

Implementing Memoization in a Streaming XQuery Processor

Yanlei Diao^{1,2}, Daniela Florescu¹, Donald Kossmann¹,
Michael J. Carey¹, Michael J. Franklin²

¹ BEA Systems, U.S.A.

{danielaf, donaldk, mcarey}@bea.com

² University of California at Berkeley, U.S.A.

{diaoyl, franklin}@cs.berkeley.edu

Abstract. In this paper, we describe an approach to boosting the performance of an XQuery engine by identifying and exploiting opportunities to share processing both within and across XML queries. We first explain where sharing opportunities arise in the world of XML query processing. We then describe an approach to shared XQuery processing based on memoization, providing details of an implementation that we built by extending the streaming XQuery processor that BEA Systems incorporates as part of their BEA WebLogic Integration 8.1 product. To explore the potential performance gains offered by our approach, we present results from an experimental study of its performance over a collection of use-case-inspired synthetic query workloads. The performance results show that significant overall gains are indeed available.

1 Introduction

XQuery [18], while not yet a standard, is already being put to use in commercial software infrastructure products for a number of different IT purposes. For example, the XQuery language (and its sub-language XPath) has been incorporated into several products for business process management and application integration. XQuery is used in several ways there – as a transformation language for defining XML data transformations, as an expression language for making branching and looping decisions based on XML workflow variables, and as a filtering and routing language for handling message broker events. XQuery is also being used in enterprise information integration products that provide virtual XML views of disparate enterprise data sources where it is the language for defining integrated views and writing queries.

As XQuery adoption gains momentum, the performance of XQuery processing becomes increasingly important. As with any query language, XQuery is amenable to a large number of optimizations, both at compile time and at runtime. In many of the uses to which XQuery is being put, significant optimization opportunities can be obtained through the discovery and exploitation of shared processing, within or across queries. For example, in publish/subscribe, query evaluation work can be shared when matching messages against a large number of subscriptions [5]. In this paper, we in-

investigate the exploitation of such sharing opportunities to boost the performance of XQuery processing. In particular, we develop *memoization* techniques for XQuery and apply them in the context of a commercial streaming XQuery processor.

Sharing in XQuery processing. Intuitively, intermediate results of XQuery processing can be shared whenever the “same” XQuery expression(s) would otherwise be evaluated more than once with the “same” XQuery variable bindings. (We will say more about what “same” means in this context in Section 2.) This can happen in several ways:

1. The same expression can occur several times in different locations within a query.
2. The same expression can occur in different queries that are evaluated together.
3. An expression can occur within a query that is evaluated multiple times (most likely with different variable bindings).
4. An expression can occur in different queries that are executed at different times (where the query context is the same across executions).

The first case is self-explanatory. The second case arises in contexts like publish/subscribe, where an incoming XML message needs to be checked against many subscription queries. An example of the third and fourth cases is a web service call or a remote database lookup modeled as an XQuery function call, where the results of the call are known to be stable over time (at least for some specified time period).

In this work, we propose a memoization-based approach to avoiding redundant work. Memoization caches the results for an expression based on its variable bindings, and it can thus support evaluation reuse in all of the above cases.

Streaming XQuery processing. Our approach is designed to work well in the context of an XML query processor that employs stream-based processing. In the context of XML query processing, streaming is important for performance, and it can occur at a fine level of granularity. A fine-grained approach is critical given that a single XML item can be arbitrarily large, containing the equivalent of an entire table’s or even database’s worth of data content. To enable fine-grained streaming, the BEA XQuery engine [6], the engine on which this work is based, represents its XML operands as sequences of (potentially nested) tokens that represent smaller constituent data pieces.

The use of a token stream representation of XML provides an XQuery processor with several ways to achieve incremental query evaluation while avoiding the materialization of its inputs. The first way is *pipelining*. A given XQuery expression can consume and produce token streams incrementally, materializing only one or a few tokens at a time in order to compute and emit its output. Of course, this requires the use of a pull-based API to be truly effective. The second way is *lazy evaluation*, a technique commonly used in the implementation of functional programming languages [11]. With this technique, a result is not actually generated until requested by a consuming expression. Moreover, in XQuery, some expressions can be evaluated based on only the first few tokens of a given input – for example, `nth(,)`, `empty(,)`, `exists(,)`, existential comparators, and positional predicates. These expressions enable an even lazier mode for XQuery processing, where only those (possibly few) tokens needed for the consuming expression are generated.

