



Unit 2: SQL

2.1. DML: Queries and Data Manipulation

2.2. SQL Exercises (Lab. sessions)

2.3. DDL: Data Definition Language

Unit 2.1 DML: Queries and Data Manipulation

**2 Lab.
Seminars
+
1 session**

1. Introduction to SQL

2. Queries

2.1. Simple queries using one table.

2.2. Simple queries with several tables

2.3. Subqueries

2.4. Universal Quantification

2.5. Quantified comparison predicates

2.6. Grouping

2.7. Set operations

2.8. Joins

3. Database updates

4. Commands for handling transactions

1. Introduction to SQL

SQL (**S**tructured **Q**uery **L**anguage) is a standard language for defining and manipulating a relational database.

Includes:

- Features from Relational Algebra (Algebraic Approach)
- Features from Tuple Relational Calculus (Logical Approach)
- Others
- SQL has evolved: SQL'92, SQL'99, SQL:2003, SQL:2008, SQL:2011, SQL:2011, SQL:2016
- We will use the basics of the language (present from the main revision SQL'92)

SQL sublanguages

DDL (Data Definition Language): Creation and modification of relational DB schemas.

DML (Data Manipulation Language): Queries and database updates.

- **SELECT:** Allows the declaration of queries to retrieve the information from the database
- **INSERT:** Performs the insertion of one or more rows in a table
- **DELETE:** Allows the user to delete one or more rows from a table
- **UPDATE:** Modifies the values of one or more columns and/or one or more rows in a table

Control Language: Dynamically changes the database properties

TEAM (teamname: char(25), director: char(30))
PK:{teamname}

CYCLIST(cnum: integer, name: char(30), age: integer, teamname: char(25))
PK:{cnum}
FK:{teamname}→ TEAM
NNV:{teamname}
NNV:{name}

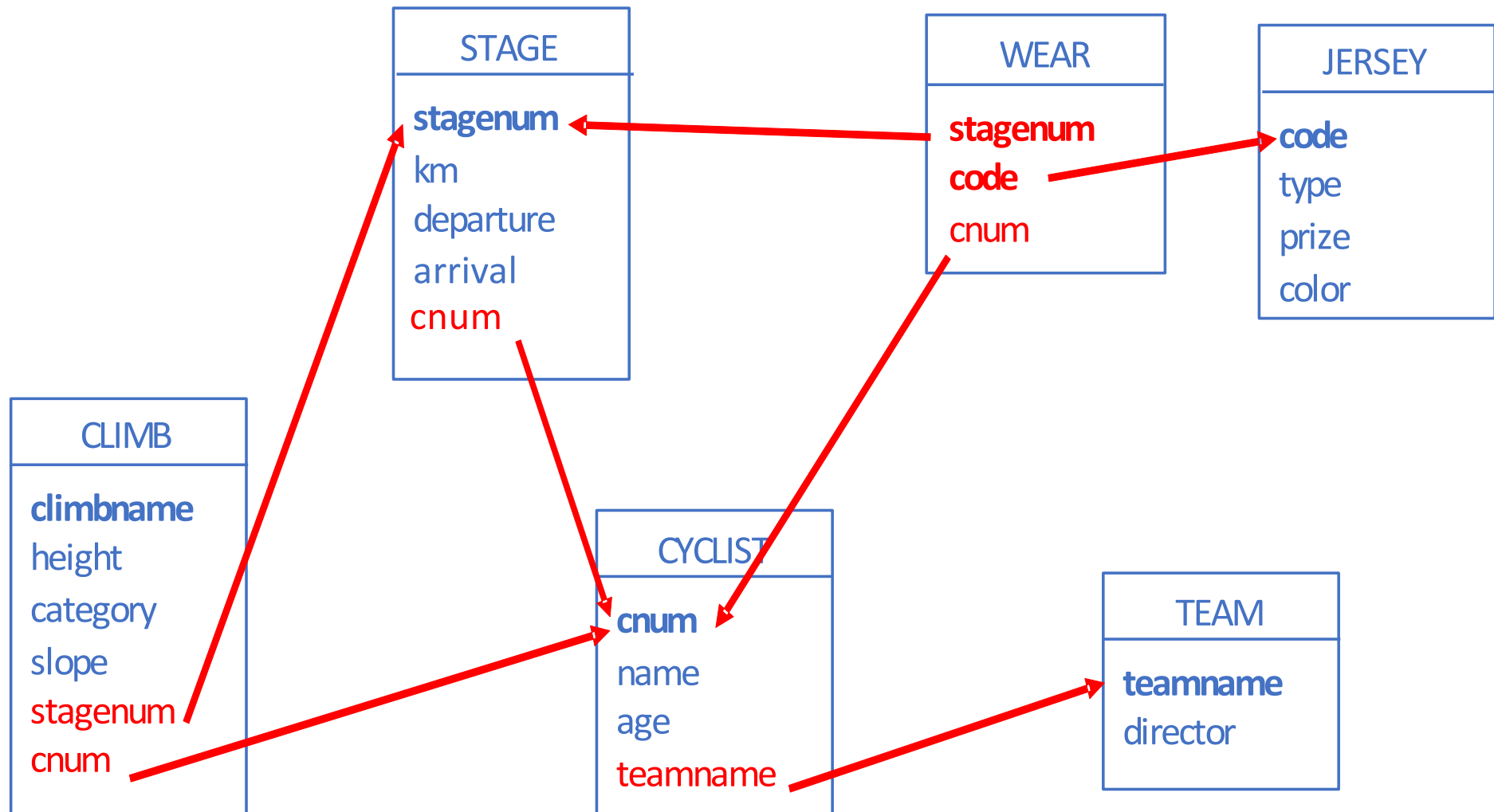
STAGE(stagenum: integer, km: integer, departure: char(35), arrival: char(35),
cnum: integer)
PK:{stagenum}
FK:{cnum}→ CYCLIST

JERSEY(code: char(3), type: char(30), prize: integer, color: char(25))
PK:{code}

CLIMB(climbname: char(30), height: integer, category: char(1), slope: real,
stagenum: integer, cnum: integer)
PK:{climbname}
FK:{stagenum}→ STAGE
FK:{cnum}→ CYCLIST
NNV:{stagenum}

WEAR(stagenum: integer, code: char(3), cnum: integer)
PK:{stagenum, code}
FK:{stagenum}→ STAGE
FK:{cnum}→ CYCLIST
FK:{code}→ JERSEY
NNV:{cnum}

Cycling race



UD 2.1 DML: Queries and Data Manipulation

1. Introduction to SQL

2. Queries

2.1. Simple queries using one table.

2.2. Simple queries with several tables

2.3. Subqueries

2.4. Universal Quantification

2.5. Quantified comparison predicates

2.6. Grouping

2.7. Set operations

2.8. Joins

3. Database updates

4. Commands for handling transactions

Example:

List the name and the age of all the cyclists.

```
SELECT name, age  
FROM Cyclist;
```


Mathematics operations

$+$, $-$, $*$, $/$, ...

Example:

List for each jersey its type and the prize in euros (suppose that the prizes are in pesetas and $1\text{€} = 166 \text{ ptas.}$) but only for those jerseys which prize is greater than 100 euros.

```
SELECT type, prize / 166  
FROM Jersey  
WHERE prize / 166 > 100;
```

Example:

List the name, the height and the category of all the climbs (order by height and category).

```
SELECT climbname, height, category  
FROM Climb  
ORDER BY height, category ;
```

Example: List the name and the height of all the climbs in the 1^a category in ascending order according to the height

1. Which tables contain the information?
2. What are the conditions for the tuples?
3. What information is going to be returned?
4. Is any order required?

```
SELECT climbname, height
FROM Climb
WHERE category = '1'
ORDER BY height ;
```

DISTINCT

Example:

List the teams of the cyclists (only the team).

```
SELECT DISTINCT teamname  
FROM Cyclist;
```

Example:

List all the information in the Team table.

```
SELECT *  
FROM Team;
```

LIKE

LIKE is followed by a pattern which will be used with strings

% The percent sign represents zero, one, or multiple characters

_ The underscore represents a single character

Example:

Obtain the name and age of the cyclists who belong to the teams whose name contains the string “cyclist”.

```
SELECT name, age
FROM Cyclist
WHERE teamname LIKE '%cyclist%'
```

Example:

List the stage numbers where the arrival city name has, as first letter, an “A”, or where the departure city name contains two or more ‘e’s

```
SELECT stagenum
```

```
FROM Stage
```

```
WHERE arrival LIKE 'A%' OR departure LIKE '%e%e%';
```

BETWEEN

Example:

Name of the cyclists with an age between 20 and 30

```
SELECT name  
FROM Cyclist  
WHERE age BETWEEN 20 AND 30;
```

$exp \text{ between } exp_1 \text{ and } exp_2 \equiv (exp \geq exp_1) \text{ and } (exp \leq exp_2)$

IS NULL

Wrong query (syntax error)

```
SELECT teamname  
FROM Team  
WHERE director = null
```

The right query is:

```
SELECT teamname  
FROM Team  
WHERE director IS NULL
```


Using the NULL VALUE

$A \alpha B$ (where α is a comparison operator) is evaluated as **undefined** if at least one A or B is null; otherwise it is evaluated to the certainty value of the comparison $A \alpha B$

Example:

```
SELECT *  
FROM T  
WHERE atrib1 > atrib2
```

If a tuple has $\text{atrib}_1 = 50$ and **atrib₂ is null**, the comparison is undefined, and therefore that tuple will not be included in the query result.

Aggregated values (non-grouped queries)

{ avg | max | min | sum | count } ([all | distinct]
expression) | count(*)

- **Distinct** is used to remove the duplicate values before the aggregated function calculate the result.
- The **NULL values** are removed before the aggregated function calculate the result.
- If the number of **selected rows is 0**, COUNT returns 0, and the rest of the functions return the NULL value.

Example:

```
SELECT 'Num. ciclysts=', COUNT(*), 'average age=', AVG(age)
FROM Cyclist
WHERE teamname = 'Banesto';
```

In non-grouped queries, the SELECT clause can only include references to aggregated functions or literals, since the functions will return just a single value.

