Pathfinder Meets DB2®

Relational XQuery Optimization Techniques

Manuel Mayr

Advisor: Torsten Grust

Technische Universität München
Munich, Germany
manuel.mayr@in.tum.de

ABSTRACT

We are taking the next big step towards the goal of a *purely relational XQuery implementation.* The *Pathfinder* XQuery compiler has been enhanced by a code generator that emits SQL. This code generator targets *off-the-shelf* relational database systems (*e.g.*, DB2®) and turns them into efficient and scalable XQuery processors. Our approach neither depends on modifications of the database kernel, nor do we rely on built-in XML-specific functionality (SQL/XML, for instance). For that reason we are able to rest this work on query optimization techniques that have proven their effectiveness for pure SQL workloads.

Here, we will describe (1) how *distribution statistics* and *statistical views* can accompany the relational encoding of an XML document to provide information about its node hierarchy, (2) the use of *generated columns* and *materialized query tables* to precompute aspects of XPath step evaluation, (3) how the system's *index design wizard* can automatically advise on the creation of indexes that, for example, enable index-only XPath location path processing, and (4) *optimization profiles*, a final fallback that enables fine-grained control over DB2's query execution plans. Performance experiments indicate the potential of the XQuery processor that results from this synthesis of *Pathfinder* and DB2.

1. INTRODUCTION

Relational database systems incorporate the most sophisticated query optimizers and statistical cost models available today. Although these systems have originally been built to operate over table-shaped data, their infrastructure has already been shown to also provide efficient support for non-relational data models and languages. The *Pathfinder* XQuery compiler [7] uses relational encodings of both, the XQuery data model and the language's dynamic semantics, to deploy RDBMSs as efficient and scalable XQuery processors.

For the XQuery dialect sketched in Table 1, *Pathfinder* employs a technique coined *loop lifting* [8] to generate an intermediate algebraic representation of an input XQuery expression. *Pathfinder* has been designed with a close eye on the, sometimes intricate, XQuery semantics. At the same time, the compiler relies on a rather restricted variant of relational algebra whose operators are chosen to match the capabilities of modern SQL query processors. This enabled the construction of a code generator that can target any SQL:1999-capable RDBMS [10].

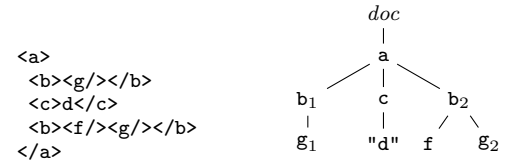
atomic literals	document order (e_1 `>>` e_2)
sequences (`(`e_1`,`e_2`)`)	node identity (e_1 `is` e_2)
variables (`$`v)	`union`, `intersect`, `except`
`let` `$`v `:=` e_1 `return` e_2	arithmetics (`+`,`-`,`*`,`idiv`,...)
`for` `$`v [`at` `$`p] `in` e_1 `return` e_2	(general) comparisons (`=`, `eq`, ...)
`if (`e_1`) then` e_2 `else` e_3	user-defined functions
e_1 `order by` e_2`,...,`e_n	`fn:doc(·)`, `fn:root(·)`, `fn:data(·)`
`unordered {`e`}`	`fn:distinct-values(·)`
node constructors	`fn:count(·)`, `fn:sum(·)`, `fn:max(·)`
XPath (e_1`/`s`[[`e_2`]]`)	`fn:position()`, `fn:last()`
`typeswitch (`e_1`) case [`$v` `as`] `$t$ `return` e_2 `...default return` e_n	

Table 1: Excerpt of XQuery dialect supported by *Pathfinder* (s: XPath step, t: sequence type).

Relational XML Document Encoding. *Pathfinder* employs a simple node-based XML encoding in which each XML node v is registered with its document order rank $pre(v)$, the number $size(v)$ of nodes in the subtree below v, and its distance $level(v)$ to the document root. In addition to these structural properties, the relational encoding contains columns $kind(v) \in \{\texttt{doc}, \texttt{elem}, \texttt{text}, \dots\}$ (node kind), $name(v)$ (tag or attribute name), and $value(v)$ (content of text, comment, or processing-instruction nodes). Figure 1 illustrates a sample XML document and its tabular encoding. The semantics of XPath location *steps* map into conjunctive range predicates over this node encoding table [8]. The evaluation of location *paths* is further supported by column $guide(v)$, a representative of the tagged rooted path that leads to v in the style of Widom's DataGuides [4]: in Figure 1, the nodes $\texttt{g}_1$ and $\texttt{g}_2$ are both reachable via the rooted path `/a/b/g` and thus are assigned identical *guide* entries.

