

SQL: Data Manipulation Language

csc343, winter 2011

Diane Horton

University of Toronto

Based on slides by Jeff Ullman

What it looks like

- ◆ The most common kind of query is a “select-from-where” statement.

SELECT desired attributes

FROM one or more tables

WHERE tuple satisfies condition

- ◆ SQL is case-insensitive.
Convention: keywords in upper case.

What vs how

- ◆ SQL is a very-high-level language.
 - ▶ Say “what” rather than “how.”
 - ▶ Avoid a lot of data-manipulation details needed in procedural languages like C++ or Java.
- ◆ The DBMS optimizes the query.
 - ▶ Allows programmer to focus on readability.

Our Running Example

Schema:

Beers(name, manf)

Bars(name, addr, license)

Drinkers(name, addr, phone)

Likes(drinker, beer)

Sells(bar, beer, price)

Frequents(drinker, bar)

Integrity constraints:

Likes[drinker] subset of Drinkers[name]

Likes[beer] subset of Beers[name]

Sells[bar] subset of Bars[name]

Sells[beer] subset of Beers[name]

Frequents[drinker] subset of Drinkers[name]

Frequents[bar] subset of Bars[name]

Example query

- ◆ Using **Beers(name, manf)**,
what beers are made by Molson?

```
SELECT name
```

```
FROM Beers
```

```
WHERE manf = 'Molson';
```

Meaning of a single-relation query

SELECT name (3) π

FROM Beers (1)

WHERE manf = 'Molson'; (2) σ

- ◆ Begin with the relation in the FROM clause.
- ◆ Apply the selection in the WHERE clause.
- ◆ Apply the projection indicated by the SELECT clause.

Result of the query

name
Bud
Bud Lite
Michelob
<i>etc ...</i>

Operational Semantics

name	manf
Bud	Molson

Tuple-variable t
loops over all
tuples

Check if
Molson

If so, include $t.name$
in the result

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

Naming the tuple variable yourself

You have the option of naming the tuple variable:

```
SELECT b.name  
FROM Beers b  
WHERE b.manf = 'Molson';
```

This query is identical to:

```
SELECT name  
FROM Beers  
WHERE manf = 'Molson';
```

Fancier SELECT

* In SELECT clauses

- ◆ When there is one relation in the FROM, a * in the SELECT clause stands for “all attributes of this relation.”
- ◆ **Example:** Using **Beers(name, manf):**

```
SELECT *  
FROM Beers  
WHERE manf = 'Molson';
```

Result of query:

name	manf
Bud	Molson
Bud Lite	Molson
Michelob	Molson
...	...

Now, the result has *all* of the attributes of Beers.

Renaming attributes

- ◆ Use "AS <new name>" to rename an attribute in the result.

- ◆ **Example:** Using **Beers(name, manf):**

```
SELECT name AS beer, manf
FROM Beers
WHERE manf = 'Molson'
```

Result of query:

beer	manf
Bud	Molson
Bud Lite	Molson
Michelob	Molson
.

Expressions in SELECT clauses

- ◆ Instead of a simple attribute name, you can use an expression in a SELECT clause.
- ◆ Operands: attributes, constants
Operators: arithmetic ops, string ops
- ◆ **Example:** Using `Sells(bar, beer, price)`:

```
SELECT bar, beer,  
       price*114 AS priceInYen  
FROM Sells;
```

Result of query

bar	beer	priceInYen
Joe's	Bud	285
Sue's	Miller	342
...

Expression that are just a constant

- ◆ Sometimes it makes sense to for the whole expression to be a constant (something that doesn't involve any attributes!).
- ◆ **Example:** Using `Likes(drinker, beer)`:

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

Result of query

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

Why would you ever do that??

- ◆ We often build *data warehouses* or create *federated databases* with data from many different data sources.
- ◆ Suppose each bar has its own database with a relation *Menu(beer, price)*.
- ◆ None of them would have a reason to include the bar's name.

- ◆ How could we build the relation `Sells(bar, beer, price)`?
- ◆ We need to query each bar and insert the name of the bar.
- ◆ For instance, at Joe's Bar we can issue the query:

```
SELECT 'Joe''s Bar', beer, price  
FROM Menu;
```

Fancier WHERE

Complex Conditions in a WHERE Clause

- ◆ We can build boolean expressions with operators that produce boolean results:
 - ◆ comparison operators:
=, <>, <, >, <=, >=.
 - ◆ and many other operators:
see section 6.1.2
- ◆ We can combine boolean expressions with:
 - ◆ Boolean operators AND, OR, NOT.

