Datenbank Spektrum (2014) 14:135–150
DOI 10.1007/s13222-014-0160-3

FACHBEITRAG

Unleashing XQuery for Data-Independent Programming

Sebastian Bächle · Caetano Sauer

Received: 7 April 2014 / Accepted: 13 May 2014 / Published online: 17 June 2014

Abstract The XQuery language was initially developed as an SQL equivalent for XML data, but its roots in functional programming make it also a perfect choice for processing almost any kind of structured and semi-structured data. Apart from standard XML processing, however, advanced language features make it hard to efficiently implement the complete language for large data volumes. This work proposes a novel compilation strategy that provides both flexibility and efficiency to unleash XQuery's potential as data programming language. It combines the simplicity and versatility of a storage-independent data abstraction with the scalability advantages of set-oriented processing. Expensive iterative sections in a query are unrolled to a pipeline of relational-style operators, which is open for optimized join processing, index use, and parallelization. The remaining aspects of the language are processed in a standard fashion, yet can be compiled anytime to more efficient native operations of the actual runtime environment. This hybrid compilation mechanism yields an efficient and highly flexible query engine that is able to drive any computation from simple XML transformation to complex data analysis, even on non-XML data. Experiments with our prototype and state-of-the-art competitors in classic XML query processing and business analytics over relational data attest the generality and efficiency of the design.

Keywords XQuery · Query optimization · Data abstraction

S. Bächle (✉) · C. Sauer
TU Kaiserslautern, P. O. Box 3049, 67653 Kaiserslautern, Germany
e-mail: baechle@cs.uni-kl.de

C. Sauer
e-mail: csauer@cs.uni-kl.de

1 Introduction

The initiators of XQuery equipped the "query language" with many ingredients of a general-purpose functional programming language: side-effect-free, composable expressions, variable bindings, recursion, and higher-order functions.[1] For data processing, however, the by far most interesting part is the widely-used for loop construct. It iterates over a sequence of data items, which itself is a basic concept of the underlying data model. Such iterative processing is a core task in data-intensive applications to filter, transform, aggregate, and sort the input data. Accordingly, XQuery can be considered as an XML-enabled data programmming language rather than a declarative query language for XML.

In practice, XQuery engines are typically designed and optimized for specific usage scenarios. Most of them focus on fast XML transformation and data extraction in main memory, while others aim for large data volumes and XML index support in database systems, or offer special features for full-text search, streaming applications, etc. As usual, such specialization comes along with limitations for more general use cases. Quick, feature-complete main-memory engines usually cannot handle large amounts of data, whereas powerful XML-enabled database backends need to convert all data into their specialized internal layout and often also lack support for certain language features.

This paper presents a novel compiler that bridges the gap between these extremes and allows to use XQuery as general-purpose language for data processing. We aim for an extensible infrastructure that offers full language support, high performance, and scalability for a great variety of systems, use cases, and data formats.

[1] New in XQuery 3.0: http://www.w3.org/TR/xquery-30/

The basis is a separation of language concepts like variable bindings and explicit looping from the concerns of data representation and data access. We extract expressions with looping semantics – the scalability bottleneck in most processors – from a query, convert them to a relational-style operator pipeline and apply set-oriented algorithms and optimizations to scale with large data volumes. By operating on an abstraction of item sequences, the system is capable to map queries seamlessly to various XML storages, but also to other data formats (e.g., relational tables, flat files, or JSON) which can be interpreted as tree structures.

Relational standard optimizations like join rewriting are already built-in, but the concept is explicitly designed to be easily extended for parallelism and platform-specific optimizations, e.g., XML path matching, indexes and native bulk data access. As a result, we get a powerful tool to carry out XML transformation, data-base-backed query evaluation, and complex data processing tasks within a single language.

Next, Sect. 2, introduces our processing model and Sect. 3, presents the basics of unrolled FLWOR expressions in a set-oriented operator pipeline. The compilation process is described in detail in Sect. 4, and Sects. 5 and 6 present techniques for join optimization and for XML and general data processing. Section 7 testifies competitiveness of our prototype called *Brackit* in several experiments. Finally, we discuss related approaches in Sect. 8, and Sect. 9 concludes the paper.

2 Processing Models

Essentially, a query can be represented as a tree of expressions that is evaluated to a sequence of items, i.e., XML nodes, atomic values, and function items. For illustration, consider the sample query and the corresponding expression tree shown in Fig. 1.

The query consists of two nested FLWOR expressions with for loop clauses. The outer loop binds the values 1, 2, and 3 successively to a variable $a and evaluates for each binding the nested FLWOR expression, which itself loops with a run variable $b over the sequence (2,3,4) to compute {$a,$b} pairs fulfilling the predicate of the where clause. Finally, the return clause produces the output $a+$b for each pair. Accordingly, we obtain the result (4,6).

2.1 Dynamic Context

Within a loop, individual subexpressions like the comparison and the summation are evaluated multiple times, but each time within a different *dynamic context*, i.e., iteration. The context consists of explicitly-bound variables and an implicitly-bound context item, its position in a context sequence, and the size of the latter. As we will show, modelling of the dynamic context is key to scalable query processing.

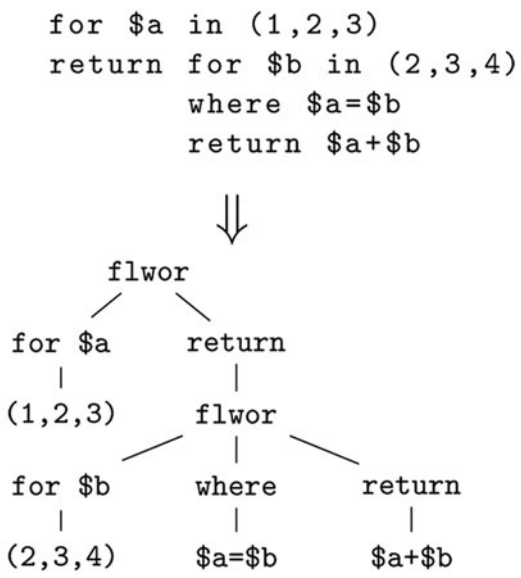


Fig. 1 Nested for loops

In general, one distinguishes between simple expressions like literals, arithmetics or comparisons, and expressions that modify the dynamic context visible to their subexpressions. The for clauses and let clauses of FLWOR expressions are the most prominent ones. They explicitly bind values to variables in the dynamic context that can be referenced in the expressions of subsequent clauses. Similarly, each step in a path expression, which is used to navigate XML trees, provides implicit context nodes to the next step.

For the evaluation, there are now two options. One option is to treat the dynamic context like a set of variables, which is accessible to each expression and updated whenever necessary, e.g., when a new value is bound to a variable. The second option is to model each specific state during processing as a separate and immutable dynamic context. Accordingly, each binding of a value to a variable and also each path step creates a new dynamic context.

In the following, we will primarily focus on how to organize the dynamic context and how to evaluate context-dependent expressions. We will not delve into XML path processing or any other XML-related aspect of the language, because this complicates the discussion, but does not influence the general design. Later in Sect. 6, we will show how to map a query efficiently to the actual data format (e.g., XML or relational tables) and properties of the data store. At the moment, however, it is adequate to treat path expressions

just like any other expression which evaluate to a sequence of items.

2.2 Iterative Processing

Most XQuery processors evaluate queries in a sequential fashion according to the XQuery Core semantics [1]. The expression tree is evaluated recursively, i.e., subexpressions are evaluated before the actual result sequence is built. Intermediate results are usually not materialized to save main memory. Instead, expressions evaluate to a *lazy sequence*, which employs a pull-based iterator concept to compute individual result items on demand [2]. In general, this makes iterative processing very space efficient.

With such lazy evaluation, the result of the expression in Fig. 1 is an iterator over the outer for loop, which flattens the results of the lazy result iterators of the three nested evaluations of the inner loop.

The dynamic context is usually represented as a mutable set of variables, because it is simpler and naturally fits the processing model. Some implementations use a global context object, whereas others split the context and store parts of it locally in the scope of the corresponding iterator. An iterator over a FLWOR sequence, for example, may hold the values of variables, which are bound by its local for and let clauses. In this case, the local scopes must be recursively passed to subexpressions and the entire dynamic context turns into a hierarchical leaf-to-root composite.

By representing data closely to the XQuery standard as sequences of items and modeling XML fragments as trees of node items, suitable abstraction layers allow iterative processors to run on top of various data stores. The straightforward design makes it also simple to integrate the new programming features like function items and partial function application.

