SQL:
DATA DEFINITION LANGUAGE

Database Schemas in SQL

- SQL is primarily a query language, for getting information from a database.
  - **Data manipulation language (DML)**
- But SQL also includes a *data-definition* component for describing database schemas.
  - **Data definition language (DDL)**

Creating (Declaring) a Relation

- Simplest form is:

```
CREATE TABLE <name> (
        <list of elements>
);
```

- To delete a relation:

```
DROP TABLE <name>;
```

Elements of Table Declarations

- Most basic element: an attribute and its 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.

Example: Create Table

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

SQL Values

- Integers and reals are represented as you would expect.
- Strings are too, except they require single quotes.
  - Two single quotes = real quote, e.g., `'Joe''s Bar'`.
- Any value can be NULL
  - Unless attribute has NOT NULL constraint
  - E.g., price REAL not null,

Dates and Times

- DATE and TIME are types in SQL.
- The form of a date value is:

  DATE ’yyyy-mm-dd’

  - Example: `DATE ’2007-09-30’` for Sept. 30, 2007.

Times as Values

- The form of a time value is:

  TIME ’hh:mm:ss’

with an optional decimal point and fractions of a second following.

  - Example: `TIME ’15:30:02.5’` = two and a half seconds after 3:30PM.

Declaring Keys

- An attribute or list of attributes may be declared PRIMARY KEY or UNIQUE.
- Either says that no two tuples of the relation may agree in all the attribute(s) on the list.

Our Running Example

Beers(name, manf)

Bars(name, addr, license)

Drinkers(name, addr, phone)

Likes(drinker, beer)

Sells(bar, beer, price)

Frequents(drinker, bar)

- Underline = *key* (tuples cannot have the same value in all key attributes).

Declaring Single-Attribute Keys

- Place PRIMARY KEY or UNIQUE after the type in the declaration of the attribute.
- Example:

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

Declaring Multiattribute Keys

- A key declaration can also be another element in the list of elements of a CREATE TABLE statement.
- This form is essential if the key consists of more than one attribute.
  - May be used even for one-attribute keys.

Example: Multiattribute Key

- The bar and beer together are the key for Sells:

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

PRIMARY KEY vs. UNIQUE

1. There can be only one PRIMARY KEY for a relation, but several UNIQUE attributes.
2. 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.

Kinds of Constraints

- Keys
- Foreign-key, or referential-integrity.
- Domain constraints
  - Constrain values of a particular attribute.
- Tuple-based constraints
  - Relationship among components.
- Assertions: any SQL boolean expression

Foreign Keys

- Values appearing in attributes of one relation must appear together in certain attributes of another relation.
- Example: in Sells(bar, beer, price), we might expect that a beer value also appears in Beers.name

Expressing Foreign Keys

- Use keyword REFERENCES, either:
  1. After an attribute (for one-attribute keys).
  2. As an element of the schema:

  FOREIGN KEY (<list of attributes>)

  REFERENCES <relation> (<attributes>)

- Referenced attributes must be declared PRIMARY KEY or UNIQUE.

Example: With Attribute

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

CREATE TABLE Sells (
  bar       CHAR(20),
  beer      CHAR(20) REFERENCES Beers(name),
  price     REAL );
```

Example: As Schema Element

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

CREATE TABLE Sells (
  bar       CHAR(20),
  beer      CHAR(20),
  price     REAL,
  FOREIGN KEY(beer) REFERENCES
      Beers(name));
```

Enforcing Foreign-Key Constraints

- If there is a foreign-key constraint from relation *R* to relation *S*, two violations are possible:

1. An insert or update to *R* introduces values not found in *S*.
2. A deletion or update to S causes some tuples of *R* to "dangle."

Actions Taken --- (1)

- Example: suppose R = Sells, S = Beers.
- An insert or update to Sells that introduces a nonexistent beer must be rejected.
- A deletion or update to Beers that removes a beer value found in some tuples of Sells can be handled in three ways…

Actions Taken --- (2)

1. *Default* : Reject the modification.
2. Cascade : Make the same changes in Sells.
   - Deleted beer: delete Sells tuple.
   - Updated beer: change value in Sells.
3. *Set NULL* : Change the beer to NULL.

Example: Cascade

- Delete the Bud tuple from Beers:
  - Then delete all tuples from Sells that have beer = ’Bud’.
- Update the Bud tuple by changing ’Bud’ to ’Budweiser’:
  - Then change all Sells tuples with beer = ’Bud’ to beer = ’Budweiser’.

Example: Set NULL

- Delete the Bud tuple from Beers:
  - Change all tuples of Sells that have beer = 'Bud' to have beer = NULL.
- Update the Bud tuple by changing 'Bud' to 'Budweiser':
  - Same change as for deletion.

Choosing a Policy

- When we declare a foreign key, we may choose policies SET NULL or CASCADE independently for deletions and updates.
- Follow the foreign-key declaration by:

ON [UPDATE, DELETE][SET NULL CASCADE]

- Two such clauses may be used.
- Otherwise, the default (reject) is used.

