SQL

DSCI 551

Wensheng Wu

SQL

- One-relation
  - Select A's, stats (max/min/avg/sum/count)

    from R

    where C

    group by A

    having count(*) > 5

    order by A

    limit 2

    offset 3
  - distinct, like

SQL

- Multi-relational
  - Join:
    - Natural join, theta join, (left, right, full) outer join, inner join, cross join
  - Set operations:
    - Union, intersect, except
    - Q1 union Q2, Q1 except Q2, Q2 except Q1
- Subquery
  - A != (Q)
  - A (not) in (Q)
  - A >= all/any (Q)
  - (not) exists (Q)
- CTE (common table expression)

SQL Introduction

create type address (street_name, street_no, city, state, zip);
create table person(id int, addr address, resume XML); // sql server

Standard language for querying and manipulating data

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

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

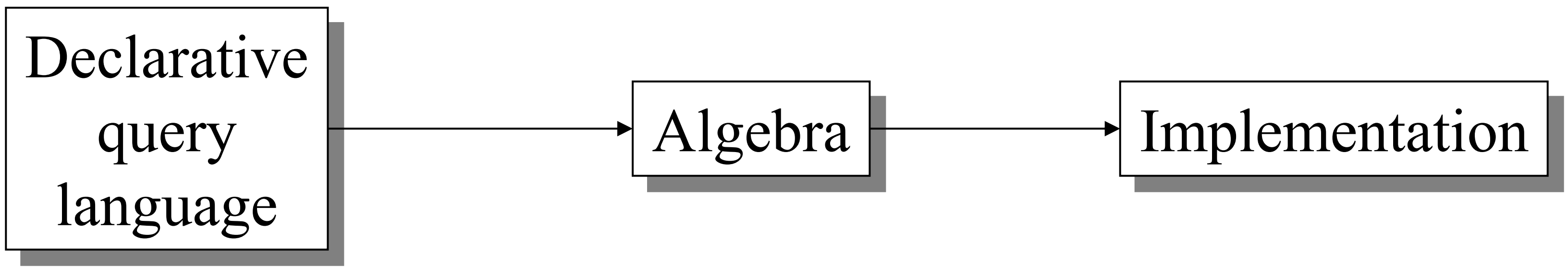
Vendors support various subsets of these.

Note: alternative name: Sequel (**S**tructured **E**nglish **QU**ery **L**anguage)
from IBM project in 70's
SQL 99: object-relational features (e.g., user-defined types), recursion
(supported in MySQL 8.0 using CTE)
SQL 2003: XML (XML type columns are not supported in MySQL)

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
(selection, projection, join, group by)
Relational bag algebra

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

Relational Algebra

- Selection (σ):
  - $\sigma_{\text{GNP} > 1000}(\text{country})$
- Join ($\bowtie$):
  - $\text{country} \bowtie_{\text{country.Capital} = \text{city.ID}} \text{city}$
- Projection (π):
  - $\pi_{\text{GNP}}(\text{country})$

Relational Algebra

- Group by (γ)
  - $\gamma_{\text{Continent, avg(LifeExpectancy)}\rightarrow\text{count(*)}> 5}(\text{country})$

```
select Continent, avg(LifeExpectancy) avg_le
from country
group by Continent
having count(*) > 5
```

- Distinct (δ)
  - $\delta_{\text{Continent, Region}}(\text{country})$

```
select distinct Continent, Region
from country
```

Relational algebra

- Set/bag operations
  - union: $\cup$, $\cup_b$
  - intersect: $\cap$, $\cap_b$
  - except: $-$, $-_b$

```
(select Language
 from countrylanguage
 where CountryCode = 'USA')
union all
(select Language
 from countrylanguage
 where CountryCode = 'CAN')
```

SQL to Relational Algebra

Declarative

select country.name, city.name
from country, city
where country.GNP > 10000 and
country.Capital = city.ID

(additional clauses ?)

