Structured Query Language

SQL Introduction

Standard language for querying and manipulating data

Structured **Q**uery **L**anguage

Many standards out there:
ANSI SQL, SQL92 (SQL2), SQL99 (SQL3), SQL:2003

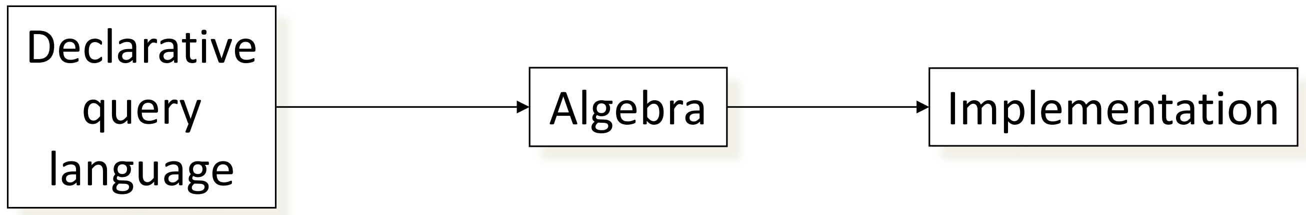
Vendors support various subsets of these.

Note: alternative name: Sequel (**S**tructured **E**nglish **QUe**ry **L**anguage)
from IBM project in 70's
SQL 99: object-relational features (e.g. user-defined types), recursion
SQL 2003: XML

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.

SQL's Place in the Big Picture



SQL
Relational calculus
(formalism behind SQL)

Relational algebra
Relational bag algebra

- Relational algebra: formalism for creating new relations from existing ones using relational operators

Agenda

- SQL DML: Data Manipulation (Sub)Language
  - SQL query
  - Relations as bags
  - Joins
  - Grouping and aggregation
  - Database modification
- SQL DDL: Data Definition (Sub)Language
  - Define/modify schemas

SQL

Select-From-Where Statements
Meaning of Queries
Subqueries

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
```

Lexical Order vs. Logical Order

Lexical

- SELECT [DISTINCT]
- FROM
- WHERE
- GROUP BY
- HAVING
- UNION
- ORDER BY

Logical

- FROM
- WHERE
- GROUP BY
- HAVING
- SELECT
- DISTINCT
- UNION
- ORDER BY

