Databases

Structured Query Language (SQL) II

Dr Konrad Dabrowski
konrad.dabrowski@durham.ac.uk

Online Office Hour:
Mondays 13:30–14:30
See Duo for the Zoom link

SQL syntax

Basic syntax of SQL queries:

SELECT [ALL | DISTINCT] *column1* [,*column2, column3, …*]
FROM *table1* [,*table2, table3, …*]
[WHERE "conditions"]

- The "conditions" in the WHERE clause can be:
  - a comparison predicate (e.g. salary > 10000)
  - a range predicate (e.g. salary BETWEEN 10000 AND 30000)
  - a set membership predicate (e.g. position IN ('Manager', 'Worker'))
  - a pattern matching predicate (e.g. address LIKE '%Glasgow%')
  - combinations of the above with AND and/or OR
- but it can also be:
  - the result of another (independent) query (called a subquery)

SQL syntax

- Three types of a subquery:

1. a single-value (scalar) subquery (single column & single row)

```
SELECT  COUNT(*)  AS  myCount
FROM  PropertyForRent
WHERE  rent > 350
```

myCount
5

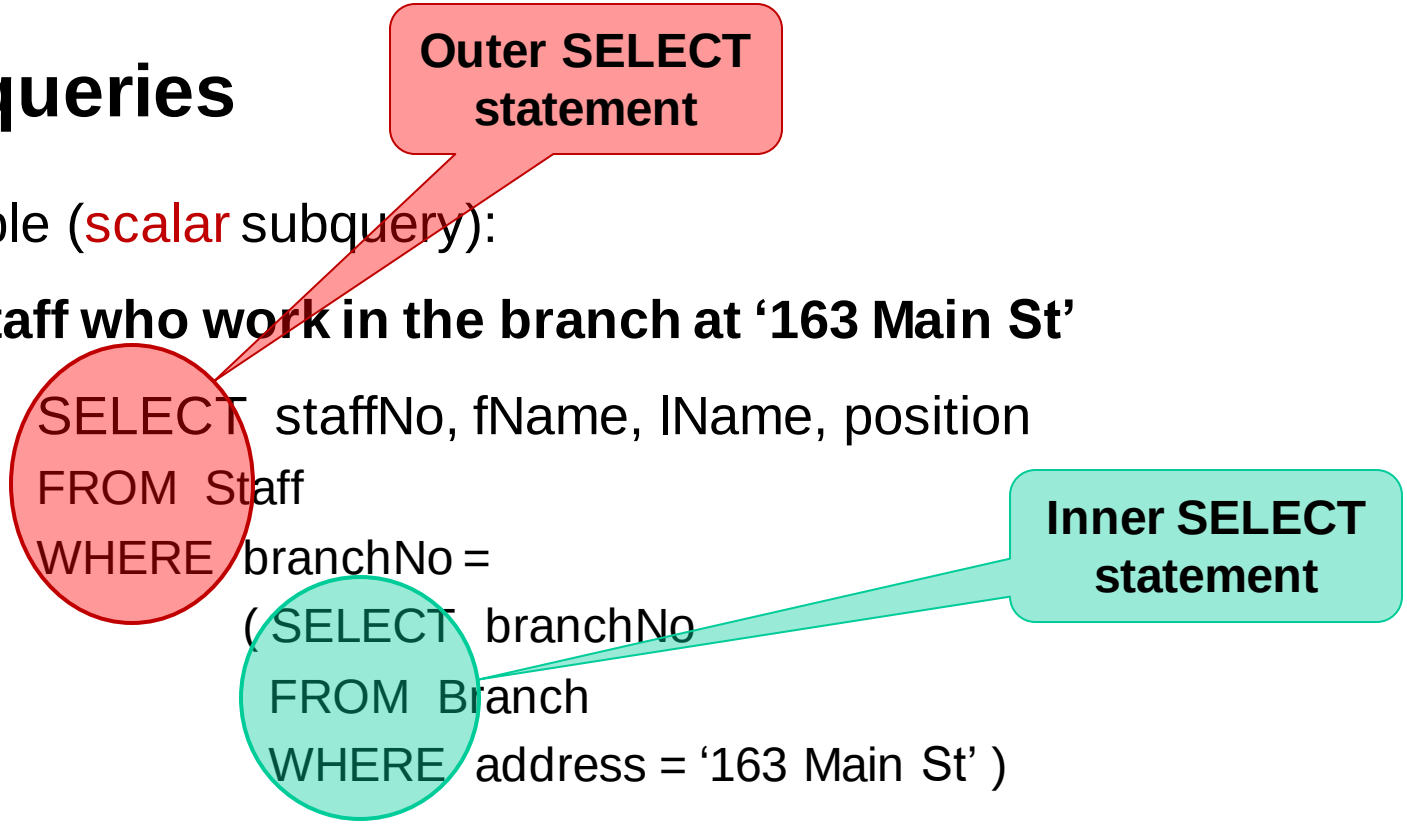
2. a multiple-value subquery (one column & multiple rows)

```
SELECT  staffNo
FROM  Staff
WHERE  position = 'Manager'
```

