Structured Query Language

Thomas Heinis

t.heinis@imperial.ac.uk
wp.doc.ic.ac.uk/theinis

Imperial College
London

SQL

- SQL (Structured Query Language) is the most prominent relational database language, used in more than 99% of database applications.
- SQL supports schema creation and modification; data insertion, retrieval, update and deletion; constraints, indexing of attributes, transactions, data access control (authorisation), plus lots more.
- Most SQL implementations also provide one or more procedural languages for writing procedures (*stored procedures*) that execute SQL statements within a RDBMS. Such procedures can be called directly by the user, by client programs, by other stored procedures or called automatically by the database when certain events happen (*triggers*), for example after a tuple is updated.
- We'll do a tour of SQL (well a subset of SQL)

Relational Algebra to SQL

Relation Algebra	SQL
$R \cup S$	R **union** S
$R \cap S$	R **intersect** S
$R - S$	R **except** S
$\prod_{attributes}(R)$	**select** attributes **from** R
$\sigma_{condition}(R)$	**from** R **where** condition
$R \times S$	R, S *or* R **cross join** S
$R \bowtie S$	R **natural join** S
$R \bowtie_{condition} S$	R **join** S **on** condition

Vocabulary

Relation Algebra	SQL	Comment
Relation	**Table**	Tables are persistent relations stored on disk.
Relational Expression	**Views**	Views are relations based on other relations. Views are not normally stored nor updateable, unless they're materialised.
Tuple	Row	Sometimes called a record
Attribute	Column	Sometimes called a field
Domain	Type	Types include char, int, float, date, time

SQL Gotcha's

Standards	Which SQL? Every SQL vendor supports a different subset of one of the SQL standards plus their own extensions. Moving a database from one vendor to another is non-trivial.
Duplicates	SQL is based on multi-sets(bags) not sets. Relations in SQL can have duplicate tuples. *Duplicate are best avoided*.
Nulls	Attributes do not need to have a value, they can be **null**. **null**s can be used to indicate a missing value, a value that is not known, a value that is private, etc. *Nulls are best avoided*.
Booleans	Booleans are based on three-valued logic (3VL). They can be **true**, **false** or **unknown**!

Types

Most SQL implementations support a wide range of types including:

int, smallint, real, double precision, float(*n*), numeric(*p*,*d*), decimal(p,d)	Most DBs support a variety of integer and floating point types. Ranges are implementation dependent. The usual arithmetic operators are available.
char, char(*n*), varchar(*n*), clob/text, ...	Strings can be fixed length (padded with spaces), varying length (upto *n*), or unlimited length clob/text. The concatenation operator is ||. *string* **like** '*pattern*' performs pattern matching where pattern can include _ for any character and % for zero or more chars, e.g. X **like** 'B%' matches any strings starting with B. Also **similar to** for regular expression matches.
bit(*n*), byte(*n*), blob	Bits, bytes, and binary large objects (blobs). Often used for audio, images, movies, files, etc.

Types

A few more types:

boolean	Booleans are based on three-valued logic (3VL). They can be **true**, **false** or **unknown**! See later for truth tables. Comparison operators include **between**, **not between**, **in**, **not in**. Examples: age **between** 45 **and** 49 for age>=45 and age<=49 name **not in** (‘Fred’, ‘Jim’, ‘Alice’)
date, time, timestamp	Dates and times are specified like: **date** ‘1994-02-25’, **time** ’12:45:02’, **timestamp** ‘1994-02-25 12:45:02’ SQL supports date and time expressions as well as timezones and intervals
...	Each RDBMS has a long list of additional types for *currency*, *xml*, *geo-spatial data*, *CAD data*, *multi-media*, etc. Some also support user-defined types.

Truth Table for 3-valued Logic

x	y	x and y	x or y	not x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	unknown			FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
unknown	TRUE			
unknown	unknown			
unknown	FALSE			
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	unknown			TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

You can complete the Truth table with the following mapping:

1 - TRUE
½ - unknown
0 - FALSE

x and y = min x, y
x or y = max x, y
not x = 1 – x

Truth Table for 3-valued Logic

x	y	x and y	x or y	not x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	unknown	unknown	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
unknown	TRUE	unknown	TRUE	unknown
unknown	unknown	unknown	unknown	unknown
unknown	FALSE	FALSE	unknown	unknown
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	unknown	FALSE	unknown	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

You can complete the Truth table with the following mapping:

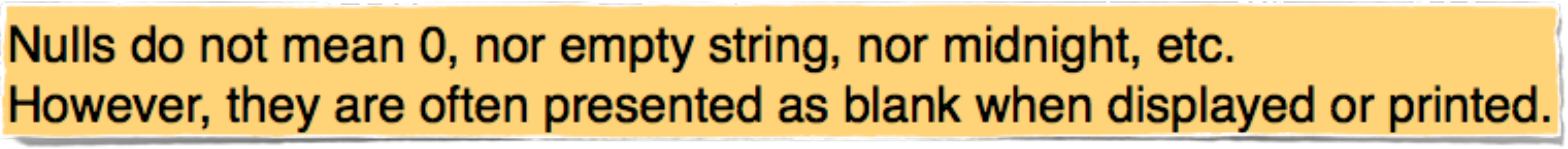
1 - TRUE
½ - unknown
0 - FALSE

x and y = min x, y
x or y = max x, y
not x = 1 – x

Nulls

SQL attributes can have the special value **null**. There are many interpretations for null, including:

Missing	There is some value, but we don’t know what it is at the moment, e.g. missing birthdate
Not applicable	No value makes sense, e.g. spouse’s name for an unmarried person
Withheld	There is a value, but we’re not entitled to record the value, e.g. an unlisted phone number.



Nulls do not mean 0, nor empty string, nor midnight, etc.
However, they are often presented as blank when displayed or printed.

Nulls

We need to understand the implications of **null**s on arithmetic and comparisons including for joins (see later).

Arithmetic	Any arithmetic that involves a **null** will result in a **null**. Note: In SQL, 0*y where y is **null** is **null**! y-y is also **null** if y is **null**!
Comparisons	Any comparison involving a **null** will result in **unknown**, e.g. x>y where y is **null** will result in **unknown**. **null** is not a constant value like **true** and can't be used in comparisons. To test if an attribute y is null, use y **is null**, or y **is not null** **null** will never match any other value (even **null** itself), unless we explicitly use **is null** or **is not null**.

Queries

Probably the most used and most complex statement in SQL is the **select** statement which is used to query (retrieve) data from a database.

select supports all the relational operators as well as sorting, grouping and aggregate functions. The relation produced by a **select** is normally returned to the user or client program, but can be used as a *subquery* in expressions.

Example:

movie(title, year, length, genre, studio, producer)

```
select title, length
from   movie
where  studio='fox' and year>1990
```

projected attributes

relation

selection condition

To return all attributes, use * for the projected attributes.

Renaming Attributes

We can rename attributes (and use expressions) in the projection part of a **select** with the **as** keyword. Renaming is useful if we have clashing attribute names that represent different things, or we want to carry out set operations on relations with differing attribute names. as can also be used to rename relations as we'll see.