Example: Setting Policy

```
CREATE TABLE Sells (
  bar     CHAR(20),
  beer    CHAR(20),
  price   REAL,
  FOREIGN KEY(beer)
     REFERENCES Beers(name)
     ON DELETE SET NULL
     ON UPDATE CASCADE
);
```

Attribute-Based Checks

- Constraints on the value of a particular attribute.
- Add CHECK(<condition>) to the declaration for the attribute.
- The condition may use the name of the attribute, but any other relation or attribute name must be in a subquery.

Example: Attribute-Based Check

```
CREATE TABLE Sells (
  bar     CHAR(20),
  beer    CHAR(20) CHECK ( beer IN
           (SELECT name FROM Beers)),
  price   REAL CHECK ( price <= 5.00 )
);
```

Timing of Checks

- Attribute-based checks are performed only when a value for that attribute is inserted or updated.
  - Example: `CHECK (price <= 5.00)` checks every new price and rejects the modification (for that tuple) if the price is more than $5.
  - Example: `CHECK (beer IN (SELECT name FROM Beers))` not checked if a beer is deleted from Beers (unlike foreign-keys).

Tuple-Based Checks

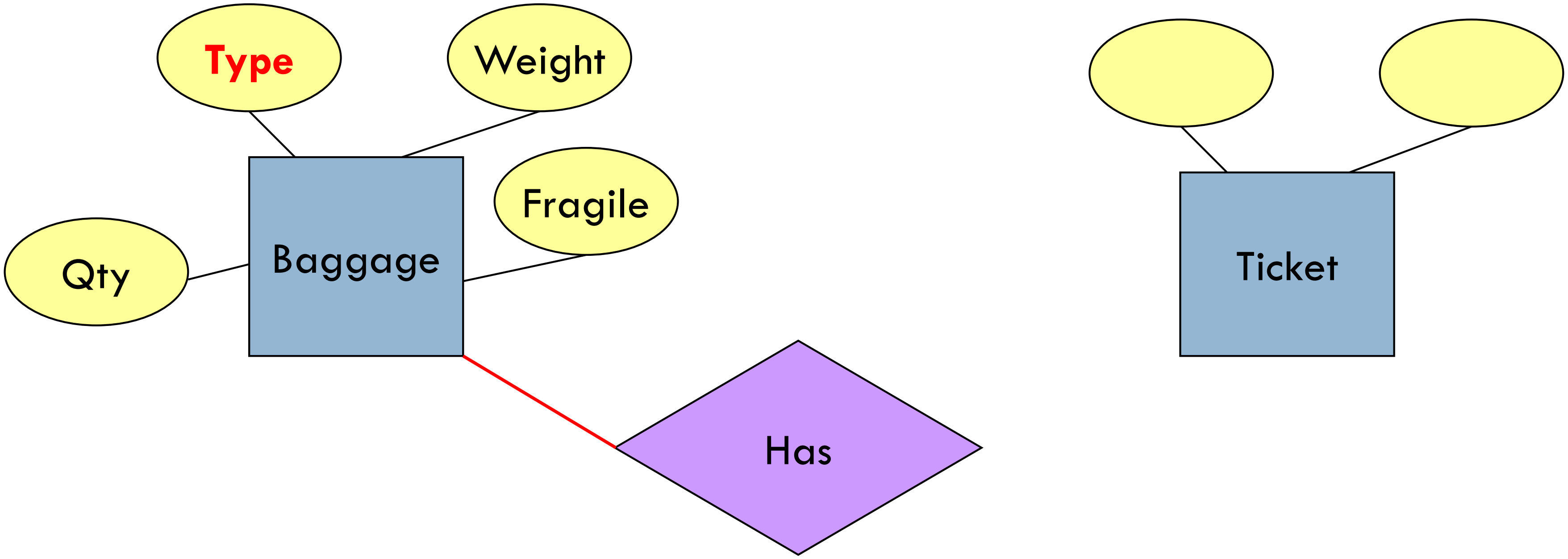
- CHECK (<condition>) may be added as a relation-schema element.
- The condition may refer to any attribute of the relation.
  - But other attributes or relations require a subquery.
- Checked on insert or update only.

Example: Tuple-Based Check

- Only Joe’s Bar can sell beer for more than $5:

```
CREATE TABLE Sells (
   bar          CHAR(20),
   beer         CHAR(20),
   price        REAL,
   CHECK (bar = ’Joe’’s Bar’ OR
                  price <= 5.00)
);
```

Asg 1 Update: Missing attribute in Baggage



INTRODUCTION TO SQL

Why SQL?

- SQL is a very-high-level language.
  - Structured Query 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."

Credit: Renee J. Miller

Database Schemas in SQL

- SQL is primarily a query language, for getting information from a database.
  - **Data manipulation language (DML)**
- But SQL also includes a *data-definition* component for describing database schemas.
  - **Data definition language (DDL)**

Select-From-Where Statements

SELECT desired attributes

FROM one or more tables

WHERE condition about tuples of
the tables

Our Running Example

- 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.

