

CS 4604: Introduction to Database Management Systems

B. Aditya Prakash

Lecture #4: SQL and Relational
Algebra---Part 2



- Most popular embedded db in the world
 - Iphone (iOS), Android, Chrome....
- (Very) Easy to use: no need to set it up
- Self-contained: data+schema
- DB on your laptop: useful for testing, understanding....

Subquery Example

- Find the name of the professor who teaches “CS 4604.”

```
SELECT Name
```

```
FROM Professors
```

```
WHERE PID =
```

Subquery Example

- Find the name of the professor who teaches “CS 4604.”

```
SELECT Name
```

```
FROM Professors
```

```
WHERE PID =
```

```
    (SELECT ProfessorPID
```

```
    FROM Teach
```

```
    WHERE (Number = 4604) AND (DeptName = ' CS' )
```

```
    );
```

- When using =, the subquery must return a single tuple

nested subqueries

- find student record with highest ssn

select *

from student

where ssn is

greater than every other ssn

nested subqueries

- find student record with highest ssn

select *

from student

where ssn *greater than every*

select ssn from student

nested subqueries

- find student record with highest ssn

```
select *
```

```
from student
```

```
where ssn > all (
```

```
    select ssn from student)
```

almost correct

nested subqueries

- find student record with highest ssn

```
select *
```

```
from student
```

```
where ssn >= all (
```

```
    select ssn from student)
```


Conditions Involving Relations

- SQL includes a number of operators that apply to a relation and produce a boolean result.
- These operators are very useful to apply on results of sub-queries.

Conditions Involving Relations

- Let R be a relation and t be a tuple with the same set of attributes.
 - **EXISTS** R is true if and only if R contains at least one tuple.
 - t **IN** R is true if and only if t equals a tuple in R .
 - t **> ALL** R is true if and only if R is unary (has one attribute) and t is greater than every value in R .
 - Can use any of the other five comparison operators.
 - If we use $<>$, R need not be unary.
 - t **> ANY** R (which is unary) is true if and only if t is greater than at least one value in R .
- We can use **NOT** to negate EXISTS, ALL, and ANY.

Subqueries Using Conditions

- Find the departments of the courses taken by the student with name 'Suri' .

```
SELECT DeptName
FROM Take
WHERE StudentPID IN
    ( SELECT PID
      FROM Students
      WHERE (Name = 'Suri' )
    );
```

Correlated vs Uncorrelated

- The previous subqueries did not depend on anything outside the subquery
 - ...and thus need to be executed just once.
 - These are called uncorrelated.
- A correlated subquery depends on data from the outer query
 - ... and thus has to be executed for each row of the outer table(s)

Correlated Subqueries

- Find course names that have been used for two or more courses.

```
SELECT CourseName
FROM Courses AS First
WHERE CourseName IN
    (SELECT CourseName
     FROM Courses
     WHERE (Number <> First.Number)
          AND (DeptName <> First.DeptName)
    );
```

Evaluating Correlated Subqueries

```
SELECT CourseName
FROM Courses AS First
WHERE CourseName IN
    (SELECT CourseName
     FROM Courses
     WHERE (Number <> First.Number)
          AND (DeptName <> First.DeptName)
    );
```

- Evaluate query by looping over tuples of First, and for each tuple evaluate the subquery.
- Scoping rules: an attribute in a subquery belongs to one of the tuple variables in that subquery's FROM clause, or to the immediately surrounding subquery, and so on.

Subqueries in FROM clauses

- Can use a subquery as a relation in a FROM clause.
- We must give such a relation an alias using the AS keyword.
- Let us find different ways of writing the query “Find the names of Professors who have taught the student whose first name is ‘Suri’.”
- The old way:

```
SELECT Professors.Name
```

```
FROM Professors, Take, Teach, Students
```

```
WHERE (Professors.PID = Teach.ProfessorPID)
```

```
    AND (Teach.CourseNumber = Take.CourseNumber)
```

```
    AND (Teach.DeptName = Take.DeptName)
```

```
    AND (Take.StudentPID = Student.PID)
```

```
    AND (Student.Name = 'Suri %');
```

