Bidirectional Analysis for XQuery Incremental Computation Program Automatic Generation

Xue-yun JIN
Beijing University of Technology, Beijing, China

Keywords: Incremental computation, Code automatic generation, Bidirectional analysis, XQuery.

Abstract. Incremental computation needs complex algorithm design and tough software maintenance. A bidirectional analysis technique is proposed to implement XQuery incremental computation program automatic generation based on the syntax of XQuery language. It can implement program transformation by bottom-up upward modification propagation analysis and top-down downward structure analysis, generate incremental computation program code automatically in light of the input data modification of the computation to obtain new result. The execution performance of the incremental computation program is better than that of recomputation.

Introduction

Incremental computation has been broadly applied in computer software field. The result of a computation may change when its input data is modified. Recomputation may result in unnecessary overload. Tiny changes in input data always have small influence on its result, so if incremental computation technique can get the influence of the change in input data on the original result, then a part of the original result can be updated to get the new result. On the other hand, XML data is semistructure data; it supports document order and axis operation, and XQuery is a functional, Turning-Complete programming language. Therefore, traditional query optimization technique on relational data can not be used in this senario. In order to solve this problem, a bidirectional Analysis technique for XQuery incremental computation program automatic generation is proposed in this paper. It performs program transformation for XQuery program, generates XQuery incremental program code automatically according to the change in the input data without index and schema information, and doesn’t need to store auxiliary data in the process of generation.

The formal description of incremental computation is: given computation \( f \), input data \( d \), the result of \( f(d) \) is \( r \). That is \( f(d)=r \). When there is a data modification \( \Delta d \) in \( d \), it causes a change \( \Delta r \) in \( r \) to make it becomes \( r' \). That is \( f(d\oplus\Delta d)=r' \) where \( a\oplus b \) means operating \( b \) on \( a \). If \( \Delta r \) can be obtained and know how to operate it on \( r \), then \( r' \) can be obtained by \( r\oplus\Delta r \). That is, \( r' = r\oplus\Delta r \).

In this paper, an incremental computation method \( \text{INC} \) is proposed to deal with \( f, d, \Delta d \) and \( r \) to obtain \( r' \). Here \( f \) is written in XQuery language, and \( d \) is XML data. That is, \( \text{INC}(f, d, \Delta d, r)=r' \). In order to achieve this aim, an incremental computation program automatic generation method \( \text{IA} \) is proposed based on bidirectional analysis for generating an incremental computation program \( p_{inc} \) to implement operating \( \Delta r \) on \( r \). That is, \( \text{IA}(f, d, \Delta d)=p_{inc} \) and \( p_{inc} (r) = r' \).

The contributions of this paper are: (1) XQuery incremental computation method. As a general-purpose method, it generates incremental computation program \( p_{inc} \) automatically; (2) Bidirectional analysis technique for incremental computation code generation. Bottom-up upward modification propagation analysis and top-down downward structure construction analysis are used to generate incremental computation program automatically for an XQuery computation with branching and user-defined function, without index, schema and auxiliary data.

Preliminaries

The core syntax of XQuery and XPath used in our work is shown in Figure 1.
Here $x$ is variable, $c$ is constant, $e$ is expression, $op$ is binary operator, $p$ is program. $e_{xpath}$ is an XPath expression which has several XPath test steps, $s$ is consist of an axis test $at$ (child test "/") or descendant test "/")", a name test $nt$ (XML element or attribute label) and an optional predication test $pt$. "*" is wild-card. $e_p$ can be null because it is an optional predication expression.