Basic query blocks. Much like a programming language compiler, *Pathfinder* uses data flow analysis techniques to isolate *basic blocks* in the intermediate algebraic plans [7]. In this particular context, a basic block is defined as the maximum section of an algebraic plan that can be equivalently translated into a single SQL query. Since the DAG-shaped

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT'08, March 25–30, 2008, Nantes, France.
Copyright 2008 ACM 978-1-59593-926-5/08/0003 ...$5.00.




```
<a>
 <b><g/></b>
 <c>d</c>
 <b><f/><g/></b>
</a>
```

(a) XML document. (b) Node hierarchy.

pre	size	level	guide	kind	value	name
0	8	0	0	doc	"sample.xml"	□
1	7	1	1	elem	□	a
2	1	2	2	elem	□	b_1
3	0	3	3	elem	□	g_1
4	1	2	4	elem	□	c
5	0	3	5	text	"d"	□
6	2	2	2	elem	□	b_2
7	0	3	6	elem	□	f
8	0	3	3	elem	□	g_2

(c) Encoding table doc (□ indicates NULL).

Figure 1: Tabular encoding of document sample.xml.

algebraic plans quite significantly diverge from the common π–σ–$\bowtie$ pattern—*e.g.*, exhibiting calls to SQL OLAP functions like DENSE_RANK—the compilation of a single XQuery query will generally yield a sequence of SQL statements which, assembled in a SQL *common table expression* (WITH ...), jointly implement the dynamic semantics of the input XQuery expression. Figure 2 depicts the SQL code emitted for Query Q6 of the XMark benchmark [11]

```
let $auction := doc("auction.xml")
return
  for $b in $auction/site/regions
  return count($b//item)
```
(Q6)

and also emphasizes the basic blocks that have been identified.

The inherent tree structure of XML data (and the resulting non-uniform value distribution in the relational encoding of such data) plus the unusual shape of the resulting query plans present a substantial challenge for the RDBMS' built-in SQL-centric query optimizer. In the following sections we will discuss how *query processing infrastructure that is already built into existing database kernels*, in this case IBM DB2® V9.1, can be exploited to fully make up for the system's lack of tree awareness.

2. TURNING DB2'S TUNING KNOBS

IBM DB2 is equipped with one of the industry's most versatile query engines but—as is typical in RDBMSs—its cost-based query optimizer remains critically dependent on the availability of proper statistics. By default, DB2 collects basic statistics about table and column cardinalities (of which Table 2 shows a sample). We will see that this is not quite sufficient if DB2 is to be used as a back-end for the *Pathfinder* XQuery compiler.

The DB2 optimizer chooses specific access paths primarily based on the *filter factor* ($0 \leqslant ff \leqslant 1$) which measures the selectivity of a given predicate p when applied to a table of source rows:

$$ff = \frac{\#\text{ of rows qualifying against } p}{\#\text{ of source rows}} \quad .$$

Table Statistics	Column Statistics
# of pages in use	Column cardinality
# of pages containing rows	Average column width
# of relocated rows (after update)	Second highest value
# of rows (cardinality)	Second lowest value
	# of NULLs in column

Table 2: Basic statistics maintained by DB2.

SEQNO	COLVALUE	COLCOUNT
1	□	3,024,329
2	person	138,236
3	category	120,839
4	text	105,114
5	date	90,182
⋮	⋮	⋮
64	featured	2,210
65	edge	999
⋮	⋮	⋮

Table 3: Tag name distribution in an 112 MB XMark document instance captured by SYSCAT.COLDIST.

In the presence of complex compound predicates, in particular, the optimizer is forced to rely on *estimated* filter factors if only basic statistics are available. Such estimates are based on assumptions about the (absence of) correlation between table columns and the even value distribution of active column domains. Whenever these assumptions are not justified, the optimizer risks to select less-than-optimal access paths. The impact at query runtime may be significant.

2.1 Distribution statistics

Pathfinder-generated SQL queries primarily operate over the tabular encodings of XML documents and fragments (see Figure 1(c)). This encoding of tree-shaped data inherently leads to non-uniform value distributions in the columns of table doc.