➔ Relational algebra

Procedural

$$\Pi_{\text{country.name, city.name}} (\sigma_{\text{GNP} > 10000}(\text{country}) \bowtie_{\text{country.Capital} = \text{city.ID}} (\text{city}))$$

SQL clauses

- Select continent, max(GNP)
- From country, city
- Where population > 10000 and country…= city…
- Group by continent => $group: {_id: "…"}
- Having count(*) > 5 =>
- Order by continent desc
- Limit 10
- Offset 10

SQL to Relational Algebra

Declarative

select country.name, city.name
from country, city
where country.GNP > 1000 and
    country.Capital = city.ID

➔ Relational algebra (RA)

Procedural

$\Pi_{\text{country.name, city.name}}$

$\bowtie_{\text{country.Capital = city.ID}}$

$\sigma_{\text{GNP > 1000}}$

country

city

algebra: operand and operator

projection: country[['name']]
filtering: country[country.GNP > 1000]
join: merge

```
country1 = country[country.GNP > 1000]
country_city = country1.merge(city,
                   left_on = Capital,
                   right_on = ID)
country_city[[name_x, name_y]]
```

SQL to Dataframes

Declarative

```
select country.name, city.name
from country, city
where country.GNP > 1000 and
        country.Capital = city.ID
```

➔ Pandas dataframe

Procedural

```
country[country.GNP > 1000].\
merge(city, left_on = 'Capital', \
        right_on  = 'ID')\
[['Name_x', 'Name_y']]
```

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

Single-Relation Queries

Our Running Example

- Most of our SQL queries will be based on the following database schema.
  - Underline indicates key attributes.

Beers(<u>name</u>, manf)

Bars(<u>name</u>, addr, license)

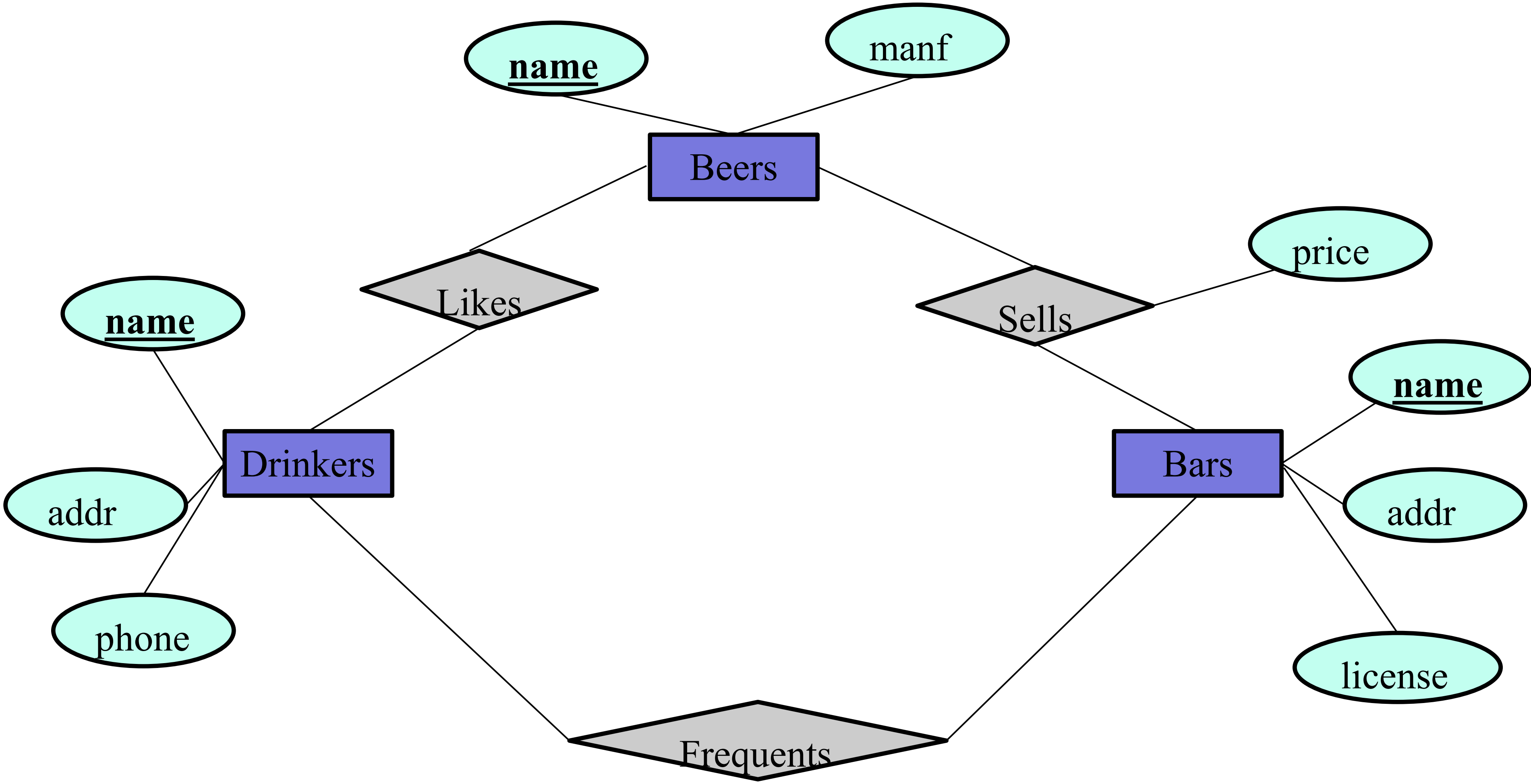
Drinkers(<u>name</u>, addr, phone)

