UNIVERSITY OF MARYLAND

UMD DATA605 - Big Data Systems

Lesson 4.2: SQL

Instructor: Dr. GP Saggese, gsaggese@umd.edu

SQL Overview

- **Relational algebra**: mathematical language to manipulate *relations*
- **SQL**: language to describe and transform data in a relational DB
  - Originally Sequel
  - Changed to Structured Query Language
- **SQL statements grouped by goal**
  - Data definition language (DDL)
    - Define schema (tables, attributes, indices)
    - Specify integrity constraints (primary key, foreign key, not null)
  - Data modification language (DML)
    - Modify data in tables
    - Insert, Update, Delete
  - Query data (DQL)
  - Control transactions
    - Specify beginning and end, control isolation level
  - Define views
  - Authorization
    - Specify access and security constraints

SQL Overview

- Data description language (DDL)

```
CREATE TABLE <name> (<field> <domain>, ... )
```

- Data modification language (DML)

```
INSERT INTO <name> (<field names>) VALUES (<field values>)
DELETE FROM <name> WHERE <condition>
UPDATE <name> SET <field name> = <value> WHERE <condition>
```

- Query language

```
SELECT <fields> FROM <name> WHERE <condition>
```

Create Table

```
CREATE TABLE r
 (A_1 D_1,
 A_2 D_2,
 ...,
 A_n D_n,
 IntegrityConstraint_1,
 IntegrityConstraint_n);
```

where:

- `r` is name of *table* (aka *relation*)
- `A_i` name of *attribute* (aka *field*, *column*)
- `D_i` domain of attribute `A_i`

- **Constraints**
  - SQL prevents changes violating integrity constraints
  - Primary key
    - Must be non-null and unique
    - `PRIMARY KEY (A_j1, A_j2, ..., A_jn)`
  - Foreign key
    - Attribute values must match primary key values in relation s
    - `FOREIGN KEY (A_k1, A_k2, ..., A_kn) REFERENCES s`
  - Not null
    - Null value not allowed for attribute
    - A_i D_i NOT NULL

SCIENCE ACADEMY

Select

```sql
SELECT A_1, A_2, ..., A_n
    FROM r_1, r_2, ..., r_m
    WHERE P;
```

- `SELECT`: select the attributes to list (i.e., projection)
- `FROM`: list of tables to be accessed
  - Define a Cartesian product of the tables
  - The query is going to be optimized to avoid to enumerate tuples that will be eliminated
- `WHERE`: predicate involving attributes of the relations in the `FROM` clause (i.e., selection)
- In `SELECT` or `WHERE` clauses, might need to use the table names as prefix to qualify the attribute name
  - E.g., `instructor.ID` vs `teaches.ID`
- A `SELECT` statement can be expressed in terms of relational algebra
  - Cartesian product $\rightarrow$ selection $\rightarrow$ projection
  - Difference: SQL allows duplicate values, relational algebra works with mathematical sets

Null Values

- An arithmetic operation with `NULL` yields `NULL`
- Comparison with NULL
  - `1 < NULL`
  - `NOT(1 < NULL)`
  - SQL yields `UNKNOWN` when comparing with `NULL` value
  - There are 3 logical values: `True`, `False`, `Unknown`
- Boolean operators
  - Can be extended according to common sense, e.g.,
  - `True AND UNKNOWN = UNKNOWN`
  - `False AND Unknown = False`
- In a `WHERE` clause, if the result is `UNKNOWN` it's not included

Group by Query

- The attributes in GROUP BY are used to form groups
  - Tuples with the same value on all attributes are placed in one group
- Any attribute that is not in the GROUP BY can appear in the SELECT clause only as argument of aggregate function

```
SELECT dept_name, AVG(salary)
    FROM instructor
    GROUP BY dept_name;

-- Error.
SELECT dept_name, salary
    FROM instructor
    GROUP BY dept_name;
```

- salary is not in GROUP BY so it must be in an aggregate function

ID	*name*	*dept_name*	*salary*
76766	Crick	Biology	72000
45565	Katz	Comp. Sci.	75000
10101	Srinivasan	Comp. Sci.	65000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
22222	Einstein	Physics	95000

dept_name	*avg_salary*
Biology	72000
Comp. Sci.	77333
Elec. Eng.	80000
Finance	85000
History	61000
Music	40000
Physics	91000

Having

- State a condition that applies to groups instead of tuples (like WHERE)
- Any attribute in the HAVING clause must appear in the GROUP BY clause
- E.g., find departments with avg salary of instructors > 42k

```
SELECT dept_name, AVG(salary) AS avg_salary
    FROM instructor
    GROUP BY dept_name
    HAVING AVG(salary) > 42000;
```

