Advanced Cardinality Estimation in the XML Query Graph Model

Andreas M. Weiner
Databases and Information Systems Group
Department of Computer Science
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
weiner@cs.uni-kl.de

Abstract: Reliable cardinality estimation is one of the key prerequisite for effective cost-based query optimization in database systems. The XML Query Graph Model (XQGM) is a tuple-based XQuery algebra that can be used to represent XQuery expressions in native XML database management systems. This paper enhances previous works on reliable cardinality estimation for XQuery and introduces several inference rules that deal with the unique features of XQGM, such as native support for Structural Joins, nesting, and multi-way merging. These rules allow to estimate the cardinalities of XQGM operators taken at runtime. Using this approach, we can support classical join reordering with appropriate statistical information, perform cost-based query unnesting, and help to find the best evaluation strategy for value-based joins. The effectiveness of our approach for query optimization is evaluated using the cost-based query optimizer of XTC, which is our prototype of a native XML database management system.

1 Introduction

Native XML database management systems (XDBMSs) can only become a respected competitor for relational-based XQuery evaluation engines, if they can make the most out of the sophisticated join operators (Structural Joins and Holistic Twig Joins) and indexes (element, content, path, and hybrid indexes) that have been proposed in recent years. A cost-based query optimizer is one of the most important parts of modern database systems that generates alternative query execution plans (QEPs), which can be judged based on their expected execution costs, whereas the challenging task for the optimizer is finding the cheapest plan.

Providing reliable cardinality estimates is key for efficient cost assignment. XQuery is the predominant query language in native XDBMSs. Internally, query languages are represented as logical algebra expressions. The *XML Query Graph Model* (XQGM) [Mat09] is a logical XQuery algebra serving as extension of the seminal Query Graph Model [PHH92] and introduces an extended XQuery data model, supports query de-correlation, and provides seamless support for Structural Joins (SJs) [AKJP+02] and Holistic Twig Joins (HTJs) [BKS02].

Most surprisingly, almost all cardinality estimation frameworks for native XDBMSs deal

only with the estimation of XPath path cardinalities. Even though XPath covers an important fragment of XQuery, restricting cardinality estimation to it is insufficient. To the best of our knowledge, the work of Teubner et al. [TGMS08] is the only one that tackles this important problem in the context of relational XQuery engines. Even though many parts of their approach can be transferred to native XDBMSs, further adjustments are necessary: Amongst others, XQGM natively supports full, semi, and outer SJs as logical algebra building blocks.

1.1 Related Work

Cost-based query optimization in semi-structured database systems emerged in the context of the Lore project [MAG+97]. In Lore, DataGuides [GW97] provide a simple means for managing the cardinalities of unique paths in XML documents. XTC's statistics manager reuses this principle to provide the basic statistical information needed to bootstrap our cardinality inference rules. If there are non-unique paths in an XML document, the DataGuide is insufficient to provide the necessary information. In the course of time, many researchers proposed concepts for estimating the path cardinalities in such situations, e. g., [AAN01, FHR+02, ZÖAI06, BEH+06, FM07, Agu10]. These approaches are mostly focusing on estimation accuracy and on minimal space consumption of their data structures. Even though path expressions are important building blocks of XQuery, these approaches do not help to optimize more complex queries (see Section 2.2).

The early work of Sartiani [Sar03] claims to discuss cardinality estimation of FLWR expressions, but focuses mostly on `for` expressions.

Having a look at native XML database management systems that provide cost-based query optimizers, e. g., Natix [FHK+02] or Timber [JAKC+02], shows that their cardinality estimation capabilities are restricted to cardinality estimation of simple path expressions. To the best of our knowledge, MonetDB/XQuery, and its respective XQuery compiler *Pathfinder*, is the only (non-native) XML database management system that supports XQuery cardinality estimation [TGMS08]. They use the general approach of abstract domain identifiers to estimate the value space that is taken by tuple items at runtime. As their cardinality inference rules are strongly tied to their logical algebra, it cannot be directly used in the context of XQGM. In contrast, our work reuses their concept of abstract domain identifiers, but introduces a novel set of inference rules that allow to gain reliable cardinality estimates for XQGM instances.

1.2 Contribution

The contribution of this paper can be summarized as follows:

- we will derive a set of inference rules that allow to perform XQuery cardinality estimation on XQGM instances.

- we will discuss the optimization of value-based joins, which especially require reliable estimates to choose the right join operator and join order to prevent bad plans.
- we describe how reliable cardinality estimates allow for cost-based query unnesting in XTC
- we show that our inferences rules provide reliable cardinality estimates for a wide range of queries and support our cost-based query optimizer in providing linear scalability for the XMark benchmark queries.

2 Preliminaries

Before we detail our inference rules, Section 2.1 briefly introduces the XML Query Graph Model, which is our internal representation for XQuery expressions. Thereafter, Section 2.2 motivates the importance of reliable cardinality estimation for efficient query evaluation in native XDBMSs.

2.1 A Brief Introduction to the XML Query Graph Model

The *XML Query Graph Model* (XQGM) is the logical XQuery algebra of the *XML Transaction Coordinator* (XTC) [HMB+10], which is our prototype of a native XDBMS supporting, amongst others, ACID transactions and cost-based query optimization [WHdS10].

2.1.1 XQGM Data Model

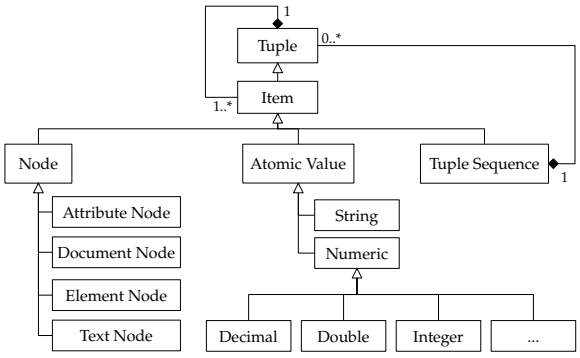


Figure 1: The XQGM data model [Mat09]

The XQGM data model extends the *XQuery Data Model* (XDM) [FMM+07], which supports *atomic values*, *items*, and *sequences* as model primitives. The XQGM Data Model supplements these primitives by *tuples*. More precisely, every XQGM Data Model object is a tuple [Mat09]. Figure 1 depicts the UML diagram of the XQGM Data Model compo-

nents. An item can be (1) a node, (2) an atomic value, or (3) an ordered list of tuples, which we denote as *tuple sequence*. Thus, tuple sequences can contain nested tuples. We refer to a tuple sequence that contains exactly one tuple as a *singleton tuple sequence* and a tuple that is formed by a single item is called a *singleton tuple*[1]. A singleton is always equal to the unique element it contains. Moreover, a tuple sequence is an XQuery sequence, if and only if it contains only singleton tuples or is equal to an empty sequence [Mat09]. During query evaluation, we postulate that only the final result must be an XQuery sequence, whereas intermediate results must not conform to the XDM, but only to the XQGM data model.

2.1.2 XQuery the XQGM Way

Before an XQGM instance is passed on to our cost-based query optimizer [WHdS10], several algebraic rewrites are applied. One of the major challenges for native XML query processing is bridging the gap between two completely different processing strategies: *node-at-a-time* and *set-at-a-time* processing. Node-at-a-time evaluation is inherent to the *XQuery Core Language* and follows a nested-loops-style evaluation approach (nested `for` loops) that is similar to sub-selects in SQL [Mat07]. Even though it is not very efficient in most cases, it can be beneficial in low-selectivity scenarios. On the other hand, set-at-a-time query evaluation is employed by almost all SJ and HTJ algorithms and is in most cases very efficient.