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)

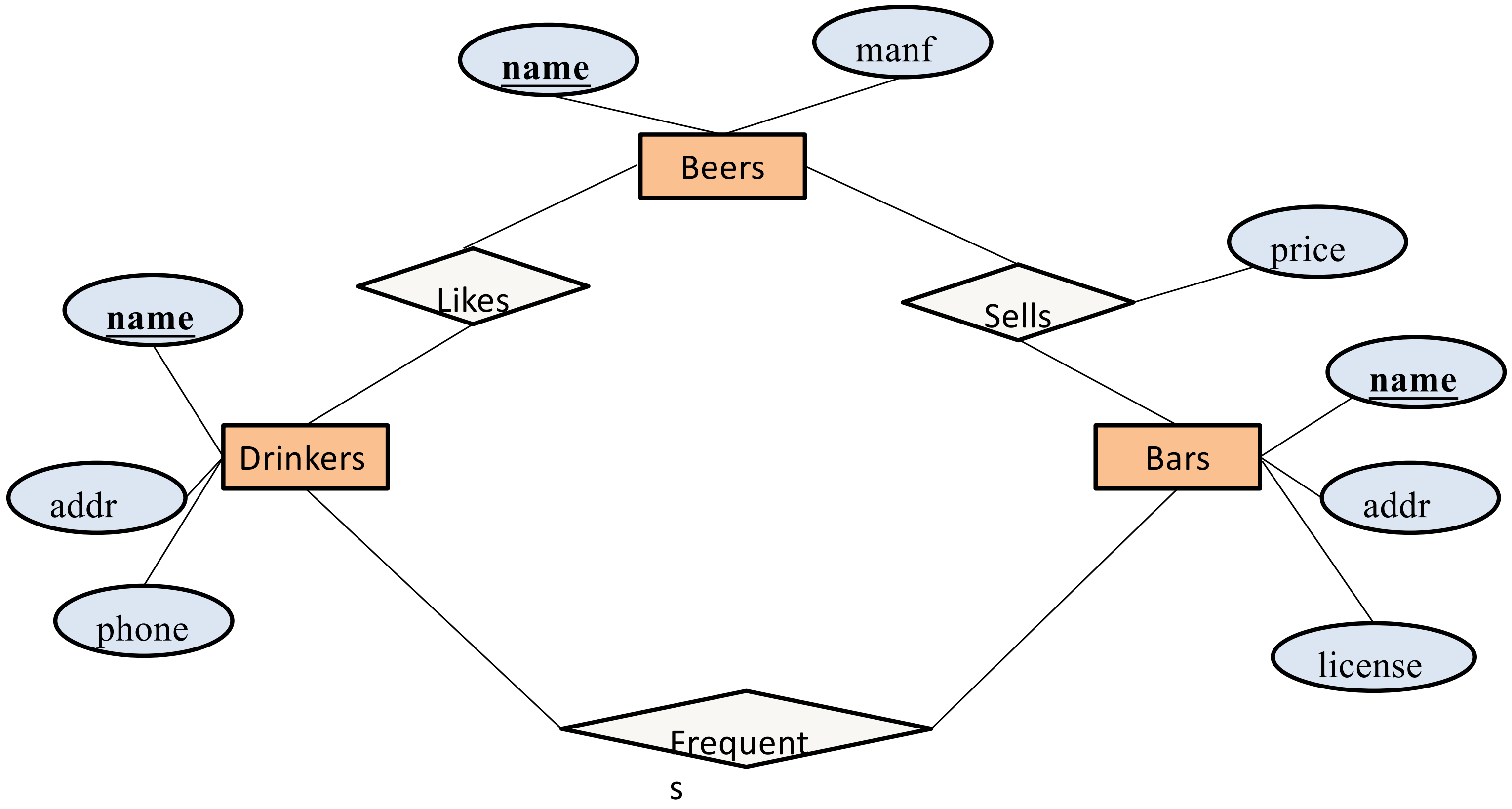
    Drinkers(name, addr, phone)

    Likes(drinker, beer)

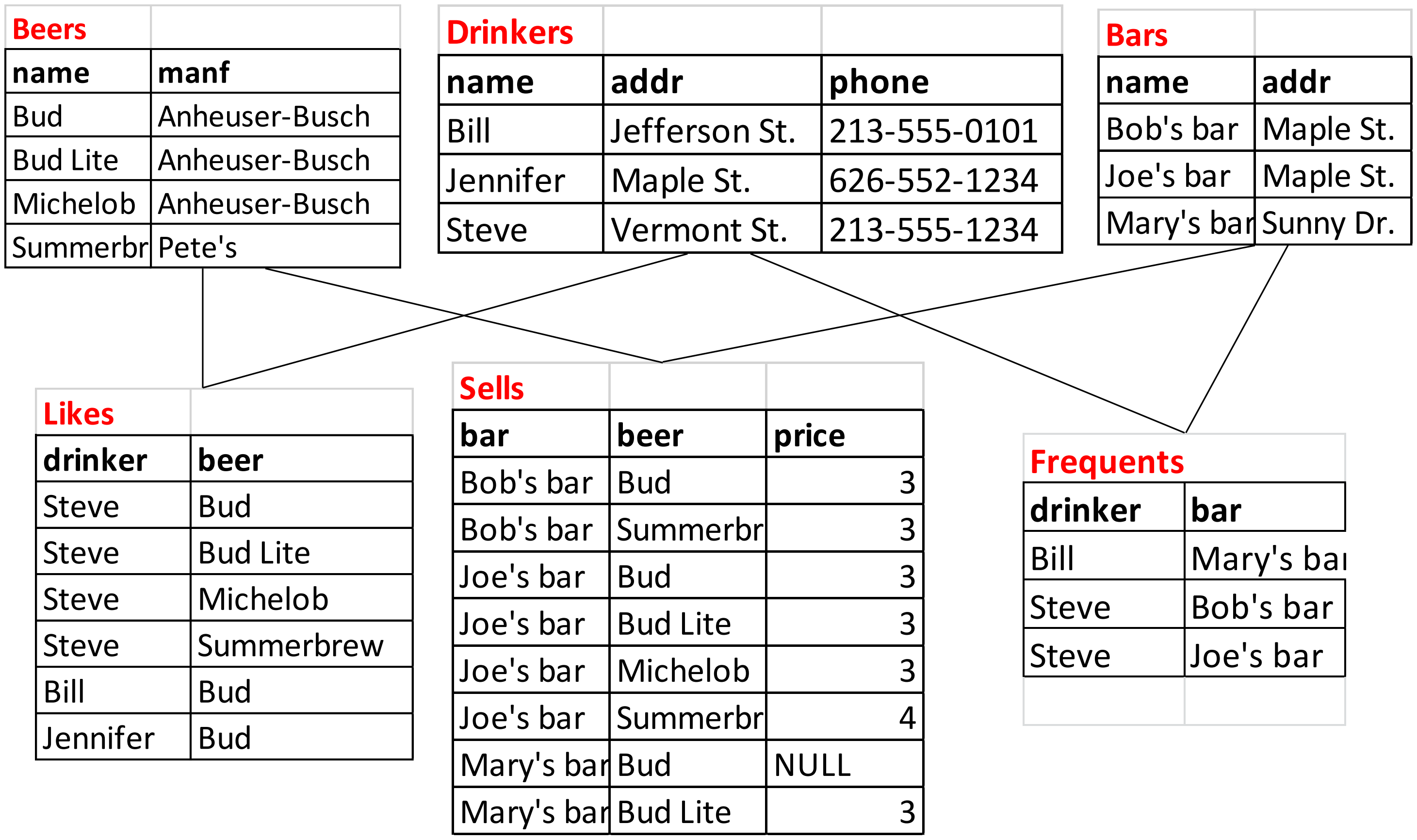
    Sells(bar, beer, price)

    Frequents(drinker, bar)

ER Diagram


name
manf
Beers
Likes
Sells
price
name
Drinkers
addr
phone
Bars
name
addr
license
Frequent
s

Tables



Beers

name	manf
Bud	Anheuser-Busch
Bud Lite	Anheuser-Busch
Michelob	Anheuser-Busch
Summerbr	Pete's

Drinkers

name	addr	phone
Bill	Jefferson St.	213-555-0101
Jennifer	Maple St.	626-552-1234
Steve	Vermont St.	213-555-1234

Bars

name	addr
Bob's bar	Maple St.
Joe's bar	Maple St.
Mary's bar	Sunny Dr.

Likes

drinker	beer
Steve	Bud
Steve	Bud Lite
Steve	Michelob
Steve	Summerbrew
Bill	Bud
Jennifer	Bud

Sells

bar	beer	price
Bob's bar	Bud	3
Bob's bar	Summerbr	3
Joe's bar	Bud	3
Joe's bar	Bud Lite	3
Joe's bar	Michelob	3
Joe's bar	Summerbr	4
Mary's bar	Bud	NULL
Mary's bar	Bud Lite	3

Frequents

drinker	bar
Bill	Mary's ba
Steve	Bob's bar
Steve	Joe's 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, with tuples listing the name of each beer by Anheuser-Busch, such as Bud.

Operational Semantics

- 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.

Another Example

Company(ticker, name, country, stockPrice)

Find all US companies whose stock price is > 50:

```
SELECT   *
FROM     Company
WHERE    country='US' AND stockPrice > 50
```

Output schema: R(ticker, name, country, stockPrice)

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
…	…	…

Another Example: Constant Expressions

- From 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

- From 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';
```

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,
  concat() in mysql).
- Lexicographic order on strings (e.g., name >= 'j').
- 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 **NOT** *case-sensitive*. 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-_ _ _ _';

(remove spaces between _)
```

The **LIKE** operator

- s **LIKE** p: pattern matching on strings
- p may contain two special symbols:
  - % = any sequence of characters
  - _ = any single character

Company(ticker, name, address, country, stockPrice)
Find all US companies whose address contains “Mountain”:

```
SELECT   *
FROM     Company
WHERE    country='USA' AND
          address LIKE '%Mountain%'
```

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 = 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

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

Reason: 2-Valued Laws != 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* OR NOT *p* = TRUE.
  - When *p* = UNKNOWN, the left side is

    MAX(½, (1 – ½))

    = ½

    != 1.

Null Values

- If x=Null then 4*(3-x)/7 is still NULL
- If x=Null   then x='Joe'   is UNKNOWN

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

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>”

What are results of these queries?

- select * from Frequents, Likes;
- select drinker from Frequents
  where bar = 'Joe''s bar'
- select beer from Likes
  where drinker = 'Jennifer' or drinker = 'Steve'

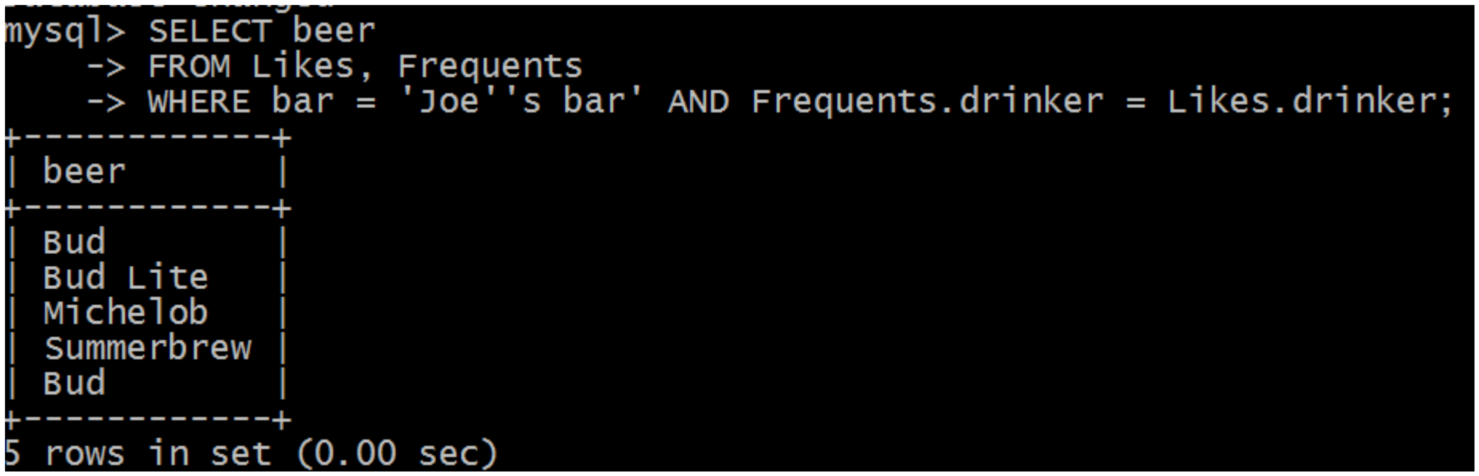
Example

- 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;
```

Result

- Why "Bud" appears twice?



```
mysql> SELECT beer
    -> FROM Likes, Frequents
    -> WHERE bar = 'Joe''s bar' AND Frequents.drinker = Likes.drinker;
+------------+
| beer       |
+------------+
| Bud        |
| Bud Lite   |
| Michelob   |
| Summerbrew |
| Bud        |
+------------+
5 rows in set (0.00 sec)
```

Formal Semantics

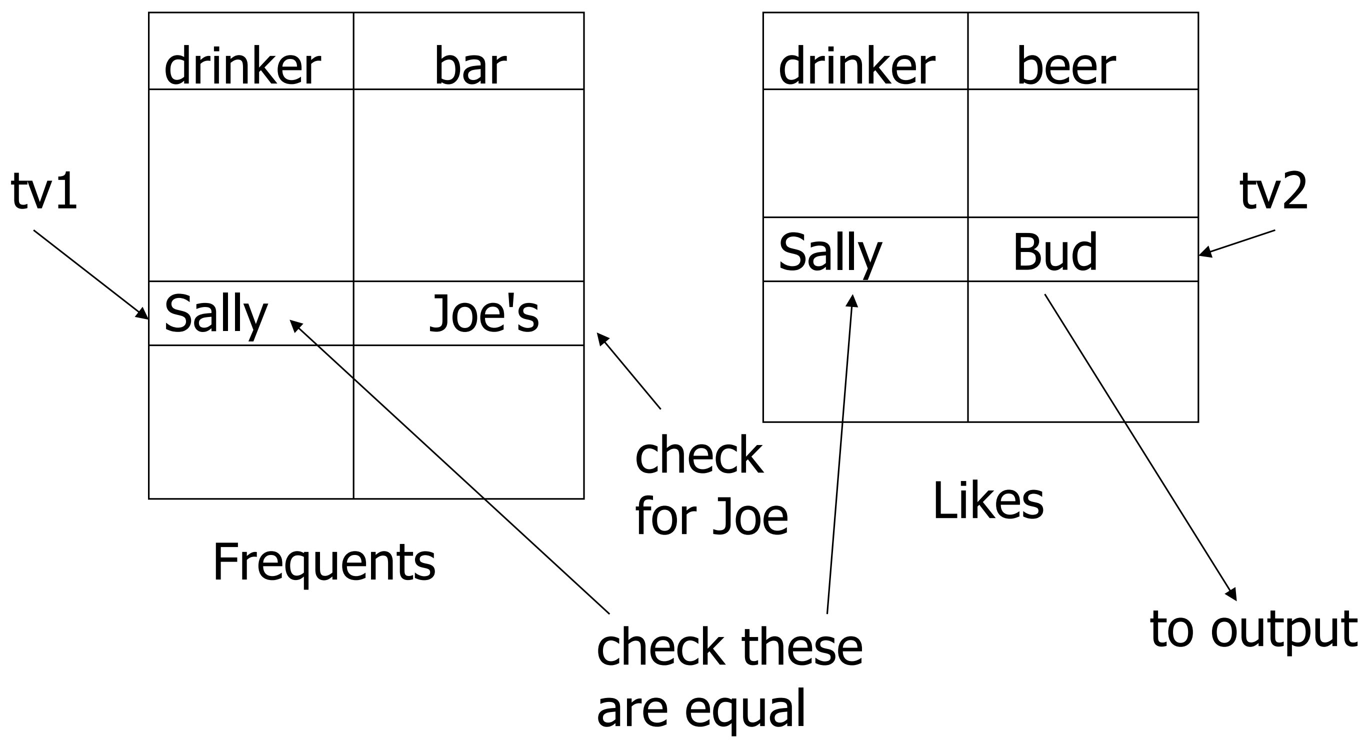
- Almost the same as for single-relation queries:
  1. Start with the product of all the relations in the FROM clause.
  2. Apply the selection condition from the WHERE clause.
  3. Project onto the list of attributes and expressions in the SELECT clause.

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

- Find beers liked by drinkers who frequent Joe's bar



Explicit Tuple-Variables

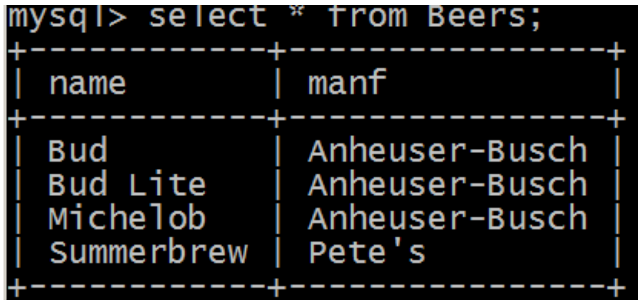
- Sometimes, a query needs to use two copies of the same relation.
- Distinguish copies by following the relation name by the name of a tuple-variable, in the FROM clause.
- It's always an option to rename relations this way, even when not essential.

