

# SQL Queries

With slides by Ph.Bonnet, J.Ulmann, M.Kifer

# Outline

- SELECT block
  - SELECT-FROM-WHERE
  - Three-valued logic
  - Aggregation
- Subqueries
  - In, =, exists, all, any
- Assertions
  - Queries as constraints

# Why SQL?

- SQL is a very-high-level language.
  - Say “what to do” rather than “how to do it.”
  - Avoid a lot of data-manipulation details needed in procedural languages like C++ or Java.
- Database management system figures out “best” way to execute query.
  - Called “query optimization.”

# Select-From-Where Statements

**SELECT** desired attributes

**FROM** one or more tables

**WHERE** condition about tuples of  
the tables

# Our Running Example

- All 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

name	manf
Bud	Anheuser-Busch

Include  $t.name$   
in the result, if so

Check if  
Anheuser-Busch

Tuple-variable  $t$   
loops over all  
tuples

# Operational Semantics --- General

- Think of a *tuple variable* visiting 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:** Using **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:** Using **Sells(bar, beer, price):**

```
SELECT bar, beer,  
       price*114 AS priceInYen  
FROM Sells;
```

# Result of Query

bar	beer	priceInYen
Joe's	Bud	285
Sue's	Miller	342
...	...	...

# Constants as Expressions

- Using `Likes(drinker, beer)`:

```
SELECT drinker,  
       'likes Bud' AS whoLikesBud  
FROM Likes  
WHERE beer = 'Bud';
```

# Result of Query

drinker	whoLikesBud
Sally	likes Bud
Fred	likes Bud
...	...

# Information Integration

- We often build “data warehouses” from the data at many “sources.”
- Suppose each bar has its own relation **Menu(beer, price)** .
- To contribute to **Sells(bar, beer, price)** we need to query each bar and insert the name of the bar.

# Information Integration

- For instance, at Joe's Bar we can issue the query:

```
SELECT 'Joe''s Bar', beer, price  
FROM Menu;
```

# Conditions in WHERE Clause

- Boolean operators AND, OR, NOT.
- Comparisons =, <>, <, >, <=, >=.
  - And many other operators that produce boolean-valued results.

# Complex Conditions

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

```
SELECT price
FROM Sells
WHERE bar = 'Joe''s Bar' AND
       beer = 'Bud';
```

# Patterns

- A condition can compare a string to a pattern by:
  - <Attribute> LIKE <pattern> or <Attribute> NOT LIKE <pattern>
- *Pattern* is a quoted string with % = “any string”; \_ = “any character.”

# LIKE

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

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

# 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 Values

- The logic of conditions in SQL is really 3-valued logic: TRUE, FALSE, UNKNOWN.
- Comparing any value (including NULL itself) with NULL yields UNKNOWN.
- A tuple is in a query answer iff 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:**

$$\begin{aligned} \text{TRUE AND (FALSE OR NOT(UNKNOWN))} &= \\ \text{MIN(1, MAX(0, (1 - } \frac{1}{2} \text{)))} &= \\ \text{MIN(1, MAX(0, } \frac{1}{2} \text{))} &= \text{MIN(1, } \frac{1}{2} \text{)} = \frac{1}{2}. \end{aligned}$$

# 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;

← UNKNOWN →                      ← UNKNOWN →

← UNKNOWN →

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

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

# Multi-block queries

- Each Select-from-where is called a block
- Multi-block queries to express set-oriented operations:
  - Union: keyword UNION between blocks
  - Intersection: keyword INTERSECT between blocks
  - Set difference: keyword EXCEPT between blocks