Example: Complex Condition

- ◆ Using `Sells(bar, beer, price)`,
find the price Joe's Bar charges for Bud:

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

Pattern operators

- ◆ A condition can compare a string to a pattern by:
 - ▶ <Attribute> LIKE <pattern> or
<Attribute> NOT LIKE <pattern>
- ◆ *Pattern* is a quoted string
 - ▶ % means "any string";
 - ▶ _ means "any single character".

Example: LIKE

- ◆ Using Drinkers(name, addr, phone)
find the drinkers with exchange 555:

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

NULL values

Missing Information

- ◆ We've assumed every tuple has a value for every attribute.
- ◆ But sometimes information is missing.
Two common scenarios:
 - ▶ *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.

How to represent that?

- ◆ One possibility: use a special value as a placeholder. E.g.,
 - ▶ If age unknown, use 0.
 - ▶ If StNum unknown, use 999999999.
- ◆ Pros and cons?
- ◆ Better solution: use a value not in any domain. We call this a **null value**.
- ◆ Tuples in SQL relations can have NULL as a value for one or more components.

Comparing NULL's to Values

- ◆ The logic of conditions in SQL is really 3-valued logic: TRUE, FALSE, UNKNOWN.
- ◆ Comparing any value (including NULL itself) with NULL yields UNKNOWN.
- ◆ A tuple is in a query result
iff
the WHERE clause is TRUE
(not FALSE or UNKNOWN).

Three-Valued Logic

- ◆ For TRUE result
 - ▶ OR: at least one operand must be TRUE
 - ▶ AND: both operands must be TRUE
 - ▶ NOT: operand must be FALSE
- ◆ For FALSE result
 - ▶ OR: both operands must be FALSE
 - ▶ AND: at least one operand must be FALSE
 - ▶ NOT: operand must be TRUE
- ◆ Otherwise, result is UNKNOWN
- ◆ **Example:** $\text{TRUE AND (FALSE OR NOT(UNKNOWN))} = \text{UNKNOWN}$

Three-Valued Logic

- ◆ A way to think about AND, OR, and NOT:
TRUE = 1, FALSE = 0, and UNKNOWN = $\frac{1}{2}$.
AND is min; OR is max, NOT(x) is $(1-x)$.

- ◆ Example:

TRUE AND (FALSE OR NOT(UNKNOWN)) =
 $1 \text{ and}(0 \text{ or not}(\frac{1}{2})) =$
 $\min(1, \max(0, (1 - \frac{1}{2}))) =$
 $\min(1, \max(0, \frac{1}{2})) =$
 $\min(1, \frac{1}{2}) = \frac{1}{2}.$

Surprising Example

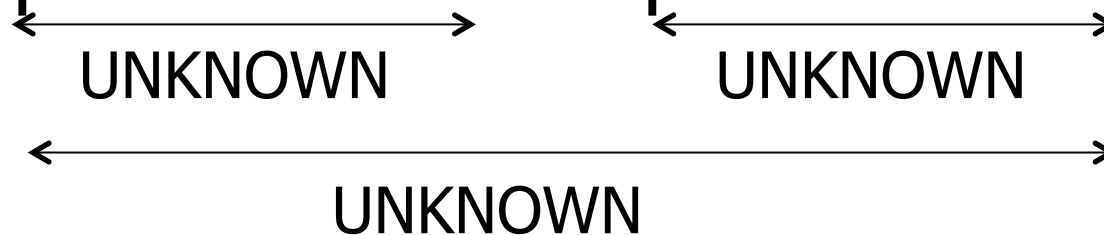
- ◆ From the following Sells relation:

bar	beer	price
Joe's Bar	Bud	NULL

SELECT bar

FROM Sells

WHERE price < 2.00 OR price >= 2.00;



2-valued laws don't always hold up in 3-valued logic!

- ◆ Some laws, like commutativity of AND, do hold in 3-valued logic.
- ◆ But others don't, e.g., the *law of the excluded middle* :
 - ▶ $p \text{ OR NOT } p = \text{TRUE}$.
 - ▶ When $p = \text{UNKNOWN}$,
 $p \text{ OR NOT } p = \max(1/2, (1 - 1/2)) = 1/2$
 - ▶ That's does not give us TRUE.

Fancier FROM

Multirelation Queries

- ◆ To combine data from more than one relation, list them all in the FROM clause.
- ◆ Like in relational algebra, distinguish attributes of the same name by "<relation>.<attribute>" .

Example: Joining Two Relations

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

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

Example: Joining Two Relations

- ◆ Alternatively can use explicit (named) tuple variables

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

- ◆ Here, it's just a convenience.
- ◆ But we know renaming is sometimes necessary.