- How does it work
  - FROM is evaluated to create a relation
  - (optional) WHERE is used to filter
  - GROUP BY collects tuples into groups
  - (optional) HAVING is applied to each group and groups are filtered
  - SELECT generates tuples of the results, applying aggregate functions to get a single result for each group

Nested Subqueries

- SQL allows using the result of a query in another query
  - Use a subquery returning one attribute (scalar subquery) where a value is used
  - Use the result of a query for set membership in the WHERE clause
  - Use the result of a query in a FROM clause :::columns ::::{.column width=60%}

```sql
SELECT tmp.dept_name, tmp.avg_salary
FROM (
  SELECT dept_name,
      AVG(salary) AS avg_salary
  FROM instructor
  GROUP BY dept_name) AS tmp
WHERE avg_salary > 42000
```

:::: ::::{.column width=40%}

dept_name	avg_salary
Finance	85000.000000000000

With

- `WITH` clause allows to define a temporary relation containing the results of a subquery
- It can be equivalent to a nested subqueries, but clearer
- E.g., find department with the maximum budget :::columns ::::{.column width=80%}

```sql
WITH max_budget(value) as (
  SELECT MAX(budget) FROM department)
  SELECT department.dept_name, budget
    FROM department, max_budget
    WHERE department.budget = max_budget.value
```

:::: ::::{.column width=20%}

dept_name	budget
Finance	120000.00

Insert

- To insert data into a relation we can specify tuples to insert
  - Tuples

```
INSERT INTO course VALUES ('DATA-605', 'Big data systems', '(
INSERT INTO course(course_id, title, dept_name, credits)
    VALUES ('DATA-605', 'Big data systems', 'Comp. Sci.', 3)
```

  - Query whose results is a set of tuples

```
INSERT INTO instructor
    (SELECT ID, name, dept_name, 18000
    FROM student
    WHERE dept_name = 'Music' AND tot_cred > 144)
```

  - Nested queries are evaluated and then inserted so this doesn't create infinite loops

```
INSERT INTO student (SELECT * FROM student)
```

  - Many DB have bulk loader utilities to insert a large set of tuples into a relation, reading from formatted text files This is much faster than INSERT statements

Update

- SQL can change a value in a tuple without changing all the other values
- E.g., increase salary of all instructors by 5%

```
UPDATE instructor SET salary = salary * 1.05
```

- E.g., conditionally

```
UPDATE instructor SET salary = salary * 1.05
    WHERE salary < 70000
```

- Nesting is allowed

```
UPDATE instructor SET salary = salary * 1.05
    WHERE salary < (SELECT AVG(salary) FROM instructor)
```

Delete

- One can delete tuples using a query returning entire rows of a table

  ```
  DELETE FROM r WHERE p
  ```

  where:

  - r is a relation
  - P is a predicate

- Remove all tuples (but not the table)

  ```
  DELETE FROM instructor
  ```

SQL Tutorial

- SQL tutorial dir
- Readme** **
  - Explains how to run the tutorial
- Three notebooks in tutorial_university
- **How to learn from a tutorial**
  - Reset the notebook
  - Execute each cell one at the time
  - Ideally create a new file and retype (!) everything
  - Understand what each cell does
  - Look at the output
  - Change the code
  - Play with it
  - Build your mental model

sql_basics.ipynb
sql_joins.ipynb
sql_nulls_and_unknown.ipynb

- ***Movie Database Example (Optional)***

Example Schema for SQL Queries

```
Movie(title, year, length, inColor, studioName, producerC\#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert\#, netWorth)
Studio(name, address, presC\#)
```

title	year	length	inColor	studioName	producerC#
Plane Crazy	1927	6	no	Disney	WD100
Casablanca	1942	102	no	Warner Bros.	HW101
Star Wars	1977	121	yes	LucasFilm	GL102
Star Wars: ESB	1980	120	yes	LucasFilm	GL102
Star Wars: TPM	1999	133	yes	LucasFilm	GL102
Raiders of the Lost Ark	1981	115	yes	Paramount	FM103

Movies

name	address	gender	birthdate
H. Ford	...	M	7/13/1942
M. Hamill	...	M	9/25/1951
C. Fisher	...	F	10/21/1956

MovieStar

movieTitle	movieYear	starName
Star Wars	1977	H. Ford
Star Wars	1977	M. Hamill
Star Wars	1977	C. Fisher
Star Wars: ESB	1980	H. Ford
Star Wars: ESB	1980	M. Hamill
Star Wars: ESB	1980	C. Fisher
Raiders of the Lost Ark	1981	H. Ford

StarsIn

name	address	cert#	netWorth
Walt Disney	...	WD100	...
Hal B. Wallis	...	HW101	...
George Lucas	...	GL102	zillions
Frank Marshall	...	FM103	...
Robert Iger	...	RI104	...