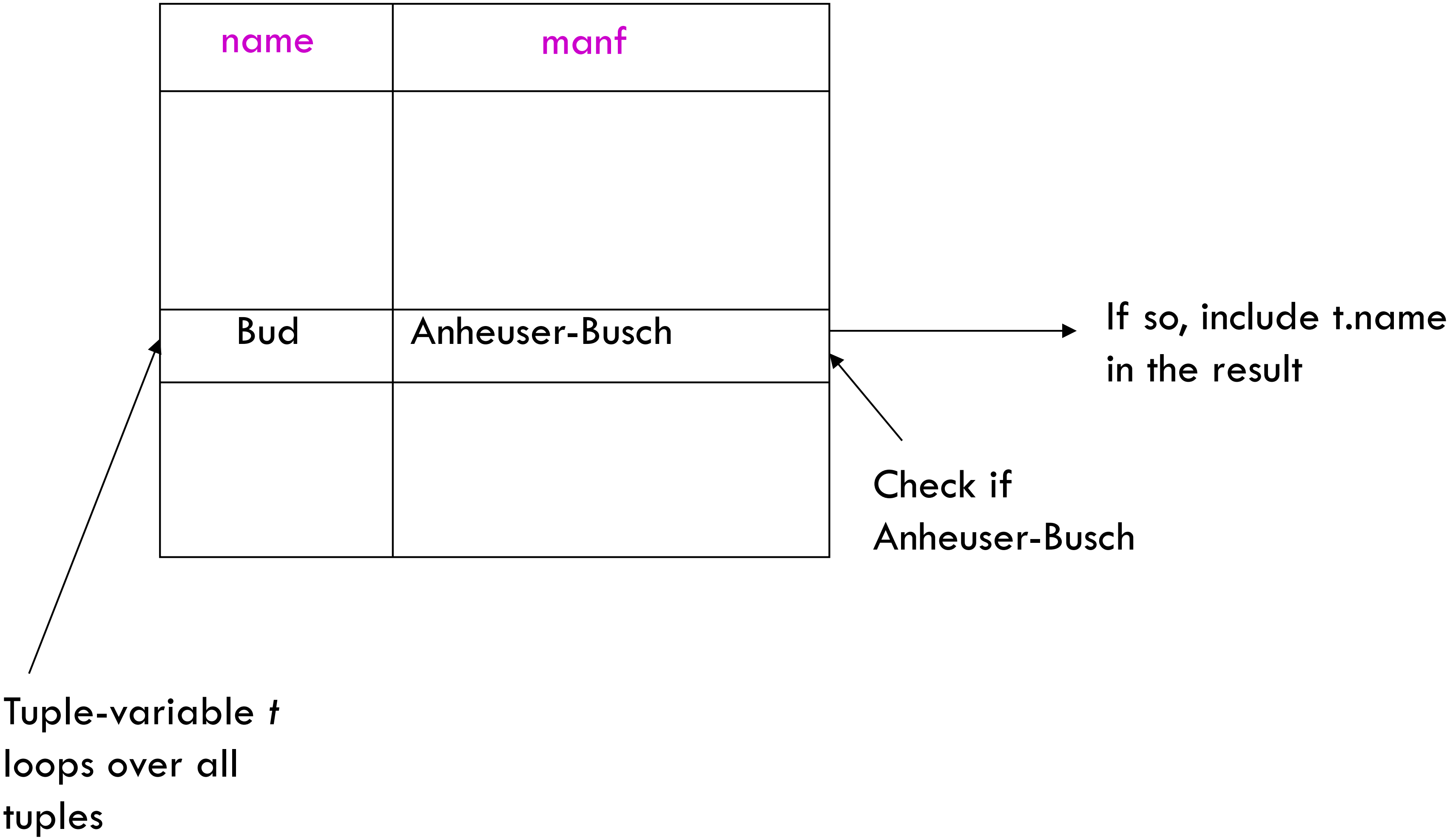
Meaning of Single-Relation Query

- 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 - General

- Think of a *tuple variable* visiting each tuple of the relation mentioned in FROM.
- Check if the tuple assigned to the tuple variable satisfies the WHERE clause.
- If so, compute the attributes or expressions of the SELECT clause using the components of this tuple.

Operational Semantics



name	manf
Bud	Anheuser-Busch

If so, include t.name in the result

Check if Anheuser-Busch

Tuple-variable *t* loops over all tuples

Example

- What beers are made by Anheuser-Busch?

```
    SELECT name
    FROM Beers
    WHERE manf = ’Anheuser-Busch’;
OR:
    SELECT t.name
    FROM Beers t
    WHERE t.manf =’Anheuser-Busch’;
```

Note: these are identical queries.

* 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 valid expression can appear as an element of a SELECT clause.
- Example: Using Sells(bar, beer, price):

```
SELECT bar, beer,
       price*95 AS priceInYen
FROM Sells;
```

Result of Query

bar	beer	priceInYen
Joe’s	Bud	285
Sue’s	Miller	342
• • •	• • •	• • •

Example: 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
…	…

Complex Conditions in WHERE Clause

- □ Boolean operators AND, OR, NOT.
- □ Comparisons =, <>, <, >, <=, >=.

