Reihe Informatik
TR-2006-011

Kappa-Join: Efficient Execution of Existential Quantification in XML Query Languages

M. Brantner[1] S. Helmer[2] C-C. Kanne[1] G. Moerkotte[1]

[1]University of Mannheim
B6, 29
68131 Mannheim, Germany
brantner|kanne|moerkotte@informatik.uni-mannheim.de

[2] Birkbeck College
University of London
United Kingdom
sven@dcs.bbk.ac.uk

Kappa-Join: Efficient Execution of Existential Quantification in XML Query Languages

Matthias Brantner[1]*, Sven Helmer[2], Carl-Christian Kanne[1], and Guido Moerkotte[1]

[1] University of Mannheim, Mannheim, Germany
brantner|kanne|moerkotte@informatik.uni-mannheim.de
[2] Birkbeck College, University of London, London, United Kingdom
sven@dcs.bbk.ac.uk

Abstract. XML query languages feature powerful primitives for formulating queries involving comparison expressions which are existentially quantified. If such comparisons involve several scopes, they are correlated, and become difficult to evaluate efficiently.
In this paper, we develop a new ternary operator, called Kappa-Join, for efficiently evaluating queries with existential quantification. In XML queries, a correlation predicate can occur conjunctively and disjunctively. Our decorrelation approach not only improves performance in the conjunctive case, but also allows decorrelation of the disjunctive case. The latter is not possible with any known technique. In an experimental evaluation, we compare the query execution times of the Kappa-Join with existing XPath evaluation techniques to demonstrate the effectiveness of our new operator.

1 Introduction

Almost every XML query language features a construct that allows to express an existentially quantified comparison of two node-set valued subexpressions in a concise manner. Unfortunately, current XML query processors lack efficiency and scalability when facing such constructs [5, 21]. The corresponding semantics resembles that of nested and correlated subqueries, which are notoriously difficult to evaluate efficiently. To illustrate this point, let us consider the following query: For hiring a teaching assistant, we search the database for a student who took an exam that was graded better than 'B'.

```
for      $s       in    //student
let      $best    :=    //exam[grade < 'B']/@id
let      $exams   :=    $s/examination/@id          Q1
where    $exams = $best
return   $s/name
```

Here, both sides of the comparison in the `where`-clause are set-valued because there are many good students and students take more than one exam. The existential semantics of the XQuery general comparison operator requires that all students are returned which have at least one exam also contained in the set `$best`.

A naïve evaluation technique evaluates the steps in order of appearance. In Q1, this means to reevaluate the value of `$best` and `$exams` for every iteration of the for loop,

* This work was supported by the Deutsche Forschungsgemeinschaft under grant MO 507/10-1.

and then check for an item that occurs in both sets. This is a wasteful strategy: A closer look at Q1 reveals that in contrast to `$exams`, the value of `$best` does not depend on the value of `$s`, making the reevaluation of `$best` unnecessary. A common strategy in such cases is to move the evaluation of `$best` out of the for loop, and to materialize and reuse the result. However, this only alleviates the problem of repeated evaluation of the expression to which `$best` is bound. To answer the query, it is still necessary to evaluate the `where`-predicate, which is a correlated nested query due to the existential semantics of the '=' operator and the fact that it refers to variables from two scopes, the independent `$best` and the dependent `$exams`.

In this paper, we are concerned with an efficient evaluation of existentially quantified *correlation predicates* such as the `where`-clause of Q1. While this area has received some attention in the past, we show that there is still unexploited optimization potential for typical query patterns in XML query languages, even in simple cases like our Query Q1. We propose the novel Kappa-Join operator that naturally fits into execution plans for quantified correlation predicates, is easy to implement and yet features a decorrelated evaluation algorithm.

Q1 is "simple" because the correlation predicate occurs on its own. What if the correlation predicates become more complex? Assume that we consider *either* good *or* senior students to be eligible for assistantship, as in the following query:

```
for      $s              in     //student
let      $best           :=     //exam[grade < 'B']/@id
let      $exams          :=     $s/examination/@id             Q2
where    $exams = $best  or $s/semester>5
return   $s/name
```

If the two clauses were combined with **and**, we could use techniques to decorrelate queries with correlation predicates that occur conjunctively. If the clauses were not correlation predicates, we could use techniques to improve performance for disjunctive predicates (e.g. Bypass operators [7]). However, there is no published technique to decorrelate *disjunctively* occurring correlation predicates.

Hence, we also present a Bypass variant of the Kappa-Join. This allows a decorrelated evaluation of disjunctively occurring correlation predicates, which has not been accomplished for any query language so far.

The main contributions of this paper are as follows:

– We introduce the novel ternary Kappa-Join operator that, while simple to implement, can efficiently evaluate complex correlated queries where the correlation predicate occurs conjunctively.
– We introduce a Bypass variant of the Kappa-Join that allows us to extend our technique to queries where the correlation predicate occurs in a disjunction.
– We provide experimental results, demonstrating the superiority of both the Kappa-Join and the Bypass-Kappa-Join compared to other evaluation techniques.

The remainder of this paper is organized as follows. In the next section we discuss basic concepts, such as dependency and correlation in XPath. In Sec. 3, we discuss the drawbacks of existing decorrelation approaches for XML query languages. Further, we introduce our novel Kappa-Join operator to efficiently evaluate queries with conjunctive correlation. Sec. 4 investigates the case of disjunctive correlation and therefore presents the novel Bypass Kappa-Join. In Sec. 5 we experimentally confirm the efficiency of our approach. The last Section concludes the paper.

2 XPath Query Processing

The problems discussed in the introduction affect most existing XML query languages. However, all of the involved issues occur even for the simple XPath language in its first version because it features nested predicates and existential semantics. In the following, we limit our discussion to XPath 1.0, because peripheral topics such as typing, query normalization, and translation into an algebraic representation can be presented in a simpler fashion for XPath than for the more powerful XQuery language. However, all of the techniques presented in this paper also apply for full-blown XQuery or similar languages, as long as they are evaluated algebraically (e.g. [25]). In fact, both queries from the introduction can be expressed in XPath syntax, as we will see below.

2.1 Normalization

The techniques presented in this paper are mainly developed to optimize comparison expressions with one dependent and one independent operand. To correctly identify such expressions, the first step of our normalization is to rewrite a predicate such that it consists of literals and the logic operators $\wedge$ and $\vee$. Each literal l consists either of a comparison or a negation thereof, i.e. l is of the form $e_1 \theta e_2$ or $not(e_1 \theta e_2)$, where $\theta \in \{=, <, \leq, >, \geq, \neq\}$.