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

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

Frequents(<u>drinker</u>, <u>bar</u>)

ER Diagram



name
manf
Beers
price
Likes
Sells
name
name
Drinkers
Bars
addr
addr
phone
license
Frequents

Tables

Beers

name	manf
Bud	Anheuser-Busch
Bud Lite	Anheuser-Busch
Michelob	Anheuser-Busch
Summerbrew	Pete's
Budweiser	Heineken

Drinkers

name	addr	phone
Bill	Jefferson St.	213-555-0101
Jennifer	Maple St.	626-552-1234
Steve	Vermont St.	213-555-1234
David	Vermont Ave.	310-384-3829

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	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
Mary's bar	Budweiser	2

Frequents

drinker	bar
Bill	Mary's bar
Steve	Bob's bar
Steve	Joe's bar
Jennifer	Joe's bar
David	Joe's bar

Some observations

- On single table
  - Pete’s and Heineken produces only one beer
  - Steve likes 4 different beers; all others single beer
  - Summerbrew/Budweiser is most expensive/cheap

- On multiple tables
  - Nobody likes Budweiser (in Beers, not in Likes)
  - David does not like any beers, but frequents bars (in Frequents, not in Likes)
  - CanDrink: drinker frequents bars which sell beers

Formulating queries (single table)

- Pete’s and Heineken produces only one beer
  - Using “not exists” subquery
  - Or finding manufacturers who produces more than one beer (using exists/self join) and takes complement (using “not in” or “!= all” subquery)
  - (note that “in” equivalent to “= any”)

- Summerbrew/Budweiser is most expensive/cheap
  - Using “>=“ all or “<= all”
  - Be careful with NULL values

Formulating queries (multiple tables)

- Nobody likes Budweiser (in Beers, not in Likes)
  - Using "not in" subquery

- David does not like any beers (in Drinkers, but not in Likes)
  - select name from Drinkers where name not in (select drinker from Likes);

- May also use "outer join"

Formulating queries (multiple tables)

- David does not like any beers, but frequents bars (in Frequents, not in Likes)
  – select drinker from Frequents natural left outer join Likes where beer is NULL;

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.

