INFS7901
Database Principles

Structured Query Language (SQL)

Rocky Chen

Some Quick Notes

- We will keep practicing SQL in this week's tutorial
- Project part 1 will due 3:00pm 6 Apr
- RiPPLE round 2 will due 3:00pm 4 Apr

Views

Null Values and Joins

Constraints and Triggers

Motivating Example for Use of Views

- Find those states for which their average college enrollment is the minimum over all states.

College(cName, state, enrollment)
Student(sID, sName, GPA, sizeHS)
Apply(sID, cName, major, decision)

```
SELECT state, avg(enrollment)
FROM College
GROUP BY state
HAVING min(avg(enrollment))
```

Wrong, cannot use nested aggregation

- One solution would be to use subquery in the FROM Clause.

```
SELECT  Temp.state, Temp.average
FROM(SELECT state, AVG(enrollment) as average
        FROM  College
        GROUP BY state) AS Temp
WHERE Temp.average in (SELECT  MIN(Temp.average) FROM  Temp)
```

Hideously ugly
Not supported in all systems

- A Better alternative is to use views

What Are Views?

- A View is a single table that is derived from other tables, which could be base tables or previously defined views
- Views can be
  - Virtual tables - that do not physically exist on disk.
  - Materialized - by physically creating the view table. These must be updated when the base tables are updated

- We can think of a virtual view as a way of specifying a table that we need to reference frequently, even though it does not physically exist.

Benefits of Using Views

- **Simplification**: View can hide the complexity of underlying tables to the end-users.
- **Security**: Views can hide columns containing sensitive data from certain groups of users.
- **Computed columns**: Views can create computed columns, which are computed on the fly.
- **Logical Data Independence**: Users and user's programs that access the database are immune from changes in the logical structure of the database.

Defining and Using Views

```
CREATE VIEW <view name>
      (<column name> {, <column name>}) AS
       <select statement> ;
```

- Example: Suppose we have the following tables:
  - Course (Course#, title,dept)
  - Enrolled (Course#, sid, mark)

```
CREATE  VIEW  CourseWithFails(dept, course#, mark) AS
          SELECT   C.dept, C.course#, mark
          FROM     Course C, Enrolled E
          WHERE    C.course# = E.course# AND mark<50
```

- This view gives the dept, course#, and marks for those courses where someone failed

Views and Security

- Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).

```
Course(Course#,title,dept)
Enrolled(Course#,sid,mark)
VIEW  CourseWithFails(dept, course#, mark)
```

- Given CourseWithFails, but not Course or Enrolled, we can find the course in which some students failed, but we can't find the students who failed.

View Updates

- View updates must occur at the base tables.
  - Ambiguous
  - Difficult

Course(Course#,title,dept)
Enrolled(Course#,sid,mark)
VIEW CourseWithFails(dept, course#, mark)

- DBMS's restrict view updates only to some simple views on single tables (called updatable views)

Example: UQ has one table for students. Should the CS Department be able to update CS students' info? Yes, Biology students? NO
Create a view for CS to only be able to update CS students.

Dropping Views

```
DROP VIEW [IF EXISTS]
view_name [, view_name] ...
 [RESTRICT | CASCADE]
```

- Dropping a view does not affect any tuples of the underlying relation.

- DROP TABLE command has options to prevent a table from being dropped if views are defined on it:
  - RESTRICT : drops the table, unless there is a view on it
  - CASCADE: drops the table, and recursively drops any view referencing it

The Beauty of Views

- Find those states for which their average college enrollment is the minimum over all states.

```
SELECT  Temp.state, Temp.average
FROM(SELECT state, AVG(enrollment) as average
        FROM  College
        GROUP BY state) AS Temp
WHERE Temp.average in (SELECT  MIN(Temp.average) FROM  Temp)
```

Hideously ugly
Not supported
in all systems

```
Create View Temp(state, average) as
            SELECT state, AVG(enrollment) AS average
            FROM College
            GROUP BY  state;

Select state, average
From Temp
WHERE average = (SELECT MIN(average) from Temp)
```

state	average
CA	25500.0000
MA	10000.0000
NY	21000.0000

state	average
MA	10000.0000

Clicker Question on Views

Consider the following table and SQL queries:

```
CREATE VIEW V AS
    SELECT a+b AS d, c
    FROM R;
```

```
SELECT d, SUM(c)
FROM V
GROUP BY d
HAVING COUNT(*) <> 1;
```

a	b	c
1	1	3
1	2	3
2	1	4
2	3	5
2	4	1
3	2	4
3	3	6

Identify, from the list below, a tuple in the result of the query:

A. (2,3)

B. (3,12)

C. (5,9)

D. All are correct

E. None are correct

Clicker Question on Views

Consider the following table and SQL queries:

```
CREATE VIEW V AS
      SELECT a+b AS d, c
      FROM R;
```

```
SELECT d, SUM(c)
FROM V
GROUP BY d
HAVING COUNT(*) <> 1;
```

a	b	c
1	1	3
1	2	3
2	1	4
2	3	5
2	4	1
3	2	4
3	3	6

V

d	c
2	3
3	3
3	4
5	5
6	1
5	4
6	6

d	Sum(C)
3	7
5	9
6	7

Identify, from the list below, a tuple in the result of the query:

A. (2,3) Wrong. "Count"
B. (3,12)
C. (5,9) Correct
D. All are correct
E. None are correct

Clickerview.sql

Views

Null Values and Joins

Constraints and Triggers

NULL Values

- A value of NULL indicates that the value is unknown.
- The predicate **IS NULL** (**IS NOT NULL**) can be used to check for null values.
- Example: Find all student names whose age is not known.

```
SELECT name
FROM Student
WHERE age IS NULL
```

Operations on NULL Values

- NULL requires a 3-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*, (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown) = unknown*
  - AND: *(true* **and** *unknown) = unknown, (false* **and** *unknown) = false,*
    (unknown **and** *unknown) = unknown*
  - NOT*: (***not** *unknown) = unknown*
- Comparisons between two null values, or between a NULL and any other value, return unknown because the value of each NULL is unknown.
  - *E.g. 5 < null or null <> null or null = null*
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.

```
select count(*)
from class
```

```
select count(fid)
from class
```

Clicker NULL Query

Determine the result of:

```
SELECT COUNT(*), COUNT(Runs)
FROM Scores
WHERE Team = 'Carp'
```

Which of the following is in the result?

A. (1,0)
B. (2,0)
C. (1,NULL)
D. All of the above
E. None of the above

Scores:

Team	Day	Opponent	Runs
Dragons	Sun	Swallows	4
Tigers	Sun	Bay Stars	9
Carp	Sun	NULL	NULL
Swallows	Sun	Dragons	7
Bay Stars	Sun	Tigers	2
Giants	Sun	NULL	NULL
Dragons	Mon	Carp	NULL
Tigers	Mon	NULL	NULL
Carp	Mon	Dragons	NULL
Swallows	Mon	Giants	0
Bay Stars	Mon	NULL	NULL
Giants	Mon	Swallows	5

clickernull.sql

Clicker NULL Query

Determine the result of:

```
SELECT COUNT(*), COUNT(Runs)
FROM Scores
WHERE Team = 'Carp'
```

Which of the following is in the result?

A. (1,0)
B. (2,0) Correct
C. (1,NULL)
D. All of the above
E. None of the above

Scores:

Team	Day	Opponent	Runs
Dragons	Sun	Swallows	4
Tigers	Sun	Bay Stars	9
Carp	Sun	NULL	NULL
Swallows	Sun	Dragons	7
Bay Stars	Sun	Tigers	2
Giants	Sun	NULL	NULL
Dragons	Mon	Carp	NULL
Tigers	Mon	NULL	NULL
Carp	Mon	Dragons	NULL
Swallows	Mon	Giants	0
Bay Stars	Mon	NULL	NULL
Giants	Mon	Swallows	5

Joins in SQL

- A Join used to combine related tuples from two relations into a single tuple in a new (result) relation.
- Join operation is needed for organizing a search space of data. This is needed when information is contained in more than one relation.
- Join relations are specified in the FROM Clause. When two relations are combined for a search, we need to know how the relations are combined.
- Based on Cartesian Product (denoted as X) There are three types of Join operations:
  - Theta-Join
  - Equi-Join
  - Natural Join

Cartesian Product in SQL

- Every row in R1 is matched with every row in R2 to form tuples in the result relation. The schema of the result relation contains all the columns from R1 and all the columns from R2

MovieStar

StarID	Name	Gender
1	Harrison Ford	Male
2	Vivian Leigh	Female
3	Judy Garland	Female

StarsIn

MovieID	StarID	Character
1	1	Han Solo
4	1	Indiana Jones
2	2	Scarlett O'Hara
3	3	Dorothy Gale

```
SELECT *FROM
MovieStar, StarsIn
```

For |R1| = m, |R2| = n, the |R1 X R2| = m * n

MovieStar x StarsIn

StarID1	Name	Gender	MovieID	StarID2	Character
1	Harrison Ford	Male	1	1	Han Solo
2	Vivian Leigh	Female	1	1	Han Solo
3	Judy Garland	Female	1	1	Han Solo
1	Harrison Ford	Male	4	1	Indiana Jones
…	…	…	…	…	…

Theta-Join in SQL

- Theta-join is the most general type of join, which allows several logical operators {=, ≠, <, ≤, >, ≥}.