One example for "hidden" comparisons are location paths or other node-set valued expressions when used as Boolean expressions. In such cases, we make the node-set valued expression the argument of an auxiliary *exists* function and compare its result to true, yielding a regular binary comparison expression.

Further, to provide efficient evaluation for disjunctively occurring comparison expressions, the second step of our normalization separates those literals that occur in a conjunction from those that occur disjunctively. To this end, we employ an operation for collecting all literals that occur conjunctively: A literal l occurs conjunctively in a predicate p_k if we replace all occurrences of l inside p_k by false and the resulting expression can be evaluated to false using Boolean simplification rules to eliminate Boolean constants, then the original literal l occurs conjunctively in p_k.

2.2 Context Dependency & Correlation

In this paper, we are concerned with efficient evaluation of existentially quantified comparison expressions that are *correlated*. In general, correlation occurs when a variable of a nested scope is compared with a variable from an enclosing scope. XPath does not have variables that can be declared by the user, but we can define correlation in terms of XPath *contexts*, as follows.

Every XPath expression is evaluated with respect to a given context, which is a sequence of *context items*. For our discussion, it is sufficient to use a definition of context item that is slightly simpler than the original XPath context item. A context item is a tuple with three components: the *context node*, the *context position*, and the *context size*. In XPath, there is one global context, which must be specified as parameter of the evaluation process. The value of some constructs depends on *local contexts* that are generated by other subexpressions. The constructs that refer to the local context are location steps, relative location paths, and calls to `position()` and `last()`. We call these expressions *dependent* expressions. Expressions whose value is independent of the local context are called *independent* expressions.

If we apply this terminology to Queries Q1 and Q2 from the introduction, given in XPath syntax, we have

$$\underbrace{//\text{student}}_{\text{independent}}[\underbrace{\text{examination}/@\text{id}}_{\text{dependent}} = \underbrace{//\text{exam}[\text{grade} <' \text{B}']/@\text{id}}_{\text{independent}}]/\underbrace{\text{name}}_{\text{dependent}} \qquad \textbf{Q1}$$

$$//\text{student}[\underbrace{\text{examination}/@\text{id}}_{\text{dependent}} = \underbrace{//\text{exam}[\text{grade} <' \text{B}']/@\text{id}}_{\text{independent}} \text{ or } \underbrace{\text{semester}}_{\text{dependent}} > \underbrace{5}_{\text{indep.}}]/\text{name} \qquad \textbf{Q2}$$

We now can define the term correlation for XPath as used in the remainder of this paper: A comparison expression with one dependent and one independent operand is called *correlation predicate*, because it compares a local context and an enclosing context. Both example queries presented in the introduction contain correlation predicates. A correlation predicate can occur both conjunctively and disjunctively. We call the former case *conjunctive correlation* and the latter *disjunctive correlation*. In Q1 there is only one comparison expression which is a special case of conjunctive correlation, i.e. one with only a single literal. Q2 is an example with disjunctive correlation.

3 Kappa-Join for Conjunctive Correlation

The key to an efficient evaluation of correlated queries is to avoid redundant computation, e.g. the evaluation of the inner independent expression. Such techniques are called decorrelation techniques and have been studied extensively in the context of relational and object-oriented systems [8, 10, 11, 17, 18, 28]. Similar techniques have been proposed for the evaluation of XQuery and XPath [5, 21]. One of them is an approach that applies decorrelation to existentially quantified comparison expressions [5]. However, this approach is suboptimal because unnecessary duplicates are generated and must be removed at the end of the evaluation.

The optimizations developed in this paper are presented in the form of algebraic operators. Hence, we need an algebra capable of evaluating XPath. We have chosen NAL as a perfect fit, since a translation from XPath to NAL is also available [4]. However, our approach is not limited to NAL and the translation of XPath into it, e.g. [25].

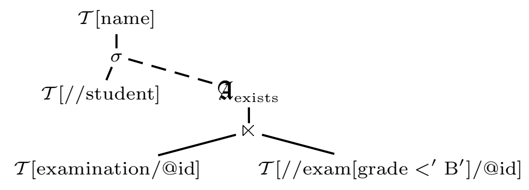
In this section, we describe our assumptions about algebraic translation and evaluation in more detail. For a more elaborate treatment of these topics, please refer to [4, 5]. We then recapitulate the decorrelation approach from [4] and discuss its drawbacks. Afterwards, we introduce the novel Kappa-Join operator that features an efficient decorrelation algorithm avoiding these drawbacks.

3.1 Algebra & Translation

The universe of the NAL algebra for XPath 1.0 is the union of the domains of the atomic XPath 1.0 types (`string, number, boolean`) and the set of ordered sequences of tuples which represent XPath contexts. Each tuple represents one context node, position and size. Special attribute names are used to hold the context node (cn), the context position (cp) and the context size (cs). We will provide explanations with an example below.

The NAL algebra features well-known operators [4, 21]. All the sequence-valued operators in the logical algebra have a corresponding implementation as an *iterator* [13] in the physical algebra. In the following, we primarily need *Selection* σ, the *Projection* Π, the *Semi-Join* $\ltimes$, and the *D-Join* $\underset{\rightarrow}{\bowtie}$. These operators are formally defined in our technical report [22].

To convert XPath queries into algebraic expressions, we use the translation introduced in [4]. We briefly recapitulate the relevant part of the translation process by elaborating on the translation result for Q1 (see Fig. 1). However, we omit the translation of subexpressions that are orthogonal to our discussion and denote them by $\mathcal{T}[e]$, where e stands for any XPath expression. For instance, we denote the translation of the location path `//student` by $\mathcal{T}[//\text{student}]$, its result is the sequence of context nodes produced by the location path.



$\mathcal{T}[\text{name}]$
σ
$\mathcal{T}[//\text{student}]$
$\mathfrak{A}_{\text{exists}}$
$\ltimes$
$\mathcal{T}[\text{examination}/@\text{id}]$
$\mathcal{T}[//\text{exam}[\text{grade} <' \text{B}']/@\text{id}]$

Fig. 1. Translation sketch for Q1

For Q1 (see Fig. 1) the algebra expression provides a selection (σ) for the only literal. In the subscript (denoted by a dashed line) of this selection, there is an Aggregation operator that aggregates the input sequence into a singleton sequence with a single attribute by applying the aggregation function $exists$. It returns true if there exists at least one tuple in the input sequence. This input sequence stems from a Semi-Join, whose input sequences in turn stem from the two operands of the comparison expression, i.e. the two (translated) location path expressions. For a comparison between two node-sets, as in the particular case of Q1, we have an existential semantics. In the equality case, this fact can be leveraged by using a Semi-Join.