Example: Complex Condition

- 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
  - % = “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

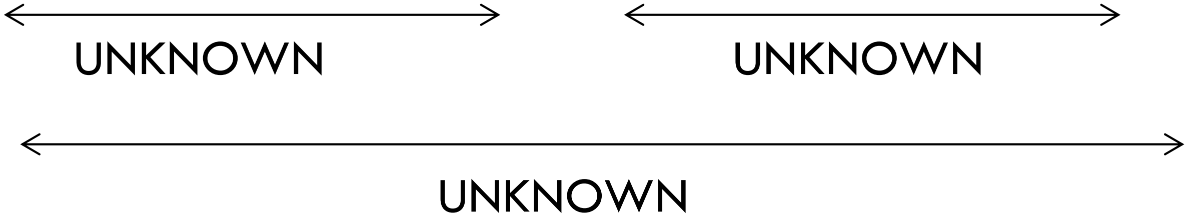
- To understand how AND, OR, and NOT work in 3-valued logic
- For TRUE result
  - OR: at least one operand must be TRUE
  - AND: both operands must be TRUE
  - NOT: operand must be FALSE
- For FALSE result
  - OR: both operands must be FALSE
  - AND: at least one operand must be FALSE
  - NOT: operand must be TRUE
- Otherwise, result is UNKNOWN

Example

□ From the following Sells relation:

bar	beer	price
Joe’s Bar	Bud	NULL

```
SELECT bar
FROM Sells
WHERE price < 2.00 OR price >= 5.00;
```



