LECTURE 11 SQL BASICS

Introduction

- SQL is the standard language for querying and manipulating (relational) data (https://www.sql50.com)
  - **SQL** stands for **S**tructured **Q**uery **L**anguage
  - Initially developed at IBM by Donald Chamberlin and Raymond Boyce in the early 1970s, and called **SEQUEL** (*Structured English Query Language*)
  - Many standards out there: SQL92, SQL3, SQL99, ……, **SQL2016**
  - **(Caveats)**Vendors support various subsets of these standards
- Why SQL?
  1. A **very-high-level, declarative language**, in which the programmer is able to avoid specifying *a lot of data-manipulation details* that would be necessary in languages like C++
  2. Its queries are "**optimized**" quite well over the years, yielding efficient query execution and performance

Components of SQL

- DDL – Data Definition Language
  - Define and modify schema
- DML – Data Manipulation Language
  - Query (glorified search) data
  - Insert data
  - Remove data
  - Update data

Some SQL syntax

- SQL is *case-insensitive*
  - In general, upper- and lower-case characters are the same, except inside quoted strings
- Single quote for a string. Two single quotes inside a string represent the single-quote (apostrophe)
- Each statement finishes with a semicolon (;).

Create Table

Format: CREATE TABLE <name> (

<list of elements>

);

- Each element is a pair consisting of an attribute and a type
- The most common types are:
  - INT or INTEGER (synonyms)
  - REAL or FLOAT (synonyms)
  - CHAR(*n*) = fixed-length string of *n* characters
  - VARCHAR(*n*) = variable-length string of **up to** *n* characters

Create Table - example

```
CREATE TABLE Sells (
        bar     CHAR(20),
        beer    VARCHAR(20),
        price   REAL
    );
```

Declaring keys

- An attribute or list of attributes may be declared PRIMARY KEY or UNIQUE
  - Each says the attribute(s) so declared functionally determines all the attributes of the relation schema
  - There can be **only one PRIMARY KEY** for a relation, but **several UNIQUE attributes**
  - No attribute of a PRIMARY KEY can ever be **NULL** in any tuple. But attributes declared UNIQUE may have NULL's, and there may be several tuples with NULL
  - Single attribute keys

```
CREATE TABLE Beers (
        name CHAR(20) UNIQUE,
        manf CHAR(20)
);
```

Multi-attribute keys

```
CREATE TABLE Sells (
        bar         CHAR(20),
        beer        VARCHAR(20),
        price       REAL,
        PRIMARY KEY (bar, beer)
);
```

Other Declarations for Attributes

- Two other declarations we can make for an attribute are:
  - **NOT NULL** means that the value for this attribute may never be NULL
  - **DEFAULT <value>** says that if there is no specific value known for this attribute's component in some tuple, use the stated <value>

```
CREATE TABLE Drinkers (
        name CHAR(30) PRIMARY KEY,
        addr CHAR(50)
        DEFAULT '123 Monroe St.',
        phone CHAR(16)
);
```

Other useful SQL DDL commands

- Add an attribute to a table

```
ALTER TABLE Bars ADD
phone CHAR(10) DEFAULT 'unlisted';
```

- Remove an attribute from a table

```
ALTER TABLE Bars
DROP license;
```

- Remove a table

```
DROP TABLE Bars
```

Table manipulation commands summary

- Create a table
- Remove a table
- Modify a table:
  - Add fields to a table
  - Remove fields from a table

Insert a new record into a table

- The insert into statement inserts a new record into a table
  - INSERT INTO *table_name* (*column1*, *column2*, *column3*, ...)
    VALUES (*value1*, *value2*, *value3*, ...);
  - INSERT INTO *table_name*
    VALUES (*value1*, *value2*, *value3*, ...);

Insert into example

```
CREATE TABLE Drinkers (
                name CHAR(30) PRIMARY KEY,
                addr CHAR(50)
                DEFAULT '123 Monroe St.',
                phone CHAR(16)
        );
Insert into Drinkers(name, addr, phone)
   Values ('John Smith', '201 Marty Ln', '555-666-7777')
```