3.2 Existing Decorrelation Techniques

We now recapitulate the decorrelation approach introduced in [5] and discuss its drawbacks. Again, we take Query Q1 to illustrate this (see Fig. 1).

The fundamental idea of decorrelation is to avoid unnecessary evaluation of the inner independent expression. In [5] this is achieved by pulling up the Semi-Join (see Fig. 1) into the top-level algebraic expression (see Fig. 2).

This expression needs some explanations. The dependent location path is connected to the outer expression using a D-Join ($\underset{\rightarrow}{\bowtie}$). The D-Join joins each tuple $t \in \mathcal{T}[//\text{student}]$ to all tuples returned by the evaluation of the dependent path $\mathcal{T}[\text{examination}/@\text{id}]$. For each t, $\mathcal{T}[\text{examination}/@\text{id}]$ is evaluated once, and free occurrences of variables in the dependent expression are substituted with the attribute values of t, i.e. the current context. At the end all resulting sequences are concatenated.[3]

The dependent expression, i.e. the evaluation using the D-Join, might produce duplicates for tuples from $\mathcal{T}[//\text{student}]$, hence the tid_A operator is needed to identify the tuples resulting from the outer expression.

The idea is to densely number the tuples, store this number in an attribute A, and use it later on to perform a duplicate elimination. To do this, we introduce an order-preserving duplicate elimination projection Π^{tid_A}, which removes multiple occurrences of the same tid-value A. It keeps the first tuple for a given A value and throws away the remaining tuples with the same value for A.

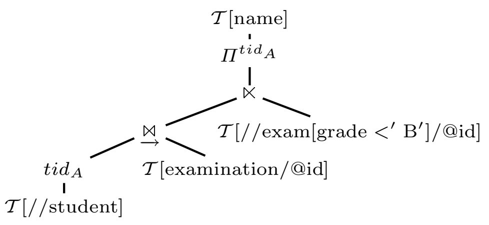


Fig. 2. Decorrelation for Query Q1

Clearly, the main advantage of this approach is that the independent expression is evaluated only once. In addition, if the Semi-Join's implementation uses a custom data structure (e.g. a hash table) to improve performance, this data structure has to be initialized only once, compared to one initialization per student in the naïve correlated evaluation from Fig. 1. However, decorrelation comes at a price: The outer expression produces duplicates which have to be eliminated. Below, we show how we can avoid them using the novel Kappa-Join. Our evaluation in Sec. 5 confirms this claim.

3.3 Kappa-Join

To avoid the above-mentioned generation of duplicates, but nevertheless gain performance by avoiding unneeded evaluations of the independent expression, we introduce the Kappa-Join operator. It combines the advantages of the evaluation strategies from Fig. 2 and Fig. 1 into one operator and capitalizes on efficient implementation techniques.

Logical Definition The *Kappa-Join* is a ternary operator, i.e. it has three argument expressions e_1, e_2, and e_3. It is defined by the equation

$$e_1 \kappa^{e_2}_{cn=cn'} e_3 := \sigma_{\mathfrak{A}_{x;exists(e_2 \ltimes_{cn=cn'} e_3)}}(e_1)$$

where cn is the context node resulting from the evaluation of e_2 and cn' the context node from e_3. As for conventional join operators, we denote the first and last producer expressions as *outer producer* and *inner producer*, respectively. The second producer expression e_2 (in the superscript) is called *link producer* because it acts as a link between the outer and inner producer within the join predicate. The outer expression e_1 and the

[3] In [25] this operator is called MapConcat.

inner expression e_3 are independent expressions, i.e. they do not depend on any of the Kappa-Join's other arguments. In contrast, the expression e_2 is dependent on e_1.

Informally, the result sequence of the operator contains all tuples from the outer producer (e_1) for which there exists at least one tuple in the link producer (e_2), when evaluated with respect to the current tuple of e_1, that satisfies the predicate p which is a comparison from attributes of e_2 and attributes of the the inner producer (e_3).

Translation with Kappa-Join There exist two alternatives to incorporate the Kappa-Join into an algebraic plan: (1) The Kappa-Join's definition matches the pattern that results from the canonical translation of correlation predicates, such as the Selection operator in Fig. 1. Hence, the Kappa-Join can replace this pattern directly after translation. (2) The other alternative is to modify the translation procedure such that a Kappa-Join is used for conjunctive correlation predicates.

Because our experiments (see Sec. 5) show that the Kappa-Join always dominates the canonical approach and it simplifies the translation procedure, we have chosen the second alternative.Fig. 3 contains the resulting algebra expression for Q1. Here, the outer producer of the Kappa-Join is mapped to the location path `//student`. The link producer is the dependent inner location path `examination/@id`, and the inner producer is the independent expression `//exam[grade<'B']/@id`.

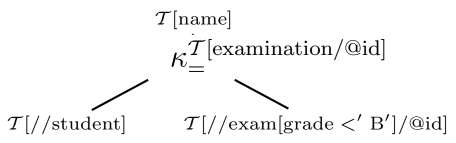

$\mathcal{T}$[name]
$\kappa_{=}^{\mathcal{T}\text{[examination/@id]}}$
$\mathcal{T}$[//student]
$\mathcal{T}$[//exam[grade <' B']/@id]

Fig. 3. Query Q1 with Kappa-Join

```
OPEN
while T ← INNERPRODUCER.NEXT
    do HASHTABLE.INSERT(T)

NEXT
while T_1 ← OUTERPRODUCER.NEXT
    do
        LINKPRODUCER.OPEN(T_1)
        while T_2 ← LINKPRODUCER.NEXT
            do
                if HASHTABLE.LOOKUP(T_2)
                  then
                      LINKPRODUCER.CLOSE
                      return T_1

        LINKPRODUCER.CLOSE
return nil
```

Fig. 4. Pseudocode for the Kappa-Join

Implementation The secret of the Kappa-Join lies in its simple, yet efficient implementation, which improves its performance beyond that of the operator combination in its logical definition. Fig. 4 shows the pseudocode for the implementation of the Kappa-Join as an iterator [13].