UNKNOWN   UNKNOWN

UNKNOWN

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: 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;
```

Example: Joining Two Relations

□ Alternatively can use explicit (named) tuple variables

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

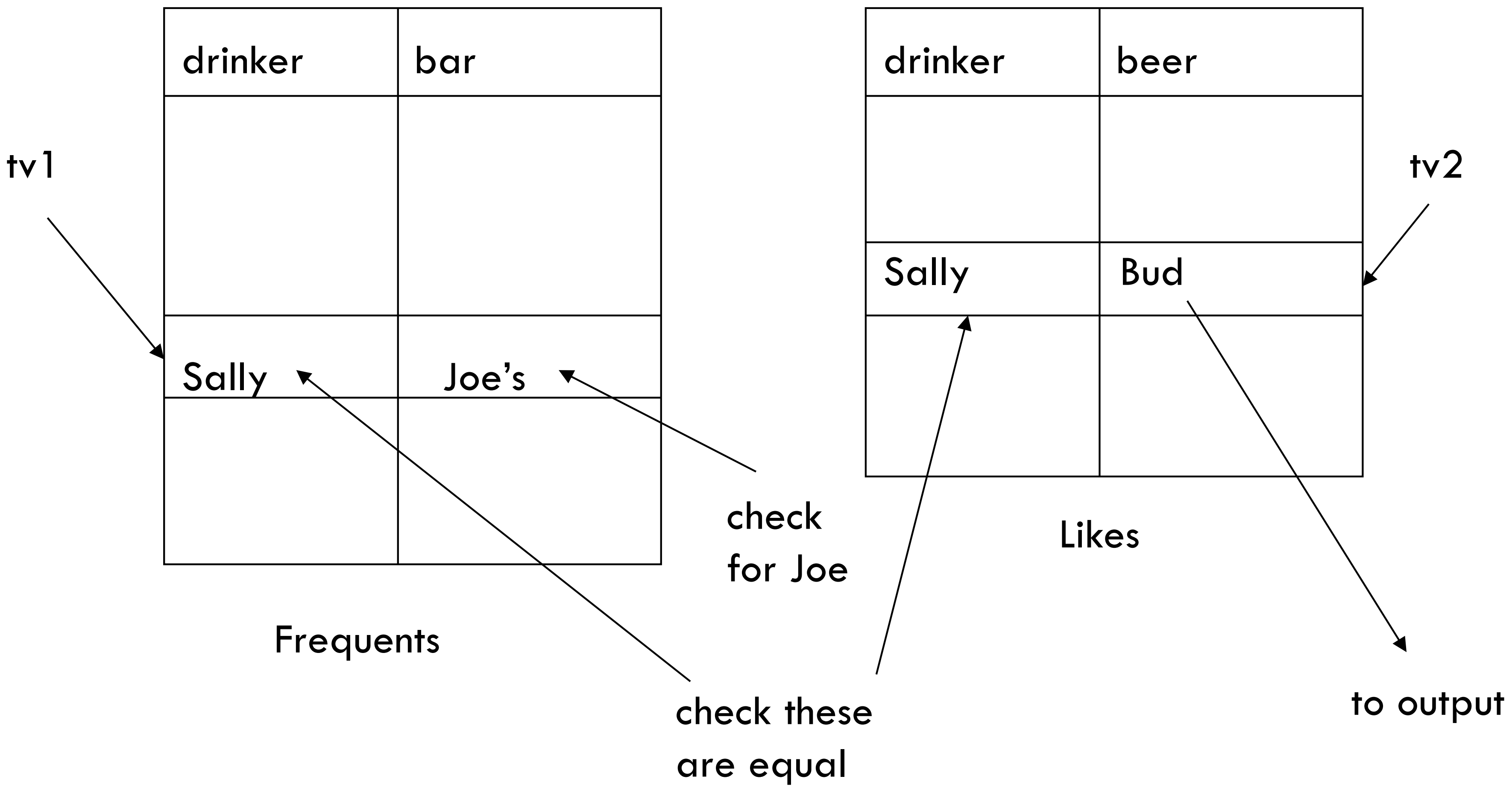
Formal Semantics

- Almost the same as for single-relation queries:
  - Start with the product of all the relations in the FROM clause.
  - Apply the selection condition from the WHERE clause.
  - 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



tv1

drinker	bar
Sally	Joe’s

Frequents

tv2

drinker	beer
Sally	Bud

Likes

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).
  - Do not produce the same pair twice like (Bud, Miller) and (Miller, Bud).

Select-From-Where Statements

SELECT desired attributes

FROM one or more tables

WHERE condition about tuples of
    the tables

