Week-9

Introduction to Database Systems

Shah Nawaz

Aggregate Functions
Having, IN, NOT IN, Co-related
Nested Query, EXISTS, NOT EXISTS,
VIEW, ANY , ALL,

Relational Algebra

?

```
SELECT COUNT(ID), State
FROM Customer
Where COUNT(ID) > 5
GROUP BY State;
```

Customer table

ID	Sales ID	Finance ID	Name	Initials	City	State	Email address	Balance
MD12	12345	C5729	Young	N	San Francisco	CA	young@xyz.com	100
MD13	23324	LA781	Stills	S	New Orleans	LA	stil@stil.org	200
MD14	57657	J7301	Furay	R	Yellow Springs	OH	buf@spring.com	150
MD15	65461	K8839	Palmer	B	New York	NY	bass@spring.com	200
...	...	...	...	...	...	...	...	...

?

SELECT COUNT(ID), State
FROM Customer
Where COUNT(ID) > 5
GROUP BY State;

No, This SQL query has a syntax error. The alias **Count** that is defined in the SELECT clause cannot be used in the WHERE clause directly.

Customer table

ID	Sales ID	Finance ID	Name	Initials	City	State	Email address	Balance
MD12	12345	C5729	Young	N	San Francisco	CA	young@xyz.com	100
MD13	23324	LA781	Stills	S	New Orleans	LA	stil@stil.org	200
MD14	57657	J7301	Furay	R	Yellow Springs	OH	buf@spring.com	150
MD15	65461	K8839	Palmer	B	New York	NY	bass@spring.com	200
...	...	...	...	...	...	...	...	...

?

`WHERE` keyword cannot be used with aggregate functions.

HAVING

- The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.
- It is used for condition (just like where clause) on aggregate function's result.
- Having clause will not execute if there is no group by.
- WHERE clause filters data from individual rows, while the HAVING clause filters data from groups of rows.

Having Clause

```
SELECT COUNT(ID) As TotalCount, State
FROM Customer
GROUP BY State
Having TotalCout > 5 ;
```

Customer table

ID	Sales ID	Finance ID	Name	Initials	City	State	Email address	Balance
MD12	12345	C5729	Young	N	San Francisco	CA	young@xyz.com	100
MD13	23324	LA781	Stills	S	New Orleans	LA	stil@stil.org	200
MD14	57657	J7301	Furay	R	Yellow Springs	OH	buf@spring.com	150
MD15	65461	K8839	Palmer	B	New York	NY	bass@spring.com	200
...	...	...	...	...	...	...	...	...

Multiple comparisons of any attribute?

```
SELECT *
FROM Agents
Where working_area = “London” OR
working_area = “Mumbai” OR
working_area = “Chennai”;
```

IN

```
SELECT *
FROM Agents
Where working_area = “London” OR
working_area = “Mumbai” OR
working_area = “Chennai”;
```

OR

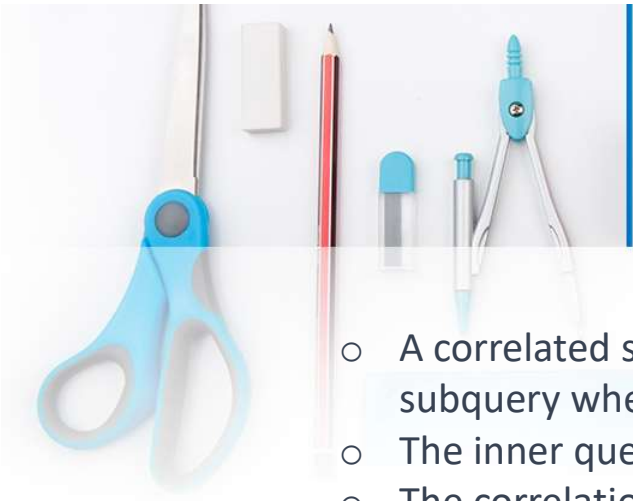
```
SELECT *
FROM agents
WHERE working_area IN('London','Mumbai','Chennai');
```

agent_code	working_area	commission
A007	Bangalore	0.15
A005	Brisban	0.14
A001	Bangalore	0.14
A003	London	0.12
A008	New York	0.12
A002	Mumbai	0.11
A006	London	0.15
A004	Torento	0.15
A011	Bangalore	0.15
A010	Chennai	0.14
A009	Hampshair	0.11
A012	San Jose	0.12