Contributions. In this paper, we present a memoization-based approach to sharing in XQuery processing. While both the multiple-query processing (MQP) problem [17] and the use of memoization for query processing [10] have been explored in other contexts, our contributions lie in the fact that shared XQuery query processing in a streaming environment adds significant new wrinkles to the problem. In particular, MQP in the relational setting has focused on SELECT-FROM-WHERE style constructs, whereas our work is aimed at supporting sharing for the much richer XQuery language. Memoization has been exploited for expensive functions (as in query processing) or repeatedly computed functions (as in dynamic programming), but it has not been studied for a large variety of XQuery expressions and in a stream-based processing environment. The main contributions of this work can be summarized as follows:

1. We set the scope for XQuery memoization, first in a simple but limited way, and then in an expanded range exploiting semantic data and expression equivalence.
2. We develop a number of query compilation techniques to identify interesting shareable XQuery expressions and to determine the granularity of memoization.
3. We also extend the runtime system, resolving the inherent tension between stream-based processing and memoization. Our solutions enable computation reuse while supporting pipelining and avoiding eager evaluation.
4. We summarize results from a performance study of our techniques in the context of the BEA XQuery engine. The results show significant performance gains for typical use cases of XQuery.
5. As this paper represents our initial approach towards adding memoization to XQuery processing, we identify several important open problems to be addressed.

The paper is as follows. Section 2 discusses basic issues related to XQuery memoization. Section 3 describes the BEA XQuery engine, the technical context of this work. Sections 4 and 5 describe how we have added memoization to this engine, focusing on the compile-time and runtime aspects, respectively. Section 6 reports experimental results. Section 7 covers related work. Section 8 concludes the paper.

2 Basics of XQuery Memoization

In this section we address the basic issues related to XQuery memoization. Memoization is an algorithmic technique that remembers the results returned by functions invoked with particular arguments and, if the function is called with the same arguments again, returns the result from memory rather than recalculating it [11, 13]. In the context of XQuery, the unit of computation that we adopt for memoization is the (XQuery) expression.

In its simplest form, XQuery memoization can be implemented in a straightforward way: results of an expression are shared whenever an *identical* XQuery expression is evaluated more than once with *identical* XQuery variable bindings, where the meaning

of “*identical*” is based on bit-wise comparison of their binary representations.¹ The usefulness of such memoization, however, is limited by the stringent requirement of identical binary representations. To expand the scope for XQuery memoization, we would like to establish *equivalence* relationships between expressions and between variable bindings, so that ample reuse of the computed results is possible. To this end, we relax the conditions for the application of memoization along two dimensions: (1) when two XQuery *data model instances* can be determined to be equivalent; and (2) when two XQuery *expressions* can be determined to be equivalent.

XML Data Equivalence. What we seek is an equivalence relationship on XQuery data model instances that meets the following requirement for safe memoization: Given an expression E, for every pair of equivalent XQuery data model instances, the two results of evaluating E on the two instances are also equivalent. XQuery has multiple equality testing predicates (=, eq, is, deep-equal(), ...) to compare data model instances. Unfortunately, none of these is satisfactory for establishing data equivalence for safe memoization (the analysis is omitted here in the interest of space). As a result, we define our own, more comprehensive (but still imperfect) equivalence relationship between XQuery data model instances:

Definition 1 (XML data equivalence) Two data model instances are equivalent *iff* one of the following conditions is true: (a) they both represent the empty sequence, or (b) they are both single atomic values, their primitive values are equal (based on the **eq** comparison on their respective primitive XML data types), and their type annotations are also equal (based on the **eq** comparison on their *xs:QName* data types), or (c) they are both nodes and they compare true via the **is** comparison, or (d) they are both sequences of the same length $l \geq 1$ and the corresponding items in the *i*th position ($1 \leq i \leq l$) are equivalent via the conditions (b) or (c).

Unfortunately, memoization based on this definition is not safe for every possible XQuery expression. For example, consider the memoization of the *string()* function for two *dateTime* instances that use different time zones but have the same normalized values (i.e., the same UTC time). Based on Definition 1, the two *dateTime* instances are equivalent (via condition (b)), however, the results of applying *string()* to these instances are not equivalent, due to the different time zones included in the output strings. As such, memoization in this case would cause erroneous results.