Example

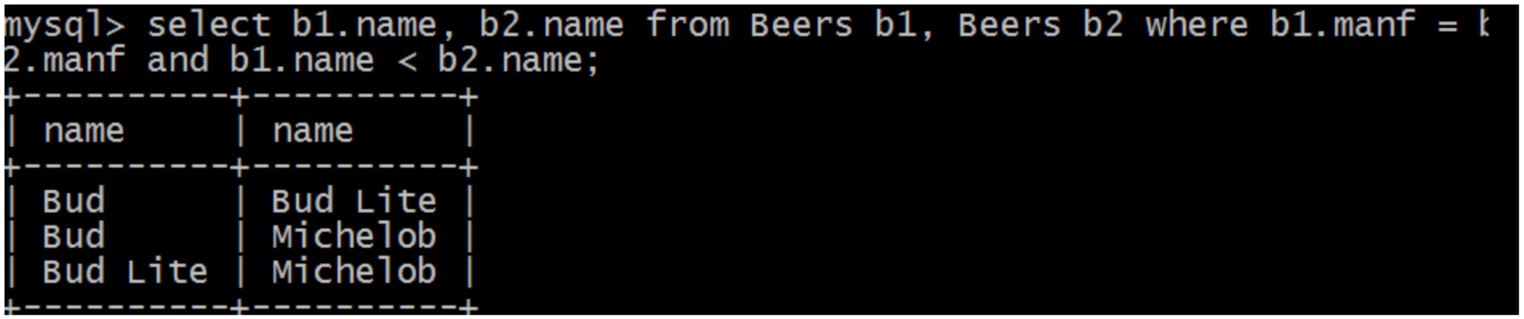
- From Beers(name, manf), find all pairs of beers by the same manufacturer.
  - Do not produce pairs like (Bud, Bud).
  - Produce pairs in alphabetic order, e.g. (Bud, Miller), not (Miller, Bud).

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

Example



```
mysql> select * from Beers;
+------------+----------------+
| name       | manf           |
+------------+----------------+
| Bud        | Anheuser-Busch |
| Bud Lite   | Anheuser-Busch |
| Michelob   | Anheuser-Busch |
| Summerbrew | Pete's         |
+------------+----------------+
```



```
mysql> select b1.name, b2.name from Beers b1, Beers b2 where b1.manf = b
2.manf and b1.name < b2.name;
+----------+----------+
| name     | name     |
+----------+----------+
| Bud      | Bud Lite |
| Bud      | Michelob |
| Bud Lite | Michelob |
+----------+----------+
```

Subqueries

Subquery in the from clause

- A parenthesized SELECT-FROM-WHERE statement (*subquery*) can be used in FROM clause
- Example:
  - select * from (select * from Beers) as b
  - Note tuple variable needed to name the relation generated by the subquer

Subquery in the where clause

- Introduced by '=' (or '!=')
  - x = (subquery)
  - x can be an attribute or a tuple of attributes
  - Subquery needs to return exactly one result
- Introduced by 'in' (or 'not in')
  - x in (subquery)
  - Subquery may return multiple results

Subquery introduced by '='

- Subquery needs to return exactly one result!

```
select * from Beers
where (name, manf) =
          (select name, manf
           from Beers where name = 'Bud');
```

Return > 1 tuple $\longrightarrow$

```
select * from Beers
where (name, manf) =
          (select name, manf
           from Beers
           where manf = 'Anheuser-Busch');
```

Subquery introduced by 'in'

- Subquery may return multiple results

```
select * from Beers
where (name, manf) in
           (select name, manf
            from Beers
            where manf = 'Anheuser-Busch');
```

Example

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

```
SELECT name, manf
FROM Beers
WHERE name IN (
     SELECT beer FROM Likes WHERE drinker = 'Steve' or
drinker = 'Bill'
);
```

The set of beers Steve or Bill likes

Without subquery

- Does this query produce the same result?

```
SELECT name, manf
FROM Beers b, Likes l
WHERE b.name = l.beer
    and l.drinker = 'Steve' or l.drinker = 'Bill';
```

Correct equivalent subquery

- Note the "distinct" and grouping (using parentheses) of two conditions on drinker

```
SELECT distinct name, manf
FROM Beers b, Likes l
WHERE b.name = l.beer
    and (l.drinker = 'Steve' or l.drinker = 'Bill');
```

Introduced by comparison operators

- <comparison operator> <any/all> (subquery)
  - Comparison operators: =, !=, <, >, <=, >=, <>
- Examples
  - x >= all (subquery)
  - x <= all (subquery)
  - x = any (subquery) // equivalent to "x in (subquery)"
  - x != all (subquery) // equivalent to "x not in (subquery)"
- Is "x != any (subquery)" equivalent to "x not in (subquery)"?

Example

- From Sells(bar, beer, price), find the beer(s) sold for the highest price.
- What about beers with "the lowest price"?

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

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

Introduced by "exists" or ”in"

- Exists
  - exists (subquery): evaluated to true if subquery has at least one result
- IN
  - X in (subquery): evaluated to true if element X is in table resulting from subquery.
- NOT EXISTS and NOT IN also possible

Example Query with EXISTS

- Where does this query do?

```
select name
from Beers b1
where not exists (
            select name
            from Beers b2
            where  b2.name <> b1.name and b2.manf = b1.manf);
```

Often Interchangable…

```
-- IN
SELECT *
FROM Employee
WHERE DeptName IN (
  SELECT DeptName
  FROM Dept
)
```

```
-- EXISTS
SELECT *
FROM Employee
WHERE EXISTS (
  SELECT 1
  FROM Dept WHERE
     Employee.DeptName = Dept.DeptName
)
```

IN vs. Exists

While there is no general rule as to whether you should prefer IN or EXISTS, but:

- IN predicates tend to be more readable than EXISTS predicates
- EXISTS predicates tend to be more expressive than IN predicates (i.e. it is easier to express very complex SEMI JOIN)
- There is no formal difference in performance. There may, however, be a huge performance difference on some databases.

Agenda

- SQL DML (Data Manipulation Language)
  - SQL query
  - Relations as bags
  - Joins
  - Grouping and aggregation
  - Database modification

- SQL DDL (Data Definition Language)
  - Define schema

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

Set Operations on Bags

- Union: {a,b,b,c} U {a,b,b,b,e,f,f} = {a,a,b,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} ∩ {b,b,c,c,c,c,d} = {b,b,c,c}
  - minimum of the two numbers of occurrences

How to implement them?

- Bag union?
- Bag intersection?
- Bag difference?