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';
```

String may also be double quoted, e.g., "Joe's Bar"

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., concat() for string concatenation in mysql, and lower() for lowering case).
- Lexicographic order on strings (e.g., name >= 'j').
- Pattern matching: s LIKE p
- Special stuff for comparing dates and times.

Example

- select concat(name, " made by ", manf) from Beers;

concat(name, " made by ", manf)
Bud made by Anheuser-Busch
Bud Lite made by Anheuser-Busch
Budweiser made by Anheuser-Busch
Michelob made by Anheuser-Busch
Summerbrew made by Pete's

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 expressions are built.
- SQL is **NOT** *case-sensitive*. In general, upper and lower case characters are the same, except inside quoted strings.

Caveat

- Table names in MySQL (running on Unix-like OS) ARE case-sensitive
  - Reason: table names are used to store metadata & data on the file system

Format/table definition

```
[ec2-user@ip-172-31-18-182 ~]$ sudo ls /var/lib/mysql/inf551 -l
total 700
-rw-rw---- 1 mysql mysql   8590 Oct  9 23:43 Bars.frm
-rw-rw---- 1 mysql mysql  98304 Oct  9 23:44 Bars.ibd
-rw-rw---- 1 mysql mysql   8590 Oct  9 23:43 Beers.frm
-rw-rw---- 1 mysql mysql  98304 Oct  9 23:44 Beers.ibd
-rw-rw---- 1 mysql mysql     65 Oct  9 23:43 db.opt
-rw-rw---- 1 mysql mysql   8622 Oct  9 23:43 Drinkers.frm
-rw-rw---- 1 mysql mysql  98304 Oct  9 23:44 Drinkers.ibd
-rw-rw---- 1 mysql mysql   8594 Oct  9 23:43 Frequents.frm
-rw-rw---- 1 mysql mysql 114688 Oct  9 23:44 Frequents.ibd
-rw-rw---- 1 mysql mysql   8596 Oct  9 23:43 Likes.frm
-rw-rw---- 1 mysql mysql 114688 Oct  9 23:44 Likes.ibd
-rw-rw---- 1 mysql mysql   8620 Oct  9 23:43 Sells.frm
-rw-rw---- 1 mysql mysql 114688 Oct  9 23:44 Sells.ibd
```

InnoDB data

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 _)

Not like

- select * from Sells where beer not like '%Bud%';

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
WHERE price < 2.00 OR price >= 2.00;
```

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

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

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 bar

FROM Sells

WHERE price < 2.00 OR price >= 2.00 OR price IS NULL

Now it includes bars in all Sells tuples

Order by

- select * from Sells order by price desc;

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

Limit n

- select * from Sells limit 5;

```
mysql> select * from Sells limit 5;
+-----------+------------+-------+
| 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 |
+-----------+------------+-------+
5 rows in set (0.00 sec)
```

Offset

- select * from Likes limit 1 offset 1;

Offset of first row to be returned (note offset starts from 0)

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

mysql> select * from Likes limit 1 offset 1;
+----------+------+
| drinker  | beer |
+----------+------+
| Jennifer | Bud  |
+----------+------+
1 row in set (0.00 sec)
```

Example of NLJ

```
select l1.drinker
from Likes l1, Likes l2
Where  l2.drinker = l1.drinker and
         l2.beer != l1.beer
```

```
for l1  in Likes:
   for l2 in Likes:
     if (l2.drinker = l1.drinker &
        l2.beer != l1.beer)
        output l1.drinker
```

```
          +----------+-------------+
          | drinker  | beer        |
          +----------+-------------+
          | Bill     | Bud         |
          | Jennifer | Bud         |
l1        | Steve    | Bud         |
          | Steve    | Bud Lite    |
          | Steve    | Michelob    |
          | Steve    | Summerbrew  |      l2
          +----------+-------------+
```

Example

Take 1:

```
for l in Likes:
    for b in Beers:
        if (l.drinker = 'Steve'    and b.name = l.beer):
            output b.manf
```

Take 2:

```
for l in Likes:
    if (l.drinker = 'Steve'):
        find b in Beers where b.name = l.beer (using an index)
            output b.manf
```

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>"

Example

- select F.drinker drinker1, bar, L.drinker drinker2, beer from Frequents F, Likes L;

Cartesian product

drinker1	bar	drinker2	beer
Steve	Bob's bar	Bill	Bud
Jennifer	Joe's bar	Bill	Bud
Steve	Joe's bar	Bill	Bud
Bill	Mary's bar	Bill	Bud
Steve	Bob's bar	Jennifer	Bud
Jennifer	Joe's bar	Jennifer	Bud
Steve	Joe's bar	Jennifer	Bud
Bill	Mary's bar	Jennifer	Bud
Steve	Bob's bar	Steve	Bud
Jennifer	Joe's bar	Steve	Bud
Steve	Joe's bar	Steve	Bud
Bill	Mary's bar	Steve	Bud
Steve	Bob's bar	Steve	Bud Lite
Jennifer	Joe's bar	Steve	Bud Lite
Steve	Joe's bar	Steve	Bud Lite
Bill	Mary's bar	Steve	Bud Lite
Steve	Bob's bar	Steve	Michelob
Jennifer	Joe's bar	Steve	Michelob
Steve	Joe's bar	Steve	Michelob
Bill	Mary's bar	Steve	Michelob
Steve	Bob's bar	Steve	Summerbrew
Jennifer	Joe's bar	Steve	Summerbrew
Steve	Joe's bar	Steve	Summerbrew
Bill	Mary's bar	Steve	Summerbrew

Example

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

```
SELECT beer
FROM Frequents, Likes
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)
```

Reason

```
mysql> select drinker from Frequents where bar = "Joe's bar";
+----------+
| drinker  |
+----------+
| Jennifer |
| Steve    |
+----------+
2 rows in set (0.00 sec)