Frequent values. This non-uniformity surfaces repeatedly but is already apparent in the distribution of XML element tag names in column name of table doc. An instance of the XMark benchmark document, for example, contains a single element with tag site (the document's root element) but, dependent on the instance's size, a comparably huge number of person elements. In consequence, the filter factors for the SQL predicates name = 'site' and name = 'person' will vary widely and a uniformity assumption for these name tests will clearly lead the optimizer astray. Similar observations hold for the encoding columns level (in any XML document, all nodes but one are located at a level > 1) and guide (due to the irregular shape of XML node hierarchies).

To properly capture such skewed distributions, IBM DB2 provides the option to collect *frequency statistics* for selected columns. Table 3 shows an excerpt of the DB2 system catalog table SYSCAT.COLDIST after it has been populated by an appropriate DB2 runstats run: the frequency of the tag name distribution now is explicit.

We are able to control the catalog space devoted to collect such explicit frequency statistics: the runstats parameter num_freqvalues n instructs the systems to capture the distribution of the n most frequent values only. At query

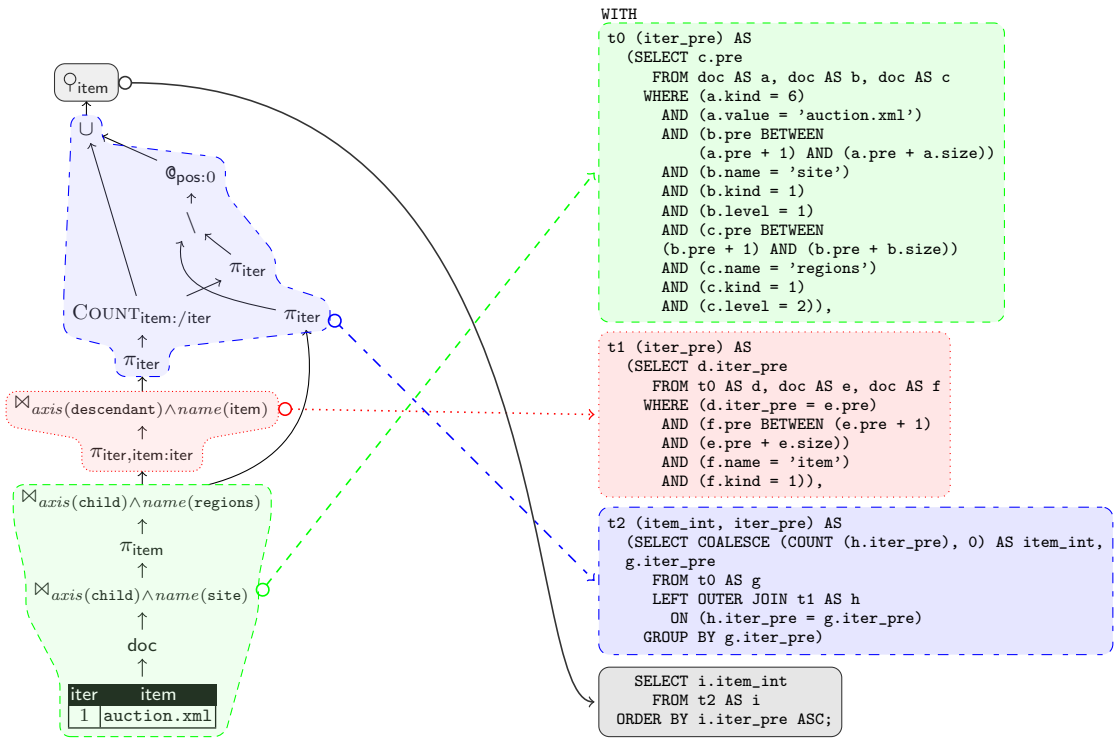


Figure 2: SQL code generated for XMark Query Q6 (the dashed regions identify basic blocks). Predicates $axis(\cdot)$ and $name(\cdot)$ [8] map the semantics of XPath axes and name/kind test onto the tabular encoding.

compilation time, the optimizer exploits such column distribution statistics as follows (let k denote the key value occurring in a predicate p like `name = k`):

1. If k is among the n most frequent values, the optimizer directly reads the number of qualifying rows off of the catalog table `SYSCAT.COLDIST`.

2. Otherwise, the optimizer falls back and assumes a uniform distribution of the non-frequent values.

If the column cardinality is sufficiently small, the system can maintain frequency statistics for the *complete* active domain of a column while investing only marginal catalog space. Since XML trees typically are shallow (*e.g.*, $1 \leqslant$ level $\leqslant 13$ for XMark instances) and only contain a limited number of distinct tag and attribute names (XMark: about 70), we instructed DB2 to collect complete value distribution statistics for these columns. In effect, `SYSTAT.COLDIST` thus implements multiple perfect histograms in the sense of [9].

