Lecture 7: Two-Table Queries

CS1106/CS6503– Introduction to Relational Databases

Dr Kieran T. Herley

2019-2020

School of Computer Science & Information Technology
University College Cork

Summary

Two-table database designs. Cartesian product. Simple (inner) joins and self joins.

Favourite Foods DB Revisited

persons

person_id	*first_name*	*last_name*	*gender*	*date_of_birth*	*street*	*town*	*county*
345678	Aoife	Ahern	F	1993-01-25	123 Brown Street	Cork	Cork
467389	Barry	Barry	M	1980-06-30	58 Green Street	Tralee	Kerry
356489	Ciara	Callaghan	F	1993-03-14	23 White Avenue	Limerick	Limerick
986347	Declan	Duffy	M	1993-11-03	101 Black Crescent	Cork	Cork
561728	Eimear	Early	F	1993-07-18	45 Red Square	Thurles	Tipperary
836467	Fionn	Fitzgerald	M	1994-06-13	17 Yellow Lane	Bandon	Cork

favourite_foods

person_id	*food*
345678	Crisps
345678	Beer
345678	Nutella
467389	Chips
467389	Chocolate
356489	Ice Cream
356489	Chocolate
986347	Pizza
986347	Beer
986347	Crisps
561728	Pizza
561728	Chocolate
561728	Brussels Sprouts
836467	Ice Cream
836467	Nutella

- `persons` table captures individuals' details
- `favourite_foods` captures who likes what
- e.g. 836467 (Fionn Fitzgerald) is partial to ice cream and Nutella

Why Not Just Mash the Two Tables Together?

persons_and_foods

person_id	*first_name*	*last_name*	···	*food*
345678	Aoife	Ahern	···	Crisps
345678	Aoife	Ahern	···	Beer
345678	Aoife	Ahern	···	Nutella
467389	Barry	Barry	···	Chips
467389	Barry	Barry	···	Chocolate
⋮	⋮	⋮	⋮	⋮
836467	Fionn	Fitzgerald	···	Nutella

Redundancy Some information is replicated several times (e.g. Aoife's details)

Anomalies

- What is we wish to update Aoife's details e.g. change address
- Need to modify multiple rows otherwise table contains inconsistent information

DB designers go to great length to avoid this sort of situation

Queries That Span Tables

persons

person_id	*first_name*	*last_name*	*gender*	*date_of_birth*	*street*	*town*	*county*
345678	Aoife	Ahern	F	1993-01-25	123 Brown Street	Cork	Cork
467389	Barry	Barry	M	1980-06-30	58 Green Street	Tralee	Kerry
356489	Ciara	Callaghan	F	1993-03-14	23 White Avenue	Limerick	Limerick
986347	Declan	Duffy	M	1993-11-03	101 Black Crescent	Cork	Cork
561728	Eimear	Early	F	1993-07-18	45 Red Square	Thurles	Tipperary
836467	Fionn	Fitzgerald	M	1994-06-13	17 Yellow Lane	Bandon	Cork

favourite_foods

person_id	*food*
345678	Crisps
345678	Beer
345678	Nutella
467389	Chips
467389	Chocolate
356489	Ice Cream
356489	Chocolate
986347	Pizza
986347	Beer
986347	Crisps
561728	Pizza
561728	Chocolate
561728	Brussels Sprouts
836467	Ice Cream
836467	Nutella

- What about queries that require info. from *both* tables?
    - List the favourite foods of the person(s) named Aoife Ahern
    - List all the persons who like pizza and beer

Cartesian Product

Cartesian Product The Cartesian product of sets $\mathcal{S}$ and $\mathcal{T}$ is the set of pairs of the form (s, t), where $s \in \mathcal{S}$ and $t \in \mathcal{T}$.

Example

$$\begin{array}{rcl}
\mathcal{S} & = & \{2, 3, 5\} \\
\mathcal{T} & = & \{a, e, i, o, u\} \\
\mathcal{S} \times \mathcal{T} & = & \begin{array}{lllllll} \{ & (2,a), & (2,e), & (2,i), & (2,o), & (2,u), & \\ & (3,a), & (3,e), & (3,i), & (3,o), & (3,u), & \\ & (5,a), & (5,e), & (5,i), & (5,o), & (5,u) & \} \end{array}
\end{array}$$

Note size of $\mathcal{S} \times \mathcal{T}$ is the size of $\mathcal{S}$ multiplied by the size of $\mathcal{T}$