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

Formal Semantics

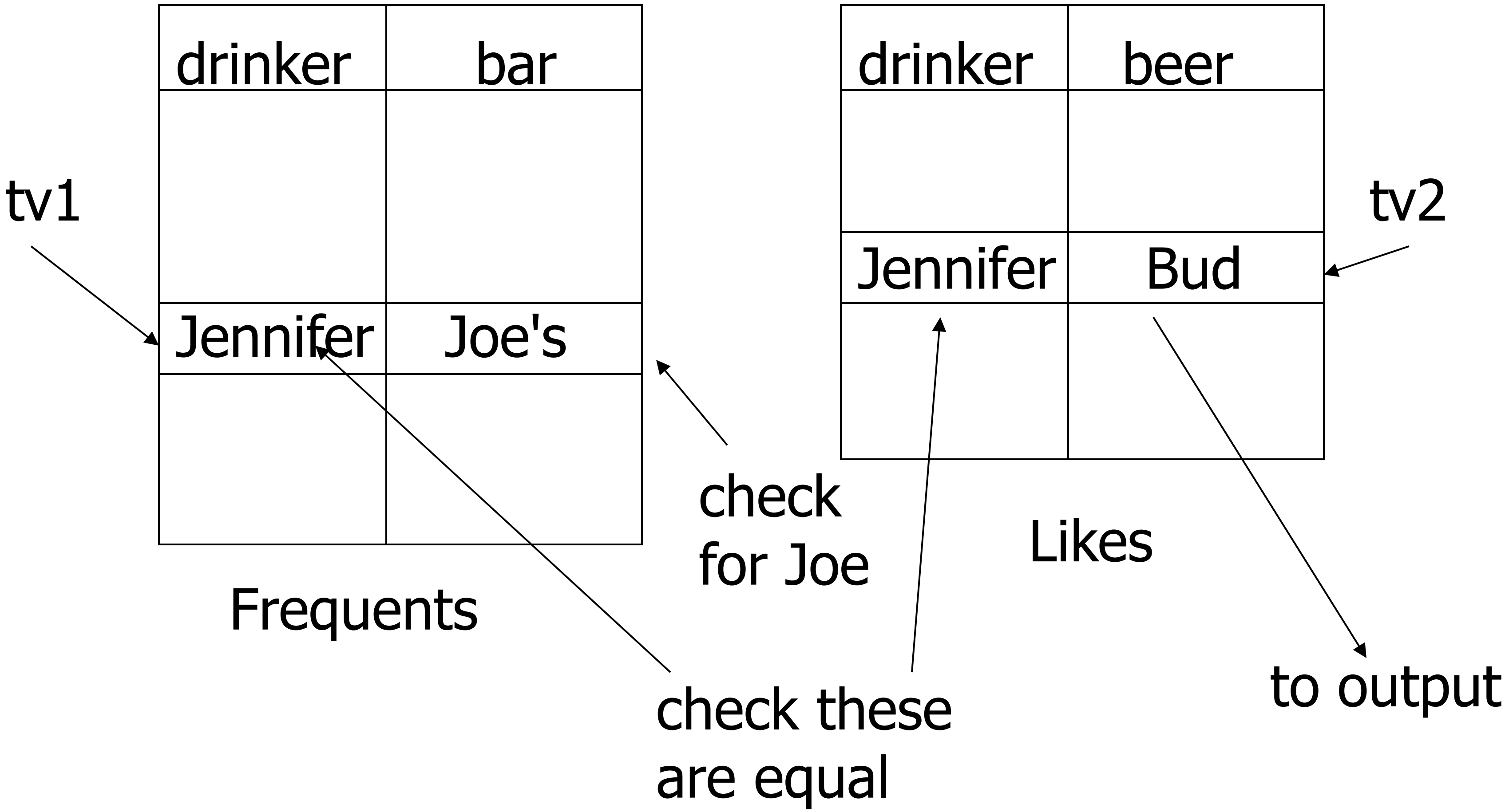
- Almost the same as for single-relation queries:
  1. Start with the product of all the relations in the FROM clause.
  2. Apply the selection condition from the WHERE clause.
  3. Project onto the list of attributes and expressions in the SELECT clause.

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 |
| Budweiser  | Anheuser-Busch |
| Michelob   | Anheuser-Busch |
| Summerbrew | Pete's         |
+------------+----------------+
5 rows in set (0.00 sec)

mysql> SELECT b1.name, b2.name FROM Beers b1, Beers b2 WHERE b1.manf = b2.manf AND
 b1.name < b2.name;
+-----------+-----------+
| name      | name      |
+-----------+-----------+
| Bud       | Bud Lite  |
| Bud       | Budweiser |
| Bud       | Michelob  |
| Bud Lite  | Budweiser |
| Bud Lite  | Michelob  |
| Budweiser | Michelob  |
+-----------+-----------+
6 rows in set (0.00 sec)
```

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 subquery

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

```
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

Example

```
mysql> SELECT beer FROM Likes WHERE drinker = 'Steve' or drinker = 'Bill';
+------------+
| beer       |
+------------+
| Bud        |
| Bud        |
| Bud Lite   |
| Michelob   |
| Summerbrew |
+------------+
```

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
        WHERE price is not NULL);
```

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

Introduced by "exists" or "not exists"

- Both form a boolean expression
- exists (subquery)
  - Evaluated to true if subquery has at least one result
- not exists (subquery)
  - Evaluated to true if subquery has no results

Example Query with EXISTS

- What 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);
```

Additional operators/examples (MySQL)

- between x and y => [x, y]
  - select * from Sells where price between 3 and 4;

- A in (x, y, z) => A = x or A = y or A = z
  - select * from Sells where price in (3, 4);

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 (multi-sets)

- Union (all): {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 (except): {a,b,b,b,c,c} - {b,c,c,c,d} = {a,b,b}
  - subtract the number of occurrences
  - except: {a, b, c} – {b, c, d}= {a}

- 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

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

- Note the removal of duplicates

```
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);
```

MySQL

- Does not support except and intersect!

