Database Architecture

Practical Assessment #2 (PR2):

XML Extension

Students: *Carlos Del Blanco Garcia*
Jordi Bericat Ruz
Professor: *Maria Teresa Bordas Garcia*
Term: *Autumn 2020-21 (Aula 1)*

Table of Contents

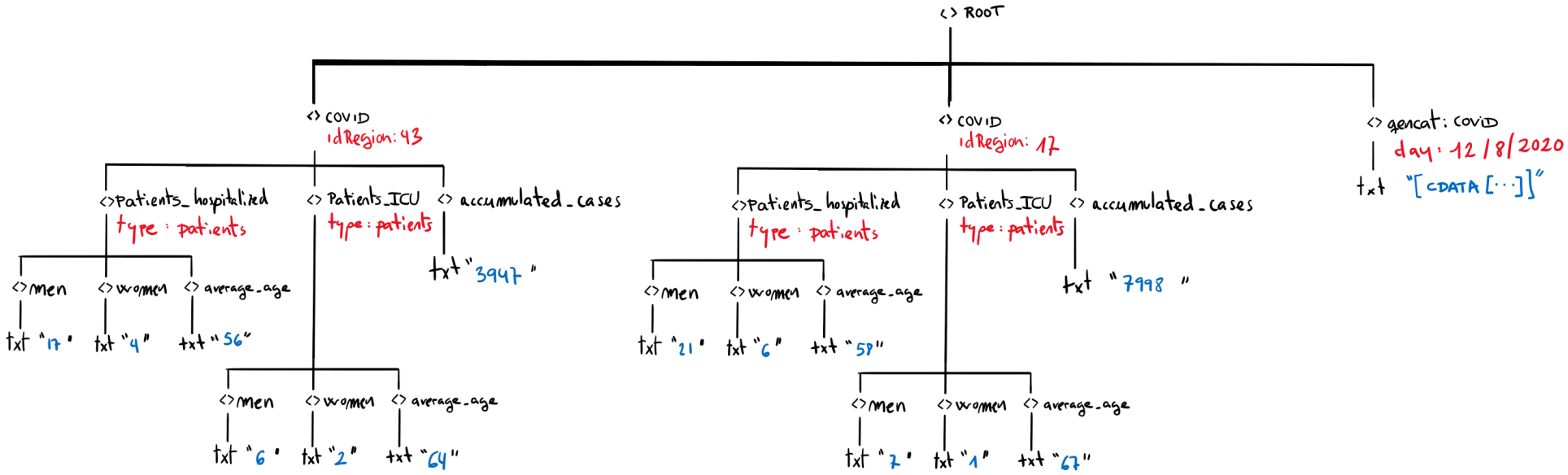
Activity 1 1
a) 1
b) 4
Activity 2 8
a) 8
b) 10
c) 13
Activity 3 14
a) 14
b) 16
Bibliography 21

Activity 1

a)

XML-tree Structure:

Before proceeding with the xml file creation itself, we thought that It would be a good practice to *graphically* model the xml structure given in the assessment statement as an xml-tree[1]. This way we can have a"one-sight" outlook of the whole xml structure and therefore make the later translation into xml code easier:



[1] *As suggested in bibliography [#1], page 42.*

XML Code:

```xml
<?xml version = "1.0" encoding = "ISO-8859-1" ?> <!-- See comments section [1] -->

<metadata <!-- See comments section [2] -->
    xmlns="http://www.uoc.edu/subjects/adb/ns" <!-- See comments section [3] -->
    xmlns:gencat="http://www.gencat.cat/dadesobertes/ns"> <!--comm. section [4]-->
    <COVID idregion="43">
        <patients_hospitalized type="patients">
            <men>17</men>
            <women>4</women>
            <average_age>56</average_age>
        </patients_hospitalized>
        <patients_ICU type="patients">
            <men>6</men>
            <women>2</women>
            <average_age>64</average_age>
        </patients_ICU>
        <accumulated_cases>3947</accumulated_cases>
    </COVID>
    <COVID idregion="17">
        <patients_hospitalized type="patients">
            <men>21</men>
            <women>6</women>
            <average_age>58</average_age>
        </patients_hospitalized>
        <patients_ICU type="patients">
            <men>7</men>
            <women>1</women>
            <average_age>67</average_age>
        </patients_ICU>
        <accumulated_cases>7998</accumulated_cases>
    </COVID>
    <gencat:COVID day="12/8/2020">  <!-- See comments section [5] -->
        <![CDATA[...]]>    <!-- See comments section [6] -->
    </gencat:COVID>
</metadata>
```

```
<!--

 ###############################################################################
 #                                                                             #
 #                                COMMENTS SECTION                             #
 #                                                                             #
 ###############################################################################

 [1] -> The prolog section of the xml document is the place where we can specify
        the encoding type (which in our case must be "ISO-8859-1").

 [2] -> I do not really know if we should use the keyword "root" instead of
        "metadata" to define the xml's tree root (most likely it does not matter,
        but better ask it in the forum to be in the safe side).

 [3] -> to set the default namespace we use the "xmlns" attribute.

 [4] -> we define a specific namespace alias (gencat) to avoid name clashes between
        the two elements with identical identification (COVID) but (probably)
        different application data structure.

 [5] -> Here we specify the alias "gencat" to this xml element (COVID) in order
        to refer to the custom namespace provided in the activity statement. This
        is required for the sake of properly integrate the external COVID xml data
        into our xml, due to the fact that this external data MIGHT NOT HAVE the
        very same structure than the defined in the default namespace, from which
        we set the default predefined xml dictionary for this xml document.

 [6] -> the structure of the external "not-xml" data is not provided, so we
        cannot define it (we use the [...] placeholder instead).
        (maybe we have to add a reference to the "summary of the latest data")

-->
```

b)

The main purpose of the extensive markup language (xml) is to stablish a proper communication mechanism among applications. To achieve this, it is necessary to strictly define a structure of elements (known as vocabulary) which implies a set rules and constraint. Here is where xml schemas come into play, since they allow defining that so-called vocabulary with a very high degree of details regarding the application data particularities.