3. a table subquery (multiple columns / rows)

```
SELECT  clientNo, viewDate
FROM  Viewing
WHERE  propertyNo='PG4'  AND  comment  IS NULL
```

Subqueries



- The inner SELECT:
  - finds the branch number of the branch at 163 Main St.
  - only one such branch (with branchNo = 'B003') $\Longrightarrow$ scalar subquery
- The outer SELECT is equivalent with:

```
SELECT staffNo, fName, lName, position
FROM Staff
WHERE branchNo = 'B003'
```

Subqueries

Example (using an aggregate function):

List all staff whose salary is greater than the average salary, and show by how much

- If we know that the average salary is 17000, then:

```
SELECT  staffNo, fName, lName, position,
        salary – 17000 As SalDiff
FROM  Staff
WHERE  salary > 17000;
```

- We cannot write "WHERE salary > AVG(salary)"
- Instead, we use a subquery:

```
SELECT  staffNo, fName, lName, position,
        salary – (SELECT AVG(salary) FROM Staff) As SalDiff
FROM  Staff
WHERE  salary > (SELECT AVG(salary) FROM Staff);
```

Nested Queries

Example (scalar subquery and multi-value subquery) – use of the operator IN:

List the properties that are handled by staff who work in the branch with address '163 Main St'

```
SELECT  propertyNo, street, city, postcode, type, rooms, rent
FROM  PropertyForRent
WHERE  staffNo IN (SELECT  staffNo
                   FROM  Staff
                   WHERE  branchNo =
                          (SELECT  branchNo
                           FROM  branch
                           WHERE  street = '163 Main St'))
```

multi-value subquery

scalar subquery

- From the innermost query outwards:
  - the first query selects the branch number of the branch at 163 Main St
  - the second selects the staff working at this branch
  - many staff $\Longrightarrow$ in the outmost query we use IN ("=" is not possible)

Nested Queries

- In multi-value subqueries:
  - use of the operator ANY (or SOME) before the subquery

    ⟹ the WHERE-condition is true if it is satisfied by at least one value returned by the subquery

Find all staff whose salary is larger than the salary of at least one member of staff at branch B003

```
SELECT staffNo, fName, lName, position, salary
FROM Staff
WHERE salary > SOME (SELECT salary
                     FROM Staff
                     WHERE branchNo = 'B003');
```

many salary values

- An alternative would be:

```
WHERE salary > (SELECT MIN(salary)
                FROM Staff
                WHERE branchNo = 'B003')
```

Nested Queries

- In multi-value subqueries:
  - use of the operator ALL before the subquery

    ⟹ the WHERE-condition is true if it is satisfied by all values returned by the subquery

Find all staff whose salary is larger than the salary of every member of staff at branch B003

```
SELECT staffNo, fName, lName, position, salary
FROM Staff
WHERE salary > ALL (SELECT salary
                    FROM Staff
                    WHERE branchNo = 'B003');
```

many salary values

- An alternative would be:

```
WHERE salary > (SELECT MAX(salary)
                FROM Staff
                WHERE branchNo = 'B003')
```

Multi-table queries

- All examples so far have a major limitation:
  - the whole information belongs to a single table

- We can extend queries to multiple tables:
  - either with subqueries that query different tables:

List all Durham-staff with a salary greater than the average London-salary

```
SELECT staffNo, fName, lName, position
FROM DurhamStaff
WHERE salary > (SELECT AVG(salary) FROM LondonStaff)
```