WRONG EXAMPLE:

```
SELECT name, AVG(age)
FROM Cyclist
WHERE teamname = 'ONCE';
```

Alias and DESCending order

Name and age of the 'Banesto' cyclists ordered from oldest to youngest. The column name must be "Banesto"

```
SELECT name AS Banesto, age  
FROM Cyclist  
WHERE teamname = 'Banesto'  
ORDER BY age DESC;
```

UD 2.1 DML: Queries and Data Manipulation

1. Introduction to SQL

2. Queries

2.1. Simple queries using one table.

2.2. Simple queries with several tables

2.3. Subqueries

2.4. Universal Quantification

2.5. Quantified comparison predicates

2.6. Grouping

2.7. Set operations

2.8. Joins

3. Database updates

4. Commands for handling transactions

If the information required by the query is in several tables, the query will include those tables in the **FROM** clause.

A query over several tables corresponds to the **Cartesian product**:

- If we do not express several conditions to connect them, the number of rows will be very high.
- If there are **foreign keys** defined, it is usual that some conditions are formed by an **equality between the foreign key** and the corresponding attributes in the table **to which it refers**.
- With **n** tables, we will typically have (at least) **n-1 connections**. (The way in which tables are connected and the attributes that are used determine the meaning of the query.)

Example: SELECT * FROM T1, T2 WHERE T1.n = T2.n

T1

t1	n
a1	b1
a2	b2
a3	b3

T2

t2	n
c1	d1
c2	b2

T1 x T2

t1	n	t2	n
a1	b1	c1	d1
a1	b1	c2	b2
a2	b2	c1	d1
a2	b2	c2	b2
a3	b3	c1	d1
a3	b3	c2	b2

Example: List pairs stage-climb which have been won by the same cyclist.

1. Which tables contain the information?

FROM Stage, Climb

2. Which rows must be selected?

WHERE Stage.cnum = Climb.cnum;

3. What attributes are going to be returned?

SELECT Stage.stagenum, climbname

Note that the columns *cnum* from *stage* and *cnum* from *Climb* are prefixed by the table name to avoid ambiguity:

SELECT Stage.stagenum, climbname

FROM Stage, Climb

WHERE Stage.cnum = Climb.cnum ;

Alias

Example: Obtain the name of the cyclists who belong to the team directed by 'Alvaro Pino'

```
SELECT C.name  
FROM Cyclist C, Team T  
WHERE C.teamname = T.teamname AND T.director = 'Alvaro Pino';
```



The alias can be used to refer any table:

Syntax: **FROM** table **[AS]** *alias*

Example:

Obtain the name of the cyclists and the stage number such that the cyclist has won that stage. Additionally, the stage must be more than 150 km long

```
SELECT C.name, S.stagenum  
FROM Cyclist C, Stage S  
WHERE C.cnum = S.cnum AND S.km > 150;
```

Example: List the names of the cyclists who belongs to the same team as 'Miguel Induráin' but who are younger than he is

1. Which tables contain the information?

FROM Cyclist

But we need to compare tuples of Cyclist with tuples of the same table, so we use this **table twice**

FROM Cyclist C1, Cyclist C2

2. Which rows must be selected?

WHERE C2.name='Miguel Induráin' AND C1.teamname = C2.teamname
AND C1.age < C2.age;

3. What attributes are going to be returned?

SELECT DISTINCT C1.name

SELECT DISTINCT C1.name FROM Cyclist C1, Cyclist C2
==> WHERE C2.name='Miguel Induráin'
AND C1.teamname = C2.teamname AND C1.age < C2.age;²⁷

UD 2.1 DML: Queries and Data Manipulation

1. Introduction to SQL

2. Queries

2.1. Simple queries using one table.

2.2. Simple queries with several tables

2.3. Subqueries

2.4. Universal Quantification

2.5. Quantified comparison predicates

2.6. Grouping

2.7. Set operations

2.8. Joins

3. Database updates

4. Commands for handling transactions

What is a subquery ?

A subquery is a query between parenthesis included inside other query.

Example: Calculate the number and length of the stages with climbs

```
SELECT DISTINCT S.stagenum, km  
FROM Stage S, Climb CL  
WHERE S.stagenum = CL.stagenum
```

Using a subquery:

```
SELECT stagenum, km  
FROM Stage  
WHERE stagenum IN (SELECT stagenum  
FROM Climb)
```

Main query

The subquery
returns the
number of the
stages

Example: Obtain the names of the cyclists who belong to the team directed by 'Alvaro Pino'.

Solved through equalities:

```
SELECT C.name  
FROM Cyclist C, Team T  
WHERE C.teamname = T.teamname AND T.director = 'Alvaro Pino';
```

Using a subquery:

```
SELECT C.name  
FROM Cyclist C  
WHERE C.teamname = ( SELECT T.teamname FROM Team T  
                     WHERE T.director = 'Alvaro Pino' );
```

The name of the cyclist is not in the table used in the subquery (Team)

This is possible only if the subquery returns one single value

Name of the team directed by 'Alvaro Pino'

Predicates accepting subqueries

Subqueries can appear in the search conditions, either in the “where” or in the “having” clauses, as arguments of some predicates:

- **Comparison predicates:** =, <>, >, <, >=, <=.
- **IN:** Checks that a value belongs to the collection (table) returned by the subquery
- **EXISTS:** It is equivalent to the existential quantifier. It checks if a subquery returns some row.

Comparison Predicates: =, <>, >, <, >=, <=

SYNTAX:

row_constructor **comparison predicate** *row_constructor*

“*row_constructor*” is either a sequence of constants or a subquery.

```
('Álvaro Pino', 28) = (SELECT name , age  
                        FROM Cyclist  
                        WHERE cnum= '666')
```

When row constructor returns **more than one column**, the lexicographic **order** will be used in the comparison of each operator.

For simplicity, we will only see queries with **one column** in the subquery.

Subqueries can be a parameter of a comparison if (and only if):

- Return only **a single row**, and
- the number of **columns match** (in number and type) with the other side of the comparison predicate.

If the **result of the subquery is empty**, the row is converted into a row with **NULL** values in all its columns and the result of the comparison will be **undefined**.

Example: Obtain the name of the climbs whose height is greater than the mean of the height of all 2nd category climbs.

1. Which tables contain the information?

Climb ==> FROM Climb

2. Which rows must be selected?

height > AVG(height) of the second category climbs

==> WHERE height > (SELECT AVG(height) FROM Climb
WHERE category = '2');



It is 1 value = 1 row

Check any height (height) with the value returned by AVG(height)

Example: Obtain the name of the climbs whose height is greater than the mean of the height of all 2nd category climbs.

1. Which tables contain the information?

Climb ==> FROM Climb

2. Which rows must be selected?

height > AVG(height) of the second category climbs

==> WHERE height > (SELECT AVG(height) FROM Climb
WHERE category = '2');

3. What attributes are going to be returned?

climbname ==> SELECT climbname ==> *1 column
with n rows*

==> SELECT climbname FROM Climb
WHERE height > (SELECT AVG(height) FROM Climb
WHERE category = '2');

Example: Obtain the name of the climbs whose height is greater than the mean of the height of all 2nd category climbs.

```
SELECT climbname FROM Climb
WHERE height > ( SELECT AVG(height)
                  FROM Climb
                  WHERE category = '2' );
```

Example: Obtain the name of the climbs whose height is greater than the mean of the height of all 2nd category climbs.

~~SELECT climbname FROM Climb
WHERE height > (SELECT height FROM Climb
WHERE category = '2');~~

1 value (1 row)

1 column with n rows

==> It can't be checked !

WRONG: (execution error).

Example: Obtain the name of the climbs whose height is greater than the mean of the height of all 2nd category climbs.

WRONG: (Execution error):

```
SELECT climbname  
FROM Climb  
WHERE height > AVG (SELECT height FROM Climb  
WHERE category = '2' );
```

Example: List the name of the departure and the arrival cities of the stages where the steepest climbs are located.

```
SELECT DISTINCT S.departure, S.arrival
FROM Stage S, Climb CL
WHERE S.stagenum = CL.stagenum
      AND slope = (SELECT MAX(slope)
                   FROM Climb) ;
```


IN Predicate

SYNTAX:

row_constructor [NOT] IN (*table_expression*)

Example:

Obtain the *stagenum* of the stages which have been won by cyclists whose age is greater than 30 years.

```
SELECT stagenum
FROM Stage
WHERE cnum IN ( SELECT cnum
                  FROM Cyclist
                  WHERE age > 30);
```

NOT IN and NULL values

E_0 NOT IN ($E_1, E_2, \dots E_n$)

The expression is true if E_0 is different to all the values of the set $E_1, E_2, \dots E_n$

What will be the result if the NULL value is in the set ?

$E_0 \neq E_1$ and $E_0 \neq E_2$ and ... $E_0 \neq \text{NULL}$... and $E_0 \neq E_n \rightarrow \text{undefined}$

Example:

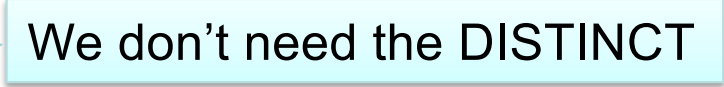
Obtain the name of the cyclists who haven't won any stage.

```
SELECT name
FROM Cyclist
WHERE cnum NOT IN ( SELECT cnum
                     FROM Stage
                     WHERE cnum IS NOT NULL );
```

Subqueries and DISTINCT

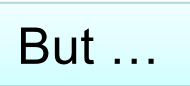
Example: Obtain the director of the teams which have one or more cyclists with a name starting with 'B'.

```
SELECT director
FROM Team
WHERE teamname IN
    (SELECT teamname FROM Cyclist
     WHERE name LIKE 'B%');
```



We don't need the DISTINCT

But ...