Given our goal of exploiting semantic data equivalence for memoization and the fact that doing so correctly for the full XQuery language is a very hard problem, our current solution is restricted to a subset of XQuery for which Definition 1 is guaranteed to provide safe memoization. Roughly speaking, every expression in this subset is such that each variable of the expression satisfies one of the following conditions: (1) the type of the free variable is a node; (2) the type of the free variable is an atomic type and the computation performed by the expression is compatible with the **eq** comparison defined on this type; or (3) the type of the free variable is a sequence of items and the items in the sequence have the same type that satisfies condition (1) or (2). As our experimental results show, even this limited definition of equivalence can provide

¹ Of course, expressions that produce non-deterministic results are not suitable for memoization. Examples of such expressions include functions that read the system time (e.g., *fn:current-dateTime()*), and user-defined functions that are declared to be variant.

significant performance improvements for XQuery processing in use cases similar to those that we would expect to see in web services, application integration, etc.

Expression equivalence. In general, two XQuery expressions are the same if and only if they return the same result for every correct binding of their variables. This question is undecidable in general, since XQuery is Turing-complete. As a result, we identify sufficient conditions for XQuery expression equivalence based on expression normalization and detection of syntactical equivalence between normalized expressions (which will be described in detail in Section 4). As our experimental results show, these conditions permit ample reuse of computations (given typical use cases) while also being efficiently computable.

Thus, our approach represents a practical compromise between an overly restrictive definition of XQuery memoization and the difficulties that arise due to the fully-general nature of XQuery. While we believe that our approach is applicable in a large number of practical situations, expanding its range is part of our ongoing work.

3 The BEA Streaming XQuery Processor

In this section we review the aspects of the BEA XQuery engine [6] that are directly relevant to our subsequent design descriptions. The representation of XML data used internally by the BEA XQuery runtime is a sequence of tokens called the *token stream*. Despite its similarity to the SAX API, the token stream models typed XML, and is accessed via a pull-based API for producing and consuming tokens lazily. Moreover, the BEGIN tokens for documents, elements or attributes are augmented to carry the ids of the nodes in order to compare nodes for both equality and document ordering. Further details on the token stream can be found in [6].

3.1 Query Compilation

The purpose of the XQuery compiler is to parse, verify, type check, normalize and optimize a query. The result of compilation is an iterator tree that can be interpreted by the runtime system.

During compilation, a query is represented as an *expression tree*. Nodes in an expression tree represent kinds of expressions and edges represent data flow dependencies. The kinds of expressions used by the BEA XQuery processor are very close to the W3C XQuery formal semantics recommendation [20], and include *Constants*, *Variables*, *FirstOrder* expressions, *SecondOrder* expressions, *IfThenElse*, *Node Constructors*, etc. All built-in functions and operators of the XQuery standard [19] share the same representation – each is a *FirstOrder* expression. Examples include all XPath axes (e.g., child, descendant, parent), the *data()* function, arithmetic functions, comparisons, and constructor functions for simple types (e.g., *float()*, *string()*). The *Match* expression carries out an XPath nodetest (i.e., kind of node and name of node). A *Node Constructor* creates a new XML structure. In this particular processor, node id generation is decoupled from Node Constructors and postponed until later when

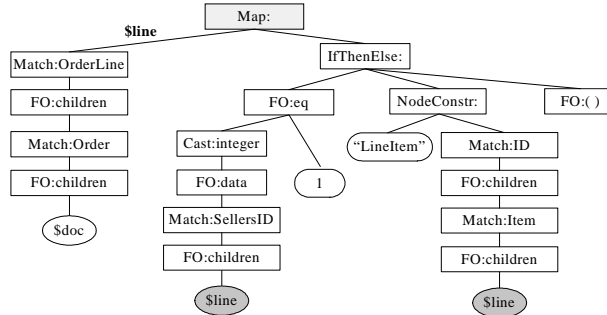


Fig. 3-1. Expression tree of query Q1.

needed for query evaluation.² The family of *SecondOrder* expressions can be further classified into *Map*, *Let*, *Sort*, etc., most of which represent the high-order functions of XQuery. *Map* will be described more closely in the example below.

The translation of an XQuery into an expression tree closely follows the W3C XQuery formal semantics [20]. For example, the *for* clause of a FLWOR query is translated into nested *Maps*, each of which defines one variable; the *where* clause into an *IfThenElse* expression; and so on. Consider query Q1 below, which requests line items in a purchase order document that have a particular seller.

Query Q1: *for* \$line *in* \$doc/Order/OrderLine
where xs:integer(data(\$line/SellersID)) eq 1
return <LineItem> {\$line/ItemID} </LineItem>

Fig. 3-1 shows the expression tree for this query, where constant expressions are shown as rounded rectangles, variables as ovals, and all other expressions as rectangles. Expressions other than constants and variables are labeled with the kind of expression (“FO” for FirstOrder), followed by a colon, followed by any optional specifications of an expression, such as a particular FirstOrder function (e.g., *children*()) or *data*()) as well as any other parameters (e.g., the *NodeTest* of a *Match* expression). *Map* is the only *SecondOrder* expression in this example. Note that the left child of the *Map* expression is labeled with the name of the variable defined in the *Map*. Uses of the variable in the right child of the *Map* are denoted by shaded ovals.