NOT IN

```
SELECT *
FROM agents
WHERE commission NOT IN(.13, .14, .12);
```

agent_code	working_area	commission	
A007	Bangalore	0.15	
A005	Brisban	0.14	X
A001	Bangalore	0.14	X
A003	London	0.12	X
A008	New York	0.12	X
A002	Mumbai	0.11	
A006	London	0.15	
A004	Torento	0.15	
A011	Bangalore	0.15	
A010	Chennai	0.14	X
A009	Hampshair	0.11	
A012	San Jose	0.12	X



Co-related Nested Query

- A correlated subquery, also known as a **correlated nested query**, is a type of SQL subquery where the inner query references columns from the outer query.
- The inner query is executed once for each row processed by the outer query.
- The correlation between the inner and outer queries is established through references to columns in both queries.
- The basic structure of a correlated subquery looks like this:

```
SELECT column1, column2, ...
FROM table1
WHERE condition_operator
(SELECT column3 FROM table2 WHERE condition_based_on_outer_query);
```

Co-related Nested Query

Employees

Employee_ID	First_Name	Last_Name	Department_ID
1	John	Doe	1
2	Jane	Smith	2
3	Jim	Brown	1
4	Jake	White	3

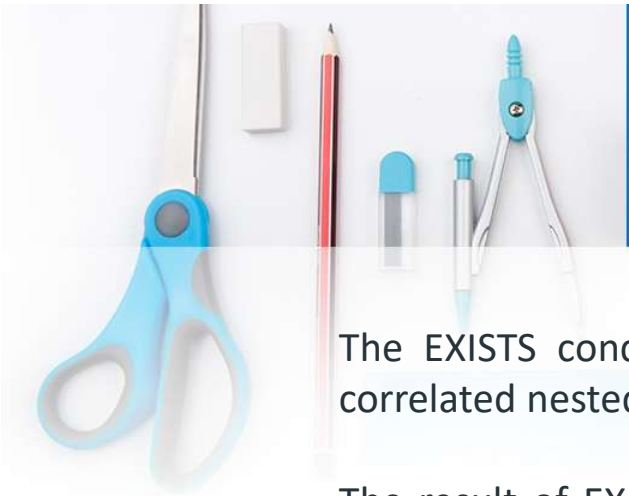
Department

DepartmentID	Department_Name
1	Sales
2	HR
3	IT

Requirement

We want to find the first names and last names of employees who work in the 'Sales' department.

```
SELECT FirstName, LastName FROM Employees
WHERE DepartmentID =
(SELECT DepartmentID  FROM Departments
   WHERE DepartmentName = 'Sales'
);
```



EXISTS

The EXISTS condition in SQL is used to check whether the result of a correlated nested query is empty (contains no tuples) or not.

The result of EXISTS is a boolean value True or False. It can be used in a SELECT, UPDATE, INSERT or DELETE statement.

EXISTS

Customers

customer_id	lname	fname	website
401	Singh	Dolly	abc.com
402	Chauhan	Anuj	def.com
403	Kumar	Niteesh	ghi.com
404	Gupta	Shubham	jkl.com
405	Walecha	Divya	abc.com
406	Jain	Sandeep	jkl.com
407	Mehta	Rajiv	abc.com
408	Mehra	Anand	abc.com

Orders

order_id	c_id	order_date
1	407	2017-03-03
2	405	2017-03-05
3	408	2017-01-18
4	404	2017-02-05

```
SELECT fname, lname
FROM Customers WHERE EXISTS
(SELECT * FROM Orders WHERE Customers.customer_id = Orders.c_id);
```

fname	lname
Shubham	Gupta
Divya	Walecha
Rajiv	Mehta
Anand	Mehra

NOT EXISTS

Customers

customer_id	lname	fname	website
401	Singh	Dolly	abc.com
402	Chauhan	Anuj	def.com
403	Kumar	Niteesh	ghi.com
404	Gupta	Shubham	jkl.com
405	Walecha	Divya	abc.com
406	Jain	Sandeep	jkl.com
407	Mehta	Rajiv	abc.com
408	Mehra	Anand	abc.com