- “Find the names of (Professors who have taught (courses taken by (student with first name ‘Suri’))).”

```
SELECT Name
FROM Professors
WHERE PID IN
    (SELECT ProfessorPID
     FROM Teach
     WHERE (Number, DeptName) IN
         ( SELECT Number, DeptName
           FROM Take, Students
           WHERE (StudentPID = PID) AND
                 (Students.Name = 'Suri%')));
```


Unrolling it further

- ```
SELECT Name
FROM Professors
WHERE PID IN
 (SELECT ProfessorPID
 FROM Teach
 WHERE (Number, DeptName) IN
 (SELECT Number, DeptName
 FROM Take
 WHERE StudentPID IN
 (SELECT PID
 FROM Students
 WHERE Name = 'Suri %'))));
```

# Data Types in SQL

**Read the  
Chapter!**

- Character strings:
  - CHAR(n): fixed-length string of n characters.
  - VARCHAR(n): string of length of up to n characters.
- Bit strings:
  - BIT(n): bit string of length n.
  - BIT VARYING(n): bit string of length upto n.
- BOOLEAN: possible values are TRUE, FALSE, and UNKNOWN
- integers: INTEGER (INT), SHORTINT.
- floats: FLOAT (or REAL), DOUBLE PRECISION.
- fixed point numbers: DECIMAL(n, d): a number with n digits, with the decimal point d positions from the right.
- dates and times: DATE and TIME

# Creating and Deleting Tables

- A table is a relation that is physically stored in a database.
- A table is **persistent**; it exists indefinitely unless dropped or altered in some way.
- Creating a table:
  - **CREATE TABLE** Students (PID VARCHAR(8), Name CHAR(20), Address VARCHAR(255));
- Deleting a table:
  - **DROP TABLE** followed by the name of the table.

# Modifying Table Schemas

- **ALTER TABLE** followed by the name of the relation followed by:
  - **ADD** followed by a column name and its data type.
  - Add date of birth (Dob) to Students:  
ALTER TABLE Students ADD Dob DATE;
  - **DROP** followed by a column name.

# Null Values

- **null** -> unknown, or inapplicable, (or ...)
- Complications:
  - 3-valued logic (true, false and *unknown*).
  - **null = null** : false!!

# Null and Default Values

- SQL allows NULL for unknown attribute values.
- NULL not allowed in certain cases.
- We can specify a default value for an attribute using the DEFAULT keyword.
  - ALTER TABLE Students ADD Gender char(1) DEFAULT '?';

# Inserting Data into a Table

- **INSERT INTO** R(A1,A2, . . . An) VALUES (v1, v2, . . . , vn).
  - (A1,A2, . . . ,An) can be a subset of R' s schema.
  - Remaining attributes get NULL values.
  - Can omit names of attributes if we provide values for all attributes and list values in standard order.
- Insertion: Instead of VALUES, can use a SELECT statement.
  - Insert into the Professors table all professors who are mentioned in Teach but are not in Professors.

```
INSERT INTO Professors(PID)
SELECT ProfessorPID
FROM Teach
WHERE ProfessorPID NOT IN
 (SELECT PID FROM Professors);
```

# Deleting Data from a Table

- **DELETE FROM** R WHERE C.
- Every tuple satisfying the condition C is deleted from R.



# Updating Data in a Table

- An update in SQL is a change to one of the tuples existing in the database.
- Example: change the name of a student so that every male student has ' Mr. ' added to the name and every female student has ' Ms. ' added to the name.
  - **UPDATE** Students  
    **SET** Name = ' Ms. ' || Name  
    **WHERE** Gender = ' F ' ;
  - **UPDATE** Students  
    **SET** Name = ' Mr. ' || Name  
    **WHERE** Gender = ' M ' ;
- Can set multiple attributes in the SET clause, separated by commas.
- The WHERE clause can involve a subquery.

# Loading Data: BULK

- Different RDBMs have different syntax.
- PostgreSQL: Use the `\copy 'filename' INTO TABLE tablename;` at the psql prompt
- File format:
  - Tab-delimited with columns in the same order as the attributes.
  - Use `\N` to indicate null values.
- SQLite: see homework 1
- Do not make assumptions about how the RDBMS will behave!
- Check to make sure your data is not corrupted.
- Do not delete the original files that contain the raw data.

