The Database Language SQL

Outline

1. Introduction
2. Single-Relation Queries
3. NULL values
4. Multirelation Queries
5. SQL92 Joins

Why SQL?

- SQL is a very-high-level language.
  - Say “what to do” rather than “how to do it.”
  - Avoid a lot of data-manipulation details needed in procedural languages like C++ or Java.
- Database management system figures out “best” way to execute query.
  - Called “query optimization.”

Select-From-Where Statements

SELECT desired attributes

Projection

FROM one or more tables

WHERE condition about tuples of the tables

Selection

Our Running Example

- All our SQL statements will be based on the following database schema.
  - Underline indicates primary key attributes
  - # indicates foreign 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>)

Outline

1. Introduction
2. Single-Relation Queries
3. NULL values
4. Multirelation Queries
5. SQL92 Joins

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, and tuples with the name of each beer by Anheuser-Busch, such as Bud.

Meaning of Single-Relation Query

- 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 --- General

- Think of a *tuple variable* visiting 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.

Operational Semantics

name	manf
Bud	Anheuser-Busch

Include t.name in the result, if so

Check if Anheuser-Busch

Tuple-variable *t* loops over all tuples

* 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: Using 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: 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
…	…	…

Example: Constants as Expressions

- 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
…	…

Example: Information Integration

- We often build "data warehouses" from the data at many "sources."
- Suppose each bar has its own relation Menu(beer, price) .
- To contribute to Sells(bar, beer, price) we need to query each bar and insert the name of the bar.

Information Integration --- (2)

- For instance, at Joe's Bar we can issue the query:

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

Complex Conditions in WHERE Clause

- Comparisons: =, <>, <, >, <=, >=
  - Note: compare with == and != in programming languages like C++, Java, etc.
- And many other operators that produce boolean-valued results: LIKE, IN, etc.
- Combined 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';
```

Patterns

- A condition can compare a string to a pattern by:
  - <Attribute> LIKE <pattern> or <Attribute> NOT LIKE <pattern>
- *Pattern* is a quoted string with
  - % = "any string"
  - _ = "any character."

Example: LIKE

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

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

Outline

1. Introduction
2. Single-Relation Queries
3. NULL values
4. Multirelation Queries
5. SQL92 Joins

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

```
SELECT bar, beer
FROM Sells
WHERE price < 2.0
```

- What is the value of the boolean expression "price < 2.0" supposed to be when price is NULL?
  - recall NULL means "some value we don’t know"

Comparing NULL's to Values (2)

- The boolean expression should evaluate to TRUE if the actual price were 1.5.
- But it should evaluate to FALSE if the actual price were 3.0.
- Since we can't decide, the boolean expression evaluates to UNKNOWN.

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:
  - value = null -> UNKNOWN
  - null = null -> UNKNOWN
- A tuple is in a query answer iff the WHERE clause is TRUE (not FALSE or UNKNOWN).

Three-Valued VS Two-Valued Logic

- Most of the 2-valued logic laws still hold in 3-valued logic:
  - AND and OR are commutative
  - AND and OR are distributive
  - AND’s absorber: FALSE
  - OR’s absorber: TRUE
  - …

Three-Valued VS Two-Valued Logic (2)

- But some laws do not hold anymore, e.g. the law of the excluded middle:
  - p OR NOT p = TRUE
- When p is UNKNOWN, the left side evaluates to UNKNOWN.

Surprising Example

- From the following Sells relation:

bar	beer	price
Joe's Bar	Bud	NULL

SELECT bar

FROM Sells

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

UNKNOWN

Three-Valued Truth Tables

OR's and AND's truth table with UNKNOWN:

A	**B**	**A OR B**	**A AND B**
TRUE	UNKNOWN	TRUE	UNKNOWN
FALSE	UNKNOWN	UNKNOWN	FALSE
UNKNOWN	UNKNWON	UNKNWON	UNKNWON

NOT's truth table:

A	**NOT A**
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

Comparing to NULL

- "value IS [NOT] NULL" checks whether *value* is [not] NULL. IS is guaranteed to always return either TRUE or FALSE.
- Function COALESCE(p1, p2, ..., pn) returns its first non-null parameter. It is useful in expression involving possibly NULL operands.

Outline

1. Introduction
2. Single-Relation Queries
3. NULL values
4. Multirelation Queries
5. SQL92 Joins

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: 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;
```