The disadvantage of the mutable dynamic context is an implied sequential evaluation order of context-dependent expressions. It requires special treatment in individual situations to perform common optimizations like parallel computation of (partial) results or application of an efficient join algorithm instead of computing and filtering the Cartesian product. As a result, processors often suffer from nested-loops semantics and poor scalability. Because of its simplicity, it is, nevertheless, the most common design used in XQuery processors for main memory [2–5].

2.3 Set-Oriented Processing

Set-oriented processors compute independent subexpressions and multiple iterations with more efficient operations, in a different order, or even in parallel. Therefore, the expression tree must be translated into a more general plan representation or algebra [6–8]. Then, rewrite operations break up all nestings and loop constructs, which allows to leverage the performance of well-known set-oriented algorithms. Various kinds of heuristics-based and cost-based optimization provide additional performance gains.

Set-orientation requires the explicit representation of each individual state of the dynamic context to compute the result of an expression for all iterations at once. This implies some overhead to materialize the dynamic contexts, but usually the performance and scalability advantages outweigh.

Compilers for relational target platforms use here elaborate concepts to map queries and data to relational algorithms and data layouts. Strict relational processing, however, requires to radically rewrite the entire query, which severely complicates the realization of language concepts beyond plain "Select-Project-Join" and requires an advanced optimizer to compile efficient plans. Even many compilers for native XML storages face similar problems. To exploit a particular data representation, e.g., node labeling or other properties of proprietary data stores, the optimizer has to cope with many special operators, ill-shaped query plans or sophisticated dependencies between operators.

In both types of systems, the programming language aspects of XQuery like user-defined functions, recursion, and, recently, also higher-order functions usually fall behind. They just do not fit into the picture of database-style processing. This disqualifies such systems as runtime environment for general data programming tasks.

Nevertheless, users often accept partial language conformance as long as a system meets their compatibility and performance needs. On the flip side, they cannot take advantage of one of XQuery's biggest strengths–the ability to represent, interpret, and process different kinds of information from diverse sources [9].

2.4 Hybrid Processing

We chose a third way for our compiler. We restructure only those parts in a query, where we can profit from set-oriented processing: FLWORs and nestings of them appearing in the binding expressions of for clauses and let clauses or in return clauses. We extract these parts from the expression tree to form a pipeline of relational-style operators. The rest is left intact. At the top of a pipeline, a special pipe expression mediates between the iterative and the set-oriented processing model. Consequently, a query is still an expression tree that is evaluated recursively, but now it may also contain one or more operator pipelines, which are evaluated in a set-oriented fashion. Figure 2 shows the rewritten expression tree of Fig. 1. The nested FLWOR expressions have been replaced by a pipe expression with a right-deep top-down pipeline of operators (starting in uppercase letters).

The dynamic context is stored in *context tuples* which are passed between expressions and operators. A tuple [1,2],

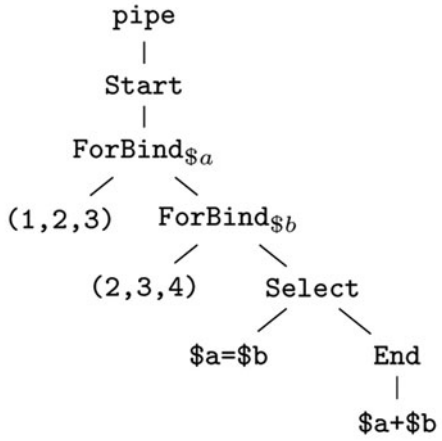


Fig. 2 Expression tree with operator pipeline

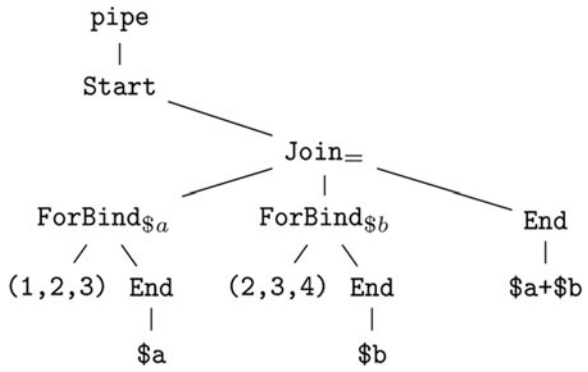


Fig. 3 Expression tree with join operator

for example, represents a dynamic context with the variable bindings $a=1 and $b=2. The fundamental difference between expressions and operators is that the former are evaluated only for a single context tuple at a time and produce an item sequence as result, whereas the latter consume and produce streams of tuples.

With this hybrid design, we are now able to extract expensive iterative sections from queries and process them in a set-oriented fashion. At the same time, we can independently optimize and compile the remaining expression parts like arithmetics, path expressions, and (recursive) function calls. The concept presented is fully composable, i.e., nestings may be arbitrarily deep and rewriting rules are independent of the surrounding expression. Furthermore, the approach is not limited to certain query constructs. It supports extensions like the XQuery Update Facility and all aspects of XQuery 3.0 like group-by clauses and window clauses.

3 FLWOR Pipelines

A FLWOR pipeline produces a stream of context tuples for the final `return` clause of a FLWOR expression. Essentially, the operators produce, filter, reorder, and aggregate the tuples as specified by the FLWOR clauses. For each clause type, there exists one corresponding operator type. Accordingly, it can be considered as a straightforward realization of the tuple stream semantics as defined in the language [9].

A pipeline begins with an artificial Start operator, which serves as input for a ForBind or LetBind operator that is introduced for the initial for clause or let clause, respectively. The process is the same for all intermediate clauses. Each operator consumes the input from the previously created operator and serves as input for the following. The End terminator at the bottom evaluates the return clause expression.

3.1 Data Flow

The data flow in a pipeline is easily explained with the help of the rewritten sample query of Fig. 2. First, the current context, in our case, the empty tuple [], is passed to the pipe expression, which initializes and returns a lazy result sequence for this context. When this sequence is iterated, the result items will be computed by evaluating the expression $a+$b for each [$a,$b] tuple returned from operator pipeline. The pipeline itself is processed with cursors[2] following the open-next-close principle [10]. The Start cursor feeds the single context tuple [] to the pipeline. Then, for each input tuple, the ForBind cursors evaluate and bind all items of their binding sequences, which may be arbitrary XQuery expressions. Binding means that the input tuples are extended by the value to be bound. The first ForBind produces the tuples [1], [2], [3], which are consumed by the second ForBind to produce the tuples [1,2],[1,3], . . . , [3,4]. Finally, Select evaluates the comparison expression to filter the tuple stream.

Obviously, the rewriting does not improve a query at all. However, it opens the door to apply various well-known optimizations from the relational world to the operator tree. As long as the final stream of output tuples is not affected, operators can be reordered, merged, or replaced to obtain a better performing pipeline.

We can easily determine whether a Select can be pulled up to reduce the number of intermediate tuples or not. Sorts, which are introduced for order by clauses, might be merged to a single one or determined to be superfluous. We might also choose to process multiple stages in parallel. In fact, various kinds of parallelism are possible because the context tuples decouple the intermediate state from the actual computation.

The most effective optimization for the sample query is shown in Fig. 3. The input pipeline is split into two indepen-

[2] Note, we explicitly distinguish between *iterators* to iterate over the items of an XQuery sequence, and *cursors* to iterate over streams of context tuples.

dent sections to form a join. The first two child branches of the join node receive both the input stream and produce the join input relations with the respective join keys. The join result is passed on to the output pipeline at the right-most node. The equi join could be computed with a hash join such that the initially nested binding sequence (2,3,4) has to be processed only once to build the hash table.

3.2 Pipe Lifting

So far, we only considered for-bound variables, which hold the single items of a binding sequence. However, XQuery also allows to bind whole item sequences to variables, e.g., with let bindings as exemplified by the query in Fig. 4. After rewriting the corresponding expression tree, we obtain the tree depicted at the bottom. The LetBind operator evaluates for each incoming context tuple a pipe expression for the rewritten FLWOR of the initial binding sequence and binds the entire result to variable $c. Accordingly, it produces the tuples [1,(2,3,4)], [2,(3,4)], and [3,4].

Technically, tuples containing sequences instead of single items are rarely a problem. Remember, it is always possible to evaluate intermediate sequences eagerly or to bind just a lazy sequence that will compute its items on demand. In many cases, it is reasonable to compute let-bound sequences directly within the pipeline, e.g., because it offers better options for optimizations such as joins. As another example, cheap lazy sequences that flow through the pipeline can cause load skew in parallel processing, because the consuming process at the end does the actual work. Therefore, we perform *pipe lifting*: We "lift" pipelines nested in LetBind operators and integrate them as artificial left join into the higher-level pipeline as shown in Fig. 5.