# Multirelation 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>” .

# Joining Two Relations

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

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

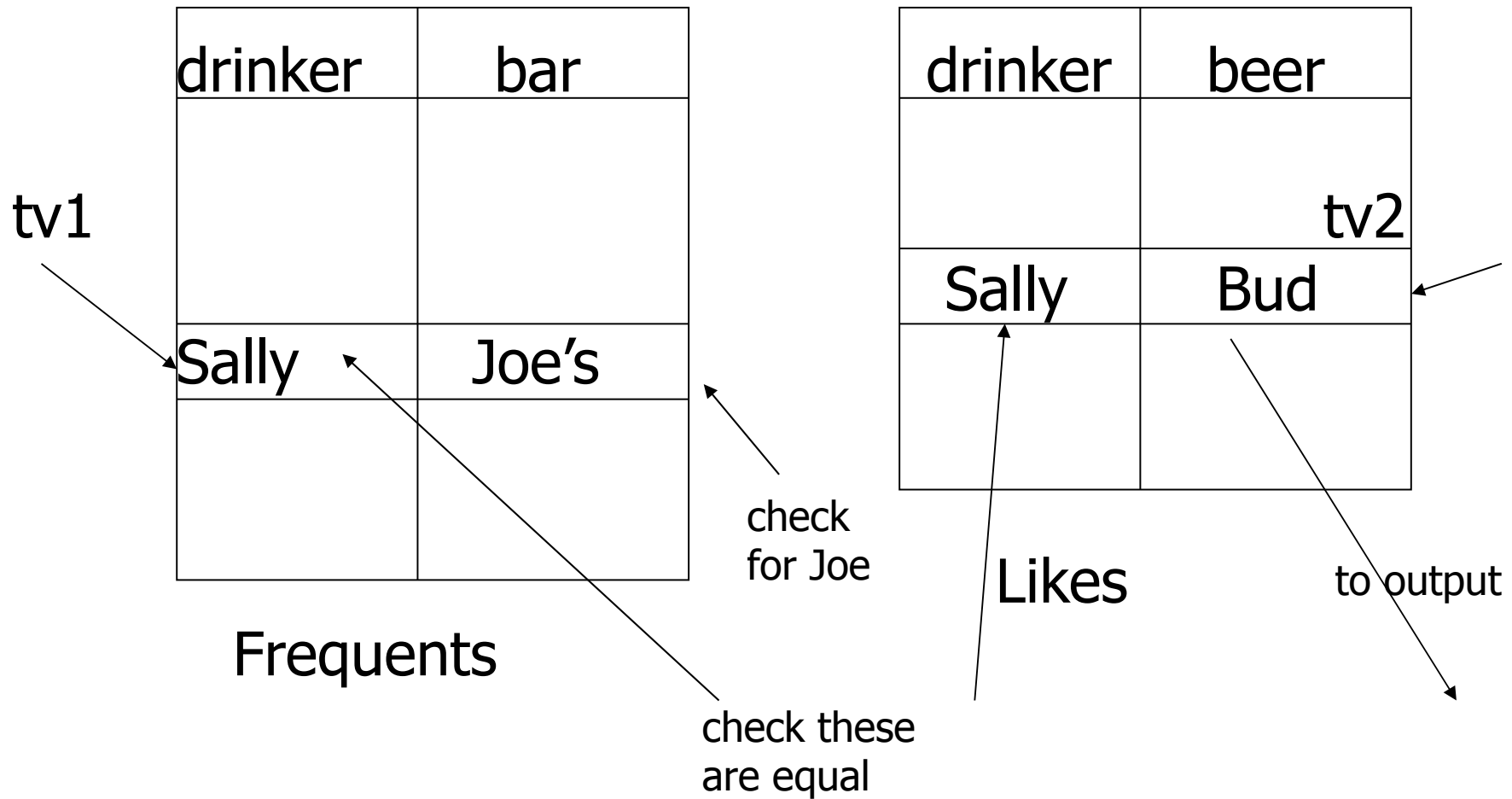
# Formal Semantics

- ◆ 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.

# Operational Semantics

- Imagine one tuple-variable for each relation in the FROM clause.
  - These tuple-variables visit each combination of tuples, one from each relation.
- If the tuple-variables are pointing to tuples that satisfy the WHERE clause, send these tuples to the SELECT clause.