MySQL will report error!

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

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.

Cross products/joins

- Cross product:

  select * from `Likes, Sells;`

  select * from `Likes join Sells;`

  select * from `Likes cross join Sells;`

- Note that in MySQL, inner join = cross join:

  select * from `Likes inner 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)
```

MySQL Joins

- http://dev.mysql.com/doc/refman/5.7/en/join.html

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 (same as *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?

- select * from `Likes JOIN Sells on Likes.beer= Sells.beer;`

Almost, except for …

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 !

Left Outer Join

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

Full outer join result

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

Outer join

- Left outer join
  - Retain dangling tuples from left relation
- Right outer join
  - Retain dangling tuples from right relation
- Full outer join
  - Retain dangling ones from both relations

Natural outer join

- Add natural before left/right
  - natural left outer join
  - natural right outer join

Example

- create table R (A int, B int);
  - insert into R values(1, 2);
  - insert into R values(2, 3);

A	B
1	2
2	3

- create table S (A int, C int);
  - insert into S values(1, 3);
  - insert into S values(1, 3);
  - insert into S values(3, 4);
  - insert into S values(3, 5);

A	C
1	3
1	3
3	4
3	5

Example

A	B
1	2
2	3

Left dangling

R

join

A	C
1	3
1	3
3	4
3	5

Right dangling

S

```
mysql> select * from R natural join S;
+------+------+------+
| A    | B    | C    |
+------+------+------+
|    1 |    2 |    3 |
|    1 |    2 |    3 |
+------+------+------+
```

Example

- select * from R natural left outer join S;

A	B	C
1	2	3
1	2	3
2	3	NULL

- select * from R natural right outer join S;

A	C	B
1	3	2
1	3	2
3	4	NULL
3	5	NULL

Note different order of attributes in two query results…

Example

- select * from R natural left outer join S
  where C is null;

A	B	C
2	3	NULL

- select R.* from R natural left outer join S
  where C is null;

A	B
2	3

Left dangling

Example: T has the same schema as R

- create table T (A int, B int);
  - insert into T values(1, 2);
  - insert into T values(1, 2);
  - insert into T values(1, 2);
  - insert into T values(2, 4);

A	B
1	2
2	3

R

A	B
1	2
1	2
1	2
2	4

T

- select * from R natural left outer join T ;

Example: T has the same schema as R

- select R.* from R natural left outer join T where T.A is null;

A	B
2	3

- Select * from R natural join T;

Another Example

```
Select l.*, t.*
from Likes l right outer join
        (select Frequents.drinker, Sells.beer
        from Frequents, Sells
        where Frequents.bar = Sells.bar) as t
on l.drinker = t.drinker and l.beer = t.beer
```

Another Example

```
Select t.*
from Likes l right outer join
          (select Frequents.drinker, Sells.beer
          from Frequents, Sells
          where Frequents.bar = Sells.bar) as t
on l.drinker = t.drinker and l.beer = t.beer
Where l.drinker is null
```

Full outer join

- MySQL does not support full outer join

R

A	B
1	2
2	3

Left dangling

join

S

A	C
1	3
1	3
3	4
3	5

Right dangling

- Alternative:

```
Select A, B, C
From R natural left outer join S where C is null
Union all
Select A, B, C
From R natural right outer join S;
```

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 (unless the column has only null values).

- 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)
```

Common table expression

- with t1 as (select drinker from Likes),
  t2 as (select drinker from Frequents)
  select * from t1 natural join t2 ;

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

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

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

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.

Caveat: MySQL does not allow subquery refers to relation to be deleted

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.

