Summary of SQL Features

- Query
  - SELECT-FROM-WHERE statements
  - Set and bag operations
  - Table expressions, subqueries
  - Aggregation and grouping
  - Ordering
  - Outerjoins

- Table creation, constraints, and content updates
Today’s Topic

- Subqueries
- Active databases
- View
- Indexing
Quantified subqueries

- A quantified subquery can be used as a value in a `WHERE` condition

- **Universal quantification** (for all):
  ... `WHERE x op ALL (subquery)` ...
  - True iff for all \( t \) in the result of \( \text{subquery} \), \( x \ op \ t \)

- **Existential quantification** (exists):
  ... `WHERE x op ANY (subquery)` ...
  - True iff there exists some \( t \) in the result of \( \text{subquery} \) such that \( x \ op \ t \)

  ❗ Beware
  - In common parlance, “any” and “all” seem to be synonyms
  - In SQL, `ANY` really means “some”

- Also may use `EXISTS (NOT EXISTS)`, `IN (NOT IN)`
Examples of quantified subqueries

• Which employees have the highest salary?
  Employee (Sid, Name, Salary)
  ```sql
  SELECT *
  FROM Employee
  WHERE Salary >= ALL
    (SELECT Salary FROM Employee);
  ```

• How about the lowest salary?
  ```sql
  SELECT *
  FROM Employee
  WHERE Salary <= ALL
    (SELECT salary FROM Employee);
  ```
More ways of getting the highest Salary

Who has the highest Salary?

- SELECT * FROM Employee e
  WHERE NOT EXISTS
    (SELECT * FROM Employee
     WHERE Salary > e.Salary);

- SELECT * FROM Employee
  WHERE Eid NOT IN
    (SELECT e1.SID
     FROM Employee e1, Employee e2
     WHERE e1.Salary < e2.Salary);
Nested Queries: Scoping

- To find out which table a column belongs to
  - Start with the immediately surrounding query
  - If not found, look in the one surrounding that; repeat if necessary
- Use `table_name.column_name` notation and `AS` (renaming) to avoid confusion
An Example

SELECT * FROM Student s
WHERE EXISTS
  (SELECT * FROM Enroll e
   WHERE SID = s.SID
   AND EXISTS
     (SELECT * FROM Enroll
      WHERE SID = s.SID
      AND CID <> e.CID))

Students who are taking at least two courses
Nested Queries

- Nested queries do not add expression power to SQL
  - For convenient
  - Write intuitive SQL queries
  - Can always use SQL queries without nesting to complete the same task (though sometime it is hard)
Database Design

- Active database
- View
- Indexing
- Thinking about the following “real-world” situation

You need to design a database for students to record student name, gpa (>0), id, email address, home department, and images. Constraints: student id is the primary key, student gpa can not be negative, a student may take up to 10 courses in a semester (unless they are senior), and whenever a new honor student is entered in the table, this student is automatically enrolled a honor course “Bio101, How Jayhawk fly”
Active Database

- Constraints are efficient ways to improve the quality of the database.
- We have covered domain constraint
- We will talk about
  - Check
  - Assertion
  - Trigger
Check

- Student GPA must be positive
- CREATE TABLE Student (  
  SId char(10),  
  Sname char(20),  
  email char(10),  
  gpa float,  
  CHECK (gpa >= 0.0)  
);
Assertion

- Make sure each student can take up to 10 course projects, unless the student is a senior (level $\geq 4$)

```sql
CREATE ASSERTION uplimitCourse
CHECK ( (SELECT MAX COUNT (sid)
            FROM student, enrollment
            WHERE level < 4
            GROUP BY sid) $\leq$ 10)
```
Trigger example

CREATE TRIGGER EECS168AutoRecruit
AFTER INSERT ON Student
REFERENCING NEW ROW AS newStudent
FOR EACH ROW
WHEN (newStudent.isHonor)
INSERT INTO Enroll
VALUES(newStudent.SID, 'BIO101');
Views

- A **view** is like a “virtual” table
  - Defined by a query, which describes how to compute the view contents on the fly
  - DBMS stores the **view definition query** instead of view contents
  - Can be used in queries just like a regular table
Why use views?