  - or by using a join operation:
    - link data from two (or more) tables together (in a single query)
    - include more than one table in the FROM clause
    - separate these tables with a comma (,)

Joins

- In joins, usually:
  - include a WHERE clause to specify the joined columns
  - we keep in the search only those rows, which have the same values on the specified columns
  - for clarity, in the SELECT clause, we can put the table name before the column name (e.g. Staff.staffNo)
  - also possible to use an alias for a table in the FROM clause (useful for distinguishing column names in case of ambiguity)
  - alias is separated from table name with a space
- Usually the syntax is:

```
SELECT  “list-of-columns”
FROM  table1, table2, …
WHERE  “search-conditions”
```

Joins

table name before the column name

List the details of all clients who have viewed a property, along with any comment supplied

```
SELECT  Client.clientNo, fName, lName, propertyNo, comment
FROM  Client, Viewing
WHERE  Client.clientNo = Viewing.clientNo
```

Join of two tables

matching columns

- Using an Alias:

```
SELECT  c.clientNo, c.fName, c.lName, v.propertyNo, v.comment
FROM  Client c, Viewing v
WHERE  c.clientNo = v.clientNo
```

alias c for Client, v for Viewing

- This type of Join is also known as a natural inner join:
  - keeps the rows that coincide in the specified columns (in the WHERE clause)
  - ignores all rows that do not meet the join conditions
  - the most common type of join

Joins

List the details of all clients who have viewed a property, along with any comment supplied

```
SELECT  c.clientNo, c.fName, c.lName, v.propertyNo, v.comment
FROM  Client c, Viewing v
WHERE  c.clientNo = v.clientNo
```

Viewing table

clientNo	propertyNo	viewDate	comment
CR56	PA14	24-May-01	Too small
CR76	PG4	20-Apr-01	Too remote
CR56	PG4	26-May-01	
CR62	PA14	14-May-01	Too dirty
CR56	PG36	28-Apr-01	

Client table

clientNo	fName	lName
CR56	Aline	Stewart
CR62	Mary	Tragear
CR76	John	John Kay

⇒ **joined table:**

clientNo	fName	lName	propertyNo	comment
CR56	Aline	Stewart	PG36	
CR56	Aline	Stewart	PA14	too small
CR56	Aline	Stewart	PG4	
CR62	Mary	Tregear	PA14	no dining room
CR76	John	Kay	PG4	too remote

Sorting a Join

For each branch, list numbers and names of staff who manage properties, and the properties they manage

```
SELECT  s.branchNo, s.staffNo, s.fName, s.lName, p.propertyNo
FROM  Staff s, PropertyForRent p
WHERE  s.staffNo = p.staffNo
ORDER BY  s.branchNo, s.staffNo, p.propertyNo;
```

branchNo	staffNo	fName	lName	propertyNo
B003	SG14	David	Ford	PG16
B003	SG37	Ann	Beech	PG21
B003	SG37	Ann	Beech	PG36
B005	SL41	Julie	Lee	PL94
B007	SA9	Mary	Howe	PA14

Three-table Join

For each branch, list staff who manage properties, including the city in which branch is located and the properties they manage

```
SELECT   b.branchNo, b.city, s.staffNo, s.fName, s.lName, p.propertyNo
FROM   Branch b, Staff s, PropertyForRent p
WHERE   b.branchNo = s.branchNo  AND  s.staffNo = p.staffNo
ORDER BY  b.branchNo, s.staffNo, p.propertyNo;
```

branchNo	city	staffNo	fName	lName	propertyNo
B003	Glasgow	SG14	David	Ford	PG16
B003	Glasgow	SG37	Ann	Beech	PG21
B003	Glasgow	SG37	Ann	Beech	PG36
B005	London	SL41	Julie	Lee	PL94
B007	Aberdeen	SA9	Mary	Howe	PA14

- Alternative formulation of the FROM and WHERE clauses:

```
FROM  (Branch b JOIN Staff s USING branchNo)
          JOIN PropertyForRent p USING staffNo
```

Inner Joins

- Instead of demanding the same column values in the matching columns:
  - we can demand different relations between the column values

List all Durham-staff who have salary 10% more than some staff member in London

```
SELECT DISTINCT dur.staffNo, dur.fName, dur.lName, dur.position, dur.salary
FROM DurhamStaff dur, LondonStaff lon
WHERE dur.salary > 1.1 * lon.salary;
```