SELECT DISTINCT T.director
FROM Cyclist C, Team T
WHERE C.teamname = T.teamname
AND C.name LIKE 'B%' ;

Nested Queries

Example:

Obtain the stage number won by cyclists who belong to teams whose director has a name beginning with 'B'.

```
SELECT stagenum FROM Stage
WHERE cnum IN
    (SELECT cnum FROM Cyclist
     WHERE teamname IN (SELECT teamname FROM Team
                        WHERE director LIKE 'B%'));
```

EXISTS Predicate

Syntax:


EXISTS (*table_expression*)

- The exists predicate is evaluated to **true** if the expression **SELECT returns at least one** row.
- In general, **IN and EXISTS are interchangeable** and, when there is no negation, they can be eliminated (creating queries using multiple tables and adding comparison with the foreign keys)*

(*) That is also true for “NOT IN” and “NOT EXISTS”

Example: Obtain the name of the cyclists who has worn a jersey with a prize lower than 8000 eur.

```
SELECT C.name FROM Cyclist C
WHERE EXISTS ( SELECT *
                FROM Jersey J, Wear W
                WHERE J.prize < 8000 AND J.code = W.code
                AND C.cnum = W.cnum );
```



Also:

```
SELECT C.name FROM Cyclist C
WHERE 0 < ( SELECT COUNT(*)
            FROM Jersey J, Wear W
            WHERE J.prize < 8000 AND J.code = W.code
            AND C.cnum = W.cnum );
```

Example: Obtain the name of the cyclists who has worn a jersey with a prize lower than 8000 eur.

```
SELECT C.name FROM Cyclist C
WHERE C.cnum IN ( SELECT W.cnum
                  FROM Wear W, Jersey J
                  WHERE J.prize < 8000
                  AND W.code = J.code );
```

Also:

```
SELECT DISTINCT C.name
FROM Cyclist C, Wear W, Jersey J
WHERE C.cnum = W.cnum
      AND W.code = J.code
      AND J.prize < 8000 ;
```

Example: Obtain the name of the cyclists who haven't won any stage.

```
SELECT name
FROM Cyclist C
WHERE NOT EXISTS (SELECT *
                  FROM Stage S
                  WHERE S.cnum = C.cnum);
```

WHERE NOT EXISTS (SELECT * FROM ...)

Is equivalent to: WHERE 0 = (SELECT COUNT(*) FROM ...)

Example: Obtain the name of the cyclists who haven't won any stage.

```
SELECT name
FROM Cyclist C
WHERE cnum NOT IN (SELECT cnum
                    FROM Stage S
                    WHERE S.cnum IS NOT NULL);
```

```
SELECT name
FROM Cyclist C, Stage S
WHERE C.cnum <> S.cnum ;
```

No sense

UD 2.1 DML: Queries and Data Manipulation

1. Introduction to SQL

2. Queries

2.1. Simple queries using one table.

2.2. Simple queries with several tables

2.3. Subqueries

2.4. Universal Quantification

2.5. Quantified comparison predicates

2.6. Grouping

2.7. Set operations

2.8. Joins

3. Database updates

4. Commands for handling transactions

Universal Quantification

SQL'92 and most DBMS nowadays do not provide the universal quantification (FORALL). We must transform the query to solve it with an EXISTS:

$$\forall X F(X) \equiv \neg \exists X \neg F(X)$$

Example:

“Obtain the name of the cyclists (if any) who have won all the stages with more than 200 km.

The query is converted into:

“Obtain the name of the cyclists such that there does not exist a stage with more than 200 km which has not be won by that cyclist”

(*) Assuming that we know that there are some stage with more than 200 km

Obtain the name of the cyclists such that there does not exist a stage with more than 200 km which has not be won by that cyclist

```
SELECT name FROM Cyclist C
WHERE NOT EXISTS ( SELECT *
                    FROM Stage S
                    WHERE km > 200 AND
                           C.cnum <> S.cnum );
```

Assuming that we know that there are some stage with more than 200 km

Problem: What happens if there is no stage with more than 200 km?

```
SELECT name FROM Cyclist C
WHERE NOT EXISTS ( SELECT *
                   FROM Stage S
                   WHERE km > 200 AND C.cnum <> S.cnum );
```

In that case, this query returns the name of all the cyclists !!!

Solution:

Check that exists at least one stage with more tan 200 Km
(ADD-ON).

```
SELECT name FROM Cyclist C
WHERE NOT EXISTS (SELECT *
                  FROM Stage S
                  WHERE km > 200 AND
                        C.cnum <> S.cnum )
AND EXISTS ( SELECT *
             FROM Stage S
             WHERE km > 200)      ;
```

Example:

List the cyclists who have worn all the (kinds of) jerseys.

```
SELECT C.name FROM Cyclist C
WHERE NOT EXISTS
      (SELECT * FROM Jersey J
      WHERE NOT EXISTS
            (SELECT * FROM Wear W
            WHERE C.cnum = W.cnum )
            AND W.código = J.código )
AND EXISTS ( SELECT * FROM jersey )
```

Example: List the name of all the cyclist who have won all the climbs in some stage and have won that stage

```
SELECT C.name FROM Cyclist C, Stage S
WHERE S.cnum = C.cnum
      AND NOT EXISTS ( SELECT * FROM Climb CL
                        WHERE CL.stagenum=S.stagenum
                        AND   C.cnum <> CL.cnum );

AND EXISTS ( SELECT * FROM Climb CL
              WHERE CL.stagenum=S.stagenum );
```

Because there could be some stage with no climbs

UD 2.1 DML: Queries and Data Manipulation

1. Introduction to SQL

2. Queries

2.1. Simple queries using one table.

2.2. Simple queries with several tables

2.3. Subqueries

2.4. Universal Quantification

2.5. Quantified comparison predicates

2.6. Grouping

2.7. Set operations

2.8. Joins

3. Database updates

4. Commands for handling transactions

SYNTAX:

row_constructor { **ALL** | **ANY** | **SOME** } (*table_expression*)

- The comparison predicate which is quantified with **ALL** is evaluated to true if it is **true for all the rows** in the table expression (if the table is empty it is also evaluated to true).
- The comparison predicate which is quantified with **ANY** or **SOME** is evaluated to true if it is **true for some** of the rows in the table expression (if the table is empty it is evaluated to false).

(*) The predicate “IN” is equivalent to the quantified comparison predicate “=any”.

Example:

Obtain the name of the climb/s and the cyclist/s who won the climb/s with the highest slope.

```
SELECT CL.climbname, C.name
FROM Climb CL, Cyclist C
WHERE CL.cnum = C.cnum
      AND CL.slope >= ALL (SELECT slope
                           FROM Climb);
```

Example:

Obtain the climbs and the cyclists who won them, such that the climb is not the one with the lowest slope.

```
SELECT CL.climbname, C.name
FROM Climb CL, Cyclist C
WHERE CL.cnum = C.cnum
      AND CL.slope > ANY (SELECT slope
                           FROM Climb);
```

(*) ANY can always be converted into an ALL by negating the condition and adding a NOT, and vice versa.

UD 2.1 DML: Queries and Data Manipulation

1. Introduction to SQL

2. Queries

2.1. Simple queries using one table.

2.2. Simple queries with several tables

2.3. Subqueries

2.4. Universal Quantification

2.5. Quantified comparison predicates

2.6. Grouping

2.7. Set operations

2.8. Joins

3. Database updates

4. Commands for handling transactions

A group is a set of rows with the same value for the subset of columns used for grouping (used in the GROUP BY).

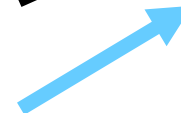
Example: Obtain the name of all the teams and the average age of the cyclists in each team.

```
SELECT teamname, AVG(age)
FROM Cyclist
GROUP BY teamname;
```

teamname	age
Banesto	22
ONCE	25
PDM	32
Banesto	25
Kelme	28
ONCE	30
Kelme	29
Banesto	28

The **aggregated functions** in the grouped queries do not work like the rest of queries. Here, they return **a value for each group** which is formed.

	teamname	age
	Banesto	22
	Banesto	25
	Banesto	28
	ONCE	25
	ONCE	30
	PDM	32
	Kelme	29
	Kelme	28



One value
for group

```
SELECT teamname, AVG(age)
FROM Cyclist
GROUP BY teamname;
```

Returns:

teamnames	age
Banesto	25
ONCE	27,5
PDM	32
Kelme	28,5

SYNTAX:

SELECT [ALL | DISTINCT] {expression₁, expression₂,..., expression_n|*}
FROM *table₁, table₂ ..., table_n*
[WHERE condition]
[GROUP BY *column₁, column₂,..., column_n*
[HAVING *conditional_expression*]]
[ORDER BY *column₁, column₂,..., column_n*]

Group, where and having

The **HAVING** clause can only appear in grouped queries, and it is an extra condition similar to **WHERE**, but applied to the groups:

- 1°) **WHERE** condition (used for the rows)
- 2°) Grouping and calculus of aggregated functions
- 3°) **HAVING** condition (used for the groups)

In the **HAVING** clause can only appear references to columns used in the group by, aggregated functions, or subqueries.

In the **SELECT** clause can only appear references to columns used in the group by, aggregated functions, or literals.

WRONG EXAMPLE:

```
SELECT teamname, name, AVG(age)  
FROM Cyclist  
GROUP BY teamname;
```

Group, where and having

The **where** clause is applied **before** grouping.

Example:

Obtain the name of the teams with a name starting with 'K', and the average age of their cyclists who are older than 25

```
5 ----> SELECT teamname, AVG(age)
1 ----> FROM Cyclist
2 ----> WHERE age > 25
3 ----> GROUP BY teamname
4 ----> HAVING teamname LIKE "K%" ;
```

In the grouped queries, it is possible to use **nested aggregated functions**.

Example:

Obtain the average age for the team with the maximum average age (of their members).

```
SELECT MAX(AVG(age))  
FROM Cyclist  
GROUP BY teamname;
```

Obtain the name of the teams and the average age of their cyclists who are older than 25, from those teams **with more than 8 cyclists who are older than 25.**

```
SELECT teamname, AVG(age)
FROM Cyclist
WHERE age > 25
GROUP BY teamname
HAVING COUNT(cnum) > 8 ;
```

Obtain the name of the teams and the average age of their cyclists who are older than 25, from those teams **with more than 8 cyclists** ~~who are older than 25~~.

```
SELECT C.teamname, AVG(C.age)
FROM Cyclist C
WHERE C.age > 25
GROUP BY C.teamname
HAVING 8 < (SELECT COUNT(*)
            FROM Cyclist C2
            WHERE C.teamname = C2.teamname);
```

Example:

Obtain the name of the cyclist and the number of climbs he has won, but only if the mean of the slope of the won climbs is greater than 10.

```
SELECT C.name, COUNT(CL.climbname)
FROM Cyclist C, Climb CL
WHERE C.cnum = CL.cnum
GROUP BY C.cnum, C.name      /*Always group by the PK */
HAVING AVG (CL.slope) >10 ;
```

Example:

Obtain the **name of the cyclists** who have won at least one stage and belong to a team with more than 5 cyclists, indicating how many stages has won that cyclist.

```
SELECT C.name, count(*)
FROM Cyclist C, Stage S
WHERE C.cnum = S.cnum
      AND 5 < ( SELECT count(*)
                  FROM Cyclist C2
                  WHERE C2.teamname = C.teamname )
GROUP BY C.cnum, C.name ;
```

UD 2.1 DML: Queries and Data Manipulation

1. Introduction to SQL

2. Queries

2.1. Simple queries using one table.

2.2. Simple queries with several tables

2.3. Subqueries

2.4. Universal Quantification

2.5. Quantified comparison predicates

2.6. Grouping

2.7. Set operations

2.8. Joins

3. Database updates

4. Commands for handling transactions

Other table combinations

There are other ways to **combine several tables** in the same query. All of them, along with the ways we have already seen, are what we have called “*table_expression*”.

