Introduction to SQL

Introduction

Select-From-Where Statements

Queries over Several Relations

Subqueries

Why SQL?

- SQL is a high-level language.
  - Expresses “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*
- SQL is primarily a query language, for getting information from a database.
  - But SQL also includes a *data-definition* component for describing database schemas

History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory in the 1970's
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
  - SQL-86, SQL-89, SQL-92
  - SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011
- Commercial systems offer most, if not all, SQL-92 features, plus varying features from later standards and special proprietary features.
  - Not all examples here may work on a particular system.

Data Definition Language

Allows the specification of not only a set of relations but also information about each relation, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- The set of *indices* to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk. (Not covered in 354)

Creating (Declaring) a Relation

- Simplest form is:

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

- To delete a relation:

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

Elements of Table Declarations

- Most basic element: 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.

Declaring Keys

- An attribute or list of attributes may be declared PRIMARY KEY or UNIQUE.
- Either says that no two distinct tuples of the relation may agree in all the attribute(s) on the list.
- So keys provide a means of uniquely identifying tuples.
- There can be only one PRIMARY KEY for a relation, but possibly several UNIQUE lists of attributes.
- No attribute of a PRIMARY KEY can ever be NULL. (Why?)

Declaring Single-Attribute Keys

- Can place PRIMARY KEY or UNIQUE after the type in the declaration of the attribute.
- Example: Declare *branch_name* as the primary key for a bank's *branch*

```
CREATE TABLE branch (
                    branch_name     CHAR (15) PRIMARY KEY,
                    branch_city     CHAR (30),
                    assets          INTEGER
);
```

Example: Multiattribute Key

- Multiattribute keys are declared separately
- E.g. 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)
);
```

Create Table Construct

- An SQL relation is defined using the **create table** command:

  CREATE TABLE r (A_1 D_1, A_2 D_2, ..., A_n D_n,
  <integrity-constraint$_1$>,
  ...,
  <integrity-constraint$_k$>);

  - r is the name of the relation
  - each A_i is an attribute name in the schema of relation r
  - D_i is the data type of values in the domain of attribute A_i

- Example:

```
CREATE TABLE branch
                (branch_name    CHAR (15) PRIMARY KEY,
                 branch_city    CHAR (30),
                 assets         INTEGER
);
```

Integrity Constraints in Create Table

- NOT NULL
- PRIMARY KEY(A_1, ..., A_n) – the attributes form a primary key
- UNIQUE (A_1, ..., A_n) – the attributes together form a candidate key
  - Note the difference between

    UNIQUE (A_1, A_2) and

    UNIQUE (A_1), UNIQUE(A_2)
- Later: Other integrity constraints

Example: Declare *branch_name* as the primary key for *branch*