The XML Query Graph Model supports both processing strategies. After an initial translation of an XQuery expression into XQGM, query unnesting tries to iteratively replace path expressions by cascades of (logical) SJs. During this rewriting process, further operators may be introduced, e. g., outer join operators helping to preserve the correct output semantics for positional predicates or for the `return` clause in the presence of empty sequences.

2.1.3 XQGM Example

Figure 2 illustrates an XQGM instance for the following query, which returns the `author` names of `book` nodes whose `price` is larger than 1.99:

```
<result> {
 for $b in doc("sample.xml")//book
 where $b/price > "1.99"
 return
  <author>{ $b/author/text() }</author>
} </result>
```

In XQGM, there exists no direct connection between operators. Instead, every operator contains so-called *tuple variables* that actually receive the tuple sequences emitted by child operators. We distinguish between three different types of quantifiers: *for* (F), *let*

[1] For the rest of this work, whenever the context is unambiguous, we just refer to them as singleton.

(L), and *exists* (E). The first and the second quantifier have a similar semantics as the corresponding XQuery constructs: A *for*-quantified tuple variable iterates over the tuple sequence it receives from its input. In contrast, *let*-quantified tuple variables deliver their connected sequences at once. Finally, *exists*-quantified tuple variables are used to check whether their inputs are non-empty tuple sequences.

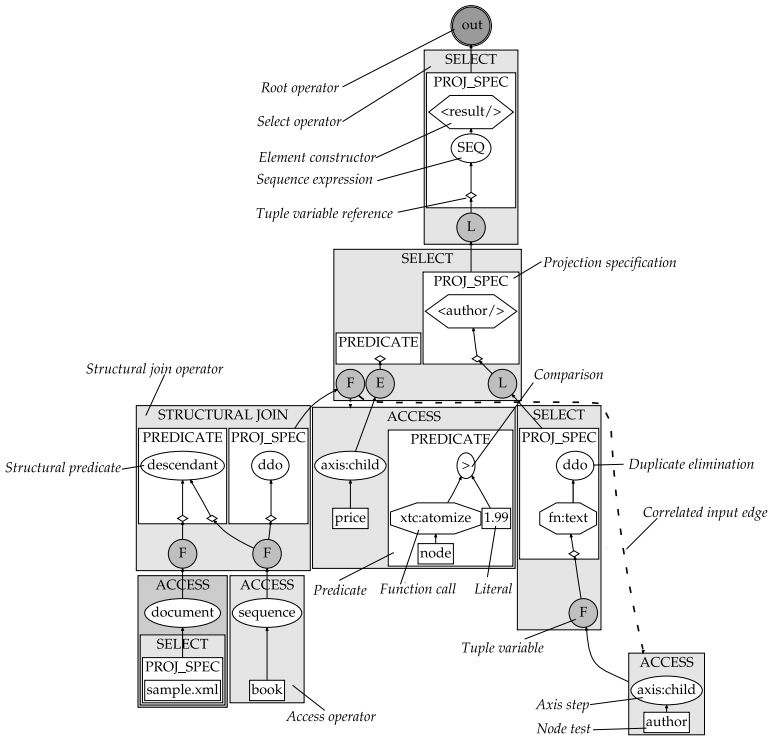


Figure 2: Sample XQGM instance

Query evaluation starts at the left-most subtree (Document Access operator). The SJ operator evaluates the structural predicate (`descendant` axis) between the virtual document root and all `book` elements. Actually, SJ is a semi-join operator, because only the tuple variable that provides `book` elements is linked to the projection specification (PROJ_SPEC). The parent Select operator receives a tuple sequence of all `book` elements and iterates over it: First, it sends the current evaluation context to the Access operator associated with it via the *exists*-quantified tuple variable[2]. The Access operator selects a sequence of all `price` element nodes that are children of the current (context) `book` element and returns only those items that satisfy the predicate. Next, the Select operator passes the current context to the right-most Access operator that returns all `author`

[2]In the graphical XQGM representation, the context passing is illustrated as a dashed line from the "sending" tuple variable to the receiving operator.

nodes that are connected to the current context node via the `child` axis. For each match, the parent Select operator evaluates the `fn:text()` function. Now, the Select operator, which is at the heart of the query graph, can produce an output: It creates a new XML element `<authors>seq</authors>`, if the sequence bound to the *exists*-quantified tuple variable is non-empty, whereas `seq` represents the tuple sequence received by the *let*-quantified tuple variable. Finally, the top-most Select operator wraps its input in an opening `<result>` and a closing `</result>` tag.

2.2 Problem Statement

Today, cardinality estimation frameworks for native XDBMSs are not prepared to handle XQuery expressions sufficiently. Most of them are only focusing on estimating the output sizes of simple XPath expressions. Let us consider a slightly simplified query from the W3C XQuery Uses Cases query set (Use Case "R", query Q13):

```
<result> {
 for $uid in distinct-values(doc("bids.xml")//userid),
 $u in doc("users.xml")//user_tuple[userid = $uid]
 let $b := doc("bids.xml")//bid_tuple[userid = $uid]
 return
  <bidder name="{$u/name}" bidCount="{ count($b)}" />
} </result>
```

The query returns for each user who has placed a bid the corresponding user name and the total number of bids. Figure 3 shows the graphical representation of the respective XQGM instance.

The subtree rooted at operator **1** delivers the result for the expression `distinct-values(doc("bids.xml")//userid)`. The Select operator **2** provides the "heartbeat" for the further workflow. Operator **3** evaluates the first value-based join. Operator **4** triggers the evaluation of the second expression in the `for` clause that is bound to `$u`. The Structural Outer Join (operator **5**) returns `(userid, user_tuple)` tuples and preserves `user_tuple` nodes even if it does not find a matching join partner (necessary for preserving the correct output semantics for empty sequences in the final `return` clause). After evaluating the value-based join, the qualifying `user_tuple` nodes are passed on to operator **6** that provides the results for the `name` attribute in the query result (operator **7**).

In the `let` clause of our sample query, there is a second value-based join (operator **10**). Operator **8** furnishes the evaluation context (dashed line) for the Access operator below operator **9**. Finally, the result is passed from operator **7** to operator **2**, which in turns sends it to operator **11**.

Even for this simple XQuery expression, present cardinality estimation frameworks fail to provide satisfying results. These frameworks are only capable of providing estimates for the outputs of the XPath expressions bound to `$uid` (see Section 1.1). This situation is really sobering, because many optimization decisions, e. g., the selection of appropriate