- Include several tables in the **FROM** clause.
- Use of **subqueries** in the conditions in the where or having clause .
- **Set table combinations**: use the set operators to combine the tables.
- **Table joins**: combine two tables by using different variants of the JOIN operator in Relational Algebra.

Set combinations

Correspond to the UNION, DIFFERENCE and INTERSECTION in the relational algebra.

- UNION
- EXCEPT (MINUS in Oracle 10)
- INTERSECT

They make possible to combine tables with **compatible** schemas.

UNION

table_expression union [all] table_expression

Performs a **union** of the rows of the tables expressed by the two “table_expression”.

Duplicates will be allowed if the option **ALL** is set.

Example:

Obtain the name of all the people participating in the cycling race.

(SELECT name FROM Cyclist)

UNION

(SELECT director FROM Team)

Obtain the name of the cyclists who have worn some jersey or have won a climb or a stage.

```
SELECT name
FROM Cyclist
WHERE cnum IN
    (SELECT cnum FROM Wear
     UNION
     SELECT cnum FROM Climb
     UNION
     SELECT cnum FROM Stage)
```

INTERSECCION

table_expression intersect table_expression

Performs a **intersection** of the rows of the tables expressed by the two “table_expression”.

Example:

Obtain the name of the cyclists with the same name as a team director.

(SELECT name FROM Cyclist)

INTERSECT

(SELECT director FROM Team)

DIFERENCE

table_expression₁ except¹ table_expression₂

Return the tuples in table_expression₁ which do not appear in table_expression₂.

Example:

Name of cyclist which do not appear in the table of teams as director

```
(SELECT name FROM Cyclist)
EXCEPT
(SELECT director FROM Team)
```

¹MINUS in Oracle 10

UD 2.1 DML: Queries and Data Manipulation

1. Introduction to SQL

2. Queries

2.1. Simple queries using one table.

2.2. Simple queries with several tables

2.3. Subqueries

2.4. Universal Quantification

2.5. Quantified comparison predicates

2.6. Grouping

2.7. Set operations

2.8. Joins

3. Database updates

4. Commands for handling transactions

JOIN

3 Types of Joins (concatenation in the relational algebra).

1. Cross join
2. Inner Join
3. Outer join

1. Cross join

table_reference₁ **cross join** *table_reference₂*

≡

SELECT * FROM *table_reference₁*, *table_reference₂*

2. Inner Join

Syntax (3 different forms):

*table_reference*₁

[**NATURAL**] [**INNER**] **JOIN**

*table_reference*₂

[**ON** condition] **USING** (*column*₁, *column*₂, ..., *column*_{*n*})

Form 1:

table₁ [inner] join table₂ on conditional expression

≡ SELECT * FROM *table1, table2*
 WHERE *conditional_expression*

Example: Obtain the names of the climbs and the number of the stage (stage) in which the climb is, if the stage length is greater than 200km.

```
SELECT climbname, CL.stagenum  
FROM Climb CL JOIN Stage S ON CL.stagenum= S.stagenum  
WHERE S.km>200
```

Form 2:

table1 **[inner] join** *table2* **using** (c1, c2,... cn)

≡ SELECT * from *table1*, *table2*
 WHERE *table1.c1* = *table2.c1*
 AND *table1.c2* = *table2.c2*
 AND..... and *table1.cn* = *table2.cn*

Example: Obtain the name of the climbs, the number of the stage in which the climb is and the length of the stage, but only if the climb is higher than 800 (height).

```
SELECT climbname, stagenum, km  
FROM Climb JOIN Stage USING (stagenum)  
WHERE height>800
```

Form 3:

table1 natural inner join *table2*

≡ *table1* join *table2* using (c1, c2,, cn)
where *table1* has n attributes.

*(It is a regular JOIN but using the **common** attributes of both tables)*

Example: Obtain the name of the cyclists of the team directed by 'Alvaro Pino'.

```
SELECT name  
FROM Cyclist NATURAL JOIN Team  
WHERE director = 'Alvaro Pino';
```

Example: Obtain the name of the climbs, their stage, and the km of the stage of those climbs higher than 800.

```
SELECT climbname, stagenum, km  
FROM Climb JOIN Stage USING (stagenum)  
WHERE height>800
```

WRONG:

```
SELECT climbname, stagenum, km  
FROM Climb CL NATURAL JOIN Stage  
WHERE CL.height>800
```

because cnum appears in both tables

Obtain the number and name of the cyclists, and the amount of jerseys worn by each of them

```
SELECT C.cnum, C.name, COUNT (DISTINCT L.code)
FROM Cyclist C, Wear W
WHERE C.cnum = W.cnum
GROUP BY C.cnum, C.name
```

```
SELECT cnum, Cyclist.name, COUNT (DISTINCT wear.code)
FROM Cyclist NATURAL INNER JOIN Wear
GROUP BY cnum, cyclist.name
```

3. OuterJoin

Combine **all** the rows from one of the tables (even if there is no correspondence for some row in the other table)

table_expression

[**NATURAL**] {**LEFT** | **RIGHT** | **FULL**} [**OUTER**] **JOIN**

table_expression

[**ON** condition| **USING** (*column₁*, *column₂*,..., *column_n*)]

Table1 **LEFT JOIN** Table2 **ON** conditional_expression

Inner join of *Table1* and *Table2* UNION tuples from **Table1** that do not appear in the inner join, using NULL values in the rest of columns

FULL: Returns all the tuples from *table1* and *table2*

Example:

Obtain the name of **all** the cyclists and the total number of stages won by each of them.

```
SELECT name, COUNT(stagenum)
FROM Cyclist NATURAL LEFT JOIN stage
GROUP BY cnum, name
```

Example:

Obtain for **all cyclist**, his/her number, name and the code of every jersey worn by that cyclist (and the number of the stage in which that cyclist has worn that jersey).

```
SELECT C.cnum, name, code, stagenum  
FROM Cyclist C LEFT JOIN Wear W ON C.cnum = W.cnum
```

List the name of all the teams, indicating how many cyclist there are in each of them

```
SELECT team.teamname, COUNT(cnum)
FROM Team LEFT JOIN Cyclist
          ON team.teamname= cyclist.teamname
GROUP BY team.teamname
```

```
( SELECT teamname, count(*)
  FROM cyclist
  GROUP BY teamname )
UNION
( SELECT teamname, 0
  FROM team
  WHERE teamname NOT IN (SELECT teamname FROM cyclist) )
```

UD 2.1 DML: Queries and Data Manipulation

1. Introduction to SQL

2. Queries

2.1. Simple queries using one table.

2.2. Simple queries with several tables

2.3. Subqueries

2.4. Universal Quantification

2.5. Quantified comparison predicates

2.6. Grouping

2.7. Set operations

2.8. Joins

3. Database updates

4. Commands for handling transactions

3. Database updates

DML (Data Manipulation Language): Queries and database updates.

- **SELECT:** Allows the declaration of queries to retrieve the information from the database
- **INSERT:** Performs the insertion of one or more rows in a table
- **DELETE:** Allows the user to delete one or more rows from a table
- **UPDATE:** Modifies the values of one or more columns and/or one or more rows in a table

The INSERT command

```
INSERT INTO table [(column1, column2,..., columnn)]  
    { DEFAULT VALUES |  
      VALUES (atom1, atom2,... atomn) |  
      table_expression }
```

- If we do **not** include the **list** of columns, we will have to specify **all the attributes** of the table.
- If we include the option “**default values**”, we will insert a single row with all the default values which were defined in the definition of the table.
- In the option (**atom_commalist**), the atoms are given by **scalar** expressions.
- In the option **table_expression**, we insert the rows which result from the execution of the expression (a **SELECT**).

Example (a **complete** tuple):

Add a cyclist with cnum 101, name 'Joan Peris', age 27, and team 'Kelme'.

```
INSERT INTO Cyclist  
VALUES (101, 'Joan Peris', 27, 'Kelme');
```

Example (an **incomplete** tuple) :

Add a cyclist with cnum 101, name 'Joan Peris', and team 'Kelme' (we don't know the age):

```
INSERT INTO Cyclist (cnum, name, teamname)  
VALUES (101, 'Joan Peris', 'Kelme');
```

Example (inserting **many** tuples):

Add into the Winner table (same schema as Cyclist) all the information of the cyclists who have won some stage.

```
INSERT INTO Winner
( SELECT cnum, name, age, teamname
FROM Cyclist
WHERE cnum IN (SELECT cnum FROM Stage) )
```

The DELETE command

DELETE FROM *table* [**WHERE** *conditional_expression*]

If we include the WHERE clause, then it will only delete the rows which make the condition true. In other case, all the tuples will be deleted.

Example:

Delete the information about the cyclist 'M. Indurain' because he is retired.

```
DELETE FROM Cyclist  
WHERE name = 'M. Indurain'
```

The UPDATE command

UPDATE *table*

SET assignment₁, assignment₂, ..., assignment_n

[**WHERE** conditional_expression]

Where the assignments are of the form::

column = {DEFAULT | NULL | scalar_expression}

Example:

Increase the *prize* of the jerseys by 10%

UPDATE jersey

SET prize = prize * 1.10

Example:

Move all the Kelme cyclists to the *K10* team.

```
UPDATE Cyclist  
    SET teamname = 'K10'  
    WHERE teamname='Kelme' ;
```

UD 2.1 DML: Queries and Data Manipulation

1. Introduction to SQL

2. Queries

2.1. Simple queries using one table.

2.2. Simple queries with several tables

2.3. Subqueries

2.4. Universal Quantification

2.5. Quantified comparison predicates

2.6. Grouping

2.7. Set operations

2.8. Joins

3. Database updates

4. Commands for handling transactions

4. Commands for handling transactions

A **transaction** is a logical unit of work consisting of one or more SQL statements that is guaranteed to be atomic with respect to recovery.

Transaction initiation::

It is **implicit**. A transaction begin with the first SQL statement in a session, or when the previous transaction ends.