```
CREATE TABLE branch
        (branch_name CHAR(15),
         branch_city  CHAR(30),
         assets       INTEGER,
         PRIMARY KEY (branch_name))
```

Drop and Alter Table Constructs

- The **drop table** command deletes all information about the dropped relation from the database.
  - DROP TABLE <name>;
- The alter table command is used to add attributes to an existing relation:

  ALTER TABLE *r* ADD *A D*

  where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A.*
  - All tuples in the relation are assigned *null* as the value for the new attribute.
- The alter table command can also be used to drop attributes of a relation:

  ALTER TABLE *r* DROP *A*

  where *A* is the name of an attribute of relation *r*
  - Dropping of attributes not supported by many databases

Basic Query Structure

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

SELECT	desired attributes
FROM	one or more tables
WHERE	condition about tuples of the tables

Basic Query Structure and Relational Algebra

- A typical SQL query has the form:

SELECT	$A_1, A_2, \ldots, A_n$
FROM	$r_1, r_2, \ldots, r_m$
WHERE	E

  - A_i represents an attribute
  - r_i represents a relation
  - E is a Boolean expression.
- This query is equivalent to the relational algebra expression.

$$\prod_{A_1, A_2, \ldots, A_n}(\sigma_E(r_1 \times r_2 \times \ldots \times r_m))$$

- The result of an SQL query is a relation.
- Note: SELECT in a SQL query is distinct from σ in relational algebra

Recall: The Beer Running Example

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

Beers(name, manf)

Bars(name, addr, license)

Customers(name, addr, phone)

Likes(customer, beer)

Sells(bar, beer, price)

Frequents(customer, bar)

Example

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

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

Result of Query

name
Export
Molson Dry
Corona
. . .

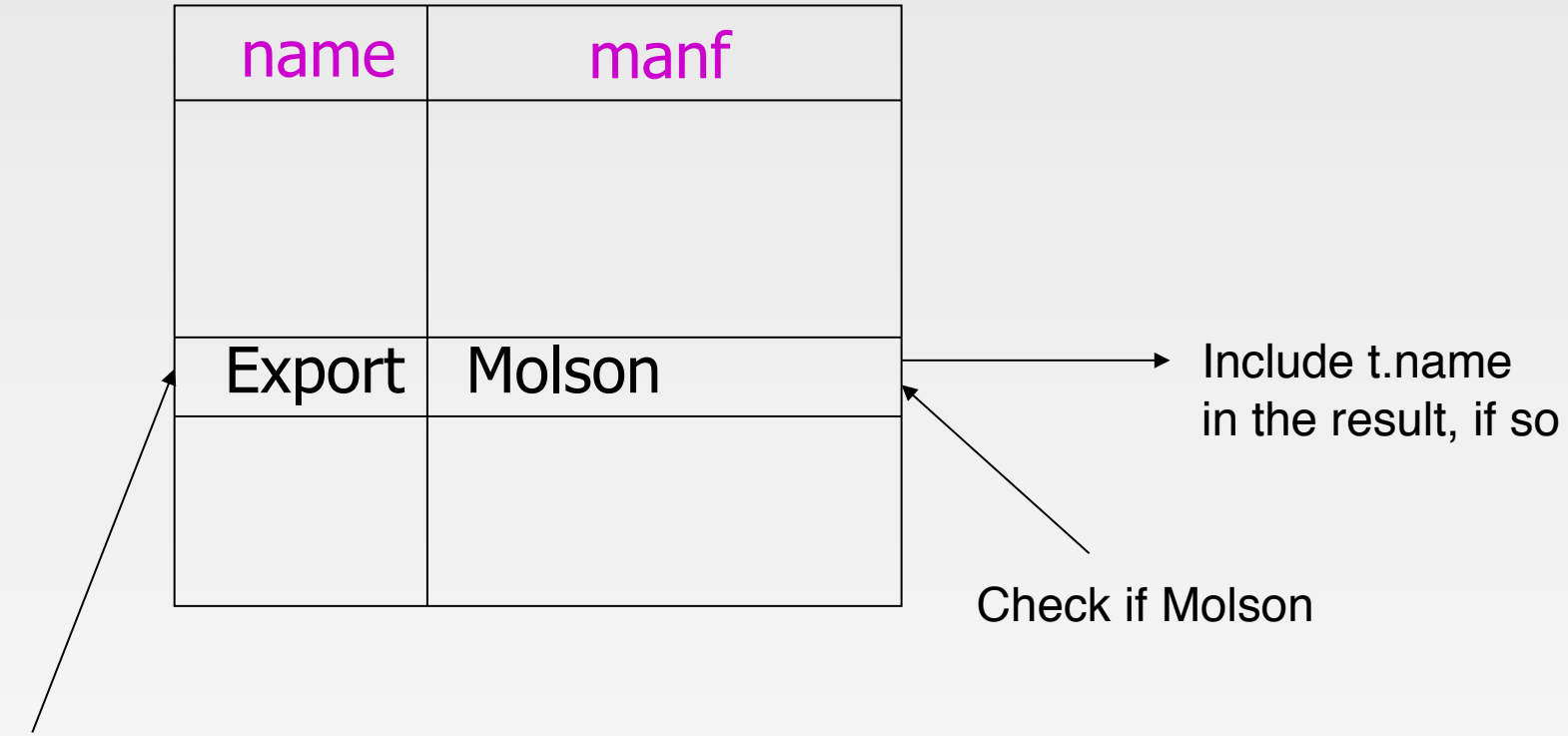
The answer is a relation with a single attribute, name, and tuples with the name of each beer by Molson, such as Export.

Semantics for a Single-Relation Query

- Take the product of the relations in the FROM clause.
- Apply the selection indicated by the WHERE clause.
- Apply the extended projection indicated by the SELECT clause.

Operational Semantics

E.g.: SELECT name
FROM Beers
WHERE manf = 'Molson';



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.

* 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 = ’Molson’;
```