Orders

order_id	c_id	order_date
1	407	2017-03-03
2	405	2017-03-05
3	408	2017-01-18
4	404	2017-02-05

```
SELECT fname, lname
FROM Customers WHERE NOT EXISTS
(SELECT * FROM Orders WHERE Customers.customer_id = Orders.c_id);
```

lname	fname
Singh	Dolly
Chauhan	Anuj
Kumar	Niteesh
Jain	Sandeep

View

In SQL, a view is a virtual table based on the result of a SELECT query. It does not store the data itself but provides a way to represent the result of a query as if it were a table.
Views are useful for simplifying complex queries, providing a layer of security by restricting access to certain columns or rows, and abstracting the underlying structure of the database.

Views provide a way to encapsulate complex queries, control access to data, and present a simplified interface to users or applications interacting with the database.

View

To create a view, you can use the CREATE VIEW statement followed by a SELECT query that defines the view.

Creating View:

```
CREATE VIEW my_view AS
SELECT column1, column2
FROM my_table
WHERE condition;
```

Querying from View:

```
SELECT * FROM my_view;
```

View

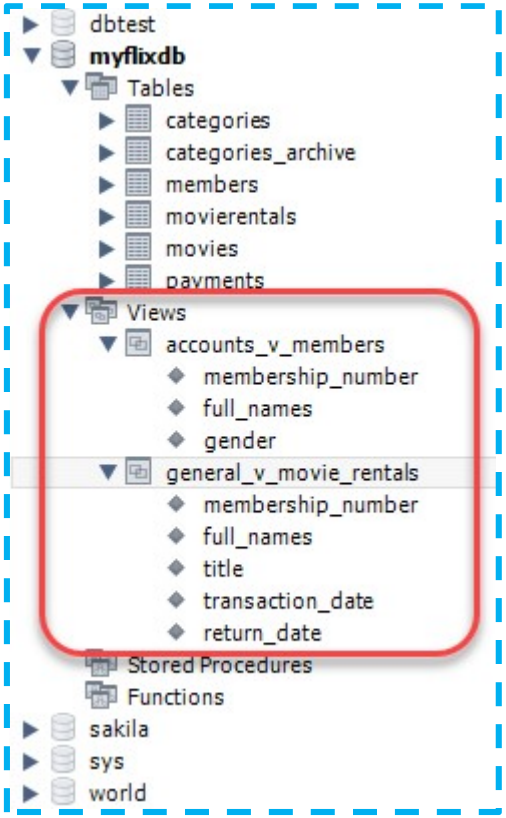
To create a view, you can use the CREATE VIEW statement followed by a SELECT query that defines the view.

Creating View:

```
CREATE VIEW employee_view AS
SELECT employee_id, first_name, last_name
FROM employees
WHERE department_id = 10;
```

Querying from View:

```
SELECT * FROM employee_view;
```



View

Deleting View:

```
DROP VIEW employee_view
```

View

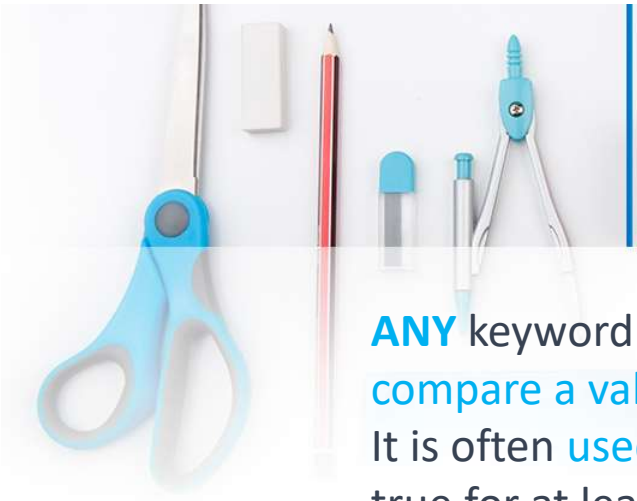
Joining tables:

```
CREATE VIEW order_details AS
SELECT orders.order_id, customers.customer_name, order_date
FROM orders
JOIN customers ON orders.customer_id = customers.customer_id;
```

View

Aggregate view:

```
CREATE VIEW product_summary AS
SELECT category, AVG(price) AS avg_price, COUNT(*) AS total_products
FROM products
GROUP BY category;
```



ANY

ANY keyword is used in conjunction with a comparison operator to compare a value to any value in a set of values.
It is often used in subqueries and conditions to check if a condition is true for at least one element in a set.

```
SELECT * FROM Teachers
WHERE age < ANY (  SELECT age   FROM Students );
```

Here, the SQL command selects rows if age in the outer query is less than any age in a subquery.