Example

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

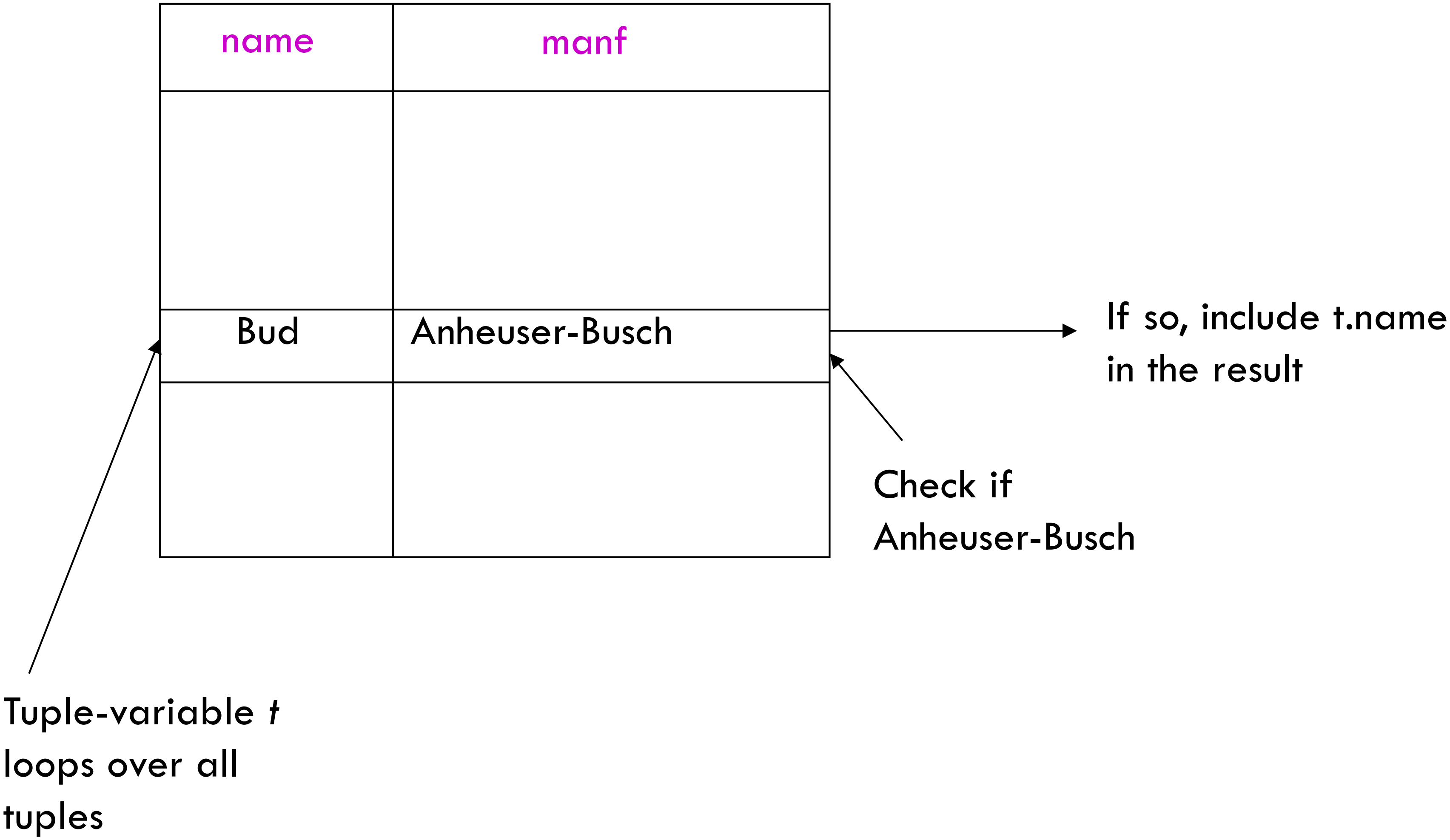
```
SELECT name
FROM Beers
WHERE manf = ’Anheuser-Busch’;
```

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.

Operational Semantics



name	manf
Bud	Anheuser-Busch

If so, include t.name in the result

Check if Anheuser-Busch

Tuple-variable *t* loops over all tuples

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

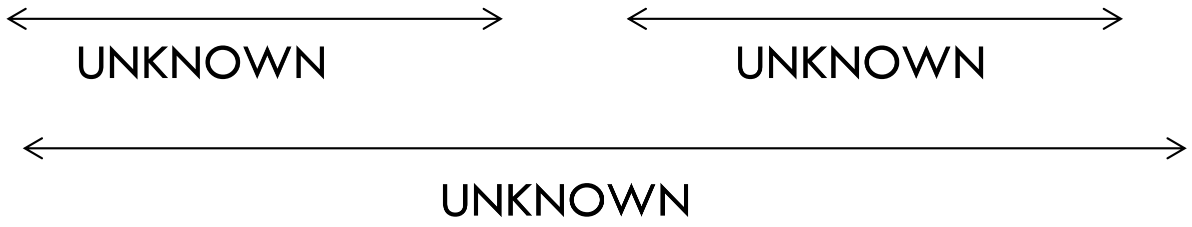
- To understand how AND, OR, and NOT work in 3-valued logic
- For TRUE result
  - OR: at least one operand must be TRUE
  - AND: both operands must be TRUE
  - NOT: operand must be FALSE
- For FALSE result
  - OR: both operands must be FALSE
  - AND: at least one operand must be FALSE
  - NOT: operand must be TRUE
- Otherwise, result is UNKNOWN

Example

- From the following Sells relation:

bar	beer	price
Joe's Bar	Bud	NULL

