>>

SQL: Queries on Multiple Tables

^ >>

❖ Queries on Multiple Tables

Queries involving a single table are useful.

Exploiting all data in the DB requires

- combining data from multiple tables
- typically involving primary/foreign key matching

Example: Which brewers makes beers that John likes?

```
select b.brewer
from   Beers b join Likes L on (b.name = L.beer)
where  L.drinker = 'John';
```

Info on brewers is in **`Beers`**; info on who likes what in **`Likes`**.

Need to combine info from both tables using "common" attributes

<< ^ >>

❖ Queries on Multiple Tables (cont)

Example **Beers** and **Likes** tuples:

```
Beers(80/-, Caledonian, Scotch Ale)     Likes(John, Sculpin)
Beers(New, Toohey's, Lager)             Likes(John, Red Nut)
Beers(Red Nut, Bentspoke, Red IPA)      Likes(Adam, New)
Beers(Sculpin, Ballast Point, IPA)      Likes(John, 80/-)
```

"Merged" tuples resulting from

- **Beers b join Likes L on (b.name = L.beer)**

```
Joined(80/-, Caledonian, Scotch Ale, John, 80/-)
Joined(New, Toohey's, Lager, Adam, New)
Joined(Red Nut, Bentspoke, Red IPA, John, Red Nut)
Joined(Sculpin, Ballast Point, IPA, John, Sculpin)
```

In the query, the **where** clause removes all tuples not related to John i.e. it removes the tuple to do with Adam

<< ^ >>

❖ Join

Join is the SQL operator that combines tuples from tables.

Such an important operation that several variations exist

- natural join matches tuples via equality on common attributes
  where common attributes means same column name
- equijoin matches tuples via equality on specified attributes
  will perform equality only on attributes you say to
- theta-join matches tuples via a boolean expression
  which could be an equality expression but it doesn't have to be
- outer join like theta-join, but includes non-matching tuples

We focus on theta-join and outer join in this course

<< ^ >>

❖ Join (cont)

Join fits into **SELECT** queries as follows:

```
SELECT Attributes
FROM   R1
       JOIN R2 ON (JoinCondition1)
       JOIN R3 ON (JoinCondition2)
       ...
WHERE  Condition
```

Can include an arbitrary number of joins.

WHERE clause typically filters out some of the joined tuples.

<< ^ >>

❖ Join (cont)

Alternative syntax for joins:

```
SELECT brewer
FROM   Likes L, Beers b
WHERE  L.beer = b.name
       AND L.drinker = 'John';
```

Join condition(s) are specified in the **`WHERE`** clause

We prefer the explicit **`JOIN`** syntax, but this is sometimes more compact

Note: duplicates could be eliminated by using **`distinct`**

<< ^ >>

❖ Join (cont)

Operational semantics of **`R1 JOIN R2 ON (Condition)`**:

```
FOR EACH tuple t1 in R1 DO
    FOR EACH tuple t2 in R2 DO
        check Condition for current
           t1, t2 attribute values
        IF Condition holds THEN
            add (t1,t2) to result
        END
    END
END
```

for every possible pair, add the pair to the result if the condition is satisfied.

Easy to generalise: add more relations, include **`WHERE`** condition

Requires one tuple variable for each relation, and nested loops over relations.
But this is not how it's actually computed!

<< ^ >>

❖ Name Clashes in Conditions

If a **SELECT** statement

- refers to multiple tables
- some tables have attributes with the same name

use the table name to disambiguate.

this is just basic namespacing

Example: Which hotels have the same name as a beer?

```
SELECT Bars.name
FROM   Bars, Beers
WHERE  Bars.name = Beers.name;
-- or, using table aliases ...
SELECT r.name
FROM   Bars r, Beers b
WHERE  r.name = b.name
```

<< ^ >>

❖ Explicit Tuple Variables

Table-dot-attribute doesn't help if we use same table twice in **SELECT**.

To handle this, define new names for each "instance" of the table

```
SELECT r1.a, r2.b  FROM  R r1, R r2  WHERE  r1.a = r2.a
```

Example: Find pairs of beers by the same manufacturer.

```
SELECT b1.name, b2.name
FROM   Beers b1 JOIN Beers b2 ON (b1.brewer = b2.brewer)
WHERE  b1.name < b2.name;
```

The **WHERE** condition is used to avoid:

- pairing a beer with itself e.g. **(New,New)**
- same pairs with different order e.g. **(New,Old) (Old,New)**

remember that join is "for all pairs, check condition and add if it's satisfied".
Note that this is n^2 (all pairs, including duplicates and pairing with self), instead of n(n-1)/2 (unique pairs), which means you need some condition to eliminate the duplicates and self-pairs if you don't want them.