ANY

Table: Teachers

id	name	age
1	Peter	32
2	Megan	43
3	Rose	29
4	Linda	30
5	Mary	41

Table: Students

id	name	age
1	Harry	23
2	Jack	42
3	Joe	32
4	Dent	23
5	Bruce	40

```
SELECT *
FROM Teachers
WHERE age < ANY (
  SELECT age
  FROM Students
);
```

id	name	age
1	Peter	32
3	Rose	29
4	Linda	30
5	Mary	41

E.g first of all, Teacher age value 32 is picked and compared with age of students. If any student age is less than teacher age 32 then all the records of Teacher (1, Peter, 32) will be returned.

ALL

SQL **ALL** compares a value of the first table with all values of the second table and returns the row if there is a match with all values.

For example, if we want to find teachers whose age is greater than all students, we can use

```
SELECT * FROM Teachers
WHERE age >  ALL (  SELECT age  FROM Students);
```

Here, if the teacher's age is greater than all student's ages, the corresponding row of the Teachers table is selected.

ALL

Table: Teachers

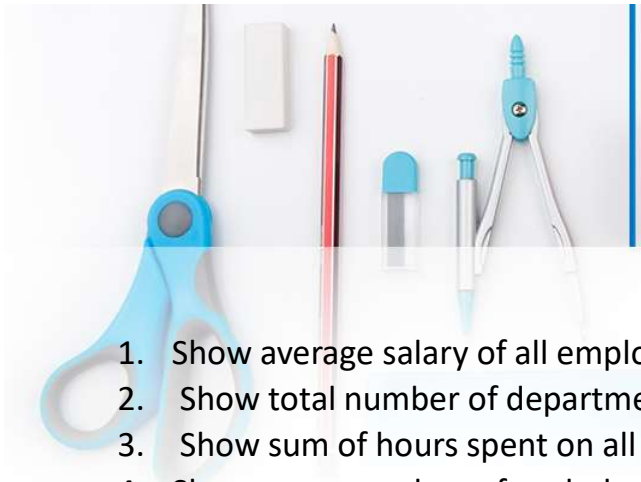
id	name	age
1	Peter	32
2	Megan	43
3	Rose	29
4	Linda	30
5	Mary	41

Table: Students

id	name	age
1	Harry	23
2	Jack	42
3	Joe	32
4	Dent	23
5	Bruce	40

```
SELECT *
FROM Teachers
WHERE age > ALL (
  SELECT age
  FROM Students
);
```

id	name	age
2	Megan	43



Practice Queries

1. Show average salary of all employees.
2. Show total number of departments.
3. Show sum of hours spent on all projects.
4. Show average salary of each department if average salary is greater than 40,000.
5. Show minimum salary of each department if minimum salary is less than 20,000.
6. Show total number of employees of each department with department name.
7. Show number of dependent of each employee.
8. Show count of projects of all departments in ascending order by department name.
9. Show count of total locations of each department which are located in Lahore.
10. For each project show project name and total number of employees working on it.

Relational Algebra

HAVING

In relational algebra, there isn't a specific symbol dedicated to the HAVING clause, as relational algebra primarily focuses on basic operations like selection (σ), projection (π), join (⋈), and grouping (γ).

The HAVING clause is a part of SQL and is used in combination with the GROUP BY clause to filter the results of a grouped query based on a specified condition.

HAVING

Example:

SELECT COUNT(ID) As TotalCount, State FROM Customer
GROUP BY State Having TotalCout > 5 ;

Above query can be represented as follows:

πTotalCount, State(σTotalCount > 5(γState; COUNT(ID) AS TotalCount(Customer)))

Pronounciation: "Projection of TotalCount and State, from the selection where TotalCount is greater than 5, applied to the grouping by State with the count of ID as TotalCount in the Customer relation."