- This type of Join is an inner join:
  - we add the term "natural", if we demand equality for the columns with the same name in the two tables (e.g. dur.salary = lon.salary)
  - inner joins still ignore all rows that do not meet the join conditions

Outer joins

- Inner join:
  - if one row of a table is unmatched, the row is omitted from the output table
- Outer join:
  - it retains (some of) the rows that do not satisfy the join conditions
- Left outer join:
  - it retains the rows of the left table that are unmatched with rows from the right table
- Right outer join:
  - retain the unmatched rows of the right table
- Full outer join:
  - retain the unmatched rows of both tables

Outer joins - Example

Branch

branchNo	bCity
B003	Glasgow
B004	Bristol
B002	London

PropertyForRent

propertyNo	pCity
PA14	Aberdeen
PL94	London
PG4	Glasgow

- The inner join of these two tables:

List the branch offices which have a property for rent in the same city

```
SELECT  b.*, p.*
FROM  Branch b, PropertyForRent p
WHERE  b.bCity = p.pCity;
```

branchNo	bCity	propertyNo	pCity
B003	Glasgow	PG4	Glasgow
B002	London	PL94	London

- The output table has rows where cities coincide
- No rows corresponding to Bristol or Aberdeen

Outer joins - Example

Branch

branchNo	bCity
B003	Glasgow
B004	Bristol
B002	London

PropertyForRent

propertyNo	pCity
PA14	Aberdeen
PL94	London
PG4	Glasgow

- The inner join of these two tables:

List the branch offices which have a property for rent in the same city

```
SELECT  b.*, p.*
FROM  Branch b, PropertyForRent p
WHERE  b.bCity = p.pCity;
```

the same as:

```
SELECT  b.*, p.*
FROM  Branch b INNER JOIN PropertyForRent p
ON  b.bCity = p.pCity;
```

Outer joins - Example

Branch

branchNo	bCity
B003	Glasgow
B004	Bristol
B002	London

PropertyForRent

propertyNo	pCity
PA14	Aberdeen
PL94	London
PG4	Glasgow

- The left outer join of these two tables:

List the branch offices and any properties that are in the same city

```
SELECT  b.*, p.*
FROM Branch b  LEFT JOIN  PropertyForRent p
                          ON b.bCity = p.pCity;
```

branchNo	bCity	propertyNo	pCity
B003	Glasgow	PG4	Glasgow
B004	Bristol	NULL	NULL
B002	London	PL94	London

- Includes the Bristol-row of the left table
  - unmatched with rows from the right table
- No rows corresponding to properties in Aberdeen

Outer joins - Example

Branch

branchNo	bCity
B003	Glasgow
B004	Bristol
B002	London

PropertyForRent

propertyNo	pCity
PA14	Aberdeen
PL94	London
PG4	Glasgow

- The right outer join of these two tables:

List all properties and any branch offices that are in the same city

```
SELECT  b.*, p.*
FROM Branch b  RIGHT JOIN  PropertyForRent p
                        ON b.bCity = p.pCity;
```

branchNo	bCity	propertyNo	pCity
NULL	NULL	PA14	Aberdeen
B003	Glasgow	PG4	Glasgow
B002	London	PL94	London

- Includes the Aberdeen-row of the right table
  - unmatched with rows from the left table
- No rows corresponding to branches in Bristol

Outer joins - Example

Branch

branchNo	bCity
B003	Glasgow
B004	Bristol
B002	London

PropertyForRent

propertyNo	pCity
PA14	Aberdeen
PL94	London
PG4	Glasgow

- The full outer join of these two tables:

List the branch offices and properties that are in the same city, along with any unmatched branches or properties

```
SELECT  b.*, p.*
FROM Branch b  FULL JOIN  PropertyForRent p
                          ON b.bCity = p.pCity;
```

branchNo	bCity	propertyNo	pCity
NULL	NULL	PA14	Aberdeen
B003	Glasgow	PG4	Glasgow
B004	Bristol	NULL	NULL
B002	London	PL94	London

- Includes all rows of both tables (unmatched entries are NULL)

Database Updates

Three SQL statements for modifying the contents of the (existing) tables in the database

- INSERT:
  - adds new rows of data to a table

```
INSERT INTO TableName [(columnList)]
VALUES (data ValueList);
```