Example: movie(title, year, length, genre, studio, producer)

```
select title as name, length/60 as hours
from   movie
where  studio='fox' and year>1990
```

For readability or to disambiguate attributes we can prefix the relation name, e.g:

```
select movie.title as name, movie.length/60 as hours
from   movie
where  movie.studio='fox' and movie.year>1990
```

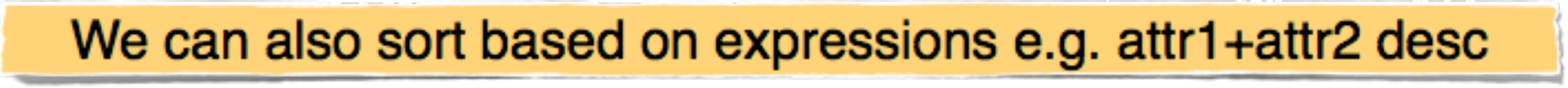
Sorting Results

In contrast to relational algebra, SQL's select statement can sort the tuples in the resulting relation. This is achieved by adding an order **by** clause at the end of the **select**.

Example: movie(title, year, length, genre, studio, producer)

```
select title, length
from   movie
where  studio=‘fox’ and year>1990
order by year desc, title asc
```

This will sort the resulting tuples first by year in descending order, then by title in ascending order. Note: we can use all the attributes of movie (e.g. year), not just those in the projection. So the order of evaluation is **from**, **where**, **order**, **select** (FWOS).



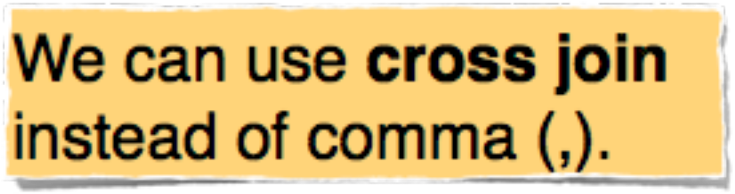
We can also sort based on expressions e.g. attr1+attr2 desc

Cartesian Product and Natural Join

The **from** clause is used to define a cartesian product or perform various joins.

Example: movie(title, year, length, genre, studio, producer)
casting(title, year, name)

```
select *
from movie , casting
```



We can use **cross join** instead of comma (,).

This is equivalent to movie × casting in relational algebra

while

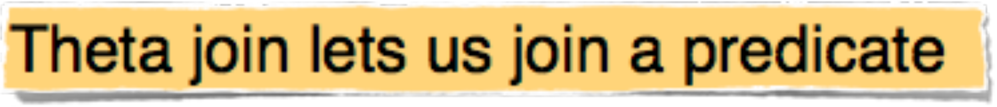
```
select *
from movie natural join casting
```

is equivalent to movie $\bowtie$ casting in relational algebra.

Theta Join

Theta join is performed with **join** and an **on** condition or a **using** attribute list.

Example: movie(title, year, length, genre, studio, producer)
casting(title, year, name)



Theta join lets us join a predicate

select title, year, name
from movie **join** casting **on** movie.producer=casting.name

This is equivalent to

$$\Pi_{\text{title, year, name}}(\sigma_{\text{movie.producer=casting.name}}(\text{movie} \times \text{casting}))$$

using can be used if we want to join on specific attributes, e.g.

select title
from movie **join** casting **using** (title, year)

which is the same as on movie.title=casting.title **and**
movie.year=casting.year

Renaming Relations

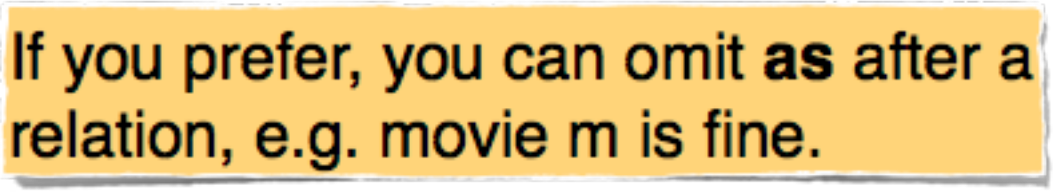
To form a query over two tuples from the same relation (self-join), we list the relation twice (i.e. perform a cartesian product on itself and rename one or both of the listed relations using the **as** keyword. Renamed relations are known as **correlation names**.

Example: movie(title, year, length, genre, studio, producer)
casting(title, year, name, address)

```
select casting1.name, casting2.name
from   casting as casting1 join casting as casting2
on         casting1.address = casting2.address and
           casting1.name < casting2.name
```

We can also use correlation names to give us a shorter name to use in other parts of the query:

```
select m.title, m.studio, a.name
from   movie m join casting c on m.producer=c.name
```



If you prefer, you can omit **as** after a relation, e.g. movie m is fine.

Multi-relation Joins

We can join as many relations as we like. The evaluation is carried out left to right unless we use parentheses. Note: in practice a query optimiser rewrites all queries for performance while maintaining the semantics of the query.

Example:

```
movie(title, year, length, genre, studio, producer)
casting(title, year, name)
studio(name, address, boss)
```

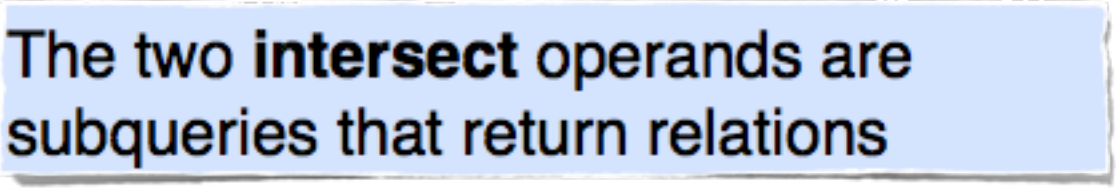
```
select casting.name, movie.producer, studio.boss
from   casting join movie using (title)
                join studio on movie.studio=studio.name
where  movie.year >= 1990
```

Union, Intersection, Difference

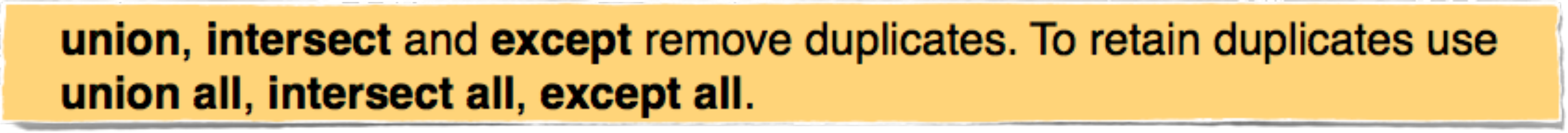
We can combine relations using the set operators union(∪), intersect(∩) and except(–). We typically use these operators on the relations generated by **select**s, which should be parenthesised.

Example: actor(name, address, gender, birthdate)
producer(name, address, networth)

```
(select name, address
 from   actor
 where  gender=‘F‘)
      intersect
(select name, address
 from   producer
 where  networth>=100000000)
```



The two **intersect** operands are subqueries that return relations



union, **intersect** and **except** remove duplicates. To retain duplicates use **union all**, **intersect all**, **except all**.