That said, to define the xml schema that will stablish the required vocabulary and set of rules for the xml structure proposed in this activity statement, we will proceed as follows:

```
<!-- ####################  definition of the xml schema ##################### -->

<?xml version = "1.0" encoding = "ISO-8859-1" ?>

<!-- see comments section [0] -->

<xsd:schema> <!-- see comments section [1] -->


<!-- ####################  definition of simple elements  ################## -->

<xsd:element name="id_type"> <!--see comments section [2.1.3]-->
    <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
            <xsd:maxInclusive value="9999"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
<xs:element name="year" type="xs:integer"/> <!--see comments section [2.2.1]-->
<xs:element name="month" type="xs:integer"/> <!--see comments section [2.2.1]-->
<xs:element name="day" type="xs:integer"/> <!--see comments section [2.2.1]-->
<xs:element name="idRegion" type="xs:integer"/> <!--see comments section [2.3.1]-
->
<xs:element name="description" type="xs:string"/> <!--see comm. section [2.3.2]--
>
<xs:element name="hospitalized" type="xs:integer"/> <!--see comm. section[2.4.1]-
->
<xs:element name="ICU" type="xs:integer"/> <!-- see comments section [2.4.2]-->
<xsd:element name="gender"> <!--see comments section [2.4.3]-->
    <xsd:simpleType>
        <xsd:restriction base="xsd:NMTOKEN">
            <xsd:enumeration value="female" />
            <xsd:enumeration value="male" />
            <xsd:enumeration value="other" />
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

```
<!-- ######################    definition of attributes    ###################### -->

<xs:attribute name="id" type="xs:id_type"/> <!-- see comments section [2.1.3] -->


<!-- ###################### definition of complex elements ###################### -->

<xsd:element name="date"> <!-- see comments section [2.2] -->
    <xsd:complexType>
        <xsd:sequence>
          <xs:element ref="year"/>
          <xs:element ref="month"/>
          <xs:element ref="day"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="region"> <!-- see comments section [2.3] -->
    <xsd:complexType>
        <xsd:sequence>
          <xs:element ref="idRegion"/>
          <xs:element ref="description"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="patients"> <!-- see comments section [2.4] -->
    <xsd:complexType>
        <xsd:sequence>
            <xs:element ref="hospitalised"/>
            <xs:element ref="ICU"/>
            <xs:element ref="gender"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>


<!-- ######################          root element          ###################### -->

<xs:element name="COVID"> <!-- see comments section [2.1.1] -->
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="date"/>
      <xs:element ref="region"/>
      <xs:element ref="patients" maxOccurs="10"/>
    </xs:sequence>
    <xs:attribute ref="id" use="required"/> <!--comm. sections [2.1.2] & [2.1.3]-->
  </xs:complexType>
</xs:element>

<!—xml schema definition end -->

</xs:schema>
```

```
<!--

 ##############################################################################
 #                                                                            #
 #                              COMMENTS SECTION                              #
 #                                                                            #
 ##############################################################################

[0] -> First, we should decide on a design method to define the xml schema.
       As seen at Bibliography [#2], there are three available approaches:

       1- "Simplest-yet-messy" approach: This way, "to create the schema we could
          simply follow the structure in the XML document and define each element
          as we find it". (footnote: literal citation from Bibliography [#2],
          section: "Create an XML Schema").

       2- "Divided Schema" approach: "The next design method is based on defining
          all elements and attributes first, and then referring to them using the
          ref attribute" (footnote: literal citation from Bibliography [#2],
          section: "Divide the Schema"). This way gets easier to read and
          maintain the xml code in complex structures.

       3- "Use of Named Types" approach: "The third design method defines classes
          or types, that enables us to reuse element definitions"(footnote: literal
          citation from Bibliography [#2], section: "Named Types").

       After analyzing the characteristics of each of all three approaches, we
       decided to use the 2nd one (Divide the Schema) since we won't be upgrading
       the xml structure (the assessment statement says nothing about scalability
       and reusability of types), but at the same time we wanted to design a
       readable (it has to be assessed) as well as maintainable xml schema (so both
       team members / students who participate on its elaboration can better
       understand and eventually improve it).

[1] -> The activity statement does not give any information about namespaces
       (xmlns), hence we neither include any reference to it in the schema root
       element declaration, nor to the "targetNamespace" attribute (the XML Schema
       will be assigned to the NULL namespace).

[2] -> NOTES ABOUT ELEMENTS & ATTRIBUTES DEFINITIONS:

  [2.1] -> NODE "COVID":

    [2.1.1] -> The "COVID" node will be the root element of this xml schema.
               On the other hand, we'll be considering that its sub-elements will
               be appearing in the same order on the instance xml documents.
               Therefore, will have to use the primitive "sequence" in the
               "COVID" element definition.

    [2.1.2] -> ATTRIBUTE "id" ("is mandatory" constraint): it must be declared
               as a mandatory attribute since it contains a "foreign key" value.
               To do so, we must explicitly declare it with the "use"
               attribute, as well as the "required" attribute (references:
               Bibliography [#1], page 49 & Bibliography [#3]).

