Report Project 3 : XQuery

Rémy Detobel & Nathan Liccardo

April 13, 2018

1 Introduction

This document aims to detail each choices and hypothesis we made during our implementation. As a reminder, we were assigned to implement three different XQuery programs. Each of those query use a part of the BDLP database (DBLP-excerpt). The structure of this report will be divided into three parts (one per program).

2 First XQuery program

For the first program, we were assigned (for each author) to return the number of co-authors and the number of joint publications with each of them. To realise this, we start by iterate on each author as below :

- ```
  for $author in //author
  return <author>
      ....
  </author>
  ```

Which will create an `<author> ... </author>` bloc for each author. Inside each of these blocs, we will find following informations (in sequence).

2.1 Author name and Co-authors informations

The author name and the number of co-authors are respectively obtained using `data` and `count` functions (defined by XQuery). Below instructions are formal definitions of `name` and `coauthors` tags :

- `<name>{data($author)}</name>`
- `<coauthors number="{count(//*[author=$author]/author)-1}"> ... </coauthors>`

2.1.1 Co-authors informations

For each co-author, we must obtain his name and the number of joint publications. Both informations will be contained inside `<coauthor> ... </coauthor>` tags (itself contained in the `coauthors` bloc). Each of the `coauthor` blocs are created by iterating on co-authors as below :

- ```
  for $coauthor in //*[author=$author]/author[not(.=$author)]
  return <coauthor>
  ....
  </coauthor>
  ```

Co-author name As for the author name, co-author name is reached using the `data` function on the `coauthor` variable. This is realised as follow :

- `<name>{data($coauthor)}</name>`

Joint publications For each co-author, we must retrieve the number of joint publications. Once again, this is achieved using the `count` function available in XQuery. Here is the final instruction :

- `<nb_joint_pubs>{count(//*[author=$author]/author[.=$coauthor])}</nb_joint_pubs>`

3 Second XQuery program

For the second request, we were assigned to give (for each proceedings) its title and titles of articles appearing in it. Each `proceeding` blocs are created by the use of an iteration :

- ```
  for $proceeding in //proceedings
      return <proceedings>
      ....
      </proceedings>
  ```

For each of these blocs, we must retrieve the title (`proc_title`) and a list of articles (`title`). This is achieved using the value of the `key` attribute (inside `proceedings` tags) and the value contained inside `crossref` tags. Regarding to the title, we obtain the below instruction :

- `<proc_title>{data($proceeding/title)}</proc_title>`

Finally, we can create the list of titles using the `key` value :

- ```
  for $inproceeding in //inproceedings[crossref=data($proceeding/@key)]
  return <title>{data($inproceeding/title)}</title>
  ```

4 Third XQuery program

For the last program, we were assigned to compute (for each pair of authors x and y, $x \neq y$) the distance between x and y. Before explaining the final request, we will describe how we use functions to implement complex operations. As there exist two complex operations, we have decided to implement two different functions : `generateTag` and `getTree`.

4.1 First function : `generateTag`

This first function is simply used to reach a well formatted XML tag. This means that the function takes an `$author`, a `$coauthor` and a `$distance` as input and return the below XML tag :

- `<distance author1="{$coAuthor}" author2="{$author}" distance="{$distance}" />`

4.2 Second function : `getTree`

Previous function only treat about output format. This means that we must, at this point, compute the distance value between two authors. This computation is realised by the `getTree` function which takes as input an `$author`, an `intervalAuthor`, a `distance` and an `allContext` (context variable). We have been decided to implement this function using recursive calls :

- The definition of the current distance (incremental value) :
  - `let $newDistance := $distance + 1`
- An iteration on all co-authors. `allContext` is a constant variable defined in the final request which is used to keep (in memory) whole author nodes :
  - `for $coAuthor in distinct-values($allContext[author=$intervalAuthor]/author)`

- On the previous iteration, we only want `coauthor` values which are, at the same time, different from `author` and `intervalAuthor` values. This is checked using the below instruction :
  - `where $coAuthor != $author and $coAuthor != $intervalAuthor`
- Finally, inside this iteration we call two functions. The first one is the recursive call to `getTree` and the second one is the inclusion of the current result into the final set using `generateTree` :
  - ```
    return (custom:getTree($author, $coAuthor, $newDistance, $allContext except
                           $allContext[author=$intervalAuthor]),
            custom:generateTag($author, $coAuthor, $newDistance))
    ```

4.3 Final instruction

Finally, we must create the `<distance> .... </distance>` bloc which contain each of the `<distance>` tags. Those tags are created using the first function and added to the final set using the second one. First call to of the recursive function is realised for each different authors in the dblp database. This means that we use the following code inside the `<distance> .... </distance>` bloc :

- ```
  for $author in distinct-values(//author)
      return custom:getTree($author, $author, 0, //*)
  ```

Which is used to initialise each of the `getTree` parameters. Remarks also that distance is set to 0 due to the fact that we start by using the same author and co-author (not added to the final set).