Value predicates and K-quantiles. In the simple encoding of Figure 1(c), column value of SQL type `VARCHAR(·)` holds *any* textual node content. The evaluation of numeric value predicates as in the XPath location path

```
doc("auction.xml")/site//open_auction[reserve > 10]
```

thus compiles into SQL range predicates of the form

```
10 < CAST(doc.value AS DECIMAL) .
```

As is, the required `CAST(·)` from `VARCHAR(·)` to `DECIMAL` inhibits the consultation of K-quantile statistics for column value (recording the value v below which K values lie). At the time of writing, *Pathfinder*'s XML encoding and SQL code generator are modified to exploit additional columns decvalue and intvalue (of SQL types `DECIMAL` and `INTEGER`, respectively) which hold textual node content cast to numeric types. For the above XPath value predicate, *Pathfinder*'s static typing phase will lead the code generator to emit the SQL predicate `10 < doc.decvalue` instead and thus allow the exploitation of K-quantile statistics for column decvalue. At the same time, additional columns like decvalue and intvalue facilitate the evaluation of implicit type casts which are pervasive in XQuery due to the language's node atomization semantics [2].

2.2 Statistical Views

While full distribution statistics already go a long way towards our goal to equip the DB2 optimizer with appropriate knowledge about XML documents, this type of statistics still fails to capture the intrinsic *structural irregularity* of tree-shaped XML data. In XQuery, XPath location steps are used to explore this tree structure. *Pathfinder* compiles a location step α`::`nt into a self-join over the encoding table doc (see Figure 2 and [8]): in the join predicate $axis(\alpha) \wedge name(nt)$, $axis(\alpha)$ represents the semantics of XPath axis α while $name(nt)$ embodies the step's name and kind test.

In the absence of more specific information, DB2 evaluates the filter factor for the predicate in doc $\underset{c_1=c_2}{\bowtie}$ doc as

$$ff = \frac{1}{\max\left(\texttt{COLCARD}(c_1), \texttt{COLCARD}(c_2)\right)} ,$$

where `COLCARD(·)` denotes the estimated cardinality of a col-

```
WITH
t0 (guide) AS
   (VALUES (8), (126), (244), (362), (480), (598)),
t1 (item_int) AS
   (SELECT COUNT (*) AS item_int
      FROM doc AS a2, doc AS a3, t0 AS a4
     WHERE (a2.kind = doc)
       AND (a2.value = 'auction.xml')
       AND (a3.pre BETWEEN (a2.pre + 1)
                       AND (a2.pre + a2.size))
       AND (a3.guide = a4.guide))
(SELECT a5.item_int
   FROM t1 AS a5);
```

Figure 3: SQL code for Query Q6 (using DataGuide-style path identifiers).

umn's active domain (Table 2). The uniformity assumptions built into this formula fails to acknowledge the irregular node distribution in table doc. Clearly, the XPath step `doc("auction.xml")/descendant::node()` selects *all* nodes of the context document[1] and thus should be assigned a filter factor of 1. As is expected, DB2 fails to infer this filter factor from the step's equivalent SQL query, primarily because of the involved range predicate (the equivalent of *axis*(descendant)):

$Q \equiv$
```
SELECT a2.*
  FROM doc AS a1, doc AS a2
 WHERE a1.kind = doc
   AND a1.value = 'auction.xml'
   AND a2.pre BETWEEN (a1.pre + 1)
                  AND (a1.pre + a1.size) .
```

At this point, DB2's *statistical views*, for which the system records statistics but not the result itself, can be of valuable help:

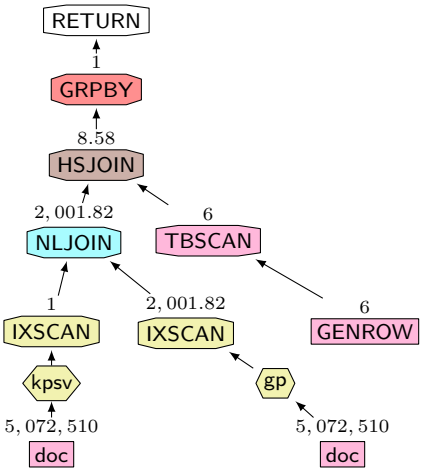
```
CREATE VIEW docaccess AS Q;
ALTER VIEW docaccess ENABLE QUERY OPTIMIZATION .
```

A statistical view permits the system to "peek" at the result of (intermediate) arbitrarily complex query expressions already at query compile time—and thus enables informed decisions about access path and join algorithm selection.