A *free variable* of an expression is a variable that is not defined by any second order expressions inside this expression, essentially representing an input of the expression. For example, “\$doc” is a free variable of the *Map*, but “\$line” is not. However, “\$line” is indeed a free variable of *IfThenElse*, the right child expression of *Map*, and of the expressions inside *IfThenElse* that contain this variable. We call a mapping between the free variables of an expression and a set of values a *binding*.

At the last step of query compilation, the compiler generates code for the query expression, resulting in an executable *iterator tree*. There is a one-to-one mapping between many of the nodes in the expression tree and iterators in the generated iterator tree; however, a few iterators implement several expressions (e.g., the *children*() iterator implements a *child*() expression plus a *Match* expression) for performance

² This decoupling raises the potential for sharing the node construction computation.

reasons. Variable expressions are implemented by a special *runtime variable* iterator that returns the value of a variable and can be *bound* to different inputs at runtime.

3.2 Runtime System

The task of the runtime system is to interpret an iterator tree to produce the query result. Like many database query engines, the BEA XQuery runtime system is based on an iterator model [9]. Its query execution model is pull-based, and data is consumed at the granularity of tokens. Using the iterator model, the runtime system naturally exploits pipelining. It also makes use of lazy evaluation; that is, an iterator only generates results on demand, with each *next()* call. Consider the iterator that implements the *empty()* function. This iterator consumes only a single token from its input in order to produce a Boolean result. The remaining input tokens are not consumed, and thus are not even generated. Other expressions where lazy evaluation is effective include positional predicates (e.g., `$line/Item[1]`) and existential quantification.

4 Query Compilation for Memoization

In this section we describe the compile-time aspects of arranging for efficient evaluation of a set of XQuery queries (referred to as the “target queries”). We restrict our attention to the cases where the target queries share the static context and most of the dynamic context (except the date and time of execution).³ The techniques presented in this section focus on sharing among common subexpressions. Although not discussed below, sharing by caching expensive methods [10] can be easily supported by indicating such expressions to the compiler.

4.1 Expression Equivalence

Our approach to expression equivalence is based on two steps. First, all expressions are normalized by applying a set of rewriting rules. The rewriting rules include ones that “normalize” queries based on the XQuery formal semantics [20], and others that are typically applied in XQuery optimization, e.g., unnesting nested FLWOR expressions whenever possible, putting predicates in conjunctive normal form, etc. More details of these rewriting rules are provided in [6].

The second step searches for syntactical equivalence between expressions. Our approach to determining equivalence is based on the notion of *variable renaming substitutions*. We say that two expressions E1 and E2 are syntactically equivalent via a renaming substitution $S = \{x_1/y_1, \dots, x_n/y_n\}$, if $\{x_1, \dots, x_n\}$ are the free variables of E1, $\{y_1, \dots, y_n\}$ are the free variables of E2, and E1 and E2 are syntactically isomorphic, up to a renaming of (free and bound) variables. For example, the expressions $(\$x+1)*(\$x-3)$ and $(\$y+1)*(\$y-3)$ are syntactically equivalent via the renaming substitution $\{\$x/\$y\}$.

³ XQuery memoization in the presence of different static and/or dynamic contexts poses difficulties that are beyond the scope of this paper; we leave that generalization for future work.

Given this definition, we develop an algorithm that detects syntactical equivalence between two expressions E1 and E2. If E1 and E2 are syntactically equivalent via the renaming substitution $S=\{x_1/y_1, \dots, x_n/y_n\}$, the algorithm returns S , otherwise it returns *null*. In the sequel, we call this algorithm “equals()”.

Given two input expressions E1 and E2, equals() iterates on E1 and E2 and their subexpressions from top to bottom, checking recursively at each level for syntactic isomorphism. Obviously two expressions are not (syntactically) equal if they are not of the same kind (e.g., constants, variables, Maps, etc). Moreover, it is clear that the details of the recursive algorithm depend on the kinds of the expressions E1 and E2. XQuery has more than 15 kinds of expressions. While our algorithm handles all of them, for brevity, here we describe only three:

Constant expressions. If E1 and E2 are both constant expressions, then they are equal via a renaming substitution S iff the given constants are equivalent via the data equivalence Definition 1 given in Section 2.