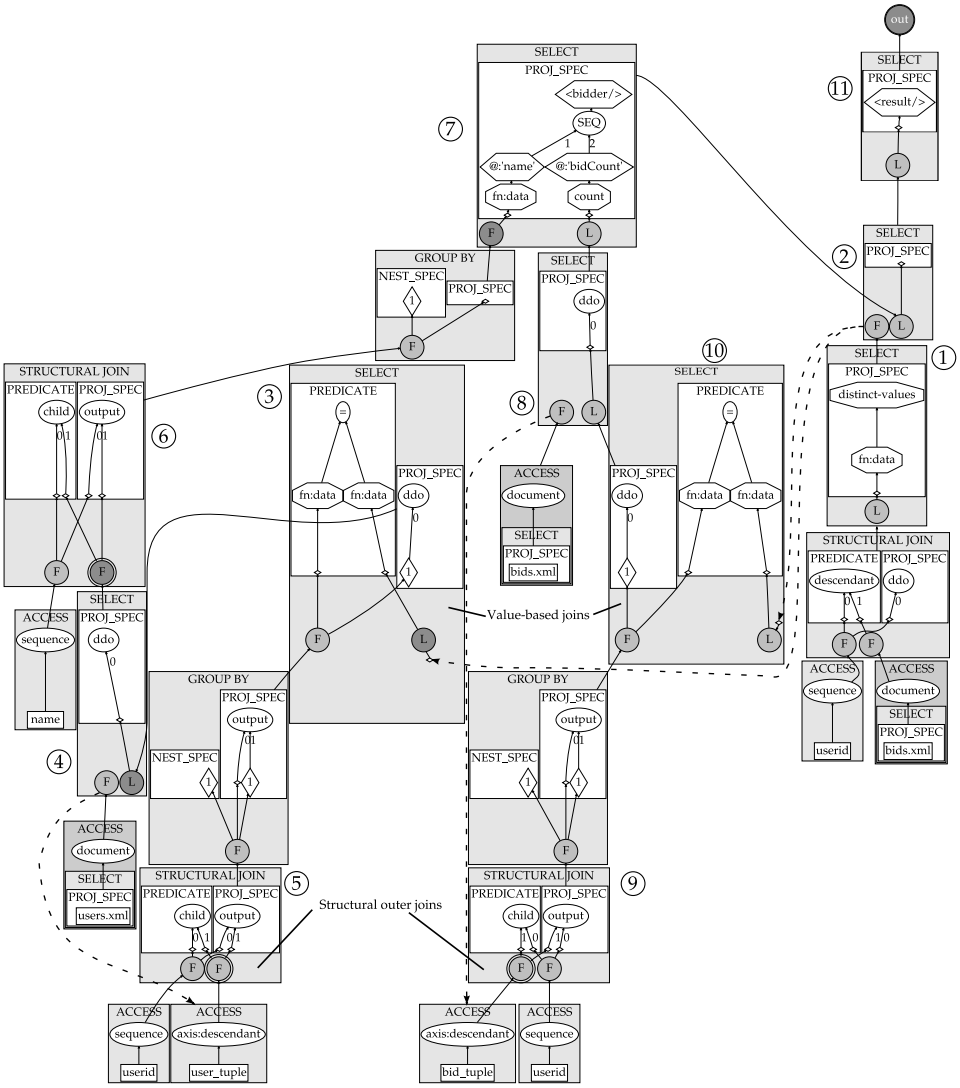


Figure 3: XQGM instance for the sample query

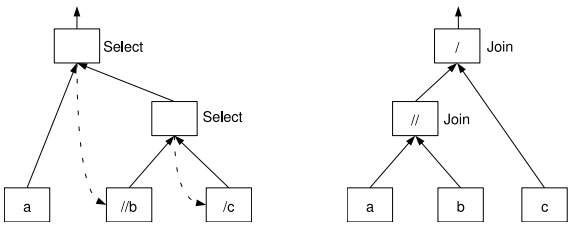
join operators for value-based joins or Structural Join reordering can only be based on vague and coarse-grained heuristics. For example, if a value-based join could be evaluated using a hash join operator, we could only guess which input should be hashed and which one probed. Here, a wrong guess could lead to tremendous performance loss. Moreover, we cannot provide cardinality estimates for the final query result. For example, this might have an impact on the selection of an appropriate materialization strategy: If there are only few final results, late materialization might be preferred over early materialization.

Even though the concept of *abstract domain identifiers* and the corresponding cardinality inference rules introduced by Teubner et al. [TGMS08] provide an elegant means for gaining reliable cardinality estimates, they rely on a completely different algebra and, hence, cannot be directly used out-of-the-box for our purposes. Consequently, a similar set of inference rules must be specified that provide support for XQGM-specific operators such as logical Structural Joins, *n*-way outer joins, and unnest operators to allow for full cardinality estimation on XQGM instances.

At the moment, XTC's query optimizer applies query unnesting [Mat07] as a heuristics. If we could provide reliable cardinality estimates for XQGM, we would be able to leave the decision whether to perform query unnesting or not to the cost-based query optimizer, which would make the query optimization process even more flexible.

3 Application Scenarios for the Inference Rules

In this section, we have a brief look at two applications of our cardinality inferences rules that complement the classical scenario for cardinality estimation, i. e., providing reliable statistical information for join reordering.



(a) Node-at-a-time (b) Set-at-a-time

Figure 4: Evaluation strategies for XML queries

In Section 2.1.2, we mentioned the importance of query unnesting for efficient query evaluation. Figure 4(a) shows how a simple XPath expression `a//b/c` is evaluated using node-at-a-time and set-at-a-time processing, respectively. During node-at-a-time processing, for every `a` node, the evaluation context for the evaluation of `//b` is provided (dashed line). By iterating over all qualified `b` nodes, the evaluation context for `/c` is furnished. Finally, every qualified `c` node is output. On the other hand, set-at-a-time query evaluation works similar to relational merge joins. Figure 4(b) illustrates how the XPath expression

`a//b/c` is evaluated using this approach: First, the location step `//` is evaluated between all `a` and `b` nodes. Afterwards, the result of the first Structural Join operator serves together with all `c` nodes as input for the second one that evaluates the location step `b/c`. In XTC, the transition from node-at-at-a-time to set-at-a-time processing is performed during a pattern-based algebraic rewrite process[3]. So far, query rewrites are applied in an eager way, i. e., as long as query unnesting is possible, it is applied. Now, using our refined cardinality information, the query optimizer can immediately abort query unnesting if the estimated cardinalities indicate that this would be counterproductive.

Efficient evaluation of value-based joins is crucial for the performance of XML database systems. In contrast to structural predicates, which can be decided using appropriate node labeling schemes without any further accesses to the document, value-based predicates need additional accesses to the document to fetch the actual content nodes. Let us have a look at XQGM instance shown in Figure 3. For example, operator **3** evaluates a value-based predicate. At the logical level, for each tuple bound to the *let*-quantified tuple variable, a complete iteration over the sub-expression rooted at the *for*-quantified tuple variable is performed. Choosing the right physical implementation is crucial for the query performance. During optimization, it is especially important to know how many tuples will be delivered via the *for*-quantified and *let*-quantified tuple variables. Hence, the evaluation order proposed by the logical algebra is not always the most efficient one: Let us assume that the optimizer can use a value-based hash join for evaluating the value-based join. If we would know that the *let*-quantified tuple variable would deliver 1,000 tuples, but the *for*-quantified tuple variable only 10, then it would be much cheaper to hash the 10 tuples (only a single evaluation of the sub-expression) and probe the input of the *let*-quantified tuple variable against it. But for a reciprocal input ratio, this decision would result in an extremely slow QEP.

4 Cardinality Inference

In this section, we introduce the various inference rules that are used for cardinality estimation in XTC. Section 4.1 introduces preliminary notions. Thereafter, Section 4.2 discusses the inference rules.

4.1 Nomenclature

Our cardinality estimation approach is based on the idea of *abstract domain identifiers* [TGMS08]. For your convenience, we repeat its definition and adjust it to our needs in the context of XQGM. Abstract domain identifiers, which will be denoted by Greek letters such as $\alpha, \beta, \ldots$, allow to estimate the value space of tuple items that are exposed by XQGM expressions at runtime.

Each XQGM operator consumes or emits tuple sequences. Let $s =< t_1, \ldots, t_n >$ be a tuple sequence where each $t_j = [i_{j1}, \ldots, i_{jm}]$ is a tuple with m items. Obviously, this tuple

[3]The example shown here is very simplistic and abstract. In reality, the patterns are much more complex and may cover numerous operators. However, the rationale behind the two evaluation strategies should be obvious.

sequence has a "fixed schema" with m columns (denoted by $c_1, \ldots, c_m$), i. e., every tuple has m (probably nested) items where each i_{jk} shares a common domain (for an arbitrarily but consistently chosen k). Table 1 illustrates this imaginary schema.

	c_1	$\ldots$	c_m
t_1	i_{11}	$\ldots$	i_{1m}
$\vdots$	$\vdots$	$\ddots$	$\vdots$
t_n	i_{n1}	$\ldots$	i_{nm}

Table 1: Imaginary schema

Let us denote the *active domain* of c_i, i. e., the set of all values taken by tuples $t_1 \ldots, t_n$ in column c_i, by α_i. Moreover, we refer to the *domain size*, i. e., the total number of distinct values in c_i, by $\|\alpha_i\|$. We define $\mathrm{dom}(o) = \{c_1^{\alpha_1}, \ldots, c_m^{\alpha_m}\}$ as the *result domain set* of the tuple sequence produced by XQGM operator o as output[4], whereas $c_i^{\alpha_i}$ is the i-th column of the tuple sequence emitted by o with the corresponding active domain α_i.