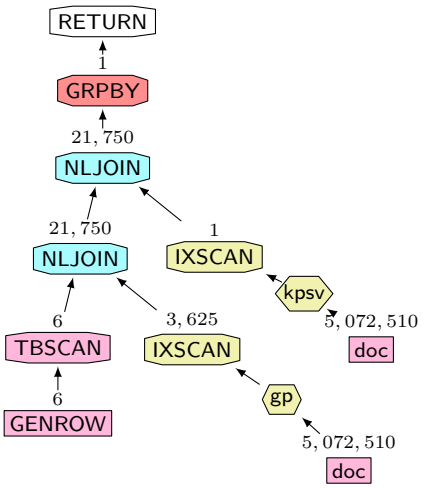
Below, let us briefly discuss the impact of the statistical view docaccess on the DB2-generated execution plans for XMark Query Q6 of Section 1. Here, for space reasons, we discuss a variant of the SQL common table expression for Q6 (Figure 3) in which *Pathfinder* exploited the DataGuide-style path information [6, 4] in column guide of table doc: the intermediate unary table t0 directly supplies the path identifiers (*i.e.*, guide column entries) for those nodes reachable by path expression `doc("auction.xml")/site/regions//item` (six distinct paths in the input XMark instance match this location path).

Without statistical view. In absence of the statistical view docaccess, DB2 emits the query execution plan depicted in Figure 4(a). We are reproducing these plans in a form closely resembling the output of DB2's visual explain facility. Nodes in these plans represent operators of DB2's variant of physical algebra—all operators relevant for the present discussion are introduced in Table 4. The plan relies

[1] *Pathfinder* judiciously introduces such steps to access the nodes of an XML document for a given URI.



(a) Without statistical view docaccess.



(b) With statistical view docaccess.

Figure 4: DB2-generated execution plans for Q6. Figures indicate estimated cardinalities.

on two indexed access paths: (1) a B-tree with key columns ⟨kind, pre, pre + size, value⟩, abbreviated kpsv, and (2) a B-tree with key ⟨guide, pre⟩, abbreviated gp (Section 2.3 sheds light on the choice of these particular indexes).

The system succeeds to perfectly estimate a cardinality of 1 for the left (outer) leg of the NLJOIN operator: the single row encoding the document node of XML document `'auction.xml'` qualifies with respect to predicate `kind = doc AND value = 'auction.xml'`. The output cardinality estimate for the NLJOIN (the join between a2 and a3 in the SQL code of Figure 3), however, is already off by a factor of more than 2,500: DB2 estimates a cardinality of 2,001.82 while we know that all nodes of the document will qualify (the 112 MB XMark instance contains 5,072,510 nodes overall). In consequence, the system severely underrates the cost of the NLJOIN and proposes an evaluation plan that comes at a price about 30 times higher than for the plan generated in the presence of the statistical view docaccess.

With statistical view. DB2's assessment of the plan cost radically changes once the statistical view docaccess is available (Figure 4(b)). Now knowing that an early self-join of

Operator	Semantics	Operator	Semantics
RETURN	Result row delivery	GRPBY	Grouping and/or aggregation
NLJOIN	Nested-loop join (left leg: outer)	TBSCAN	Temporary table scan
IXSCAN	B-tree scan	kpsv	Access index with key columns kpsv
GENROW	Literal table	doc	XML encoding table

Table 4: Relevant DB2 physical plan operators.

doc will return 5,072,010 nodes, the corresponding NLJOIN is postponed. Instead, the optimizer chooses to use the DataGuide information in table `t0` (the literal table of six rows surfaces as a GENROW operator in the plan) and introduces an indexed NLJOIN that uses B-tree index gp to selectively access those nodes reachable via the location path `doc("auction.xml")/site/regions//item`. The value distribution statistics for column guide in `SYSCAT.COLDIST` lead the optimizer to a perfect estimate of 21,750 `item` element nodes that will be retrieved by the index accesses. Only the second NLJOIN will perform the document access and touch no more than the required 21,750 nodes. This plan evaluates XMark Query Q6 as efficient as we could hope for—which is remarkable, considering that a SQL-centric query optimizer and engine took all responsibility for the evaluation of an XQuery expression.