MovieExec

name	address	presC#
LucasFilm	...	GL101
Disney	...	RI103
Warner Bros.	...	...

Studio

SQL: Data Definition

- CREATE TABLE

```
CREATE TABLE movieExec (
    name char(30),
    address char(100),
    cert# integer primary key,
    networth integer);

CREATE TABLE movie (
    title char(100),
    year integer,
    length integer,
    inColor smallint,
    studioName char(20),
    producerC# integer references
        movieExec(cert#) );
```

- Must define movieExec before movie. Why?

SQL: Data Manipulation

- INSERT

```
INSERT INTO StarsIn values('King Kong', 2005, 'Naomi Watts
```

```
INSERT INTO StarsIn(starName, movieTitle, movieYear)
    values('Naomi Watts', 'King Kong', 2005);
```

- DELETE

```
DELETE FROM movies WHERE movieYear < 1980;
```

  - Syntax is fine, but this command will be rejected. Why?

```
DELETE FROM movies
    WHERE length < (SELECT avg(length) FROM movies);
```

  - Problem: as we delete tuples, the average length changes
  - Solution:
    - First, compute avg length and find all tuples to delete
    - Next, delete all tuples found above (without recomputing avg or retesting the tuples)

SQL: Data Manipulation

- UPDATE
  - Increase all movieExec netWorth's over 100,000 USD by 6%, all other accounts receive 5%
  - Write two update statements:

```
UPDATE movieExec SET netWorth = netWorth * 1.06
    WHERE netWorth > 100000;
UPDATE movieExec SET netWorth = netWorth * 1.05
    WHERE netWorth <= 100000;
```

  - The order is important
  - Can be done better using the case statement

```
UPDATE movieExec SET netWorth =
    CASE
        WHEN netWorth > 100000 THEN netWorth * 1.06
        WHEN netWorth <= 100000 THEN netWorth * 1.05
    END;
```

SQL Single Table Queries

- Movies produced by Disney in 1990: note the *rename*

```
SELECT m.title, m.year
    FROM movie m
    WHERE m.studioname = 'disney' AND m.year = 1990;
```

  - The SELECT clause can contain expressions

```
SELECT title || ' (' || to_char(year) || ')' AS titleyear
```

```
SELECT 2014 - year
```

  - The WHERE clause support a large number of different predicates and combinations thereof

```
year BETWEEN 1990 and 1995
```

```
title LIKE 'star wars%'
```

```
title LIKE 'star wars _'
```

Single Table Queries

- Find distinct movies sorted by title

```
SELECT DISTINCT title
    FROM movie
    WHERE studioname = 'disney' AND year = 1990
    ORDER by title;
```

- Average length of a movie

```
SELECT year, avg(length)
    FROM movie
    GROUP BY year;
```

- **GROUP BY:** is a very important concept that shows up in many data processing platforms
  - What it does:
    - Partition the tuples by the group attributes (*year* in this case)
    - Do something (*compute avg* in this case) for each group
    - Number of resulting tuples == number of groups

Single Table Queries

- Find movie with the maximum length

```
SELECT title, year
    FROM movie
    WHERE movie.length = (SELECT max(length) FROM movie);
```

  - The smaller "subquery" is called a "nested subquery"

- Find movies with at most 5 stars: an example of a correlated subquery

```
SELECT *
    FROM movies m
    WHERE 5 >= (SELECT count(*)
                FROM starsIn si
                WHERE si.title = m.title AND
                      si.year = m.year);
```

  - The "inner" subquery counts the number of actors for that movie.

SCIENCE ACADEMY

Single Table Queries

- Rank movies by their length

```
SELECT title, year,
    (SELECT count(*)
    FROM movies m2
    WHERE m1.length <= m2.length) AS rank
    FROM movies m1;
```

  - Key insight: A movie is ranked 5th if there are exactly 4 movies with longer length.
  - Most database systems support some sort of a *rank* keyword for doing this
  - The above query doesn't work in presence of ties, etc.