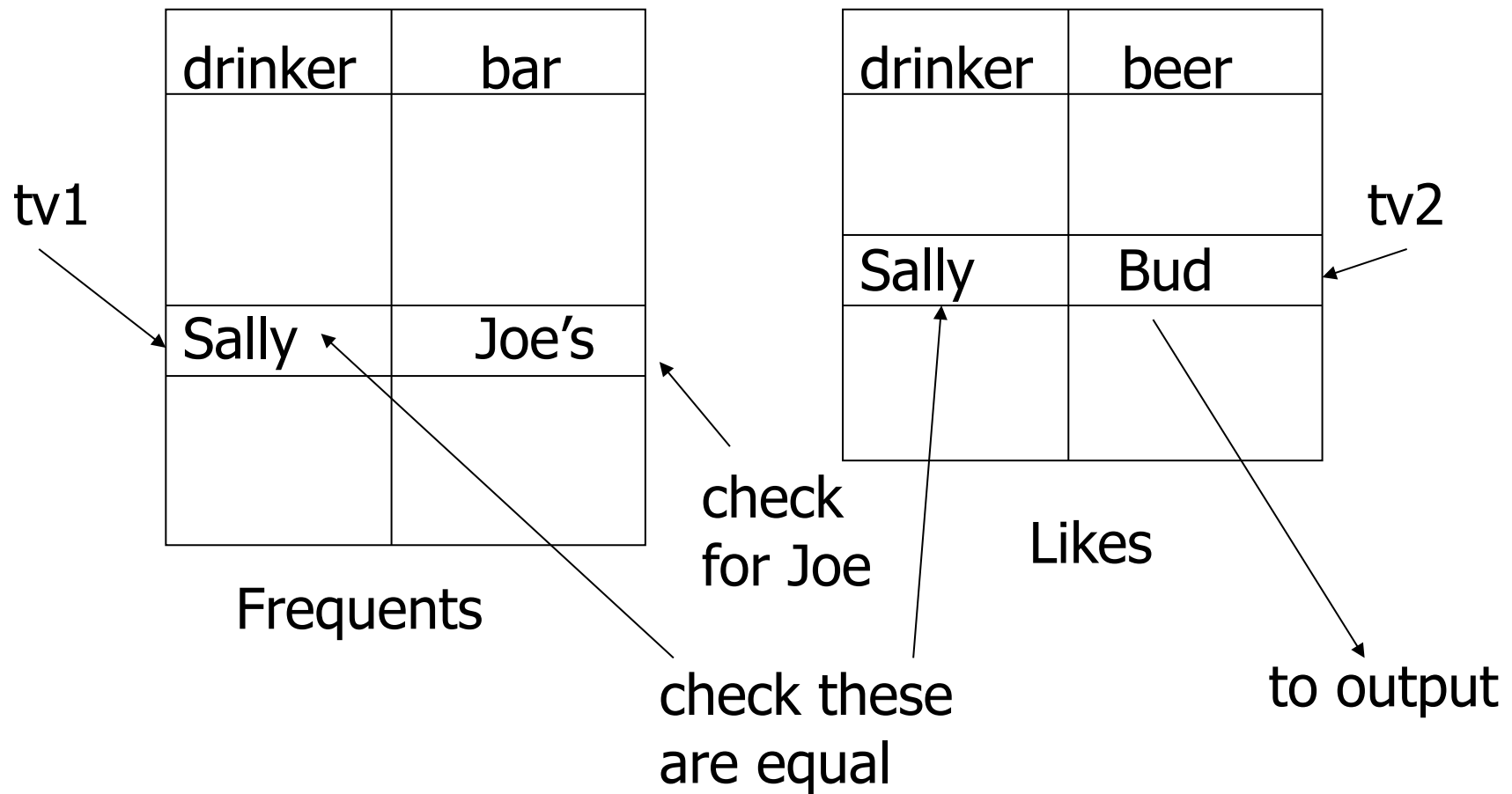
Formal Semantics

- ◆ Almost the same as for single-relation queries:
 - Start with the **Cartesian product** of all the relations in the FROM clause.
(We'll see other kinds of join later.)
 - Apply the selection condition from the WHERE clause.
 - Project onto the list of attributes and expressions in the SELECT clause.

Operational Semantics

- ◆ Imagine one tuple-variable for each relation in the FROM clause.
 - ▶ These tuple-variables visit each combination of tuples, one from each relation.
- ◆ If the tuple-variables are pointing to tuples that satisfy the WHERE clause, send these tuples to the SELECT clause.

Example



Self-joins

- ◆ Sometimes, a query needs to use two copies of the same relation.
- ◆ Distinguish copies by following the relation name by giving the tuple-variable a name in the FROM clause.

Example: Self-Join

- ◆ From **Beers(name, manf)**, find all pairs of beers by the same manufacturer.
 - ◆ Do not produce pairs like (Bud, Bud).
 - ◆ Do not produce the same pair twice like (Bud, Miller) and (Miller, Bud).