FXQL is a functional XML query language based on $\lambda$ calculus as intermediate language used to describe the query plan of XQuery computation. Every XQuery computation will be translated to an FXQL program at first. It provides enormous primitive functions to implement data process and program logic control. The core syntax of FXQL is shown in Figure 2.

```
\begin{align*}
\text{e} & := c \mid x \mid e_{xpath} \mid e_1 \; op \; e_2 \mid x(e_1, \ldots, e_9) \mid e_{nt} \; e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \\
\text{some} \mid \text{every} \mid x \in e_1 \text{ satisfies } e_2 \mid \text{let } x := e_1 \text{ [where } e_2 \text{ return } e_3 \text{] for } x \in e_1 \text{ [where } e_2 \text{ return } e_3 \text{]}
\end{align*}
```

Figure 1. XQuery/XPath Core Syntax.

```
P := \{ e \; \text{declare function } x(x_1, \ldots, x_n) \{ e \} \; p \mid e_{xpath} := e_{xpath} \; \varnothing \}
```

Figure 2. Core Syntax of FXQL.

Modification region $\text{reg}$ is a tuple $(i, k, r_{new})$ used to describe the influence on XQuery computation result caused by data modification where $i$ and $k$ are integers and $r_{new}$ is an XML node sequence. It means the influence range caused by data modification is $k$ items from the $i$th item, and the new result can be obtained by update these $k$ items by $r_{new}$. Modification region group $\text{regs}$ is a collection which consists of one or more modification region $\text{reg}$.

### Bidirectional Analysis Rule

Bidirectional analysis rule is the core of XQuery incremental computation automatic generation method based on bidirectional analysis. Except downward analysis rule and upward analysis rule, there is a start rule used as the first rule to invoke incremental algorithm.

$\Gamma$, $\Gamma_d$ and $\Gamma_u$ represent the context environment of start rule, downward analysis rule (called IncDown) and upward analysis rule (called IncUp). These environments store expression closure, function closure, modification region group and modification operation. The format of start rule is $\Gamma \leftarrow e:p'$, and the format of IncDown rule is $\Gamma_d \leftarrow \text{reg}$, they mean that under specific context $\Gamma$ or $\Gamma_d$, the transformation result of expression $e$ is incremental computation program $p$. The format of IncUp rule is $\Gamma_u \leftarrow e: \text{regs}'$, it means under specific context $\Gamma_u$, modification region group $\text{regs}$ for expression $e$ can be found. $d$ is input data, $\Delta d$ is input data modification. Some helper functions are used to describe these rules. These helper functions are shown in Table 1.

<table>
<thead>
<tr>
<th>Helper Function</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v = \text{eval}(e,f,d)$</td>
<td>Under $\Gamma$, evaluate $e$ on $d$, the result is $v$.</td>
</tr>
<tr>
<td>$v = \text{eval}(e,f,\Delta d)$</td>
<td>Under $\Gamma$, evaluate $e$ after data modification $\Delta d$ takes place on $d$, the result is $v$.</td>
</tr>
<tr>
<td>$p = \text{gen}(\text{regs})$</td>
<td>Generate Incremental Computation program $p$ according to $\text{regs}$.</td>
</tr>
<tr>
<td>$p = \text{p-create}(p)$</td>
<td>Package incremental computation $p$ to a complete XQuery program $p'$.</td>
</tr>
<tr>
<td>$\text{regs} = \text{IA-REGS}(e, \Delta d)$</td>
<td>Generate $\text{regs}$ according to $e$, $d$ and $\Delta d$. IA-REGS is modification region group capture algorithm.</td>
</tr>
<tr>
<td>$\text{reg} = \text{c-reg}(i, k, r_{new})$</td>
<td>Generate new $\text{reg}$ according to $i$, $k$ and $r_{new}$.</td>
</tr>
<tr>
<td>$\text{reg} = \text{u-reg}(i, k, r_{new})$</td>
<td>Update an original $\text{reg}$ to a new one $\text{reg}'$ according to the value of $i$, $k$, and $r_{new}$.</td>
</tr>
<tr>
<td>$n = s$</td>
<td>Count number of items in a sequence $s$.</td>
</tr>
</tbody>
</table>

When generating incremental update code, the first called rule is start rule $\text{START}$ (shown in Fig.7). It copies $\Gamma$ to $\Gamma_d$, and calls IncDown rule to deal with FXQL expression $e$ to obtain incremental computation program $p$. Then $p$ is packed according to XQuery syntax, to get a program $p'$ which can incrementally maintain the result $r$ of an expression $e$.