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)

Set Semantics for Set Operations

- 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.
- 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.
- And since intersection and difference uses set semantics, union uses it too

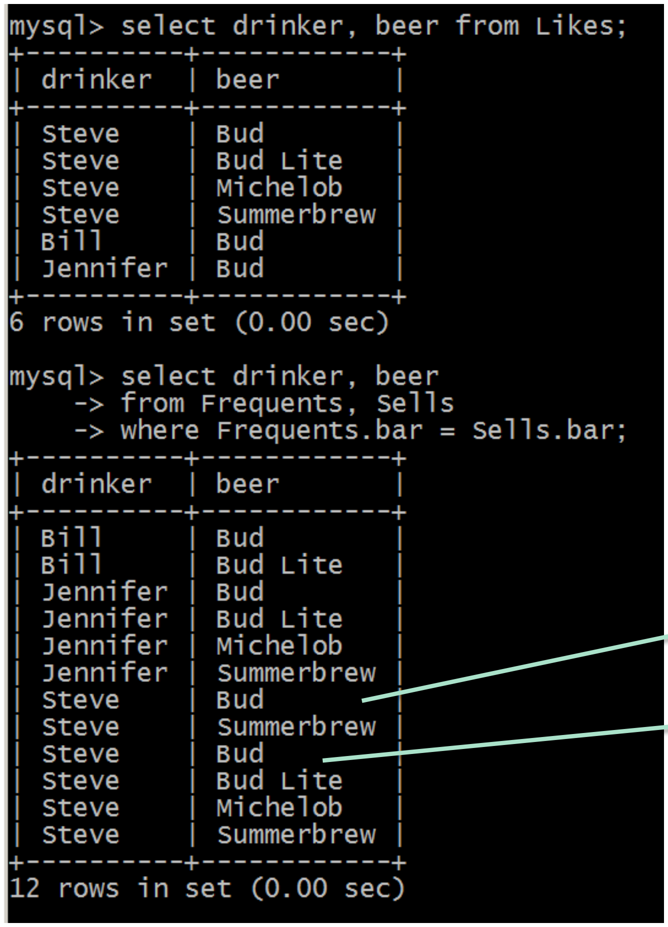
Example

- 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, or
  2. The drinker frequents at least one bar that sells the beer.

Query

```
Select drinker, beer
From Likes
Union
Select drinker, beer
From Frequents, Sells
Where Frequents.bar = Sells.bar;
```

Individual results



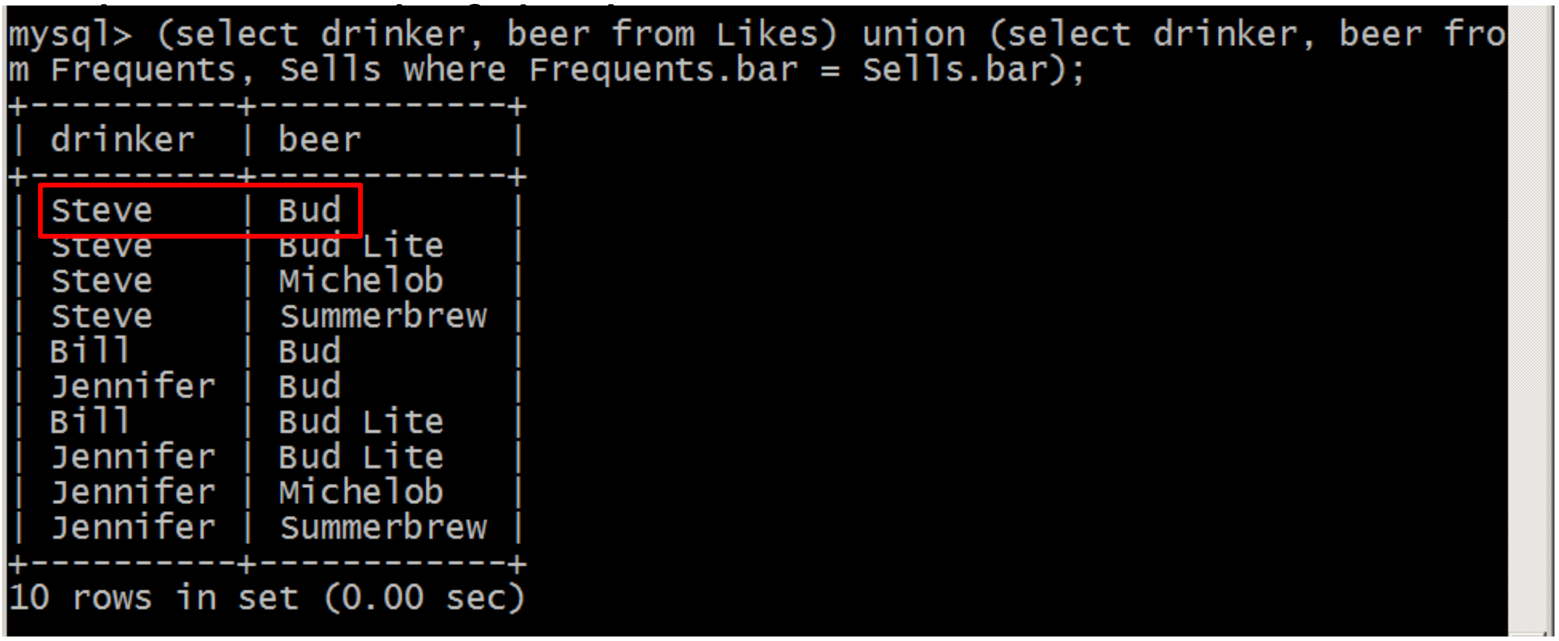
```
mysql> select drinker, beer from Likes;
+----------+------------+
| drinker  | beer       |
+----------+------------+
| Steve    | Bud        |
| Steve    | Bud Lite   |
| Steve    | Michelob   |
| Steve    | Summerbrew |
| Bill     | Bud        |
| Jennifer | Bud        |
+----------+------------+
6 rows in set (0.00 sec)

mysql> select drinker, beer
    -> from Frequents, Sells
    -> where Frequents.bar = Sells.bar;
+----------+------------+
| drinker  | beer       |
+----------+------------+
| Bill     | Bud        |
| Bill     | Bud Lite   |
| Jennifer | Bud        |
| Jennifer | Bud Lite   |
| Jennifer | Michelob   |
| Jennifer | Summerbrew |
| Steve    | Bud        |
| Steve    | Summerbrew |
| Steve    | Bud        |
| Steve    | Bud Lite   |
| Steve    | Michelob   |
| Steve    | Summerbrew |
+----------+------------+
12 rows in set (0.00 sec)
```

(Steve, Bud)
appears twice

Union result

- No



```
mysql> (select drinker, beer from Likes) union (select drinker, beer fro
m Frequents, Sells where Frequents.bar = Sells.bar);
+----------+-----------+
| drinker  | beer      |
+----------+-----------+
| Steve    | Bud       |
| Steve    | Bud Lite  |
| Steve    | Michelob  |
| Steve    | Summerbrew|
| Bill     | Bud       |
| Jennifer | Bud       |
| Bill     | Bud Lite  |
| Jennifer | Bud Lite  |
| Jennifer | Michelob  |
| Jennifer | Summerbrew|
+----------+-----------+
10 rows in set (0.00 sec)
```

Controlling Duplicate Elimination

- Force the result to be a set by
  SELECT DISTINCT . . .
  - May distinct multiple attributes
  - E.g., select distinct a, b, c

- 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.