More Joins

leftrelation JOIN-OPERATOR **rightrelation**

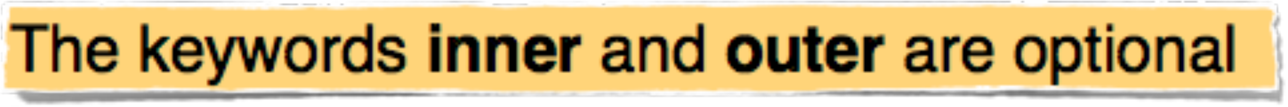
inner join returns tuples when there is at least one match in both relations. This corresponds to the join we've seen in relational algebra.

left outer join is like inner join but includes all tuples from the left relation, even if there are no matches in the right relation. Nulls are used for missing values.

right outer join is like inner join but includes all tuples from the right relation, even if there are no matches in the left relation. Nulls are used for missing values.

full outer join is like inner join but includes all umatched tuples from both relations. Nulls are used for missing values.

The joins above can be natural joins (joined by matching attributes) or theta joins (joined by a condition).



The keywords **inner** and **outer** are optional

Natural Outer Joins

Although natural and theta joins are often what's required, there are occasions when we'd like to retain tuples that don't match, outerjoins give us this capability.

Examples:

L

a	b	c
1	2	3
4	5	6
7	8	9

R

b	**c**	**d**
2	3	A
2	3	B
6	8	C

L natural left join R

a	b	c	d
1	2	3	A
1	2	3	B
4	5	6	null
7	8	9	null

L natural right join R

a	b	c	d
1	2	3	A
1	2	3	B
null	6	8	C

nulls are used for missing values

L natural full join R

a	b	c	d
1	2	3	A
1	2	3	B
4	5	6	null
7	8	9	null
null	6	8	C

Theta Outer Joins

We can also perform theta outerjoins using **left outer join**, **right outer join**, or **full outer join** along with an **on** condition.

Example: movie(title, year, length, genre, studio, producer)
casting(title, year, name)

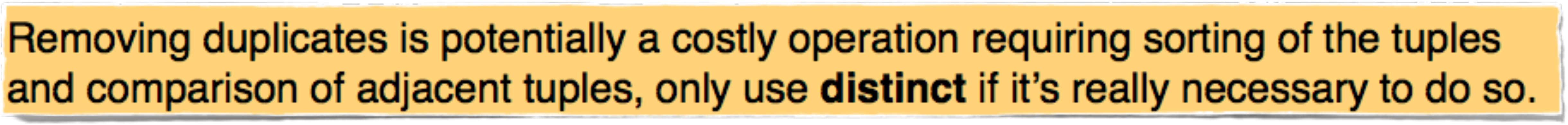
select title, year, name
from movie **left outer join** casting **on**
movie.producer=casting.name **and** movie.year=casting.year

Eliminating Duplicates

Unlike relational algebra, SQL queries can potentially produce duplicate tuples. We can eliminate them by adding the keyword **distinct** after the keyword **select**. Recall that **union**, **intersect** and **except** eliminate duplicates unless **all** is used.

Example: movie(title, year, length, genre, studio, producer)
casting(title, year, name)

```
select distinct title, year, name
from movie left outer join casting on
    movie.producer=casting.name and movie.year=casting.year
```



Removing duplicates is potentially a costly operation requiring sorting of the tuples and comparison of adjacent tuples, only use **distinct** if it's really necessary to do so.

Aggregation Functions

The aggregate functions **sum**, **avg**, **min**, **max** and **count** can be used in a projection list to calculate a single value, either from the whole resulting relation or from a part of it - see grouping later). The parameter is typically an *attribute* but can be an *expression*. Fortunately, **null**s are excluded from these calculations.

count(**distinct** *attribute*) counts the number of distinct values of the attribute. **count**(*) is an aggregate function that counts the number of tuples in a relation or group (including nulls)

Example:

```
select    count(*)   as professors,
          sum(salary) as totalsalary,
          avg(salary) as averagesalary,
          min(age)  as youngest,
          max(age) as oldest
from  employee
where position = ‘Professor‘
```

Note: this query will produce a relation with a single tuple with 5 attributes.

Grouping

The **select** statement can group the tuples in a resulting relation. This is achieved by providing a list of grouping attributes in a **group by** clause. If aggregate functions are used in the projection list they are applied to each group.

Example:

```
select          department,
                count(*)   as professors,
                sum(salary) as totalsalary,
                avg(salary) as averagesalary,
                min(age)  as youngest,
                max(age) as oldest
        from  employee
        where position = 'Professor'
        group by department
        order by totalsalary desc
```

Any non-aggregates used in the projection must be included in the **group by** list.

This query will produce a relation with one tuple for each department. The results are sorted in descending order by totalsalary.

Example

```
select department,
            count(*)   as professors,
            sum(salary) as totalsalary,
            avg(salary) as averagesalary,
            min(age)  as youngest,
            max(age)  as oldest
from  employee
where position = ‘Professor’
group by department
order by totalsalary desc
```

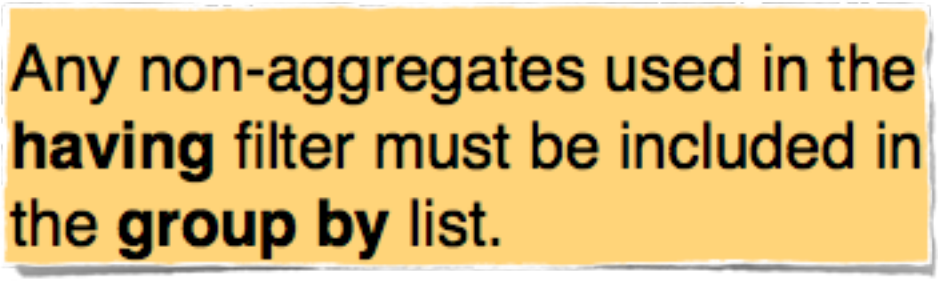
department	professors	totalsalary	averagesalary	youngest	oldest
physics	30	3000000	100000	35	68
computing	25	2000000	80000	40	65
...					
materials	4	360000	90000	50	62

Filtering groups by aggregate functions

We can filter groups using a predicate with aggregate functions that is applied to each group by adding a **having** clause after the **group by** clause.

Example:

```
select    department,
              count(*)   as professors,
              sum(salary) as totalsalary,
              avg(salary) as averagesalary,
              min(age)  as youngest,
              max(age) as oldest
          from  employee
          where position = ‘Professor’
          group by department
          having count(*)>=10
          order by totalsalary desc
```



Any non-aggregates used in the **having** filter must be included in the **group by** list.

This query will produce a relation with one tuple for each department that has at least 10 professors. The results are sorted in descending order by totalsalary.

Example

```
select department,
      count(*)    as professors,
      sum(salary) as totalsalary,
      avg(salary) as averagesalary,
      min(age)    as youngest,
      max(age)    as oldest
from  employee
where position = ‘Professor‘
group by department
having count(*)>=10
order by totalsalary desc
```

department	professors	totalsalary	averagesalary	youngest	oldest
physics	30	3000000	100000	35	68
computing	25	2000000	80000	40	65
...					
~~materials~~	~~4~~	~~360000~~	~~90000~~	~~50~~	~~62~~

Subqueries

One of the most powerful features of **select**s is that they can be used as subqueries in expressions by enclosing them in parentheses i.e. (*subquery*). SQL supports scalar, set and relations subqueries.