Cartesian Product In SQL

Can use CROSS JOIN to form Cartesian product

```
SELECT *
FROM persons CROSS JOIN favourite_foods;
```

person_id	*first_name*	*last_name*	⋯	*person_id*	*food*
345678	Aoife	Ahern	⋯	345678	Crisps
467389	Barry	Barry	⋯	345678	Crisps
356489	Ciara	Callaghan	⋯	345678	Crisps
986347	Declan	Duffy	⋯	345678	Crisps
561728	Eimear	Early	⋯	345678	Crisps
836467	Fionn	Fitzgerald	⋯	345678	Crisps
345678	Aoife	Ahern	⋯	345678	Beer
467389	Barry	Barry	⋯	345678	Beer
⋮	⋮	⋮	⋮	⋮	
836467	Fionn	Fitzgerald	⋯	836467	Nutella

$$
\begin{aligned}
\#\,\text{rows()} \quad &= \\
&\#\,\text{rows(persons)}\times \\
&\#\,\text{rows(favourite\_food}
\end{aligned}
$$

$$
\begin{aligned}
\#\,\text{cols()} \quad &= \\
&\#\,\text{cols(persons)}+ \\
&\#\,\text{cols(favourite\_foods}
\end{aligned}
$$

Cartesian Product In SQL cont'd

Most rows in cross join are meaningless

person_id	*first_name*	*last_name*	···	*person_id*	*food*	
345678	Aoife	Ahern	···	345678	Crisps	*
467389	Barry	Barry	···	345678	Crisps	
356489	Ciara	Callaghan ···		345678	Crisps	
986347	Declan	Duffy	···	345678	Crisps	
561728	Eimear	Early	···	345678	Crisps	
836467	Fionn	Fitzgerald ···		345678	Crisps	
345678	Aoife	Ahern	···	345678	Beer	*
467389	Barry	Barry	···	345678	Beer	
⋮	⋮	⋮	⋮	⋮		
836467	Fionn	Fitzgerald	···	836467	Nutella	

Cartesian Product In SQL cont'd

Most rows in cross join are meaningless , but not all . . .

person_id	*first_name*	*last_name*	···	*person_id*	*food*	
345678	Aoife	Ahern	···	345678	Crisps	*
467389	Barry	Barry	···	345678	Crisps	
356489	Ciara	Callaghan ···		345678	Crisps	
986347	Declan	Duffy	···	345678	Crisps	
561728	Eimear	Early	···	345678	Crisps	
836467	Fionn	Fitzgerald ···		345678	Crisps	
345678	Aoife	Ahern	···	345678	Beer	*
467389	Barry	Barry	···	345678	Beer	
⋮	⋮	⋮	⋮	⋮		
836467	Fionn	Fitzgerald	···	836467	Nutella	

Cartesian Product In SQL cont'd

- On closer inspection *some* rows in the product are potentially meaningful

person_id	*first_name*	*last_name*	⋯	*person_id*	*food*	
345678	Aoife	Ahern	⋯	345678	Crisps	*
467389	Barry	Barry	⋯	345678	Crisps	
356489	Ciara	Callaghan ⋯		345678	Crisps	
986347	Declan	Duffy	⋯	345678	Crisps	
561728	Eimear	Early	⋯	345678	Crisps	
836467	Fionn	Fitzgerald ⋯		345678	Crisps	
345678	Aoife	Ahern	⋯	345678	Beer	*
467389	Barry	Barry	⋯	345678	Beer	
⋮	⋮	⋮	⋮	⋮		
836467	Fionn	Fitzgerald	⋯	836467	Nutella	

- The highlighted rows are formed by coupling
  - 345678's (i.e. Aoife's) row from the `persons` table
  - 345678's rows from the `favourite_foods` table

  These rows collectively contain information on Aoife's weaknesses
- If only we could filter out the "useless" rows from the product and retain only the "useful ones"

Joins In SQL

- Consider the familiar SELECT-FROM template

  SELECT $*$ **FROM** $\boxed{X}$;

  here X specifies a table to which query is to be applied

-

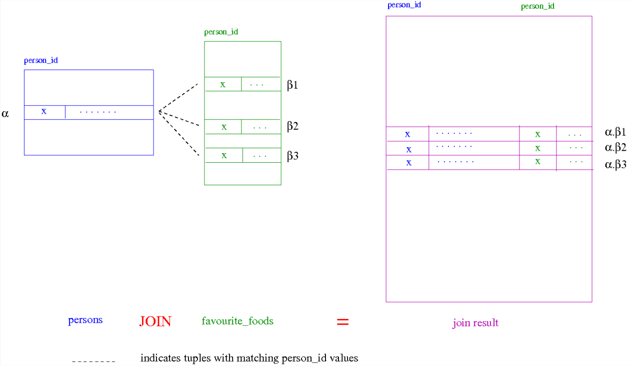
  ```
  SELECT *
  FROM persons JOIN favourite_foods
      ON persons.person_id = favourite_foods . person_id ;
  ```

  applies the SELECT-FROM to a following "table" (our X for this query)