Union all

- "Union all" returns all duplicates

```
(select drinker, beer
from Likes)
union all
(select drinker, beer
from Frequents, Sells
where Frequents.bar = Sells.bar);
```

Alternative to EXCEPT

- Only work for set-based
  - (i.e., no duplicates in A & B)
- A – B => A – (A & B) => select A … where not exists (A & B)

```
select drinker, beer
from Likes k
where not exists
  (select drinker, beer
   from Frequents, Sells
   where Frequents.bar = Sells.bar
         and Frequents.drinker = k.drinker
         and Sells.beer = k.beer);
```

Another example

- B - A

Distinct is needed here for set-semantics

```
select distinct drinker, beer
from Frequents, Sells
where Frequents.bar = Sells.bar
and not exists
  (select drinker, beer
   from Likes k
   where Frequents.drinker = k.drinker
          and Sells.beer = k.beer);
```

Alternative to intersect

- Only work for set-based (i.e., no duplicates in A & B)
- A intersect B => select A … where exists (B & A)

```
select drinker, beer
from Likes k
where exists
  (select drinker, beer
   from Frequents, Sells
   where Frequents.bar = Sells.bar
          and Frequents.drinker = k.drinker
          and Sells.beer = k.beer);
```

Agenda

- SQL DML (Data Manipulation Language)
  - SQL query
  - Relations as bags
  - Joins
  - Grouping and aggregation
  - Database modification

- SQL DDL (Data Definition Language)
  - Define schema

Join Expressions

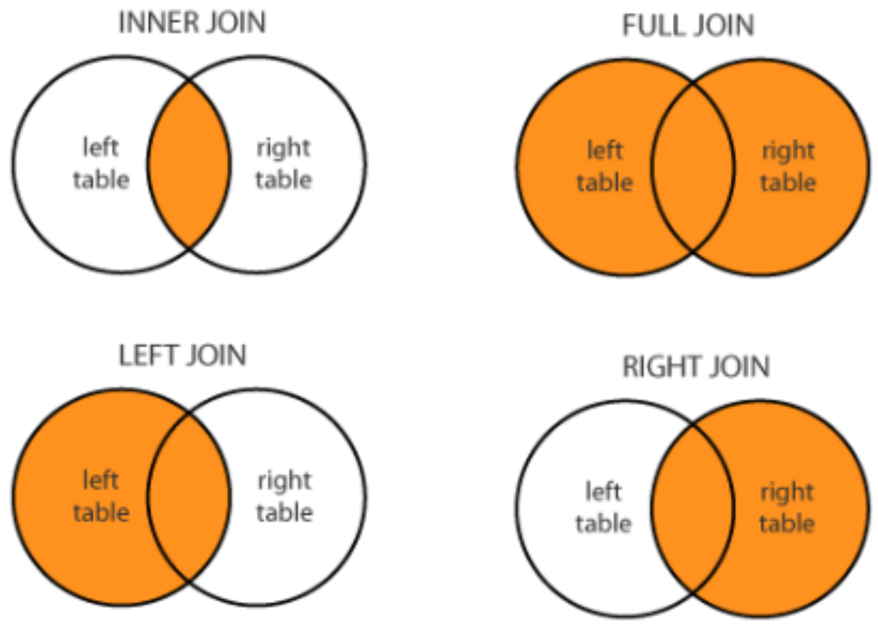
- SQL provides a number of join expressions
  - But using bag semantics, not the set semantics.

- These expressions can be used in place of relations in a FROM clause.

Different Types Of Joins

- (INNER) JOIN: Select records that have matching values in both tables.
- LEFT (OUTER) JOIN: Select records from the first (left-most) table with matching right table records.
- RIGHT (OUTER) JOIN: Select records from the second (right-most) table with matching left table records.
- FULL (OUTER) JOIN: Selects all records that match either left or right table records.

Different Types of Joins



Cross products/joins

- Cross product:

  select * from `Likes, Sells;`

  select * from `Likes join Sells;`

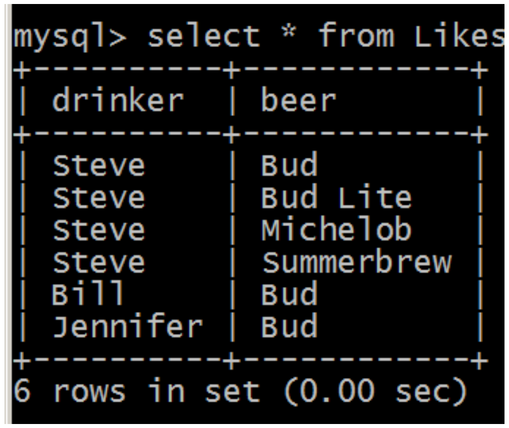
  select * from `Likes cross join Sells;`
- Relations can be parenthesized subqueries, as well.

Natural join

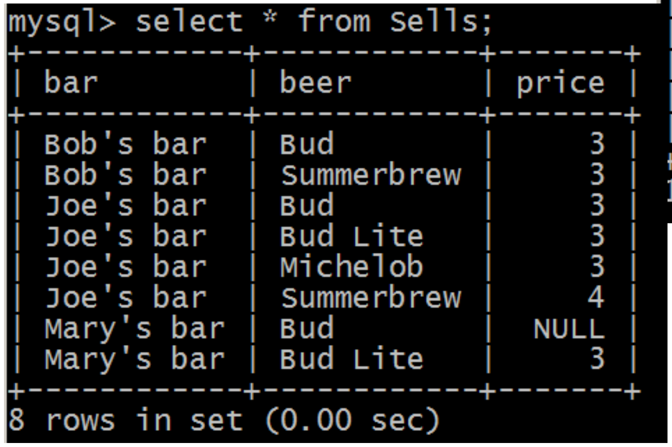
- Two tuples naturally join if they have the same value on the common attributes