Result of Query:

name	manf
Export	Molson
Molson Dry	Molson
Corona	Molson
. . .	. . .

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.
- Keyword AS is optional, but helps readability
- Example: Using Beers(name, manf):

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

Result of Query:

beer	manf
Export	Molson
Molson Dry	Molson
Corona	Molson
. . .	. . .

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*76  AS  priceInYen
FROM Sells;
```

Result of Query

bar	beer	priceInYen
Joe's	Export	285
Sue's	Sleeman	342
…	…	…

Example: Constants as Expressions

- Using Likes(customer, beer):

```
SELECT customer,
                    'likes Export' AS whoLikesExport
FROM Likes
WHERE beer = 'Export';
```

Result of Query

customer	whoLikesExport
Sally	likes Export
Fred	likes Export
…	…

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

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

Complex Conditions in WHERE Clause

- Boolean operators AND, OR, NOT.
- Comparisons =, <>, <, >, <=, >=.
  - And many other operators that produce Boolean-valued results.

Example: Complex Condition

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

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

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 Customers(name, addr, phone) find Customers with exchange 555:

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

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.
- Comparing any value (including NULL itself) with NULL yields UNKNOWN.
- A tuple is in a query answer iff 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

- Consider the following Sells relation:

bar	beer	price
Joe's Bar	Export	NULL

```
SELECT bar
FROM Sells
WHERE price < 2.00 OR price >= 2.00;
```

Query result?

Surprising Example

- From the following Sells relation:

bar	beer	price
Joe's Bar	Export	NULL

```
SELECT bar
FROM Sells
WHERE price < 2.00 OR price >= 2.00;
```

  - The WHERE clause evaluates to UNKNOWN.
  - So Joe's Bar isn't SELECT'ed.

Reason:
2-Valued Laws != 3-Valued Laws

- Some common laws, like commutativity of AND, hold in 3-valued logic.
- But not others, e.g., the *law of the excluded middle* :
  - *p* OR (NOT *p)* = TRUE in classical logic.
  - When *p* = UNKNOWN, the left side is

    MAX(½, (1 – ½)) = ½ ≠ 1.

Nulls and 3-Valued Laws

- Note that if the relations in a query have no NULL values, then
  - all the parts in a WHERE clause will be either TRUE or FALSE and
  - the value for the WHERE clause for each tuple will be either TRUE or FALSE (i.e. there will be no UNKNOWNs)

Multirelation Queries

- Interesting queries often combine data from more than one relation.
- We can include several relations in one query by listing them all in the FROM clause.
- Distinguish attributes with the same name by

  “<relation>.<attribute>” .

Example: Joining Two Relations

- Using relations Likes(customer, beer) and Frequents(customer, 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.customer = Likes.customer;
```