The aim of IncDown is to find incremental computation program structure according to the
constructing procedure of the computation value top-downly, and generate incremental computation program according to the \( \text{regs} \) obtained by upward analysis rule \( \text{IncUp} \). Part of \( \text{IncDown} \) rules are shown in Figure 3.

\[
\begin{align*}
\Gamma_e = \Gamma & \quad \Gamma \vdash e : p \quad p' = \text{p-create}(p) \\
\Gamma \vdash e : p' & \quad (\text{START}) \\
(\Gamma, x \rightarrow e_i) \& e_i' : p & \quad (\text{D-CONST}) \\
(\Gamma, x \rightarrow (x_1, \ldots, x_n)) \& e_i : p & \quad (\text{D-ViABLE}) \\
(\Gamma, x \rightarrow e_i) \& e_i' & \quad (\text{D-ViABLE}) \\
\text{eval}(e_i, \Gamma_e, d) & = \text{n-eval}(e_i, \Gamma_e, d) \\
\text{n-eval}(e_i, \Gamma_e, d) & = \text{true} \quad p = \text{gen}((0, 0, \text{n-eval}(e_i, \Gamma_e, d))) \\
\text{eval}(e_i, \Gamma_e, d) & = \text{false} \\
\text{IncDown} & = \text{n-eval}(e_i, \Gamma_e, d) \\
\text{eval}(e_i, \Gamma_e, d) & = \text{true} \quad p = \text{gen}((0, 0, \text{n-eval}(e_i, \Gamma_e, d))) \\
\text{n-eval}(e_i, \Gamma_e, d) & = \text{false} \\
\text{eval}(e_i, \Gamma_e, d) & = \text{false} \\
\text{eval}(e_i, \Gamma_e, d) & = \text{false} \\
\text{eval}(e_i, \Gamma_e, d) & = \text{false} \\
\text{eval}(e_i, \Gamma_e, d) & = \text{false}
\end{align*}
\]

Figure 3. Start Rule and IncDown Rules.

Here \( \text{D-CONST}, \text{D-ViABLE}, \text{D-ViABLE}, \text{D-FUNDEF} \) and \( \text{D-FUNCALL} \) deal with const expression, variable expression, local variable expression, function definition expression, function call expression respectively. For conditional expression \( e ::= \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \), the value of it depends on the value of \( e_1 \). If the data modification doesn’t affect the value of \( e_1 \), just maintain the original result. Otherwise, \( e_2 \) or \( e_3 \) should be computed. Therefore, there are 4 rules \( \text{D-If1}, \text{D-If2}, \text{D-If3} \) and \( \text{D-If4} \). Rule \( \text{D-Un} \) is responsible for the primitives that we can not deal with.

IncUp rule generate and maintain modification region group according to \( d, \Delta d \) and \( e \). Parts of IncUp are shown in Figure 4.

\[
\begin{align*}
\text{null} & \quad \Gamma_e = \Gamma_x(x) \quad e_i \text{ is spath}_{e_i} \quad \text{regs} = \text{IA-XP}(e_i \& d, d) \\
\Gamma_x(x) \rightarrow \text{regs} & \quad (\text{U-ViABLE}) \\
\Gamma_x(x) \rightarrow e_i & \quad \text{regs} \quad (\text{U-ViABLE}) \\
\Gamma_x(x) \rightarrow e_i & \quad \text{regs} \\
\text{eval}(e_i, \Gamma_e, d) & = \text{n-eval}(e_i, \Gamma_e, d) \\
\text{n-eval}(e_i, \Gamma_e, d) & = \text{true} \quad \text{regs} \\
\text{n-eval}(e_i, \Gamma_e, d) & = \text{false} \\
\text{eval}(e_i, \Gamma_e, d) & = \text{false} \\
\text{eval}(e_i, \Gamma_e, d) & = \text{false} \\
\text{eval}(e_i, \Gamma_e, d) & = \text{false}
\end{align*}
\]

Figure 4. IncUp Rules.