Scalar subquery A subquery that produces a single value. Typically a select with an aggregate function.

Set subquery A subquery that produces a set of distinct values (a single column). Typically used for (i) set membership using operators **in** or **not in**, or (ii) set comparisons using operators **some**(**any**) or **all.**

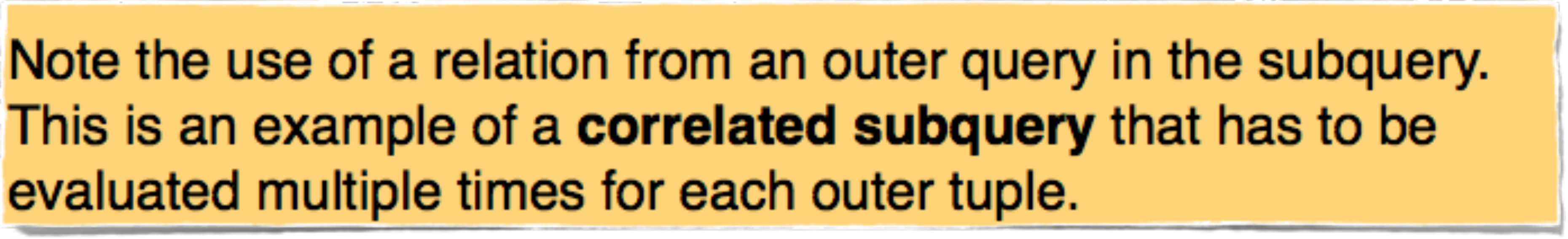
Relation subquery A subquery that produces a relation. Typically used as an operand of (i) products, joins, **union**s, **intersect**s, **except**s, (iii) operators **exists** or **not exists** to test if a relation is empty or not, (iv) operators **not unique** or **unique** to test if a relation has duplicates or not.

Scalar Subquery

A select that produces a single value. Scalar subqueries can be used in any expression, e.g. in projection lists, in **where** and **having** clauses. Are often selects with a single aggregate function.

Example: movie(title, year, length, genre, studio, producer)
casting(title, year, name)

```
select title,
        (select count(name)
         from   casting
         where  casting.title=movie.title) as numactors
from   movie;
```



Note the use of a relation from an outer query in the subquery. This is an example of a **correlated subquery** that has to be evaluated multiple times for each outer tuple.

Join Instead of Subquery

Joins can often be used instead of subqueries. Clearer to use joins when possible.

Example: movie(title, year, length, genre, studio, producer)
casting(title, year, name)

```
select title, year
        (select count(name)
         from casting
         where casting.title=movie.title) as numactors
from   movie;
```

Alternative with joins:

```
select title, year
        count(name) as numactors
from movie join casting using (title)
group by title, year
```

Set Membership Subqueries

Subqueries that produce a set of values can be used to test if a value is a member of the set by using the in or not in operators.

Example: movie(title, year, length, genre, studio, producer)
casting(title, year, name)
studio(name, address, boss)

```
select title
from   movie
where  studio in (select name from studio
   where address like ‘C%’)
```

We can extend the approach to tuple values enclosed in parentheses:

```
select name
from   casting
where  (title, year) not in
          (select title, year from movie where genre=‘sf’)
```

Join instead of a subquery

Example1:

```
select title from   movie
where  studio in (select name from studio
                  where address like ‘C%’)
```

Alternative:

```
select title
from   movie join studio on studio.name=movie.studio
where  studio.address=‘C%’
```

Example2:

```
select name
from   casting
where  (title, year) not in
                  (select title, year from movie where genre=‘sf’)
```

Alternative:

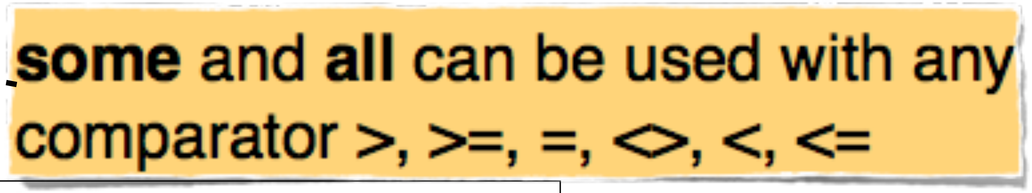
```
select name
from   casting join movie using (title, year)
where  genre<>‘sf’
```

Set Comparison Subqueries

We can use subqueries to compare a value against *some* or *all* values returned by a subquery, using the **some (=any)** and **all** functions respectively.

Example: movie(title, year, length, genre, studio, producer)

```
select title
from   movie m1
where  year <  some(select year from movie m2
                             where m2.title=m1.title)
```



some and **all** can be used with any comparator >, >=, =, <>, <, <=

All requires:

```
select name
from   employee
where  salary <> all(select salary from employee
                             where position='Professor')
```

Set Comparison Subquery Examples

Example SOME:

```
SELECT ProductName FROM Product WHERE Id = SOME
        (SELECT ProductId FROM OrderItem  WHERE Quantity = 1)
```

Example ALL:

```
SELECT DISTINCT FirstName + ' ' + LastName as CustomerName FROM
Customer, Order
WHERE Customer.Id = Order.CustomerId AND TotalAmount > ALL
        (SELECT AVG(TotalAmount) FROM [Order]  GROUP BY CustomerId)
```

Relation subqueries

The **exists** and **not exists** functions with a subquery argument can be used to test whether a relation is empty (has no tuples) or not.