- UPDATE:
  - modifies existing data in a table

```
UPDATE  TableName
SET  columnName1 = dataValue1 [,columnName2= data Value2...]
[WHERE  searchCondition];
```

- DELETE:
  - removes rows of data from a table

```
DELETE FROM TableName
[WHERE searchCondition];
```

INSERT

```
INSERT INTO TableName [(columnList)]
VALUES (DataValueList);
```

- columnList is optional:
  - if not specified, then the columns are enumerated in the order given when the table was created
  - if specified, then the columns that are omitted receive the value NULL (unless the DEFAULT option was used in the table creation)
  - if specified, DataValueList must match one-to-one with columnList

e.g. 1:
```
INSERT INTO Staff
VALUES ('SG16', 'Alan', 'Brown', 'Assistant', 10000, 'B003');
```

e.g. 2:
```
INSERT INTO Staff(staffNo, fName, lName, salary)
VALUES ('SG16', 'Alan', 'Brown', 10000);
```

is the same as:

```
INSERT INTO Staff(staffNo, fName, lName, position, salary, branch)
VALUES ('SG16', 'Alan', 'Brown', NULL, 10000, NULL);
```

UPDATE

```
UPDATE  TableName
SET  columnName1 = dataValue1 [,columnName2= data Value2..]
[WHERE  searchCondition];
```

- WHERE is optional:
  - if specified, then only the rows that satisfy the searchCondition are updated
  - if not specified, then all rows are updated
- The new dataValue(s):
  - must be compatible with the data type(s) of the corresponding column(s)

e.g. 1:
```
UPDATE Staff
SET salary = salary * 1.03;
```
(i.e. 3% salary increase to all rows)

e.g. 2:
```
UPDATE Staff
SET salary = salary * 1.05
WHERE position = 'Manager';
```
(i.e. 5% salary increase to managers)

DELETE

DELETE FROM TableName
[WHERE searchCondition];

- WHERE is optional:
  - if specified, then only the rows that satisfy the searchCondition are deleted
  - if not specified, then all rows are deleted

e.g. 1: DELETE FROM Viewing
WHERE propertyNo = 'PG4'; (i.e. delete property PG4 from the table)

e.g. 2: DELETE FROM Viewing; (i.e. delete all the entries in the table)

Note: this is not equivalent to deleting the whole table!

To delete the whole table: DROP TABLE Viewing;

Data Definition Language (DDL) - overview

Basic commands:

- CREATE: Create a new table
  - assign a name to the table and define the names and domains of each of the columns in the table
- ALTER: Amend the relation schema (i.e. table structure)
  - if it is necessary to change the structure of a table because of design error, or just because the design has changed
- Specify integrity & referential constraints
  - PRIMARY KEY, FOREIGN KEY
- DROP: Delete a table
- CREATE VIEW: Define a virtual table
  - virtual relation (table) that appears to the user
  - it is derived from a query on a “real table”

Main domain types in SQL

- **CHAR(n):** character string of fixed length *n*
- **VARCHAR(n):** character string of variable length at most *n*
- **BIT(n):** bit string of fixed length *n*
- **INTEGER:** large positive / negative integer values
- **SMALLINT:** small positive / negative integer values (from –32 768 to 32 767)
- **NUMERIC(p,d):** a (positive / negative) decimal number with at most:
  - **precision p:** total number of all digits (integer + decimal, excluding the decimal point)
  - **scale d:** total number of decimal digits

- Example: 12.851 has precision 5 and scale 3.

Other domain types in SQL

- We can define also our custom domain types, specifically for our needs

change name of a known type:

```
CREATE DOMAIN Postcode AS VARCHAR(8);
```

with additional constraints:

```
CREATE DOMAIN SexType AS CHAR(1)
        CHECK (VALUE IN ('M', 'F'));
```

more complex (nested) definitions:

```
CREATE DOMAIN StaffNumber AS VARCHAR(5)
        CHECK (VALUE IN (SELECT staffNo
                         FROM Staff));
```

Constraints

- Referential actions when defining a table
  (i.e. for FOREIGN KEY):
  - ON UPDATE
    (what to do when the corresponding **primary key** is **updated**)
  - ON DELETE
    (what to do when the corresponding **primary key** is **deleted**)

- Available options for these actions:
  - CASCADE
    (when update / delete: update the foreign key / delete the tuple)
  - SET NULL
    (when update / delete: set the foreign key to NULL)
  - SET DEFAULT
    (when update / delete: set the foreign key to the default value)
  - NO ACTION
    (when update / delete: do nothing – dangerous!!!)