In its `open` method, the Kappa-Join builds a data structure, e.g. a hash-table, containing the attributes from the inner producer that are part of the join predicate. In its next method, the Kappa-Join initializes the link producer for every tuple T_1 from its outer producer. Like a Semi-Join, it then probes the hash table with tuples T_2 from the link producer until a matching one is found, and returns the outer tuple as soon as it finds a match. The Kappa-Join does not always enumerate all tuples from the dependent link producer, while at the same time only building the hash table once.

Compared to the algebra plan from Fig. 2, the plan in Fig. 3 using the Kappa-Join has three main advantages: (1) it avoids to enumerate all tuples from the link producer because it immediately returns a result if one match is found (see Line 9). (2) It does not produce duplicates of tuples from the outer producer because the result contains at most one tuple from $\mathcal{T}[//\mathrm{student}]$, and (3) consequently saves the cost of a final duplicate elimination. These effects combine to yield the speedup that can be achieved.

4 Kappa-Join for Disjunctive Correlation

In the previous section we demonstrated how complex correlation predicates that occur in a conjunction can be evaluated efficiently. However, as shown in our Example Q2, correlation predicates can also occur disjunctively. Several optimization techniques for queries containing noncorrelated disjunctive predicates have been proposed [6, 7, 16]. One of them is the Bypass technique [7] that is used to avoid unnecessary evaluation. However, to the best of our knowledge, nobody has shown how to decorrelate disjunctively occurring correlation predicates. In this section, we show how this can be achieved.

4.1 Problem

Consider the canonical algebra plan for Query Q2 (see Fig. 5). This algebra expression is similar to the one presented in Fig. 1 for Q1, except for the *or* function call in the subscript of the Selection. Disjunctively occurring literals are translated using an *or* function call. It evaluates to true if either of its producer expressions does.

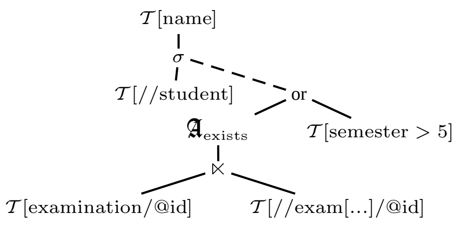

$\mathcal{T}[\mathrm{name}]$
σ
$\mathcal{T}[//\mathrm{student}]$
or
$\mathfrak{A}_{\mathrm{exists}}$
$\mathcal{T}[\mathrm{semester} > 5]$
$\ltimes$
$\mathcal{T}[\mathrm{examination}/@\mathrm{id}]$
$\mathcal{T}[//\mathrm{exam}[\ldots]/@\mathrm{id}]$

Fig. 5. Translation sketch for Q2

The pattern used for the correlation predicate does not match the definition of the Kappa-Join because of the extra literal. Hence, we cannot proceed as for Query Q1. The only technique currently available to improve the canonical plan is the so-called shortcut evaluation of the disjunction, which means that we can avoid evaluation of the expensive correlation predicate for those students where the cheaper literal $\mathrm{semester} > 5$ is true. Below, we recall the Bypass technique which does exactly that.

4.2 Bypass Technique

The Bypass technique was used to prevent the unnecessary evaluation of predicates that occur disjunctively [7]. For this, the Bypass technique adds a new class of operators to the conventional algebra. In contrast to regular operators Bypass operators have *two* output sequences. The first sequence contains the tuples that qualify for the operator's predicate. The second sequence consists of those tuples that do not qualify the operator's predicate. The two disjoint sequences are called *true-* and *false-sequence*. The existing Bypass technique provides a Bypass Selection, a Bypass Join and a Bypass Semi-Join [7]. For the purpose of this paper, we only need the Bypass Selection.

Consider as a first example the algebra representation of Q2 extended by a Bypass Selection operator ($\sigma^{\pm}$) for evaluating the cheaper predicate $\mathrm{semester} > 5$. The resulting plan is shown in Fig. 6. Here and in the following, the false-sequence is indicated by dotted lines. The evaluation according to this plan starts with computing all result tuples for the outer expression ($//\mathrm{student}$).

The Bypass Selection divides these tuples into two disjoint sequences. The true-sequence contains the students that fulfill the predicate $\mathrm{semester} > 5$. Accordingly, the false-sequence contains the tuples that fail this predicate. The tuples of both sequences form two separate paths which are merged by $\dot{\cup}$. The tuples from the false-sequence must pass the second Selection operator computing the complex correlation predicate. This operator is responsible for filtering out those tuples that do not qualify for any of the two predicates. The two sequences are disjoint. Hence, no duplicate elimination is required by $\dot{\cup}$. However, as the XPath semantics requires its result to be in document order, a merge as in merge-sort may be required. This can be done, for example, by numbering the tuples before use or use node ids if they represent order. The final processing of $\mathcal{T}[\mathrm{name}]$ completes the result.

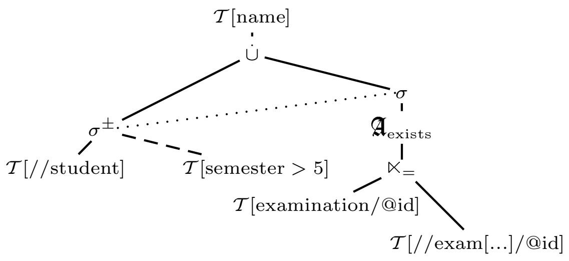

$\mathcal{T}[\mathrm{name}]$
$\dot{\cup}$
σ
$\sigma^{\pm}$
$\mathfrak{A}_{\mathrm{exists}}$
$\mathcal{T}[//\mathrm{student}]$
$\mathcal{T}[\mathrm{semester} > 5]$
$\ltimes_{=}$
$\mathcal{T}[\mathrm{examination}/@\mathrm{id}]$
$\mathcal{T}[//\mathrm{exam}[\ldots]/@\mathrm{id}]$

Fig. 6. Q2 with Bypass Selection

Looking at Fig. 6, we are in for a surprise: The Bypass Selection we introduced to allow shortcut evaluation of the disjunction made the Kappa-Join pattern reappear! We discuss in the following subsection how to leverage this for decorrelation of disjunctive queries with a single correlation predicate.

4.3 Kappa-Join for a Single Disjunctive Correlation Predicate

Query Q2 contains a single correlation predicate within a disjunction. Bypass plans have the advantage that the expression in the false-sequence can be optimized separately. In general, whenever there is only a single correlation predicate per disjunction we can apply decorrelation. As seen in Fig. 6, we can again recognize the pattern that allows us to integrate the Kappa-Join for the conjunctive case. In the false-sequence of Fig. 6, we can use the Kappa-Join, yielding the expression shown in Fig. 7.