Explanation:

•γState; COUNT(ID) AS TotalCount(Customer): This represents the Group By operation, grouping the tuples in the Customer relation by the "State" attribute and calculating the count of IDs for each group.
•σTotalCount > 5: This represents the Selection operation, applying a condition to filter groups where the TotalCount is greater than 5.
•πTotalCount, State: This represents the Projection operation, selecting the "TotalCount" and "State" attributes from the result.

IN

In relational algebra, the IN operator from SQL is typically expressed using a combination of operations. Let's consider a scenario where you want to retrieve all orders from the "Orders" relation where the customer ID is in a specified set {101, 102, 105}. The SQL query for this is:

SELECT * FROM Orders WHERE CustomerID IN (101, 102, 105);

In relational algebra, you might represent this as follows:

$$\sigma \text{CustomerID} \in \{101, 102, 105\}(\text{Orders})$$

Pronunciation: "Selection where CustomerID is in the set {101, 102, 105}, applied to the Orders relation."
Explanation:

- σ represents the Selection operation.
- "CustomerID \in {101, 102, 105}" is the condition specifying that the CustomerID should be in the set {101, 102, 105}.
- OrdersOrders is the relation name.

NOT IN

Let's consider a scenario where you want to retrieve all orders from the "Orders" relation where the customer ID is not in a specified set {101, 102, 105}. The SQL query for this is:

SELECT * FROM Orders WHERE CustomerID NOT IN (101, 102, 105);

In relational algebra, you might represent this as follows:

$$\sigma \text{CustomerID} \notin \{101, 102, 105\}(\text{Orders})$$

Pronunciation:"Selection where CustomerID is not in the set {101, 102, 105}, applied to the Orders relation."
Explanation:

- σ represents the Selection operation.
- "CustomerID NOT IN {101, 102, 105}" is the condition specifying that the CustomerID should not be in the set {101, 102, 105}.
- OrdersOrders is the relation name.

EXISTS

Let's consider a scenario where you want to retrieve all customers who have placed at least one order. The SQL query for this could be:

```
SELECT * FROM Customers WHERE
EXISTS (SELECT * FROM Orders WHERE Orders.CustomerID = Customers.CustomerID);
```

In relational algebra, you might represent this as follows:

$$\text{Customers} \bowtie (\pi\ \text{Orders.CustomerID} = \text{Customers.CustomerID}\ (\text{Orders}))$$

Pronunciation: “Projection of all attributes from Customers, where there exists a tuple in the Cartesian Product of Customers and Orders such that the CustomerID matches between the two relations."

NOT EXISTS

Let's consider a scenario where you want to retrieve all customers who have not placed any orders. The SQL query for this could be:

```
SELECT * FROM Customers WHERE NOT EXISTS
(SELECT * FROM Orders WHERE Orders.CustomerID = Customers.CustomerID);
```

In relational algebra, you might represent this as follows:

Customers − (Customers ⋈ Orders)

$$\text{Customers} \bowtie - (\pi \text{ Orders.CustomerID} = \text{Customers.CustomerID (Orders)})$$

Pronunciation:"Projection of all attributes from Customers, where there does not exist a match in the Cartesian Product of CustomerID from Orders and CustomerID from Customers, applied to the Customers relation."

CREAT VIEW

Let's consider a scenario where you want to create a view named “LargeValue" that includes orders with a total amount greater than $1000. The SQL query to create such a view might look like this:

```
CREATE VIEW LargeValue AS
SELECT OrderID, CustomerID, OrderDate, TotalAmount FROM Orders
WHERE TotalAmount > 1000;
```

In relational algebra, you might represent this as follows:

$$\text{LargeValue} \leftarrow \pi_{\text{OrderID, CustomerID, OrderDate, TotalAmount}}(\sigma_{\text{TotalAmount} > 1000}(\text{Orders}))$$

Pronunciation: Create a view named LargeValue, which is the projection of OrderID, CustomerID, OrderDate, and TotalAmount from the selection of Orders where TotalAmount is greater than 1000."

CREAT VIEW

Explanation:

•σTotalAmount > 1000(Orders) represents the Selection operation, filtering the tuples from the Orders relation where the TotalAmount is greater than $1000.

•πOrderID, CustomerID, OrderDate, TotalAmount represents the Projection operation, selecting specific attributes from the result of the selection.

•HighValueOrders← represents the assignment of the result to a view named HighValueOrders.