```
SELECT bar
FROM Sells
WHERE price < 2.00 OR price >= 5.00;
```



UNKNOWN

UNKNOWN

UNKNOWN

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: 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;
```

Example: Joining Two Relations

- Alternatively can use explicit (named) tuple variables

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

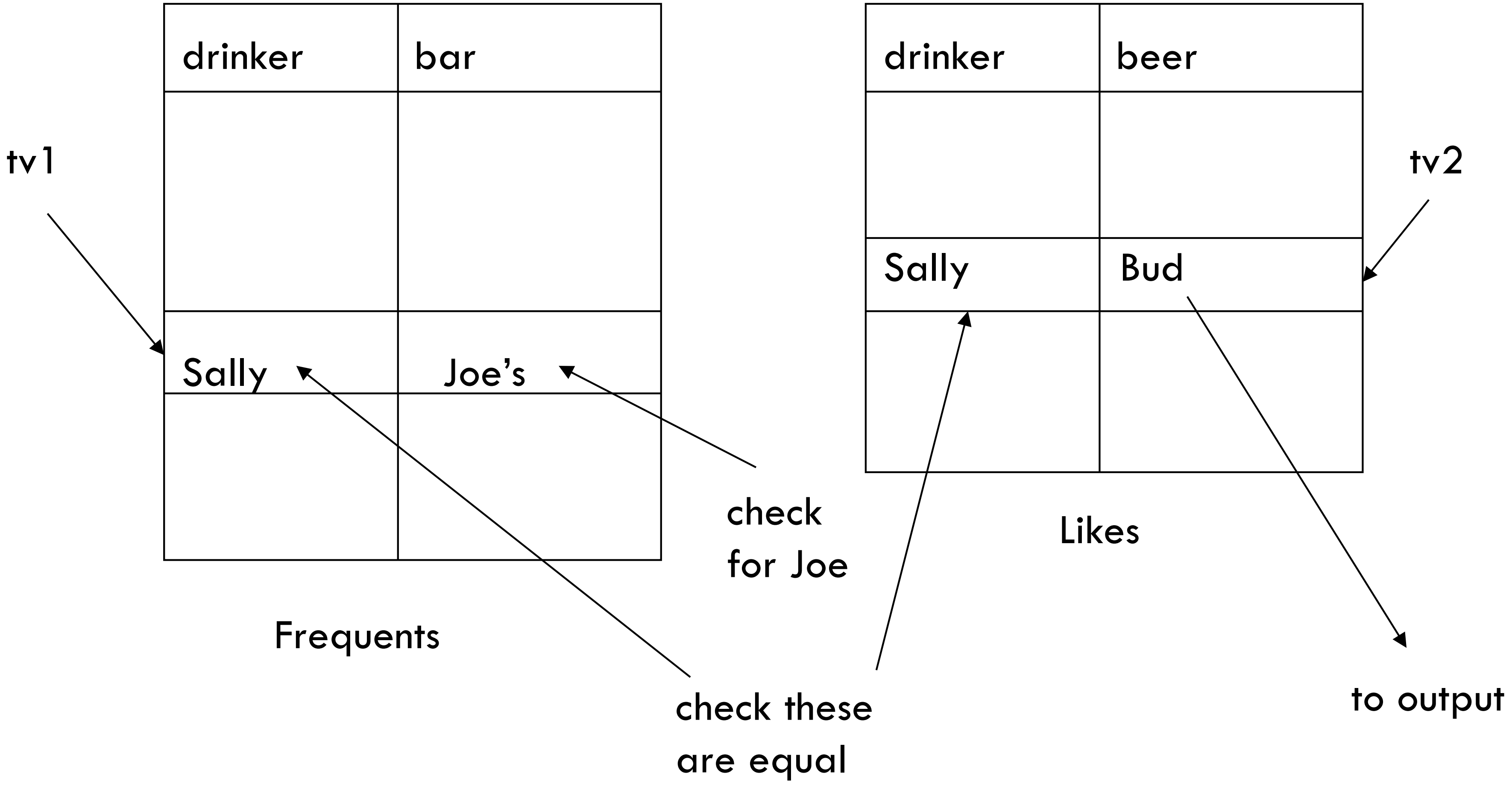
Formal Semantics

- Almost the same as for single-relation queries:
  - Start with the product of all the relations in the FROM clause.
  - Apply the selection condition from the WHERE clause.
  - 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

tv1

Sally

Joe's

Frequents

check for Joe

drinker

beer

tv2

Sally

Bud

Likes

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).
  - Do not produce the same pair twice like (Bud, Miller) and (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 Likes, (SELECT drinker
             FROM Frequents
         WHERE bar = ’Joe’’s Bar’)JD
WHERE Likes.drinker = JD.drinker;
```

Subqueries often obscure queries

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

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

Simple join query

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.
  - Remember SQL's 3-valued logic.

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 work:

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

Query + Subquery Solution

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

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’);
```

The price at which Joe sells Bud

What if price of Bud is NULL?

Query + Subquery Solution

```
SELECT bar
FROM Sells
WHERE beer = ’Miller’ AND
    price = (SELECT price
                FROM Sells
                WHERE beer = ’Bud’);
```

What if subquery returns multiple values?

Recap: Conditions in WHERE Clause

- Boolean operators AND, OR, NOT.
- Comparisons =, <>, <, >, <=, >=.
- LIKE operator

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, ≥ \$90,000 and ≤ \$100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000

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.

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

Drinkers who frequent Joe’s Bar

Subqueries often obscure queries

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

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

Simple join query

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.
  - Remember SQL’s 3-valued logic.

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 work:

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

Query + Subquery Solution

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

Sells(<u>bar</u>, <u>beer</u>, price)

```
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

What if price of Bud is NULL?

Query + Subquery Solution

```
SELECT bar
FROM Sells
WHERE beer = ’Miller’ AND
      price = (SELECT price
               FROM Sells
               WHERE beer = ’Bud’);
```

What if subquery returns multiple values?

Recap: Conditions in WHERE Clause

- Boolean operators AND, OR, NOT.
- Comparisons =, <>, <, >, <=, >=.
- LIKE operator

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between \$90,000 and \$100,000 (that is, ≥ \$90,000 and ≤ \$100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000

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.

The IN Operator

- <value> IN (<subquery>) is true if and only if the <value> is a member of the relation produced by the subquery.
  - Opposite: <value> NOT IN (<subquery>).
- IN-expressions can appear in WHERE clauses.
- WHERE col IN (value1, value2, …)

IN is Concise

-
  ```
  SELECT * FROM Cartoons
   WHERE LastName IN (‘Simpsons', 'Smurfs’, ’Flintstones’)
  ```

-
  ```
  SELECT * FROM Cartoons
  WHERE LastName = ‘Simpsons’
  OR LastName = ‘Smurfs’
  OR LastName = ‘Flintstones’
  ```

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

IN vs. Join

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

SELECT R.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.

Query + Subquery Solution

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

What if subquery returns multiple values?

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.

The IN Operator

- <value> IN (<subquery>) is true if and only if the <value> is a member of the relation produced by the subquery.
  - Opposite: <value> NOT IN (<subquery>).
- IN-expressions can appear in WHERE clauses.
- WHERE col IN (value1, value2, …)

IN vs. Join

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

SELECT R.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.

Back to our original query…

```
SELECT bar
FROM Sells
WHERE beer = ’Miller’ AND
      price = (SELECT price
               FROM Sells
               WHERE beer = ’Bud’);
```

Use IN() or = ANY()

Recap

- IN( ) is equivalent to = ANY( )
- For ANY( ), you can use other comparison operators such as >, <,... etc, but not applicable for IN( )

- The < >ANY operator, however, differs from NOT IN:
  - < >ANY means not = a, or not = b, or not = c
  - NOT IN means not = a, and not = b, and not = c.
  - <>ALL means the same as NOT IN.

Example: =ANY

Sells

Bar	**Beer**	**Price**
Jane	Miller	3.00
Joe	Miller	4.00
Joe	Bud	3.00
Jack	Bud	4.00
Tom	Miller	4.50