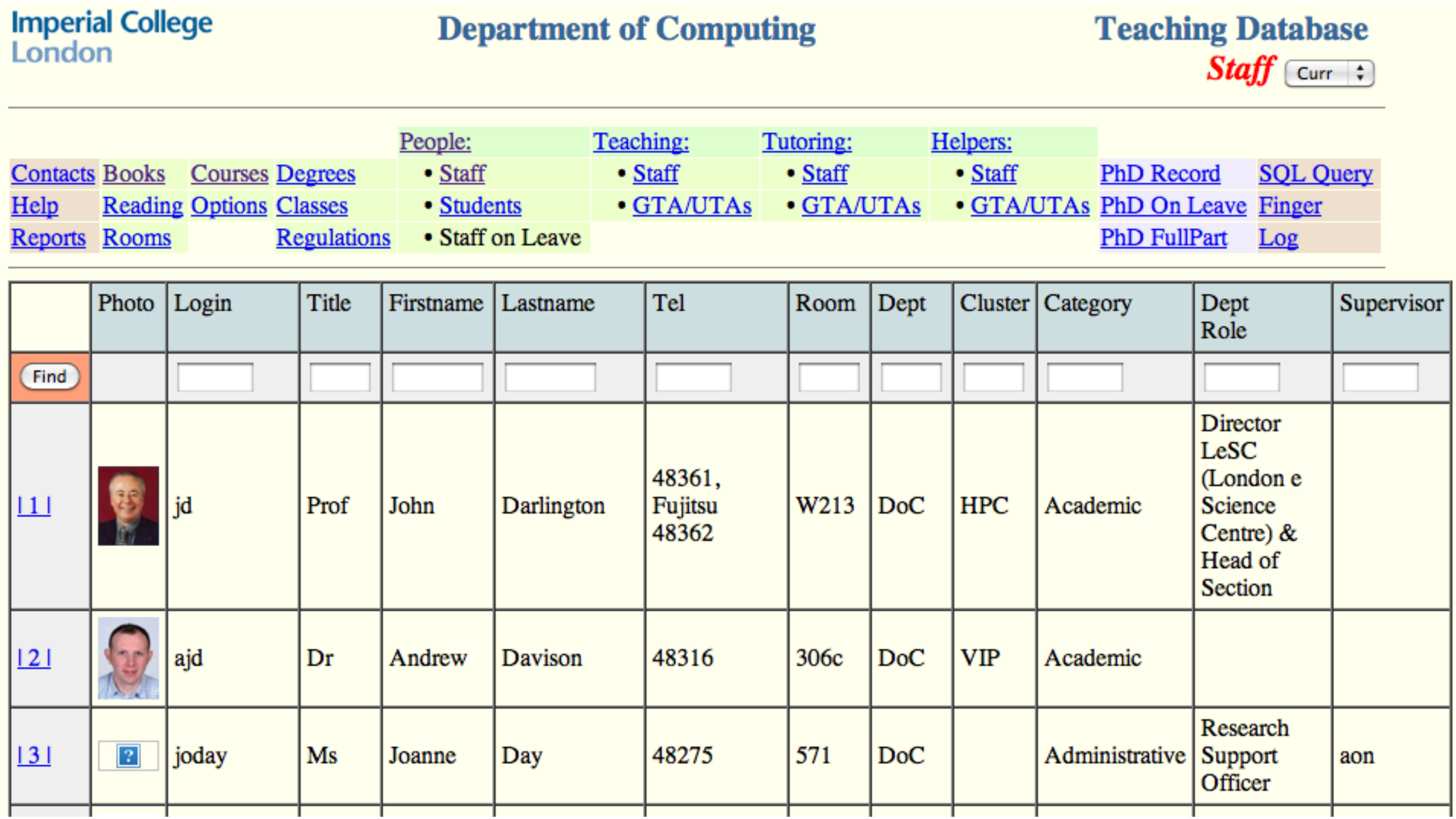
The **not unique** function can be used to test whether a relation has duplicates, or hasn’t duplicates with **unique**.

Example:

```
movie(title, year, length, genre, studio, producer)
casting(title, year, name)
studio(name, address, boss)

select title
from   movie m1
where  not exists(select * from movie m2
                         where m2.title=m1.title and m2.year<>m1.year
)
```

DoC Teaching Database



Imperial College London

Department of Computing

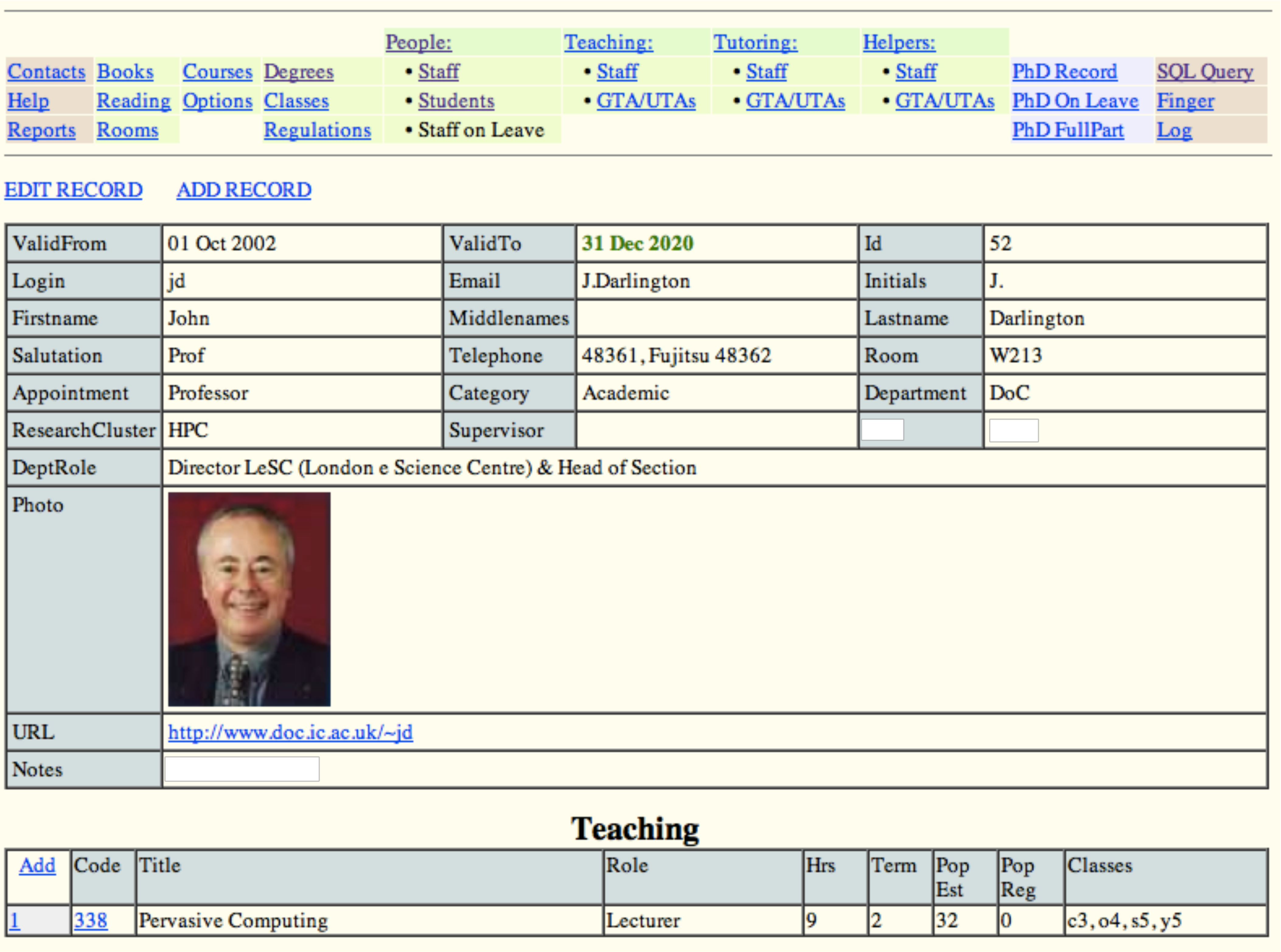
Teaching Database

Staff Curr

				People:	Teaching:	Tutoring:	Helpers:		
Contacts	Books	Courses	Degrees	• Staff	• Staff	• Staff	• Staff	PhD Record	SQL Query
Help	Reading	Options	Classes	• Students	• GTA/UTAs	• GTA/UTAs	• GTA/UTAs	PhD On Leave	Finger
Reports	Rooms		Regulations	• Staff on Leave				PhD FullPart	Log

	Photo	Login	Title	Firstname	Lastname	Tel	Room	Dept	Cluster	Category	Dept Role	Supervisor
Find												
| 1 |		jd	Prof	John	Darlington	48361, Fujitsu 48362	W213	DoC	HPC	Academic	Director LeSC (London e Science Centre) & Head of Section	
| 2 |		ajd	Dr	Andrew	Davison	48316	306c	DoC	VIP	Academic		
| 3 |	?	joday	Ms	Joanne	Day	48275	571	DoC		Administrative	Research Support Officer	aon