- Example: For each student, list the students who are older than him/her (i.e., the first student).

```
SELECT A. sName, B. sName
FROM Student A, Student B
WHERE  A.age < B.age
```

Equi-Join and Natural Join

- *Equi-Join*: A special case of Theta join where condition contains only ***equalities.***
- The SQL NATURAL JOIN is a type of Equi-Join and is structured in such a way that, **columns with same name of associated tables will appear once only**.
- Natural Join : Guidelines
  - The associated tables have one or more pairs of identically named columns.
  - The columns must be the same data type.

Natural join of tables with no pairs of identically named columns will return the cross product of the two tables.

Natural Join Examples

MovieStar

StarID	Name	Gender
1	Harrison Ford	Male
2	Vivian Leigh	Female
3	Judy Garland	Female

StarsIn

MovieID	StarID	Character
1	1	Han Solo
4	1	Indiana Jones
2	2	Scarlett O'Hara
3	3	Dorothy Gale

```
Select *
From MovieStar natural join StarsIN
```

StarID	Name	Gender	MovieID	Character
1	Harrison Ford	Male	1	Han Solo
1	Harrison Ford	Male	4	Indiana Jones
3	Judy Garland	Female	3	Dorothy Gale
2	Vivian Leigh	Female	2	Scarlett O'Hara

Inner and Outer Joins

- **Inner Join**

  This is the default join, in which a tuple is included in the result relation, only if matching tuples exist in both relations.

- **Outer Join**

  Outer joins can include the tuples that do not satisfy the join condition, i.e, a matching tuple does not exist, which is indicated by a NULL value

  – Full Outer Join: Includes all rows from both tables.
  – Left Outer Join: Includes all rows from first table.
  – Right Outer Join: Includes all rows from second table .

SELECT <attribute list>
FROM *table_reference* {LEFT|RIGHT} [OUTER] JOIN *table_reference ON* < search_condition>

Inner and Outer Joins Examples

R

A	B
1	2
3	3

S

B	C
2	4
4	6

Natural Inner Join

A	B	C
1	2	4

Natural Left outer Join

A	B	C
1	2	4
3	3	Null

Natural Right outer Join

A	B	C
1	2	4
Null	4	6

Natural outer Join

A	B	C
1	2	4
3	3	Null
Null	4	6

Outer join (without the Natural) will use the keyword "on" for specifying the condition of the join.

Outer join not implemented in MYSQL
Outer join is implemented in Oracle

Views

Null Values and Joins

Constraints and Triggers

Semantic Constraints

- Keys, entity constraints and referential integrity are structural constraints that are managed by the DBMS.
- Semantic constraints can be specified using CHECK and ASSERTION statements.
- The constraint is satisfied by a database state if no combination of tuples in the database state violates the constraint.

Semantic Constraints: Check

- Semantic constraints over a single table are specified using Check conditional-expressions

```
CREATE TABLE  Student
   ( snum  INTEGER,
     sname  CHAR(32),
     major  CHAR(32),
     standing CHAR(2)
     age  REAL,
     PRIMARY KEY  (snum),
     CHECK  ( age >= 10
           AND age < 100 );
```

Check constraints are checked when tuples are inserted or modified

Constraints Over Multiple Relations

- Constraints that cannot be defined in one table are defined as ASSERTIONs which are not associated with any table.

Example: *Every MovieStar needs to star in at least one Movie.*

Movie(MovieID, Title, Year)
StarsIn(MovieID, StarID, Role)
MovieStar(StarID, Name, Gender)

```
CREATE ASSERTION totalEmployment
CHECK
( NOT EXISTS (
     SELECT StarID
     FROM MovieStar
     WHERE StarID not in (   SELECT StarID
                             FROM StarsIn)));
```

Not tested in exams

Triggers

- An active database is a database that includes an event-driven architecture. Triggers are a procedure that start automatically if specified changes occur to the DBMS.

- A trigger has three parts:
  1. Event (activates the trigger)
  2. Condition (tests whether the trigger should run)
  3. Action (procedure executed when trigger runs)

- Database vendors did not wait for trigger standards! So trigger format depends on the DBMS

NOTE: triggers may cause cascading effects.

Triggers: Example (SQL:1999)

```
CREATE TRIGGER youngStudentUpdate
    AFTER INSERT ON Student
REFERENCING NEW TABLE NewStudent
FOR EACH STATEMENT
    INSERT INTO
        YoungStudent(snum, sname, major, standing, age)
        SELECT  snum, sname, major, standing, age
        FROM    NewStudent N
        WHERE  N.age <= 18;
```

event

newly inserted tuples

apply once per statement

action

Can be either before or after