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

Subqueries

Subqueries

- ◆ A parenthesized SELECT-FROM-WHERE statement (*subquery*) can be used as a value in a number of places, including FROM and WHERE clauses.
- ◆ **Example:** in place of a relation in the FROM clause, we can use a subquery and then query its result.
 - ◆ Must use a tuple-variable to name tuples of the result.

Example: Subquery in FROM

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

SELECT beer

FROM Likes, (SELECT drinker
FROM Frequents
WHERE bar = 'Joe's Bar') JD

Drinkers who
frequent Joe's Bar

WHERE Likes.drinker = JD.drinker;

Subqueries often obscure queries

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

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

Simple join query

Subqueries That Return One Tuple

- ◆ If a subquery is guaranteed to produce one tuple, then the subquery can be used as a value.
 - ▶ Usually, the tuple has one component.
 - ▶ Remember SQL's 3-valued logic.

Example: Single-Tuple Subquery


- ◆ Using `Sells(bar, beer, price)`, find the bars that serve Miller for the same price Joe charges for Bud.
- ◆ Two queries would surely work:
 - Find the price Joe charges for Bud.
 - Find the bars that serve Miller at that price.

Query + Subquery Solution

```
SELECT bar  
FROM Sells  
WHERE beer = 'Miller' AND
```

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

The price at
which Joe
sells Bud



What if price of Bud is NULL?

Query + Subquery Solution


SELECT bar

FROM Sells

WHERE beer = 'Miller' AND

price = (SELECT price
FROM Sells
WHERE beer = 'Bud');

What if subquery
returns multiple
values?



More operators for use in WHERE

The Operator ANY

- ◆ $x = \text{ANY}(\langle \text{subquery} \rangle)$ is a boolean condition that is true iff x equals at least one tuple in the subquery result.
 - ◆ = could be any comparison operator.
- ◆ **Example:** $x \geq \text{ANY}(\langle \text{subquery} \rangle)$ means x is not the uniquely smallest tuple produced by the subquery.
 - ◆ Note tuples must have one component only.

The Operator ALL

- ◆ $x <> \text{ALL}(<\text{subquery}>)$ is true iff for every tuple t in the relation, x is not equal to t .
 - ◆ That is, x is not in the subquery result.
- ◆ $<>$ can be any comparison operator.
- ◆ **Example:** $x \geq \text{ALL}(<\text{subquery}>)$ means there is no tuple larger than x in the subquery result.

Example: ALL

- ◆ From **Sells(bar, beer, price)**, find the beer(s) sold for the highest price.

SELECT beer

FROM Sells

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

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

The IN Operator

- ◆ `<tuple> IN (<subquery>)` is true if and only if the tuple is a member of the relation produced by the subquery.
 - ▶ Opposite: `<tuple> NOT IN (<subquery>)`.
- ◆ IN-expressions can appear in WHERE clauses.

Example: IN

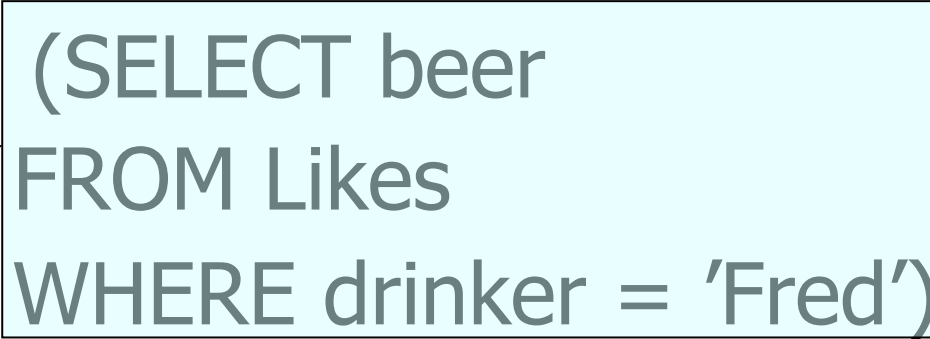
- ◆ Using **Beers(name, manf)** and **Likes(drinker, beer)**, find the name and manufacturer of each beer that Fred likes.

SELECT *

FROM Beers

WHERE name IN

The set of
beers Fred
likes



(SELECT beer
FROM Likes
WHERE drinker = 'Fred');

IN vs. Join

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

```
SELECT R.a  
FROM R  
WHERE b IN (SELECT b FROM S);
```

IN is a Predicate About R's Tuples

SELECT a

FROM R

WHERE b IN

(SELECT b FROM S);

Two 2's

One loop, over
the tuples of R

a	b
1	2
3	4

R

b	c
2	5
2	6

S

(1,2) satisfies
the condition;
1 is output once.

This Query Pairs Tuples from R, S

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

Double loop, over
the tuples of R and S

a	b
1	2
3	4

R

b	c
2	5
2	6

S

(1,2) with (2,5)
and (1,2) with
(2,6) both satisfy
the condition;
1 is output twice.