Natural join

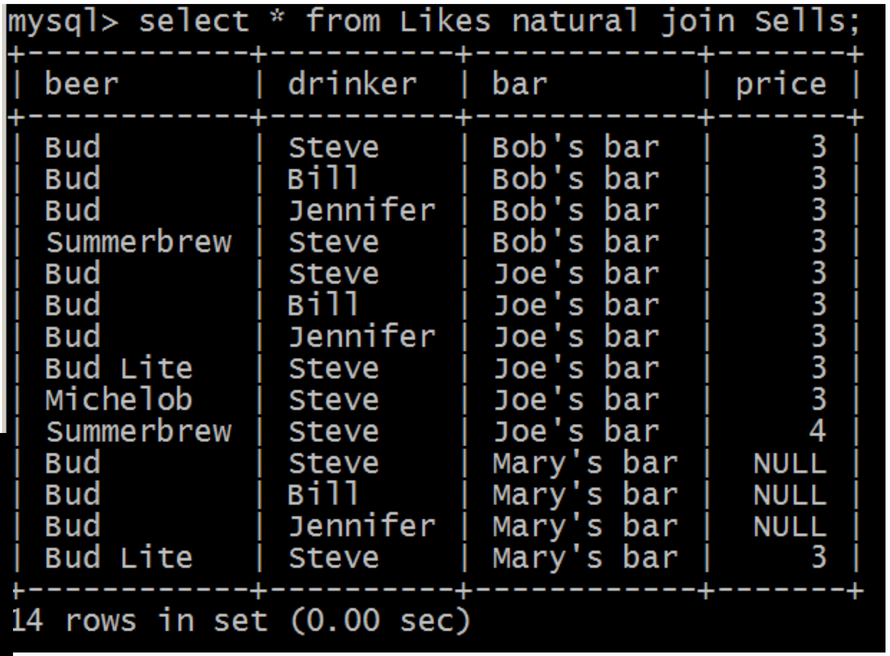
select * from `Likes NATURAL JOIN Sells;`



```
mysql> select * from Likes
+----------+------------+
| drinker  | beer       |
+----------+------------+
| Steve    | Bud        |
| Steve    | Bud Lite   |
| Steve    | Michelob   |
| Steve    | Summerbrew |
| Bill     | Bud        |
| Jennifer | Bud        |
+----------+------------+
6 rows in set (0.00 sec)
```



```
mysql> select * from Sells;
+------------+------------+-------+
| bar        | beer       | price |
+------------+------------+-------+
| Bob's bar  | Bud        |     3 |
| Bob's bar  | Summerbrew |     3 |
| Joe's bar  | Bud        |     3 |
| Joe's bar  | Bud Lite   |     3 |
| Joe's bar  | Michelob   |     3 |
| Joe's bar  | Summerbrew |     4 |
| Mary's bar | Bud        |  NULL |
| Mary's bar | Bud Lite   |     3 |
+------------+------------+-------+
8 rows in set (0.00 sec)
```



```
mysql> select * from Likes natural join Sells;
+------------+----------+------------+-------+
| beer       | drinker  | bar        | price |
+------------+----------+------------+-------+
| Bud        | Steve    | Bob's bar  |     3 |
| Bud        | Bill     | Bob's bar  |     3 |
| Bud        | Jennifer | Bob's bar  |     3 |
| Summerbrew | Steve    | Bob's bar  |     3 |
| Bud        | Steve    | Joe's bar  |     3 |
| Bud        | Bill     | Joe's bar  |     3 |
| Bud        | Jennifer | Joe's bar  |     3 |
| Bud Lite   | Steve    | Joe's bar  |     3 |
| Michelob   | Steve    | Joe's bar  |     3 |
| Summerbrew | Steve    | Joe's bar  |     4 |
| Bud        | Steve    | Mary's bar |  NULL |
| Bud        | Bill     | Mary's bar |  NULL |
| Bud        | Jennifer | Mary's bar |  NULL |
| Bud Lite   | Steve    | Mary's bar |     3 |
+------------+----------+------------+-------+
14 rows in set (0.00 sec)
```

Theta Join

- R JOIN S ON <condition> is a theta-join, using <condition> for selection.
- Example: using Drinkers(name, addr, phone) and Frequents(drinker, bar):

  select * from `Drinkers JOIN Frequents ON name = drinker;`

  gives us all (*n, a, p, d, b*) quadruples such that drinker n/*d* lives at address *a,* has phone *p,* and frequents bar *b*.

Expressing natural join using theta join

- select * from `Likes NATURAL JOIN Sells;`
- Same as the following or not?
- select * from `Likes JOIN Sells on Likes.beer= Sells.beer;`

Motivation for Outerjoins

Explicit joins in SQL:

```
 Product(name, category)
 Purchase(prodName, store)

SELECT Product.name, Purchase.store
FROM    Product JOIN Purchase ON
            Product.name = Purchase.prodName
```

Same as:

```
 SELECT Product.name, Purchase.store
 FROM    Product, Purchase
 WHERE   Product.name = Purchase.prodName
```

But Products that were never sold will be lost !

Outer Joins

- Left outer join:
  - Include the left tuple even if there's no match
- Right outer join:
  - Include the right tuple even if there's no match
- Full outer join:
  - Include the both left and right tuples even if there's no match

Null Values and Outerjoins

Left outer joins in SQL:

Product(name, category)
Purchase(prodName, store)

```
SELECT Product.name, Purchase.store
FROM   Product LEFT OUTER JOIN Purchase ON
          Product.name = Purchase.prodName
```

Product

Name	Category
Gizmo	Gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
Iphone	AT&T

	Name	Category	ProdName	Store
Inner join	Gizmo	Gadget	Gizmo	Wiz
	Camera	Photo	Camera	Ritz
	Camera	Photo	Camera	Wiz
Left outer	OneClick	Photo	NULL	NULL
Right outer	NULL	NULL	Iphone	AT&T

Left Outer Join

Product(name, category)
Purchase(prodName, store)

```
SELECT Product.name, Purchase.store
FROM    Product LEFT OUTER JOIN Purchase ON
           Product.name = Purchase.prodName
```

Name	Category	ProdName	Store
Gizmo	Gadget	Gizmo	Wiz
Camera	Photo	Camera	Ritz
Camera	Photo	Camera	Wiz
OneClick	Photo	NULL	NULL
NULL	NULL	Iphone	AT&T

Name	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

Find Non-joining Tuples

```
SELECT Product.name, Purchase.store
FROM    Product LEFT OUTER JOIN Purchase ON
        Product.name = Purchase.prodName
WHERE Purchase.store IS NULL
```

Name	Category	ProdName	Store
Gizmo	Gadget	Gizmo	Wiz
Camera	Photo	Camera	Ritz
Camera	Photo	Camera	Wiz
OneClick	Photo	NULL	NULL
NULL	NULL	Iphone	AT&T

Name	Store
OneClick	NULL

Semi-Join

- Let’s imagine we want authors who actually also have books. Then we can write:

```
-- Using IN
FROM author
WHERE author.id IN (SELECT book.author_id FROM book)
-- Using EXISTS
FROM author
WHERE EXISTS (
    SELECT 1 FROM book WHERE book.author_id = author.id
)
```

Agenda

- SQL DML (Data Manipulation Language)
  - SQL query
  - Relations as bags
  - Grouping and aggregation
  - Database modification
- SQL DDL (Data Definition Language)
  - Define schemas

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.

Example: 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

- DISTINCT inside an aggregation causes duplicates to be eliminated before the 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 of a specific column (e.g., count(price)), and can never be the minimum or maximum of a column.
- If there are no non-NULL values in a column, then the result of the aggregation is NULL.

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;
```

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;
```

Restriction on 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.
  - Why?

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.