NAME	ADDR	PHONE
John Smith	201 Marty Ln	555-666-7777

Insert into example

- `Insert into Drinkers(name, phone)`
  `Values ('Sophie Green', '111-222-3333')`
- `Insert into Drinker(name)`
  `Values ('John Doe')`

NAME	ADDR	PHONE
John Smith	201 Marty Ln	555-666-7777
Sophie Green	123 Monroe St.	111-222-3333
John Doe	123 Monroe St.	NULL

Write SQL commands to create the following table

Employees

EmployeeID	LastName	FirstName	DeptID	Age
001	John	Smith	100	20
002	Why	Me	NULL	NULL
003	What	NULL	NULL	18

SQL query

- SELECT-FROM-WHERE statements
  - SELECT: desired attributes, expressions from the attributes
  - FROM: one or more tables
  - WHERE: condition about tuples of the table
  - Using table `Beers(name, manf)` to find all beers made by Busch.

```
SELECT name
FROM Beers
WHERE manf = 'Busch'
```

  - The answer is a relation with a single attribute name, and tuples with the name of each beer by Busch, such as Bud

Single-Relation Query

- Operations
  1. Begin with the relation in the FROM clause
  2. Apply the condition indicated by the WHERE clause to each tuple
  3. Select the fields (or expressions from the fields) indicated by the SELECT clause
- Semantics
  1. To implement this algorithm, think of a ***tuple variable*** ranging over each tuple of the relation mentioned in FROM
  2. Check if the "current" tuple satisfies the WHERE clause
  3. If so, compute the attributes or expressions of the SELECT clause using the components of this tuple

Single-relation query example

```
SELECT name
FROM Beers
WHERE manf = 'Busch'
```

Beers

Name	manf
'Bud'	'Busch'
'Miller Lite'	'MillerCoors'
'Bud Lite'	'Busch'
'Blue Moon'	'Molson Coors'
'Corona Extra'	'Constellation Brands AB InBev'
'Michelob'	'Busch'

Name	manf
'Bud'	'Busch'
'Bud Lite'	'Busch'
'Michelob'	'Busch'

Name
'Bud'
'Bud Lite'
'Michelob'

* In SELECT clauses

- When there is one relation in the FROM clause, * in the SELECT clause stands for "all attributes of this relation."
- Example using Beers(name, manf):

```
SELECT *
FROM Beers
WHERE manf = 'Busch';
```

Name	manf
'Bud'	'Busch'
'Bud Lite'	'Busch'
'Michelob'	'Busch'

Renaming Attributes

- If you want the result to have different attribute names, use "AS <new name>" to rename an attribute
- Example using Beers(name, manf):

```
SELECT name AS beer, manf
FROM Beers
WHERE manf = 'Busch';
```

Beer	manf
'Bud'	'Busch'
'Bud Lite'	'Busch'
'Michelob'	'Busch'

Expressions in SELECT clauses

- Any expression that makes sense can appear as an element of a SELECT clause
- Example: from Sells(bar, beer, price):

```
SELECT bar, beer, price * 132 AS priceInYen
FROM Sells;
```

Sells

Bar	Beer	Price
'Sue Bar'	'Bud'	1.00
'Joe Bar'	'Bud'	2.00
'Fat Daddy'	'Bud'	3.00

Bar	Beer	PriceInYen
'Sue Bar'	'Bud'	132.00
'Joe Bar'	'Bud'	264.00
'Fat Daddy'	'Bud'	396.00

Complex Conditions in WHERE Clause

- What you can use in WHERE:
  - attribute names of the relation(s) in FROM
  - comparison operators: =, <>, <, >, <=, >=
  - apply arithmetic operations: stockprice*2
  - operations on strings (e.g., "||" for concatenation)
  - Lexicographic order on strings
  - Pattern matching: s LIKE p
  - Special stuff for comparing dates and times.
  - Conditions in the WHERE clause can use AND, OR, NOT, and parentheses in the usual way

    Boolean conditions are built

Complex Conditions in WHERE Clause

- From Sells(bar, beer, price), find the price Joe's Bar charges for "cheap" beers:

```
SELECT price
FROM Sells
WHERE bar = 'Joe Bar' AND
          price < 5.0;
```

