

# CS411

## Database Systems

06a: SQL-1

The Basics— Select-From-Where

# Why Do We Learn This?

# SQL Introduction

Standard language for querying and manipulating data

**Structured Query Language**

Many standards out there: SQL92, SQL2, SQL3, SQL99

Vendors support various subsets of these, but all of what we'll be talking about.

# Why SQL?

- SQL is a very-high-level language, in which the programmer is able to avoid specifying a lot of data-manipulation details that would be necessary in languages like C++.
- What makes SQL viable is that its queries are “optimized” quite well, yielding efficient query executions.

# Select-From-Where Statements

- The principal form of a query is:

SELECT	desired attributes
FROM	one or more tables
WHERE	condition about tuples of the tables

# Single-Relation Queries

# Our Running Example

- Most of our SQL queries will be based on the following database schema.
  - Underline indicates key attributes.

Beers(name, manf)

Bars(name, addr, license)

Drinkers(name, addr, phone)

Likes(drinker, beer)

Sells(bar, beer, price)

Frequents(drinker, bar)

# Example

- Using Beers(name, manf), what beers are made by Anheuser-Busch?

```
SELECT name
```

```
FROM Beers
```

```
WHERE manf = 'Anheuser-Busch' ;
```



# Result of Query

name
'Bud'
'Bud Lite'
'Michelob'

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

# Meaning of Single-Relation Query

- Begin with the relation in the FROM clause.
- Apply the selection indicated by the WHERE clause.
- Apply the extended projection indicated by the SELECT clause.

# Operational Semantics

- To implement this algorithm think of a *tuple variable* ranging over each tuple of the relation mentioned in FROM.
- Check if the “current” tuple satisfies the WHERE clause.
- If so, compute the attributes or expressions of the SELECT clause using the components of this tuple.

## \* 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 = 'Anheuser-Busch' ;
```

## Result of Query:

name	manf
'Bud'	'Anheuser-Busch'
'Bud Lite'	'Anheuser-Busch'
'Michelob'	'Anheuser-Busch'

Now, the result has each of the attributes of Beers.

# Renaming Attributes

- If you want the result to have different attribute names, use “AS <new name>” to rename an attribute.
- Example based on Beers(name, manf):

```
SELECT name AS beer, manf
```

```
FROM Beers
```

```
WHERE manf = 'Anheuser-Busch'
```

## Result of Query:

beer	manf
'Bud'	'Anheuser-Busch'
'Bud Lite'	'Anheuser-Busch'
'Michelob'	'Anheuser-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 * 120 AS priceInYen  
FROM Sells;
```



# Result of Query

bar	beer	priceInYen
Joe's	Bud	300
Sue's	Miller	360
...	...	...

# Complex Conditions in WHERE Clause

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

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

# Selections

## What you can use in WHERE:

attribute names of the relation(s) used in the 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.

# Important Points

- Two single quotes inside a string represent the single-quote (apostrophe).
- Conditions in the WHERE clause can use AND, OR, NOT, and parentheses in the usual way boolean conditions are built.
- SQL is *case-insensitive*. In general, upper and lower case characters are the same, except inside quoted strings.

# Patterns

- 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.”

# Example

- From Drinkers(name, addr, phone) find the drinkers with exchange 555:

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

# Motivating Example for Next Few Slides

- From the following Sells relation:

bar	beer	price
....	....	...

```
SELECT bar
FROM Sells
WHERE price < 2.00 OR price >= 2.00;
```

# Null Values



# NULL Values

- Tuples in SQL relations can have NULL as a value for one or more components.
- 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.

# Comparing NULL's to Values

- The logic of conditions in SQL is really 3-valued logic: TRUE, FALSE, UNKNOWN.
- When any value is compared with NULL, the truth value is UNKNOWN.
- But a query only produces a tuple in the answer if its truth value for the WHERE clause is TRUE (not FALSE or UNKNOWN).

