, all the actual relations.)
- Consider a person who needs to know a customer's loan number but has no need to see the loan amount. This person should see a relation described, in SQL, by
(`select customer_name, loan_number`
`from borrower, loan`
`where borrower.loan_number = loan.loan_number`)
- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a “**virtual relation**” is called a **view**.

View Definition

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

create view v as < query expression >

where <query expression> is any legal SQL expression. The view name is represented by v.

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is **not** the same as creating a new relation by evaluating the query expression. Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

CREATE VIEW

- View always up-to-date
 - Responsibility of the DBMS and not the user
- **DROP VIEW** command
 - Dispose of a view

```
V1:  CREATE VIEW  WORKS_ON1
      AS SELECT   Fname, Lname, Pname, Hours
      FROM       EMPLOYEE, PROJECT, WORKS_ON
      WHERE      Ssn=Essn AND Pno=Pnumber;

V2:  CREATE VIEW  DEPT_INFO(Dept_name, No_of_emps, Total_sal)
      AS SELECT   Dname, COUNT (*), SUM (Salary)
      FROM       DEPARTMENT, EMPLOYEE
      WHERE      Dnumber=Dno
      GROUP BY   Dname;
```

More View Examples

- A view consisting of branches and their customers
create view all_customer as
*(select branch_name, customer_name
from depositor, account
where depositor.account_number =
account.account_number)
union
(select branch_name, customer_name
from borrower, loan
where borrower.loan_number = loan.loan_number);*
- Find all customers of the Perryridge branch
**select customer_name
from all_customer
where branch_name = 'Perryridge';**

CSIE30600/CSIEB0290 Database Systems

More SQL 65

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view v_1 is said to **depend directly** on a view v_2 if v_2 is used in the expression defining v_1
- A view v_1 is said to **depend on** view v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view v is said to be **recursive** if it depends on itself.

CSIE30600/CSIEB0290 Database Systems

More SQL 66

View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:
repeat
Find any view v_i in e_1
Replace the view v_i by the expression defining v_i
until no more views are present in e_1
- As long as the view definitions are not recursive, this loop will terminate

CSIE30600/CSIEB0290 Database Systems

More SQL 67

View Implementation, View Update, and Inline Views

- Complex problem of efficiently implementing a view for querying
- **Query modification** approach
 - Modify view query into a query on underlying base tables
 - Disadvantage: inefficient for views defined via complex queries that are time-consuming to execute

CSIE30600/CSIEB0290 Database Systems

More SQL 68

View Implementation

- **View materialization approach**
 - Physically create a temporary view table when the view is first queried
 - Keep that table on the assumption that other queries on the view will follow
 - Requires efficient strategy for automatically updating the view table when the base tables are updated

View Implementation (cont'd.)

- **Incremental update strategies**
 - DBMS determines what new tuples must be inserted, deleted, or modified in a materialized view table

View Update and Inline Views

- Update on a view defined on a single table without any aggregate functions
 - Can be mapped to an update on underlying base table
- View involving joins
 - Often not possible for DBMS to determine which of the updates is intended

Schema Change Statements

- **Schema evolution commands**
 - Can be done while the database is **operational**
 - Does **not** require recompilation of the database schema

The DROP Command

- **DROP** command
 - Used to drop named schema elements, such as tables, domains, or constraint
- Drop behavior options:
 - **CASCADE** and **RESTRICT**
- Example:
 - `DROP SCHEMA COMPANY CASCADE;`

The ALTER Command

- **Alter table actions** include:
 - Adding or dropping a column (attribute)
 - Changing a column definition
 - Adding or dropping table constraints
- Example:
 - `ALTER TABLE COMPANY.EMPLOYEE ADD COLUMN Job VARCHAR(12);`
- To drop a column
 - Choose either `CASCADE` or `RESTRICT`

The ALTER Command (cont'd.)

- Change constraints specified on a table
 - Add or drop a named constraint

```
ALTER TABLE COMPANY.EMPLOYEE  
DROP CONSTRAINT EMPSUPERFK CASCADE;
```

SQL Benefits

- Declarative languages: program is a prescription for *what* data is to be retrieved, rather than a *procedure* describing *how* to retrieve the data
- When we write an SQL select query, we do not make any assumptions about the **order of evaluation**
- *Can be automatically optimized!*
 - Decision about order and evaluation plan is left to the optimizer
 - Optimizer has the resources to make sophisticated decisions

SQL Limitations

- Not flexible enough for some applications
 - Some queries cannot be expressed in SQL
 - Non-declarative actions can't be done from SQL, e.g., printing a report, interacting with user/GUI
 - SQL queries may be just one small component of complex applications
- Hard to program for performance!
- Trade-off: *automatic optimization of queries expressed in powerful languages is hard*

CSIE30600/CSIEB0290 Database Systems

More SQL 77

Limitations: Set Functions

- Set functions: sum, avg, max, min and count
- What about **median**
 - Given a sequence of numbers a_1, \dots, a_n
 - Median is the value a_k s.t. $k = \text{FLOOR}((n+1)/2)$
- Can't write
 - `SELECT median(amount) FROM Account`

CSIE30600/CSIEB0290 Database Systems

More SQL 78

Limitations: Transitive Closure

- Employee manages Employee
- Find all employees managed by Mary

Manager	Emp
---------	-----

Null	Mary
------	------

Mary	John
------	------

Mary	Jane
------	------

John	Mark
------	------

Mark	Susan
------	-------

- SQL:1999 added a **WITH RECURSIVE** construct to compute transitive closure. (not yet supported by many DBMS)

Assignment 3

- Textbook exercises:
 - Exercises: 3.11, 3.12, 3.14, 3.15, 3.16
- Due date: **Nov 11, 2020**