# Example



# 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.

# Self-Join

- From **Beers(name, manf)**, find all pairs of beers by the same manufacturer.
  - Do not produce pairs like (Bud, Bud).
  - Produce pairs in alphabetic order, e.g. (Bud, Miller), not (Miller, Bud).

```
SELECT b1.name, b2.name
FROM Beers b1, Beers b2
WHERE b1.manf = b2.manf AND
      b1.name < b2.name;
```

# Bag Semantics

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

# 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.

# Join Expressions

- SQL provides several versions of (bag) joins.
- These expressions can be stand-alone queries or used in place of relations in a FROM clause.

# Products and Natural Joins

- Natural join:  
R NATURAL JOIN S;
- Product:  
R CROSS JOIN S;
- **Example:**  
Likes NATURAL JOIN Sells;
- Relations can be parenthesized subqueries, as well.

# Theta Join

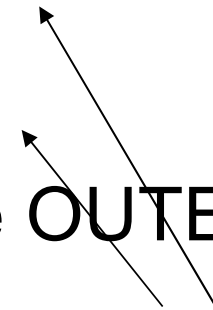
- R JOIN S ON <condition>
- **Example:** using **Drinkers(name, addr)** and **Frequents(drinker, bar):**

```
Drinkers JOIN Frequents ON  
    name = drinker;
```

gives us all  $(d, a, d, b)$  quadruples such that drinker  $d$  lives at address  $a$  and frequents bar  $b$ .

# Outerjoins

- ◆ R OUTER JOIN S is the core of an outerjoin expression. It is modified by:
  1. Optional NATURAL in front of OUTER.
  2. Optional ON <condition> after JOIN.
  3. Optional LEFT, RIGHT, or FULL before OUTER.
    - ◆ LEFT = **pad dangling tuples** of R only.
    - ◆ RIGHT = **pad dangling tuples** of S only.
    - ◆ FULL = **pad** both; this choice is the default.



Only one  
of these

# Sorting

- Clause for sorting the result set

```
SELECT DISTINCT price  
FROM Sells  
SORT BY price DESC;
```

```
SELECT DISTINCT price  
FROM Sells  
SORT BY bar ASC, price DESC;
```

# Aggregations

- SUM, AVG, COUNT, MIN, and MAX can be applied to a column in a SELECT clause to produce that aggregation on the column.
- Also, COUNT(\*) counts the number of tuples.

# Aggregation

- From **Sells(bar, beer, price)**, find the average price of Bud:

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

# Eliminating Duplicates in an Aggregation

- Use DISTINCT inside an aggregation.
- **Example**: find the number of *different* prices charged for Bud:

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

# NULL's Ignored in Aggregation

- NULL never contributes to a sum, average, or count, and can never be the minimum or maximum of a column.
- But if there are no non-NULL values in a column, then the result of the aggregation is NULL.
  - **Exception:** COUNT of an empty set is 0.

# Example: Effect of NULL's


```
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.



# Grouping

- 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.

# Example: Grouping

- From **Sells(bar, beer, price)**, find the average price for each beer:

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

beer	AVG(price)
Bud	2.33
...	...

# Example: Grouping

- 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 all drinker-bar-price triples for Bud.

Then group them by drinker.

# SELECT Lists With Aggregation

- ◆ If any aggregation is used, then each element of the SELECT list must be either:
  1. Aggregated, or
  2. An attribute on the GROUP BY list.

# Illegal Query Example

- You might think you could find the bar that sells Bud the cheapest by:

```
SELECT bar, MIN(price)
```

```
FROM Sells
```

```
WHERE beer = 'Bud';
```

- But this query is illegal in SQL.

# HAVING Clauses

- HAVING <condition> may follow a GROUP BY clause.
- If so, the condition applies to each group, and groups not satisfying the condition are eliminated.

# Example: HAVING

- From `Sells(bar, beer, price)` and `Beers(name, manf)`, find the average price of those beers that are either served in at least three bars or are manufactured by Pete's.

Beer groups with at least  
3 non-NULL bars and also  
beer groups where the  
manufacturer is Pete's.

# Solution

