

SQL, Part 2

Relations as Bags
Grouping and Aggregation
Database Modification
Defining Schemas and Views

1

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)

2

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

3

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.

4

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.

5

Motivation: Efficiency

- ◆ When doing projection in relational algebra, 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.

6

Controlling Duplicate Elimination

- ◆ Force the result to be a set by
`SELECT DISTINCT . . .`
- ◆ Force the result to be a bag (i.e., don't eliminate duplicates) by `ALL`, as in
`. . . UNION ALL . . .`

7

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.

8

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.

9

Join Expressions

- ◆ SQL provides a number of expression forms that act like varieties of join in relational algebra.
 - ▶ But using bag semantics, not set semantics.
- ◆ These expressions can be stand-alone queries or used in place of relations in a `FROM` clause.

10

Products and Natural Joins

- ◆ Natural join is obtained by:
`R NATURAL JOIN S;`
- ◆ Product is obtained by:
`R CROSS JOIN S;`
- ◆ Example:
`Likes NATURAL JOIN Serves;`
- ◆ Relations can be parenthesized subexpressions, as well.

11

Theta Join

- ◆ `R JOIN S ON <condition>` is a theta-join, using `<condition>` for selection.
- ◆ 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 .

12

Outerjoins

- ◆ R OUTER JOIN S is the core of an outerjoin expression. It is modified by:
 1. Optional NATURAL in front of OUTER.
 2. Optional ON <condition> after JOIN.
 3. Optional LEFT, RIGHT, or FULL before OUTER.
 - ◆ LEFT = pad dangling tuples of R only.
 - ◆ RIGHT = pad dangling tuples of S only.
 - ◆ FULL = pad both; this choice is the default.

13

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.

14

Example: Aggregation

- ◆ From Sells(bar, beer, price), find the average price of Bud:

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

15

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';
```

16

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.
- ◆ But if there are **no** non-NULL values in a column, then the result of the aggregation is NULL.

17

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
non-NULL price.

18

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.**

19

Example: Grouping

- ◆ From Sells(bar, beer, price), find the average price for each beer:

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

20

Example: Grouping

- ◆ From Sells(bar, beer, price) and Frequent(drinker, bar), find for each drinker the average price of Bud at the bars they frequent:

```
SELECT drinker, AVG(price)
FROM Frequent, Sells
WHERE beer = 'Bud' AND
      Frequent.bar = Sells.bar
GROUP BY drinker;
```

Compute drinker-bar-price of Bud tuples first, then group by drinker.

21

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.

22

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? Note **bar** is neither aggregated nor on the GROUP BY list.

23

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.

24

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 makes sense within a group; i.e., it is either:
 1. A grouping attribute, or
 2. Aggregated.

25

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.

26

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.

27

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.

28

Insertion

- ◆ To insert a single tuple:

```
INSERT INTO <relation>  
VALUES ( <list of values> );
```

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

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

29

Specifying Attributes in INSERT

- ◆ We may add to the relation name a list of attributes.
- ◆ There are two reasons to do so:
 1. We **forget, or don't wish to use**, 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.

30

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');
```

31

Inserting Many Tuples

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

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

32

Example: Insert a Subquery

- ◆ Using `Frequents(drinker, bar)`, enter into the new relation

PotentialBuddies(name)

all of Sally's "potential buddies," i.e., those drinkers who frequent at least one bar that Sally also frequents.

33

Solution

The other drinker

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

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

34

Deletion

- ◆ To delete tuples satisfying a condition from some relation:

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

We don't specify any attributes of relation, since the entire tuple is deleted.

35

Example: Deletion

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

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

36

Example: Delete all Tuples

- ◆ Make the relation Likes empty:

```
DELETE FROM Likes;
```

- ◆ Note no WHERE clause needed.

37

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.

38

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?

39

Semantics of Deletion -- 2

- ◆ 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.

40

Updates

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

```
UPDATE <relation>
```

```
SET <list of attribute assignments>
```

```
WHERE <condition on tuples>;
```

41

Example: Update

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

```
UPDATE Drinkers
```

```
SET phone = '555-1212'
```

```
WHERE name = 'Fred';
```

42

Example: Update Several Tuples

- ◆ Make \$4 the maximum price for beer:

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

43

Defining a Database Schema

44

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, indexes, and triggers, which we’ll introduce later.

45

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>;
```

46

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.

47

Example: Create Table

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

48

Dates and Times

- ◆ DATE and TIME are types in SQL.
- ◆ The form of a date value is:
DATE 'yyyy-mm-dd'
 - ▶ Example: DATE '2002-09-30' for Sept. 30, 2002.

49

Times as Values

- ◆ The form of a time value is:
TIME 'hh:mm:ss'
with an optional decimal point and fractions of a second following.
 - ▶ Example: TIME '15:30:02.5' = two and a half seconds after 3:30PM.

50

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.

51

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)  
);
```

52

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.

53

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)  
);
```

54

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.

55

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.

56

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>.

57

Example: Default Values

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

58

Effect of Defaults -- 1

- ◆ 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');
```

59

Effect of Defaults -- 2

- ◆ 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.

60

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';
```

61

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;
```

62

Defining Views

63

Views

- ◆ A view is a “virtual table,” a relation that is defined in terms of the contents of other tables and views.
- ◆ Declare by:
`CREATE VIEW <name> AS <query>;`
- ◆ In contrast, a relation whose value is really stored in the database is called a *base table*.

64

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;
```

65

Example: Accessing a View

- ◆ You may query a view as if it were a base table.
 - ▶ There is a limited ability to modify views if the modification makes sense as a modification of the underlying base table.
- ◆ Example:

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

66

What Happens When a View Is Used?

- ◆ The DBMS starts by interpreting the query as if the view were a base table.
 - ▶ Typical DBMS turns the query into something like relational algebra.
- ◆ The queries defining any views used by the query are also replaced by their algebraic equivalents, and “spliced into” the expression tree for the query.

67

Example: View Expansion



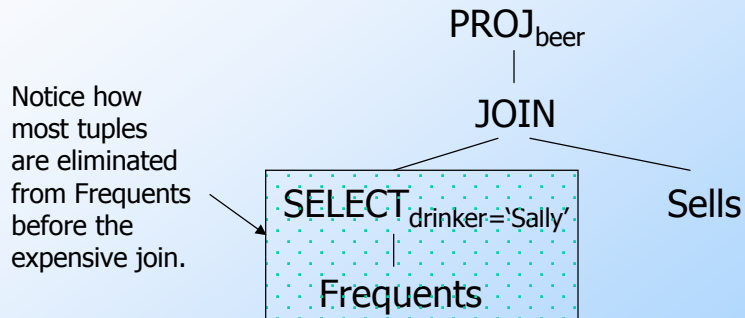
68

DMBS Optimization

- ◆ It is interesting to observe that the typical DBMS will then “optimize” the query by transforming the algebraic expression to one that can be executed faster.
- ◆ Key optimizations:
 1. Push selections down the tree.
 2. Eliminate unnecessary projections.

69

Example: Optimization



70