Transaction completion:

- **COMMIT**: The transaction ends **successfully**, making the database changes permanent.
- **ROLLBACK**: The transaction **aborts**, backing out any changes made by the transactions.

Example:

The 'Banesto' team changes its name to 'BanQue'

/ Here we must write a command (SET CONSTRAINT) to differ the evaluation of the foreign key *teamname* in *Cyclist*. We will study this command in the next section */*

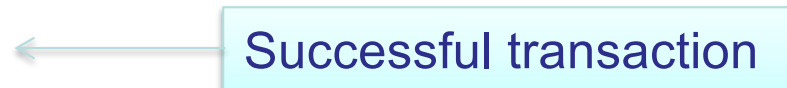
UPDATE Team SET name = 'BanQue'

WHERE teamname = 'Banesto';

UPDATE Cyclist SET teamname = 'BanQue'

WHERE teamname = 'Banesto';

COMMIT;



Successful transaction

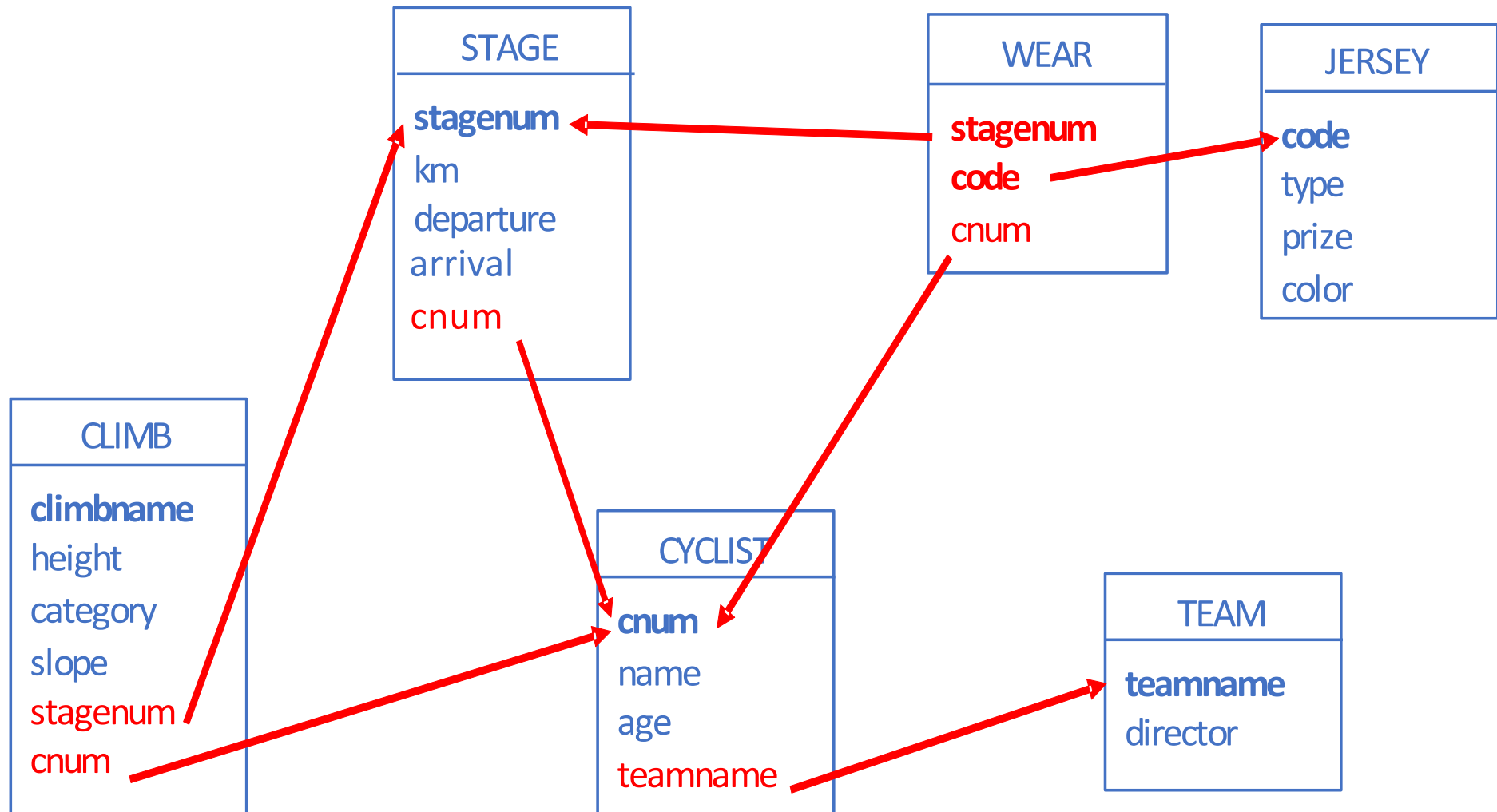
SQL Queries review

Operator	Relational Algebra	SQL
selection	$R \text{ Donde } F$	SELECT ... FROM R WHERE F
projection	$R [A_i, A_j \dots, A_k]$	SELECT $A_i, A_j \dots, A_k$ FROM R
Cartesian product	$R_1 \times R_2, \dots \times R_n$	SELECT ... FROM R_1, R_2, \dots, R_n o SELECT...FROM R_1 CROSS JOIN $R_2, \dots,$ CROSS JOIN R_n
join	$R_1 \bowtie R_2$	SELECT... FROM R_1 NATURAL JOIN R_2
union	$R_1 \cup R_2$	SELECT * FROM R_1 UNION SELECT * FROM R_2
difference	$R_1 - R_2$	SELECT * FROM R_1 EXCEPT SELECT * FROM R_2
intersection	$R_1 \cap R_2$	SELECT * FROM R_1 INTERSECT SELECT * FROM R_2

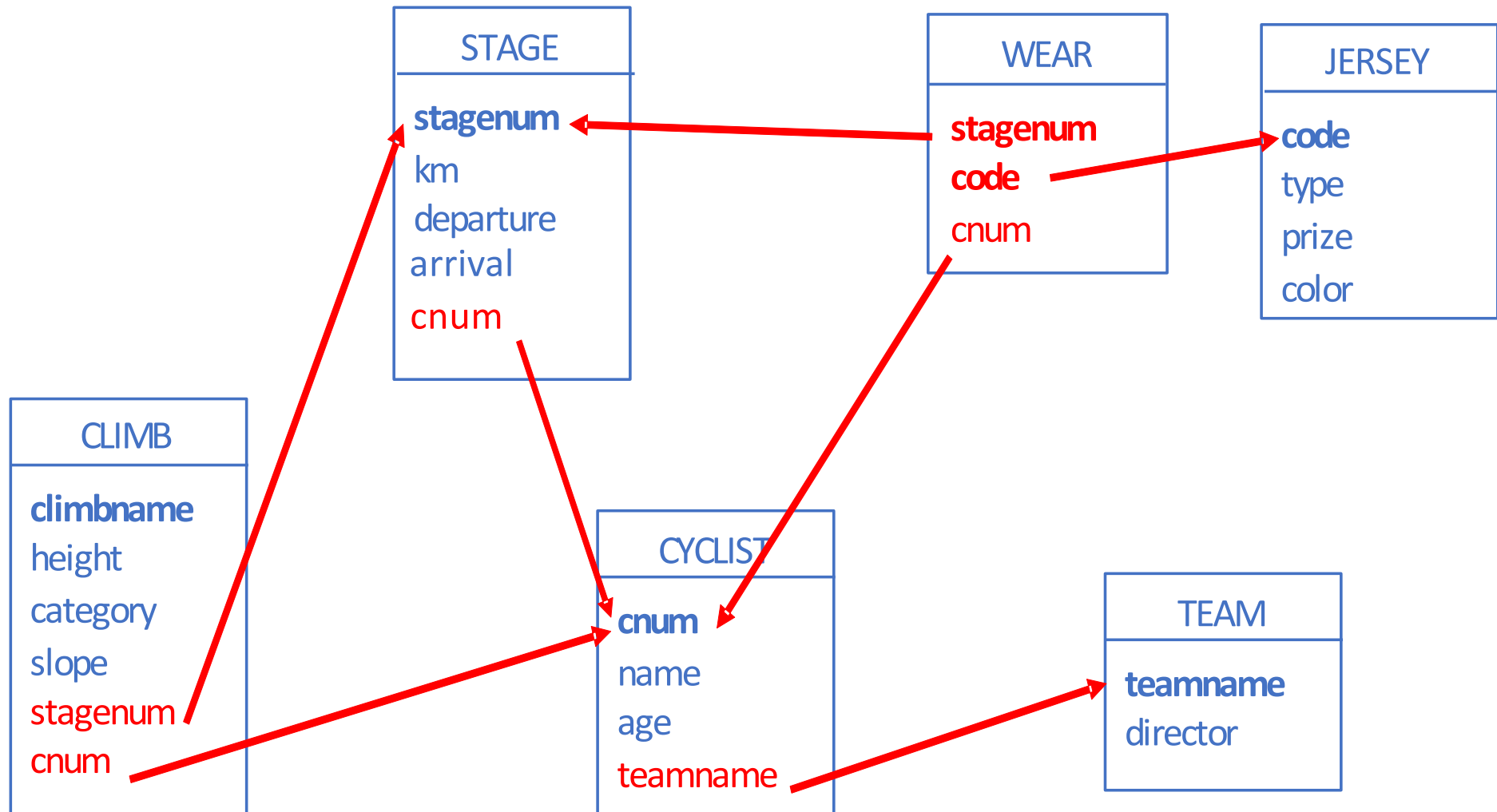
Some exercises

1. List the name of the youngest cyclist. (21)
2. List the value of the attribute stagenum and the departure city for those stages with no climbs. (16)
3. List the name of the departure and the arrival of the stages where the steepest climbs are located. (19)
4. List the name of the cyclists who have won all the climbs in one stage and have also won the stage. (27)
5. List the code and the color of those jerseys which have only been worn by cyclists of the same team. (29)
6. List the name of the cyclists who belong to a team which has more than five cyclists and they have also won one or more stages. Please also indicate how many stages he has won.
7. List the name of all the cyclists who belong to a team which has more than five cyclists indicating how many stages he has won. (35)
8. List the cyclist number and the name of the cyclists who have not worn all the jerseys worn by the cyclist with number 20. (40)
9. List the name of the teams and the average age of the cyclists of those teams who have the highest average age of all the teams. (36)
10. List the name of those teams such that their cyclists have only won climbs of category = 1. (30)