```
SELECT beer, AVG(price)
FROM Sells
GROUP BY beer
HAVING COUNT(bar) >= 3 OR
beer IN (SELECT name
```

```
FROM Beers
WHERE manf = 'Pete's');
```

Beers manu-  
factured by  
Pete's.

# HAVING Conditions

- ◆ Anything goes in a subquery.
- ◆ Outside subqueries, they may refer to attributes only if they are either:
  1. A grouping attribute, or
  2. Aggregated(same condition as for SELECT clauses with aggregation).

# Outline

- SELECT block
  - SELECT-FROM-WHERE
  - Three-valued logic
  - Aggregation
- Subqueries
  - In, =, exists, all, any
- Assertions
  - Queries as constraints

# Subqueries

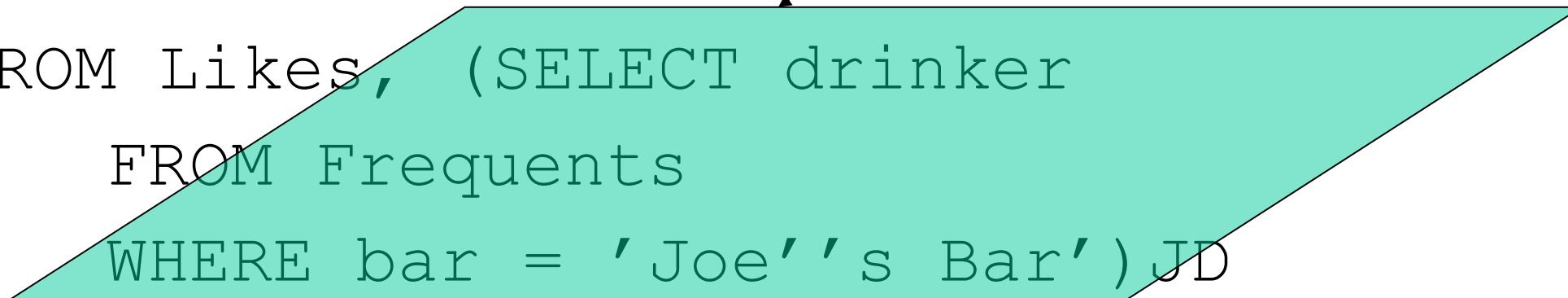
- A parenthesized SELECT-FROM-WHERE statement (*subquery*) can be used as a value in a number of places, including FROM and WHERE clauses.
- **Example:** in place of a relation in the FROM clause, we can use a subquery and then query its result.
  - Must use a tuple-variable to name tuples of the result.

# Example: Subquery in FROM

- Find the beers liked by at least one person who frequents Joe's Bar.

Drinkers who  
frequent Joe's Bar

```
SELECT beer
FROM Likes, (SELECT drinker
              FROM Frequents
              WHERE bar = 'Joe's Bar') JD
WHERE Likes.drinker = JD.drinker;
```



# Single-Tuple Subqueries

- If a subquery is guaranteed to produce one tuple, then the subquery can be used as a value (implicit type cast).
  - Usually, the tuple has one component.
  - A run-time error occurs if there is no tuple or more than one tuple
  - UGLY but practical

# Single-Tuple Subquery

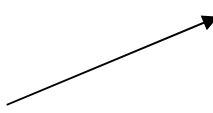
- ◆ Using `Sells(bar, beer, price)`, find the bars that serve Miller for the same price Joe charges for Bud.
- ◆ Two queries would surely work:
  1. Find the price Joe charges for Bud.
  2. Find the bars that serve Miller at that price.

# Query + Subquery Solution

```
SELECT bar  
FROM Sells  
WHERE beer = 'Miller' AND
```

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

The price at  
which Joe  
sells Bud



# The IN Operator

- `<tuple> IN (<subquery>)` is true if and only if the tuple is a member of the relation produced by the subquery.
  - Opposite: `<tuple> NOT IN (<subquery>)`.
- IN-expressions can appear in WHERE clauses.

# Example: IN

- Using **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



# Find the difference