The first child branch, i.e., the outer relation of the left join is empty and just passes through the input stream from above. The second child branch (inner relation) consists of the lifted pipeline and a let-bound help variable $ok. Both branches are terminated by End operators which contribute both the same constant join key. Effectively, this turns the left join into a loop over the nested pipeline.

Because lifting breaks the visibility scope of lifted variable bindings, a Count operator is introduced after the outer ForBind of variable $a to keep track of the iteration a specific context tuple belongs to. It produces the [$a,$grp] tuples [1,1], [2,2], and [3,3] as input for the lifted pipeline. Note, this is equivalent to a standard count clause in XQuery 3.0. The output of the left join is fed to a Concat operator, which resembles a guarded evaluation of the lifted return expression. The subsequent GroupBy uses the count variable to restore the semantics of the let binding.

Within lifted parts, it may happen that we do not produce a result for a specific context, e.g., when the binding sequence of a ForBind is empty or when a Select filters all tuples.

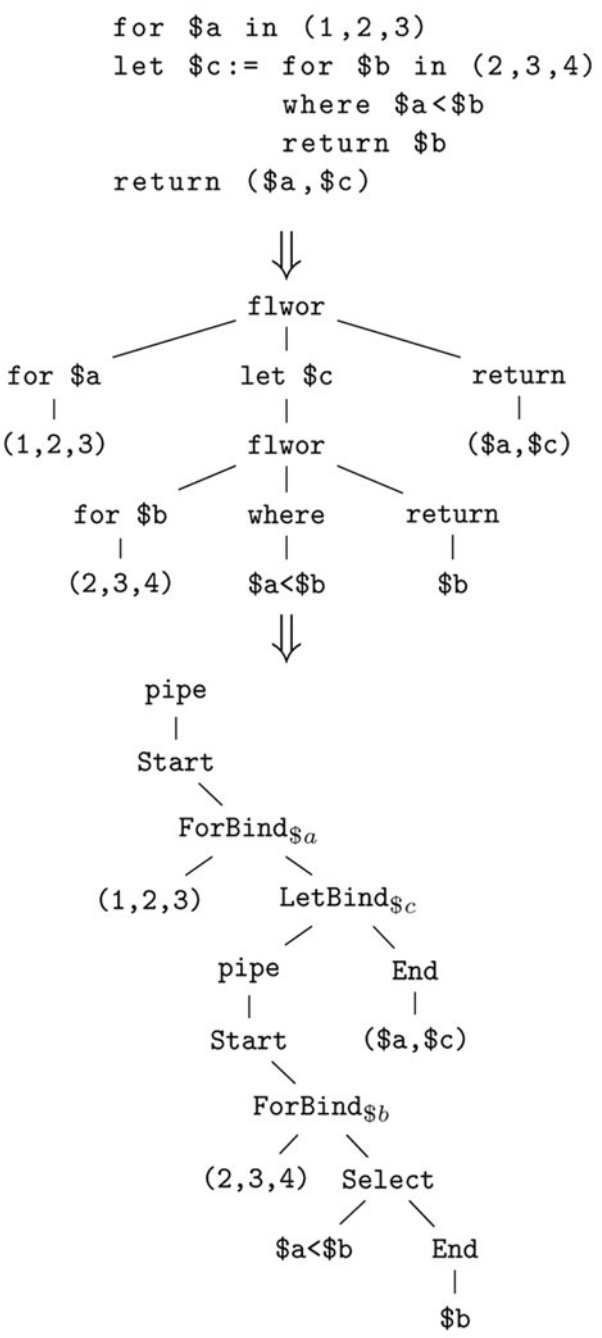


Fig. 4 Nested FLWOR in let binding

In this case, the left join semantics keeps the outer iteration alive to preserve the nesting semantics. The Concat operator expresses the conditional evaluation of a lifted pipeline section by referring to the help variable $ok. Optionally, the explicit Concat operator can be compiled to cheap condition checks within the affected operators. They must only ensure to emit at least one tuple for each distinct $grp, i.e., each outer iteration. Whenever an operator, e.g., a Select, is about

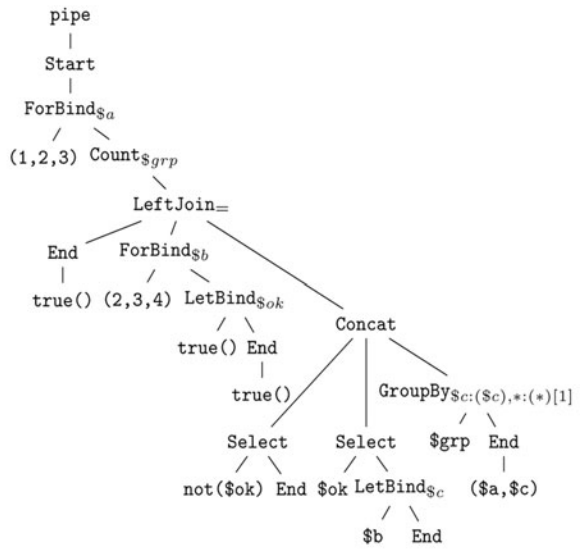


Fig. 5 Lifted operator pipeline

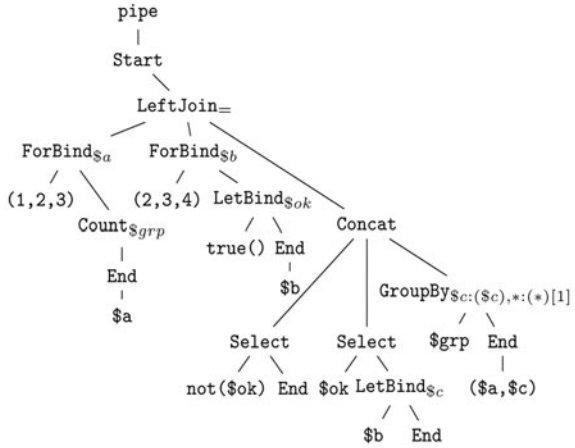


Fig. 6 Join in lifted pipeline

to discard the last tuple of the current iteration, it unbinds $grp and emits it as "dead" tuple instead. This signals subsequent operators to simply pass-through the tuple as a dead iteration.

The key $grp also identifies all [$a,$grp,$b,$c] tuples that belong to the same iteration of the outer loop. For each group, the GroupBy operator[3] outputs a single tuple, where $c is re-bound to the ordered sequence of all incoming $c's. The variable bindings from outside of the lifted part are reduced to a single representative by simply copying them from the first group tuple to the output tuple. Note, it actually does not matter from which input tuple they are copied, because the outer variables are necessarily bound to the same values in each tuple of the group. Variable bindings from within the lifted part like $b are semantically out of scope and can be discarded. As a result, we obtain the same output tuples as in the nested version.

As the major difference to the unlifted version, we compute the single items of let-bound sequences now directly within the operator tree and not isolated in a lazy sequence. Obviously, this is an advantage for parallel processing and improves chances for other optimizations like join rewriting. In Fig. 5, e.g., it is much easier to identify and exploit formerly hidden join semantics which can be leveraged as shown in Fig. 6.

[3] The aggregation specification for non-grouping variables of a GroupBy operator is represented as comma-separated list of XQuery-like expressions in the subscript. The specification $c:($c), for example, says that variable $c is aggregated using the sequence constructor (). The asterisk serves as wildcard for specifying a default aggregation expression.

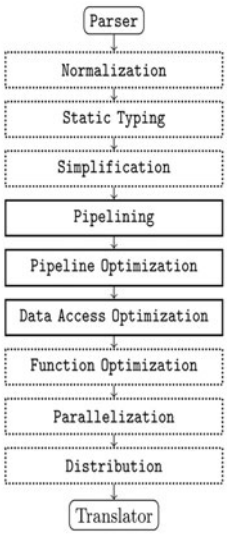
4 Compilation Process

The compilation phase consists of several rewriting stages and the final assembly of the executable plan. An overview of the rewriting process is given in Fig. 7.

4.1 Rewriting and Optimization

At first, the initial expression tree created by the parser is normalized. In contrast to other compilers, however, we do not normalize everything to language constructs of the XQuery Core model, because this would also break up all path expressions into sequences of FLWOR expressions. As we will see in Sect. 6, it is benefical to preserve them in the first place. Thereafter, the query is checked for static typing errors and annotated with typing information, which can be used for data-type-specific optimizations in subsequent steps.