Formal Semantics

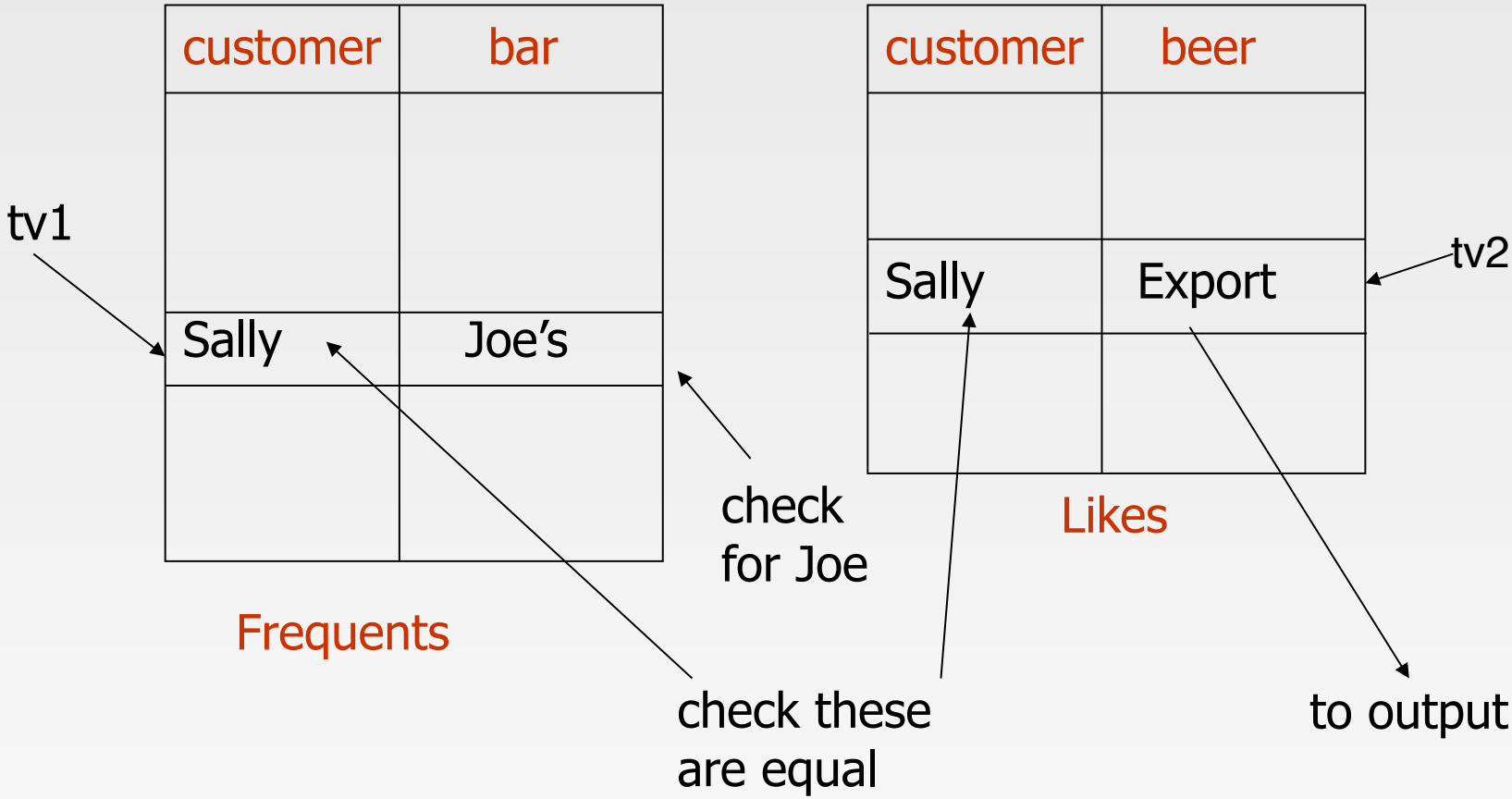
- Almost the same as for single-relation queries:
  1. Start with the product of all the relations in the FROM clause.
  2. Apply the selection condition from the WHERE clause.
  3. Project onto the list of attributes and expressions in the SELECT clause.

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

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



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 it's not essential.

Example: Self-Join

- From Beers(name, manf), find all pairs of beers by the same manufacturer.
  - Do not produce pairs like (Export, Export).
  - Produce pairs in alphabetic order, e.g. (Export, Sleeman), not (Sleeman, Export).

