Introduction to SQL

Select-From-Where Statements

Multirelation Queries

Subqueries

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)

Likes2(drinker, beer) → Likes2 !!!

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

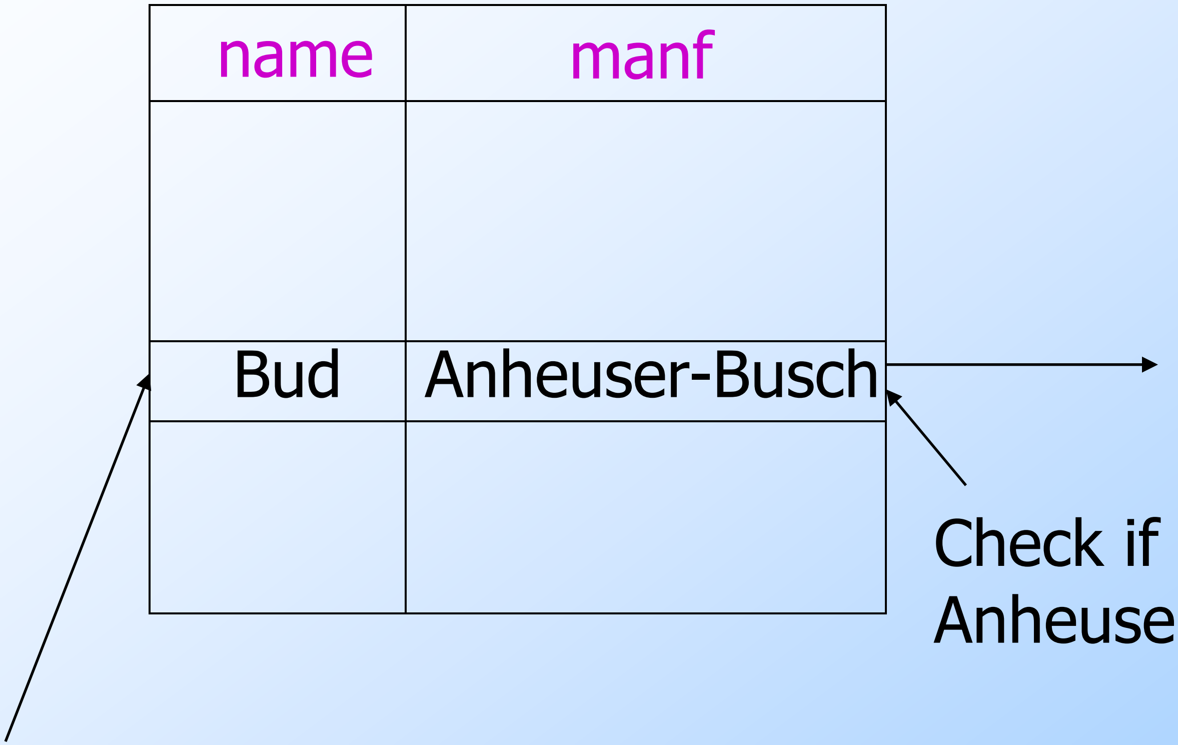
name
Bud
Bud Lite
Michelob
. . .

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
…	…	…

Example: Constants as Expressions

- Using Likes2(drinker, beer):

```
SELECT drinker,
     ’likes Bud’ AS whoLikesBud
FROM Likes2
WHERE beer = ’Bud’;
```

Result of Query

drinker	whoLikesBud
Sally	likes Bud
Fred	likes Bud
…	…

Example: 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 --- (2)

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

```
SELECT 'Joes Bar', beer, price
FROM Menu;
```

Complex Conditions in WHERE Clause

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

Example: Complex Condition

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

```
SELECT price
FROM Sells
WHERE bar = 'Joes 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."

Example: 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's to 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 = 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.

Surprising Example

- From the following Sells relation:

bar	beer	price
Joe’s Bar	Bud	NULL

SELECT bar

FROM Sells

WHERE $\underbrace{\underbrace{\text{price < 2.00}}_{\text{UNKNOWN}}\ \text{OR}\ \underbrace{\text{price >= 2.00}}_{\text{UNKNOWN}}}_{\text{UNKNOWN}}$;

Reason: 2-Valued Laws != 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 OR NOT p = TRUE.
  - When p = UNKNOWN, the left side is MAX(½, (1 – ½)) = ½ != 1.

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>" .

Example: Joining Two Relations

- Using relations Likes2(drinker, beer) and Frequents(drinker, bar), find the beers liked by at least one person who frequents Joe’s Bar.