Fig. 7 Query rewriting pipeline



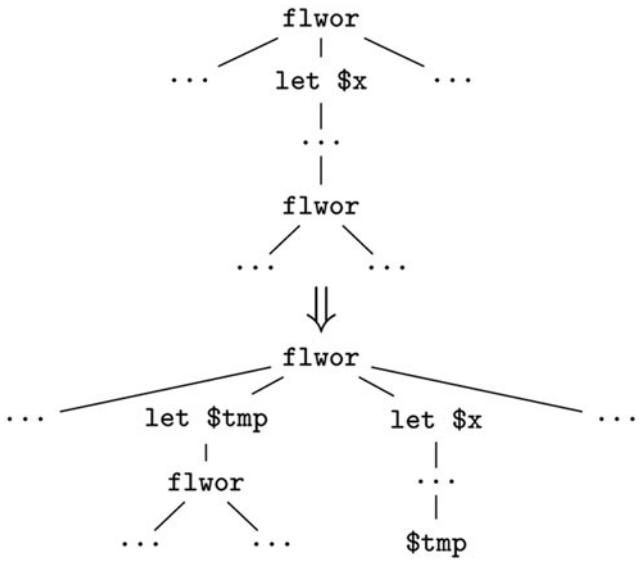


Fig. 8 Pull-out of nested FLWOR's

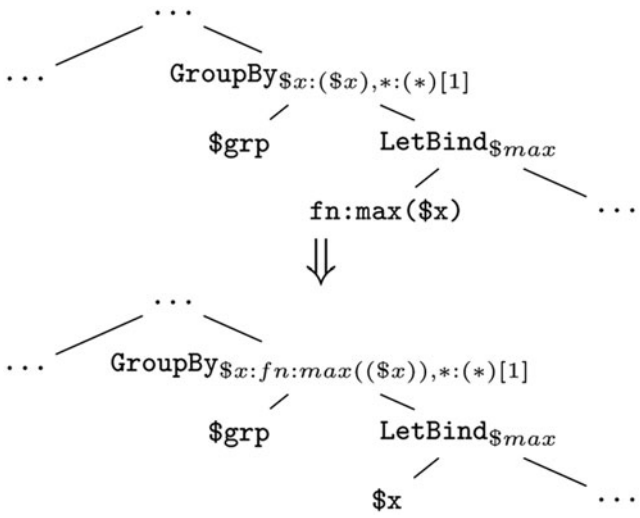


Fig. 9 Inlining of aggregation functions

The simplification stage performs standard pruning operations like the removal of unused variables and constant folding. Furthermore, the expression tree is prepared to simplify data access and pipeline rewritings in the following stages. For example, deeply nested FLWORs are extracted into let clauses of superordinate FLWOR as exemplified in Fig. 8. This increases chances for pipe lifting. Similarly, one can extract shared prefixes from path expressions to avoid multiple evaluation of the same paths at runtime.

Note, we perform here rule-based optimization and some of these rewritings might be sub-optimal in some queries or for a particular data set. Currently, we leave this field explicitly open for more elaborate cost-based optimization, which exploits statistical information [11].

In following stages, we transform FLWOR expressions into operator pipelines as explained in Sect. 3 and perform optimizations like predicate (i.e., Select operators) pullup, predicate combination, and sort elimination. Whenever possible, we also defer binding operators, i.e., push them down in the tree. This reduces the size of intermediate context tuples and saves CPU time if context tuples are filtered before the bound variable is referenced the first time. The same rationale applies to GroupBy and OrderBy operators. Generally, optimization is similar to relational systems; we only have to observe variable dependencies between operators.

Another promising source for efficiency gains is the analysis of the surrounding expressions of variable references. As example, aggregates of variables can be computed directly in GroupBy operations to save the memory for buffering the grouped values (see Fig. 9).

The sketched list of possible rewritings is certainly not exhaustive, but it illustrates the manifold dimensions for pipeline improvements.

Next, we perform pipe lifting as presented in Sect. 3. In the join rewriting stage, we identify joins as explained in detail in Sect. 5 and re-arrange the linear pipeline to a more efficient join tree.

The following two stages take care of the physical data access operations (see Sect. 6 for details) and calls to user-defined functions (see Sect. 4.3). These are the central stages for platform-specific optimizations. For example, the framework allows to insert specialized operators to efficiently handle certain operator combinations at once. A typical class of such operators are tailored twig operators for XML processing [12].

Finally, we envision two additional stages to introduce data and operator parallelism in the pipelines and to take care of data partitioning and query processing in distributed environments.

4.2 Plan Generation

The final expression tree is compiled into an executable query plan in a single pass. The compilation process is straightforward, because expressions do not depend on their ancestors except for variable references. Necessarily, subexpressions have to be compiled first. Only local variables in the visibility scope of a subexpression must be declared beforehand and undeclared again when they go out of scope in the declaring expression. External variables are visible throughout the query unless they are shadowed by local declarations.

The context item, context size, and context position are special kinds of variables. They are either referenced implicitly by an expression or explicitly via `.`, last() and position().

We treat them consistently like ordinary variables, i.e., expressions that modify theseparts of the dynamic context (e.g., filter expressions), declare them as special variables, which are resolved during compilation of expressions referring to these elements.

A variable table keeps track of all declared variables and their corresponding scopes. At the end of the compilation pass, we use this table to resolve all variable references to positions in the context tuples. Accordingly, a variable reference translates at runtime into a constant array lookup in a context tuple.

Pipelines are compiled equally simple. Beginning at the Start operator of the corresponding pipe expression, the pipeline is translated in a recursive top-down pass along the output edges. The binding and resolution mechanism for variables is also identical.

Note that the actual realization of an operator pipe-line is independent of the compilation process. The system can either employ a pull-based or a push-based operator model. The pull-based model is widely used in all kinds of database systems, because it matches the bottom-up perspective of conventional query plans. The push-based model is almost identical, but better suits to the top-down perspective.

4.3 Functions

Functions turn XQuery into a real programming language. There are two types: built-in functions and declared functions. Built-in functions are (native) operations provided by the runtime. The standard already defines a substantial library for various purposes, e.g., aggregation, string manipulation, and access to external resources. Declared functions are user-defined subqueries which can be directly or indirectly recursive.

Function parameters are handled like external variables in a query that are bound as arguments by the caller. To call a function, we evaluate the argument expressions, perform dynamic type checking if necessary, and pass the arguments as a context tuple to the function. In some situations, we must perform dynamic type-checking for result sequence, too. This is the standard way to handle a function call; and also the only way how to invoke a built-in function.

In contrast to already optimized built-in functions, optimization of declared functions is a task for the query compiler itself. Non-recursive functions, for example, can be inlined to avoid the overhead of function invocation and to increase chances for further optimization. The function call is replaced in the expression tree by the function body and references to function arguments are replaced by the corresponding argument expressions. Again, special care has to be taken to discover typing errors at runtime.

Recursive functions can benefit from well-known compiler techniques like tail-call optimization [13], and higher-order functions and partial function application can draw from the compilation of functional languages [14].

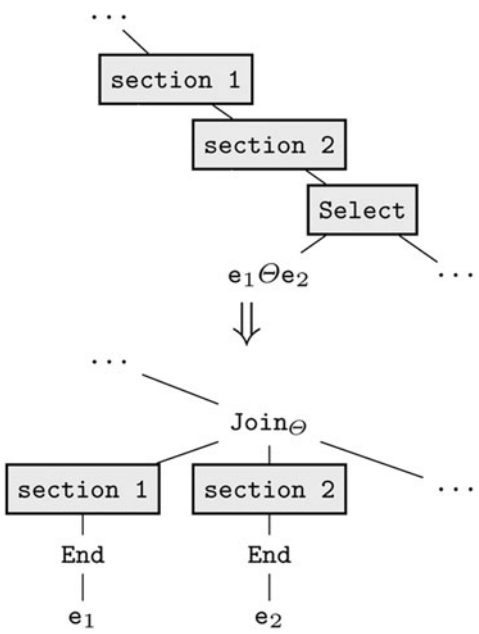


Fig. 10 General join pattern

5 Join Processing

Efficient join support is the most important way to fight the data explosion of a Cartesian product. In contrast to SQL, however, joins in XQuery are sometimes non-trivial to detect and to compute. For simplicity, we split therefore the discussion of join detection and the actual join processing.