```
SELECT Bar
FROM Sells
WHERE Beer = ‘Miller' AND Price =
          ANY(SELECT Price
                 FROM Sells
                 WHERE Beer=‘Bud')
```

Result

Bar
Jane
Joe

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 (only) beer made by their manufacturer.

Credit: Renee J. Miller

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. (Some DBMS consider this ambiguous.)

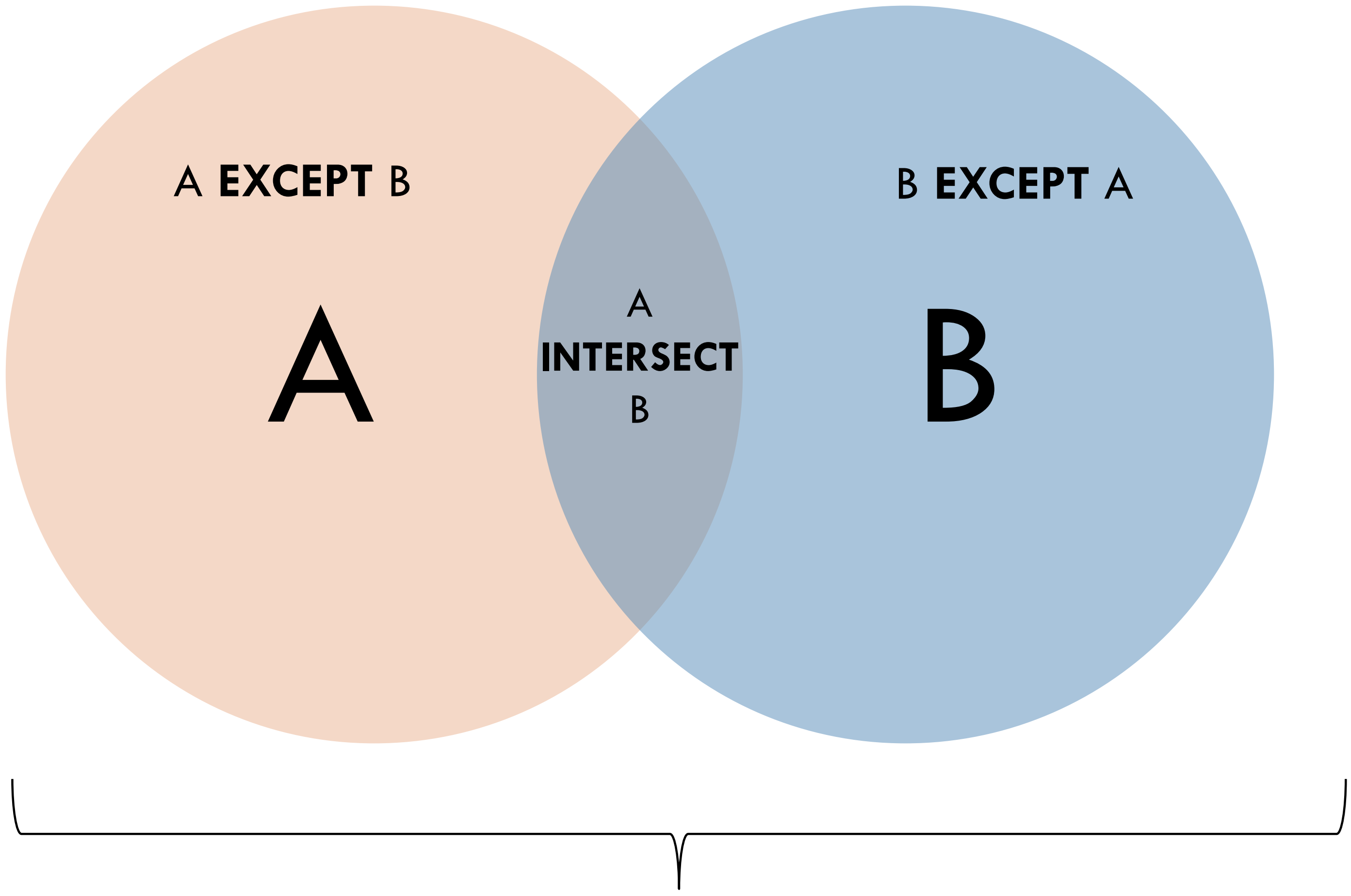
Set of beers with the same manf as b1, but not the same beer

Notice the SQL "not equals" operator

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>)

Visually



A **EXCEPT** B

B **EXCEPT** A

A

A **INTERSECT** B

B

A **UNION** B
(no duplicates)

Example: Intersection

- Using Likes(drinker, beer), Sells(bar, beer, price), and Frequents(drinker, bar), find the drinkers and beers such that:
  - The drinker likes the beer, and
  - The drinker frequents at least one bar that sells the beer.

Solution

subquery is really a stored table.

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

The drinker frequents a bar that sells the beer.