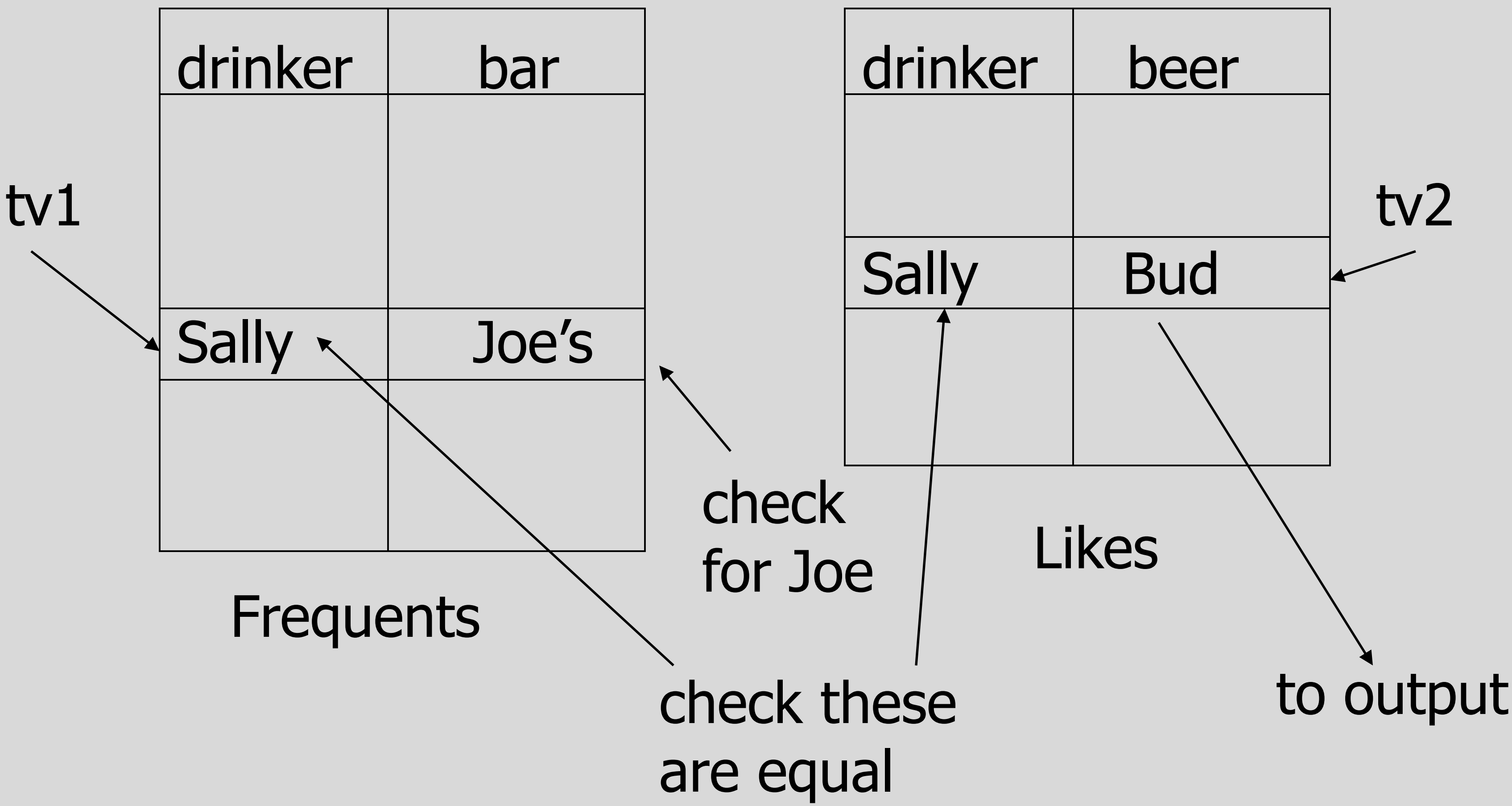
Formal Semantics

- 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


drinker
bar
Sally
Joe's
Frequents
tv1
drinker
beer
Sally
Bud
Likes
tv2
check
for Joe
check these
are equal
to output

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: Self-Join

- 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;
```

Outline

1. Introduction
2. Single-Relation Queries
3. NULL values
4. Multirelation Queries
5. SQL92 Joins

Join Expressions

- Up to 1992, the only way to express a join was as follows:

```
SELECT drinker
FROM Frequents, Bars
WHERE bar = name AND address = 'Maple St.'
```

cartesian product…

… followed by
a selection

Join Expressions (2)

- Starting from 1992, SQL provides several versions of joins.
- These expressions can be standalone queries or used in place of relations in a FROM clause. Example:
  - stand-alone:

    R NATURAL JOIN *S* ;
  - in FROM clause:

    SELECT * FROM *R* NATURAL JOIN *S* ;

Join Expressions (3)

- The goal is twofold:
  - Segregate join conditions and selection conditions for *clarity* (theta joins).
  - *Shorten* queries by making the join condition implicit (natural join)

Theta Joins and Natural Joins

- Theta join:

  R JOIN *S* ON <join condition>;

  The join condition is explicitly stated.

- Natural join:

  R NATURAL JOIN *S* ;

  The join condition is implicit: equate attributes with the same name in R and S.

Example: Theta Joins

- Query page 46 can be rewritten as follows:

```
SELECT drinker
FROM Frequents JOIN Bars ON bar = name
WHERE address = 'Maple St.'
```

Example: Natural Joins

- Assume we change the schema of Frequents from Frequents(drinker, **bar**) to Frequents(drinker, **name**) to make the natural join possible.
- Then query page 46 can be rewritten as follows:

SELECT drinker

FROM Frequents NATURAL JOIN Bars

WHERE address = 'Maple St.'

Favoring Natural Joins

- To make the use of natural joins possible, database designers must:
  - give the same name to attributes that are subject to join (in all the tables)
  - give *distinct* names to attributes that are *not* subject to join (in all the tables)
- If not, users won't be able to use natural joins: they will have to resort to theta joins.

Inner VS Outer Joins

- By default, joins are *inner* joins: dangling tuples are not output.
- The INNER keyword does exist, but it is seldom used since it is the default. (Just as ASC in ORDER BY is seldom used.)
- To include dangling tuples in the result, use OUTER join.

Inner VS Outer Joins (2)

- When using OUTER, you can further specify one the following:
  - LEFT: only dangling tuples from the left table (i.e. R) are output
  - RIGHT: only dangling tuples from the right table (i.e. S) are output
  - FULL: all dangling tuples, whether from R or S, are output. This is the default.

Joins: Summary

	Theta Join	**Natural join**
Inner Join (default)	R INNER JOIN S ON <cond.>	R INNER NATURAL JOIN S
Outer Join	R [opt.] OUTER JOIN S ON <cond.>	R NATURAL [opt.] OUTER JOIN S

[opt.] = [LEFT | RIGHT | FULL]

Joins: Summary (2)

- Theta VS natural relates to whether the join condition is explicit (theta join) or implicit (natural join).
  - There is no default.
- Inner VS outer relates to whether dangling tuples must be excluded (inner) or output (outer).
  - Default is inner.

SQL92 Products

- SQL92 also provides a new expression for products:

  R CROSS JOIN *S* ;

SELECT * FROM *R* CROSS JOIN *S* ;

is equivalent to the old form:

SELECT * FROM *R*, *S* ;

SQL – Continued

Outline

1. Subqueries
2. Set Operators
3. Multiset Semantics
4. Aggregation and Grouping

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.

Drinkers who frequent Joe's Bar

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

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.
  - A run-time error occurs if there is no tuple or more than one tuple.

Example: Single-Tuple Subquery

- Using Sells(<u>bar</u>, <u>beer</u>, price), find the bars that serve Miller for the same price Joe charges for Bud.
- Two queries would surely work:
  1. Find the price Joe charges for Bud.
  2. 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

Subqueries That Return Any Number of Tuples

- Subqueries that return any number of tuples can be used in the WHERE clause with the following boolean operators:
  - IN
  - EXISTS
  - *operator* ANY
  - *operator* ALL

where *operator* is any scalar-comparison operator

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

The set of beers Fred likes

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 beer by their manufacturer.

Example: EXISTS

Notice scope rule: manf refers to closest nested FROM with a relation having that attribute.

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

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

The Operator ANY

- *x* = ANY(<subquery>) is a boolean condition that is true iff *x* equals at least one tuple in the subquery result.
  - ANY acts as a *generalized OR*
  - = could be any comparison operator.
- Example: *x* > ANY(<subquery>) means *x* is not the uniquely smallest tuple produced by the subquery.
  - Note tuples must have one component only.

The Operator ALL

- *x* <> ALL(<subquery>) is true iff for every tuple *t* in the relation, *x* is not equal to *t*.
  - ALL acts as a *generalized AND*
  - That is, *x* is not in the subquery result.
- <> can be any comparison operator.
- Example: *x* >= ALL(<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.

Standalone VS Correlated Subquery

- A subquery can be either standalone or correlated (to the main query). This affects:
  - when and how often the DBMS evaluates the subquery,
  - whether or not the developer can test the subquery (to check its result).

Example: Standalone Subquery