Pattern in WHERE clause

- WHERE clauses can have conditions in which a string is compared with a pattern, to see if it matches
- General form: `<Attribute> LIKE <pattern>` or `<Attribute> NOT LIKE <pattern>`
  - Pattern is a quoted string with % = “any string”; _ = “any character.”
    - From Drinkers(name, addr, phone) find the drinkers with exchange 555:

```
SELECT name
FROM Drinkers
WHERE phone LIKE '%555-_ _ _ _';
```

NULL value

- Tuples in SQL relations can have NULL as a value for one or more components. Its Meaning depends on context; Two common cases:
  - ***Missing value*** : e.g., we know Joe's Bar has some address, but we don't know what it is
  - ***Inapplicable*** : e.g., the value of attribute *spouse* for an unmarried person
- When any value is compared with NULL, the result value is an **UNKNOWN**
- The logic of conditions in SQL is really a 3-valued logic: TRUE, FALSE, UNKNOWN
  - A query only produces a tuple in the answer if its value for the WHERE clause is **TRUE** (not FALSE or UNKNOWN)

Example

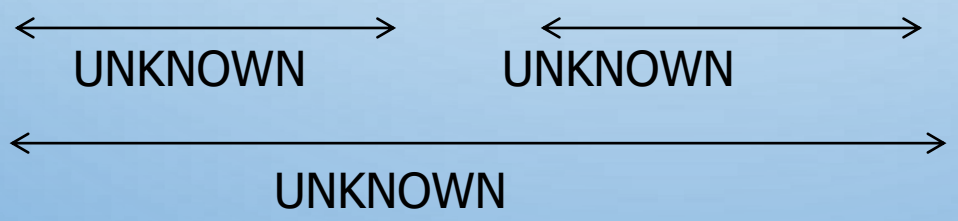
Sells

bar	beer	Price
Joe's	Bud	NULL

SELECT BAR

FROM SELLS

WHERE PRICE < 2.00 OR PRICE >= 2.00



Three-valued logic

- To understand how AND, OR, and NOT work in 3-valued logic, think of TRUE = 1, FALSE = 0, and UNKNOWN = ½, AND = MIN; OR = MAX, NOT(*x*) = 1-*x*.
- Example:

  TRUE AND (FALSE OR NOT(UNKNOWN))

  = MIN(1, MAX(0, (1 - ½)))

  = MIN(1, MAX(0, ½)

  = MIN(1, ½)

  = ½

Example

Sells (bar, beer, price)

bar	beer	Price
Joe's	Bud Lite	NULL
Joe's	Bud	NULL
Big Mama	Bud	2.00

```
SELECT bar
FROM Sells
WHERE beer = 'Bud' OR price < 3.00
```

Explicit test for NULL

- Can test for NULL explicitly:
  - x IS NULL
  - x IS NOT NULL

```
SELECT *
FROM   Person
WHERE  age < 25  OR  age >= 25 OR age IS NULL
```

Exercise

- From Sells(bar, beer, price), find the bars that sell ‘Bud Lite’ for less than 4 dollars.

Exercise

- From Students(Sid, SSN, LName, FName, Phone, GPA), find the student id, last name, first name, and GPA of all students whose GPA is above 3.5 and whose phone number has an area code of 850 (phone number format: xxx-xxx-xxxx).

Multi-relation Queries

- Interesting queries often combine data from more than one relation, we can address several relations in one query by listing them all in the FROM clause.
- Basic multi-relation queries:
  - The Cartesian product of the multiple relations in the FROM clause:
  - Use relations **Likes**(drinker, beer) and **Frequents**(drinker, bar) for the following query. Let **Likes** have 4 tuples and **Frequents** have 6 tuples, how many columns are in the output of the following query? How many tuples?

```
SELECT *
FROM    Likes, Frequents
```

Multi-relation Queries

- Attributes in different relations may have the same name. Distinguish attributes of the same name by "<relation>.<attribute>"
- Using relations **Likes**(drinker, beer) and **Frequents**(drinker, bar) for the following query. Let **Likes** have 4 tuples and **Frequents** have 6 tuples, how many columns are in the output of the following query? How many tuples?

```
SELECT Likes.drinker, Likes.beer
FROM   Likes, Frequents
```

