Outerjoin

- Suppose we perform the join $R \bowtie_C S$
- It may be the case that...
  - There is a tuple of R ...
  - ... which has no tuple of S to join.
  - R is therefore said to be **dangling**
- Same situation can exist for a tuple of S.
- **Outerjoin** preserves dangling tuples
  - This is done by padding attributes appropriate with our good friend, the NULL value.

Example

R

A	B
1	2
4	5

S

B	C
2	3
6	7

(1,2) from R joins with (2,3) from S. Tuples (4,5) from R and (6,7) from S are dangling.

R OUTERJOIN S =

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

SQL and outerjoin

- R outer join S
  - This is the central part of an outerjoin expression
- The expression can be modified as either
  - **R natural outer join S** or
  - **R outer join S on condition**
- Also required: **left**, **right**, **full** before **outer**
  - **left**: null padding of tuples in R only
  - **right**: null padding of tuples in L only
  - **full**: pad both (i.e., the default)

Syntax page

Patrons(name, addr, phone)
Frequents(patron, pub)

```
select * from Patrons
     full outer join Frequents on name = patron;
```

```
select * from Patrons
     left outer join Frequents on name = patron;
```

```
select * from Patrons
     right outer join Frequents on name = patron;
```

SQL and aggregations

- AgOps can be applied to a column in a **select** clause
  - **sum**
  - **avg**
  - **count**
  - **min**
  - **max**
- These produce the aggregation on that column
- Also: **count(*)**
  - Result is the number of tuples

Example: aggregation

- Consider the relation:
  - Sells(pub, beer, price)
- Want an answer to the following:
  - Find the average price of Blue

```
select avg(price)
from   Sells
where  beer = 'Blue';
```

Example: eliminate aggregate duplicates

- Consider the relation:
  - Sells(pub, beer, price)
- Want an answer to the following:
  - Find the number of different prices charged for Labatt's Blue

```
select count(distinct price)
from   Sells
where  beer = 'Blue';
```

Our friend NULL

- NULLs are ignored in aggregations
  - A null never contributes to a **sum**, **avg** or **count**
  - It can never be the **min** or **max** of a column
- However!
  - If there are no non-null values in a column, then the result of the aggregation is itself null
  - Only exception: **count** of an empty set is 0

```
select count(*)
from   Sells
where  beer = 'Blue';
```

The number of pubs that sell Blue.

```
select count(price)
from   Sells
where  beer = 'Blue';
```

The number of pubs that sell Blue at a known price.

Grouping

- **select-from-where** may be followed with **group by** and a list of attributes
- Resulting relation has tuples grouped according to values in listed attributes
  - Any aggregation is applied only within the group.
- Consider the relation:
  - Sells (pub, beer, price)
- Want an answer to the following:
  - Find the average price fo each beerof

```
select beer, avg(price)
from   Sells
group by beer;
```

beer	avg(price)
...	...
Blue	2.33
Guinness	7.6
...	...

Example

- Consider the relations:
  - Sells(pub, beer, price)
  - Frequents(patron, pub)
- Want an answer to the following:
  - Find, for each patron, the average price of Guinness at the pubs they frequent

```
select patron, avg(price)
from   Frequents, Sells
where  beer = 'Guinness' AND
       Frequents.pub = Sells.pub
group by patron;
```

Compute all patron-pub-price triples for Guinness...

... then group them by patron.

Select and aggregation: restrictions

- We must be careful with aggregations
- If any AgOp is used, then one of the following must be true for each **select** element
  - Either it is aggregated, or
  - It is an attribute on the **group by** list
- Counter-example:
  - Find the pub that sells Blue for the lowest price

```
select pub, min(price)
from   Sells
where  beer = 'Blue';
```

having clauses

- A **group by** clause may be followed by a **having <condition>** clause
- With such a following clause, condition applies to the group
  - Groups not satisfying the condition are eliminated from the result.
- Example: Consider Sells(pub, 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 Phillips.

Example solution

```
select beer, avg(price)
from   Sells
group by beer
having count(pub) >=3 or
    beer in (select name
             from Beers
             where manf = 'Phillips');
```

Beers groups with at least three non-null pubs and also beer groups where the manufacturer is Phillips.