FirstOrder expressions. If E1 and E2 are both FirstOrder Expressions, then they are equal via the renaming substitution S iff they have the same operator and the same number of children subexpressions, and the children subexpressions are pairwise equal via the same substitution S .

Map expressions. Assume that both E1 and E2 are Map expressions of the form:

E1=*for* \$var1 *in* expr1 *return* expr2

E2=*for* \$var2 *in* expr3 *return* expr4

First, the algorithm will test the structural equivalence of expr1 and expr3. If this succeeds with renaming substitution S then the algorithm continues; otherwise it fails. In the positive case, the renaming $\{\$var1/\$var2\}$ is added to the current renaming substitution S and the algorithm will continue by testing the structural equivalence of expr3 and expr4 via the new S . In case the test succeeds and an augmented substitution S is returned, the end result of the test is the substitution S *without* the renaming of the internal variables $\{\$var1/\$var2\}$. Otherwise the test fails.

Note that the complexity of the structural test equals() is linear in the size of the input expressions. Given that the potentially interesting expressions for sharing among the target queries include *all* the subexpressions of these queries, the structural test needs to be applied to all possible pairs in the Cartesian product of the subexpressions of the target queries, yielding a very expensive algorithm. Next, we describe the technique used in our implementation to avoid this exponential complexity.

4.2 Applying the Algorithm

Identifying common subexpressions is implemented as an additional step taken by the compiler after query parsing, normalization and optimization, but before code generation. In this step, the compiler iterates over the target queries and identifies common subexpressions both inside each query and between this query and earlier queries.

For each query, the compiler performs a depth first search in the query expression tree to identify the *maximal shareable subexpressions*: For each subexpression encountered that is not a constant or a variable, the compiler performs three tasks: (1) Apply *hashing* on the subexpression, ignoring all the variable names, and use the hashing result to probe the in-memory storage of all distinct subexpressions, each of

which serves as a representative of an equivalence class. (2) If representatives with the same hashing result exist, for each of them call equals() on the representative and the subexpression in hand. (3) If any representative is equivalent to the subexpression, apply *heuristics* to filter out uninteresting cases of common subexpressions (such as inexpensive operations, e.g., a simple addition, and expressions that are not very expensive but could return large results e.g., a child path expression with wildcards). If a representative passes all these tests, the compiler determines that the representative matches the subexpression, and stops further traversal into this subexpression. Otherwise, it updates the storage of equivalence classes with the unmatched subexpression and continues the search in the children of this subexpression.

As an example, consider query Q1 from Section 3 together with Q2 given below.

Query Q2: *for* \$item *in* \$doc/Order/OrderLine
where xs:integer(data(\$item/BuyersID)) eq 8
return <LineItem> {\$item/Item/ID} </LineItem>

After query Q2 is parsed, normalized, and optimized, it is represented by an expression tree similar to that in Fig. 3-1 except for the *if* expression (i.e., the leftmost branch of *IfThenElse*). Table 1 shows the results of the compilation actions applied to the expression tree of Q2 for identifying maximal shareable subexpressions after all subexpressions of Q1 have been processed. The rows contain the subexpressions considered in order of the depth first search. As Table 1 shows, Map is filtered by the first step of hashing because it contains a different path expression and a different constant in its *if* expression. Match:OrderLine and NodeConstr are the two maximal common expressions identified. Note that although FO:children and FO:() are equivalent via equals(), they are filtered by our heuristics as being uninteresting sharing cases.

Table 1. Identifying Common Subexpressions between Q1 and Q2

Expression	(1) hashing	(2) equals	(3) heuristics
Map	No		
Match:OrderLine	Yes	Yes	Yes
IfThenElse	No		
FO:eq	No		
Cast:integer	No		
FO:data	No		
Match:SellersID	No		
FO:children	Yes	Yes	No
NodeConstr	Yes	Yes	Yes
FO:()	Yes	Yes	No

The implementation of code generation is also modified to take into account the identified common subexpressions. The compiler again iterates over the query set in the same order. For each query, code generation proceeds recursively in the expression tree as before, except for common subexpressions. For the instances of a common subexpression, a *CacheIterator* (which will be described in detail in the next section) is created for each instance, but all such CacheIterators point to the same memo table, which is where the results of memoization are cached at runtime.

Fig. 4-1 shows the iterator trees generated for Q1 and Q2. The two queries have separate iterator trees. The structure of each iterator tree is similar to the expression

formance perspective, but it is more difficult to integrate into a stream-based XQuery processor. These two caching schemes are described in more detail in Section 5.2.