Multi-relation Queries

- **Query:** Using relations **Likes**(drinker, beer) and **Frequents**(drinker, bar), find the beers liked by at least one person who frequents Joe's Bar.

```
SELECT Likes.beer
FROM   Likes, Frequents
WHERE  Frequents.bar = 'Joe''s' AND
       Frequents.drinker = Likes.drinker;
```

Semantics

- Almost the same as for single-relation queries:
  1. Start with the (Cartesian) product of all the relations in the FROM clause
  2. Apply the selection condition from the WHERE clause
  3. Project onto the list of attributes and expressions in the SELECT clause

SELECT a1, a2, …, ak
FROM R_1 AS x_1, R_2 AS x_2, …, R_n AS x_n
WHERE Conditions

Semantics

SELECT a1, a2, …, ak
FROM R_1 AS x_1, R_2 AS x_2, …, R_n AS x_n
WHERE Conditions

```
Answer = {}
for x1 in R1 do
    for x2 in R2 do
        …..
            for xn in Rn do
                if Conditions
                    then Answer = Answer U {(a1,…,ak)}
return Answer
```

Multi-relation queries exercise

- **Query:** Using relations **Likes**(drinker, beer) and **Sells**(bar, beer, price), find the bars that sell beers that Sally likes.

Explicit Tuple Variables

- Sometimes, a query needs to use two copies of the same relation
  - Distinguish copies by following the relation name with the name of a tuple-variable, in the FROM clause
  - It's always an option to rename relations this way, even when not essential

```
SELECT s1.bar, s1.beer, s1.price
FROM Sells s1, Sells s2
WHERE s1.beer = s2.beer AND
      s1.price < s2.price;
```

Exercise

- From students(Sid, SSN, LName, FName, Phone, GPA) and takes(Sid, course), what does the following query do?

```
SELECT t.course
FROM students s, takes t
WHERE s.FName = 'Ashley' AND s.LName = 'Brown' AND
      s.sid = t.sid;
```

SubQueries

- A parenthesized SELECT-FROM-WHERE statement (*subquery*) can be used as a value in a number of places, including FROM and WHERE clauses
  - Example: in place of a relation in the FROM clause, we can place another query, and then query its result
    - ❖ Better use a tuple-variable to name tuples of the result
- Subqueries that return Scalar
  - If a subquery is guaranteed to produce **one tuple with one component,** then the subquery can be used as a value
    - ❖ "Single" tuple often guaranteed by the key constraint
    - ❖ A run-time error occurs if there is no tuple or more than one tuple

Subquery example

- From Sells(bar, beer, price), find the bars that serve Miller for the same price Joe charges for Bud
  1. Find the price Joe charges for Bud
  2. Find the bars that serve Miller at that price

```
SELECT bar
FROM Sells
WHERE beer = 'Miller' AND
        price = (SELECT price
                 FROM Sells
                 WHERE bar = 'Joe Bar'
                  AND beer = 'Bud')
```

The IN Operator

- <tuple> IN <relation> is true if and only if the tuple is a member of the relation
  - <tuple> NOT IN <relation> means the opposite
  - IN-expressions can appear in WHERE clauses
  - The <relation> is often a subquery

Query: From Beers(name, manf) and Likes(drinker, beer), find the name and manufacturer of each beer that Sally likes

```
SELECT *
  FROM Beers
  WHERE name IN (     SELECT beer
                      FROM Likes
                      WHERE drinker = 'Sally'
                       );
```

Exercise

- **Query:** Using relations **Likes**(drinker, beer) and **Frequents**(drinker, bar), **Sells**(bar, beer, price), find the beers that some Joe's frequent customers like, but not sold by Joe's.