5.1 Join Recognition

Fortunately, it is most of the time relatively easy to identify join semantics in a pipeline. One has to look for a Select operator with a comparison predicate $\Theta \in \{=, <, >, <=, \dots\}$ over two loop-independent operand expressions. We speak of loop independency, when all variable references in one operand and all variables they transitively depend on are independent of the variables referenced in the other operand and bound after the latest-bound variable referenced by the latter. In other words, there are two consecutive sections of binding operators followed by a Select, in which all variable bindings of the second section are independent of variables bindings of the first section, and the operand expressions of the predicate disjointly refer to one of them. Figure 10 illustrates the situation. Section 2 does not reference a variable of section 1 and the operand expression e_1 refers only to variables bound in section 1 or above, whereas operand

expression e_2 refers only to variables bound in section 2 or before section 1.

When we find this pattern in a query, we can rewrite the linear pipeline to the form shown at the bottom of Fig. 10. The Select is replaced by a Join operator with the two independent sections 1 and 2 as outer input and inner input, respectively. The operand expressions of the predicate provide the join keys.

Recall the first join rewriting for the pipeline in Fig. 2. The left-hand operator expression of the equality comparison is the variable reference $a, the right-hand operator expression is $b. Obviously, $b is bound in the pipeline after $a and the binding expression (2,3,4) is independent of $a, too. This allows us to replace the Select with a join and to move the ForBind of $b to the right join branch as shown in Fig. 3.

Figure 11 shows a more difficult example. In the upper expression tree, the predicate refers disjointly to the consecutive, independent variables $a and $b. However, the right-hand operand of the predicate expression also references variable $c, which depends on $a and is bound after variable $b. Accordingly, the predicate must be rearranged and the respective LetBind operator must be moved up as shown at the bottom of Fig. 11 to build the join.

Obviously, reshaping of a pipeline to match the join pattern can become arbitrarily complex. Especially, if the join predicate has to be adjusted as in the example above. Accordingly, there is no guarantee that a potential join will always be detected. In practice, however, it seems reasonable to assume that predicates with join semantics are rather simple.

Join semantics may also occur in filter expressions of the general form $s_2[e_2 \Theta e_1]$, where e_2 is a predicate over the current context provided by s_2 and where e_1 is a predicate over the context of a surrounding loop over a sequence s_1. In this situation, we can rewrite the filter expression to an equivalent FLWOR [1] and compute the join in the pipeline.

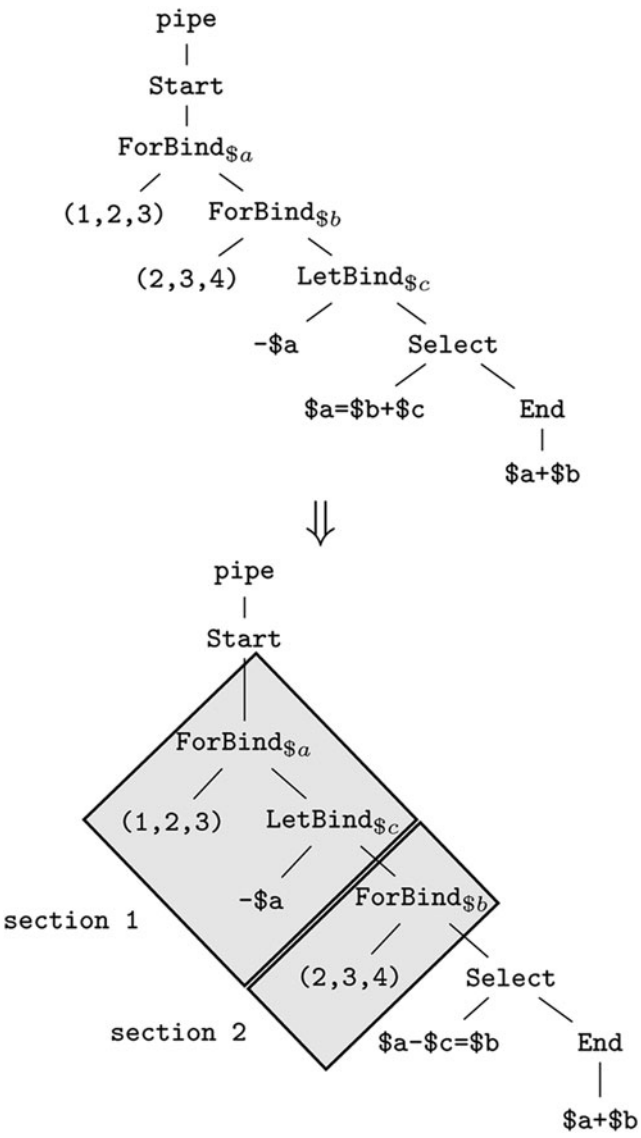


Fig. 11 Join pattern extraction

5.2 Algorithms

Some aspects of XQuery make join processing a little bit more complex than in SQL. First, there is the strong emphasis on order. All operations must preserve the input order unless reordering is explicitly required (e.g., in orderBy clauses) or if the requirement is overidden (e.g., by fn:unordered()). Second, comparisons in XQuery follow subtle typing and type conversion rules – a concession to work smoothly with untyped and semi-structered data. In consequence, a join operator has to cope with untyped data and mixed-type sequences if the types of the operands cannot be statically determined.

As a result of the above restrictions, a sort-merge join is only useful if the operands can be determined to be of a single type and if output order can be ignored or is compensated by an explicit sort afterwards. More typically, join algorithms will keep one input (the inner) in a memory table, e.g., a hash table or a sorted lookup table and probe it with the other input (the outer). We refer to [15] for implementation details.

At this point, we come back to the discussion of variable dependencies of the two join branches1 and 2 illustrated in Fig. 10. Conceptually, both branches must be re-evaluated for every incoming context tuple. This is particularly undesirable for the table-based join algorithms mentioned above, because a re-init-ialization of the right input translates to costly clearing and re-building of the lookup table.

In the best case, both sections and both operand expressions are free of external variable references so that the entire join must be computed only once. This reflects the common situation in relational settings. In the general case, however, both input sections and operand expressions have dependencies to variables bound upstream. Naively, one could try to resolve this situation by duplicating and prepending the com-

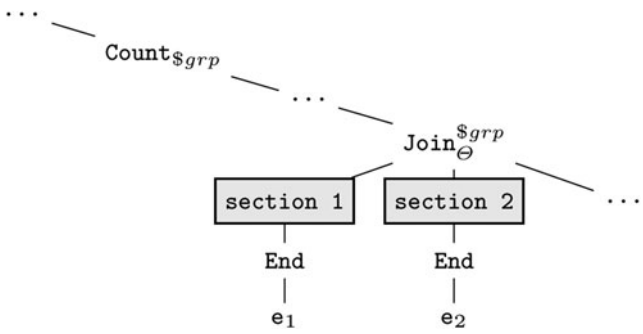


Fig. 12 Join group demarcation

mon input pipeline to both join branches. Both inputs could then be processed once and completely independent of each other. Depending on the size and complexity of the copied section, however, this may perform even worse than before. Therefore, one will in practice rather need to find a middle way between both extremes.

For our prototype we developed a special technique called *join group demarcation*, which minimizes the number of (re-) initializations for the inner join table. As depicted in Fig. 12, it introduces an artificial `Count` operator just after the closest variable binding on which the inner join branch depends. By checking for changes of this variable, the join operator can easily decide if the already built join table can be reused for an incoming context tuple or if it must be discarded. As the experiments in Sect. 7 show, this strategy can lead to substantial performance gains. Further details can be found in [16].

6 XML and Data Abstraction

XQuery knows two different kinds of data: Atomic types like strings, numerics, and date types, and XML for modeling structured and semi-structured data. The language provides convenient primitives for navigating, constructing and updating of XML structures. Because the compiler does not presume a specific physical data layout, each aspect is open for platform-optimized code. In fact, the seamless support of storage-specific compilation is one of the biggest strengths of the compiler. For brevity, we focus the discussion on navigation because it is the most performance-critical part.

XPath is the XQuery language subset for navigating XML trees. It describes path patterns and content predicates as sequences of path steps that evaluate to sorted, duplicate-free sequences of nodes. Each path starts from a sequence of context nodes, which are either bound to a variable or part of the current context sequence. Typically evaluated through iterative traversals of the XML tree, we can assume that a respective navigational solution can be realized in every system.

The performance penalty of generic navigation routines, however, varies considerably between different storages. The main source of inefficiency is the overhead of translating operations between different abstractions. For example, consider a path step operation, which returns the right sibling of an XML node in the tree. It will translate to a cheap pointer dereference operation for a linked tree structure in main memory, but translates to an expensive scan if the XML tree is stored in a relational table on external memory.

The second source of inefficiency is indirectly caused by a mismatch in the granularity of operations used by the compiler and supported by the storage layer. While node-wise tree traversal is for instance quite efficient in main memory, it is usually prohibitively expensive to navigate data on external storage.