Beers manufactured by Phillips

Note the use of the subquery.

Having clauses: restrictions

- Anything may appear in a **having** subquery...
- ... but outside of subqueries, attributes may only be referenced if they are either:
  - a grouping attribute, or
  - aggregated
- These are the same restrictions as apply to select clauses with aggregation

Modifying databases

- A modification command does not return a result
  - Queries return results
  - Modification commands change 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 or values of an existing tuple or tuples
- (We leave aside for now changes that can be made to database schema.)

1. Insertion

- Insertion of a single tuple:

```
insert into <relation name>
values (<list of values>);
```

- Example: Add to Likes(patron, beer) the fact that Agnes likes Chimay Blue

```
insert into Likes
values ('Agnes', 'Chimay Blue');
```

Specifying attributes during insertion

- To the relation name, we may add a list of attributes
- There are reasons why we may wish to do this
  - We do not have, ready to hand, the standard order of attributes in the relation (i.e., we've forgotten)
  - We do not have values for all attributes, and therefore we need the DBMS to fill in missing components with NULLs or default values
- Repeating the last example:

```
insert into Likes(beer, patron)
values ('Chimay Blue', 'Agnes');
```

Note how the order of attributes is not the same as our usual listing for this relation.

Specifying default values

- In a **create table** statement, we can follow an attribute with a default value
  - Use the keyword **default**
- When a tuple is inserted that has no value for that attribute...
  - ... then the default will be used.

```
create table patrons (
    name  char(30) primary key,
    addr  char(50) default '3141 Shelbourne Street',
    phone char(16)
);
```

Insert with default value

```
insert into Patrons(name)
values ('Cliff');
```

name	addr	phone
...	...	...
Cliff	3141 Shelbourne Street	null
...	...	...

Inserting many tuples

- An entire result of some query may be inserted into a relation

```
insert into <relation>
(<subquery>);
```

- Example:
  - Using Frequents(patron, pub), enter all of Sammy's buddies into a new relation named PotBuddies (i.e., all those patrons who frequent at least one bar Sammy also frequents)

```
insert into PotBuddies
(select p2.patron
 from Patrons p1, Patrons p2
 where p1.patron = 'Sammy' and
       p2.patron <> 'Sammy' and
       p1.pub = p2.pub
);
```

The other patron...

Pairs of Patron tuples where the first is for Sammy, the second is for someone else, and the pubs are the same.

2. Deletion

- Deletion of tuples satisfying a condition from some relation

```
delete from <relation>
where <condition>;
```

- Example: Delete from Likes(patron, beer) the fact that Abe likes Blue (yay!)

```
delete from Likes
where patron = 'Abe' and
      beer = 'Blue';
```

Deleting all tuples

- Example: remove all tuples from Likes (i.e., make the relation empty).

```
delete from Likes;
```

- Note there is no **where** clause.
- Presumably the programmer writing the clause knows what they are doing (e.g., like "\rm –rf *" in Unix)

Deleting some tuples

- Example:
  - Delete from Beers(name, manf) all beers for which there is another beer from the same manufacturer.
  - Equivalently: Only keep in Beers(name, manf) those tuples for manufacturers that make a single beer

```
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 are represented by the tuple **b**.

What the h*ll just happened?

Deleting some tuples

- Example:
  - Delete from Beers(name, manf) all beers for which there is another beer from 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 are represented by the tuple **b**.

Semantics of deletion

- Pretend Coors Brewing makes only 'Coors' and 'Coors Light'
  - They and Molson are now together, so beer choice is increasingly confusing for someone born in the 1960s
- Suppose we come to the tuple **b** for 'Coors' first
  - The subquery is non-empty, because of the 'Coors Light' tuple
  - Therefore delete 'Coors'.
- But now when **b** comes to the tuple for 'Coors Light', is that tuple deleted, too?

Semantics of deletion

- Answer: **yes**.
- The reason: Deletion is a two stage process
  - Mark all tuples for which the **where** condition is satisfied
  - Delete the marked tuples