```
SELECT beer
FROM Likes2, Frequents
WHERE bar = 'Joes Bar' AND
  Frequents.drinker =
      Likes2.drinker;
```

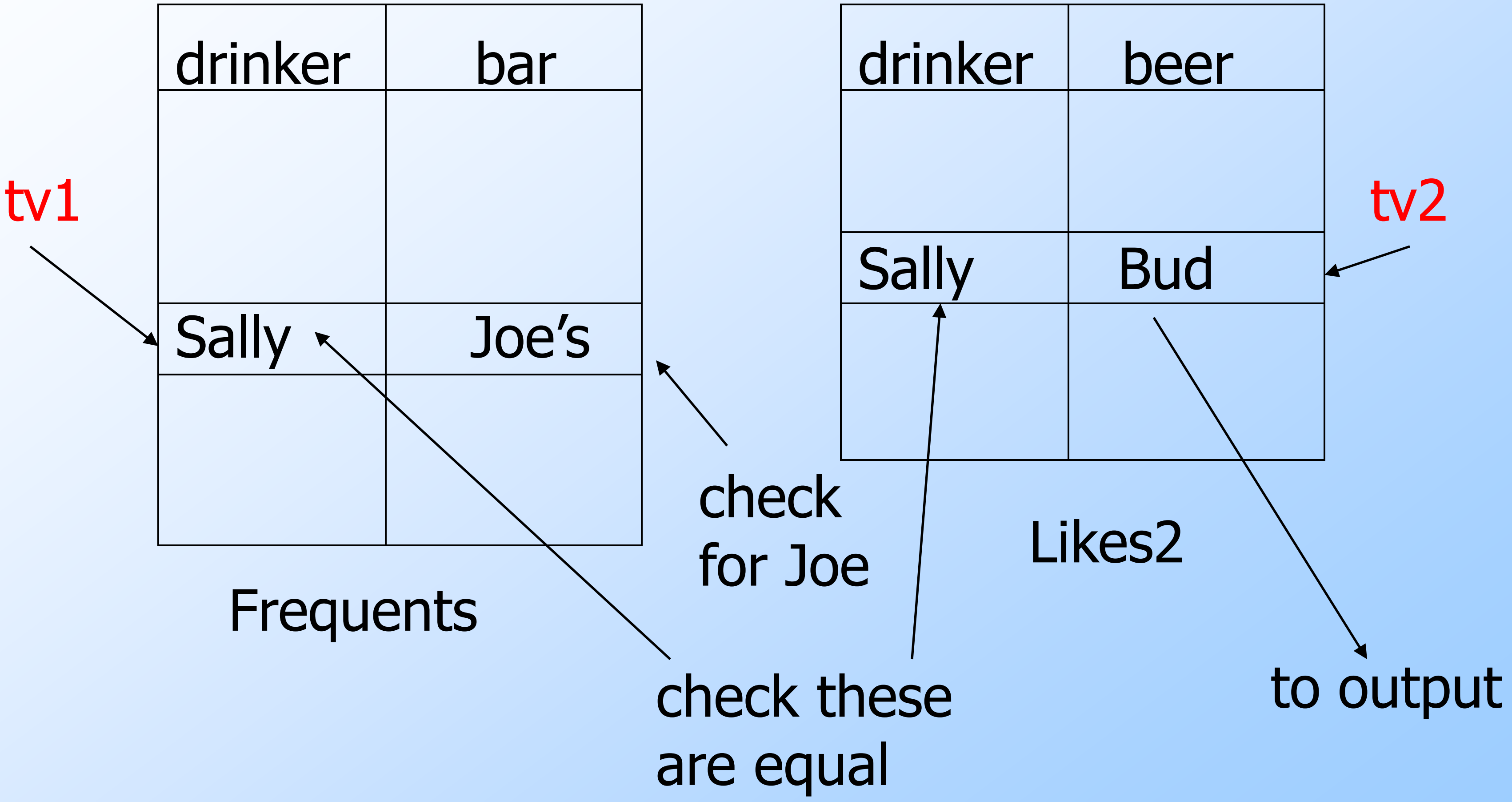
Formal Semantics

- 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



drinker	bar
Sally	Joe's

Frequents

drinker	beer
Sally	Bud

Likes2

tv1

tv2

check for Joe

check these are equal

to output

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: 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;
```

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 Likes2, (SELECT drinker
    FROM Frequents
    WHERE bar = 'Joes Bar') JD
WHERE Likes2.drinker=JD.drinker;
```

Subqueries That Return One Tuple

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

Example: 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 = 'Joes 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 Likes2(drinker, beer), find the name and manufacturer of each beer that Fred likes.

```
SELECT *
FROM Beers
WHERE name IN (SELECT beer
               FROM Likes2
               WHERE drinker = 'Fred');
```

The set of beers Fred likes

What is 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's Tuples

```
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.

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

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

Notice the SQL "not equals" operator

The Operator ANY

- *x* = ANY(<subquery>) 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* >= ANY(<subquery>) means *x* is not the uniquely smallest tuple produced by the subquery.
  - Note tuples must have one component only.

The Operator ALL

- *x* <> ALL(<subquery>) 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.
- <> can be any comparison operator.
- Example: *x* >= ALL(<subquery>) 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.

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>) MINUS (<subquery>)

  /* Some DBMSs use EXCEPT instead of MINUS */

Example: Intersection

- Using Likes2(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

Notice trick: subquery is really a stored table.

```
(SELECT * FROM Likes2)
    INTERSECT
(SELECT drinker, beer
 FROM Sells, Frequents
 WHERE Frequents.bar = Sells.bar
);
```

The drinker frequents a bar that sells the beer.

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.

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 Likes2(drinker, beer):

```
(SELECT drinker FROM Frequents)
     MINUS ALL /* Oracle doesn’t support */
(SELECT drinker FROM Likes2);
```

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

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:

```
Likes2 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*.