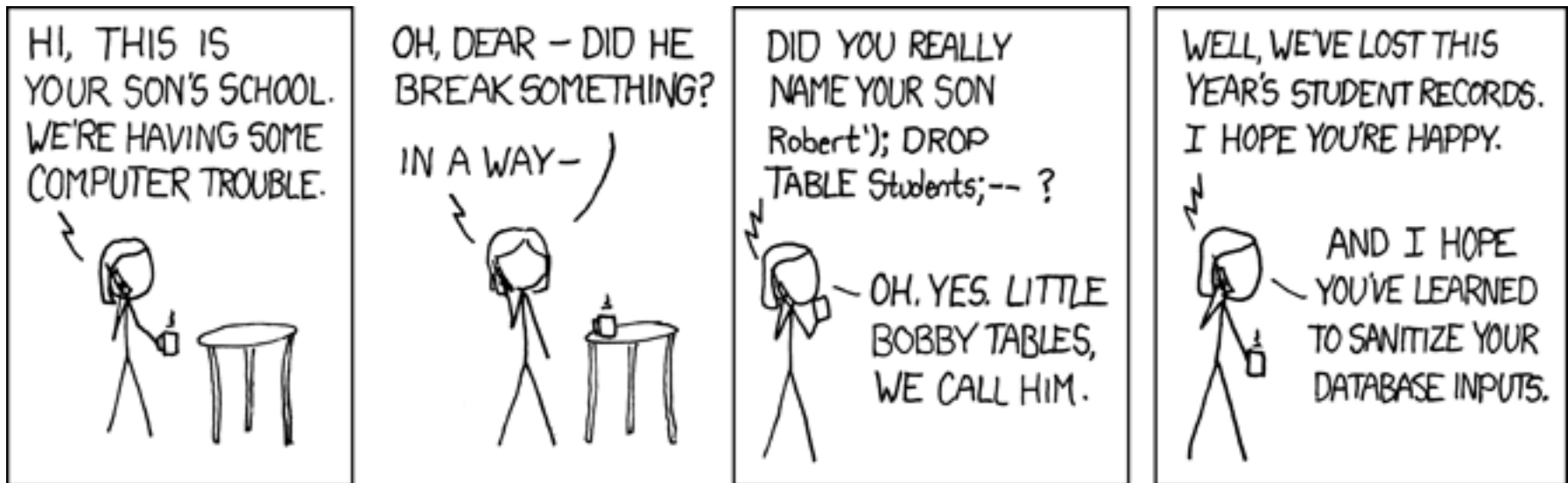
# Saving Data

- Use the `pg_dump` program:
  - `pg_dump -t table database`
- Use `man pg_dump` for more information.

# Constraints in Relational Algebra and SQL

# Maintaining Integrity of Data

- Data is **dirty**.
- How does an application ensure that a database modification does not corrupt the tables?



# Maintaining Integrity of Data

- Data is **dirty**.
- How does an application ensure that a database modification does not corrupt the tables?
- Two approaches:
  - Application programs check that database modifications are consistent.
  - Use the features provided by SQL.

# Integrity Checking in SQL

- PRIMARY KEY and UNIQUE constraints.
- FOREIGN KEY constraints.
- Constraints on attributes and tuples.
- Triggers (schema-level constraints).
  
- How do we express these constraints?
- How do we check these constraints?
- What do we do when a constraint is violated?

# Keys in SQL

- A set of attributes  $S$  is a key for a relation  $R$  if every pair of tuples in  $R$  **disagree on at least one attribute** in  $S$ .
- Select one key to be the **PRIMARY KEY**; declare other keys using **UNIQUE**.



# Primary Keys in SQL

- Modify the schema of Students to declare PID to be the key.
  - CREATE TABLE Students(  
    PID VARCHAR(8) **PRIMARY KEY**,  
    Name CHAR(20), Address VARCHAR(255));
- What about Courses, which has two attributes in its key?
  - CREATE TABLE Courses(Number integer, DeptName:  
    VARCHAR(8), CourseName VARCHAR(255), Classroom  
    VARCHAR(30), Enrollment integer,  
    **PRIMARY KEY (Number, DeptName)**  
);

# Effect of Declaring PRIMARY KEYS

- Two tuples in a relation cannot agree on all the attributes in the key. DBMS will reject any action that inserts or updates a tuple in violation of this rule.
- A tuple cannot have a NULL value in a key attribute.

