SQL

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, Java.

- What makes SQL viable is that its queries are "optimized" quite well, yielding efficient query executions.

SQL Queries

- Principal form:

SELECT desired attributes
FROM tables (or tuple variables)
WHERE condition over tuple variables;

[Bag Semantics, by default]

SQL Semantics

[Important to understand, esp. to write sub-queries correctly]

- Consider a tuple variable t_i for each relation R_i in the FROM clause. Then, execute the following:

for t_1 in R_1

  for t_2 in R_2

    …..

      If $<t_1, t_2, t_3, …., t_n>$ satisfies the WHERE condition, then output the SELECT attributes of $<t_1, t_2, t_3, …., t_n>$

Example

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

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

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

[Here, technically, Frequents and Likes are tuple variables. The “bar” is an attribute of the implicit tuple variable.]

Star as List of All Attribute

Beers(name, manf)

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

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

Renaming columns

Beers(name, manf)

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

beer
Bud
Bud Lite
Michelob

Example

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';
```

- Note: two single-quotes in a character string represent one single quote.
- Conditions in WHERE clause can use logical operators.

Explicit Tuple Variables

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

- Use explicit tuple variables as aliases of the relations.

Example Beers(name, manf)

- Find pairs of beers by the same manufacturer.

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

- Why do we need (b1.name < b2.name) ?

Subqueries

A query result can be used in the where-clause of another query.

Example: Sells(bar, beer, price)

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

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

- Scoping: An attribute refers to the most closely nested relation.
- Parentheses around subquery are essential.
- **NEXT:** Using subqueries with IN, EXISTS, ANY, ALL operators.

Subqueries: The IN Operator

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

Example

- Find the name and manufacturer of beers that Fred likes.

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

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

- Also: NOT IN.

EXISTS

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

Example: Beers(name, manf)

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

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

- **Scoping:** To refer to outer Beers in the inner subquery, we need to create an explicit tuple variable b1.
- A subquery that refers to values from a surrounding query is called a *correlated subquery*.

Quantifiers – ANY, ALL

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

Example

Sells(bar, beer, price)

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

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

Class Problem

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

Union, Intersection, Difference

- "(subquery) UNION (subquery)" produces the union.
- Similarly, INTERSECT, EXCEPT.
  - Oracle uses MINUS instead of EXCEPT.

Example Likes(drinker, beer); Sells(bar, beer, price); Frequents(drinker, bar)

- Find the drinkers and beers such that the drinker likes the beer and frequents a bar that serves it.

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

Forcing Set/Bag Semantics

- Default for select-from-where is bag;
  - Force set semantics using 'SELECT DISTINCT'

- Default for union, intersection, or difference is set.
  - Force bag semantics using 'UNION ALL' etc.

Example: Sells(bar, beer, price)

- Find the different prices for beers.

```
SELECT DISTINCT price
FROM   Sells;
```

Aggregations

Recall the aggregate operator $\gamma_{A,\,F(B)}(R)$.

Equivalent SQL:

```
SELECT    A, F(B)
FROM      R
GROUP BY A
```

Example Sells(bar, beer, price)

Find the average sales price for each beer.

```
SELECT    beer, AVG(price)
FROM      Sell
GROUP BY beer;
```

Aggregation Example

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

Note: grouping occurs after the ×, σ operations.

Illegal Aggregation – 1

Sells(bar, beer, price)

```
SELECT    bar, beer SUM(price)
FROM      Sells
WHERE     beer = 'Bud‘
GroupBy   bar;
```

Illegal. Why?

Illegal Aggregation – 2

Sells(bar, beer, price)

- Find the bar that sells Bud the cheapest.

```
SELECT    bar, MIN(price)
FROM      Sells
WHERE     beer = 'Bud';
```

- **Illegal**. Why?
- Rule: *Each* element of a `SELECT` clause must either be aggregated or appear in a group-by clause.

Problem: How would we find that bar?

HAVING clause

- HAVING clauses are selections on groups, after grouping and aggregation has been done.

Beers(name, manf); Sells(bar, beer, price)

- Find the average price of those beers that are either served in at least 3 bars or manufactured by Busch.

```
SELECT    beer, AVG(price)
FROM      Sells
GROUP BY  beer
HAVING    COUNT(*) >= 3   OR
          beer IN (SELECT name
                   FROM   Beers
                   WHERE manf = 'Busch');
```

Order of Evaluation

- FROM and WHERE (to get an intermediate table)
- GROUP BY
- HAVING
- SELECT

DB Modifications

- *Modification* = insert, delete, or update.

Syntax

- INSERT INTO relation VALUES (list of values).
- INSERT INTO relation (subquery).
- DELETE FROM relation WHERE condition
- UPDATE relation SET assignments
  WHERE condition.

Defining a Database Schema

CREATE TABLE name (list of elements).

Elements: attributes and their types; key declarations; constraints.

- CREATE *X* for views, indexes, assertions, triggers.
- DROP *X* name deletes the element of kind *X* of that name.

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