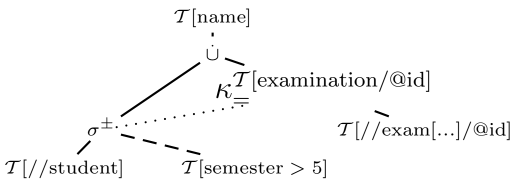

$\mathcal{T}[\mathrm{name}]$
$\dot{\cup}$
$\mathcal{T}[\mathrm{examination}/@\mathrm{id}]$
$\mathcal{K}_{=}$
$\sigma^{\pm}$
$\mathcal{T}[//\mathrm{exam}[\ldots]/@\mathrm{id}]$
$\mathcal{T}[//\mathrm{student}]$
$\mathcal{T}[\mathrm{semester} > 5]$

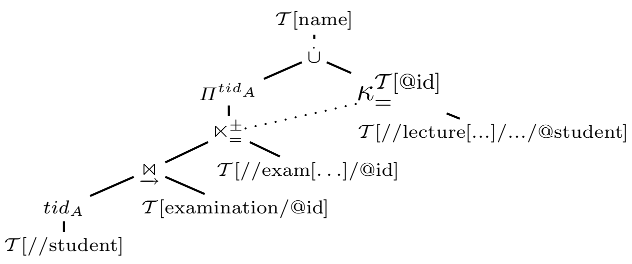
Fig. 7. Q2 with Bypass Selection and Kappa-Join

In this case, the plan takes advantage of both: (1) shortcut evaluation of the literals connected by disjunction, and (2) decorrelation of correlation predicates allowing efficient execution if the cheaper predicate in the disjunction fails.

4.4 Kappa-Join for Multiple Disjunctive Correlation Predicates

We have seen that the Bypass technique facilitates decorrelation if there is only one correlation predicate in the disjunction. Unfortunately, if there is more than one, we are again at a loss. Consider as an example the following Query Q3. In addition to the good students, we also want to query the database for students that have already been a teaching assistant for a given lecture.

//student[examination/@id= //exam[grade $<$ 'B']/@id or
@id = //lecture[title='NCT']/helpers/helper/@student]/name **Q3**

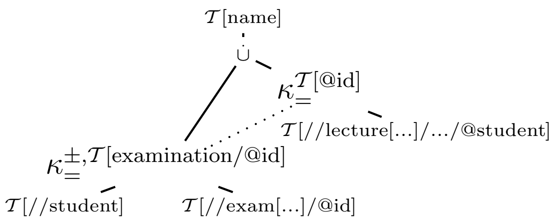


$\mathcal{T}$[name]
$\dot\cup$
Π^{tid_A}
$\mathcal{K}_{=}^{\mathcal{T}[@id]}$
$\ltimes^{\pm}_{=}$
$\mathcal{T}$[//lecture[...]/.../@student]
$\overrightarrow{\bowtie}$
$\mathcal{T}$[//exam[. . .]/@id]
tid_A
$\mathcal{T}$[examination/@id]
$\mathcal{T}$[//student]

Fig. 8. Incorrect decorrelated bypass plan for Q3

We would like to decorrelate both correlation predicates. At first glance, it is tempting to apply the decorrelation strategy that was discussed in Sec. 3.2. Fig. 8 shows an algebra expression for Q3 applying this technique, but using a Bypass Semi-Join instead of a regular Semi-Join. However, this approach is not feasible. The first D-Join on the leftmost branch of the plan eliminates those items produced by $//\mathrm{student}$ for which the dependent expression $\mathrm{exmination}/@\mathrm{id}$ produces an empty result. If we have a conjunctive query, this is no problem.

However, the $//\mathrm{student}$ items failing the first disjunct could still qualify for the second disjunct, and dropping them as in Fig. 8 produces an incorrect result. Note that the Bypass Semi-Join does not help: it comes too late. Problems of this kind are often solved by using an outer join, or in this case outer D-Join. However, this would still require duplicate elimination on tid_A as shown in the true-sequence.



$\mathcal{T}$[name]
$\dot\cup$
$\mathcal{K}_{=}^{\mathcal{T}[@id]}$
$\mathcal{T}$[//lecture[...]/.../@student]
$\mathcal{K}_{=}^{\pm,\mathcal{T}[examination/@id]}$
$\mathcal{T}$[//student]
$\mathcal{T}$[//exam[...]/@id]

Fig. 9. Bypass plan sketches for Q2 with Kappa-Join

It turns out that we can do much better by applying the *Bypass Kappa-Join*. As every bypass operator, the Bypass Kappa-Join has two result sequences. The true-sequence is the same as for the regular Kappa-Join. The tuples in the false-sequence are the ones from the outer producer for which there was no match in the inner producer or for which the link producer returned an empty result.

In the false-stream, we now have our familiar pattern and can employ the decorrelation strategy as if the correlation predicate was a single correlation predicate. Fig. 9 shows the result. This plan finally has everything we want: (1) the evaluation of both correlation predicates can be done in a decorrelated fashion, (2) the Kappa-Join avoids unneeded duplicate generation and elimination for both correlation predicates, and (3) we have shortcut evaluation and only evaluate the second correlation predicate if the first fails.

5 Evaluation

To show the effectiveness of our approach, we ran experiments with different XPath evaluation engines against our canonical and optimized approaches. Additionally, we performed measurements that compare the existing decorrelation strategy against the new Kappa-Join operator. We chose the freely available engines

- Xalan C++ 1.8.0 using Xerces C++ version 2.6.0,
- Saxon for Java 8.7.1,
- Berkeley DB XML 2.0.9 (DBXML) using libpathan 1.99 as XPath engine,
- MonetDB 4.8.0 using MonetDB-XQuery-0.8.0,
- the evaluator provided by the XMLTaskForce [19] (XTF), and
- Natix [9] for the execution of the canonical, decorrelated (ICDE06 [5]), and Kappa-Join plans.

5.1 Environment

The environment we used to perform the experiments consisted of a PC with an Intel Pentium 4 CPU at 2.40GHz and 1 GB of RAM, running Linux 2.6.11-smp. The Natix C++ library was compiled with gcc 3.3.5 with optimization level 2.

For Xalan, Saxon, and XTF we measured the net time to *execute* the query. The time needed to parse the document and generate the main memory representation is subtracted from the elapsed evaluation time. For the evaluation of MonetDB, Berkeley DB XML and Natix, we imported the documents into the database. The time needed for this is not included in the execution times. The queries were executed several times with an empty buffer pool and without any indexes.