# Three-Valued Logic

- To understand how AND, OR, and NOT work in 3-valued logic, think of TRUE = 1, FALSE = 0, and UNKNOWN =  $\frac{1}{2}$ .
- AND = MIN; OR = MAX, NOT( $x$ ) =  $1-x$ .
- Example:

TRUE AND (FALSE OR

NOT(UNKNOWN)) = MIN(1, MAX(0, (1 -  $\frac{1}{2}$  ))) =

MIN(1, MAX(0,  $\frac{1}{2}$  )) = MIN(1,  $\frac{1}{2}$  ) =  $\frac{1}{2}$ .

# Surprising Example

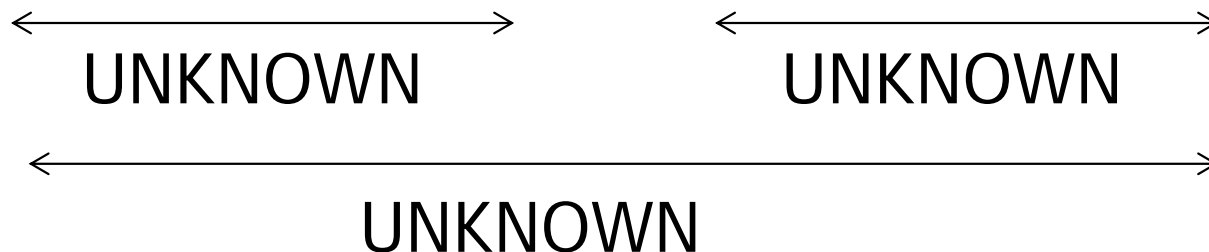
- From the following Sells relation:

bar	beer	price
Joe's Bar	Bud	NULL

SELECT bar

FROM Sells

WHERE price < 2.00 OR price >= 2.00;



# Reason: 2-Valued Laws $\neq$ 3-Valued Laws

- Some common laws, like the commutativity of AND, hold in 3-valued logic.
- But others do not; example: the “law of excluded middle,”  $p \text{ OR NOT } p = \text{TRUE}$ .
  - When  $p = \text{UNKNOWN}$ , the left side is  $\text{MAX}( \frac{1}{2}, (1 - \frac{1}{2}) ) = \frac{1}{2} \neq 1$ .

# Testing 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
```

Now it includes all Persons

# Multi-Relation Queries

# 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.
- Distinguish attributes of the same name by “<relation>.<attribute>”



# Example

- Using relations Likes(drinker, beer) and Frequents(drinker, bar), find the beers liked by at least one person who frequents Joe's Bar.
- Tip: Always prefix with relation name to make it clear/easier to read.

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

# Another Example

Product (pname, price, category, maker)

Purchase (buyer, seller, store, product)

Company (cname, stockPrice, country)

Person(pname, phoneNumber, city)

Find names of people living in Champaign that bought gizmo products, and the names of the stores they bought from

```
SELECT  pname, store
FROM    Person, Purchase
WHERE   pname=buyer AND city="Champaign"
        AND product="gizmo"
```

# Disambiguating Attributes

Find names of people buying telephony products:

Product (name, price, category, maker)

Purchase (buyer, seller, store, product)

Person(name, phoneNumber, city)

```
SELECT Person.name
FROM   Person, Purchase, Product
WHERE  Person.name=Purchase.buyer
      AND Purchase.product=Product.name
      AND Product.category="telephony"
```

# Semantics

- Almost the same as for single-relation queries:
  1. Start with the 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.

# Explicit Tuple-Variables

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

# Example

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

# Meaning (Semantics) of SQL Queries

**SELECT** a1, a2, ..., ak  
**FROM** R1 AS x1, R2 AS x2, ..., Rn AS xn  
**WHERE** Conditions

1. Nested loops:

```
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
```

# Meaning (Semantics) of SQL Queries

**SELECT** a1, a2, ..., ak  
**FROM** R1 AS x1, R2 AS x2, ..., Rn AS xn  
**WHERE** Conditions

3. Translation to Relational algebra:

$\Pi_{a1, \dots, ak} ( \sigma_{\text{Conditions}} (R1 \times R2 \times \dots \times Rn) )$

Select-From-Where queries are precisely Select-Project-Join



# Exercises

Product ( pname, price, category, maker)

Purchase (buyer, seller, store, product)

Company (cname, stock price, country)

Person( per-name, phone number, city)

**Ex #1:** Find people who bought telephony products.

**Ex #2:** Find names of people who bought American products

**Ex #3:** Find names of people who bought American products and did not buy French products

**Ex #4:** Find names of people who bought American products and they live in Champaign.

**Ex #5:** Find people who bought stuff from Joe or bought products from a company whose stock prices is more than \$50.

# Behind the Scene: The Birth of SQL

- Chamberlin, D. D. and Boyce, R. F. 1974. SEQUEL: A structured English query language. In Proceedings of the 1974 ACM SIGFIDET (Now Sigmod) Workshop on Data Description, Access and Control (Ann Arbor, Michigan, May 01 - 03, 1974). FIDET '74. ACM, New York, NY, 249-264.

SEQUEL: A STRUCTURED ENGLISH QUERY LANGUAGE

by

Donald D. Chamberlin  
Raymond F. Boyce

IBM Research Laboratory  
San Jose, California

**ABSTRACT:** In this paper we present the data manipulation facility for a structured English query language (SEQUEL) which can be used for accessing data in an integrated relational data base. Without resorting to the concepts of bound variables and quantifiers SEQUEL identifies a set of simple operations on tabular structures, which can be shown to be of equivalent power to the first order predicate calculus. A SEQUEL user is presented with a consistent set of keyword English templates which reflect how people use tables to obtain information. Moreover, the SEQUEL user is able to compose these basic templates in a structured manner in order to form more complex queries. SEQUEL is intended as a data base sublanguage for both the professional programmer and the more infrequent data base user.

# The very FIRST SQL

As in SQUARE, the simplest SEQUEL expression is a mapping which specifies a table, a domain, a range, and an argument, as illustrated by Q1.

Q1. Find the names of employees in the toy department.

```
SELECT    NAME
FROM      EMP
WHERE     DEPT = 'TOY'
```

The mapping returns the entire set of names which qualify according to the test `DEPT = 'TOY'`.

# The first JOIN SQL

name. This is illustrated by Q10, which implements what Codd (7) would call a "join" between the SALES and SUPPLY tables on their ITEM columns:

Q10. List rows of SALES and SUPPLY concatenated together whenever their ITEM values match.