Exercise

- **Query:** Using relations **Likes**(drinker, beer) and **Frequents**(drinker, bar), **Sells**(bar, beer, price), find the beers that some Joe's frequent customers like, but not sold by Joe's.

```
SELECT Likes.beer
FROM Likes, Frequents
WHERE Likes.drinker = Frequents.drinker AND
      frequents.bar = 'Joe''s'
      AND likes.beer NOT IN (SELECT beer
                             FROM Sells
                             WHERE bar = 'Joe''s');
```

The EXISTS Operator

- EXISTS(<relation>) is true if and only if the <relation> is not empty
  - Being a Boolean-valued operator, EXISTS can appear in WHERE clauses

Query: From Beers(name, manf), find those beers that are the only beer by their manufacturer

```
SELECT name
  FROM Beers b1
  WHERE NOT EXISTS(
      SELECT *
      FROM Beers
      WHERE manf = b1.manf AND
            name <> b1.name);
```

Scope rule: manf refers to closest nested FROM with a relation having that attribute.

The Operator ANY

- *x* = ANY(<relation>) is a Boolean condition meaning that *x* equals at least one tuple in the relation
- Similarly, = can be replaced by any of the comparison operators
  - Example: *x* >= ANY(<relation>) means *x* is not smaller than some tuples in the relation
  - Note tuples must have one component only

The Operator ALL

- *x* <> ALL(<relation>) is true if and only if for every tuple *t* in the relation, *x* is not equal to *t*
  - That is, *x* is not a member of the relation.
- The <> can be replaced by any comparison operator
  - Example: *x* >= ALL(<relation>) means there is no tuple larger than *x* in the relation

Query: From Sells(bar, beer, price), find the beer(s) sold for the highest price

```
SELECT beer
FROM Sells
WHERE price >= ALL(
                    SELECT price
                    FROM Sells);
```

Exercise

- From Sells(bar, beer, price), find the bar that sells the cheapest Bud Lite.

Exercise

- From Sells(bar, beer, price), find the bar that sells the cheapest Bud Lite.

```
Select bar
From Sells
Where beer = 'Bud Lite' and
        price <= All(Select price
                     from Sells
                     where beer = 'Bud Lite')
```

Set operation over queries

- The default for union, intersection, and difference is set semantics, and is expressed by the following forms, each involving subqueries:
  - (subquery) UNION (subquery)
  - (subquery) INTERSECT (subquery)
  - (subquery) EXCEPT (subquery)

  - The result is a set: duplications are removed.

Example

- Happy Drinker: From relations Likes(drinker, beer), Sells(bar, beer, price) and Frequents(drinker, bar), find the drinkers and beers such that:
  1. The drinker likes the beer, and
  2. The drinker frequents at least one bar that sells the beer

```
(SELECT * FROM Likes)
 INTERSECT
(SELECT drinker, beer
 FROM Sells, Frequents
 WHERE Frequents.bar = Sells.bar
);
```

Bag semantics and set semantics

- The SELECT-FROM-WHERE statement uses bag semantics
  - **Selection:** preserve the number of occurrences
  - **Projection:** preserve the number of occurrences (no duplicate elimination)
  - **Cartesian product, join:** no duplicate elimination

Sells

Bar	Beer	Price
'Sue Bar'	'Bud'	2.00
'Joe Bar'	'Bud'	2.00
'Fat Daddy'	'Bud'	3.00

```
SELECT Price
FROM Sells
```

Price
2.00
2.00
3.00

Controlling Duplicate Elimination

- Force the result to be a set by SELECT DISTINCT
  - From Sells(bar, beer, price), find all the different prices charged for beers:

```
SELECT DISTINCT price
FROM Sells;
```

- Force the result to be a bag (i.e., don't eliminate duplicates) by ALL, as in . . . UNION ALL . . .
  - Lists drinkers who frequent more bars than they like beers, and does so as many times as the difference of those counts

```
(SELECT drinker FROM Frequents)
  INTERCEPT ALL
(SELECT drinker FROM Likes);
```

Aggregations

- SUM, AVG, COUNT, MIN, and MAX can be applied to a column in a SELECT clause to produce that aggregation on the column
  - e.g. COUNT(*) counts the number of tuples
- Query: From Sells(bar, beer, price), find the average price of Bud

```
SELECT AVG(price)
FROM Sells
WHERE beer = 'Bud'
```

Eliminating Duplicates in an Aggregation

- DISTINCT inside an aggregation causes duplicates to be eliminated before the aggregation
- Query: find the number of different prices charged for Bud:

```
SELECT COUNT(DISTINCT price)
FROM Sells
WHERE beer = 'Bud';
```