Documents We generated two different sets of documents. The first set is used for the example queries Q1-Q3 used throughout this paper. These documents were generated using the ToXgene data generator [1][4]. The smallest document contains 50 employees, 100 students, 10 lectures and 30 exams. With each document we quadrupled these

[4] The DTD as well as the generator template file are listed in the appendix of a technical report.

numbers. That is, the biggest document contains 51200 employees, 102400 students, 10240 lectures and 30720 exams. This led to moderate document sizes between 59kB and 43MB.

The second set is used for the comparison of the existing decorrelation strategy and the new Kappa-Join operator. We generated seven document structured according th the following template:

```
<?xml version='1.0'?>
<gen>
<e1 id='0'> <e2 id='0'/> ... i-e2 nodes <e2 id='i'/> </e1>
                         ...
<e1 id='0'> <e2 id='0'/> ... i-e2 nodes <e2 id='i'/> </e1>
<e3 id='RandomNumber'/>
</gen>
```

Each of the documents contains 1000 `e1` nodes and 1000 `e3` nodes. For each document we varied the number of `e2` nodes (under an `e1` node) between 10 and 500 nodes. This gave us documents between 252kB and 13M.

Queries We executed performance measurements for all example queries (Q1, Q2, and Q3) presented throughout this paper. For Natix, we generated several different query evaluation plans for each of the queries. For each of the queries we generated the canonical plan as specified in [4]. For example, Fig. 1 shows the plans for Q1. Further, we generated plans incorporating our optimization strategies. Fig. 10 gives a mapping from names for optimized query evaluation plans to figures that illustrate the techniques used.

Query	Name	Figure
Q1	decorr	Fig. 2
	kappa	Fig. 3
Q2	bypass	Fig. 6
	kappa	Fig. 7
Q3	bypasskappa	Fig. 9

Fig. 10. Query Evaluation Plans

Additionally, we executed performance measurements that compare the existing decorrelation strategy with our Kappa-Join operator. Therefore, we executed the following query on the synthetic data set:

/gen/e1[e2/@id = /gen/e3/@id] **Q4**

5.2 Results and Interpretation

Fig. 11 contains the results of our performance measurements (elapsed time in seconds). The best execution time(s) for each column in all tables are printed in bold face. Those that did not finish within 6 hours are marked by DNF (did not finish). For MonetDB the evaluation of some queries ran out of memory on bigger documents. These cases are denoted by OOM.

Subfigures 11(a), 11(b), and 11(c) show the execution times for Q1, Q2, and Q3, respectively. For all queries on all documents, our decorrelated approach performs and

Evaluator	Documents 1	2	3	4	5	6
Xalan	0.30	0.38	6.17	95.6	1552	DNF
DBXML	0.07	0.66	11.6	336	DNF	DNF
MonetDB	0.31	0.38	2.05	36.1	OOM	OOM
Saxon	0.21	0.28	0.53	1.49	11.14	141
XTF	0.40	4.72	82.8	DNF	DNF	DNF
Natix						
●canonical	0.25	2.62	38.2	583	9637	DNF
●decorr	**0.02**	**0.03**	**0.06**	**0.19**	**0.75**	2.99
●kappa	**0.02**	**0.03**	**0.06**	**0.19**	**0.75**	**2.88**

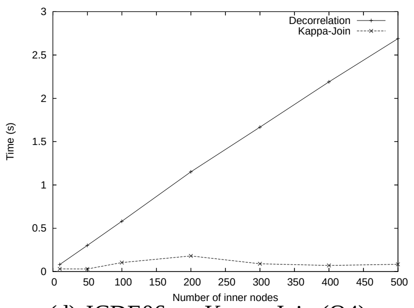
(a) Query Q1

Evaluator	Documents 1	2	3	4	5	6
Xalan	**0.02**	0.23	3.63	54.7	893	12453
DBXML	0.06	0.39	6.87	207	DNF	DNF
MonetDB	0.25	0.36	2.02	36.2	OOM	OOM
Saxon	0.22	0.30	0.62	1.44	7.82	85.4
XTF	0.76	8.60	9180	DNF	DNF	DNF
Natix						
●canonical	0.16	1.64	20.9	333	5598	DNF
●bypass	0.16	1.59	20.7	323	5436	DNF
●kappa	0.03	**0.05**	**0.16**	**0.60**	**2.51**	**9.91**

(b) Query Q2

Evaluator	Documents 1	2	3	4	5	6
Xalan	0.06	0.75	12.6	199	3201	DNF
DBXML	0.30	1.61	30.2	4057	DNF	DNF
MonetDB	0.31	0.50	3.29	62.9	OOM	OOM
Saxon	0.20	0.28	0.54	1.48	10.9	138
XTF	0.48	5.14	94.8	DNF	DNF	DNF
Natix						
●canonical	0.37	3.49	DNF	DNF	DNF	DNF
●bypasscanonical	0.37	3.43	48.1	749	12492	DNF
●bypasskappa	**0.02**	**0.04**	**0.10**	**0.35**	**1.44**	**5.91**

(c) Query Q3



(d) ICDE06 vs. Kappa-Join (Q4)

Fig. 11. Performance measurements

scales best. Especially for the disjunctive queries Q2 and Q3, the performance of all other approaches drops considerably when executed on bigger documents. In contrast, our plans containing the Kappa-Join (Q2) and Bypass Kappa-Join (Q3) almost scale linearly with the size of the document.

For Q1 the execution times of the existing decorrelation approach (called ICDE06 [5]) behave similar to those of the Kappa-Join. This is because all students took very few exams, i.e. only between one and three. For this reason, we compared those two strategies on the synthetic data set. Subfigure 11(d) contains a comparison between the two strategies. The execution times of the existing decorrelation strategy grow linearly with the number of `e2` nodes per `e1` node. This is because it has to enumerate all `e2` nodes and finally perform a duplicate elimination on the appropriate `e1` nodes. The execution times of the Kappa-Join operator almost scale linearly with the document size. The Kappa-Join does not need to enumerate all `e2` nodes and saves the cost of a final duplicate elimination.