```
SALES, SUPPLY
WHERE SALES . ITEM = SUPPLY . ITEM
```

# Subqueries

Boolean Operators IN, EXISTS,  
ANY, ALL

# 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 key constraint.
  - A run-time error occurs if there is no tuple or more than one tuple.

# Example

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

# Query + Subquery Solution

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



# The IN Operator

- $\langle \text{tuple} \rangle \text{ IN } \langle \text{relation} \rangle$  is true if and only if the tuple is a member of the relation.
  - $\langle \text{tuple} \rangle \text{ NOT IN } \langle \text{relation} \rangle$  means the opposite.
- IN-expressions can appear in WHERE clauses.
- The  $\langle \text{relation} \rangle$  is often a subquery.

# Example

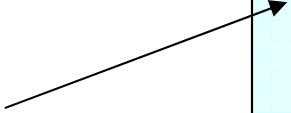
- From Beers(name, manf) and Likes(drinker, beer), find the name and manufacturer of each beer that Fred likes.

SELECT \*

FROM Beers

WHERE name IN (SELECT beer  
FROM Likes  
WHERE drinker = 'Fred');

The set of  
beers Fred  
likes



# 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.
- Example: From Beers(name, manf), find those beers that are the only beer by their manufacturer.

# Example Query with EXISTS

SELECT name

FROM Beers b1

WHERE NOT EXISTS(  
SELECT \*

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

SELECT \*

FROM Beers

WHERE manf = b1.manf AND

name <> b1.name);

Set of beers with the same manf as b1, but not the same beer

Notice the SQL "not equals" operator

# The Operator ANY

- $x = \text{ANY}(\langle \text{relation} \rangle)$  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:  $x \geq \text{ANY}(\langle \text{relation} \rangle)$  means  $x$  is not smaller than all tuples in the relation.
  - Note tuples must have one component only.

# The Operator ALL

- Similarly,  $x \neq \text{ALL}(\langle \text{relation} \rangle)$  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  $\neq$  can be replaced by any comparison operator.
- Example:  $x \geq \text{ALL}(\langle \text{relation} \rangle)$  means there is no tuple larger than  $x$  in the relation.

# Example

- 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);

price from the outer  
Sells must not be  
less than any price.

# Relations as Bags



# Relational Algebra: Operations on Bags (and why we care)

- Union:  $\{a,b,b,c\} \cup \{a,b,b,b,e,f,f\} = \{a,a,b,b,b,b,c,e,f,f\}$ 
  - *add* the number of occurrences
- Difference:  $\{a,b,b,b,c,c\} - \{b,c,c,c,d\} = \{a,b,b\}$ 
  - subtract the number of occurrences
- Intersection:  $\{a,b,b,b,c,c\} \cap \{b,b,c,c,c,c,d\} = \{b,b,c,c\}$ 
  - minimum of the two numbers of occurrences
- Selection: preserve the number of occurrences
- Projection: preserve the number of occurrences (no duplicate elimination)
- Cartesian product, join: no duplicate elimination

Read Section 5.3 of the book for more detail

# Bag Semantics for SFW Queries

- 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

# Union, Intersection, and Difference

- Union, intersection, and difference of relations are expressed by the following forms, each involving subqueries:
  - ( subquery ) UNION ( subquery )
  - ( subquery ) INTERSECT ( subquery )
  - ( subquery ) EXCEPT ( subquery )

# Example

- 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.

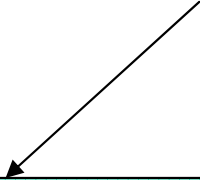
# Solution

(SELECT \* FROM Likes)

INTERSECT

(SELECT drinker, beer  
FROM Sells, Frequents  
WHERE Frequents.bar = Sells.bar  
);

The drinker frequents  
a bar that sells the  
beer.



# Bag Semantics for Set Operations in SQL

- Although the SELECT-FROM-WHERE statement uses bag semantics, the default for union, intersection, and difference is set semantics.
  - That is, duplicates are eliminated as the operation is applied.

# Motivation: Efficiency

- When doing projection in relational algebra, it is easier to avoid eliminating duplicates.
  - Just work tuple-at-a-time.
- When doing intersection or difference, it is most efficient to sort the relations first.
  - At that point you may as well eliminate the duplicates anyway.

# Controlling Duplicate Elimination

- Force the result to be a set by  
`SELECT DISTINCT . . .`
- Force the result to be a bag (i.e., don't eliminate  
duplicates) by `ALL`, as in `. . . UNION ALL .`  
`. .`



## Example: DISTINCT

- From Sells(bar, beer, price), find all the different prices charged for beers:

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

- Notice that without DISTINCT, each price would be listed as many times as there were bar/beer pairs at that price.

# Example: ALL

- Using relations `Frequents(drinker, bar)` and `Likes(drinker, beer)`:

```
(SELECT drinker FROM Frequents)
```

```
EXCEPT ALL
```

```
(SELECT drinker FROM Likes);
```

- Lists drinkers who frequent more bars than they like beers, and does so as many times as the difference of those counts.