According to [TGMS08], for the abstract domain identifiers α and β, we define the reflexive and transitive *inclusion relationship* $\beta \sqsubseteq \alpha$ as follows: $\beta \sqsubseteq \alpha \Longleftrightarrow \forall b \in \beta : b \in \alpha$. Finally, we will denote the assignment of an inferred value by $=^!$ and an inferred inclusion relationship by $\sqsubseteq^!$.

4.2 Inference Rules

After providing the preliminaries, we are now ready to discuss the different inferences rules that are used to estimate the cardinalities of XQGM (sub)-expressions. The definition of the various inference rules is mostly based on the notation introduced by [TGMS08].

4.2.1 Access Operators

Figure 5 shows the inference rules for Access operators. In XQGM, query evaluation starts at a document's root node. Rule CARD-DOC-ACCESS simply assigns the value 1 to the abstract domain identifier α, because there is only a single document root in the document.

Rule CARD-ACCESS is responsible for deriving the cardinality estimates for basic Access operators. An Access operator provides a tuple stream having an element or attribute name e as filter and may evaluate an optional predicate p. We estimate its cardinality by the total number of element or attribute names having the corresponding name. The selectivity of predicate p can be determined by employing histograms or by simply using the famous 10% heuristics of System R [SAC$^+$79].

Finally, CARD-ACCESS-WITH-CONTEXT serves for estimating the cardinality of access

[4]Please note, here we assume that the identifiers $\alpha_1, \ldots, \alpha_m$ have not been used before.

operators whose output depends on an evaluation context. Such operators are employed for XQGM sub-expressions that cannot be unnested according to the unnesting rules defined by Mathis [Mat07]. Here, the abstract domain identifier β of the context-providing operator is used as a starting point for cardinality inference. In expression $\sigma(b\,\theta\,e)$, b is the current context item, θ is the corresponding XPath axis and e is the tuple stream issued by the current access operator.

$$\frac{}{\text{dom}(\text{DocAccess}) = \{c^{\alpha} \wedge \|\alpha\| =^{!} 1\}} \quad \text{(CARD-DOC-ACCESS)}$$

$$\frac{}{\text{dom}(\text{Access}_{e\,p}) = \{c^{\alpha} \wedge \|\alpha\| =^{!} |e| \cdot \sigma(p)\}} \quad \text{(CARD-ACCESS)}$$

$$\frac{b^{\beta} \in \text{dom}(\square) \wedge \square \text{ provides evaluation context}}{\text{dom}(\text{Access}_{e\,\theta}) = \{a^{\alpha} \wedge \|\alpha\| =^{!} |e| \cdot \sigma(b\,\theta\,e)\}} \quad \text{(CARD-ACCESS-WITH-CONTEXT)}$$

Figure 5: Inference rules for access operators

For example, in Figure 3, the left-most Access operator (below sub-expression 1) accesses the sequence of all `userid` nodes and does not evaluate a predicate. The right-most operator simply accesses the document root of document `bids.xml`. Sub-expression 9 in Figure 3 shows an Access operator (left-most access operator) that is a match for rule CARD-ACCESS-WITH-CONTEXT. Here, The Select operator (Operator 8) provides the evaluation context, θ is the XPath `descendant` axis, and e corresponds to the element name `bid_tuple`.

4.2.2 Cardinality Estimation for Structural Joins

In XQGM, Structural Joins are the basic building blocks for the evaluation of XPath path expressions. Whenever possible, we use Structural Semi Joins to reduce intermediate results. Figure 6 illustrates the inference rules for the six different Structural Join types.

For the representation of the various join types, we use the well-known symbols: $\ltimes_p$ (Structural Left Semi Join), $\rtimes_p$ (Structural Right Semi Join), $\bowtie_p$ (Structural Full Join), $⟗_p$ (Structural Left-outer Join), and $⟖_p$ (Structural Right-outer Join). The Structural Join evaluates a structural predicate p described as follows: $a_i\,\theta\,b_j$, where a_i is an item of tuple sequence q_1, b_j is an item of tuple sequence q_2, and θ is an XPath axis, e. g., `descendant`.

Even though the definitions of the inference rules seem to be cumbersome at first sight, their rationale is very simple: For estimating the result size of active domains affected by the structural predicate, we rely on two data structures. In the case of path expressions describing linear and unique paths, we use the *path synopsis*, which is an extension to the seminal DataGuide [GW97], to derive accurate cardinalities for the expression. If the path expression is more complex or involves non-unique paths, we approximate the cardinality using XTC's XPath cardinality estimation framework called *EXsum* [Agu10].

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q_1) \wedge b_j^{\beta_j} \in \mathrm{dom}(q_2)}{|q_1 \ltimes_{a_i \theta b_j} q_2| = \|\alpha_i\| \cdot \sigma\big(a_i[\theta\, b_j]\big) \wedge \mathrm{dom}(q_1 \ltimes_{a_i \theta b_j} q_2) = \Big\{ c_2^{\gamma_2} \big| \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \mathrm{dom}(q_1) \backslash \{a_i^{\alpha_i}\} \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot \big[1 - (1 - 1/10)^{|q_1|/\|\gamma_1\|}\big] \Big\} \cup \Big\{ c^{\gamma} \big| \gamma \sqsubseteq^! \alpha_i \wedge \|\gamma\| =^! |q_1 \ltimes_{a_i \theta b_j} q_2| \Big\}} \quad \text{(CARD-SJ-1)}$$

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q_1) \wedge b_j^{\beta_j} \in \mathrm{dom}(q_2)}{|q_1 \rtimes_{a_i \theta b_j} q_2| = \|\beta_j\| \cdot \sigma(a_i\, \theta\, b_j) \wedge \mathrm{dom}(q_1 \rtimes_{a_i \theta b_j} q_2) = \Big\{ c_2^{\gamma_2} \big| \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \mathrm{dom}(q_2) \backslash \{b_j^{\beta_j}\} \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot \big[1 - (1 - 1/10)^{|q_2|/\|\gamma_1\|}\big] \Big\} \cup \Big\{ c^{\gamma} \big| \gamma \sqsubseteq^! \beta_j \wedge \|\gamma\| =^! |q_1 \rtimes_{a_i \theta b_j} q_2| \Big\}} \quad \text{(CARD-SJ-2)}$$

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q_1) \wedge b_j^{\beta_j} \in \mathrm{dom}(q_2) \wedge a_i \theta b_j \text{ is location step}}{|q_1 \ltimes_{a_i \theta b_j} q_2| = \|\beta_j\| \cdot \sigma(a_i\, \theta\, b_j) \wedge \mathrm{dom}(q_1 \ltimes_{a_i \theta b_j} q_2) = \Big\{ c_2^{\gamma_2} \big| \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \mathrm{dom}(q_1) \cup \mathrm{dom}(q_2) \backslash \{a_i^{\alpha_i}, b_j^{\beta_j}\} \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot \big[1 - (1 - 1/10)^{(|q_1|+|q_2|)/\|\gamma_1\|}\big] \Big\} \cup \Big\{ c^{\gamma} \big| \gamma \sqsubseteq^! \alpha_i \wedge \|\gamma\| =^! \|\alpha_i\| \cdot \sigma\big(a_i[\theta b_j]\big) \Big\} \cup \Big\{ c^{\gamma} \big| \gamma \sqsubseteq^! \beta_j \wedge \|\gamma_j\| =^! \|\beta_j\| \cdot \sigma(a_i \theta b_j) \Big\}} \quad \text{(CARD-SJ-3)}$$