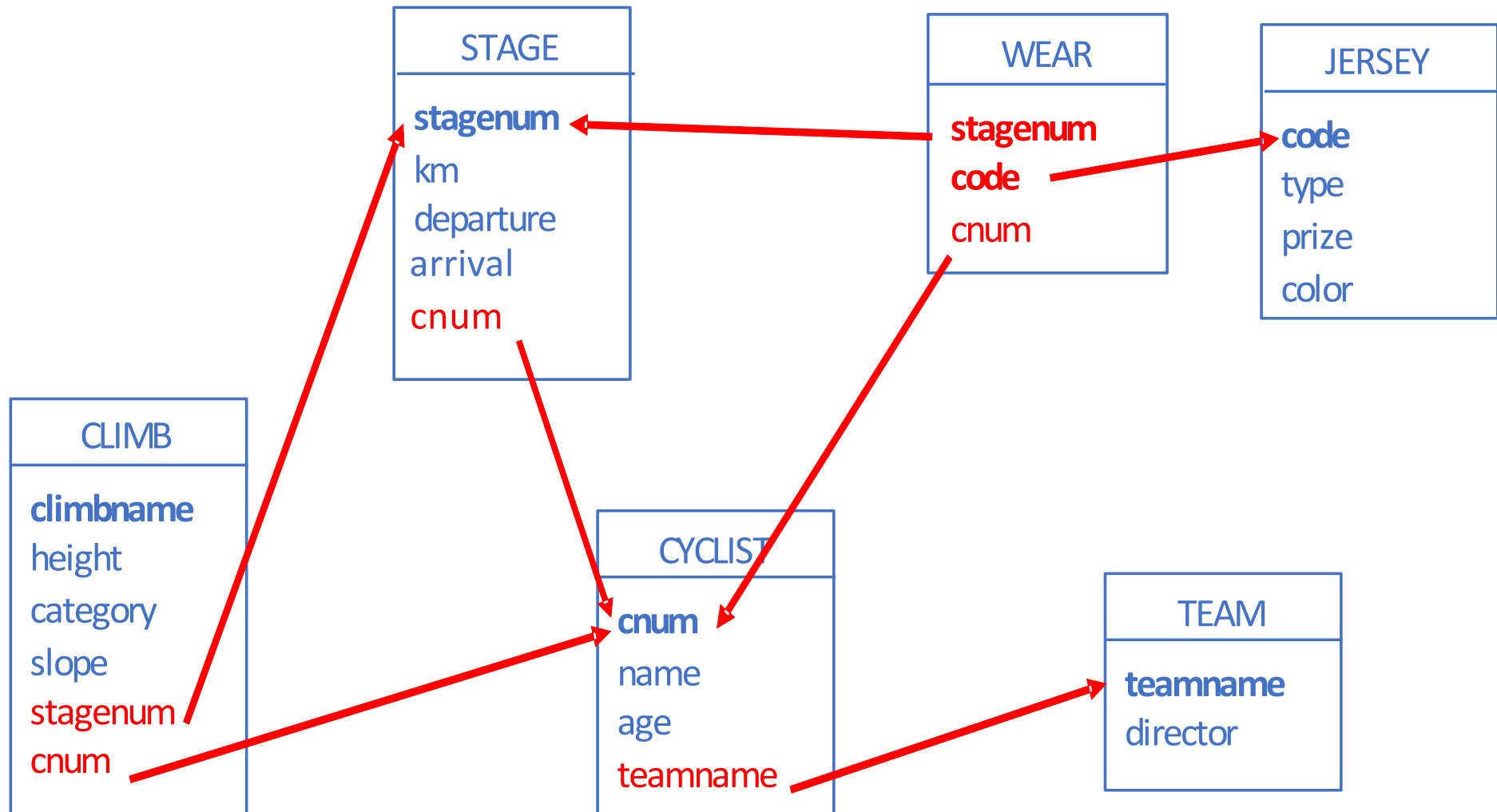
1. List the name of the youngest cyclist. (21)



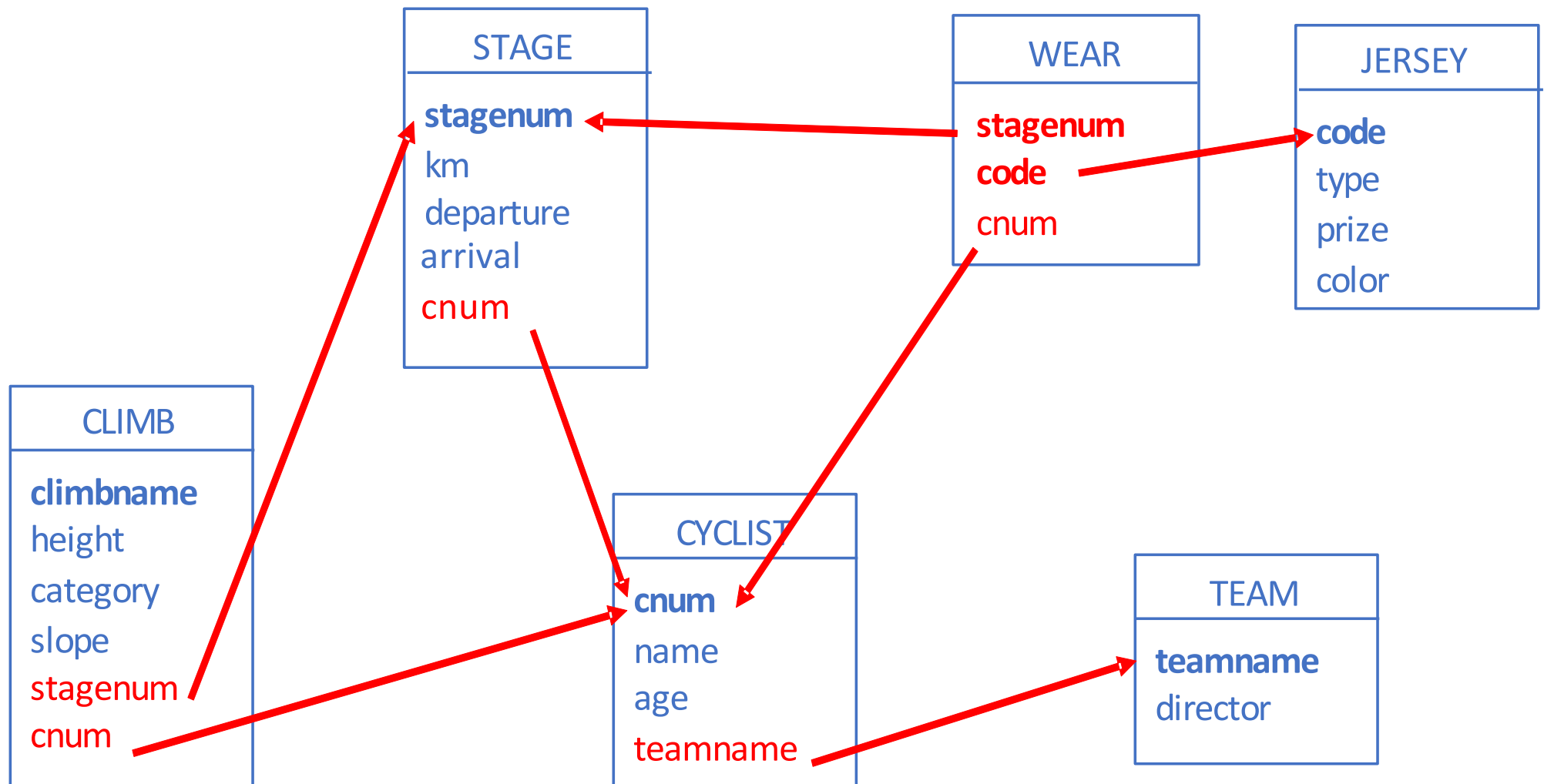
2. List the value of the attribute stagenum and the departure city for those stages with no climbs. (16)



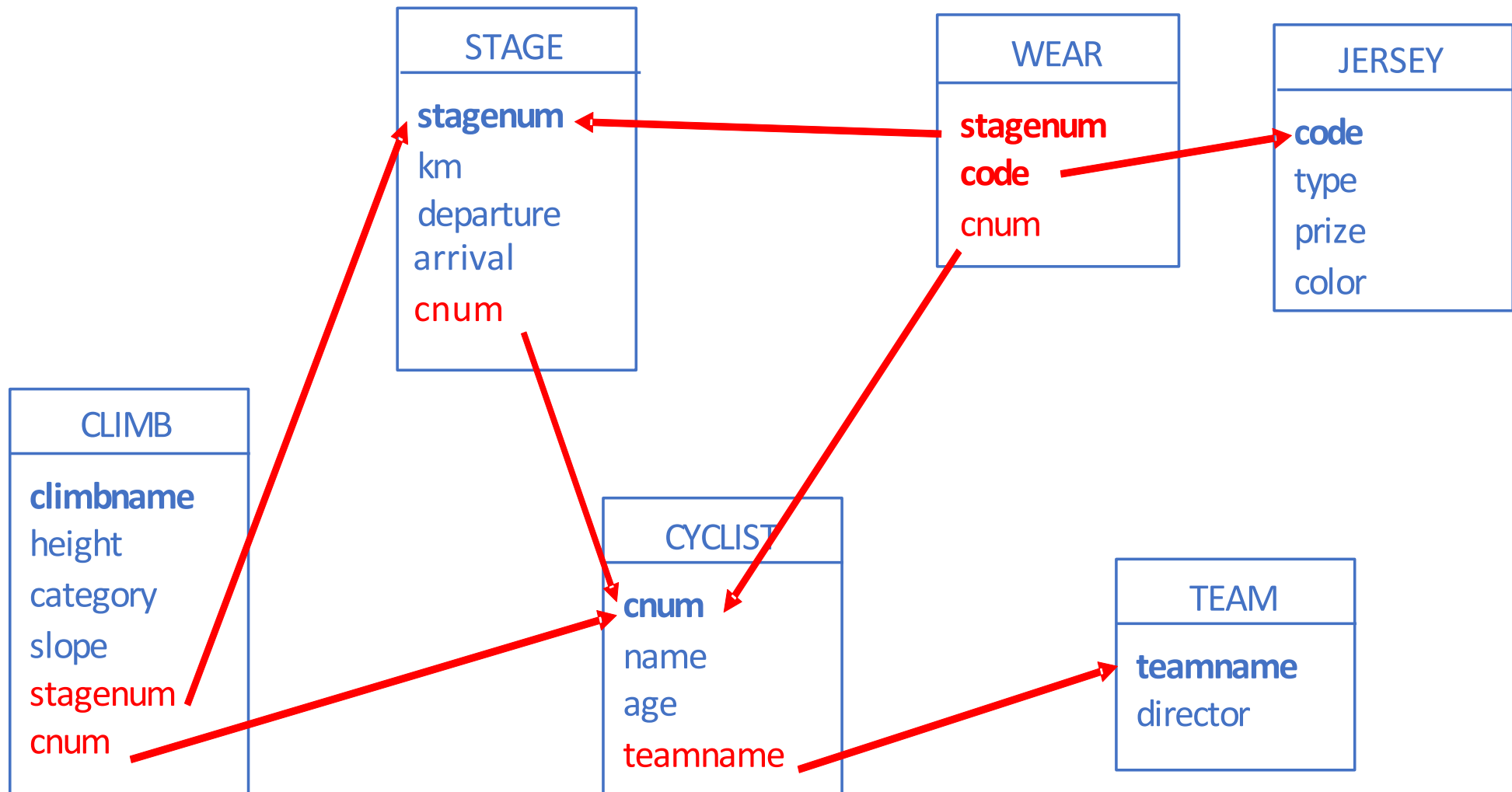
3. List the name of the departure and the arrival of the stages where the steepest climbs are located. (19)