6 Related Work

Work on XPath evaluation falls into three general categories. In the first category, we have main memory interpreters like Xalan, XSLTProc, and [12]. Clearly, these approaches do not scale well. In the second category, we find work where XML is shredded into relational systems and XPath is evaluated on this shredded representation. In this category we find approaches like Pathfinder [3]. The problem with this approach are the numerous joins that have to be executed. Finally, the third category uses a native (tree) algebraic approach. Here, we find SAL [2], TAX [15], yet another algebra [27], and [4]. None of the approaches in any of three classes performs decorrelation.

There are, however, special techniques such as avoiding unnecessary ordering operations [14], schema-based optimization [20], and replacing reverse axes [24]. These techniques are orthogonal to our technique.

In the relational and object-oriented context decorrelation has been studied extensively [8, 10, 11, 17, 18, 28]. Similar techniques have been proposed for the evaluation of XQuery and XPath [5, 21]. Gottlob et.al [12] also proposed an approach that avoids multiple evaluations of XPath expressions.

Several optimization techniques for queries containing disjunctive predicates have been proposed [6, 7, 16]. One of them is the Bypass technique [7] which we extend with support for decorrelation. Because bypass operators have two output streams, which are unioned later, the resulting expression forms a directed acyclic graph (DAG). Strategies for implementing Bypass operators and query evaluation engines that support DAG-structured query plans can be found in [7, 23, 26].

7 Conclusion

We demonstrate how to efficiently evaluate XML queries featuring existentially quantified correlation predicates. To this end, we have introduced the novel Kappa-Join operator that naturally fits into algebraic execution plans for quantified correlation predicates. It is simple to implement and yet highly efficient. However, if disjunctions come into play, *all* known decorrelation techniques fail. By injecting the Kappa-Join with the Bypass technique, we are also able to perform decorrelated evaluation if the correlation predicate occurs in a disjunction. All other approaches cannot evaluate such a case efficiently. Our performance measurements show that the Kappa-Join outperforms existing approaches by up to two orders of magnitude.

Acknowledgments We would like to thank Simone Seeger for her comments on the manuscript.

A Logical Operator Definitions

Order plays a crucial role in the semantics of XPath, and NAL is an algebra on sequences. This fact has important implications on properties of the algebra (e.g. commutativity) that are relevant for optimizations (see [21] for an example). Hence, we give the formal definitions of all operators used throughout this paper. We start with some notations and afterwards provide the formal definitions of the operators and bypass operators.

A.1 Notation

A sequence-valued expression e results in several tuples with the same attributes $\mathcal{A}(e)$. The attributes of a single tuple t are also referred to as $\mathcal{A}(t)$. Tuple and function concatenation are denoted by $\circ$. The set of free variables of an expression e is defined as

$\mathcal{F}(e)$. For an expression e possibly containing free variables, and a tuple t, we denote by $e(t)$ the result of evaluating e where bindings of free variables are taken from attribute bindings provided by t. Of course this requires $\mathcal{F}(e) \subseteq \mathcal{A}(t)$. The concatenation of tuples is denoted by $\circ$.

For sequences e we use $\alpha(e)$ to denote the first element of a sequence. The function τ retrieves the tail of a sequence and $\oplus$ concatenates two sequences. We denote the empty sequence by ϵ.

A.2 Algebra Definitions

In the following, we provide the formal definitions of the operators used throughout this paper.

The Selection selects qualifying tuples according to predicate p:

$$\sigma_p(e) := \begin{cases} \alpha(e) \oplus \sigma_p(\tau(e)) & \text{if } p(\alpha(e)) \\ \sigma_p(\tau(e)) & \text{else.} \end{cases}$$

Aggregates its input sequence e into a singleton sequence with a single attribute a by applying the aggregation function f:

$$\mathfrak{A}_{a;f}(e) := \{[a : f(e)]\}$$

Within the Semi-Join the predicate p checks for tuple existence in e_2 to decide on including tuple in e_1:

$$e_1 \ltimes_p e_2 := \begin{cases} \alpha(e_1) \oplus (\tau(e_1) \ltimes_p e_2) & \text{if } \exists x \in e_2 \;\; p(\alpha(e_1) \circ x) \\ \tau(e_1) \ltimes_p e_2 & \text{else} \end{cases}$$

The D-Join joins each tuple t_i in e_1 to all tuples in e_2, which depend on t_i:

$$e_1 \underset{\rightarrow}{\bowtie} e_2 := \alpha(e_1) \overline{\times} e_2(\alpha(e_1)) \oplus \tau(e_1) \underset{\rightarrow}{\bowtie} e_2.$$

where

$$e_1 \overline{\times} e_2 := \begin{cases} \epsilon & \text{if } e_2 = \epsilon \\ (e_1 \circ \alpha(e_2)) \oplus (e_1 \overline{\times} \tau(e_2)) & \text{else.} \end{cases}$$

with $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) \neq \emptyset$.

The tid operators densely numbers the tuples, and stores this number in an attribute A. It is defined as follows:

$$tid_A(e) := \alpha(e) \circ [A : pos] \oplus tid_A(\tau(e)).$$

The Tid-Duplicate Elimination is an order preserving duplicate elimination. It keeps the first tuple for a given A value and throws away the remaining tuples with the same value for A.

$$\Pi_A^{tid_B}(e) := \begin{cases} \alpha(e)|_A \oplus \Pi_A^{tid_B}(\tau(e)) & \text{if } \alpha(e).B \notin \Pi_B(\tau(e)) \\ \Pi_A^{tid_B}(\tau(e)) & \text{else} \end{cases}$$

A.3 Bypass Algebra Definitions

In order to ease the formal description for the bypass operators, their definitions come in two halves. The bypass selection, denoted by $\sigma^{\pm}$, is divided into the part yielding the true-sequence σ^{+} and the half yielding the false-sequence σ^{-}.

$$\sigma_p^{+}(e) := \alpha(e) \oplus \sigma_p(\tau(e)) \text{ if } p(\alpha(e))$$
$$\sigma_p^{-}(e) := \alpha(e) \oplus \sigma_p(\tau(e)) \text{ if not } p(\alpha(e))$$

The Bypass Kappa-Join operator ($e_1 \kappa_p^{\pm;e_2} e_3$) is defined in terms of the Bypass Selection and also comes in two halves:

$$e_1 \kappa_p^{+;e_2} e_3 := \sigma^{+}_{\mathfrak{A}_{x;exists}(e_2 \ltimes_p e_3)}(e_1)$$
$$e_1 \kappa_p^{-;e_2} e_3 := \sigma^{-}_{\mathfrak{A}_{x;exists}(e_2 \ltimes_p e_3)}(e_1)$$

B Translation of XPath to NAL

A location step s_i may contain an arbitrary number h of predicates p_k and has the general form $a_i :: t_i[p_1]\ldots[p_h]$. The pattern for translating [4] a location step $a_i :: t_i[p_1]\ldots[p_h]$ with predicates is

$$\Phi[p_h] \circ \cdots \circ \Phi[p_1] \circ \mathcal{T}[a_i :: t_i]$$

where Φ is an auxiliary translation function for predicates, returning a filtering functor which operates on algebraic expressions.