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q_1) \wedge b_j^{\beta_j} \in \mathrm{dom}(q_2) \wedge a_i \theta b_j \text{ is predicate step}}{|q_1 \ltimes_{a_i \theta b_j} q_2| = \|\alpha_i\| \cdot \sigma\big(a_i[\theta b_j]\big) \wedge \mathrm{dom}(q_1 \ltimes_{a_i \theta b_j} q_2) = \Big\{ c_2^{\gamma_2} \big| \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \mathrm{dom}(q_1) \cup \mathrm{dom}(q_2) \backslash \{a_i^{\alpha_i}, b_j^{\beta_j}\} \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot \big[1 - (1 - 1/10)^{(|q_1|+|q_2|)/\|\gamma_1\|}\big] \Big\} \cup \Big\{ c^{\gamma} \big| \gamma \sqsubseteq^! \alpha_i \wedge \|\gamma\| =^! \|\alpha_i\| \cdot \sigma\big(a_i[\theta b_j]\big) \Big\} \cup \Big\{ c^{\gamma} \big| \gamma \sqsubseteq^! \beta_j \wedge \|\gamma_j\| =^! \|\beta_j\| \cdot \sigma(a_i \theta b_j) \Big\}} \quad \text{(CARD-SJ-4)}$$

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q_1) \wedge b_j^{\beta_j} \in \mathrm{dom}(q_2)}{|q_1 \ltimes\!\!\!\subset_{a_i \theta b_j} q_2| = \|\beta_j\| \wedge \mathrm{dom}(q_1 \ltimes\!\!\!\subset_{a_i \theta b_j} q_2) = \Big\{ b_j^{\beta_j} \Big\} \cup \Big\{ c_2^{\gamma_2} \big| \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \mathrm{dom}(q_1) \cup \mathrm{dom}(q_2) \backslash \{a_i^{\alpha_i}, b_j^{\beta_j}\} \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot \big[1 - (1 - 1/10)^{(|q_1|+|q_2|)/\|\gamma_1\|}\big] \Big\} \cup \Big\{ c^{\gamma} \big| \gamma \sqsubseteq^! \alpha_i \wedge \|\gamma\| =^! \|\alpha_i\| \cdot \sigma\big(a_i[\theta b_j]\big) \Big\}} \quad \text{(CARD-SJ-5)}$$

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q_1) \wedge b_j^{\beta_j} \in \mathrm{dom}(q_2)}{|q_1 \rtimes\!\!\!\supset_{a_i \theta b_j} q_2| = \|\alpha_i\| \wedge \mathrm{dom}(q_1 \rtimes\!\!\!\supset_{a_i \theta b_j} q_2) = \Big\{ a_i^{\alpha_i} \Big\} \cup \Big\{ c_2^{\gamma_2} \big| \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \mathrm{dom}(q_1) \cup \mathrm{dom}(q_2) \backslash \{a_i^{\alpha_i}, b_j^{\beta_j}\} \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot \big[1 - (1 - 1/10)^{(|q_1|+|q_2|)/\|\gamma_1\|}\big] \Big\} \cup \Big\{ c^{\gamma} \big| \gamma \sqsubseteq^! \beta_j \wedge \|\gamma\| =^! \|\beta_j\| \cdot \sigma(a_i \theta b_j) \Big\}} \quad \text{(CARD-SJ-6)}$$

Figure 6: Inference rules for Structural Joins

The expression $\sigma\big(a_i[\theta\, b_j]\big)$ returns the selectivity of a_i items connected to b_j items via the θ axis, i. e., the percentage of a_i nodes satisfying the structural predicate. On the other hand, $\sigma(a_i\, \theta\, b_j)$ returns the selectivity of b_j items. For the remaining items, we follow the idea of Teubner et al. [TGMS08] to use a generalization of the classical 10% rule to estimate the new cardinalities of active domains that are not directly affected (i. e., independent) by the structural predicate: $\|\gamma_2\| =^! \|\gamma_1\| \cdot \big[1 - (1 - 1/10)^{|q|/\|\gamma_1\|}\big]$, where $|q|$ is the cardinality of input operator q, γ_1 is the active domain cardinality of a column contained

in the input tuple sequence of q, and $\|\gamma_2\|$ is the inferred cardinality for the corresponding active domain in the output tuple sequence.

Rules CARD-SJ-1 and CARD-SJ-2 depict the inference rules for Structural Left Semi Joins and Right Semi Joins, respectively. Here, the result domain set is equal to the domain set of the left or right input operator, respectively. The cardinalities of the active domains' join items are calculated using the path synopsis or EXsum and all remaining cardinalities of the active domains in the output domain set are approximated using the generalized 10% rule.

For the cardinality estimation of Structural Full Joins, rules CARD-SJ-3 and CARD-SJ-4 show the corresponding definitions, which distinguish between the evaluation of location steps and predicate steps. Both rules estimate the output cardinality of the join operator and the active domains of join items a_i and b_j using the path synopsis. Once again, the generalized 10% rule helps to approximate the active domains for items that are independent of the join predicate.

In Figure 3, several Structural Outer Joins are used (e. g., operators **5**, **6**, and **9**). Inference rules CARD-SJ-5 and CARD-SJ-6 allow for cardinality inference of Structural Left-outer and Structural Right-outer Joins, respectively[5]. The active domain of the join item whose tuple sequence contributes to the output's outer part remains unchanged, and the cardinality of the other join item is adjusted according to EXsum's estimation. For all remaining active domains, the generalized 10% rule is applied.

4.2.3 Inference Rules for Grouping, Unnesting, and Miscellaneous Operators

$$\frac{a_i^{\alpha_i} \in \mathrm{dom}(q)}{|\mathrm{GroupBy}_i(q)| = \|\alpha_i\|} \quad \text{(CARD-GROUP-BY)} \qquad \frac{a_i^{\alpha_i} \in \mathrm{dom}(q)}{|\mathrm{Unnest}_i(q)| = \mathrm{flatCard}(q)} \quad \text{(CARD-UNNEST)}$$

$$\frac{\square \in \{\mathrm{Split}, \mathrm{Project}\} \wedge \mathrm{dom}(q') \subseteq \mathrm{dom}(q)}{\mathrm{dom}(\square(q)) = \{a_i^{\alpha_i} \,|\, a_i^{\alpha_i} \in \mathrm{dom}(q')\}} \quad \text{(CARD-MISC-1)} \qquad \frac{\square \in \{\mathrm{Sort}, \mathrm{DDO}\}}{|\square(q)| = |q| \cdot 2/3} \quad \text{(CARD-MISC-2)}$$

Figure 7: Inference rules for grouping, unnesting, and miscellaneous operators

The GroupBy and the Unnest operator allow for nesting and unnesting of tuple sequences w. r. t. a specific item position, respectively. Rule CARD-GROUP-BY shows the cardinality inference for nesting. The cardinality of the result tuple sequence is primarily determined by the cardinality of the active domain according to which the nesting is performed. The active domain set remains unchanged.

The cardinality inference rule for the Unnest operator (CARD-UNNEST) approximates the cardinality of the result tuple sequence using the function *flatCard*, which calculates the cardinality of the Cartesian product of the values of item i and the values of the remaining active domains.

[5] Please note, we do not use Structural Full-outer Joins in XQGM.

The Split operator sends its input to multiple consumers and the Project operator serves as classical projection operator. Rule CARD-MISC-1 simply derives the result domain set by considering those active domains that are referred to in the projection specification, where q is the input tuple sequence and q' is the output tuple sequence. Again, the cardinality remains unchanged.