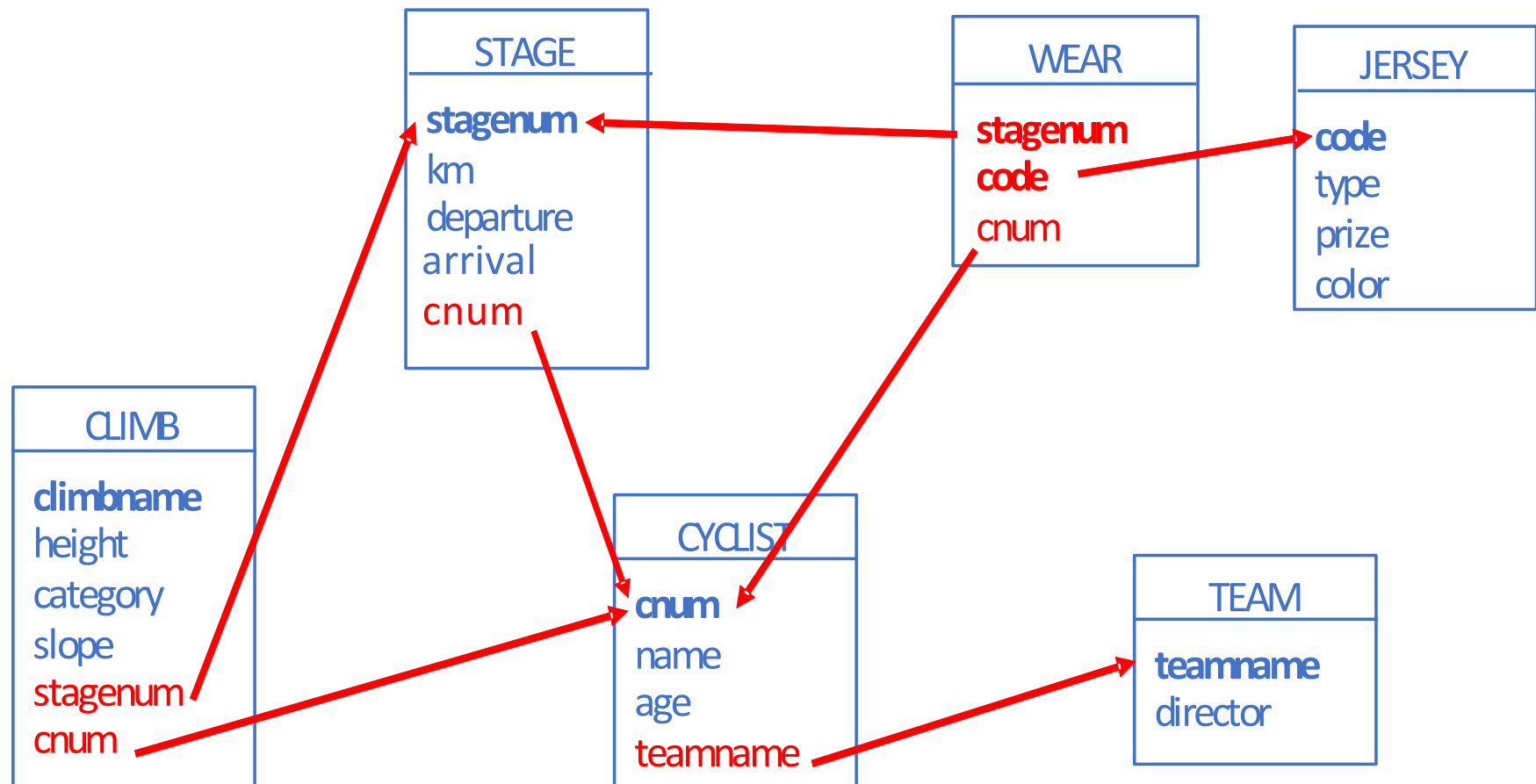
4. List the name of the cyclists who have won all the climbs in one stage and have also won the stage. (27)



5. List the code and the color of those jerseys which have only been worn by cyclists of the same team. (29)

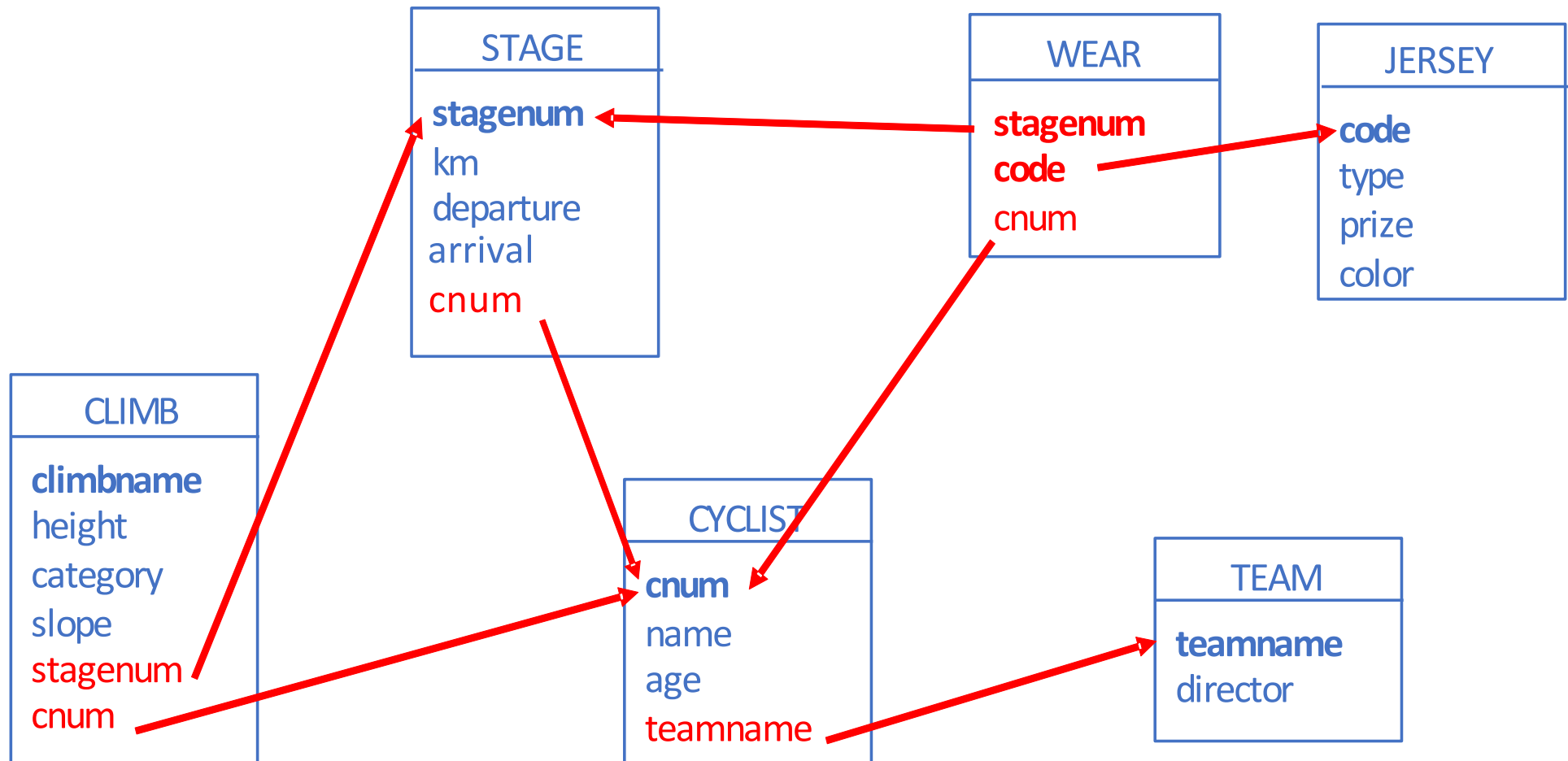


6. List the name of the cyclists who belong to a team which has more than five cyclists and have also won one or more stages. Please also indicate how many stages he has won.