5.1 Memo Table Lookup

The purpose of a memo table is to map the values of the free variables of a common subexpression to a (completely or partially) cached result. To this end, the memo table is implemented as a hierarchy of hash tables. Each level in this hierarchy corresponds to one free variable and is probed using the value of that free variable. Probing a memo table with a set of values bound to the free variables results in either a reference to the cached result (i.e., a memo table hit) or a *null* pointer (i.e., a memo table miss).

In order to probe the memo table and to record new entries in the memo table in the event of lookup misses, it is crucial to know the values of the free variables. As stated at the beginning of this section, a naïve implementation of memo table lookup could break lazy evaluation and pipelined processing of these values, thus adversely affecting the performance. At this point, we place an important restriction on the notion of data equivalence used for memo table lookup: we disregard condition (d) in Definition 1 (given in Section 2) for establishing equivalence between sequences of items. In other words, we only cache results of an expression if the type of each free variable of this expression is either an atomic type (e.g., integer or date) or a node (e.g., element or document). We do not cache results if the type of a free variable is a sequence of items. Our implementation of this restriction is based on type checking on free variables at compile time.

The issue with lazy evaluation still exists even with this restriction. The value of a free variable can contain an arbitrarily large number of tokens (e.g., for a document), which might be produced by another complex expression that we wish to evaluate lazily. Fortunately, this restriction does enable us to probe the memo table by only looking at the first token of the value of each free variable. If the type of the free variable is a node, the first token will contain the *id* of the node and we can use this id to probe the memo table. If the type of free variable is an atomic type, instead, we can extract the whole value from the first token and use that to probe the memo table.

5.2 Result Caching

Implementing complete result caching is easy, once the memo table lookup issue has been solved. The queries that contain a common subexpression each instantiate a `CacheIterator` for this common subexpression. The `CacheIterators` of those queries, however, share the same memo table. This memo table is used in the following way: For each binding of the free variables, the memo table is probed as described before. If the result has been cached, the `CacheIterator` returns it token by token (whenever its `next()` method is called) from the cached result. If the memo table lookup fails, the common subexpression is fully evaluated using the current binding, the entire result is cached, and the memo table is updated with the (binding, result) pair. The next time when the common subexpression needs to be evaluated with the same binding of the free variables (within the same query or for another query), the stored result is reused.

Complete Caching is simple, but it computes the entire result of a common subexpression which may never be needed. For performance reasons, we would like to compute the results of a common subexpression just as lazily as in other situations; that is, results stored in the memo table are generated only when they are needed by the consumers. This gives rise to the idea of Partial Caching that is able to cache partial results across queries and compute additional parts of the results later, if needed.

To implement Partial Caching, we need to keep the iterator tree of the common subexpression in addition to the partial results. We will use this iterator tree when additional results are needed which have not been produced yet. Furthermore, we must preserve the state of all iterators in the iterator tree, in particular, the bindings of the iterators that represent the free variables in the iterator tree. In general, preserving such states across queries can be costly and may involve further materialization.

We currently focus on a special case, for which preserving the state is relatively simple; that is, the common subexpression is *resumable*. An expression is resumable if its free variables are bound only once in one invocation of the query execution. This condition can be checked at compile time based on the static types of the expressions that compute the values of the free variables. Common examples of such expressions occur in web services where path expressions are prefixed with an external variable that will be bound to each incoming message. For a resumable expression, we simply store the iterator tree together with the partial results in the memo table and use the iterator tree whenever additional results need to be produced (with guaranteed correct state of the iterator tree). Finally, some support is also provided to prevent a query from closing its iterator tree that has been stored in a memo table at the end of its processing. Details are omitted here due to space constraints.

6 Performance Evaluation

We have implemented our techniques for shared XQuery processing in a Java-based prototype system extending the BEA XQuery processor. In this section we evaluate the effectiveness of these techniques in the context of message brokering, where queries are executed over XML message payloads (or messages, for short). The workloads are derived from use cases collected from BEA customers. These use cases demonstrate common ways of using XQuery in practice.

Starting from these use cases, we created a set of workload queries based on the Purchase Order schema from the Universal Business Language (UBL) library [15]. We also created purchase order messages using a tool based on the AlphaWorks XML generator [1]. We used a set of 1000 10KB messages in our experiments. The performance metric used is *Query Execution Time*. This is the average time for executing a set of queries on each message from the input set. It does not include the message parsing time. We compared the performance of individual execution of the queries (“no caching”) and query execution with memoization (“caching”). Complete caching and partial caching perform the same, if not otherwise stated. All the experiments were performed on a Pentium III 850 Mhz processor with 768MB memory running Sun JVM 1.4.2 on Linux 2.4. The JVM maximum allocation pool was set to 500 MB. All data structures including the memo tables fit in the 500 MB allocation pool.

