SQL Queries

- Principal form:

  `SELECT` desired attributes
  `FROM` tuple variables —
  range over relations
  `WHERE` condition about t.v.'s

Running example relation schema:

```
Beers(name, manf)
Bars(name, addr, license)
Drinkers(name, addr, phone)
Likes(drinker, beer)
Sells(bar, beer, price)
Frequents(drinker, bar)
```

Example

What beers are made by Anheuser-Busch?

```
Beers(name, manf)
```

```
SELECT name
FROM Beers
WHERE manf = 'Anheuser-Busch';
```

- Note single quotes for strings.

name
Bud
Bud Lite
Michelob

Formal Semantics of Single-Relation SQL Query

1. Start with the relation in the `FROM` clause.
2. Apply σ, using condition in `WHERE` clause.
3. Apply π using attributes in `SELECT` clause.

Equivalent Operational Semantics

Imagine a *tuple variable* ranging over all tuples of the relation. For each tuple:

- Check if it satisfies the `WHERE` clause.
- Print the attributes in `SELECT`, if so.

Star as List of All Attributes

```
Beers(name, manf)
```

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

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

Renaming columns

```
Beers(name, manf)

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

beer
Bud
Bud Lite
Michelob

Expressions as Values in Columns

```
Sells(bar, beer, price)
```

```
SELECT bar, beer,
    price*120 AS priceInYen
FROM Sells;
```

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

- Note no `WHERE` clause OK.

- Trick: If you want an answer with a particular string in each row, use that constant as an expression.

  Likes(<u>drinker</u>, <u>beer</u>)

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

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

Example

Find the price Joe's Bar charges for Bud.

```
Sells(bar, beer, price)

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

- Note: two single-quotes in a character string represents one single quote.
- Conditions in `WHERE` clause can use logical operators `AND`, `OR`, `NOT` and parentheses in the usual way.
- Remember: SQL is *case insensitive.* Keywords like `SELECT` or `AND` can be written upper/lower case as you like.
    - ❖ Only inside quoted strings does case matter.

Patterns

- % stands for any string.
- _ stands for any one character.
- "Attribute LIKE pattern" is a condition that is true if the string value of the attribute matches the pattern.
  - Also NOT LIKE for negation.

Example

Find drinkers whose phone has exchange 555.

```
Drinkers(name, addr, phone)

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

- Note patterns must be quoted, like strings.

Multirelation Queries

- List of relations in `FROM` clause.
- Relation-dot-attribute disambiguates attributes from several relations.

Example

Find the beers that the frequenters of Joe's Bar like.

```
Likes(drinker, beer)
Frequents(drinker, bar)

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

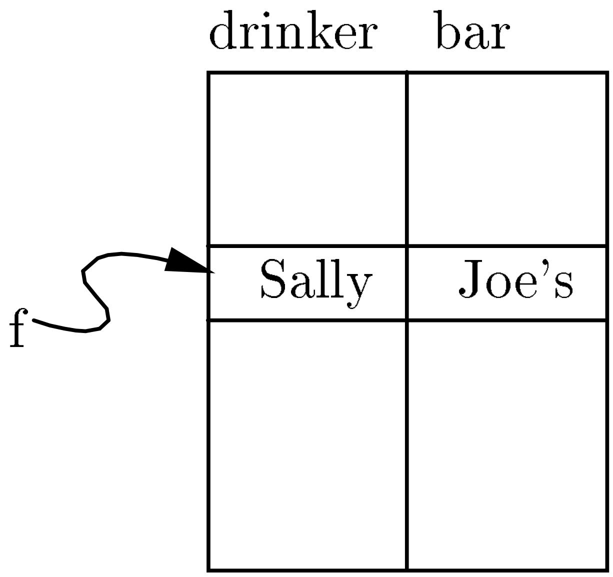
Formal Semantics of Multirelation Queries

Same as for single relation, but start with the product of all the relations mentioned in the `FROM` clause.

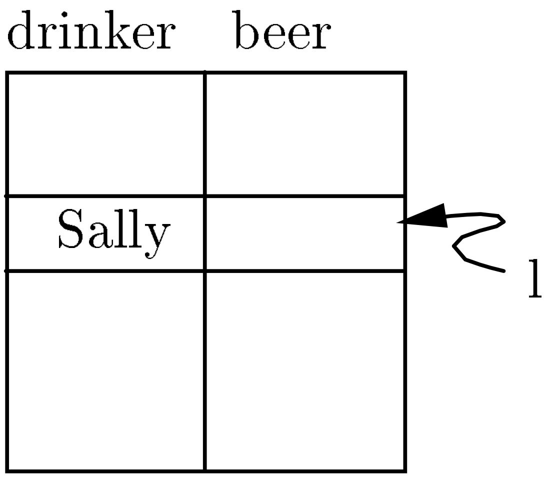
Operational Semantics

Consider a tuple variable for each relation in the `FROM`.

- Imagine these tuple variables each pointing to a tuple of their relation, in all combinations (e.g., nested loops).
- If the current assignment of tuple-variables to tuples makes the `WHERE` true, then output the attributes of the `SELECT`.



Frequents



Likes

Explicit Tuple Variables

Sometimes we need to refer to two or more copies of a relation.

- Use *tuple variables* as aliases of the relations.

Example

Find pairs of beers by the same manufacturer.

```
Beers(name, manf)

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

- SQL2 permits `AS` between relation and its tuple variable; Oracle 7.3.2 does not.
- Note that `b1.name < b2.name` is needed to avoid producing (Bud, Bud) and to avoid producing a pair in both orders.

Subqueries

Result of a select-from-where query can be used in the where-clause of another query.

Simplest Case: Subquery Returns a Single, Unary Tuple

Find bars that serve Miller at the same price Joe charges for Bud.

```
Sells(bar, beer, price)

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

- Notice the *scoping rule*: an attribute refers to the most closely nested relation with that attribute.
- Parentheses around subquery are essential.

The `IN` Operator

"Tuple `IN` relation" is true iff the tuple is in the relation.

Example

Find the name and manufacturer of beers that Fred likes.

```
Beers(name, manf)
Likes(drinker, beer)

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

- Also: `NOT IN`.

`EXISTS`

"`EXISTS`(relation)" is true iff the relation is nonempty.

Example

Find the beers that are the unique beer by their manufacturer.

```
Beers(name, manf)

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

- Note scoping rule: to refer to outer `Beers` in the inner subquery, we need to give the outer a tuple variable, `b1` in this example.
- A subquery that refers to values from a surrounding query is called a *correlated subquery.*

Quantifiers

`ANY` and `ALL` behave as existential and universal quantifiers, respectively.

- Beware: in common parlance, “any” and “all” seem to be synonyms, e.g., “I am fatter than any of you” vs. “I am fatter than all of you.” But in SQL:

Example

Find the beer(s) sold for the highest price.

```
Sells(bar, beer, price)

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

Problem

Find the beer(s) not sold for the lowest price.