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

- Note: Could have used AS in the FROM clause:

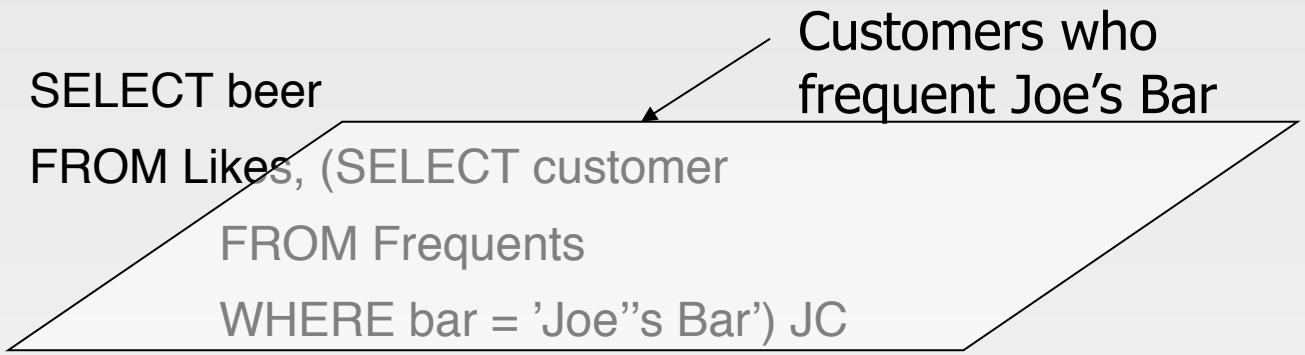
```
FROM Beers AS b1, Beers AS b2
```

Subqueries

- Recall: the result of a query is a relation.
- 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 customer
             FROM Frequents
             WHERE bar = 'Joe''s Bar') JC
WHERE Likes.customer = JC.customer;
```

Subqueries That Return One Tuple

- If a subquery is guaranteed to produce one tuple, then that subquery can be used as a value.
  - Usually, the tuple has one attribute.
  - A run-time error occurs if there is no tuple or more than one tuple.

Example: Single-Tuple Subquery

- Using Sells(bar, beer, price), find the bars that serve Sleeman for the same price Joe charges for Export.
- Two queries would certainly work:
  1. Find the price Joe charges for Export.
  2. Find the bars that serve Sleeman at that price.

Query + Subquery Solution

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

The price at which Joe sells Export

The IN Operator

- <tuple> IN (<subquery>)

  is true if and only if the tuple is a member of the relation produced by the subquery.

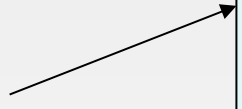
  - Opposite: <tuple> NOT IN (<subquery>).
- IN-expressions can appear in WHERE clauses.

Example: IN

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