  - subset of Cartesian product of `persons` and `favourite_foods` tables
  - but includes only those (combined) rows where the `person_id` value from the `persons` row and that of the `favourite_foods` match

- Note use of `persons.person_id` notation to distinguish between two `person_id` columns in product

Joins In SQL cont'd



```
for each row-pair (alpha, beta) from persons X favourite_foods do
    if alpha.person_id and beta.person_id are equal then
        Include (alpha, beta) in result
```

Back to Our Query

```
SELECT *
FROM persons JOIN favourite_foods
  ON persons.person_id = favourite_foods . person_id ;
```

person_id	*first_name*	*last_name*	···	*person_id*	*food*
345678	Aoife	Ahern	···	345678	Crisps
345678	Aoife	Ahern	···	345678	Beer
345678	Aoife	Ahern	···	345678	Nutella
467389	Barry	Barry	···	467389	Chips
467389	Barry	Barry	···	467389	Chocolate
356489	Ciara	Callaghan	···	356489	Ice Cream
356489	Ciara	Callaghan	···	356489	Chocolate
986347	Declan	Duffy	···	986347	Pizza
986347	Declan	Duffy	···	986347	Beer
986347	Declan	Duffy	···	986347	Crisps
561728	Eimear	Early	···	561728	Pizza
561728	Eimear	Early	···	561728	Chocolate
561728	Eimear	Early	···	561728	Brussels Sprouts
836467	Fionn	Fitzgerald	···	836467	Ice Cream
836467	Fionn	Fitzgerald	···	836467	Nutella

Fifteen rows each formed by glueing together a row from `favourite_foods` with the row from `persons` that corresponds to the individual whose food preference it is

An Example

- List all the students who like pizza
- Can use WHERE clause to further filter results

```
SELECT *
FROM persons JOIN favourite_foods
    ON persons.person_id = favourite_foods . person_id
WHERE food = 'Pizza';
```

- Can interpret as a standard SELECT-FROM-WHERE that is applied to the join of the two tables (on person_id values)– essentially result of previous query
- Yields following result

person_id	*first_name*	*last_name*	· · ·	*person_id*	*food*
986347	Declan	Duffy	· · ·	986347	Pizza
561728	Eimear	Early	· · ·	561728	Pizza

A Tidier Version

-

  ```
  SELECT first_name, last_name, ' likes ', food
  FROM
      persons AS p
      JOIN favourite_foods AS f
      ON p.person_id = f.person_id
  WHERE food = 'pizza';
  ```

- Clause `persons AS p`
  - attaches shorter name (`p`) to table `persons`
  - can use `p` instead of `persons` throughout query
  - improves readability

Reasoning About Join Queries

- Recall

```
SELECT first_name, last_name, '  likes ', food
FROM
    persons AS p
    JOIN
     favourite_foods AS f
    ON
        p.person_id = f.person_id
WHERE food = 'Pizza';
```

- Can be helpful to consider query execution in three stages [1]
    1. ("Create" full Cartesian product $\mathcal{C}$ of tables `persons` and `favourite_foods`)
    2. ("Filter" $\mathcal{C}$ to retain (in $\mathcal{C}'$) only those rows in $\mathcal{C}$ that satisfy the ON condition)
    3. Filter $\mathcal{C}'$ to retain (in $\mathcal{R}$) only those rows in $\mathcal{C}'$ that satisfy the WHERE condition and return $\mathcal{R}$ as the result of the query

[1] Conceptual model only; actual mechanics of query execurion may follow different approach

Example 3

- List all pairs of students who hail from the same county (not the same as all those from a specific county)
- Use just the persons table, but forms join of table with itself (two “copies”)!

Example 3

- List all pairs of students who hail from the same county (not the same as all those from a specific county)
- Use just the persons table, but forms join of table with itself (two “copies”)!
- First stab:

```
SELECT
    p1.first_name, p1.last_name, ' and ',
    p2.first_name, p2.last_name, ' both come from ',
    p1.county
FROM
    persons AS p1
    JOIN persons AS p2
    ON p1.county = p2.county;
```

- Any problems with this?