The third source of inefficiency results from locality effects. The order and volume in which data is accessed by a query will considerably affect on performance. Again, this is particularly true for data on external storage, but even operations on in-memory data will be penalized by poor locality.

6.1 Platform-Specific Compilation

The clear separation of set-oriented data flow aspects and data representation in the compiler makes it easy to replace generic path expressions with platform-specific equivalents. In the following, we sketch some of the optimization strategies for leveraging efficient bulk methods in favor of costly node-wise traversals. A detailed discussion of data access optimization be found in [16].

Figure 13 gives an example of a typical situation in a native XML database system [5, 17]. Assume static typing can resolve that variable $n is bound to XML nodes in the native XML store, which supports both random access and efficient XPath-like scan access. The compiler can make use of the native operations by replacing the path expression, e.g., with a call of a built-in scan() function. Depending on the storage and the size of the data, this can easily affect query time by orders of magnitude. The fs:ddo function (distinct

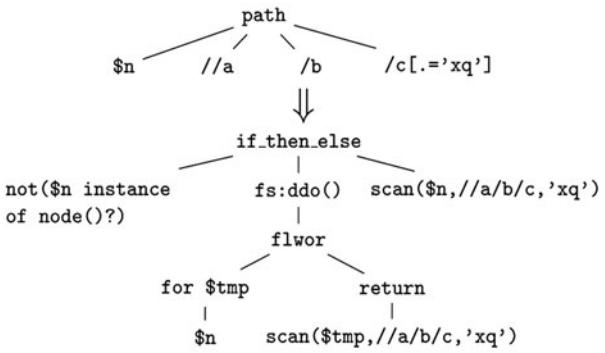


Fig. 13 Platform-specific path processing

document order) is used to enforce the correct ordering of the output and to guard against duplicates if $n is bound to a sequence of nodes.

As the example shows, the core principle of platform-specific compilation is relatively simple. Static typing and resolution of the starting sequences of path expressions (variable $n in the example) help to identify all data sources involved in the query. With respect to the capabilities of the actual store, the compiler can then select the most efficient, logically equivalent evaluation strategy.

Besides the mentioned bulk access and scan operations, index-support is crucial for most database applications. In the literature, many different types and variations of XML indexes have been proposed. There are typed and untyped context indexes, indexing schemes for elements and attributes with varying clusterings, path indexes, and combined structure and content (CAS) indexes. Whatever is available in a system to deliver the data as specified in the query is an appropriate replacement for potentially slower navigational access.

Of course, the optimizer must not be restricted to a single path expression at a time only. In most applications, e.g., it is wise to apply filtering by content very early because it drastically reduces the amount of data to be processed. For this reason, focused CAS indexes that index content of nodes on specified paths are often ideal in XML databases [17]. The availability of a CAS index for the pattern //product/price[xs:double], e.g., is ideal to speed-up the following query:

```
for $p in //product
where $p/price > 100.00
return $p/name
```

A lot of research has also been done in the field of path and twig matching algorithms for XML [12]. This class of algorithms mainly implements specialized XML-aware join variants to match structural patterns in a document. As desired, these algorithms may be provided as operators or expressions to the compiler.

Clearly, further discussion of specific XML storage techniques is out of scope of this paper. However, the above nicely exemplifies the wide range of possibilities available for improving a query with platform-specific operations.

Of course, the search space for the optimizer quickly grows with the number of alternatives to evaluate a path expression. However, the general complexity of a query, i.e., the number of intermediate results, the join ordering, etc. is usually not affected by the choice for the physical data access and can be determined in the pipeline rewriting stages. A feedback loop or a subsequent reshaping of the pipeline is advisable if physical properties like, e.g., access locality or sort orders can be exploited.

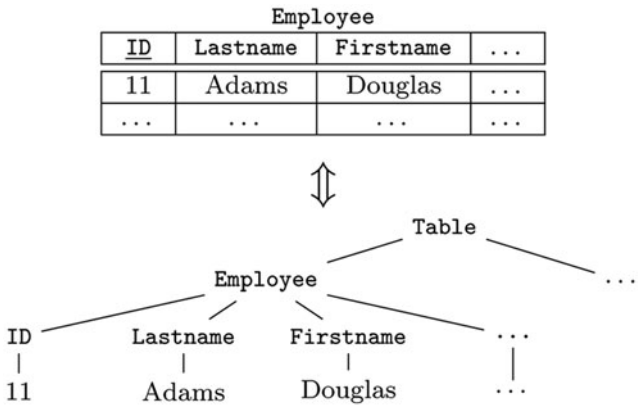


Fig. 14 Mapping of a relational table

6.2 Non-XML Data

Conceptually, the focus on XML is not a limitation because any kind of data can also be encoded as XML. In practice, however, it is often desirable to process also non-XML data (e.g., tabular data or records) without prior conversion. In fact, we can easily make use of XQuery's bulk processing and programming capabilities by abstracting from its data model as sequences of semi-structured values.

For illustration, consider the simple mapping of a relational table like in Fig. 14. Each row of the table can be considered as a tree structure of height 3. For this mapping, the XQuery

```
for $e in io:table('emp.tbl')/Employee
where $e/Lastname = 'Adams'
return $e/ID
```

is equivalent to the SQL query

```
SELECT ID
FROM Employee
WHERE Lastname = 'Adams'
```

whereby the function io:table returns the table data from a file emp.tbl logically as navigatable tree.

The compiler can derive that the io:table function actually returns a sequence of tuples, for which it must not perform the standard navigation routines to evaluate the path expressions. Instead, it can completely bypass the navigation and translate it directly into tuple field accesses. Accordingly, the sample query can be processed without the need to convert or wrap the base data and it runs as efficient as a table scan in a relational database.

In general, such logical mapping works for any kind of hierarchical data model – even with deep and complex nestings. This allows us to make use of the query compiler for process-

ing any kind of structured or semi-structured data format such as JSON. With such flexibility we can even address data processing tasks beyond classic database scenarios like driving distributed computations in MapReduce environments [18].

7 Experiments

We realized the presented concepts in our open-source XQuery engine called *Brackit*.[4] It fully supports XQuery 1.0, the group by and count clauses of XQuery 3.0, and implements the XQuery Update Facility.

7.1 Main-Memory Processing

In our first experiment, we compared the main-memory performance of our engine against current state-of-the-art competitors with the widely accepted XMark benchmark. Because of the spread in runtime and system performance, we choose the scale factor 0.01, i.e., the benchmark queries were evaluated on 12 MB documents in main memory. For all benchmarks, we used a dual Intel Xeon server with 2.66 GHz and 4 GB main memory running Ubuntu Linux 10.04 64-Bit. Java-based engines were tested with Oracle Java 1.6 64-Bit and we ensured to perform a sufficient amount of warmup runs for optimizations of the JIT compiler before the measurements. The results were serialized to a /dev/null output stream.

Table 1 shows the fastest runtimes for each query and system out of 10 timed runs. Clearly, our prototype shares the lead with Qizx and delivers high performance for each query type. The results for the queries Q8-Q12 also reveal that Qizx is the only other engine for main memory that comes with join support. All other engines suffer from the nested-loops semantics in these queries. Saxon achieved slightly better results than the other engines without join support, because it creates on-demand indexes at runtime, which help to reduce the effects of quadratic scaling. Figure 15 summarizes the results of the five fastest processors in this experiment graphically.