- Set operations

```
SELECT name
    FROM movieExec
union/intersect/minus
    SELECT name FROM
        movieStar
```

Single Table Queries

- Set Comparisons

```
SELECT *
    FROM movies
    WHERE year IN [1990, 1995, 2000];
```

```
SELECT *
    FROM movies
    WHERE year NOT IN (
        SELECT EXTRACT(year from birthdate)
        FROM MovieStar
    );
```

Multi-Table Queries

- Key:
  - Do a join to get an appropriate table
  - Use the constructs for single-table queries
  - You will get used to doing all at once
- Examples:

```sql
SELECT title, year, me.name AS producerName
    FROM movies m, movieexec me
    WHERE m.producerC# = me.cert#;
```

Multi-Table Queries

- Consider the query:

```
SELECT title, year, producerC#, count(starName)
    FROM movies, starsIn
    WHERE title = starsIn.movieTitle AND
          year = starsIn.movieYear
    GROUP BY title, year, producerC#
```

- What about movies with no stars?
- Need to use **outer joins**

```
SELECT title, year, producerC#, count(starName)
    FROM movies LEFT OUTER JOIN starsIn
    ON title = starsIn.movieTitle AND year = starsIn.movie
    GROUP BY title, year, producerC#
```

- All tuples from 'movies' that have no matches in starsIn are included with NULLs
- So if a tuple (m1, 1990) has no match in starsIn, we get (m1, 1990, NULL) in the result
- The count(starName) works correctly then
- Note: count(*) would not work correctly (NULLs can have unintuitive behavior)

Other SQL Constructs

- Views

```
CREATE VIEW DisneyMovies
    SELECT *
        FROM movie m
        WHERE m.studioname = 'disney';
```

  - Can use it in any place where a table name is used
  - Views are used quite extensively to:
    - Simplify queries
    - Hide data (by giving users access only to specific views)
  - Views may be *materialized* or not

SCIENCE ACADEMY

Other SQL Constructs

- NULLs
  - Value of any attribute can be NULL
  - Because: value is unknown, or it is not applicable, or hidden, etc.
  - Can lead to counterintuitive behavior
  - For example, the following query does not return movies where length = NULL

    ```
    SELECT * FROM movies
    WHERE length >= 120 OR length <= 120
    ```
  - Aggregate operations can be especially tricky
- Transactions
  - A transaction is a sequence of queries and update statements executed as a single unit
  - For example, transferring money from one account to another
    - Both the *deduction* from one account and *credit* to the other account should happen, or neither should
- Triggers
  - A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database

Other SQL Constructs

- Integrity Constraints
  - Predicates on the database that must always hold
  - Key Constraints: Specifying something is a primary key or unique

```
CREATE TABLE customer (
    ssn CHAR(9) PRIMARY KEY,
    cname CHAR(15),
    address CHAR(30),
    city CHAR(10),
UNIQUE (cname, address, city));
```

  - Attribute constraints: Constraints on the values of attributes

```
bname char(15) not null
balance int not null, check (balance>= 0)
```

Integrity Constraints

- Referential integrity: prevent dangling tuples

```
CREATE TABLE branch(bname CHAR(15) PRIMARY KEY, ...);
CREATE TABLE loan(..., FOREIGN KEY bname REFERENCES branch);
```

- Can tell the system what to do if a referenced tuple is being deleted

Integrity Constraints

- Global Constraints
  - Single-table

    ```
    CREATE TABLE branch (..., bcity CHAR(15), assets INT,
        CHECK (NOT(bcity = 'Bkln') OR assets>5M))
    ```

  - Multi-table

    ```
    CREATE ASSERTION loan-constraint
    CHECK (NOT EXISTS
        (SELECT* FROM loan AS L WHERE NOT EXISTS
            (SELECT* FROM borrower B, depositor D, account A
            WHERE B.cname = D.cname AND D.acct_no = A.acct_no
                AND L.lno= B.lno)))
    ```

Additional SQL Constructs

- SELECT subquery factoring
  - To allow assigning a name to a subquery, then use its result by referencing that name

```
WITH temp AS (
    SELECT title, avg(length)
    FROM movies
    GROUP BY year)
SELECT COUNT(*) FROM temp;
```

  - Can have multiple subqueries (multiple with clauses)
  - Real advantage is when subquery needs to be referenced multiple times in main select
  - Helps with complex queries, both for readability and maybe performance (can cache subquery results)

Another SQL Construct

- `SELECT HAVING` clause
  - Used in combination with `GROUP BY` to restrict the groups of returned rows to only those where condition evaluates to true

```
SELECT year, count(*)
    FROM movies
    WHERE year > 1980
    GROUP BY year
    HAVING COUNT(*) > 10;
```

  - Difference from `WHERE` clause is that it applies to summarized group records, and where applies to individual records

SCIENCE ACADEMY