Here, \( \text{U-CONST}, \text{U-ViABLE1}, \text{U-ViABLE}, \text{U-FUNDEF} \) and \( \text{U-FUNDEF} \) are used to deal with const expression, variable expression, local variable definition expression, function definition expression, function call expression respectively. For conditional expression \( e ::= \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \), there are still 4 rules \( \text{U-If1}, \text{U-If2}, \text{U-If3} \) and \( \text{U-If4} \). Rule \( \text{U-Un} \) is responsible for the primitives that we can not deal with. It just calls \( \text{n-eval} \) to obtain a modification region to replace the original result.

**Experiments**

The hardware of the experiment is: the CPU is Intel(R) T6670 @2.20GHz, RAM is 3.00GB, Windwos7 professional. It uses 13 use cases which overlaps all kinds of XQuery syntax. \textit{bib.xml} provided by W3C is used as XML data source. 5 data sources of size 10M, 20M, 40M, 100M and 400M have information of 32768, 66536, 131072, 327680, and 131070 books respectively.

Generation time \( T_c \), incremental maintenance time \( T_i \) and recomputation time \( T_r \) are recorded. \( T_c \) denotes the time period to generate an incremental computation program. \( T_i \) denotes the time period of executing the incremental computation program. \( T_r \) denotes recomputation time period.

The result of experiment is: (1) According to Figure 5(a), the bigger the XML data size, the
bigger the \( T_c \); (2) According to Figure 5(b), when the size of XML data increases, the distance between \( T_i \), \( T_c \) and \( T_r \) increase too. Therefore, this method has advantage when the size of data source is big. (3) Define speed-up ratio \( s_r = T_i / T_r \). In Figure 5(c), \( s_r \) is between 2 to 5.

![Figure 5. Experiment Result.](image)

Relative Works

Nowadays, incremental computation is acknowledged as a technique which can improve performance in various fields [1]. XML data is semi-structure data, and the standard language of XML data query XQuery is functional language which has notable difference from SQL language. Therefore, the relative technique in relational database field can not be used on XML data query optimization, and new technique and method should be studied to implement incremental computation. Relative works [2,3] all study the incremental maintenance of XML materialized view. Paper [2] use XAM (XML Access Modules) to represent XML storage, index or materialized view, use \textit{select} to implement join operation on Cartesian product in tree pattern algebra. Because there are lots of items need to be maintained, some pruning operation should be done according to the data modification, and incremental maintenance is implemented by computing updated parts of materialized view according to data modification and the tree pattern algebra. Paper [3] uses Tree algebra of Galax which is established on tuple. It translates XQuery program into Tree algebra of Galax in order to avoid the complexity of XQuery syntax, and only deal with the computation with simple operator with combined semantics. It provides update propagation rule according to every algebra operation. In query phase, it stores auxiliary data for every operator. In update propagation phase, all algebra operators in Tree algebra are processed recursively, and stores auxiliary data to promise the execution of the propagation rule. Paper [4] tries to develop an incremental maintenance method for XQuery materialized view, but not with this bidirectional method. Paper[5] uses the theory of incremental computation and proposes a dynamic scheduling for online integration plans, which employs data increment monitoring system which is able to dynamically change the data integration plans whenever it is necessary.

Conclusion

We develop a kind of XQuery incremental computation program automatic generation technique. It employs incremental computation code generation technique based on bidirectional analysis and XPath modification region analysis method to do program transform for an XQuery computation,
and generate incremental computation program code automatically for this computation. It can obtain new result according to the original result of the computation before data modification. The executing performance of the incremental computation code is better than that of recomputation.

**References**