Table 1 Main-memory performance on XMark benchmark scale factor 0.1 (12 MB) in ms

	Q1	Q2	Q3	Q 4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20
Brackit	1	3	4	3	1	2	3	4	5	16	5	3	<1	4	<1	1	1	3	3	2
BaseX 6.7	1	4	9	8	2	22	54	3066	3504	440	6777	1571	6	35	3	3	6	4	21	11
MXQuery 0.6.0	106	147	200	147	142	166	299	1087300	1536482	8354	1605515	112463	150	158	116	104	134	161	199	381
Qexo 1.11	9	11	16	Err	Err	Err	Err	Err	287	67	Err	Err	5	17	6	8	7	1	24	Err
Qizx 4.1	2	5	3	1	1	2	5	4	5	15	4	2	1	4	<1	<1	1	2	3	2
Saxon HE 9.3	1	3	6	4	2	1	4	1288	1537	262	752	233	7	19	1	2	4	3	19	4
VXQuery 0.1	3	23	43	19	6	181	371	5058	1756	Err	8969	8863	9	123	4	11	30	Err	81	18
XQilla 2.2.4	11	17	53	33	16	23	44	12426	14755	1598	40922	13342	28	48	4	7	18	17	56	93
Zorba 1.4.0	6	14	31	40	7	<1	132	6466	10305	1554	6404	2116	31	53	4	6	22	9	67	38

7.2 XML Database Processing

In our second experiment, we used the XMark benchmark with scale factor 1 to assess *BrackitDB*, a native XML database (XDBMS) with full ACID support that internally uses *Brackit* as compiler. The XML store of *BrackitDB* is inspired by the path-oriented storage of the native XDBMS XTC and supports the same set of advanced CAS, path and

[4] Source code available at http://brackit.org

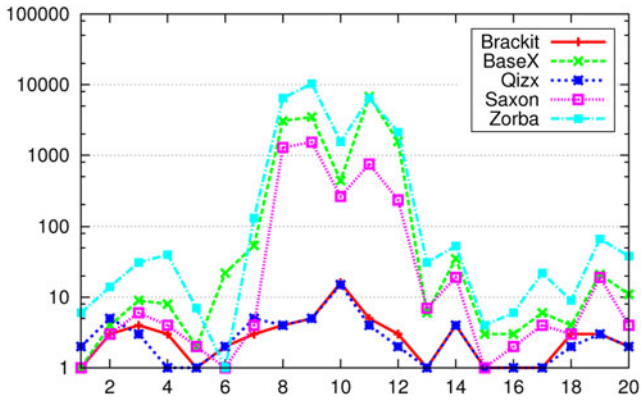


Fig. 15 Top 5 main-memory performance in ms

content indexes [17]. For the benchmarks, however, we did not create any indexes.

We tested the system in two configurations. In the first configuration, queries were compiled in the same manner as in the main-memory setup, i.e., the XML tree was navigated and filter predicates were applied within the engine itself. In the second configuration, we enabled storage-specific optimizations and let the compiler push down the node name filters to the storage layer to speed up child and descendant navigation.

For comparison, we ran the queries also on other XML (-enabled) database systems. As in BrackitDB, we did not perform manual tuning of the systems, e.g., by defining additional indexes, but note that some systems (e.g., BaseX) create indexes per default. We solely adjusted the maximum heap size for the Java-based systems to 1.5 GB, which was sufficient for each system to handle the workload.

Table 2 and Fig. 16 show the fastest result out of 5 timed runs for each system. The ability to recognize and process joins efficiently again proves to be an advantage over most competitors. Only MonetDB is also able to perform the join queries without a drastic decrease in performance. Due to the different storage designs, the general performance characteristic of each system is less homogeneous than in the main-memory experiment. Some systems achieve extraordinary fast results for certain queries, but this is often not the result of superior compilation techniques. The queries hit a sweet spot of the underlying XML storage or a suitable (default) index is applicable (e.g., Q1 in BaseX).

The results for *BrackitDB* draw generally a balanced picture for each of the 20 queries. It is not a surprise that the storage-specific compilation outperforms the default configuration. The relative speed-up for a query varies between 4 % and 67 %. Note, the difference results solely from the optimization of physical data access operations. The crucial set-oriented aspects including correlated nestings have been optimized in both configurations in the independent rewriting and join rewriting stages.

Table 2 XML database performance on XMark benchmark scale factor 1 (112 MB) in ms

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20
BrackitDB	80	139	391	435	107	1680	3531	445	1444	4094	3614	2273	157	1714	172	187	205	170	1806	545
BrackitDB opt.	65	137	396	356	96	201	1478	408	1394	3395	3523	2200	148	979	52	87	194	157	1411	345
BaseX 6.7	3	75	181	161	56	150	1242	917	1427	2902	1638091	381363	176	841	41	37	80	93	288	142
Exist 1.4.0	172	348	1071	295672	160	23	74	3237363	5388218	257660	7344781	7342385	99	540	78	380	405	168	855	660
MonetDB 1.1.11	52	88	220	409	61	30	44	273	334	1260	760	501	96	508	81	98	127	74	244	210
Sedna 3.4.66	43	398	240	137	27	9	21	560674	723272	66860	975952	224120	271	762	30	33	474	184	899	142

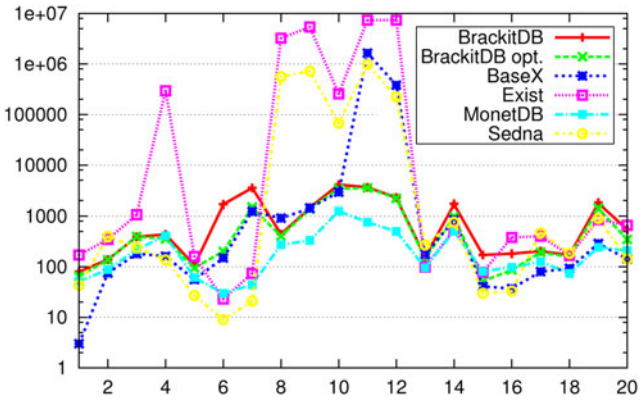


Fig. 16 XDBMS performance in ms

The curve shapes indicate that the quality of our query plans is competitive throughout all queries. For the simple queries, however, BrackitDB did not reach the best marks of the other systems. A closer look at the queries and the competitors reveals that query plan quality is here not the decisive factor. Merely, the other systems benefit from faster storage engines or had additional indexes available.

7.3 Scalability

We repeated the benchmark for the scale factors 0.01-10 (12 MB-11 GB) with the optimized configuration to investigate the scalability of the system. The results are shown in Fig. 17.

In general, the system scales well to large data volumes and does not reveal any unexpected effects. The result size of Q11 and Q12 grows quadratically with the size of the document. Thus, the growth in response times is characteristic for the workload and does not result from suboptimal query plans. For the 11 GB document instance, we observed that the response time for some queries, e.g., Q19, grows a bit more than before. This effect is caused by additional data accesses for the result construction in the `return` clauses. This is typical for XQuery and leads to poor data locality

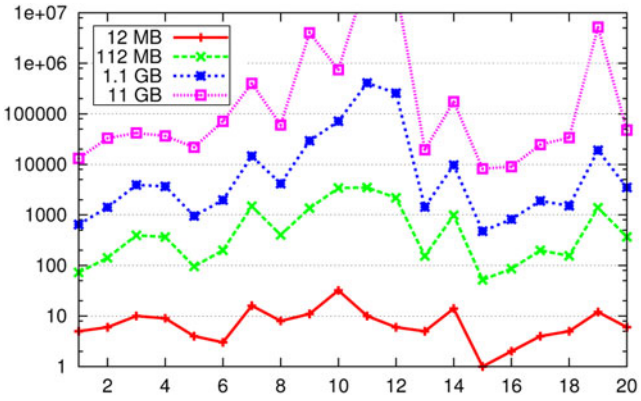


Fig. 17 Scalability of *BrackitDB* in ms

and random I/O, which results in longer response times. To avoid this, the query engine should "piggy-back" data for the result construction whenever possible.

7.4 XQuery for Relational Data

Our last experiment showcases the versatility of XQuery when paired with our compiler. We took the dataset of the relational decision support benchmark TPC-H and the SQL queries Q2 and Q6 of the benchmark. The query Q2 is a complex join query over 5 tables and a correlated subquery with another join of 4 tables. Query Q6 is a simple filter and aggregation scan over a single table.

We translated the SQL queries to equivalent queries in XQuery as exemplified in Sect. 6 and evaluated them over normal files in which we stored the relational data. We tested two setups. In the first setup, we ran the queries directly on the '|'-separated text files generated by the *dbgen* tool of the benchmark. For the correct datatypes, our custom io:table function loaded the schema information from a separate XML configuration file. In the second setup, we stored the table data in files with a simple binary encoding.

For comparison, we report also the response times for the relational databases PostgreSQL 8.4 and DB2 9.7. To get comparable results, we did not create indexes. Instead, we created detailed statistics and configured both systems with sufficient memory for the database buffer and the queries to ensure that the systems could perform the computation completely in memory. The generated database had a size of 1 GB (scale factor 1). Figure 18 shows the fastest results on hot buffers out of 5 runs for each system.