    [2.1.3] -> ATTRIBUTE "id" ("1-9999" range constraint): On the other hand,
               this attribute needs to be restricted to values in between 1-9999,
               thus, we also might have to declare this integer attribute as a
               simpleType and then apply a "range" restriction to its values by
               means of the "minInclusive" & "maxInclusive" attributes
               (references: Bibliography [#1], page 52).
```

[2.2] -> NODE "date": Must be a complex type defined as a SEQUENCE (references: Bibliography [#1], page 48).

[2.2.1] -> ELEMENTS "year", "month" & "day" (in that order): We could define these as simple type sub-elements (derived from integer built-in types) of the complex type element "date". This way we make sure these three elements contain an Integer value (and nothing else).

[2.3] -> NODE "region": The activity statement does not stablish any constraint about this element, hence we can simply declare it as a complexType with no constraints at all.

[2.3.1] -> ELEMENT "idRegion": "integer" built-in simple type (references: Bibliography [#1], page 50)

[2.3.2] -> ELEMENT "description": "string" built-in simple type (references: Bibliography [#1], page 50)

[2.4] -> NODE "patients": Since this node can only be repeated as much as 10 times, we must stablish the cardinality of this complexType element by means of the minOccurs and maxOccurs attributes (references: Bibliography [#1], page 49).

[2.4.1] -> ELEMENT "hospitalized": "integer" built-in simple type (references: Bibliography [#1], page 50)

[2.4.2] -> ELEMENT "ICU": "integer" built-in simple type (references: Bibliography [#1], page 50)

[2.4.3] -> ELEMENT "gender": Must be a simple Type derived from an NTOKEN existing built-in type defined as an ENUMERATION (references: Bibliography [#1], page 48)

Activity 2

a)

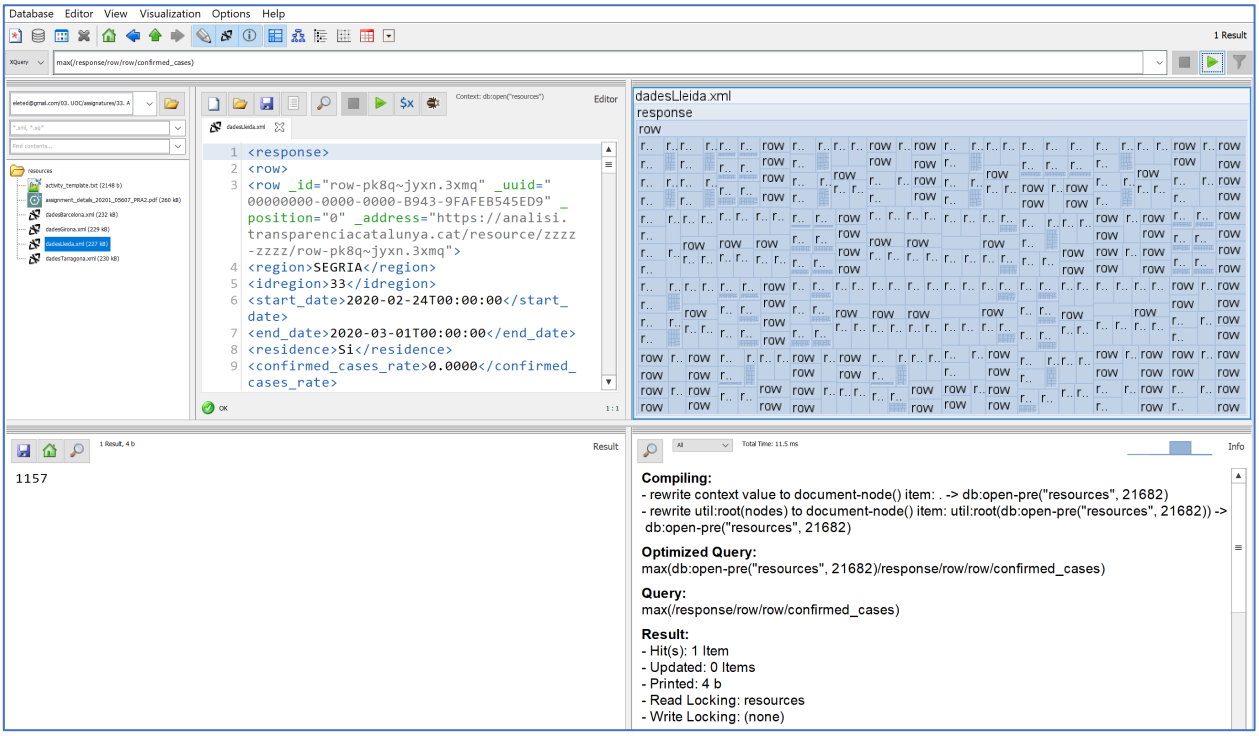
To define an *xPath* expression that complies with the constraints stablished on this activity statement, we came-up with the following strategy:

STEP 1 → DEFINE A XPATH SUB-QUERY

First, we find a way to retrieve the "*confirmed_cases*" node that contains the maximum[2] value among all repetitions of the same element in the provided xml:

```
max(/response/row/row/confirmed_cases)
```

If we run the above's *xPath* expression, the only result we got is actually the correct "*confirmed_cases*" node with maximum value:



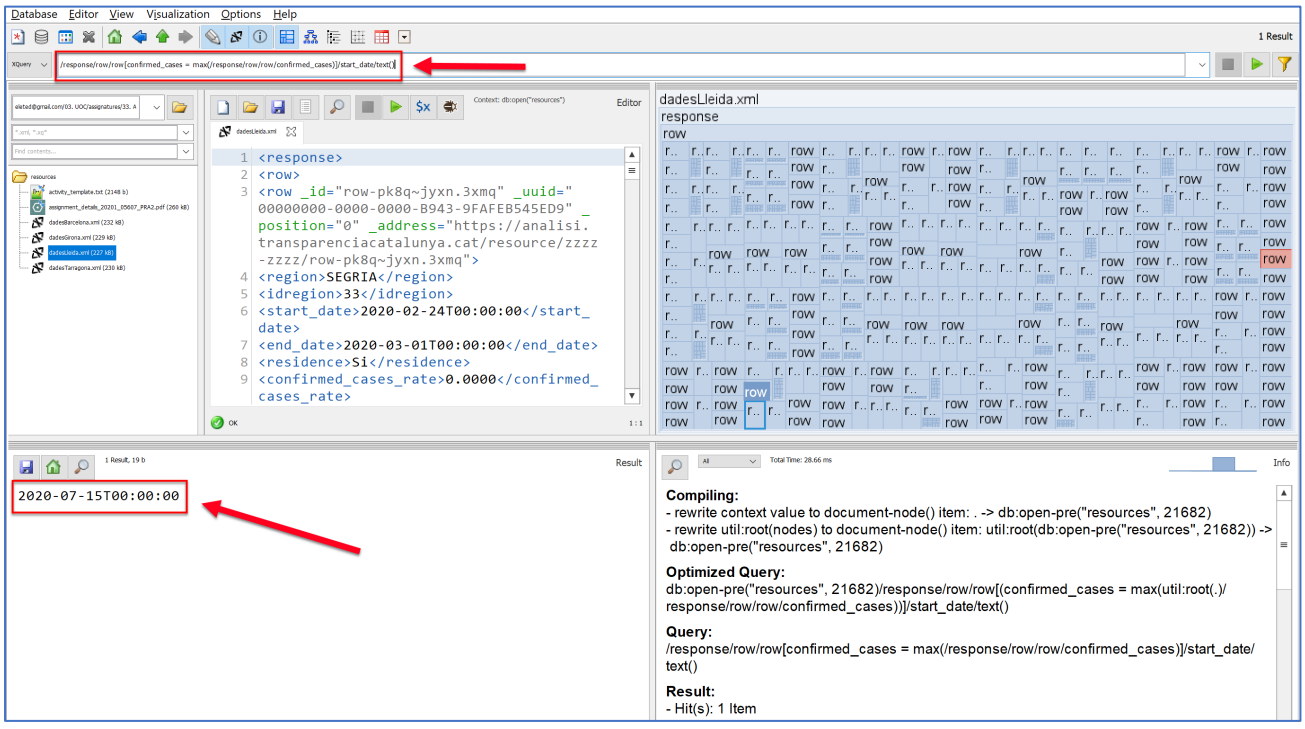
[2] *The xPath 2.0 version includes the **max()** built-in function which allowed us to select the maximum value of an element of the hierarchy, compared with other repetitions of the same element (references: Bibliography [#4]). However, In order to run xPath 2.0 expressions, we also had to download the newest version of the "BaseX" xml processor from Bibliography [#5], (the one provided in the subject's resources area was unstable in our systems).*

STEP 2 → XPATH MAIN EXPRESSION

On this step, first we should stablish the direction on which the evaluation is going to proceed by setting the *axe*[3] (**green**) in the path expression. In our scenario, we will be going down in the hierarchy (from top) to its 4th level (*/response/row/row/start_date/*). Afterwards, we just have to tweak the *predicate* expression (**red** + **pink**) in order to retrieve the element that matches the xpath sub-query (**pink**) return value defined in the step #1 (that is; the maximum value contained in a "*confirmed_cases*" element). Finally, to exclude all the nodes selected by the *axe* but the *start-date* element, we will have to specify it in the expression's *node-test*[4] (**yellow**). Putting it all together we obtain the following *xPath* expression:

```
/response/row/row[confirmed_cases = max(/response/row/row/confirmed_cases)]/start_date/text()
```

Finally, if we run it in the xml processor, we get the requested information:



You will find the .txt file requested for this activity on the delivery zip file (*/activity_2/activity_2A.txt*)

[3] Bibliography [#1], page 57

[4] Bibliography [#1], page 59

b)

To simplify the process of defining the xPath expression requested in this activity, we are going to follow a "divide & conquer" strategy:

Step #1:

Right now, we will only retrieve the "*confirmed_cases*" element value corresponding to the 2nd previous element (row) to the one that contains the maximum "*confirmed_cases*" element value (we can achieve this simply by modifying the *xPath* expression defined in the activity 2-A). The difference is that we now include in our selection a set of data belonging to a previous xml element (from the evaluation point of view) by means of the **"preceding::row[2]"**[5] "*axe component*" (**blue**). More precisely, this axe allows us to go back two elements in the evaluation direction, starting from the element selected by the left part of the axe (**dark green**). Finally, we add to the final *xPath* expression axe (underlined) a last "*axe component*" (**light green**) to indicate the next level in the hierarchy we want to select ("*confirmed_cases*" node) along with the node-test (**yellow**) that allows us to retrieve only the value we need. Putting it all together we get the following *xPath* expression:

```
/response/row/row[confirmed_cases =
max(/response/row/row/confirmed_cases)]/preceding::row[2]/confirmed_cases/text()
```

Step #2:

At this point we have the means to individually obtain the necessary values to calculate the increment of confirmed cases between date ranges (that is, by using the *xPath* expressions we defined in the activity 2-A and the Step #1 on this activity). To implement this calculation, we can use the operators defined in the *xPath* specification[6] to calculate the increment of confirmed cases (minus operation). In "pseudo-code", what we are going to perform is the following subtract operation:

```
((xPath_from_activity2A)-(xPath_from_step#1))
```

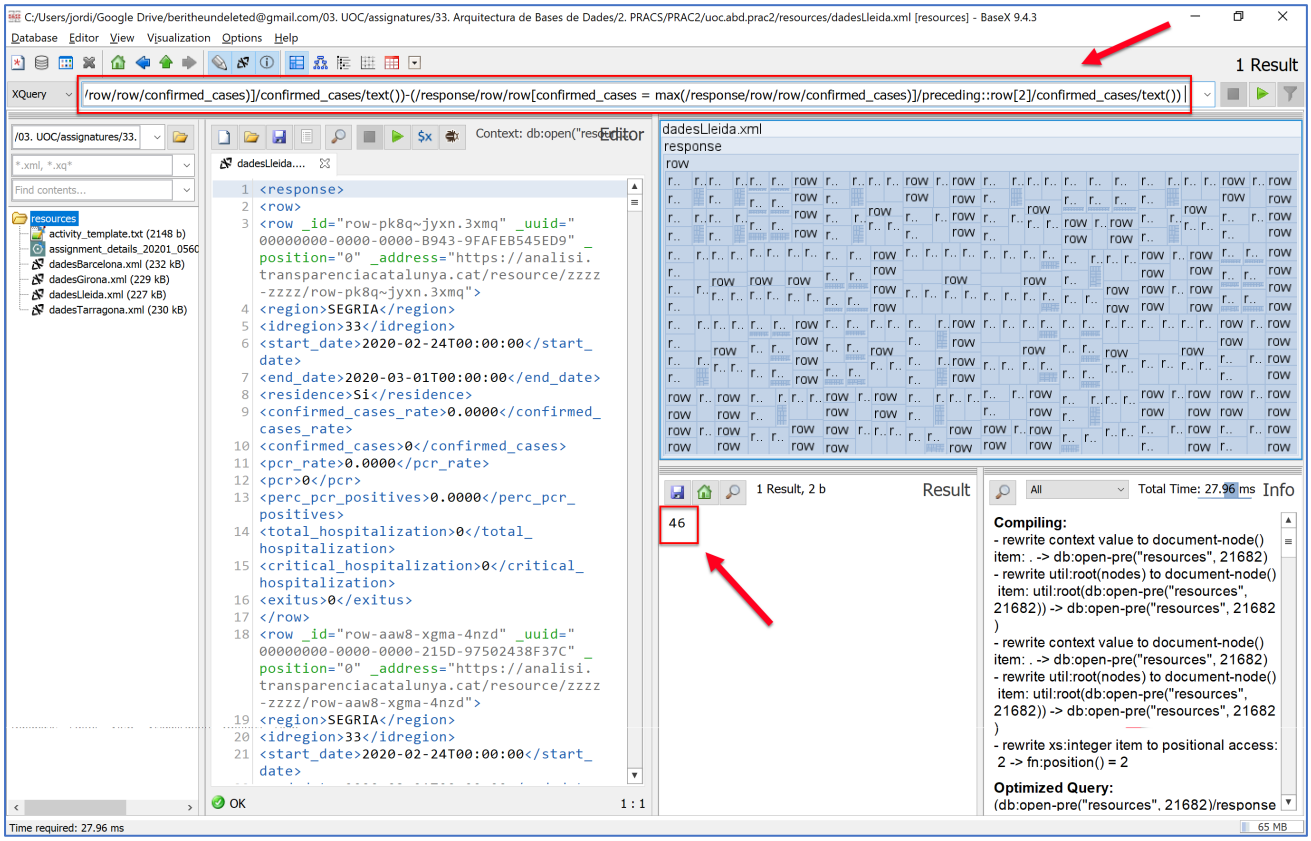
[5] *References: Bibliography [#6]*

[6] *References: Bibliography [#1], page 59*

In the above's template, replacing the *xPath* "placeholders" (**in color**) for the actual expressions will result in the following *xPath* "composite" expression:

```
(/response/row/row[confirmed_cases =
max(/response/row/row/confirmed_cases)]/confirmed_cases/text())-
(/response/row/row[confirmed_cases =
max(/response/row/row/confirmed_cases)]/preceding::row[2]/confirmed_cases/text
())
```

As we can see below, the result we obtain after running it is exactly what we were looking for:



Step #3:

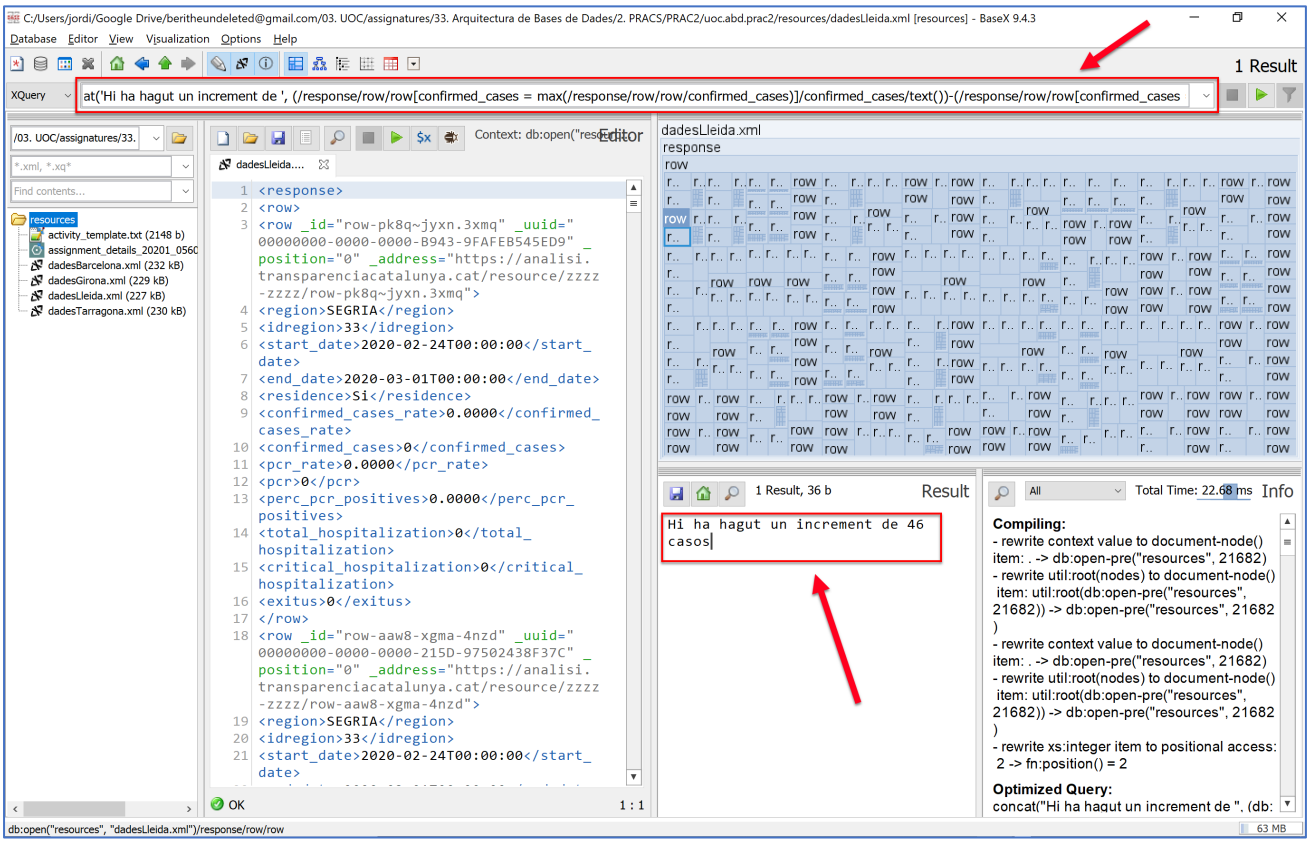
This step consists on using the built-in xPath *concat()* function to concatenate different xPath expressions to generate the output as requested in the assessment statement (we still do not include the xPath expression defined in the previous step, since we want to describe the whole process in a more detailed way):

```
concat('Hi ha hagut un increment de ',step-2_xPath_expression,' casos')
```

Finally, replacing the xPath *placeholder* (**pink**) for the expression defined in the step #2 will result in the *xPath* expression that gives us exactly the result requested on this assessment statement.