Note: result of SQL query may be a BAG!

The Exists Operator

- ◆ EXISTS(<subquery>) is true if and only if the subquery result is not empty.
- ◆ **Example:** From **Beers(name, manf)** , find those beers that are the unique (only) beer made by their manufacturer.

Example: EXISTS

```
SELECT name  
FROM Beers b1  
WHERE NOT EXISTS (
```

Notice scope rule: manf refers to closest nested FROM with a relation having that attribute. (Some DBMS consider this ambiguous.)

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

```
SELECT *  
FROM Beers  
WHERE manf = b1.manf AND  
      name <> b1.name);
```

Notice the SQL "not equals" operator

Set operations

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>)

Example: Intersection

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

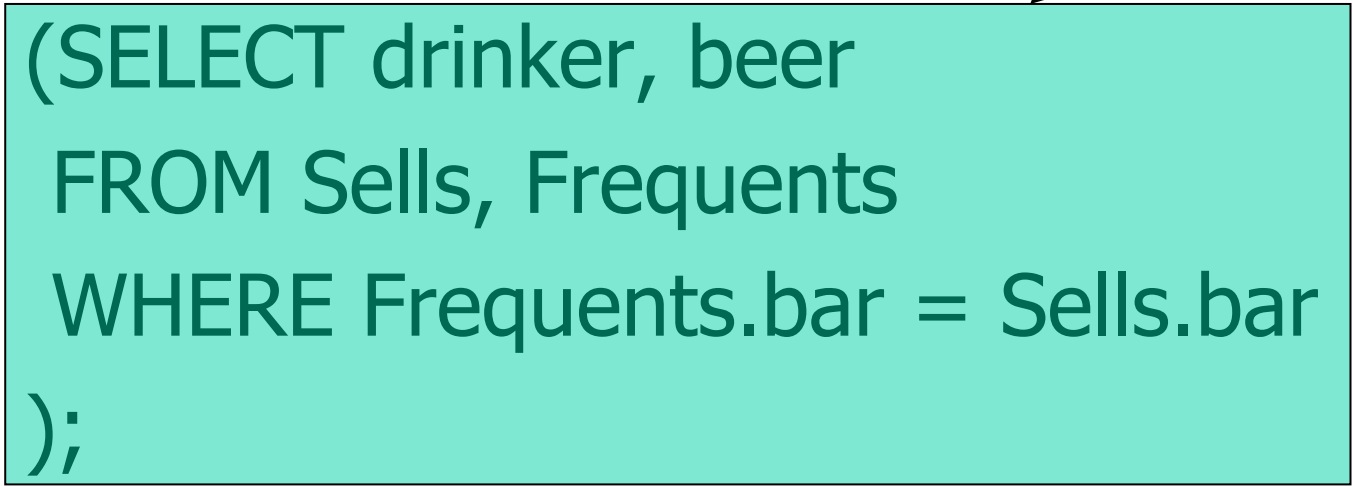
Notice trick:
subquery is
really a stored
table.

Solution



```
(SELECT * FROM Likes)
```

INTERSECT



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

The drinker frequents
a bar that sells the
beer.

Bag Semantics

- ◆ 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.
- ◆ For intersection or difference, it is most efficient to sort the relations first.
 - ▶ At that point you may as well eliminate the duplicates anyway.

Controlling Duplicate Elimination

- ◆ Force the result to be a set by
SELECT DISTINCT . . .
 - ◆ Be careful: SQL2003 allows a null to be retained
- ◆ 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.

Example: ALL

- ◆ Using relations **Frequents(drinker, bar)** and **Likes(drinker, beer)**:

```
(SELECT drinker FROM Frequents)
  EXCEPT ALL
  (SELECT drinker FROM Likes);
```

- ◆ Lists drinkers who frequent more bars than they like beers, and does so as many times as the difference of those counts.

Join Expressions

- ◆ SQL provides several versions of (bag) joins.
- ◆ These expressions can be stand-alone queries or used in place of relations in a FROM clause.

Join Expressions

- ◆ SQL provides several versions of (bag) joins.
- ◆ These expressions can be stand-alone queries or used in place of relations in a FROM clause.

Other kinds of join

Products and Natural Joins

- ◆ Natural join:

`R NATURAL JOIN S;`

- ◆ Product:

`R CROSS JOIN S;`

- ◆ Example:

`Likes NATURAL JOIN Sells;`

- ◆ Relations can be parenthesized subqueries, as well.

Theta Join

- ◆ `R JOIN S ON <condition>`
- ◆ **Example:** using `Drinkers(name, addr)` and `Frequents(drinker, bar)`:

```
Drinkers JOIN Frequents ON  
    name = drinker;
```

gives us all (d, a, d, b) quadruples such that drinker d lives at address a and frequents bar b .