A Closer Look

- Query completion
  - Typical row from Cartestian product

p_id		county		p_id		county	
x	· · ·	Cork	· · ·	y	· · ·	Kerry	· · ·
from persons$_1$				from persons$_2$			

  - Typical row from product filtered by ON condition

p_id		county		p_id		county	
x	· · ·	Cork	· · ·	y	· · ·	Cork	· · ·
from persons$_1$				from persons$_2$			

A Closer Look

- Query completion
  - Typical row from Cartestian product

p_id		county		p_id		county	
x	...	Cork	...	y	...	Kerry	...

from persons$_1$ from persons$_2$

  - Typical row from product filtered by ON condition

p_id		county		p_id		county	
x	...	Cork	...	y	...	Cork	...

from persons$_1$ from persons$_2$

- Slight problem
  - Result may contain duplicates (Aoife-Declan and Declan-Aoife)

p_id		county		p_id		county	
x	...	Cork	...	y	...	Cork	...
y	...	Cork	...	x	...	Cork	...

from persons$_1$ from persons$_2$

A Closer Look

- Query completion
  - Typical row from Cartestian product

p_id		county		p_id		county	
x	···	Cork	···	y	···	Kerry	···

from $\texttt{persons}_1$ from $\texttt{persons}_2$

  - Typical row from product filtered by ON condition

p_id		county		p_id		county	
x	···	Cork	···	y	···	Cork	···

from $\texttt{persons}_1$ from $\texttt{persons}_2$

- Slight problem
  - Result may contain duplicates (Aoife-Declan and Declan-Aoife)

p_id		county		p_id		county	
x	···	Cork	···	y	···	Cork	···
y	···	Cork	···	x	···	Cork	···

from $\texttt{persons}_1$ from $\texttt{persons}_2$

  - May also contain self-pairs (Aoife-Aoife)

p_id		county		p_id		county	
x	···	Cork	···	x	···	Cork	···

from $\texttt{persons}_1$ from $\texttt{persons}_2$

Our Query

- First attempt lists each pair twice (Aoife-Declan and Declan-Aoife) and also “self pairs” like (Aoife-Aoife)
-

```
SELECT
    p1.first_name, p1.last_name, ' and ',
    p2.first_name, p2.last_name, ' both come from ',
    p1.county
FROM
    persons AS p1
    JOIN persons AS p2
    ON p1.county = p2.county
        AND p1.person_id < p2.person_id;
```

- The < clause filters out duplicates and self pairs

Our Query Again

- Could rephrase as follows:

```
SELECT
    p1. first_name , p1.last_name,  ' and' ,
    p2. first_name , p2.last_name, ' both come from ',
    p1.county
FROM
    persons AS p1
    JOIN persons AS p2
    ON p1.county = p2.county
WHERE p1.person_id < p2.person_id;
```

- Bothe the ON-condition and WHERE-condition filter rows; so the < criterion can migrate into the latter

Who Likes Pizza and Beer?

- List all those individuals who like pizza *and* beer
- Idea: three-way join

  (`persons` × `favourite_foods` × `favourite_foods`)

  with ON condition that ensures:

  - three-way match on `person_id`
  - `favourite_foods.food` (first copy) should equal 'Pizza'
  - `favourite_foods.food` (second copy) should equal 'Beer'

The Query

```
SELECT first_name, last_name, f1.food, f2.food
FROM
    persons AS p
    JOIN favourite_foods AS f1
    JOIN favourite_foods AS f2
    ON
        p.person_id = f1.person_id
        AND f1.person_id = f2.person_id
        AND f1.food = ’Pizza’
        AND f2.food = ’Beer’;
```

Our Query

- Typical row from Cartesian product:

p_id		p_id	food	p_id	food
x	· · ·	y	Nutella	z	Crisps
from persons		from fav_foods$_1$		from fav_foods$_2$	

  No meaningful relationship among the fragments from the "joinee" tables

Our Query

- Typical row from Cartesian product:

p_id		p_id	food	p_id	food
x	···	y	Nutella	z	Crisps
from persons		from fav_foods$_1$		from fav_foods$_2$	

  No meaningful relationship among the fragments from the "joinee" tables

- Typical row product filtered by ON-condition:

p_id		p_id	food	p_id	food
x	···	x	Pizza	x	Beer
from persons		from fav_foods$_1$		from fav_foods$_2$	

  Note `person_id` (here abbrev. to `p_id`) must match up and `food` values must be Pizza and Beer respectively

Note

- SQL supports several types of join
- We have been using the INNER JOIN (the default)
- There are also OUTER, LEFT, RIGHT joins etc. which we may see later

Notes and Acknowledgements