The pictures of the two queries look very different. In the complex join query, *Brackit* is more than one order of magnitude faster than the relational systems. This clear result is caused by a better handling of the correlated subquery, which is performed in all systems using nested loops. Like both DBMS, our compiler generated a bushy operator tree with hash joins and a final sort. But in contrast to the relational

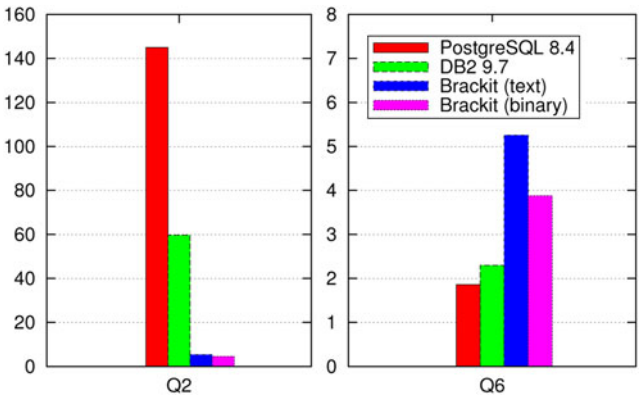


Fig. 18 TPC-H queries Q2 and Q6 in sec

systems, our prototype is able to reuse hash join tables in the nested query between iterations, as detailed in Sect. 5.2.

The query `Q6` scans the lineitem table (726 MB raw data, ∼6 Mio rows) and sums up the revenue of each qualified row. In this discipline, both relational systems are about 2 times faster than our prototype on binary files. However, this is again the result of a more efficient scan-and-filter logic in the database systems than the result of a superior query plan.

In summary, this experiment underlines the great utility of pairing a versatile language like XQuery with a set-oriented, storage-independent runtime. With minimal effort, we can efficiently perform general data processing tasks on top of different data models and representations.

8 Related Work

A broad overview of XQuery processing models and the realization of nested, iterator-based evaluation in various products is presented in [2]. In the following, we give a brief overview of the unnesting techniques employed by other set-oriented compilers. A more detailed discussion can be found in [16].

Pathfinder [8] is a compiler for relational database backends, which requires to have all data, i.e., atomic values, sequences, and entire XML documents, encoded in a ternary table layout. Variable bindings and iteration scopes are *loop-lifted* to turn nested-loops into operations on "unrolled" tables. A specialized join-operator speeds up XML processing. Pathfinder has been proven to be very efficient on large data sets. It is especially successful on top of the MonetDB backend [19] because the ternary table layout and the frequent equi join operations for loop-lifting suit the system's infrastructure. However, the strict relational view complicates the sometimes subtle semantics of XQuery (e.g., in comparisons) as well as the functional aspects. Furthermore, loop-lifting causes query plans to quickly grow in size and complexity [20].

XQGM is the logical, tuple-based operator graph representation of a query in XTC [6]. So-called correlated edges indicate context dependencies, i.e., nestings, between operators. Extensive unnesting rules eliminate these correlations and apply independent join operations instead. A great variety of physical operator alternatives is available to compile the resulting XQGM into an executable plan. Query rewriting and optimization in XQGM is clearly more complex than in our approach, because the correlated edges turn the operator tree into a directed graph.

NAL [7] is an algebra of tuple-based operators, which accommodate XQuery's FLWOR bindings. After transforming a query to an algebraic form, the compiler uses a set of equivalence rules to rewrite nestings to more efficient set-oriented constructs. Although the approach is generally portable to different architectures, it bases on nested tuples, which complicates variable handling during compilation and the implementation of efficient physical operators.

The Galax compiler [15] features a complete XQuery algebra that distinguishes between XML operators, tuple operators, and a third group of explicit boundary operators, which connect the two other parts of the algebra. Similar to our compiler, expression trees are compiled directly and FLWORs are compiled to tuple operator trees. Nestings are modeled as dependent join operations. Galax also supports basic optimizations for unnesting and value-based joins. Further optimizations are not mentioned.

9 Conclusions and Outlook

This work prosposes a novel compiler framework that builds on XQuery's bulk processing and data abstraction capabilities. The compiler extracts the language-inherent looping semantics of FLWOR expressions and turns them into relational-style operator pipelines. The pipelines are then subject to set-oriented optimization rules and algorithms. The concept developed is fully composable, i.e., query nestings may be arbitrarily deep and rewrite rules are independent of the surrounding expression. Furthermore, the approach is not limited to certain query constructs and custom functionality can be plugged in flexibly as functions, custom expression types and even custom operators.

We also showed how the storage-independent design of the compiler allows to use the query language even for other structured and semi-structured data types like relational tuples or JSON. This seamless mapping to other data formats at the logical level is complemented by the possibility to compile data access operations to efficient native operations of the underlying data store.

In experiments, we demonstrated the general efficiency of the concept. Our prototype achieves top performance in classic XML processing in main memory and yields competitive query plans when compared to the fastest native XML database systems. In an experiment with XQuery evaluated on top of relational data, we could also impressively show that efficiency is not a matter of the query language, but of the suitable abstraction. Altogether, this brings us one step closer to our goal – a swiss army knife for processing large amounts of various kinds of data.

At the logical level, we envision for the future the adaptation of state-of-the-art algorithms like join enumeration, statistics, cost-based query optimization. Likewise interesting is the examination of how structures and idioms of concrete data models interact with applicability and effectiveness of optimization rules. Likely, one will identify sets of universally applicable and specific optimizations, which can be bundled to optimization profiles for different targets.

At the physical level, we scratched so far only at the surface of possible optimizations. Aside smart tracing and mapping routines for particular classes of storages, plenty of challenges are worth to look at. In the days of cache-optimized storage structures and algorithms, for example, a portable compiler framework needs to go new ways for augmenting query plans with additional information for exploiting and optimizing data locality.

References

1. Draper D, Dyck M, Fankhauser P, Fernández MF, Malhotra A, Rose K, Rys M, Siméon J, Wadler P (2010) XQuery 1.0 and XPath 2.0 Formal semantics (2nd Edition)–W3C recommendation 14 December 2010. http://www.w3.org/TR/xquery-semantics/. Accessed 12 May 2014
2. Bamford R, Borkar VR, Brantner M, Fischer PM, Florescu D, Graf DA, Kossmann D, Kraska T, Muresan D, Nasoi S, Zacharioudaki M (2009) XQuery reloaded. PVLDB 2(2):1342–1353
3. Kay M (2008) Ten reasons why Saxon XQuery is fast. IEEE Data Eng Bull 31(4):65–74
4. Meier W (2003) eXist: An open source native XML database. WWsDS 2593:169–183. http://link.springer.com/chapter/10.1007%2F3-540-36560-5_13
5. Grün C (2010) Storing and querying large XML instances. Dissertation, University of Konstanz
6. Mathis M (2009) Storing, indexing, and querying XML documents in native database management systems. Dissertation, TU Kaiserslautern
7. May N, Helmer S, Moerkotte G (2004) Nested queries and quantifiers in an ordered context. ICDE 239–250
8. Grust T, Rittinger J, Teubner J (2008) Pathfinder: XQuery off the relational shelf. IEEE Data Eng Bull 31(4):7–14
9. Robie J, Chamberlin D, Dyck M, Snelson J (2010) XQuery 3.0: An XML query language – W3C working draft 14 December 2010. http://www.w3.org/TR/xquery-30/. Accessed 12 May 2014
10. Graefe G (1993) Query evaluation techniques for large databases. ACM Comput Surv 25(2):73–170
11. Weiner AM (2011) Advanced cardinality estimation in the XML query graph model. BTW 180:207–226. http://www.bibsonomy.org/bibtex/2de252266ade0e6d8c56ddcd5bfdf5729/dblp
12. Bruno N, Koudas N, Srivastava D (2002) Holistic twig joins: optimal XML pattern matching. SIGMOD 310–321
13. Abelson H, Sussman GJ, Sussman J (1985) Structure and interpretation of computer programs. MIT Press Cambridge, MA, USA. http://dl.acm.org/citation.cfm?id=26777
14. Peyton Jones SL, Lester DR (1992) Implementing functional languages: a tutorial. Prentice Hall, New York
15. Re C, Siméon J, Fernández MF (2006) A complete and efficient algebraic compiler for XQuery. IEEE Computer Society ICDE 14. http://dl.acm.org/citation.cfm?id=1129874
16. Bächle S (2013) Separating key concerns in query processing – set-orientation, physical data independence, and parallelism. Dissertation, TU Kaiserslautern
17. Mathis M, Härder T, Schmidt K (2009) Storing and indexing XML documents upside down. CSRD 24(1–2):51–68
18. Sauer C, Bächle S, Härder T (2013) Versatile XQuery processing in MapReduce. ADBIS 8133:204–217
19. Boncz PA, Grust T, van Keulen M, Manegold S, Rittinger J, Teubner J (2006) MonetDB/XQuery: a fast XQuery processor powered by a relational engine. SIGMOD 479–490
20. Grust T, Mayr M, Rittinger J (2009) XQuery join graph isolation: celebrating 30+ years of XQuery processing technology. ICDE 1167–1170