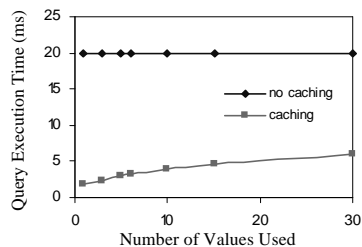


Fig. 6-1. Vary the number of distinct values used for the parameterized query

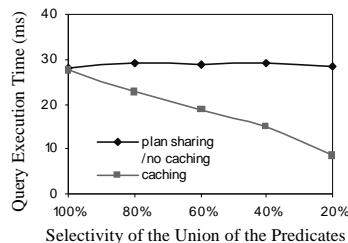


Fig. 6-2. Vary the selectivity of the union of the predicates

The first experiment was conducted in the context of subscriptions using parameterized queries, which is a common way that service instances subscribe to a message broker. The parameterized query that we used is provided below:

```

for $price in $doc/Order/OrderLine/Item/BasePrice/PriceAmount
where float(data($price)) lt $value
return $price

```

Thirty queries (i.e., subscriptions) were generated from this template. To obtain different degrees of query similarity, we varied the number of distinct values that the variable `$value` can take, called the *domain size*, from 1 to 30. For a given domain size n , the values between 1 and n were evenly distributed in the thirty queries.

The results are shown in Fig. 6-1. It is clear that memoization provides huge benefits for this workload. When all thirty queries use the same value, “caching” achieves a 10.7x performance gain. As the domain size increases, its performance benefit decreases slowly, obtaining a factor of 3.3 when every query uses a distinct value.

The next experiment focuses on the use case of message transformation for subscribers. In this case, the message broker provides a function called `summarizeOrderLine` (not shown here) for summarizing the line items of interest to each subscriber. An example subscription query is illustrated below.

```

for $line in $doc/Order/OrderLine
where xs:integer(data($line/Item/ID)) ge 1 and xs:integer(data($line/Item/ID)) le 20
return summarizeOrderLine($doc, $line)

```

In this test, we used five queries similar to the query above and fixed their selectivity to 20% each using a range predicate on the ID of line items. We varied the overlap among queries by changing the constants in the range predicates so that the selectivity of the union of the 5 predicates varied from 100% (no overlap) to 20% (full overlap).

With query overlap, the same *OrderLine* may satisfy multiple predicates, and memoization over the function `summarizeOrderLine()` can avoid redundant work among queries. This type of sharing, however, cannot be captured by the traditional plan-level sharing approach of the relational world [17]. Here, plan sharing is equivalent to individual execution of queries. The results of this experiment are shown in Fig. 6-2. It can be seen that as the overlap among queries increases, the performance of “caching” improves remarkably but that of “no caching” (or plan sharing) does not.

We also conducted experiments to evaluate the effectiveness of our approach for web services calls and message routing using path expressions, and to compare complete and partial caching for workloads where lazy evaluation is crucial. The results of these experiments show that significant overall performance gains are available.

7 Related Work

Our work is related to a number of areas in the databases and functional programming communities. We attempt to provide a rough overview of related work here.

Query execution techniques based on sorting or hashing have been used to eliminate redundant computation of expensive methods [10]. For SQL trigger execution, algorithms have been developed to extract invariant subqueries from triggers' condition and action [8]. These techniques used in the relational setting are related to ours for XQuery processing in the case of single expression, multiple bindings.

In the pioneering work on multi-query optimization [17], the problem of *MQO* is formulated as “merging” local access plans of a set of queries into a global plan with reduced execution cost. Along this line, advanced algorithms have been proposed to approximate the optimal global plan [4, 16]. Our work addresses XQuery--a much richer language--and uses different techniques for identifying common subexpressions. Moreover, as our results show, our work provides finer-grained (i.e., binding-based) sharing of intermediate results than merging relational-style access plans.

There has been a large body of work on materialized views, which *precomputes* the views used by a set of queries and stores the results to speed up query processing [7]. In contrast, our techniques consider expressions with parameters, and cache and recover expression results “on the fly” while running a set of queries. View selection [1, 3, 14] decides what subqueries to materialize based on cost models and/or reliable statistics, similar in spirit to our technique of selecting interesting shareable computations. Using partial result caching, our approach has the advantage of avoiding unnecessary materialization over the cached computations.