```
SELECT bar, beer, price
FROM Sells
WHERE price = (SELECT MAX(price)
               FROM Sells);
```

- The query outputs the (bar, beer, price) tuples with the highest price of all.

Example: Standalone Subquery (2)

- The subquery is standalone because it does not refer to the main query in any way. As a result:
  - The DBMS evaluates the subquery only once, before processing the main query.
  - The developer can test the subquery by running it without the main query.

Example: Correlated Subquery

```
SELECT bar, beer, price
FROM Sells s        correlation
WHERE price = (SELECT MAX(price)
               FROM Sells
               WHERE bar = s.bar);
```

- The query outputs the (bar, beer, price) tuples with the highest price within each bar.

Example: Correlated Subquery (2)

- The query is correlated: it refers to the tuple being examined in the main query. As a result:
  - The DBMS evaluates the subquery for each tuple it examines in the main query.
  - The developer can’t test the subquery as it is: they must replace s.bar with some value(s).

Example: Correlated Subquery (3)

```
SELECT *
FROM Bars
WHERE NOT EXISTS
  (SELECT *
   FROM Frequents
   WHERE bar = Bars.name);
```

correlation

- The query outputs the bars that no drinker frequents.

Outline

1. Subqueries
2. Set Operators
3. Multiset Semantics
4. Aggregation and Grouping

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

Notice trick: subquery is really a stored table.

Solution

The drinker frequents a bar that sells the beer.

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

Outline

1. Subqueries
2. Set Operators
3. Multiset Semantics
4. Aggregation and Grouping

Multiset Semantics

- The SELECT-FROM-WHERE statement uses multiset semantics:
  - duplicate tuples are not eliminated from output
- The default for union, intersection, and difference is set semantics:
  - 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 . . .
- Force the result to be a multiset (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.

Outline

1. Subqueries
2. Set Operators
3. Multiset Semantics
4. Aggregation and Grouping

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

- Use DISTINCT inside an 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, 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.
  - Exception: COUNT of an empty set (i.e. all values are NULL) 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

- We may follow a SELECT-FROM-WHERE expression by GROUP BY and a list of attributes.
- The relation that results from the 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;
```

beer	AVG(price)
Bud	2.33
1664	2.50
…	…

Note: Tuples with a NULL value, if any, form a distinct group

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

~~SELECT beer, MIN(price)~~

~~FROM Sells~~

~~GROUP BY bar;~~

- Problem: beer (in SELECT) is not a grouping attribute: tuples in a given group may have distinct beer values.
- Therefore "beer" does not denote a well-defined value: this is why the query is illegal.

Illegal Query Example (2)

- Compare with the following query:

```
SELECT bar, MIN(price)
FROM Sells
GROUP BY bar;
```

- Here, bar (in SELECT) denotes a well-defined value: all the tuples of a given group have the same bar value by construction. This query is legal.

Illegal Query Example (3)

- 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. It can be seen as a special case of the previous rule with no grouping attribute.

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.
- Note: HAVING expresses a condition on *groups*, whereas WHERE expresses a condition on *tuples*.

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
HAVING COUNT(bar) >= 3 OR
    beer IN (SELECT name
             FROM Beers
             WHERE manf = 'Pete''s');
```

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

Beers manu-factured 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, for the same reasons).

Queries: Clause Order

SELECT …
FROM …
[WHERE …]
[GROUP BY …
  [HAVING …]]
[ORDER BY …]

Note: clause order is strict; [ ] denotes optional clauses

Queries: Execution Order

6. SELECT …
1. FROM …
2. WHERE …
3. GROUP BY …
4. HAVING …
5. ORDER BY …