

Apply caution

- Some algebraic laws hold in both 2-valued and 3-valued logic
 - e.g., commutative property of logical-and
- Others, however, do not:
 - e.g., law of the excluded middle
 - $p \text{ or not } p == \text{true}$ (!!)

Multirelation queries

- The interesting SQL queries usually combine data from more than one relation
- The **from** clause permits us to list those relations needed in a query
- To distinguish between attributes having the same name in separate relations, we use the dot syntax
 - <relation>.<attribute>

Multirelation queries: using two relations

- Consider two relations:
 - Likes(patron, beer)
 - Frequents(patron, pub)
- How do we find the beers liked by at least one person who frequents Bard and Banker?

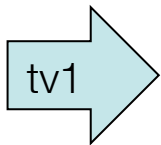
```
select beer
from Likes, Frequents
where pub = "Bard and Banker" and
       Frequents.patron = Likes.patron;
```

Formal semantics

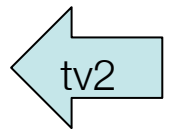
- Understanding how the query is actually implemented is almost the same as with single-relation queries
 1. Start with the product of all the relations in the **from** clause
 2. Apply the selection condition from the **where** clause
 3. Project using the list of attributes and expressions in the **select** 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 point to relation tuples satisfying the **where** clause...
 - ...send these tuples to the **select** clause.



patron	pub
...	...
Abe	Bard and Banker
...	...



patron	beer
...	...
Abe	Guinness
...	...

For these values of tv1 and tv2, check that tv1.pub is "Bard and Banker", and then check if tv1.patron is the same as tv2.patron

Explicit tuple variables

- Some queries need two copies of the same relation
- We can introduce our own tuple variables to accomplish this.
- Distinguish copies of relations:
 - Use new name following the relation as it is mentioned in the **from** clause
 - Even if we not need to use such explicit variables, we always have the option available

Example

- Consider the relation:
 - Beers(name, manf)
- Want an answer to the following:
 - Find all pairs of beers made by the same manufacturer
 - Do not produce pairs like (Blue Buck, Blue Buck)
 - Produce pairs in alphabetic order: (Amnesiac, Blue Buck) is okay, (Blue Buck, Amnesiac) is not

```
select b1.name, b2.name
from   Beers b1, Beers b2
where  b1.manf = b2.manf and
       b1.name < b2.name
```

Lexicographic ordering is used when relational operators are used on strings.

Subqueries

- A **from** clause need not always refer to an actual table/relation.
- Any expression that produces a relation can appear in the **from**!
- These expression are called **subqueries**
 - Can also appear in a **where** clause
- Example:
 - In place of a table/relation, we can use a subquery and the query its result
 - Must use a tuple variable to name tuples of the subquery result

Example

- Consider the relations:
 - Likes(patron, beer)
 - Frequents(patron, pub)
- Want an answer to the following:
 - Find the beers liked by at least one person who frequents Felicita's.

```
select beer
from Likes, (select patron
              from Frequents
              where pub = 'Felicita''s') FP
where Likes.patron = FP.patron;
```

Subqueries special case

- If a subquery is guaranteed to produce one tuple...
 - ... then the subquery can be used as a **value**
- To obtain this guarantee, we usually depend upon computing a relation with one tuple and one attribute
- However:
 - If there is no tuple in the query...
 - ... or if the query produces more than one tuple ...
 - ... then a run-time error occurs.

Example

- Consider the relation:
 - Sells(pub, beer, price)
- Want an answer to the following:
 - Find the pubs that serve Guinness for the same price Moe's sells Smithwick's

```
select pub
from Sells
where beer = 'Guinness' AND
      price = (select price
                from Sells
                where pub = 'Moe''s' AND
                      beer = 'Smithwick''s');
```

in operator

- <tuple> **in** (<subquery>):
 - Is true if and only if the tuple is a member of the relation produced by the subquery
 - Opposite would be expressed as: <tuple> **not in** (<subquery>)
- **in**-expressions can appear in **where** clauses.

Example

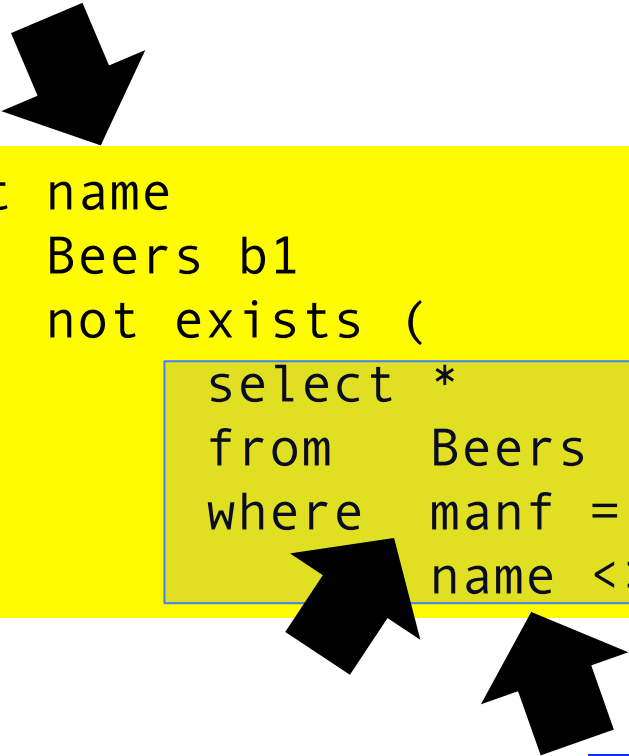
- Consider the relations:
 - Beers(name, manf)
 - Likes(patron, beer)
- Want an answer to the following:
 - Find the name and manufacturer of each beer that Barack likes

```
select *  
from   Beers  
where  name in (select beer  
                from   Likes  
                where  patron = 'Barack');
```

exists operator

- **exists** (<subquery>):
 - Is true if and only if the **subquery result is not empty**
- Interesting oddity:
 - Many of the queries using exists actually use the negation
 - Trick / hard part / crafty hackery / sublime talent is knowing when some natural-language statement can be inverted into a **not exists** clause.
 - Usually involves careful use of name scope
- Example:
 - Using Beers(name, manf), find those beers that are the unique beer made by their manufacturer.

Example



```
select name
from   Beers b1
where  not exists (
    select *
    from   Beers
    where  manf = b1.manf AND
           name <> b1.name);
```

Scope rule: **name** and **manf** refer to the closest nested from with a relation having that attribute.