```
concat('Hi ha hagut un increment de ',(/response/row/row[confirmed_cases =
max(/response/row/row/confirmed_cases)]/confirmed_cases/text())-
(/response/row/row[confirmed_cases =
max(/response/row/row/confirmed_cases)]/preceding::row[2]/confirmed_cases/text
()),' casos')
```

If we run the xPath expression in the XML processor we get the expected result:



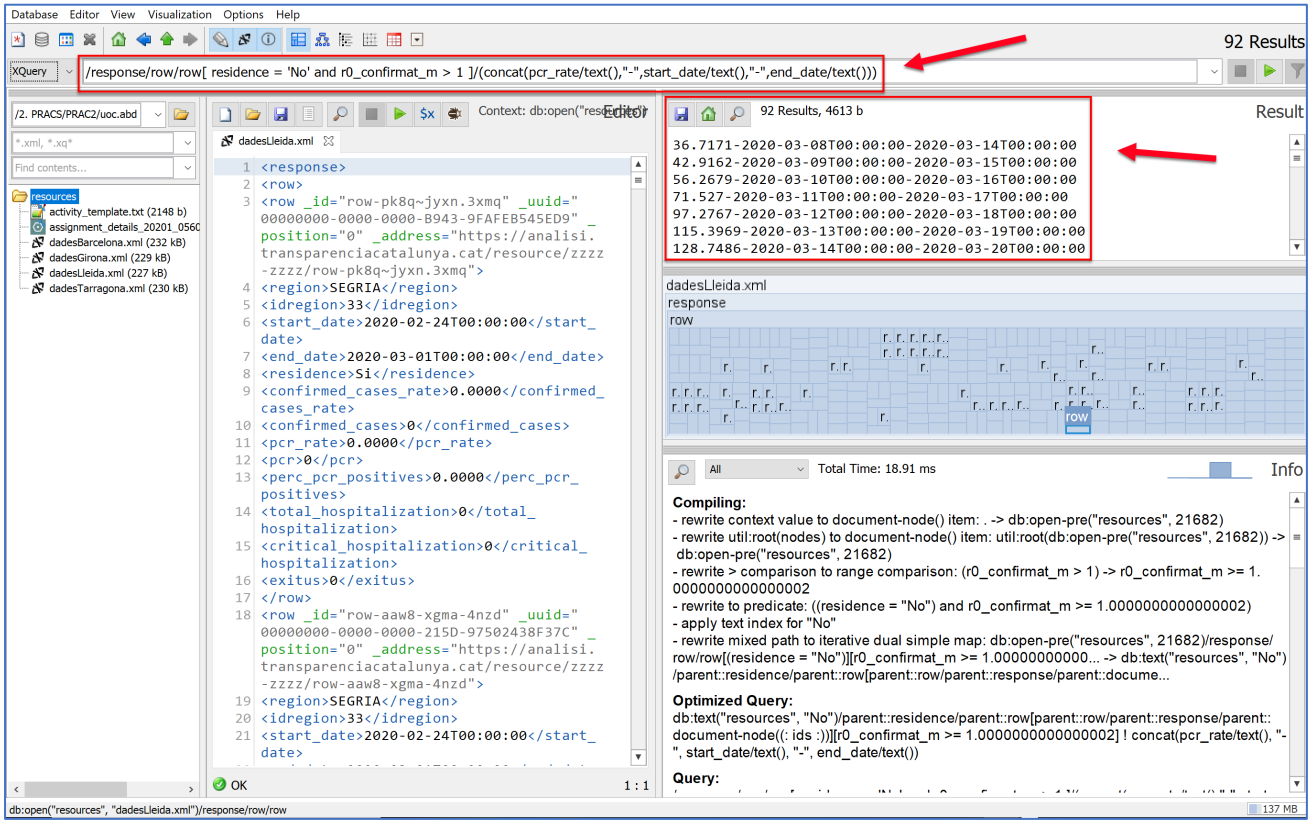
You will find the .txt file requested for this activity on the delivery zip file (*/activity_2/activity_2B.txt*)

c)

In the same way we did in the previous activity, we can define an axe (**green**) that allows us to select all the elements contained in each inner “row” node, so we can operate with the whole set of selected elements. More precisely; this way we can define a “custom” node-test expression (**yellow**) that retrieves an union of element values under the same node. To do so, we use the *concat()* xPath built-in string function along with every element axe + node-test selection we want to include in the union (underlined). The predicate (**light green**) restricts the results to this activity’s statement specifications:

```
/response/row/row[residence = 'No' and r0_confirmat_m > 1 ]
/(concat(r0_confirmat_m/text(),"-",start_date/text(),"-",end_date/text()))
```

As we can see below, if we run the avobe’s xPath expression in our XML processor, we get the expected results:



You will find the .txt file requested for this activity on the delivery zip file (*/activity_2/activity_2C.txt*)

Activity 3

a)

On the one hand, there are five clauses under the acronym FLWOR:

- F stands for “FOR”
- L stands for “LET”
- W stands for “Order By”
- R stands for “Return”

On the other hand, though, a statement FLWOR can be composed with variable combinations of the above’s clauses, some of them being optional:

One or more optional for clauses that join one or more iterated variables with input sequences that can be XQuery expressions as well as XPath expressions; they can be node sequences or atomic valued sequences. The result is an iteration that binds a variable to each element returned by the expression.

Examples:

```
a) for $x in document("list.xml") ...
b) for $x in (1 to 10) ...
c) for $x at $i in doc("books.xml") ...
```

An optional let clause that declares and assigns a value to the variable given for the iteration that avoids repeating the same expression and as the clause for the assigned expression can be an XQuery expression as well as an XPath expression and can return a sequence of nodes or a sequence of atomic values; it is not, however, an iteration clause.

Example:

```
a)  let $x := (1 to 5)
```

An optional where clause that applies a filter to the iteration, specifies one or more criteria for the result, if it appears to be only one time, of all the for and let clauses.

Example:

```
a)  where $x/preu>100 and $x/preu< 150
```

An optional order by clause that specifies the type of order to be followed from the result, if not specified, the result will be ordered according to the clauses defined previously.

Example:

```
a)  order by $x/@category, $x/preu
```

A return clause: each xQuery expression has one and it has to be the last one, it specifies what the result is.

Example:

```
a)  return $x/preu
```

b)

Next you will find our *xQuery* sentence proposal. The needed steps to obtain the results requested on this activity sentence are properly documented below in the form of *xQuery comments (**green**)*:

```
(:#########################################################################

 STEP 1 -> First we define a variable set to refer the xml databases:

#########################################################################:)

 declare variable $docLleida := doc("dadesLleida.xml");
 declare variable $docTarragona := doc("dadesTarragona.xml");
 declare variable $docGirona := doc("dadesGirona.xml");
 declare variable $docBarcelona := doc("dadesBarcelona.xml");

(:#########################################################################

 STEP 2 -> Now we only retrieve the maximum value among all the
           "confirmed_cases" elements in the "dadesLleida.xml" DB, so later we
           could retrieve the related "start_date" element value and then
           filter the results retrieved from the other xml databases using
           this value as a key (that is, we retrieve the "start_date" element).
           We achieve this by means of assigning the value returned by the
           max() function into a variable using the "let" clause:

#########################################################################:)

 let $max_LLD := max($docLleida/response/row/row/confirmed_cases)

(:#########################################################################

 STEP 3 -> Next, we select the parent element (using the "for" clause) that
           has the "confirmed_cases" maximum value (condition defined in the
           "where" clause) from the "dadesLleida.xml" DB.