Requirements on HAVING Conditions

- These conditions may refer to any relation or tuple-variable in the FROM clause.
- They may refer to attributes of those relations, as long as the attribute is either:
    1. A grouping attribute, or
    2. Aggregated.

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.

Solution

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

```
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.

Any problem?

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

Agenda

- SQL DML (Data Manipulation Language)
  - SQL query
  - Relations as bags
  - Grouping and aggregation
  - Database modification

- SQL DDL (Data Definition Language)
  - Define schemas

Database Modifications

- A modification command does not return a result as a query does, but it changes the database in some way.
- There are three kinds of modifications:

1. *Insert* a tuple or tuples.
2. *Delete* a tuple or tuples.
3. *Update* the value(s) of an existing tuple or tuples.

Insertion

- To insert a single tuple:

  INSERT INTO <relation>

  VALUES (<list of values>);
- Recall Cartesian-product view of relation

- Example: add to Likes(drinker, beer) the fact that Sally likes Bud.

```
INSERT INTO Likes
VALUES('Sally', 'Bud');
```

Specifying Attributes in INSERT

- We may add to the relation name a list of attributes.
- Recall set-of-functions view of relation
- There are two reasons to do so:
  1. We forget the standard order of attributes for the relation.
  2. We don't have values for all attributes, and we want the system to fill in missing components with NULL or a default value.

Example: Specifying Attributes

- Another way to add the fact that Sally likes Bud to Likes(drinker, beer):

```
INSERT INTO Likes(beer, drinker)
VALUES('Bud', 'Sally');
```

Inserting Many Tuples

- We may insert the entire result of a query into a relation, using the form:

```
INSERT INTO <relation>
( <subquery> );
```

Example: Insert Using a Subquery

- Using Frequents(drinker, bar), enter into the new relation PotBuddies(name) all of Sally's “potential buddies,” i.e., those drinkers who frequent at least one bar that Sally also frequents.

Solution

The other drinker

```
INSERT INTO PotBuddies
(SELECT d2.drinker
 FROM Frequents d1, Frequents d2
 WHERE d1.drinker = 'Steve' AND
   d2.drinker <> 'Steve' AND
   d1.bar = d2.bar
);
```

Pairs of Drinker tuples where the first is for Steve, the second is for someone else, and the bars are the same.

Deletion

- To delete tuples satisfying a condition from some relation:

```
DELETE FROM <relation>
WHERE <condition>;
```

Example: Deletion

- Delete from Likes(drinker, beer) the fact that Sally likes Bud:

```
DELETE FROM Likes
WHERE drinker = 'Steve' AND
  beer = 'Bud';
```

Example: Delete all Tuples

- Make the relation Likes empty:

```
DELETE FROM Likes;
```

- Note no WHERE clause needed.

Example: Delete Many Tuples

- Delete from Beers(name, manf) all beers for which there is another beer by the same manufacturer.

```
DELETE FROM Beers b
WHERE EXISTS (
  SELECT name FROM Beers
  WHERE manf = b.manf AND
   name <> b.name);
```

Beers with the same manufacturer and a different name from the name of the beer represented by tuple b.

Semantics of Deletion

- Suppose Anheuser-Busch makes only Bud and Bud Lite.
- Suppose we come to the tuple *b* for Bud first.
- The subquery is nonempty, because of the Bud Lite tuple, so we delete Bud.
- Now, when *b* is the tuple for Bud Lite, do we delete that tuple too?

Semantics of Deletion

- The answer is that we *do* delete Bud Lite as well.
- The reason is that deletion proceeds in two stages:

1. Mark all tuples for which the WHERE condition is satisfied in the original relation.
2. Delete the marked tuples.

```
SELECT * FROM Beers b
WHERE EXISTS (
     SELECT name FROM Beers
     WHERE manf = b.manf AND
name <> b.name);
```

Updates

- To change values of certain attributes in certain tuples of a relation:

```
UPDATE <relation>
SET <list of attribute assignments>
WHERE <condition on tuples>;
```

Example: Update

- Change drinker Fred's phone number to 555-1212:

```
UPDATE Drinkers
SET phone = '555-1212'
WHERE name = 'Fred';
```

Example: Update Several Tuples

- Make $4 the maximum price for beer:

```
UPDATE Sells
SET price = 4.00
WHERE price > 4.00;
```

Agenda

- SQL DML (Data Manipulation Language)
  - SQL query
  - Relations as bags
  - Grouping and aggregation
  - Database modification
- SQL DDL (Data Definition Language)
  - Define schemas

Defining a Database Schema

- A database schema comprises declarations for the relations ("tables") of the database.
- Many other kinds of elements may also appear in the database schema, including views, indices, and triggers.

Declaring a Relation

- Simplest form is:

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

- And you may remove a relation from the database schema by:

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

Elements of Table Declarations

- The principal element is a pair consisting of an attribute and a 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
);
```

Declaring Keys

- An attribute or list of attributes may be declared PRIMARY KEY or UNIQUE.
- These each say the attribute(s) so declared functionally determine all the attributes of the relation schema.
- There are a few distinctions to be mentioned later.

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 Versus UNIQUE

- The SQL standard allows DBMS implementers to make their own distinctions between PRIMARY KEY and UNIQUE.
  - Example: some DBMS might automatically create an *index* (data structure to speed search) in response to PRIMARY KEY, but not UNIQUE.

Required Distinctions

- However, standard SQL requires these distinctions:
  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.

Other Declarations for Attributes

- Two other declarations we can make for an attribute are:
  1. NOT NULL means that the value for this attribute may never be NULL.
  2. DEFAULT <value> says that if there is no specific value known for this attribute's component in some tuple, use the stated <value>.

Example: Default Values

```
CREATE TABLE Drinkers (
 name CHAR(30) PRIMARY KEY,
 addr CHAR(50)
   DEFAULT '123 Sesame St.',
 phone CHAR(16)
);
```

Effect of Defaults

- Suppose we insert the fact that Sally is a drinker, but we know neither her address nor her phone.
- An INSERT with a partial list of attributes makes the insertion possible:

```
INSERT INTO Drinkers(name)
VALUES('Sally');
```

Effect of Defaults

- But what tuple appears in Drinkers?

name	addr	phone
'Sally'	'123 Sesame St'	NULL

- If we had declared phone NOT NULL, this insertion would have been rejected.

Adding Attributes

- We may change a relation schema by adding a new attribute ("column") by:

```
ALTER TABLE <name> ADD
    <attribute declaration>;
```

- Example:

```
ALTER TABLE Bars ADD
phone CHAR(16) DEFAULT 'unlisted';
```

Deleting Attributes

- Remove an attribute from a relation schema by:

```
ALTER TABLE <name>
    DROP <attribute>;
```

- Example: we don't really need the license attribute for bars:

```
ALTER TABLE Bars DROP license;
```