The were b1.name < b2.name ensures == can't pass (takes care or self-referential), and now there is a strict ordering (so pair a,b will only ever appear as a,b and never b,a which takes care of the duplicates).

<< ^ >>

❖ Outer Join

Join only produces a result tuple from t_R and t_S where ...

- there are appropriate values in both tuples
- so that the join condition is satisfied

```
SELECT * FROM R JOIN S WHERE (Condition)
```

Sometimes, we want a result for *every* **R** tuple

- even if some **R** tuples have no matching **S** tuple

These kinds of requests often include "for each" or "for every"

<< ^ >>

❖ Outer Join (cont)

Example: for each suburb with a bar, find out who drinks there.

The previous join type we were doing was a theta-join, where the ON (b1.brewer = b2.brewer) was the boolean condition to join on.

Theta-join only gives results for suburbs where people drink.

we want to also know the one's where no one drinks there!

```
SELECT B.addr, F.drinker
FROM   Bars B
       JOIN Frequents F ON (F.bar = B.name)
ORDER  BY addr;
```

```
   addr     | drinker
------------+---------
 Coogee     | Adam
 Coogee     | John
 Kingsford  | Justin
 Sydney     | Justin
 The Rocks  | John
```

But what if we want all suburbs, even if some have are no drinkers?

This is from an older and simpler instance of the beers database.

<< ^ >>

❖ Outer Join (cont)

An outer join solves this problem.

For *R* **OUTER JOIN** *S* **ON** (*Condition*)

- all "tuples" in *R* have an entry in the result
- if a tuple from *R* matches tuples in *S*,
  we get the normal join result tuples
- if a tuple from *R* has no matches in *S*,
  the attributes supplied by *S* are **NULL**

this is pretty intuitive

This outer join variant is called **LEFT OUTER JOIN**.

This probably isn't correct, but remember it as a left outer join because it is the first table (the left table) that will always be in the result.

<< ^ >>

❖ Outer Join (cont)

Solving the example query with an outer join:

```
SELECT B.addr, F.drinker
FROM   Bars B
       LEFT OUTER JOIN Frequents F on (F.bar = B.name)
ORDER BY B.addr;
```

```
    addr     | drinker
-------------+---------
 Coogee      | Adam
 Coogee      | John
 Kingsford   | Justin
 Randwick    |
 Sydney      | Justin
 The Rocks   | John
```

Note that Randwick is now mentioned (because of the Royal Hotel).

<< ^ >>

❖ Outer Join (cont)

Operational semantics of **`R1 LEFT OUTER JOIN R2 ON (Cond)`**:

```
FOR EACH tuple t1 in R1 DO
    nmatches = 0
    FOR EACH tuple t2 in R2 DO
        check Cond for current
           t1, t2 attribute values
        IF Cond holds THEN
            nmatches++
            add (t1,t2) to result
        END
    END
    IF nmatches == 0 THEN
        t2 = (null,null,null,...)
        add (t1,t2) to result
    END
END
```


i.e. if the first tuple doesn't match with any of the second tuples (i.e. you compare 1 of the first type with all N of the second type), then you add a nulled out second tuple as the partner for that first tuple. If it had >= 1 matches, then don't add the null partner because it already has valid partner(s).

<< ^ >>

❖ Outer Join (cont)

Many RDBMSs provide three variants of outer join:

- *R* **`LEFT OUTER JOIN`** *S*
  - behaves as described above
- *R* **`RIGHT OUTER JOIN`** *S*
  - includes all tuples from *S* in the result
  - **`NULL`**-fills any *S* tuples with no matches in *R*

note that this isn't the same as doing S left outer join R

- *R* **`FULL OUTER JOIN`** *S*
  - includes all tuples from *R* and *S* in the result
  - those without matches in other relation are **`NULL`**-filled

<< ^

❖ Subqueries

The result of a query can be used in the **WHERE** clause of another query.

Case 1: Subquery returns a single, unary tuple

```
SELECT * FROM R WHERE R.a = (SELECT S.x FROM S WHERE Cond1)
```

Case 2: Subquery returns multiple values

```
SELECT * FROM R WHERE R.a IN (SELECT S.x FROM S WHERE Cond2)
```

This approach is often used in the initial discussion of SQL in some textbooks.

These kinds of queries can generally be solved *more efficiently* using a join

```
SELECT * FROM R JOIN S ON (R.a = S.x) WHERE Cond
```

Produced: 3 Oct 2020