Conventionally, every XQGM operator returns tuple sequences sorted in document order and tries to reduce duplicates to a minimum. If duplicates cannot be avoided, e. g., if a full join using the `descendant` axis is performed, additional duplicate elimination might become necessary. Sort and Distinct-Doc-Order (DDO) retain a certain sort order or eliminate duplicates. Inference rule CARD-MISC-2 describes the estimation of the output cardinality. We expect that most tuple sequences are almost duplicate free and sorted, therefore, we assume that two thirds of their input tuples will "survive".

4.2.4 Inference Rules for Merge and Select

Rules CARD-MERGE-1 and CARD-MERGE-2 show the inference rules for the Merge operator. The Merge operator contains only *for*-quantified tuple variables and calculates the Cartesian product on its input tuple streams. Each Merge operator contains a so-called merge specification that describes a complex selection predicate on the Cartesian product. The predicate selects all tuples that have equal values for given positions in the tuple sequence. For the sake of simplicity, we also use the 10% rule to determine the output cardinality.

$$\frac{}{|\text{Merge}(q_1,\ldots,q_n)| = \prod_{i=1}^{n} |q_i| \cdot 1/10 \wedge \text{dom}(\text{Merge}(q_1,\ldots,q_n)) \supseteq \Big\{ c_2^{\gamma_2} \,\big|\, \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \cup_{j=1}^{n} \text{dom}(q_j) \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot [1-(1-1/10)^{\sum_{i=1}^{n} |q_i|/\|\gamma_1\|}] \Big\}} \quad \text{(CARD-MERGE-1)}$$

$$\frac{q_i \text{ delivers outer sequence}}{|\text{Merge}(q_1,\ldots,q_n)| = |q_i| \wedge \text{dom}(\text{Merge}(q_1,\ldots,q_n)) \supseteq \Big\{ c_2^{\gamma_2} \,\big|\, \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \cup_{j=1}^{n} \text{dom}(q_j) \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot [1-(1-1/10)^{|q_i|/\|\gamma_1\|}] \Big\}} \quad \text{(CARD-MERGE-2)}$$

$$\begin{aligned} &|\text{Select}_p(q_1,\ldots,q_n)| = \prod_{q \in \mathcal{Q}_{\text{for}} \cup \mathcal{Q}_{\text{let}}} |q| \cdot 1/10 \wedge \text{dom}(\text{Select}(q_1,\ldots,q_n)) = \mathcal{Q}'_{\text{for}} \cup \mathcal{Q}'_{\text{let}} \cup \mathcal{Q}'_{\text{exists}} \wedge \\ &\mathcal{Q}'_{\text{for}} = \big\{ c_2^{\gamma_2} \big| \gamma_2 \sqsubseteq \gamma_{1j} \wedge c_{1j}^{\gamma_1} \in \text{dom}(q_j) \wedge q_j \in \mathcal{Q}_{\text{for}} \wedge \|\gamma_2\| =^! \|\gamma_{1j}\| \cdot [1-(1-1/10)^{|q_j|/\|\gamma_{1j}\|}] \big\} \wedge \\ &\mathcal{Q}'_{\text{let}} = \big\{ c_2^{\gamma_2} \big| \gamma_2 \sqsubseteq \gamma_{1j} \wedge c_{1j}^{\gamma_1} \in \text{dom}(q_j) \wedge q_j \in \mathcal{Q}_{\text{let}} \wedge \|\gamma_2\| =^! \|\gamma_{1j}\| \cdot [1-(1-1/10)^{|q_j|/\|\gamma_{1j}\|}] \big\} \wedge \\ &\mathcal{Q}'_{\text{exists}} = \big\{ c_2^{\gamma_2} \big| \gamma_2 \sqsubseteq \gamma_{1j} \wedge c_{1j}^{\gamma_1} \in \text{dom}(q_j) \wedge q_j \in \mathcal{Q}_{\text{exists}} \wedge \|\gamma_2\| =^! \|\gamma_{1j}\| \cdot [1-(1-1/2)^{|q_j|/\|\gamma_{1j}\|}] \big\} \end{aligned} \quad \text{(CARD-SEL)}$$

Figure 8: Inference rules for merge and select

Rule CARD-MERGE-2 handles a special case: If there exists an outer tuple variable in the Merge operator, the outer semantics well-known from outer joins is used, i. e., for every tuple sequence where a match does not exist, the tuple still appears in the Cartesian product and all non-matching items are replaced by empty sequences [Mat09]. Here, the output cardinality is simply determined by the cardinality of the tuple sequence associated with the operator connected to the outer tuple variable.

The Select operator also calculates the Cartesian product on its input streams. In contrast to the Merge operator, the Select operator can contain tuple variables with mixed quantifiers: *for*, *let*, or *exists*. The Select operator is the most versatile operator in XQGM, as it allows to express value-based joins and simple selection predicates as well as triggering XQuery *for-let* bindings. Rule CARD-SEL describes the cardinality inference for Select operators, where Q_q (Q'_q) is the set of all columns that are connected to q-quantified (output) tuple variables. The *for*-quantified tuple variables "drive" the output generation process. On the other hand, the tuple sequences bound to *let*-quantified tuple variables are "nested" into the results generated by *for*-quantified tuple variables, whereas *exists*-quantified tuple variables only serve for existence tests and do not contribute to the output.

4.2.5 Inference Rules for Set Operators

$$\frac{|q_1| = |q_2| = \ldots = |q_n|}{|\cup_{i=1}^{n} q_i| = \sum_{i=1}^{n} |q_i| \wedge \mathrm{dom}(\cup_{i=1}^{n} q_i) = \cup_{k=1}^{|\mathrm{dom}(q_1)|} \left\{ c_2^{\gamma_2} \middle| \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_{1k}^{\gamma_1} \in \mathrm{dom}(q_1) \wedge \|\gamma_2\| =^! \sum_{i=1}^{n} \|\gamma_{ik}\| \right\}} \quad \text{(CARD-UNION)}$$

$$\frac{a_1^{\alpha_1} \in \mathrm{dom}(q_1) \wedge \ldots \wedge a_n^{\alpha_n} \in \mathrm{dom}(q_n) \wedge |q_k| = \min\{|q_1|, \ldots, |q_n|\}}{|\cap_{i=1}^{n} q_i| = |q_k| \cdot 2/3 \wedge \mathrm{dom}(\cap_{i=1}^{n} q_i) = \left\{ c_2^{\gamma_2} \middle| \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \mathrm{dom}(q_k) \wedge \|\gamma_2\| =^! \|\gamma_1\| \cdot \left[1 - (1 - 1/10)^{|q_k|/\|\gamma_1\|}\right] \right\}} \quad \text{(CARD-INTERSECT)}$$

$$\frac{}{|q_1 \setminus q_2| = |q_1| \cdot 1/10 \wedge \mathrm{dom}(q_1 \setminus q_2) = \left\{ c_2^{\gamma_2} \middle| \gamma_2 \sqsubseteq^! \gamma_1 \wedge c_1^{\gamma_1} \in \mathrm{dom}(q_1) \wedge c_1^{\gamma_1} \notin \mathrm{dom}(q_2) \wedge |\gamma_2\| =^! \|\gamma_1\| \cdot \left[1 - (1 - 1/10)^{|q_1|/\|\gamma_1\|}\right] \right\}} \quad \text{(CARD-DIFFERENCE)}$$

Figure 9: Inference rules for set operators

Finally, XQGM provides three set operators: Union, Intersect, and Difference. In XQGM, Union and Intersect are n-way operators and only Difference is a binary operator.