- To hide data from users
- To hide complexity from users
- Logical data independence
  - If applications deal with views, we can change the underlying schema without affecting applications
  - Recall physical data independence: change the physical organization of data without affecting applications
- To provide a uniform interface for different implementations or sources
  - Real database applications use tons of views
Creating and dropping views

- Example: EECS647roster
  - `CREATE VIEW EECS647Roster AS
    SELECT SID, name, age, GPA
    FROM Student
    WHERE SID IN (SELECT SID FROM Enroll
      WHERE CID = 'EECS647');`

- To drop a view
  - `DROP VIEW view_name;`
Using views in queries

- Example: find the average GPA of EECS647 students
  - `SELECT AVG(GPA) FROM EECS647Roster;`
  - To process the query, replace the reference to the view by its definition
  - `SELECT AVG(GPA)
    FROM (SELECT SID, name, age, GPA
      FROM Student
      WHERE SID IN (SELECT SID
        FROM Enroll
        WHERE CID = 'EECS647'));`
Modifying views

- Does not seem to make sense since views are virtual
- But does make sense if that is how users see the database
- Goal: modify the *base tables* such that the modification would appear to have been accomplished on the view
- Be careful!
  - There may be one way to modify
  - There may be many ways to modify
  - There may be no way to modify
A simple case

CREATE VIEW StudentGPA AS
    SELECT SID, GPA FROM Student;

DELETE FROM StudentGPA WHERE SID = 123;

translates to:

DELETE FROM Student WHERE SID = 123;
An impossible case

CREATE VIEW HighGPAStudent AS
    SELECT SID, GPA FROM Student
    WHERE GPA > 3.7;

INSERT INTO HighGPAStudent
    VALUES(987, 2.5);

- No matter what you do on Student, the inserted row will not be in HighGPAStudent
A case with too many possibilities

CREATE VIEW AverageGPA(GPA) AS
    SELECT AVG(GPA) FROM Student;

- Note that you can rename columns in view definition

UPDATE AverageGPA SET GPA = 2.5;

- Set everybody’s GPA to 2.5?
- Adjust everybody’s GPA by the same amount?
- Just lower Lisa’s GPA?
SQL92 updateable views

- More or less just single-table selection queries
  - No join
  - No aggregation
  - No subqueries

- Arguably somewhat restrictive

- Still might get it wrong in some cases
  - See the slide titled “An impossible case”
  - Adding WITH CHECK OPTION to the end of the view definition will make DBMS reject such modifications
Indexes for Performance Tuning

- An index is an auxiliary persistent data structure
  - Search tree (e.g. B-tree, R-tree)
  - Lookup table (e.g., hash table)
- An index on $R.A$ can speed up accesses of the form
  - $R.A = \text{value}$ (lookup)
  - $R.A > \text{value}$ (range query)
- An index on $(R.A_1, \ldots, R.A_n)$ can speed up
  - $R.A_1 = \text{value}_1 \land \ldots \land R.A_n = \text{value}_n$
  - $(R.A_1, \ldots, R.A_n) > (\text{value}_1, \ldots, \text{value}_n)$

More on indexes in the second half of this course!
Creating and dropping indexes

- **CREATE [UNIQUE] INDEX index_name ON table_name (column_name_1, ..., column_name_n);**
  - With **UNIQUE**, the DBMS will also enforce that 
    \{column_name_1, ..., column_name_n\} is a key of 
    table_name
  - e.g. **CREATE INDEX name_index ON STUDENT (SName);**

- **DROP INDEX index_name;**

- Typically, the DBMS will automatically create indexes 
  for **PRIMARY KEY** and **UNIQUE** constraint declarations
Choosing indexes to create

More indexes = better performance?

- Indexes take space
- Indexes need to be maintained when data is updated
- Indexes have one more level of indirection

Optimal index selection depends on both query and update workload and the size of tables
What’s next?

- Database meets Concurrency
  - Transaction (will be covered after we talk about transaction management)