DOC Teaching Database



				People:	Teaching:	Tutoring:	Helpers:		
Contacts	Books	Courses	Degrees	• Staff	• Staff	• Staff	• Staff	PhD Record	SQL Query
Help	Reading	Options	Classes	• Students	• GTA/UTAs	• GTA/UTAs	• GTA/UTAs	PhD On Leave	Finger
Reports	Rooms		Regulations	• Staff on Leave				PhD FullPart	Log

EDIT RECORD ADD RECORD

ValidFrom	01 Oct 2002	ValidTo	**31 Dec 2020**	Id	52
Login	jd	Email	J.Darlington	Initials	J.
Firstname	John	Middlenames		Lastname	Darlington
Salutation	Prof	Telephone	48361, Fujitsu 48362	Room	W213
Appointment	Professor	Category	Academic	Department	DoC
ResearchCluster	HPC	Supervisor			
DeptRole	Director LeSC (London e Science Centre) & Head of Section				
Photo					
URL	http://www.doc.ic.ac.uk/~jd				
Notes					

Teaching

Add	Code	Title	Role	Hrs	Term	Pop Est	Pop Reg	Classes
1	338	Pervasive Computing	Lecturer	9	2	32	0	c3, o4, s5, y5

DOC Teaching Database

Common Attributes

id, opened, openedby, updated, updatedby, validfrom, validto

Main Relations

staff(login, email, lastname, firstname, telephone, room, deptrole, department)
student(login, email, lastname, status, entryyear, externaldept)
course(code, title, syllabus, term, classes, popestimate)
class(***degreeid***, yr, degree, degreeyr, major, majoryr, letter, letteryr)
degree(title, code, major, grp, letter, years)
book(code, title, authors, publisher)

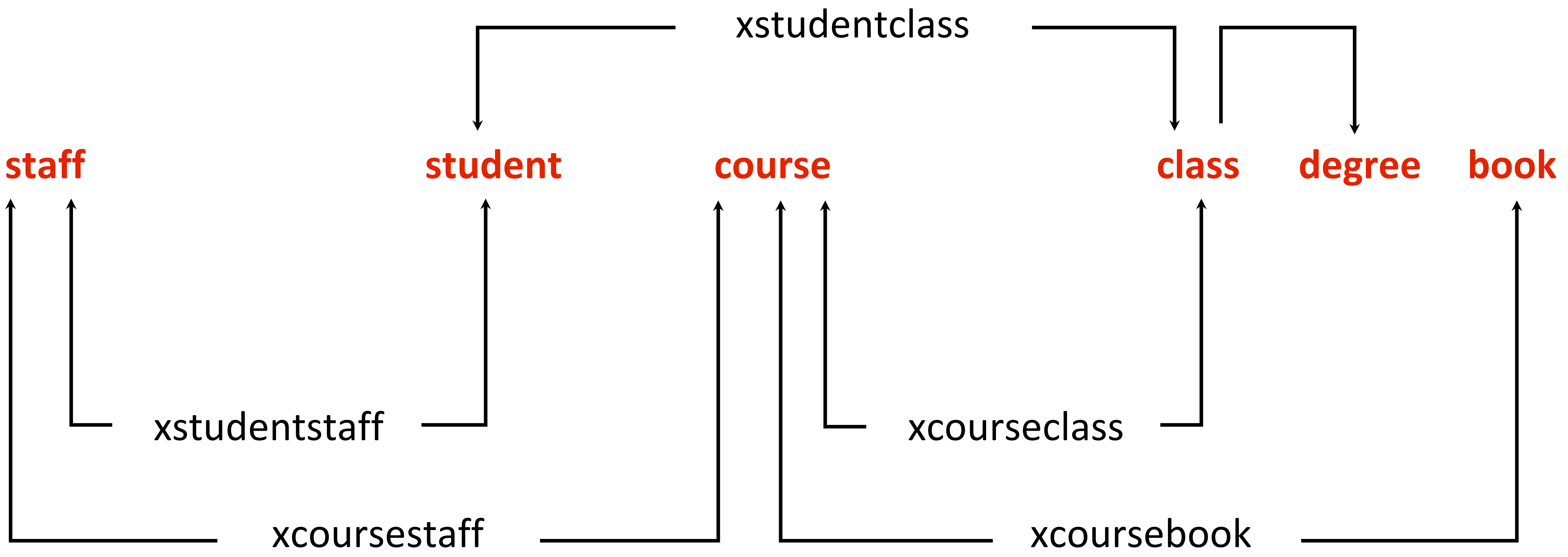
Many-to-Many Joining Relations

xcourseclass(***courseid***, ***classid***, required, examcode)
xcoursebook(***courseid***, ***bookid***, rating)
xcoursestaff(***courseid***, ***staffid***, staffhours, role, term)
xstudentclass(***studentid***, ***classid***)
xstudentstaff(***studentid***, ***staffid***, role, grp, projecttitle)

Q. List all staff who do not have a College or Department email address, sort results by lastname

Q. List all staff with the same lastname, show names of staff and their namesake(s)

DoC Teaching Database



Each relation has several temporal views (*named relational expressions*), e.g.:

coursecurr - courses for current year
course0910 - courses for academic year 2009-2010, similar for 0809 etc.

Examples 1

Q. List all staff who do not have a College or Department email address, sort results by lastname.

Q. List all staff with the same lastname, show names of staff and their namesake(s)

Solutions 1

Q. List all staff who do not have a College or Department email address, sort by lastname.

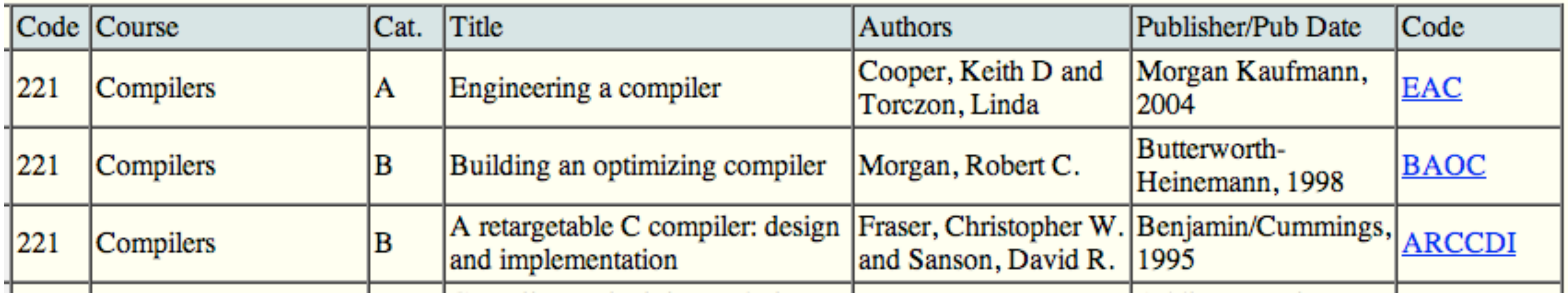
```
select id, lastname, email
from   staffcurr
where  not(email like '%imperial.ac.uk' or email like '%doc.ic.ac.uk')
order by lastname
```