NULL's in Aggregation

- NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column
  - The aggregation function ignore the rows with NULL values.
- But if there are no non-NULL values in a column, then the result of the aggregation is NULL

```
Select count(*)
From Sells
Where beer = 'Bud'
```

The number of bars that sell Bud

```
Select count(price)
From Sells
Where beer = 'Bud'
```

The number of bars that sell Bud at a known price

Group By

- We may follow a SELECT-FROM-WHERE expression by GROUP BY and a list of attributes
  - The relation that results from the SELECT-FROM-WHERE is grouped according to the values of all those attributes, and any aggregation is applied only within each group
- Query: From Sells(bar, beer, price), find the average price for each beer:

```
SELECT beer, AVG(price)
FROM Sells
GROUP BY beer
```

Example

- Query: From Sells(bar, beer, price) and Frequents (drinker, bar), find for each drinker the average price of Bud at the bars they frequent:

```
SELECT drinker, AVG(price)
FROM Frequents, Sells
WHERE beer = 'Bud' AND
      Frequents.bar = Sells.bar
GROUP BY drinker;
```

Compute drinker-bar-price of Bud tuples first, then group by drinker

Modifications

- A modification command does NOT return a result as a query does, but it changes the database in some way
- There are three kinds of modifications:
  1. **Insert** a tuple or tuples
  2. **Delete** a tuple or tuples
  3. **Update** the value(s) of an existing tuple or tuples

Insertion

- To insert a single tuple:

  INSERT INTO <relation>

  VALUES (<list of values>);

- Example: add to Likes(drinker, beer) the fact that Sally likes Bud:

```
INSERT INTO Likes
VALUES('Sally', 'Bud');
```

Specifying Attributes in INSERT

- We may add to the relation a list of attributes
- There are two reasons to do so:
  1. We forget the standard order of attributes for the relation
  2. We don't have values for all attributes, and we want the system to fill in missing components with NULL or default values
- Another way to add the fact that Sally likes Bud to Likes(drinker, beer):

```
INSERT INTO Likes(beer, drinker)
VALUES('Bud', 'Sally');
```

Inserting Many Tuples

- We may insert the entire result of a query into a relation, using the form:

  INSERT INTO <relation>

  (<subquery>);

E.g.,

```
INSERT INTO Beers(name)
        (SELECT beer from Sells);
```

Example: Insert a Subquery

- Using Frequents(drinker, bar), enter into the new relation PotBuddies (name) all of Sally's "potential buddies," i.e., those drinkers who frequent at least one bar that Sally also frequents

The other drinker

Pairs of Drinker tuples where the first is for Sally, the second is for someone else, and the bars are the same

```
INSERT INTO PotBuddies
(SELECT d2.drinker
 FROM Frequents d1, Frequents d2
 WHERE d1.drinker = 'Sally' AND
   d2.drinker <> 'Sally' AND
   d1.bar = d2.bar
);
```

Deletion

- To delete tuples satisfying a condition from some relation:

  DELETE FROM <relation>

  WHERE <condition>;

- Example: Delete from Likes(drinker, beer) the fact that Sally likes Bud:

```
DELETE FROM Likes
WHERE drinker = 'Sally' AND
      beer = 'Bud';
```

Delete all Tuples

- Make the relation Likes empty:

```
DELETE FROM Likes;
```

- Note no WHERE clause needed

Delete Many Tuples

- Delete from Beers(name, manf) all beers for which there is another beer by the same manufacturer.

```
DELETE FROM Beers b
WHERE EXISTS
      (
       SELECT name FROM Beers a
       WHERE a.manf = b.manf AND
       a.name <> b.name
      );
```

Beers with the same manufacturer and a different name from the name of the beer represented by tuple b

Updates

- To change certain attributes in certain tuples of a relation:

  UPDATE <relation>

  SET <list of attribute assignments>

  WHERE <condition on tuples>;

- Example: Change drinker Fred's phone number to 555-1212:

```
UPDATE Drinkers
SET phone = '555-1212'
WHERE name = 'Fred';
```

Update Several Tuples

- Increase price that is cheap:

```
UPDATE Sells
SET price = price * 1.07
WHERE price < 3.0;
```