XQuery memoization differs from memoization in functional programming [11, 12] in two aspects. First, instead of named functions, the memoization target for XQuery is expressions, making effective detection of shareable expressions critical. Second, while memoization in functional programming is usually based on a simple “values in, value out” execution model, our approach is realized in a complex processing model that produces results lazily and pipelines them to the extent possible.

8 Conclusions

In this paper, we described a memoization-based approach to sharing in XQuery processing. We first provided an analysis of data and expression equivalence for XQuery memoization. We then addressed issues related to efficient implementation. We developed a number of query compilation techniques to identify interesting shareable expressions, and extended a pipelined runtime system with efficient memo table lookup and different caching schemes. All of these techniques have been implemented in the context of a commercial XQuery processor and their effectiveness was demonstrated using query workloads derived from the common uses of XQuery in practice.

While our results are promising, we view this as a first step towards solving the problem of efficient sharing in XQuery processing. There are many interesting and important problems to be addressed in our future work. First, as data and expression equivalence is crucial to the applicability of memoization, a thorough analysis in the

context of the full XQuery language will be a main focus of our work. Second, we will consider normalization rules that rewrite queries especially to enable memoization. Such rewriting is aimed at turning variables of an expression from node-based to value-based, thus expanding the opportunities for reusing the computed results. Third, a complete solution to partial result caching that supports lazy evaluation requires further investigation. Finally, due to the diverse characteristics of shareable expressions, selective memoization is key to the performance of memoization. We plan to extend the set of compile-time heuristics and develop runtime adaptability to select those shareable expressions beneficial from the cost point of view.

Acknowledgements

We would like to thank Andrew Eisenberg for pointing out the *eq* anomaly that arises when applying the function `string()` to `dateTime` objects.

References

1. Agrawal, S., Chaudhuri, S., and Narasayya, V. Automated Selection of Materialized Views and Indexes in SQL Databases. In *Proc. of VLDB 2000*, 496-505.
2. AlphaWorks. XML Generator. Aug 2001. <http://www.alphaworks.ibm.com/tech/xmlgenerator>.
3. Chirkova, R., Halevy, A., and Suci, D. A Formal Perspective on the View Selection Problem. In *Proc. of VLDB 2001*, 59-68.
4. Dalvi, N., Sanghai, S., Roy, P., et al. Pipelining in Multi-Query Optimization. In *Proc. of PODS 2001*, 59-70.
5. Diao, Y., and Franklin, M. Query Processing for High-Volume XML Message Brokering. In *Proc. of VLDB 2003*, 261-272.
6. Florescu, D., Hillery, C., Kossmann, D., et al. The BEA/XQL Streaming XQuery Processor. In *Proc. of VLDB 2003*, 997-1008.
7. Gupta, A., and Mumick, I.S. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Bulletin*, 18(2), 1995, 3-18.
8. Llirbat F., Fabret F., and Simon E. Eliminating Costly Redundant Computation from SQL Trigger Executions. In *Proc. of ACM SIGMOD 1997*, 428-439.
9. Graefe, G. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys* 25(2), 1993, 73-170.
10. Hellerstein, J., and Naughton J. Query Execution Techniques for Caching Expensive Methods. In *Proc. of ACM SIGMOD 1997*, 423-434.
11. Hudak, P. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys* 21(3), 1989, 359-411.
12. Hughes, J. Lazy Memo-functions. In *Functional Programming Languages and Computer Architecture*. Springer-Verlag LNCS 201, 1985, 129-146.
13. Michie, D. "Memo" Functions and Machine Learning. *Nature*, 218, 1968, 19-22.
14. Mistry, H., Roy, P., Sudarshan, S., et al. Materialized View Selection and Maintenance Using Multi-Query Optimization. In *Proc. of ACM SIGMOD 2001*, 307-318.
15. OASIS. Universal Business Language. Feb. 2003. <http://www.oasis-open.org/committees/ubl/>.
16. Roy, P., Seshadri, S., Sudarshan, S., et al. Efficient and Extensible Algorithms for Multi-Query Optimization. In *Proc. of SIGMOD 2000*, 249-260.
17. Sellis, T. Multiple-Query Optimization. In *ACM TODS* 13(1), 1988, 23-52.
18. XQuery 1.0: An XML Query Language. <http://www.w3.org/TR/xquery/>.
19. XQuery 1.0 and XPath 2.0 Functions and Operators. <http://www.w3.org/TR/xpath-functions/>.
20. XQuery 1.0 and XPath 2.0 Formal Semantics. <http://www.w3.org/TR/xquery-semantics>.