#########################################################################:)

 for $Agrupated_LLD in $docLleida /response/row/row
 where $Agrupated_LLD/confirmed_cases = $max_LLD

(:#########################################################################

 STEP 4 -> Now, using the data from the node selected (one) in the previous
           step using the "for" clause, we can set an xPath predicate filter
           to retrieve ALL the nodes (xml data can be segmented) that contain
           the same value (start_date) in their respective regional XML DB. We
           can do this using the "let" clause, as follows:

#########################################################################:)

 let $Agrupated_TGN := $docTarragona/response/row/row[start_date = $Agru-
pated_LLD/start_date]
 let $Agrupated_GIR := $docGirona/response/row/row[start_date = $Agru-
pated_LLD/start_date]
 let $Agrupated_BCN := $docBarcelona/response/row/row[start_date = $Agru-
pated_LLD/start_date]
```

```
(:#############################################################################

 STEP 5 -> Finally, we define the xQuery return clause, which generates an
           structured XML string including two main nodes or branches:
          (1) AgrupatedData and (2) IsolatedData.

          1. AgrupatedData: Here, from within the <ConfirmedCases> xml
                            subelement we retrieve (for each region) the nodes
                            we already stored in the $Agrupated_XXX variables.
                            However, since we only want the maximum value on a
                            single date (the same one as in the Lleida region
                            with max "confirmed_cases" values), we will have
                            to apply another predicate filter on the parent
                            node (so we can also retrieve the
                            "confirmed_cases_rate" value).

          2. IsolatedData:  we need to show for every region the five days
                            containing the highest confirmed_Cases without
                            considering data from the other region (it's
                            mandatory to get data containing Residence="No"),
                            so we can show the values "confirmed_cases" -
                            "start_date". In a more detailed way- In every
                            region (using for) we filter for every row node
                            which has residence value as "No" through the
                            clause "where" then as we have to provide the 5
                            highest confirmed cases we first sort the output
                            using the clause "order by" and finally we just
                            take the first five using "count" reaching 5.
                            From here we return the information using the
                            requested output format, showing the confirmed
                            cases and its "start_date" from the own region.

#############################################################################:)
```

1. AgrupatedData:

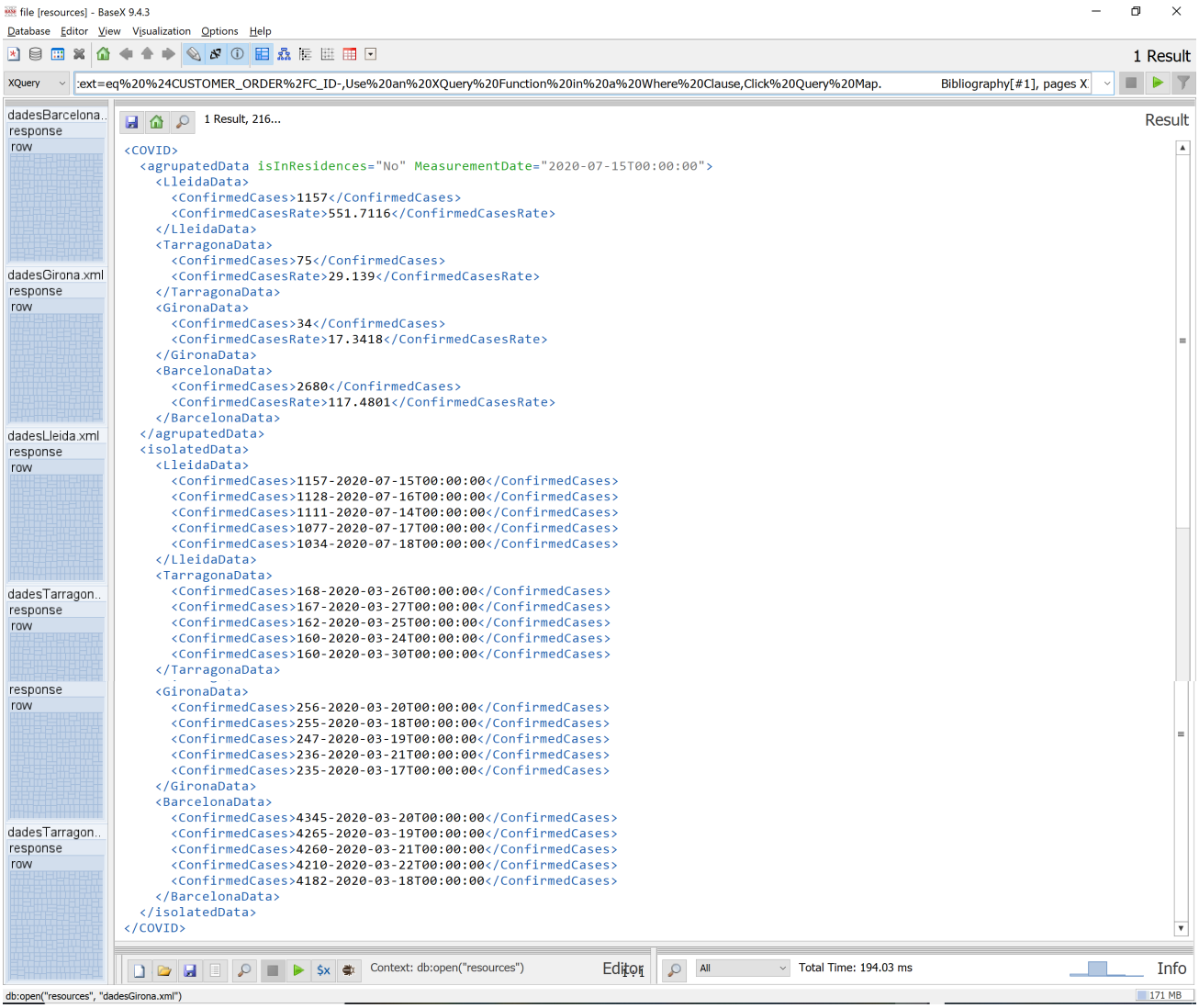
```
return
 <COVID>
    <agrupatedData isInResidences="{$Agrupated_LLD/residence/text()}" MeasurementDate="{$Agrupated_LLD/start_date/text()}">
        <LleidaData>
            <ConfirmedCases>{$Agrupated_LLD/confirmed_cases/text()}</ConfirmedCases>
            <ConfirmedCasesRate>{$Agrupated_LLD/confirmed_cases_rate/text()}</ConfirmedCasesRate>
        </LleidaData>
        <TarragonaData>
            <ConfirmedCases>{$Agrupated_TGN[confirmed_cases = max($Agrupated_TGN/confirmed_cases)]/confirmed_cases/text()}</ConfirmedCases>
            <ConfirmedCasesRate>{$Agrupated_TGN[confirmed_cases = max($Agrupated_TGN/confirmed_cases)]/confirmed_cases_rate/text()}</ConfirmedCasesRate>
        </TarragonaData>
        <GironaData>
            <ConfirmedCases>{$Agrupated_GIR[confirmed_cases = max($Agrupated_GIR/confirmed_cases)]/confirmed_cases/text()}</ConfirmedCases>
            <ConfirmedCasesRate>{$Agrupated_GIR[confirmed_cases = max($Agrupated_GIR/confirmed_cases)]/confirmed_cases_rate/text()}</ConfirmedCasesRate>
        </GironaData>
        <BarcelonaData>
            <ConfirmedCases>{$Agrupated_BCN[confirmed_cases = max($Agrupated_BCN/confirmed_cases)]/confirmed_cases/text()}</ConfirmedCases>
            <ConfirmedCasesRate>{$Agrupated_BCN[confirmed_cases = max($Agrupated_BCN/confirmed_cases)]/confirmed_cases_rate/text()}</ConfirmedCasesRate>
        </BarcelonaData>
    </agrupatedData>