Q. List all staff with the same lastname, show names of staff and their namesake(s)

```
select s1.id, s1.firstname, s1.lastname,
       s2.id, s2.firstname, s2.lastname
from   staffcurr s1 join  staffcurr s2 on s1.lastname=s2.lastname and
                                    s1.id < s2.id
order by s1.lastname
```

Examples 2

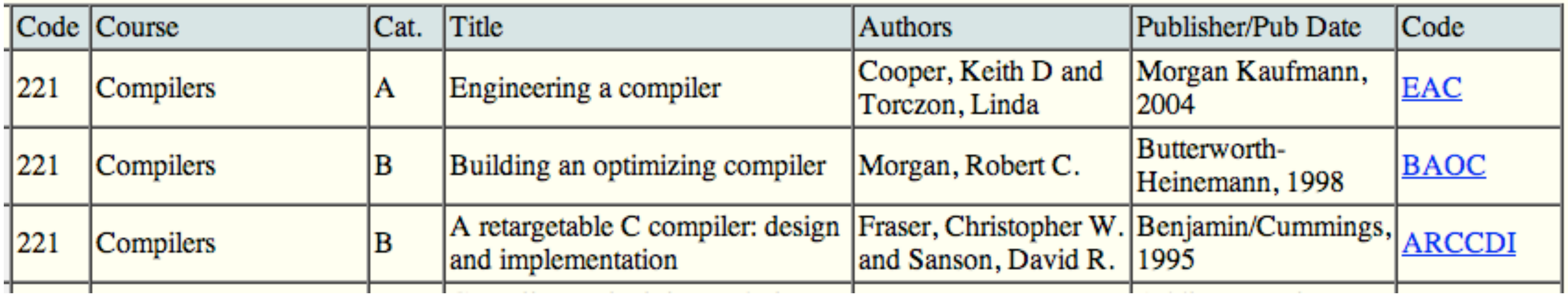
Q. List all books recommended for courses taught by Prof Kelly, similar to:



Code	Course	Cat.	Title	Authors	Publisher/Pub Date	Code
221	Compilers	A	Engineering a compiler	Cooper, Keith D and Torczon, Linda	Morgan Kaufmann, 2004	EAC
221	Compilers	B	Building an optimizing compiler	Morgan, Robert C.	Butterworth-Heinemann, 1998	BAOC
221	Compilers	B	A retargetable C compiler: design and implementation	Fraser, Christopher W. and Sanson, David R.	Benjamin/Cummings, 1995	ARCCDI

Solution 2

Q. List all books recommended for courses taught by Prof Kelly, similar to:

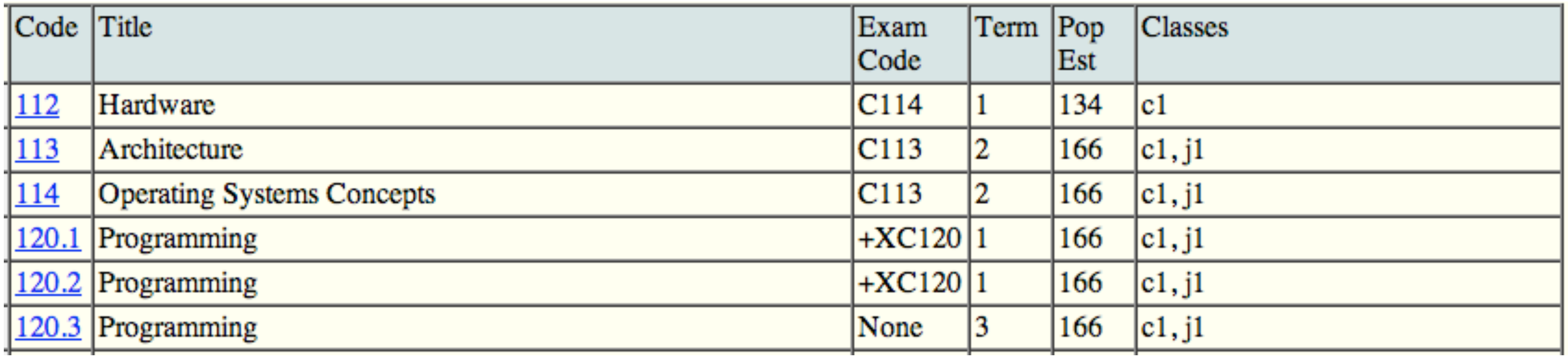


Code	Course	Cat.	Title	Authors	Publisher/Pub Date	Code
221	Compilers	A	Engineering a compiler	Cooper, Keith D and Torczon, Linda	Morgan Kaufmann, 2004	EAC
221	Compilers	B	Building an optimizing compiler	Morgan, Robert C.	Butterworth-Heinemann, 1998	BAOC
221	Compilers	B	A retargetable C compiler: design and implementation	Fraser, Christopher W. and Sanson, David R.	Benjamin/Cummings, 1995	ARCCDI

```
select c.code, c.title, xb.rating, b.title, b.authors, b.publisher,
     b.code
from  staffcurr s join xcoursestaffcurr xc on s.id=xc.staffid
           join coursecurr c on xc.courseid=c.id
           join xcoursebookcurr xb on c.id=xb.courseid
           join bookcurr b on xb.bookid=b.id
where s.lastname='Kelly'
order by c.code, xb.rating
```

Examples 3

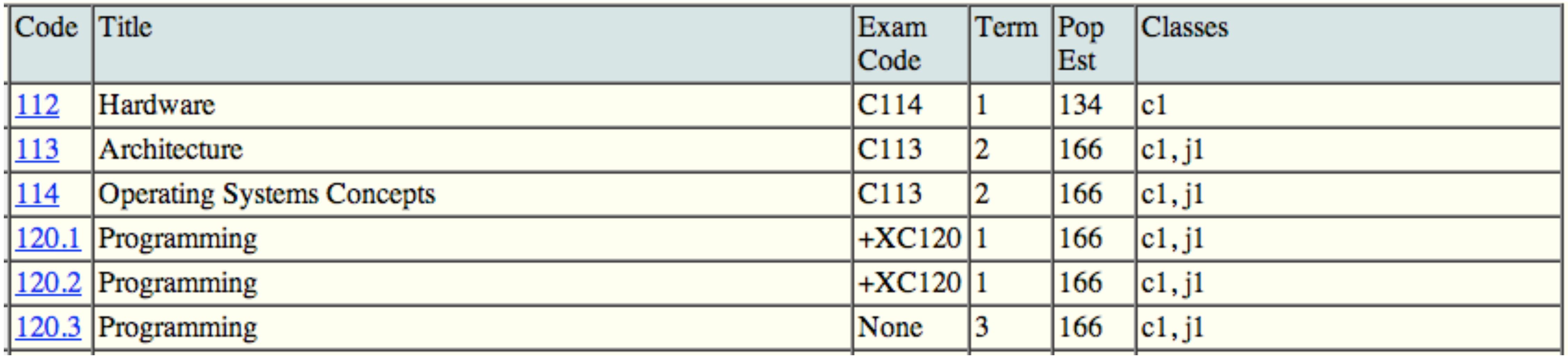
Q List courses being taken by student with login rf611, similar to