ANY

The SQL query you provided selects all rows from the "Teachers" table where the "age" is less than any of the ages in the result of the subquery that selects ages from the "Students" table. The equivalent relational algebra expression might look like this:

SELECT * FROM Teachers WHERE age < ANY (SELECT age FROM Students);"

In relational algebra, you might represent this as follows:

$$\pi \text{Teachers } (\sigma \text{age} < \text{ANY}(\pi \text{age}(\text{Students}))(\text{Teachers}))$$

Explanation:

- πage(Students): This expression represents the projection (π) of the 'age' attribute from the 'Students' relation.
- ANY(πage(Students)): The ANYANY operator is applied to the result of the subquery, meaning it returns true if the 'age' in the 'Teachers' relation is less than any of the ages in the subquery.
- σage<ANY(πage(Students))(Teachers): This is the final selection, filtering rows in the 'Teachers' relation where the 'age' is less than any age obtained from the subquery.

ALL

The SQL query you provided selects all rows from the "Teachers" table where the "age" is less than all of the ages in the result of the subquery that selects ages from the "Students" table. The equivalent relational algebra expression might look like this:

SELECT * FROM Teachers WHERE age < ALL (SELECT age FROM Students);"

In relational algebra, you might represent this as follows:

πTeachers (σage<ALL(πage(Students)) (Teachers))

Explanation:

•πage(Students): This expression represents the projection (π) of the 'age' attribute from the 'Students' relation.

•ALL(πage(Students)): The ALLALL operator is applied to the result of the subquery, meaning it returns true if the 'age' in the 'Teachers' relation is less than all of the ages in the subquery.

•σage<ALL(πage(Students))(Teachers): This is the final selection, filtering rows in the 'Teachers' relation where the 'age' is less than all ages obtained from the subquery.

UNION

Let's consider a scenario where you want to retrieve the distinct values of "City" from both the "Customers" and "Suppliers" tables using the UNION operator. The SQL query and its relational algebra expression might look like this:

SELECT City FROM Customers UNION SELECT City FROM Suppliers;

In relational algebra, you might represent this as follows:

πCity(Customers) $\cup$ πCity(Suppliers)

Pronunciation: "Projection of the 'City' attribute from the 'Customers' table unioned with the projection of the 'City' attribute from the 'Suppliers' table."

Explanation:

- πCity(Customers) represents the Projection operation, selecting the "City" attribute from the "Customers" relation.
- πCity(Suppliers) represents the Projection operation, selecting the "City" attribute from the "Suppliers" relation.
- $\cup$ represents the Union operation, combining the distinct values of "City" from both projections.

UNION ALL

Let's consider a scenario where you want to retrieve all values of "City" from both the "Customers" and "Suppliers" tables, including duplicates, using the UNION ALL operator. The SQL query and its relational algebra expression might look like this:

SELECT City FROM Customers UNION ALL SELECT City FROM Suppliers;

In relational algebra, you might represent this as follows:

πCity(Customers) ∪ ALL πCity(Suppliers)

Pronunciation: "Projection of the 'City' attribute from the 'Customers' table union all with the projection of the 'City' attribute from the 'Suppliers' table."

Explanation:

- πCity(Customers) represents the Projection operation, selecting the "City" attribute from the "Customers" relation.
- πCity(Suppliers) represents the Projection operation, selecting the "City" attribute from the "Suppliers" relation.
- UALL represents the Union All operation, combining all values of "City" from both projections, including duplicates.

INTERSECT

Let's consider a scenario where you want to retrieve the common values of "City" from both the "Customers" and "Suppliers" tables using the INTERSECT operator. The SQL query and its relational algebra expression might look like this:

SELECT City FROM Customers INTERSECT SELECT City FROM Suppliers;

In relational algebra, you might represent this as follows:

$$\pi City(Customers) \cap \pi City(Suppliers)$$

Pronunciation: "Projection of the 'City' attribute from the 'Customers' table intersect with the projection of the 'City' attribute from the 'Suppliers' table."

Explanation:

- πCity(Customers) represents the Projection operation, selecting the "City" attribute from the "Customers" relation.
- πCity(Suppliers) represents the Projection operation, selecting the "City" attribute from the "Suppliers" relation.
- $\cap$ represents the Intersection operation, retrieving the common values of "City" from both projections.

Thank You all!