Create Table Construct

- An SQL relation is defined using the **create table** command:

```
CREATE TABLE R(
        Attribute1   Domain1   [NOT NULL | UNIQUE],
        Attribute2   Domain2   [NOT NULL | UNIQUE],
        … ,
        Integrity & Referential constraints);
```

- A good strategy:
  - before the CREATE TABLE R(…);
    it is always safe to write DROP TABLE R;
- Two options for DROP TABLE R:
  - RESTRICT (default option: the DROP is rejected if there are other objects that depend for their existence upon the continued existence of R)
  - CASCADE (the DROP proceeds anyway and SQL drops all dependent objects)

Create Table Construct (Example)

```
DROP  TABLE Person CASCADE;

CREATE  TABLE Person(
        name                    VARCHAR(30) NOT NULL UNIQUE,
        Passport                INTEGER ON DELETE SET NULL
                                        ON UPDATE CASCADE,
        SSN                     INTEGER,
        age                     SHORTINT,
        gender                  SexType,
        married                 BIT(1) DEFAULT 0,
        PRIMARY KEY             (SSN),
        FOREIGN KEY             (Passport) REFERENCES
                                    PassportTable(PassportNo)
);
```

Alter Table Construct (Example)

Used to change the schema of a relation:

add new attributes:

```
ALTER TABLE Person
        ADD salary INTEGER;
```

remove attributes:

```
ALTER TABLE Person
        DROP married CASCADE;
```

change attribute characteristics:

```
ALTER TABLE Person
        ALTER gender SET DEFAULT 'F';
```

Defining Views

- View: a relation (table) that:
  - depends on other relations and
  - is not physically stored as a table
- Main use of views:
  - for presenting different information to different users
  - simplify complex queries

Employee (ssn, name, department, project, salary)

```
CREATE VIEW Developers AS
  SELECT name, project
  FROM   Employee
  WHERE  department = 'Development';
```

Payroll has access to Employee, others only to Developers

Defining Views

- A view that depends on two relations:

Person (name, city)

Purchase (buyer, seller, product, shop)

```
CREATE VIEW DurhamView AS
    SELECT buyer, seller, product, shop
    FROM   Person, Purchase
    WHERE  Purchase.buyer = Person.name AND
           Person.city = 'Durham';
```

- We have a new virtual table:
  "the purchases of people who live in Durham"

  DurhamView (buyer, seller, product, shop)

Querying a View

- We can query views like all other relations (tables):

DurhamView (buyer, seller, product, shop)
Product (name, maker, category)

"Find all names of Durham citizens who bought shoes, and the name of the shop they used"

```
SELECT  buyer, shop
FROM    DurhamView, Product
WHERE   DurhamView.product = Product.name  AND
        Product.category = 'shoes'
```

- Views are computed *on-demand* (thus *slower*)
- Queries on views are "translated" into queries on the original tables

Querying a View

```
SELECT  buyer, DurhamView.shop
FROM    DurhamView, Product
WHERE   DurhamView.product = Product.name  AND
        Product.category = 'shoes'
```

```
SELECT  buyer, Purchase.shop
FROM    Person, Purchase, Product
WHERE   Purchase.buyer = Person.name AND
        Person.city = 'Durham' AND
        Purchase.product = Product.name  AND
        Product.category = 'shoes'
```

Updating Views

- How can we update a table that does not exist?

Employee (ssn, name, department, project, salary)

```
CREATE VIEW Developers AS
  SELECT name, project
  FROM Employee
  WHERE department = 'Development'
```

```
CREATE VIEW Paying AS
  SELECT ssn, name, salary
  FROM Employee
  WHERE salary > 40000
```

if we make the following insertions:

```
INSERT INTO Developers
VALUES ('John', 'Project5')
```

Invalid operation:
we cannot add an entry to Employee without the primary key (ssn)!

```
INSERT INTO Paying
VALUES (82463, 'John', 43000)
```

```
INSERT INTO Employee
VALUES (82463, 'John', NULL,
                       NULL, 43000)
```