Within each predicate our normalization already collected those literals that occur conjunctively. Translating a predicate that contains all conjunctively occuring literals $p_k = l_{k1} \wedge \cdots \wedge l_{km_k}$ that do not include positional clauses simply results in a translation into Selection operators:

$$\Phi[l_{k1} \wedge \cdots \wedge l_{km_k}] := \sigma_{\mathcal{T}[l_{km_k}]} \circ \cdots \circ \sigma_{\mathcal{T}[l_{k1}]}$$

After the semantic analysis all clauses are broken down into function calls: $l_{kj} = f_1 \circ \cdots \circ f_r$. For example, `or`, `not`, and comparisons are all evaluated by function calls. All disjunctively occuring literals are also translated using Selection operators. However, they have the `or` function calls as subscript. It is translated straightforward

$$\mathcal{T}[\mathrm{or}(e_1, \ldots, e_n)] := \mathrm{or}(\mathcal{T}[e_1], \ldots, \mathcal{T}[e_n])$$

For the special case of comparisons between two node-sets, as in the particular case of Q1 and Q2, the XPath semantics specifies existential semantics. In the (in)equality case, this fact can be leveraged by using a Semi-Join. More formally, the translation function [4] for a comparison operation $\in \{=, \neq\}$ between two node-sets into NAL is defined as

$$\mathcal{T}[e_1 \theta e_2] := \mathfrak{A}_{x;exists}(\mathcal{T}[e_1] \ltimes_{cn\theta cn'} \Pi^{D}_{cn':cn}(\mathcal{T}[e_2]))$$

References

1. D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: a template-based data generator for XML. In *Proceedings of the ACM Sigmod, Madison, USA*, 2002.
2. C. Beeri and Y. Tzaban. SAL: An algebra for semistructured data and XML. In *WebDB (Informal Proceedings)*, pages 37–42, 1999.
3. P. A. Boncz, T. Grust, S. Manegold, J. Rittinger, and J. Teubner. Pathfinder: Relational xquery over multi-gigabyte XML inputs in interactive time. Technical Report INS-E0503, CWI, March 2005. MonetDB 4.8.0, Pathfinder 0.8.0.
4. M. Brantner, S. Helmer, C-C. Kanne, and G. Moerkotte. Full-fledged Algebraic XPath Processing in Natix. In *Proceedings of the ICDE Conference, Tokyo, Japan*, pages 705–716, 2005.
5. M. Brantner, C-C. Kanne, S. Helmer, and G. Moerkotte. Algebraic optimization of nested xpath expressions. In *Proceedings of the ICDE Conference, Atlanta*, page 128, 2006.
6. F. Bry. Towards an efficient evaluation of general queries: quantifier and disjunction processing revisited. In *Proceedings of ACM SIGMOD Conference, Oregon, USA*, pages 193–204, 1989.
7. J. Claußen, A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimization and evaluation of disjunctive queries. *IEEE Trans. Knowl. Data Eng.*, 12(2):238–260, 2000.
8. U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the VLDB Conference, Brighton, England*, pages 197–208, 1987.
9. T. Fiebig, S. Helmer, C-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *VLDB Journal*, 11(4):292–314, 2002.
10. C. Galindo-Legaria and M. Joshi. Orthogonal optimization of subqueries and aggregation. In *Proceedings of ACM SIGMOD Conference, Santa Barbara, USA*, pages 571–581, 2001.
11. R. A. Ganski and H. K. T. Wong. Optimization of nested sql queries revisited. In *Proceedings of the ACM SIGMOD, San Francisco, California*, pages 23–33. ACM Press, 1987.
12. G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: Improving time and space efficiency. In *Proceedings of the ICDE Conference, Bangalore, India*, pages 379–390, 2003.
13. Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.
14. J. Hidders and P. Michiels. Avoiding unnecessary ordering operations in xpath. In *Proceedings of the DBPL Conference, Potsdam, Germany*, pages 54–70, 2003.
15. H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. Tax: A tree algebra for xml. In *Prcoceedings of the DBPL Conference, Frascati, Italy*, pages 149–164, 2001.
16. M. Jarke and J. Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.
17. W. Kiessling. SQL-like and Quel-like correlation queries with aggregates revisited. ERL/UCB Memo 84/75, University of Berkeley, 1984.
18. W. Kim. On optimizing an SQL-like nested query. *j-TODS*, 7(3):443–469, September 1982.
19. C. Koch. XMLTaskForce XPath evaluator, 2004. Released 2004-09-30.
20. A. Kwong and M. Gertz. Schema-based Optimization of XPath Expressions. Technical report, University of California Davis, 2002.
21. N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. In *Proceedings of the ICDE Conference, Boston, MA, USA*, pages 239–250, 2004.
22. M.Brantner, S.Helmer, C-C. Kanne, and G. Moerkotte. Kappa-join: Efficient execution of existential quantification in xml query languages. Technical report, University of Mannheim, 2006.

23. T. Neumann. *Efficient Generation and Execution of DAG-Structured Query Graphs*. PhD thesis, University of Mannheim, 2005.
24. D. Olteanu, H. Meuss, T. Furche, and F. Bry. Xpath: Looking forward. In *XML-Based Data Management and Multimedia Engineering Workshops XMLDM, MDDE, and YRWS, Prague, Czech Republic*, pages 109–127, 2002.
25. C. Re, J. Siméon, and M. F. Fernández. A complete and efficient algebraic compiler for xquery. In *Proceedings of the ICDE Conference, Atlanta, USA*, page 14, 2006.
26. P. Roy. Optimization of DAG-structured query evaluation plans. Master's thesis, Indian Institute of Technology, Bombay, 1998.
27. C. Sartiani and A. Albano. Yet another query algebra for xml data. In *Proceedings of the IDEAS Conference, Edmonton, Canada*, pages 106–115, 2002.
28. P. Seshadri, H. Pirahesh, and T. Y. Cliff Leung. Complex query decorrelation. In *Proceedings of the ICDE Conference, New Orleans, USA*, pages 450–458, 1996.