Chapter 13
Normalization and Translation of XQuery

Norman May
SAP Research, CEC Karlsruhe Vincenz-Prießnitz-Str. 1 Germany

Guido Moerkotte
Database Research Group University of Mannheim, Germany

ABSTRACT

Early approaches to XQuery processing proposed proprietary techniques to optimize and evaluate XQuery statements. In this chapter, the authors argue for an algebraic optimization and evaluation technique for XQuery as it allows us to benefit from experience gained with relational databases. An algebraic XQuery processing method requires a translation into an algebra representation. While many publications already exist on algebraic optimizations and evaluation techniques for XQuery, an assessment of translation techniques is required. Consequently, they give a comprehensive survey for translating XQuery into various query representations. The authors relate these approaches to the way normalization and translation is implemented in Natix and discuss these two steps in detail. In their experience, their translation method is a good basis for further optimizations and query evaluation.

INTRODUCTION

As XQuery is used in an increasing number of applications, the execution time of these queries becomes more important for the acceptance of this query language. Especially for queries where potentially large amounts of XML are processed, strategies to reduce the query processing time need to be applied. The first XQuery processors often implemented a number of heuristics for this purpose. As XQuery becomes more popular, specific storage and index structures as well as specialized execution strategies were implemented.

Recently algebraic optimization techniques, as they are standard in relational databases, were used to build XQuery processors where optimizations are stated as algebraic equivalences, e.g. (Naughton et al., 2001; Jagadish et al., 2002; Fiebig et al., 2002; May et al., 2006; Ozcan et al., 2005; Nicola & van der Linden, 2005; Florescu et al., 2004; Boncz et al., 2006; Liu et al., 2005; Pal et al., 2005). This development is motivated by the ability (1) to apply

DOI: 10.4018/978-1-61520-727-5.ch013

optimizations known from relational databases and adopt them for XML processing, (2) to prove the correctness of query optimizations based on a formally defined query representation. Relational query optimizers today approach query optimization in a sequence of six steps:

1. **Scan and Parse the query statement** to analyze the lexical structure of the query.
2. **Normalization of the query, translation into an internal representation, type checking, and semantic analysis**: This phase checks the semantic correctness of the query. At the same time, additional information is attached to the parse tree, e.g. type information, references to schema information, or references to available statistics. As the parse tree is not the most convenient representation of the query to apply optimizations, it is translated into an internal query representation – usually an algebraic or calculus representation. The translation step may require some normalization to be applied before. The XQuery specification gives some rules for typing XQuery expressions (Draper et al., 2007), but more precise results can sometimes be obtained. In this chapter, we will focus on this optimization phase, in particular the normalization and translation step.
3. **First heuristic optimization phase**: In this phase the query optimizer applies heuristic optimizations. Some of these optimizations are hard to implement in a cost-based optimizer, e.g. predicate-move-around. Other optimizations applied in this phase prepare the query so that the search space of the cost-based optimizer is increased, e.g. query unnesting (May et al., 2004), (May et al., 2006) or view merging, and thus, often drastically improve the overall quality of the resulting query execution plan.
4. **Cost-based optimization phase**: Based on cardinality estimates and a cost-model for possible implementations of a query, the cost-based optimizer generates equivalent plan alternatives, called query execution plan (QEP). These alternatives differ in the order of the involved operators or their implementations. Among all alternatives examined in this phase, the most efficient one is chosen. Several metrics are used as efficiency criteria, e.g. resource consumption or expected query execution time.
5. **Second heuristic optimization phase**: This phase applies heuristic optimizations that are not considered in the previous phase, e.g. merging adjacent operators.
6. **Code generation**: This phase transforms the QEP into an executable form.

In this chapter, we first discuss how an XQuery query can be translated into an internal representation that is close the well-known QGM-model used in IBM Starburst/DB2 (Haas et al., 1989). As several other query optimizers use a similar query representation, it is desirable to reuse their infrastructure to implement a query optimizer for XQuery. As (Grust et al., 2004) have pointed out, some care is required when a relational database is used to evaluate XQuery. Others discussed algebraic optimizations for XQuery based on native XML database management systems, e.g. (May et al., 2006; May, 2007; Re et al., 2006).

After surveying existing approaches to algebraic XQuery optimization and translation approaches, we introduce an algebra to represent XQuery statements that serve as input to algebraic optimizations. This algebra is defined over sequences of tuples as it is required to preserve the semantics of XQuery. Hence, it is not possible to directly apply algebraic optimizations known for SQL and relational databases to XQuery.

Then, we define the fragment of XQuery that is supported by our translation approach. This fragment covers a large fraction of the XQuery 1.0 language. After that, we introduce a number of normalization rules that prepare the XQuery