University of California San Diego

CSE 232B Project - Final Report

J. Sidrach

March 17th, 2017

1 Overview

The objective of this project is to design and implement an efficient *XQuery* evaluator. The specific semantics are a simplification (not a strict subset) of *XQuery 1.0*[1]. The grammar's specification is formally defined in `docs/references/Simplified XQuery Semantics.pdf`. The most notable differences are the lack of types and the removal of most of the grammar rules, while still allowing the queries to be expressive enough for the purposes of this project.

More detailed information about the structure of the project's source code, along with instructions on how to build and run it, can be found in the `README.md` file under the project's root directory. Additionally, the `javadoc` documentation is available under `docs/javadoc/`.

2 XQuery Evaluator

The first phase of the *XQuery Evaluator* is to parse an input query and build its Abstract Syntax Tree (AST). This task is relatively easy with the help of the *ANTLR4*[2] library. *ANTLR*, given the definition of the grammar, auto-generates the necessary code to achieve this task in just a couple of function calls, while also providing base classes for the traversal of the AST.

The next phase consists in the actual evaluation of the query. To do so, the AST is traversed using the *ANTLR* visitor pattern (as opposed to listener). The visitor pattern allows more control on this traversal, or even avoid, under certain circumstances, to visit some sub-trees altogether. For instance, whenever there are two conditions joined with the binary `or` operator, if the first one evaluates to true there is no need to traverse/evaluate the second one.

The most relevant class for the evaluation of the query is `XQueryVisitor`. The evaluation of every rule depends on the current value of two of its properties: a list of nodes and a map from variable names to the list of nodes each one represents. Some type of rules, like *XPath* filters or *XQuery* conditions, are only allowed to read these two variables but not to modify them. To deal with this restriction, the methods that visit these rules store first the current value of both properties. Then, once the sub-tree has been visited, their values are restored. A similar approach is used to deal with the scope of variables. All rules that introduce new variables first store a copy of the current value of the map. Once the new variables should no longer be in scope, the stored copy of the map is restored.

[1] XQuery 1.0: An XML Query Language (https://www.w3.org/TR/xquery)

[2] ANTLR: ANother Tool for Language Recognition (http://www.antlr.org)

The project also contains unit and integration tests for every rule in the grammar. Each test case consists of a given *XQuery* and its expected *XML* result, and it verifies that the evaluation of the input *XQuery* results in exactly the same *XML* as the provided one. In total, there are more than 120 test cases, and all of them are run as an intermediate step in the build process.

3 XQuery Optimizer

The *XQuery Optimizer* rewrites "`for [...] where [...] return [...]`" expressions into equivalent ones using the join operator (added for this purpose), when some restrictions are satisfied. The description of this process is documented in `docs/references/Join Optimization.pdf`. The *XQuery* grammar has been extended, introducing the join operator. The implementation of the join is based on the *Hash Join* algorithm. For every children of the left and right sub-queries with the same hash, the result of the join includes a new node containing both. This hash (a string) only depends on the attributes specified in the join operator.

The query optimization is done in three stages, all based on an *ANTLR* visitor, `XQuerySerializer`. This visitor takes a query and returns an equivalent query (`String -> String`), allowing the composition of the three stages (visitors) easily. The optimizer uses visitors, instead of string manipulations, to be able to access to the AST and its information in the rewrite process.

- The first stage, implemented in `XQueryVarsRenamer`, simply renames all variables so their names are unique (there won't be name collisions with the current restrictions, but this simplifies more complex optimizations).

- The second stage first verifies that the given query can be rewritten - it checks that it conforms to the sub-grammar for which the optimization is defined. If this is the case, it rewrites the query using the join operator, and it pushes back the selections of the form of `$var = "Constant"`. The rewrite process also works in cases where more than one join operator is needed, nesting them.

- The third stage, implemented in `XQueryFormatter`, formats/indents the query so that it is easier to read and debug.

The test suite for the query optimizer verify that the evaluation of the output of each stage yields the same result as the evaluation of the original query. Additionally, the tests also check that the stages are idempotent (going through a stage once should result in the same output as going through it twice).

All in all, the applied optimizations are somewhat limited to really specific cases. However, the developed solution is easy to extend to support more general/complex optimizations, following the defined structure of stages.