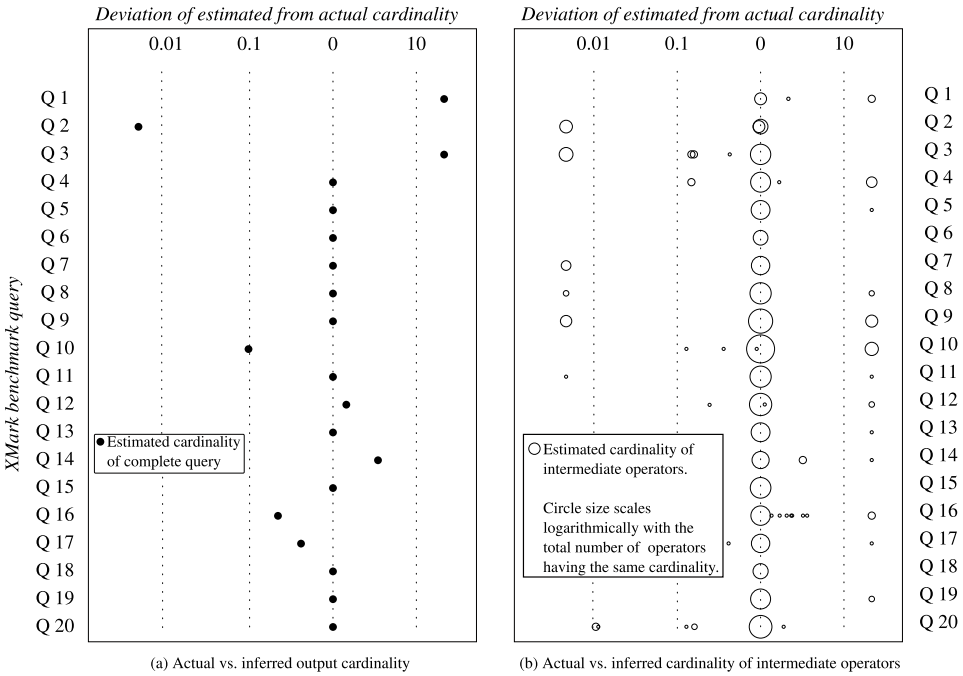
Rule CARD-UNION describes the cardinality inference for the Union operator. Here, we assume that all input operators ($q_1 \ldots q_n$) have the same output cardinality and all input tuple sequences have the same active domains whose value range may differ. In this case, γ_{ik} denotes the active domain of operator i in column k.

The cardinality inference for the Intersect operator is described by rule CARD-INTERSECT. In this situation, q_k denotes the first operator whose cardinality is minimal w. r. t. the cardinality of the remaining operators. In experiments with our query optimizer, we found out that a constant factor of $2/3$ is a good heuristics for the selectivity of the n-way Intersect operator.

Finally, rule CARD-DIFFERENCE illustrates the cardinality inference of the binary Difference operator that reuses the standard formula well-known from the relational context.

5 Empirical Evaluation

Finally, this section discusses the empirical evaluation of the inference rules. In this context, we are not focussing on the cardinality inferences alone. Instead, we are interested in their interplay with the cost-based query optimizer and its ability to derive scalable QEPs.



(a) Actual vs. inferred output cardinality

(b) Actual vs. inferred cardinality of intermediate operators

Figure 10: Actual vs. inferred cardinalities for the XMark benchmark queries

Our experiments were conducted on an Intel XEON quad core (3350) computer (2.66 GHz CPUs, 4 GB of main memory, 500 GB of external memory) running Linux with kernel version 2.6.14. Our native XDBMS server—implemented using Java version 1.6.0_07—was configured with a page size of 16 KB and a buffer size of 256 16-KB frames. The experimental results for the query execution times reflect the average values of five runs on a cold database buffer.

For the empirical evaluation, we used the XMark benchmark queries—a set of simple to complex XQuery expressions [SWK+02]—that serve very well to test the effectiveness and the ability of an XQuery processor to provide scalable QEPs. If not mentioned otherwise, we used an XMark document with scaling factor $f = 1.0$ that corresponds to an approximate document size of 110 MB.

In our first experiment, we used the cardinality inference rules to estimate the final output cardinality of each query and of each individual XQGM operator. Let us first have a

look at the results for estimated final output cardinalities. In Figure 10 a), we recorded the deviation of the estimated value from the actual cardinality on the x-axis. If there is a filled circle drawn at position 0, this means that there was no deviation between the estimated and the actual cardinality at all. On the other hand, if a circle is drawn at 10 or 0.1, then the estimated cardinality became victim of a 10-fold overestimate or a 10-fold underestimate, respectively. All circles depicted at the left-hand side of 0.01 and on the right-hand of 10.0 summarize overestimates and underestimates beyond these limits.

For 14 out of 20 queries, the cardinality estimates are close or equal to the actual cardinality. For queries Q 1–Q 3, the final cardinality estimates deviate significantly from the actual cardinality. For query Q1, the selection predicate has a much lower selectivity than estimated using the 10 % heuristics. This problem can be easily overcome using refined statistics on value distribution, e. g., histograms. For queries Q 2 and Q 3, the selectivity of the positional predicates was not estimated correctly. Unfortunately, the error was propagated up to the estimate of the final query result. Nevertheless, the cardinalities for all performance-critical operators, such as access paths and SJs were estimated correctly. Therefore, the ability of the query optimizer to provide scalable QEPs is not affected in these situations—as indicated by Figure 11.

Our second experiment looked at the deviation between estimated and actual cardinalities for individual XQGM operators (Figure 10 b). In Figure 10 a), all circles have the same size. In contrast, in Figure 10 b), we use the same notation as in Figure 10 a), except the fact that the circles are not filled anymore and their size is scaled logarithmically w. r. t. the total number of operators that have the same deviation ratio. For example, for the majority of operators in query Q 10, the inference rules estimated the correct output cardinality (in fact, 88 % of the inferred cardinalities were correct), hence, the largest circle is drawn at position 0. Moreover, the tiny circles between 0.1 and 0 and the medium-sized circle on the right-hand side of position 10 indicate outliers, from which the estimation rules were able to recover successfully. In total, for almost all queries, the inference rules provided for the majority of operators exact cardinality estimates. Even though the inference rules produced some outliers, they have little effect on the shape of the final QEP, because they mostly are related to GroupBy and Unnest operators that cannot be removed and where no alternative implementation exists.

Our third experiment shows that the inference rules are robust enough to support the query optimizer in deriving scalable query plans. Figure 11 depicts the execution times of the XMark benchmark queries on different document sizes (ranging from 110 KB to 1.1 GB). Besides the selection of implementations and SJ reordering, the cost-based query optimizer [WHdS10] selected indexes based on the recommendations of XTC's auto-indexing feature [SH10]. For these tasks, reliable cardinality information is crucial to derive sufficiently efficient QEPs.

The execution times of most queries scale linearly with the document size. For small documents (size $\leq$ 10 MB), the average scale factor is even at most 6.85, i. e., an increase of the document size by factor 10 results only in a 6.85 times longer execution time. For the largest document in our experiment (1.1 GB), we still get an average scale factor of 10.5 for all queries except of Q11 and Q12. In contrast, queries Q11 and Q12 are very complex, include non-selective joins, and produce very large intermediate results that

scale quadratically with the document size. Therefore, the execution time of optimal plans increases quadratically, too. Hence, this is not an error of the optimizer but simply reflects the document and query characteristics.

To once again emphasis the importance of reliable cardinality estimates for the evaluation of value-based joins, for query Q9, using the cardinality inference rules, the optimizer was able to propose a plan that was almost 18 times faster than without refined cardinality information. Though, it is noteworthy how query optimization time (from query parsing to the generation of the physical plan) relates to the overall execution time: On average, 97.62% of the time is spent for query execution and only 1.54% of the time was consumed by query optimization. Hence, cardinality inference has only a low impact on optimization time. As indicated by earlier experiments, non-optimized QEPs are up to two orders of magnitude slower than their optimized counterparts. Consequently, it is worth spending this small amount of time to get significantly better results.