Code	Title	Exam Code	Term	Pop Est	Classes
112	Hardware	C114	1	134	c1
113	Architecture	C113	2	166	c1, j1
114	Operating Systems Concepts	C113	2	166	c1, j1
120.1	Programming	+XC120	1	166	c1, j1
120.2	Programming	+XC120	1	166	c1, j1
120.3	Programming	None	3	166	c1, j1

Solution 3a

Q List courses being taken by student with login rf6111, similar to

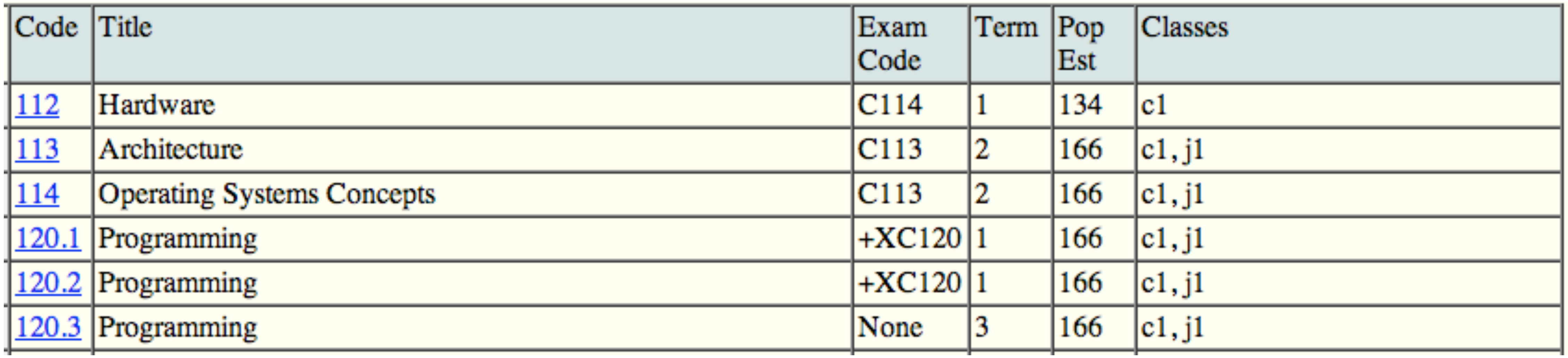


Code	Title	Exam Code	Term	Pop Est	Classes
112	Hardware	C114	1	134	c1
113	Architecture	C113	2	166	c1, j1
114	Operating Systems Concepts	C113	2	166	c1, j1
120.1	Programming	+XC120	1	166	c1, j1
120.2	Programming	+XC120	1	166	c1, j1
120.3	Programming	None	3	166	c1, j1

```
select c.code, c.title, xc.examcode, c.term, c.popestimate, c.classes
from   studentcurr s join xstudentclasscurr xa on s.id=xa.studentid
              join classcurr ca on xa.classid=ca.id
              join xcourseclasscurr xc on ca.id=xc.classid
              join coursecurr c on xc.courseid=c.id
where  s.login='rf611'
order by c.code
```

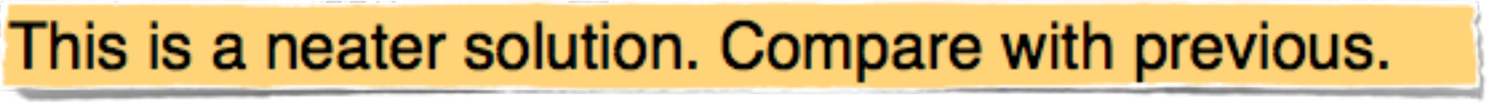
Solution 3b

Q List courses being taken by student with login rf611, similar to



Code	Title	Exam Code	Term	Pop Est	Classes
112	Hardware	C114	1	134	c1
113	Architecture	C113	2	166	c1, j1
114	Operating Systems Concepts	C113	2	166	c1, j1
120.1	Programming	+XC120	1	166	c1, j1
120.2	Programming	+XC120	1	166	c1, j1
120.3	Programming	None	3	166	c1, j1

```
select c.code, c.title, xc.examcode, c.term, c.popestimate, c.classes
from   studentcurr s join xstudentclasscurr xa on s.id=xa.studentid
              join xcourseclasscurr xc on xa.classid=xc.classid
              join coursecurr c on xc.courseid=c.id
where  s.login='rf611'
order by c.code
```



This is a neater solution. Compare with previous.

Examples 4

Q. List all PPT tutors and their PPT tutees (role=PPT) order staff and students:
Tutor lastname, tutor firstname, Tutee lastname, firstname suitably sorted.

Q. List all PPT tutors and how many PPT tutees they have, suitably sorted.

Solutions 4

Q. List all PPT tutors and their PPT tutees (role=PPT) order staff and students:
Tutor lastname, tutor firstname, Tutee lastname, firstname suitably sorted.

```
select s.id, s.lastname, s.firstname, t.id, t.lastname, t.firstname
from   staffcurr s join xstudentstaffcurr x on s.id=x.staffid
            join studentcurr t on t.id=x.studentid
where  x.role='PPT'
order by s.lastname, s.firstname, t.lastname, t.firstname
```

Q. List all PPT tutors and how many PPT tutees they have, suitably sorted.

```
select s.id, s.lastname, s.firstname, count(t.id)
from   staffcurr s join xstudentstaffcurr x on s.id=x.staffid
            join studentcurr t on t.id=x.studentid
where  x.role='PPT'
group by s.id, s.lastname, s.firstname
order by s.lastname, s.firstname
```

Examples 5

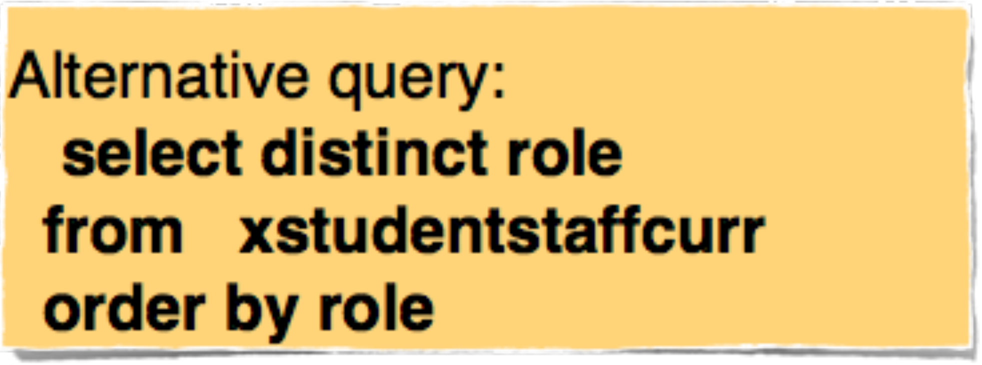
Q. List all tutoring roles

Q. List all tutoring roles and how many tutors there are for each role

Solutions 5

Q. List all tutoring roles

```
select role
from   xstudentstaffcurr
group by role
order by role
```



Alternative query:

```
  select distinct role
 from   xstudentstaffcurr
 order by role
```

Q. List all tutoring roles and how many tutors there are for each role

```
select role, count(distinct staffid) as tutors
from   xstudentstaffcurr
group by role
order by role
```