Applying Database Optimization Techniques on XQuery

CSE 232B (Project Milestone 3)

The XQuery engines built in the first part of the project miss the query optimization opportunities we know from the database literature. This may lead to unacceptable performance, as the following example shows.

Example 1 *Consider the following query.*

```
for $b in doc("input")/book,
    $a in doc("input")/entry,
    $tb in $b/title,
    $ta in $a/title
where $tb eq $ta
return
    <book-with-prices>
       { $tb,
         <price-review>{ $a/price }</price-review>,
         <price>{ $b/price }</price> }
    </book-with-prices>
```

Assume the query is evaluated on an input file with 10^5 *books and* 10^55 *reviews. Then in the brute force evaluation of the plan the where clause will be evaluated* 10^{10} *times, which is unacceptable from a performance point of view, since there is a much more ecient way to evaluate the query. In particular, one may:*

1. *collect a tuple set B, consisting of all tuples (b;tb) of books and their titles*
2. *collect a tuple set E, consisting of all tuples (a;ta) of entries and their titles*
3. *join the tuple sets B and E on the titles and derive a new tuple set R consisting of tuples (b; tb; a; ta)*
4. *produce a book with prices element for every tuple of R.*

The above plan can employ efficient join methods for Step 3. For example, one may index the tuples of B by title and then for each tuple of E find matching tuples of B using the index. Other efficient join methods can also be used (e.g., sort merge join or hash join). □

1 Introducing A Join Operator

Let us introduce a new XQuery operator, called join that

1. inputs two lists of tuples. Each tuple is of the form

```
<tuple>
  <a1 >v1 </a1 >
   ...
  <an >vn </an >
</tuple>
```

where the strings `a1, a2, ... an` are the attribute names. Each attribute value `v1, v2, ... vn` is a list of XML elements in the general case. The tuples of each list are homogeneous, in the sense that all tuples have the same attributes.

2. inputs two lists of attribute names, originating in the two tuple lists. We augment the XQuery syntax with a "list of constants" notation. In particular, we denote the list of attributes a1,...,an as [a1,...,an].

3. and outputs a list of tuples, where the output order is non-specified.

Using the join operator we can write more efficient versions of our queries, as the following example shows.

Example 2 *We rewrite the query of Example 1, using the join operator.*

```
for $tuple in join (for $b in doc("input")/book,
                        $tb in $b/title
                    return <tuple> <b> {$b} </b> <tb> {$tb} </tb> </tuple>,

                    for $a in doc("input")/entry,
                        $ta in $a/title
                    return <tuple> <a> {$a} </a> <ta> {$ta} </ta> </tuple>,

                    [tb], [ta] )
return
     <book-with-prices>
      { $tuple/tb/*,
        <price-review>{ $tuple/a/*/price }</price-review>,
        <price>{ $tuple/b/*/price }</price> }
     </book-with-prices>
```

The join operator above has four arguments. The first two arguments deliver the book tuples and entry tuples, while the third and fourth arguments specify that the join is on attributes tb and ta. □

2 Yet Another Subset of XQuery

Consider the following subset of XQuery, where there are no nested FLWR expressions and hence it is easier to introduce join operations. One may construct a corresponding syntax:

```
XQuery ::=  'for' V1 'in' Path1',' ..., Vn 'in' Pathn
            'where' Cond 'return' Return


Path   ::= ('doc('FileName')'|Var) Sep n1 Sep ... Sep nm
       |   ('doc('FileName')'|Var) Sep n1 Sep ... Sep 'text()'


Sep    ::= '/'
       |   '//'


Return ::= Var
       |   Return ',' Return
       |   '<'n'>' Return '</'n'>'
       |   Path
```

```
Cond   ::= (Var|Constant) 'eq' (Var|Constant)
       |   Cond 'and' Cond


Constant ::= StringLiteral
```

Queries of the above subset can be rewritten to make efficient use of the join operator, as the following example shows.

Example 3 *The following query returns triplets of books, where the first book has a first author named John, the second book has a common author with the first book and a common author with the third book. Notice that the query below may generate multiple triplets with the same books - even triplets where the same three books appear in the same order.*

```
for $b1 in doc("input")/book,
    $aj in $b1/author/first/text(),
    $a1 in $b1/author,
    $af1 in $a1/first,
    $al1 in $a1/last,

    $b2 in doc("input")/book,
    $a21 in $b2/author,
    $af21 in $a21/first,
    $al21 in $a21/last,
    $a22 in $b2/author,
    $af22 in $a22/first,
    $al22 in $a22/last,

    $b3 in doc("input")/book,
    $a3 in $b3/author,
    $af3 in $a3/first,
    $al3 in $a3/last

where $aj eq "John" and
      $af1 eq $af21 and $al1 eq $al21 and
      $af22 eq $af3 and $al22 eq $al3

return <triplet> {$b1, $b2, $b3} </triplet>
```

Notice how the rewriting uses two joins. Notice also that the joins are on pairs of attributes. The firrst join is on the pairs of attributes [af1, al1], respectively [af21, al21]. The second join is on the pairs of attributes [af22, al22] and [af3, al3].

```
for $tuple in join (
                    join (for $b1 in doc("input")/book,
                              $aj in $b1/author/first/text(),
                              $a1 in $b1/author,
                              $af1 in $a1/first,
                              $al1 in $a1/last,
                          where $aj eq "John"
                          return <tuple>
                                   <b1>{$b1}</b1>,
```

```
                                <aj>{$aj}</aj>,
                                <a1>{$a1}</a1>,
                                <af1>{$af1}</af1>
                                <al1>{$al1}</al1>
                              </tuple>,

                        for $b2 in doc("input")/book,
                            $a21 in $b2/author,
                            $af21 in $a21/first,
                            $al21 in $a21/last,
                            $a22 in $b2/author,
                            $af22 in $a22/first,
                            $al22 in $a22/last
                        return <tuple>
                                 <b2>{$b2}</b2>,
                                 <a21>{$a21}</a21>,
                                 <af21>{$af21}</af21>,
                                 <al21>{$al21}</al21>,
                                 <a22>{$a22}</a22>,
                                 <af22>{$af22}</af22>,
                                 <al22>{$al22}</al22>
                               </tuple>,
                        [af1, al1], [af21, al21]
                    ),


                    for $b3 in doc("input")/book,
                        $a3 in $b3/author,
                        $af3 in $a3/first,
                        $al3 in $a3/last,
                    return <tuple>
                              <b3>{$b3}</b3>,
                              <a3>{$a3}</a3>,
                              <af3>{$af3}</af3>,
                              <al3>{$al3}</al3>
                           </tuple>,


                    [af22, al22], [af3, al3]
                )
return <triplet> {$tuple/b1/*, $tuple/b2/*, $tuple/b3/*} </triplet>
```

□

This milestone's deliverables comprise

1. a query rewriter that takes as input a query of the above syntax, detects the joins, and outputs a query that makes the join explicit (like the queries in Examples 2 and the second query in Example 3.
2. an extension of your milestone 2 evaluation engine that supports input queries with the join keyword. The join should be implemented using a hash joind algorithm as discussed in class.