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

The set of beers Fred likes



A Subtle Example

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

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

The First 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

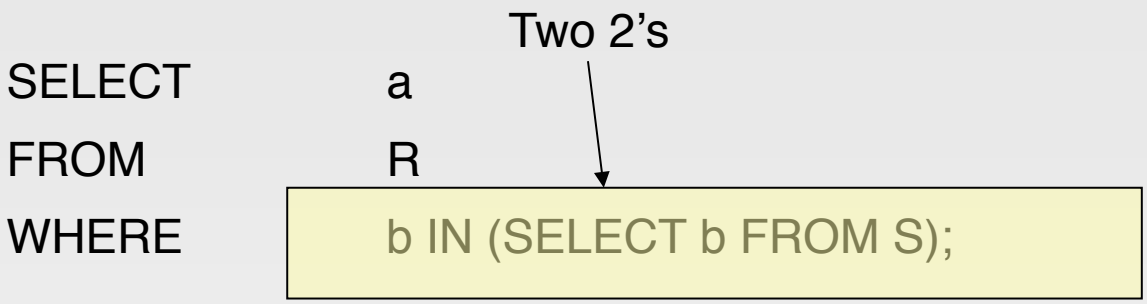
b	c
2	5
2	6

S

(1,2) with (2,5) and (1,2) with (2,6) both satisfy the condition;

1 is output twice.

IN is a Predicate About R's Tuples



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;

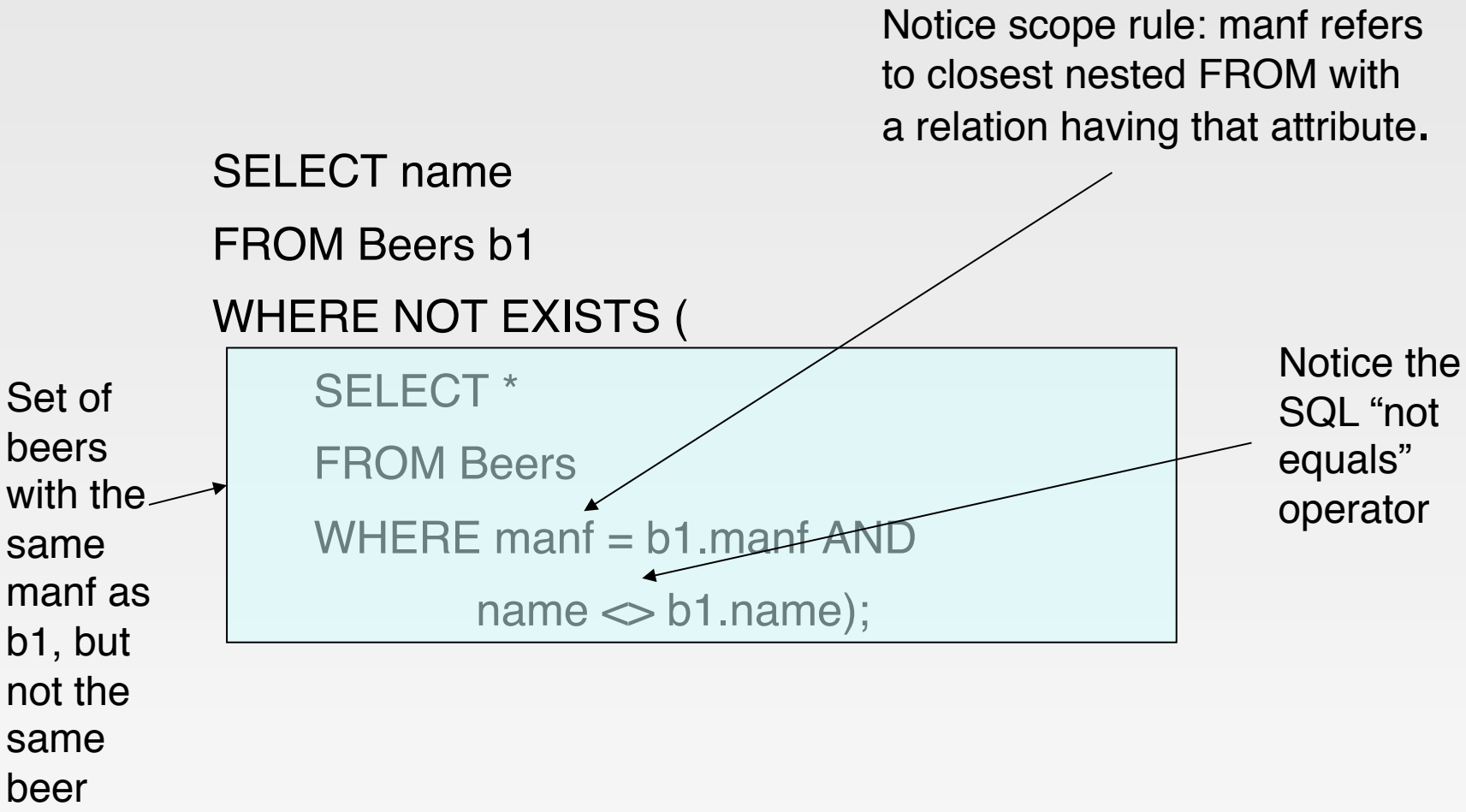
1 is output once.

Note: This example depends on SQL being a language based on bags

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 only beer made by their manufacturer.

Example: EXISTS



The Operator ANY

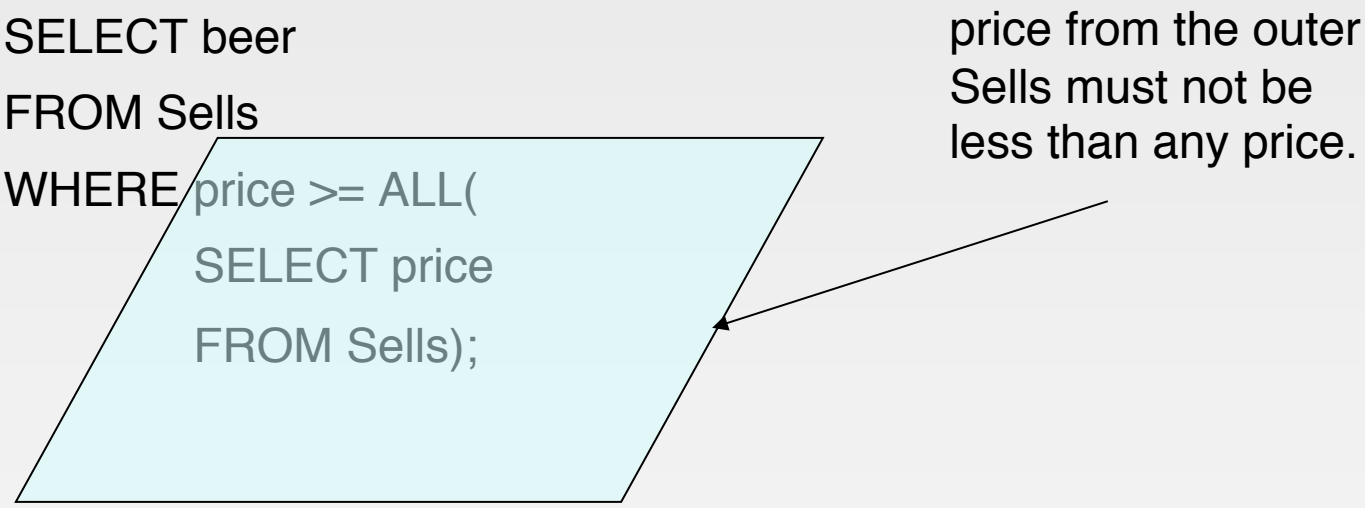
- *x* = ANY(<subquery>) is a boolean condition that is true iff *x* equals at least one tuple in the subquery result.
- = could be any comparison operator.
  - **Example**: *x* > ANY(<subquery>) means *x* is not the uniquely smallest tuple produced by the subquery.
- Synonym: SOME

The Operator ALL

- *x* <> ALL(<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* >= 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.



Union, Intersection, and Difference

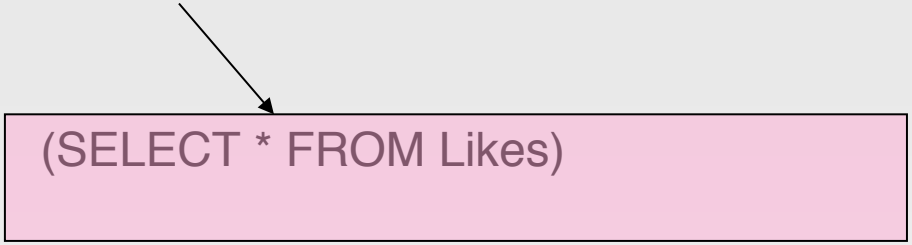
- Union, intersection, and difference of relations are expressed by the following forms, each involving subqueries:
  - (<subquery>) UNION (<subquery>)
  - (<subquery>) INTERSECT (<subquery>)
  - (<subquery>) EXCEPT (<subquery>)

Example: Intersection

- Using Likes(customer, beer), Sells(bar, beer, price), and Frequents (customer, bar), find the Customers and beers such that:
  1. The customer likes the beer, and
  2. The customer frequents at least one bar that sells the beer.

Solution

Notice trick: subquery is really a stored table.



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

The customer frequents a bar that sells the beer.

INTERSECT

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

Aside: This is an example of a query that departs from the outer SELECT-FROM-WHERE template

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 a 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 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(customer, bar) and Likes(customer, beer):

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

- Lists Customers 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.

Products and Natural Joins

- Natural join:

  R NATURAL JOIN S;

- Cartesian 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 Customers(name, addr) and Frequents(customer, bar):

  Customers JOIN Frequents ON

  name = customer;

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

End: SQL