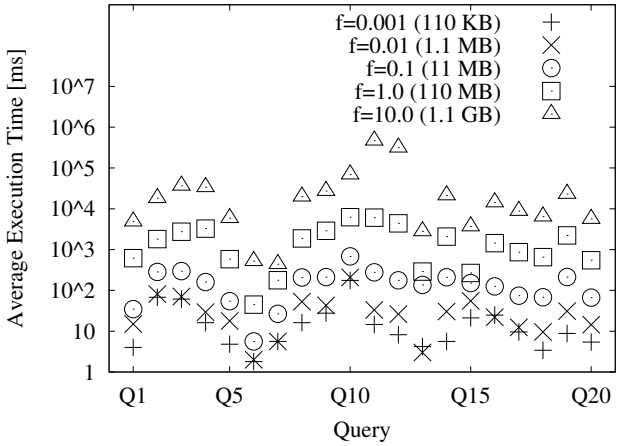


Figure 11: Scalability on XMark benchmark queries

6 Conclusions and Future Work

In this paper, we introduced and experimentally evaluated a set of inference rules that allow for effective cardinality estimation in native XDBMSs. To the best of our knowledge, this is the first approach that enables more precise cardinality estimation in such systems. Moreover, we have discussed how we can use the inference rules to provide termination criteria for cost-based query unnesting and to support an appropriate selection of value-based join algorithms. In combination with a rich set of rewrite rules [WHdS10] and our generalized access path [HMB$^+$10], the inference rules provide the foundation for XTC's cost-based query optimizer and enable it to derive scalable QEPs for a wide range of XQuery expressions. Though our experimental results are promising, there is still room for optimization. By refining the generalized 10% heuristics and by focussing on a more precise treatment of positional predicates, we expect further improvements in estimation accuracy.

References

[AAN01] Ashraf Aboulnaga, Alaa R. Alameldeen, and Jeffrey F. Naughton. Estimating the Selectivity of XML Path Expressions for Internet Scale Applications. In *Proc. VLDB Conf.*, pages 591–600, 2001.

[Agu10] Jose Aguiar Moraes Filho. *Summarizing XML Documents: Contributions, Empirical Studies, and Challenges*. PhD thesis, University of Kaiserslautern, Germany, 3 2010.

[AKJP+02] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. ICDE Conf.*, pages 141–154, 2002.

[BEH+06] Andrey Balmin, Tom Eliaz, John Hornibrook, et al. Cost-Based Optimization in DB2 XML. *IBM Systems Journal*, 45(2):299–320, 2006.

[BKS02] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proc. SIGMOD Conf*, pages 310–321, 2002.

[FHK+02] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, et al. Anatomy of a Native XML Base Management System. *VLDB Journal*, 11(4):292–314, 2002.

[FHR+02] Juliana Freire, Jayant R. Haritsa, Maya Ramanath, et al. StatiX: Making XML Count. In *Proc. SIGMOD Conf.*, pages 181–191, 2002.

[FM07] Damien K. Fisher and Sebastian Maneth. Structural Selectivity Estimation for XML Documents. In *Proc. ICDE Conf.*, pages 626–635, 2007.

[FMM+07] Mary Fernández, Ashok Malhotra, Jonathan Marsh, et al. XQuery 1.0 and XPath 2.0 Data Model (XDM)—W3C Recommendation 23 January 2007. `http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/`, 2007.

[GW97] Roy Goldman and Jennifer Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. VLDB Conf.*, pages 436–445, 1997.

[HMB+10] Theo Härder, Christian Mathis, Sebastian Bächle, et al. Essential Performance Drivers in Native XML DBMSs. In *Proc. SOFSEM Conf.*, volume 5901 of *LNCS*, pages 29–46, 2010.

[JAKC+02] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, et al. TIMBER: A Native XML Database. *VLDB Journal*, 11(4):274–291, 2002.

[MAG+97] Jason McHugh, Serge Abiteboul, Roy Goldman, et al. Lore: A Database Management System for Semistructured Data. *ACM SIGMOD Record*, 26(3):54–66, 1997.

[Mat07] Christian Mathis. Integrating Structural Joins into a Tuple-Based XPath Algebra. In *Proc. BTW*, volume 103 of *LNI*, pages 242–261, 2007.

[Mat09] Christian Mathis. *Storing, Indexing, and Querying XML Documents in Native XML Database Management Systems*. PhD thesis, University of Kaiserslautern, Germany, 2009.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. SIGMOD Conf.*, pages 39–48, 1992.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, et al. Access Path Selection in a Relational Database Management System. In *Proc. SIGMOD Conf.*, pages 23—34, 1979.

[Sar03] Carlo Sartiani. A General Framework for Estimating XML Query Cardinality. In *Proc. DBPL*, pages 257–277, 2003.

[SH10] Karsten Schmidt and Theo Härder. On The Use of Query-Driven XML Auto-Indexing. In *Proc. ICDE SMDB Workshop*, 2010.

[SWK+02] Albrecht Schmidt, Florian Waas, Martin L. Kersten, et al. XMark: A Benchmark for XML Data Management. In *Proc. VLDB Conf*, pages 974–985, 2002.

[TGMS08] Jens Teubner, Torsten Grust, Sebastian Maneth, and Sherif Sakr. Dependable Cardinality Forecasts for XQuery. *Proc. VLDB Endowment*, 1(1):463–477, 2008.

[WHdS10] Andreas M. Weiner, Theo Härder, and Renato Oliveira da Silva. Visualizing Cost-Based XQuery Optimization. In *Proc. ICDE Conf.*, pages 1165–1168, 2010.

[ZÖAI06] Ning Zhang, M. Tamer Özsu, Ashraf Aboulnaga, and Ihab F. Ilyas. XSEED: Accurate and Fast Cardinality Estimation for XPath Queries. In *Proc. ICDE*, page 61, 2006.

A XQGM Components

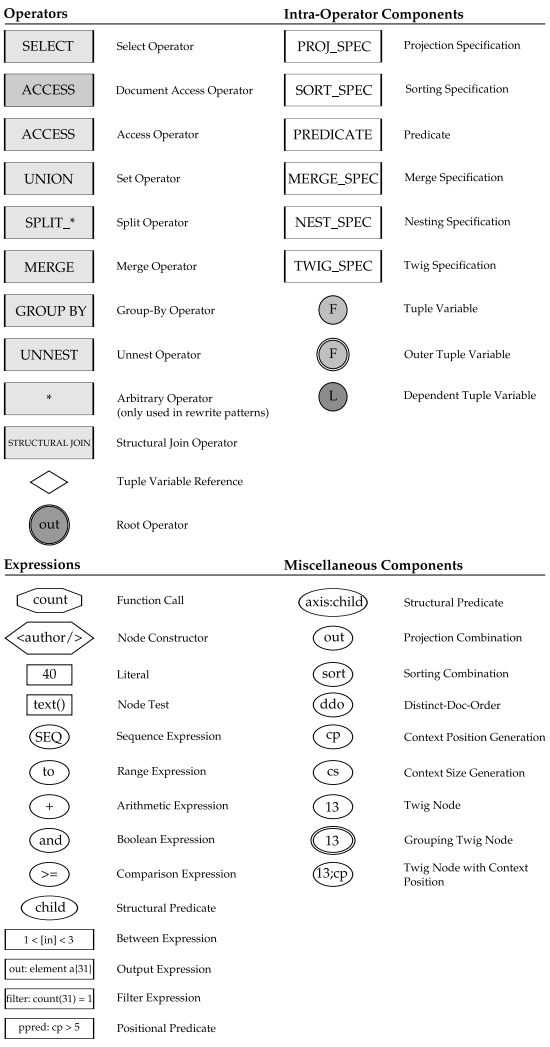


Figure 12: Overview XQGM components [Mat09]