# Other Keys in SQL

- If a relation has other keys, declare them using the UNIQUE keyword.
- Use UNIQUE in exactly the same places as PRIMARY KEY.
- There are two differences between PRIMARY KEY and UNIQUE:
  - A table may have **only one PRIMARY KEY** but more than one set of attributes declared UNIQUE.
  - A tuple **may have NULL values in UNIQUE** attributes.

# Enforcing Key Constraints

- Upon which actions should an RDBMS enforce a key constraint?
- Only tuple update and insertion.
- RDBMS searches the tuples in the table to find if any tuple exists that agrees with the new tuple on all attributes in the primary key.
- To speed this process, an RDBMS automatically creates an efficient search index on the primary key.
- User can instruct the RDBMS to create an index on one or more attributes

# Foreign Key Constraints

- **Referential integrity constraint:** in the relation Teach (that “connects” Courses and Professors), if Teach relates a course to a professor, then a tuple corresponding to the professor **must** exist in Professors.
- How do we express such constraints in Relational Algebra?
- Consider the Teach(ProfessorPID, Number, DeptName) relation.

*every non-NULL value of ProfessorPID in Teach must be a valid ProfessorPID in Professors.*

- **RA**  $\pi_{\text{ProfessorPID}}(\text{Teach}) \subseteq \pi_{\text{PID}}(\text{Professors})$ .

# Foreign Key Constraints in SQL

- every non-NULL value of ProfessorPID in Teach must be a valid ProfessorPID in Professors.
- In Teach, declare ProfessorPID to be a foreign key.
- CREATE TABLE Teach(ProfessorPID VARCHAR(8) REFERENCES Professor(PID), Name VARCHAR(30) ...);
- CREATE TABLE Teach(ProfessorPID VARCHAR(8), Name VARCHAR(30) ..., FOREIGN KEY ProfessorPID REFERENCES Professor(PID));
- If the foreign key has multiple attributes, use the second type of declaration.

# Requirements for FOREIGN KEYS

- If a relation  $R$  declares that some of its attributes refer to foreign keys in another relation  $S$ , then these attributes **must** be declared UNIQUE or PRIMARY KEY in  $S$ .
- Values of the foreign key in  $R$  must appear in the referenced attributes of some tuple in  $S$ .

# Enforcing Referential Integrity

- **Three** policies for maintaining referential integrity.
- **Default policy:** reject violating modifications.
- **Cascade policy:** mimic changes to the referenced attributes at the foreign key.
- **Set-NULL policy:** set appropriate attributes to NULL.



# Default Policy for Enforcing Referential Integrity

- **Reject** violating modifications. There are **four cases**.
  - Insert a new Teach tuple whose ProfessorPID is not NULL and is not the PID of any tuple in Professors.
  - Update the ProfessorPID attribute in a tuple in Teach to a value that is not the PID value of any tuple in Professors.
  - Delete a tuple in Professors whose PID value is the ProfessorPID value for one or more tuples in Teach.
  - Update the PID value of a tuple in Professors when the old PID value is the value of ProfessorPID in one or more tuples of Teach.

# Cascade Policy for Enforcing Referential Integrity

- Only applies to deletions of or updates to tuples in the referenced relation (e.g., Professors).
- If we delete a tuple in Professors, delete all tuples in Teach that refer to that tuple.
- If we update the PID value of a tuple in Professors from  $p1$  to  $p2$ , update all value of ProfessorPID in Teach that are  $p1$  to  $p2$ .