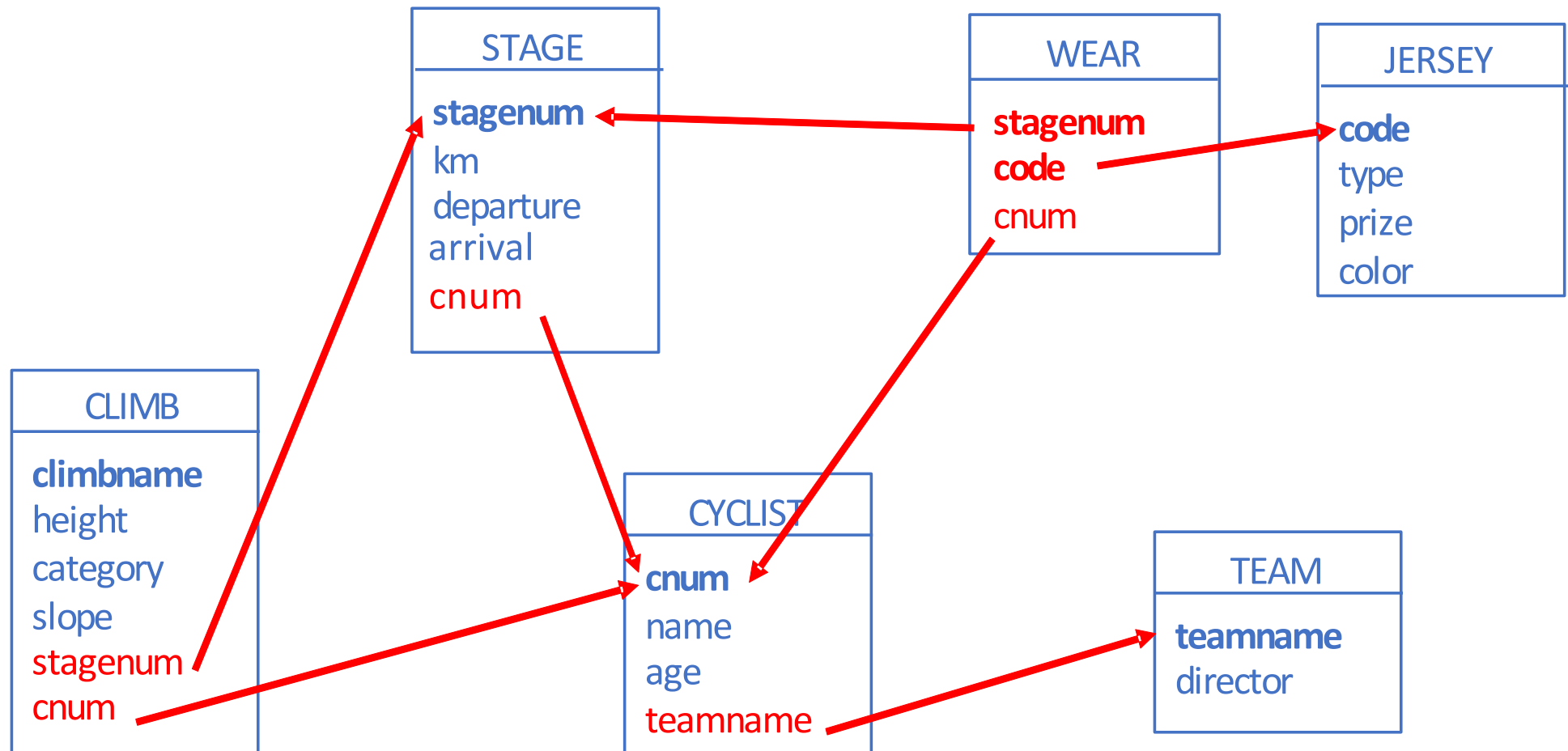
See the difference with the previous one !

7. List the name of all the cyclists who belong to a team which has more than five cyclists indicating how many stages he has won.

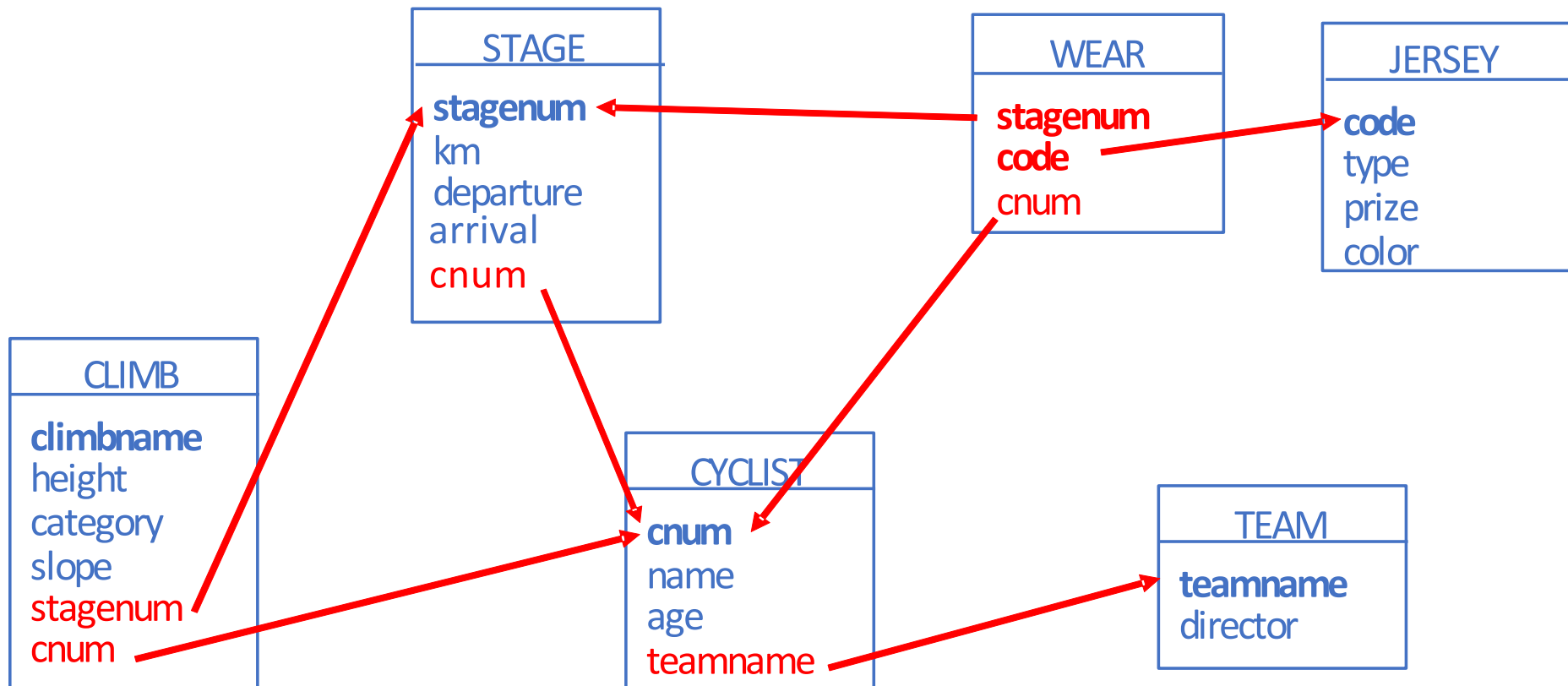


Cyclist C such as there is some jersey worn by the 20 but not by the cyclist C

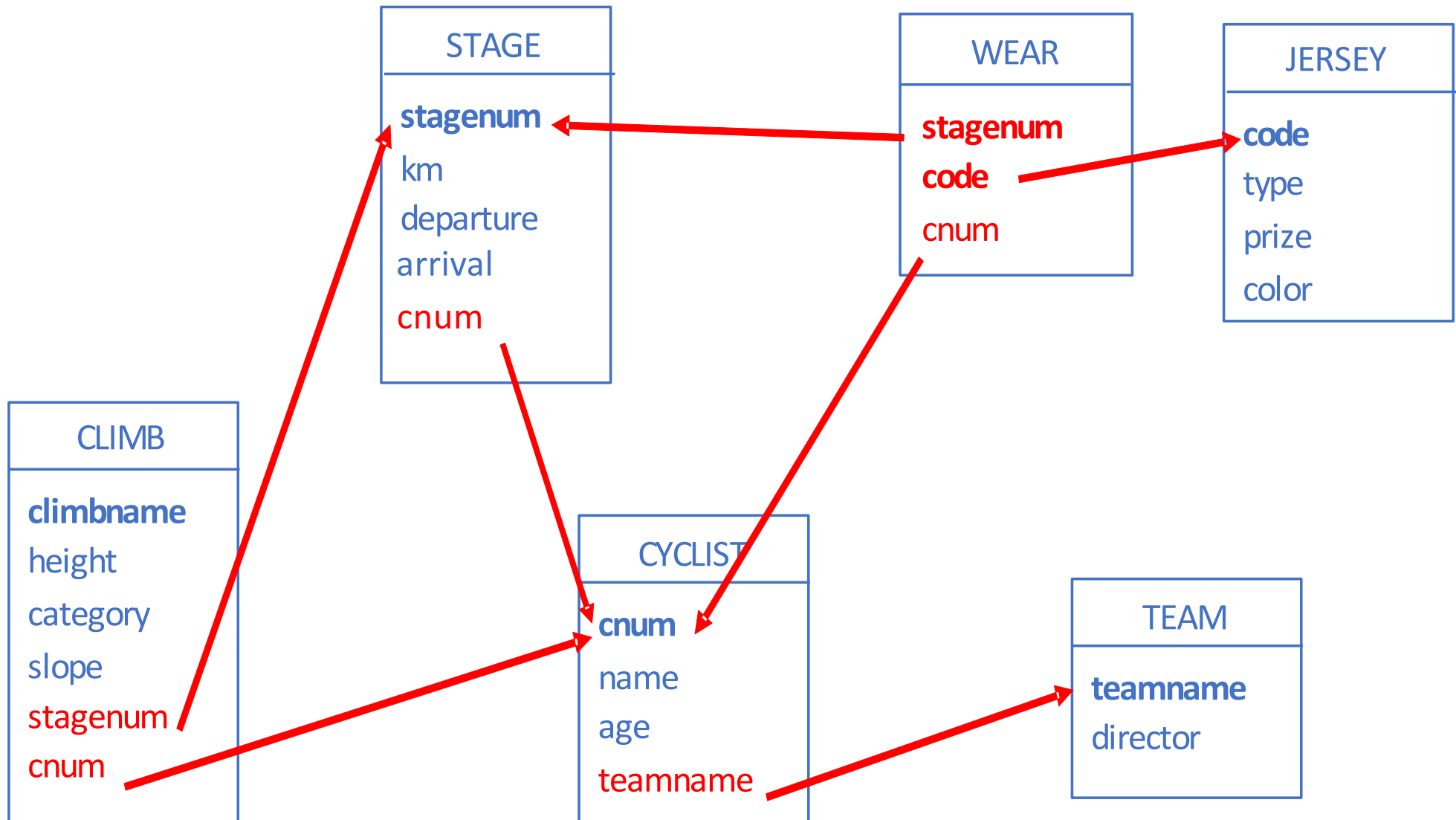
8. List the cyclist number and the name of the cyclists who have not worn all the jerseys worn by the cyclist with number 20. (40)



9. List the name of the teams and the average age of the cyclists of those teams who have the highest average age of all the teams. (36)



10. List the name of those teams such that their cyclists have only won climbs of category = 1. (30)



Cycling race