    (...)
```

2. IsolatedData:

```
    <isolatedData>
        <LleidaData>
            {for $Isolated_LLD in $docLleida//row
            where $Isolated_LLD/residence = 'No'
            order by xs:integer($Isolated_LLD/confirmed_cases) descending
            count $item
            where $item le 5
            return
            <ConfirmedCases>{$Isolated_LLD/confirmed_cases/text(), '-', $Isolated_LLD/start_date/text()}</ConfirmedCases>}
        </LleidaData>
        <TarragonaData>
            {for $Isolated_TGN in $docTarragona//row
            where $Isolated_TGN/residence = 'No'
            order by xs:integer($Isolated_TGN/confirmed_cases) descending
            count $item
            where $item le 5
            return
            <ConfirmedCases>{$Isolated_TGN/confirmed_cases/text(), '-', $Isolated_TGN/start_date/text()}</ConfirmedCases>}
        </TarragonaData>
        <GironaData>
            {for $Isolated_GIR in $docGirona//row
            where $Isolated_GIR/residence = 'No'
            order by xs:integer($Isolated_GIR/confirmed_cases) descending
            count $item
            where $item le 5
            return
            <ConfirmedCases>{$Isolated_GIR/confirmed_cases/text(), '-', $Isolated_GIR/start_date/text()}</ConfirmedCases>}
        </GironaData>
        <BarcelonaData>
            {for $Isolated_BCN in $docBarcelona//row
            where $Isolated_BCN/residence = 'No'
            order by xs:integer($Isolated_BCN/confirmed_cases) descending
            count $item
            where $item le 5
            return
            <ConfirmedCases>{$Isolated_BCN/confirmed_cases/text(), '-', $Isolated_BCN/start_date/text()}</ConfirmedCases>}
        </BarcelonaData>
    </isolatedData>
</COVID>
```

If we run the xQuery sentence on the xml processor we obtain the xml data as requested:



Side notes:

- Documentation references:
  - Bibliography [#1], page 60 (section 2.3.2)
  - Bibliography [#8], [#9], [#10], [#11], [#12]

- You will find the .txt file requested for this activity on the delivery zip file (*/activity3/activity_3B.txt*)

Bibliography

1. **UOC Resources →** *Databases Architectures Module 2: Relational Extensions*
2. **XML Schema Example at w3.org →** *https://www.w3schools.com/xml/schema_example.asp*
3. **XML Attributes definition examples →** *https://www.w3schools.com/xml/schema_simple_attributes.asp*
4. **Oracle xPath Reference (Number functions) →** https://docs.oracle.com/cd/E35413_01/doc.722/e35419/dev_xpath_functions.htm#autoId41
5. **BaseX XML Processor →** http://files.basex.org/releases/9.4.3/BaseX943.zip
6. **xPath expressions →** https://stackoverflow.com/questions/9857756/how-to-get-the-preceding-element
7. **Oracle xPath Reference (Functions on strings) →** https://docs.oracle.com/cd/E35413_01/doc.722/e35419/dev_xpath_functions.htm#autoId14
8. https://stackoverflow.com/questions/30206069/xquery-find-max-value-and-return-relevant-information
9. https://www.tutorialspoint.com/xquery/xquery_max.htm
10. https://www.w3schools.com/xml/xquery_functions.asp
11. http://edutechwiki.unige.ch/en/XQuery_tutorial_-_basics#:~:text=declare%20variable%20%24x%20%3A%3D%207.5,the%20datatype%20of%20the%20result.
12. https://docs.oracle.com/cd/E13162_01/odsi/docs10gr3/datasrvc/Add%20a%20Where%20Clause%20to%20a%20Query.html#:~:text=eq%20%24CUSTOMER_ORDER%2FC_ID-,Use%20an%20XQuery%20Function%20in%20a%20Where%20Clause,Click%20Query%20Map.