Dangling tuples and outer joins

Dangling tuples

- ◆ With natural join, some tuples may have nothing to match.
- ◆ They are therefore left out of the results.
- ◆ We say that they are *dangling*.

Example: dangling tuples

R =

A	B
1	2
4	5

S =

B	C
2	3
6	7

(1,2) joins with (2,3), but the other two tuples are dangling.

R NATURAL JOIN S =

A	B	C
1	2	3

Outerjoin

Outerjoin preserves dangling tuples by padding them with NULLs in the other relation.

R =

A	B
1	2
4	5

S =

B	C
2	3
6	7

R OUTERJOIN S =

A	B	C
1	2	3
4	5	NULL
NULL	6	7

Left outerjoin

Preserves dangling tuples on the LHS by padding them with NULL on the RHS.

R =

A	B
1	2
4	5

S =

B	C
2	3
6	7

R LEFT OUTERJOIN S =

A	B	C
1	2	3
4	5	NULL

Right outerjoin

Preserves dangling tuples on the RHS by padding them with NULL on the LHS.

R =

A	B
1	2
4	5

S =

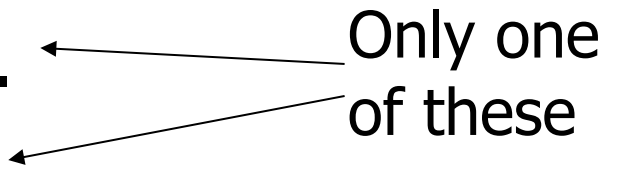
B	C
2	3
6	7

R RIGHT OUTERJOIN S =

A	B	C
1	2	3
NULL	6	7

Variations

◆ Options for an “R OUTER JOIN S” expression:

1. NATURAL in front of OUTER.
 2. ON <condition> after JOIN.
 3. LEFT, RIGHT, or FULL before OUTER.
- ◆ FULL is the default.
- Only one of these
- 

Aggregation

Computing on a column

- ◆ We often want to compute something across the values in a column.
- ◆ SUM, AVG, COUNT, MIN, and MAX can be applied to a column in a SELECT clause.
- ◆ Also, COUNT(*) counts the number of tuples.
- ◆ We call this aggregation.

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

- ◆ To stop duplicates from contributing to the aggregation, use `DISTINCT` inside.
- ◆ **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, and can never be the minimum or maximum of a column (unless every value is NULL).
- ◆ If there are no non-NULL values in a column, then the result of the aggregation is NULL.
 - ◆ **Exception:** COUNT of an empty set is 0.

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 tuples in a relation

Grouping

- ◆ If we follow a SELECT-FROM-WHERE expression with GROUP BY <attributes>
 - ▶ The tuples are grouped according to the values of 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;
```

beer	AVG(price)
Bud	2.33
...	...

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;
```

Compute all drinker-bar-price triples for Bud.

Then group them 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.

HAVING Clauses

- ◆ Sometimes we don't want all the groups of tuples.
- ◆ If we follow a GROUP BY clause with HAVING <condition>
 - ◆ Only groups satisfying the condition are kept.

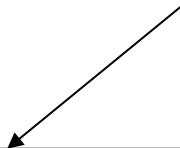
Example: HAVING

- ◆ 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

```
SELECT beer, AVG(price)
FROM Sells
GROUP BY beer
```


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



```
HAVING COUNT(bar) >= 3 OR
```

```
beer IN (SELECT name
FROM Beers
WHERE manf = 'Pete"s');
```

Beers manufactured by Pete's.



Requirements on HAVING Conditions

- ◆ Anything goes in a subquery.
- ◆ Outside subqueries, they may refer to attributes only if they are either:
 1. A grouping attribute, or
 2. Aggregated(same condition as for SELECT clauses with aggregation).

Ordering your results

ORDER BY

- ◆ To put the tuples in order, add this as the final clause:
ORDER BY <attribute list> [DESC]
- ◆ The default is ascending order; DESC overrides it.
- ◆ The attribute list can include expressions:
ORDER BY sales+rentals
- ◆ The ordering is the last thing done before the SELECT (so all attributes are still available).