Char and Varchar in MySQL

- char(n)
  - Right-pad with spaces to store exactly n characters
  - Trailing spaces are removed when retrieved

- varchar(n)
  - Does not pad with spaces
  - Store length as prefix (one byte if length < 255, otherwise 2 bytes)

Example

Value	**CHAR(4)**	**Storage Required**	**VARCHAR(4)**	**Storage Required**
''	' '	4 bytes	''	1 byte
'ab'	'ab '	4 bytes	'ab'	3 bytes
'abcd'	'abcd'	4 bytes	'abcd'	5 bytes
'abcdefgh'	'abcd'	4 bytes	'abcd'	5 bytes

MySQL Integers

Type	Storage (Bytes)	Minimum Value (Signed/Unsigned)	Maximum Value (Signed/Unsigned)
TINYINT	1	-128	127
		0	255
SMALLINT	2	-32768	32767
		0	65535=2^16-1
MEDIUMINT	3	-8388608	8388607
		0	16777215= 2^24-1
INT	4	-2147483648	2147483647
		0	4294967295=2^32-1
BIGINT	8	-9223372036854775808	9223372036854775807
		0	18446744073709551615

Example: Create Table

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

Dates and Times

- DATE and DATETIME in MySQL

  create table test(a date, b datetime);

  insert into test values('2016-1-1', '2016-1-1 3:10:10')
- The form of a date/datetime value is:

  'yyyy-mm-dd'

  'yyyy-mm-dd hh:mm:ss'

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.

- MySQL creates a B+-tree index for each primary key and unique attribute
  - E.g., show index from Sells;
  - Or: show index in Sells;

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.

Multi-column PK/unique

- create table Test (a int, b int, primary key (a, b));

- create table Test (a int, b int, unique(a, b));

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;
```

Modifying Attributes

- alter table Beers modify name varchar(200);

```
mysql> desc Beers;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| name  | varchar(100) | NO   | PRI |         |       |
| manf  | varchar(100) | NO   | PRI |         |       |
+-------+--------------+------+-----+---------+-------+
2 rows in set (0.01 sec)

mysql> alter table Beers modify name varchar(200);
Query OK, 4 rows affected (0.01 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> desc Beers;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| name  | varchar(200) | NO   | PRI |         |       |
| manf  | varchar(100) | NO   | PRI |         |       |
+-------+--------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

Creating/dropping an index

- create index sells_price_idx on Sells(price);
  – To drop: drop index price_idx on Sells;

```
mysql> show index in Sells;
+-------+------------+-----------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+
---------+---------------+
| Table | Non_unique | Key_name        | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type |
 Comment | Index_comment |
+-------+------------+-----------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+
---------+---------------+
| Sells |          0 | PRIMARY         |            1 | bar         | A         |           8 |     NULL | NULL   |      | BTREE      |
         |               |
| Sells |          0 | PRIMARY         |            2 | beer        | A         |           8 |     NULL | NULL   |      | BTREE      |
         |               |
| Sells |          1 | sells_price_idx |            1 | price       | A         |           8 |     NULL | NULL   | YES  | BTREE      |
         |               |
+-------+------------+-----------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+
```

```
mysql> desc Sells;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| bar   | varchar(100) | NO   | PRI |         |       |
| beer  | varchar(100) | NO   | PRI |         |       |
| price | int(11)      | YES  | MUL | NULL    |       |
+-------+--------------+------+-----+---------+-------+
```

Rename an attribute

- alter table <table name> rename column <old_column_name> to <new column name>;

Query execution plan

- explain select * from Sells where price > 2;

id	select_type	table	partitions	type	possible_keys	key
1	SIMPLE	Sells	NULL	range	price_idx	price_idx

Not using union or subquery

key_len	ref	rows	filtered	Extra
9	NULL	7	100.00	Using where; Using index

Estimate of rows to be examined

- Type range: only rows in a given range are retrieved
- Possible keys: possible indexes to choose
- Key: the index actually chosen
- Using index: index used to find qualified rows in the table

Resources

- EXPLAIN command output format
  - https://dev.mysql.com/doc/refman/5.5/en/explain-output.html