```
SELECT a
FROM R, S
WHERE R.b = S.b;
```

```
SELECT a
FROM R
WHERE b IN (SELECT b FROM S);
```

# IN is a Predicate About R

```
SELECT a
FROM R
WHERE b IN (SELECT b FROM S);
```

Two 2's

One loop, over  
the tuples of R

a	b
1	2
3	4

R

b	c
2	5
2	6

S

(1,2) satisfies  
the condition;  
1 is output once.

# Tuples from R, S

```
SELECT a
FROM R, S
WHERE R.b = S.b;
```

Double loop, over  
the tuples of R and S

a	b
1	2
3	4

R

b	c
2	5
2	6

S

(1,2) with (2,5)  
and (1,2) with  
(2,6) both satisfy  
the condition;  
1 is output twice.

# The Exists Operator

- EXISTS(<subquery>) is true if and only if the subquery result is not empty.
- **Example:** From **Beers(name, manf)** , find those beers that are the unique beer by their manufacturer.

# Example: EXISTS

```
SELECT name  
FROM Beers b1  
WHERE NOT EXISTS (
```

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

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

```
SELECT *  
FROM Beers  
WHERE manf = b1.manf AND  
      name <> b1.name);
```

Notice the SQL "not equals" operator

# The Operator ANY

- $x = \text{ANY}(\langle \text{subquery} \rangle)$  is a boolean condition that is true iff  $x$  equals at least one tuple in the subquery result.
  - $=$  could be any comparison operator.
- **Example:**  $x \geq \text{ANY}(\langle \text{subquery} \rangle)$  means  $x$  is not the uniquely smallest tuple produced by the subquery.
  - Note tuples must have one component only.

# The Operator ALL

- $x \langle \rangle \text{ALL}(\langle \text{subquery} \rangle)$  is true iff for every tuple  $t$  in the relation,  $x$  is not equal to  $t$ .
  - That is,  $x$  is not in the subquery result.
- $\langle \rangle$  can be any comparison operator.
- **Example:**  $x \geq \text{ALL}(\langle \text{subquery} \rangle)$  means there is no tuple larger than  $x$  in the subquery result.

# Example: ALL

- 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.

# Outline

- SELECT block
  - SELECT-FROM-WHERE
  - Three-valued logic
  - Aggregation
- Subqueries
  - In, =, exists, all, any
- Assertions
  - Queries as constraints

# Assertions

- These are database-schema elements, like relations or views.
- Defined by:

```
CREATE ASSERTION <name>  
CHECK (<condition>);
```

- Condition may refer to any relation or attribute in the database schema.

# Example: Assertion

- In `Sells(bar, beer, price)`, no bar may charge an average of more than \$5.

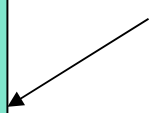
```
CREATE ASSERTION NoRipoffBars CHECK (  
  NOT EXISTS (  

```

```
    SELECT bar FROM Sells  
    GROUP BY bar  
    HAVING 5.00 < AVG(price)
```

```
  ));
```

Bars with an  
average price  
above \$5



# Example: Assertion

- In `Drinkers(name, addr, phone)` and `Bars(name, addr, license)`, there cannot be more bars than drinkers.

```
CREATE ASSERTION FewBar CHECK (  
    (SELECT COUNT(*) FROM Bars) <=  
    (SELECT COUNT(*) FROM Drinkers)  
);
```

# Timing of Assertion Checks

- In principle, we must check every assertion after every modification to any relation of the database.
- A clever system can observe that only certain changes could cause a given assertion to be violated.
  - **Example:** No change to Beers can affect FewBar. Neither can an insertion to Drinkers.
- More on this when we cover transactions

# Take Away Points

- SQL universal relational query language
  - Result set is a table
- 1 Block
  - select, from, where, group by, having, sort by clauses
- Several blocks
  - Multi-block queries
    - Blocks connected by union, except, intersect
  - Subqueries in from or where clause
    - Blocks connected by in, =, any, all, exists