Modifying a Database

Database Modifications

- ◆ Queries return a relation.
- ◆ A *modification* command does not; it changes the database in some way.
- ◆ 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> );
```

- ◆ Can insert several lists of values at once.

- ◆ Example:

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

Specifying Attributes in INSERT

- ◆ In the INSERT statement, we can name the attributes to be inserted.
- ◆ 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 without having to give that value every time.

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');
```

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');
```

Aside: Specifying Default Values

- ◆ In a CREATE TABLE statement, we can follow an attribute by
DEFAULT <value>
- ◆ When an inserted tuple has no value for that attribute, the default will be used.

Example: Default Values

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

Example: Default Values

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

Resulting tuple:

name	address	phone
Sally	123 Sesame St	NULL

Inserting Tuples from a Query

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

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


Example: Insert 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 Sally,
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 = 'Sally' AND
d2.drinker <> 'Sally' 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 = 'Sally' AND
      beer = 'Bud';
```

Example: Delete all Tuples

- ◆ If there is no WHERE clause, *all* tuples are deleted.
- ◆ E.g., this makes the relation Likes empty:

```
DELETE FROM Likes;
```

Example: Delete Some 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 --- (1)

- ◆ 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 --- (2)

- ◆ **Answer:** 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.
 2. Delete the marked tuples.

Updates

- ◆ To change the value of certain attributes in certain tuples:

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;
```

Views

The idea

- ◆ A *view* is a relation defined in terms of stored tables (called *base tables*) and other views.
- ◆ Two kinds:
 1. *Virtual* = not stored in the database; just a query for constructing the relation.
 2. *Materialized* = actually constructed and stored. Expensive to maintain!

Declaring Views

- ◆ Declare by:

```
CREATE [MATERIALIZED] VIEW  
    <name> AS <query>;
```

- ◆ Default is virtual.

Example: View Definition

- ◆ **CanDrink(drinker, beer)** is a view “containing” the drinker-beer pairs such that the drinker frequents at least one bar that serves the beer:

```
CREATE VIEW CanDrink AS
  SELECT drinker, beer
  FROM Frequents, Sells
  WHERE Frequents.bar = Sells.bar;
```

Example: Accessing a View

- ◆ Query a view as if it were a base table.

```
SELECT beer FROM CanDrink  
WHERE drinker = 'Sally';
```

- ◆ Updates:

- ◆ Not possible with virtual views – they don't "exist"!
- ◆ DBMS may allow them, under limited circumstances, with materialized views.

SQL: Data Definition Language

Database Schemas in SQL

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

Declaring a Relation

- ◆ Simplest form is:

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

- ◆ To delete a relation:

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

Elements of Table Declarations

- ◆ The most basic element is an attribute and its type.
- ◆ The most common types are:
 - ◆ INT or INTEGER (synonyms).
 - ◆ REAL or FLOAT (synonyms).
 - ◆ CHAR(n) = fixed-length string of n characters.
 - ◆ VARCHAR(n) = variable-length string of up to n characters.

Example: Create Table

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

Types

For full details, see the `psql` documentation, chapter 8.

SQL Values

- ◆ Integers and reals are represented as you would expect.
- ◆ Strings are too, except they require single quotes.
 - ▶ To put a single quote in a string, use two single quote, e.g., 'Joe''s Bar'.
- ◆ Attributes of any type can be NULL
 - ▶ Unless attribute has a NOT NULL constraint
 - ▶ E.g., price REAL not null,

Dates and Times

- ◆ DATE and TIME are types in SQL.

- ◆ The form of a date value is:

DATE 'yyyy-mm-dd'

- ◆ **Example:** DATE '2007-09-30'
for Sept. 30, 2007.

Times as Values

- ◆ The form of a time value is:
TIME 'hh:mm:ss'
- ◆ An optional decimal point and fractions of a second can follow.
- ◆ **Example:** TIME '15:30:02.5'
for two and a half seconds after
3:30 p.m.

Keys

Declaring Keys

- ◆ An attribute or list of attributes may be declared PRIMARY KEY or UNIQUE.
- ◆ Both say that no two tuples of the relation may agree in all the attribute(s) on the list.
- ◆ Declaring a primary key gives the DBMS a hint that it should build an index on this attribute(s).
- ◆ There are several other differences ...

PRIMARY KEY vs. UNIQUE

- ◆ A relation can have only one PRIMARY KEY.
 - ◆ But can have several UNIQUE attributes.
- ◆ No attribute of a PRIMARY KEY can be NULL in any tuple.
 - ◆ But attributes declared UNIQUE may be NULL.
 - ◆ There may even be several tuples with NULL.

Our Running Example

Beers(name, manf)

Bars(name, addr, license)

Drinkers(name, addr, phone)

Likes(drinker, beer)

Sells(bar, beer, price)

Frequents(drinker, bar)

Remember that underline indicates *key*

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

- ◆ If a key involves several attributes, you can't declare it where you declare a single attribute.
- ◆ Instead, make it a separate element in the CREATE TABLE statement.
- ◆ (You can declare single-attribute keys this way too).

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)  
);
```


Other kinds of constraints

Kinds of Constraints

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

Expressing Foreign Keys

- ◆ Use keyword REFERENCES, either:
 1. After an attribute (for one-attribute keys).
 2. As a separate element of the schema:
FOREIGN KEY (<list of attributes>
REFERENCES <relation> (<attributes>)
- ◆ Referenced attributes must be declared PRIMARY KEY or UNIQUE in their home table.