# Set-NULL Policy for Enforcing Referential Integrity

- Also applies only to deletions of or updates to tuples in the referenced relation (e.g., Professors).
- If we delete a tuple in Professors, set the ProfessorPID attributes of all tuples in Teach that refer to the deleted tuple to NULL.
- If we update the PID value of a tuple in Professors from p1 to p2, set all values of ProfessorPID in Teach that are p1 to NULL

# Specifying Referential Integrity Policies in SQL

- SQL allows the database designer to specify the policy for deletes and updates independently.
- Optionally follow the declaration of the foreign key with **ON DELETE** and/or **ON UPDATE** followed by the policy: **SET NULL** or **CASCADE**.
- Constraints can be circular, e.g., if there is a one-one mapping between two relations.
- In this case, SQL allows us to defer the checking of constraints.
- For your project, you do not have to consider deferring constraints.

# Constraining Attributes and Tuples

- SQL also allows us to specify constraints on attributes in a relation and on tuples in a relation.
  - Disallow courses with a maximum enrollment greater than 100.
  - A chairperson of a department must teach at most one course every semester.
- How do we express such constraints in SQL?
- How can we change our minds about constraints?
- A simple constraint: NOT NULL
  - Declare an attribute to be NOT NULL after its type in a CREATE TABLE statement.
  - Effect is to disallow tuples in which this attribute is NULL.

# Attribute-Based CHECK Constraints

- Disallow courses with a maximum enrollment greater than 100.
- This constraint only affects the value of a single attribute in each tuple.
- Follow the declaration of the Enrollment attribute with the CHECK keyword and a condition.
- CREATE TABLE Courses(...  
    Enrollment INT CHECK (Enrollment <= 100) ...);
- The condition can be any condition that can appear in a WHERE clause.
- CHECK statement may use a subquery to mention other attributes of the same or other relations.
- An attribute-based CHECK constraint is checked **only when the value of that attribute changes.**

# Tuple-Based CHECK Constraints

- Tuple-based CHECK constraints are checked whenever a tuple is inserted into or updated in a relation.
- Designer may add these constraints after the list of attributes in a CREATE TABLE statement.
- A chairperson of a department teaches at most one course in any semester.

```
CREATE TABLE Teach(...
```

```
 CHECK ProfessorPID NOT IN
```

```
 ((SELECT ProfessorPID FROM Teach)
```

```
 INTERSECT
```

```
 (SELECT ChairmanPID FROM Departments)
```

```
)
```

```
);
```

# Modifying Constraints

- SQL allows constraints to be named.
- Use CONSTRAINT followed by the name of the constraint in front of PRIMARY KEY, UNIQUE, or CHECK.
- Can use constraint names in ALTER TABLE statements to delete constraints: say DROP CONSTRAINT followed by the name of the constraint.
- Can add constraints in an ALTER TABLE statement using ADD CONSTRAINT followed by an optional name followed by the (required) CHECK statement.



# Assertions

- These are database-schema elements, like relations
- Defined by:
  - CREATE ASSERTION <name>  
CHECK (<condition>);
- Condition may refer to any relation or attribute in the database schema.

# Assertions: Example

- Can't have more courses than students ('Pigeonhole Principle')

```
CREATE ASSERTION FewStudents CHECK (
 (SELECT COUNT(*) FROM Students) <=
 (SELECT COUNT(*) FROM Courses)
);
```

# Triggers: Motivation

- Assertions are powerful, but the DBMS often can't tell when they need to be checked.
- Attribute- and tuple-based checks are checked at known times → but not powerful.
- Triggers let the user decide when to check for any condition.

# Triggers

- Trigger: procedure that starts automatically if specified changes occur to the DBMS
- A trigger has three parts:
  - **Event** (activates the trigger)
  - **Condition** (tests whether the triggers should run)
  - **Action** (what happens if the trigger runs)

Maintain a unary relation `New_Courses` which has the list of brand new courses

# Triggers

- Trigger: procedure that starts automatically if specified changes occur to the DBMS
- A trigger has three parts:
  - **Event** (activates the trigger)
  - **Condition** (tests whether the triggers should run)
  - **Action** (what happens if the trigger runs)

```
CREATE TRIGGER incr_count
AFTER INSERT ON Teach // Event
REFERENCING NEW ROW AS new
FOR EACH ROW
WHEN (new.id NOT IN (SELECT ID FROM Courses)) // Condition
INSERT INTO New_Courses(id) VALUES(new.id); // Action
```

# OK, what could have been done?