Example: Declaring a foreign key with an attribute

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

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

Example: Declaring a foreign key as a schema element

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

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

Enforcing Foreign-Key Constraints

- ◆ How/when can the DBMS ensure that:
 1. the referenced attributes are PRIMARY KEY or UNIQUE?
 2. the values actually exist?

When could a foreign key constraint *become* violated?

- ◆ Suppose there is a foreign-key constraint from relation R to relation S .
- ◆ Two things could create a violation:
 1. An insert or update to R could introduce values not found in S .
 2. A deletion or update to S could cause some tuples of R to “dangle.”
- ◆ You define what the DBMS should do.

Reaction Policies

Possible actions to take

- ◆ **Example:** suppose $R = \text{Sells}$, $S = \text{Beers}$.
- ◆ An insert or update to **Sells** that introduces a beer that's not in **Beers** must be rejected.
- ◆ A deletion or update to **Beers** that removes a beer found in **Sells** can be handled in three ways ...

Three options

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

Example: Cascade

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

Example: Set NULL

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

Declaring a Policy

- ◆ The default policy is to reject.
- ◆ But when we declare a foreign key, we may choose SET NULL or CASCADE (independently for deletions and updates).
- ◆ Follow the foreign-key declaration by:
ON [UPDATE, DELETE][SET NULL, CASCADE]
- ◆ Can specify this for both UPDATE and DELETE, or just one (or neither).

Example: Declaring a Policy

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

Check constraints

Attribute-Based Checks

- ◆ Constraints on the value of a single attribute.
- ◆ Add CHECK(<condition>) to the declaration for the attribute.
- ◆ The condition may refer directly to that attribute.
- ◆ But any other relation or attribute name must be in a subquery.

Example: Attribute-Based Check

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

Naming your constraint

- ◆ If you name your constraint, you will get more helpful error messages.
- ◆ Add
 CONSTRAINT <name>
before the
 CHECK (<condition>)

Example: named constraint

```
CREATE TABLE Sells (  
    bar    CHAR(20) ,  
    beer   CHAR(20) ,  
    price  REAL  
        CONSTRAINT cheap  
        CHECK ( price <= 5.00 )  
);
```

How NULLS affect a Check

- ◆ A CHECK only fails if the expression evaluates to false.
- ◆ It is not picky like a WHERE condition.

Value of condition	CHECK outcome	WHERE outcome
True	Pass	Pass
False	Fail	Fail
Unknown	Pass	Fail

When Checks are done

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

Tuple-Based Checks

- ◆ CHECK (<condition>) may be added as a relation-schema element instead of added to the declaration of a particular attribute.
- ◆ Then the condition may refer to *any* attribute of the relation.
 - ◆ But other attribute or relation name must be in a subquery.
- ◆ Checked only on insert or update.

Example: Tuple-Based Check

- ◆ Only Joe's Bar can sell beer for more than \$5:

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

Assertions

- ◆ These are not defined inside any relation.
- ◆ Defined at the same level as a table, by:

```
CREATE ASSERTION <name>  
CHECK (<condition>);
```

- ◆ Condition may refer to *any* relation or attribute in the database schema.

Example: Assertion

- ◆ In `Sells(bar, beer, price)`, no bar may charge an average of more than \$5.

```
CREATE ASSERTION NoRipoffBars CHECK (  
  NOT EXISTS (  
    SELECT bar FROM Sells  
    GROUP BY bar  
    HAVING 5.00 < AVG(price)  
  ));
```

Bars with an average price above \$5

Example: Assertion

- ◆ In Drinkers(name, addr, phone) and Bars(name, addr, license), there cannot be more bars than drinkers.

```
CREATE ASSERTION FewBar CHECK (  
    (SELECT COUNT(*) FROM Bars) <=  
    (SELECT COUNT(*) FROM Drinkers)  
);
```

When Assertions are Checked

- ◆ In principle, we must check every assertion after every modification to any relation of the database.
- ◆ A clever system can observe that only certain changes could cause a given assertion to be violated.
 - ◆ **Example:** No change to Beers can affect FewBar. Neither can an insertion to Drinkers.

Controlling When

- ◆ Attribute- and tuple-based checks are checked at known times, but are not powerful.
- ◆ Assertions are powerful, but the DBMS often can't tell when they need to be checked.
- ◆ Triggers let you control when to check an assertion.

Complex and costly

- ◆ Assertions are complex for a DBMS to implement well.
- ◆ They can be very expensive to run.
- ◆ Not all DBMSs support them.
- ◆ PostgreSQL does not support them.