Table 5 documents the effect on the complete XMark benchmark query suite [11] once we supplied the query optimizer with the `docaccess` statistical view. The table reports on the relative improvement in execution time when queries are run against a 112 MB XML document instance (the 100 % line marks the execution time in the absence of statistical views.). Dependable filter factor estimates lead to DB2 to plans whose execution time improve by up to two orders of magnitude. On a dual 3.2 GHz Intel Xeon™ Linux 2.6 host with 8 GB of primary and SCSI-based secondary disk memory, the far majority of queries executes in significantly less than one second (the notoriously complex Queries Q9–Q12 are an exception). Most interestingly, these execution times clearly undercut those required by the DB2 built-in pureXML™ XQuery processor: in [6] we reported an up to five-fold advantage that increases with growing XML input sizes.

2.3 Further DB2 Gadgets on the Workbench

More than 30 years of research and development in relational query processing engines have produced an extensive set of tools and techniques, many of which are available—in one form or another—in DB2 V9. Here, we sketch a number of further options offered by the larger DB2 kernel infrastructure and how they can help to turn the system into a efficient and scalable XQuery processor. All of these we will scrutinize in the course of this Ph.D. project.

Autonomous index design. Since the *Pathfinder* compiler emits pure SQL, we may call on the DB2 design advisor `db2advis` and let the system adapt its internals to the typical queries generated by the XQuery compiler.[2] The system's autonomous proposals for B-tree indexes are of particular interest in this context: reasonable proposals indicate that the *Pathfinder*-generated SQL common table expressions constitute a workload that DB2 indeed can cope with.

In fact, once we fed the SQL code for XMark Query Q6 (Figure 3), `db2advis` suggested the B-tree indexes kpsv (with composite key ⟨kind, pre, pre + size, value⟩) and gp (with key ⟨guide, pre⟩) that we have discussed earlier in Section 2.2. Both indexes are interesting in the sense that their keys are prefixed with *low*-selectivity columns: (1) there are only six distinct XML node kinds registered in column kind, and (2) there are significantly less distinct paths (column guide) than distinct nodes (column pre) in a typical XML document. Such low-selectivity key prefixes effectively lead to the construction of *partitioned B-trees*. This kind of B-tree enjoys a number of desirable properties [5]—a name-partitioned B-tree, for example, implies zero-redundancy tag name storage if compression is used on the name-prefixed keys [1]. Further, a B-tree with key ⟨guide, pre⟩ partitions the XML nodes of a document based on their rooted paths. B-tree gp thus supports DataGuide-based node acccess and effectively materializes the result of (selected) XPath location paths.

Note that, since *Pathfinder*-generated SQL queries exhibit a high degree of regularity, any such index will be of high utility to the database system.

Generated columns and materialized query tables. The just mentioned regularity of queries immediately suggests that it may be worthwhile to *precompute* specific query parts that recur in XPath and XQuery evaluation. *Pathfinder* offers a number of hooks where such an investment in storage space may pay off in terms of reduced query execution time. The evaluation of XPath axis steps over the encoding of table doc, for example, leads to the evaluation of range predicates of the form `a2.pre BETWEEN (a1.pre) AND (a1.pre + a1.size)`. The sum pre + size is omnipresent in *Pathfinder*-generated queries (in fact, column size is rarely accessed otherwise). We thus created an additional *generated column* directly containing the sum—such columns may be indexed which is crucial for XPath location step evaluation (`db2advis` indeed proposed an index with key ⟨kind, pre, pre + size, value⟩).

Whenever an XML node is used in a context where an atomic value is required, the XQuery semantics implicitly adds an atomization (`fn:data(·)`) operation. In non-validated XML documents, atomization entails the traversal of document subtrees plus the collection and concatenation of text node contents to compute the string value of the atomized nodes [2]. Since this costly operation is pervasive in XQuery, we consider the construction of precomputed *atomization indexes* which will benefit from DB2's materialized query table and row compression capabilities [3].

Optimization profiles. With release DB2 V9, IBM introduced *optimization profiles* as a means to reach deep into the optimizer's execution plan generation process. Optimization profiles, specified in the form of XML fragments, can outvote local optimizer decisions with respect to access path selection, join algorithm choice, and query (un)nesting. Meant to be used with the SQL query of Figure 3, the following XML snippet instructs the optimizer to access the DataGuide B-tree gp first and only then perform the self-NLJOIN of table doc:

[2] Only with the recent DB2 V9.5 release IBM has added early pureXML™ support to `db2advis`.

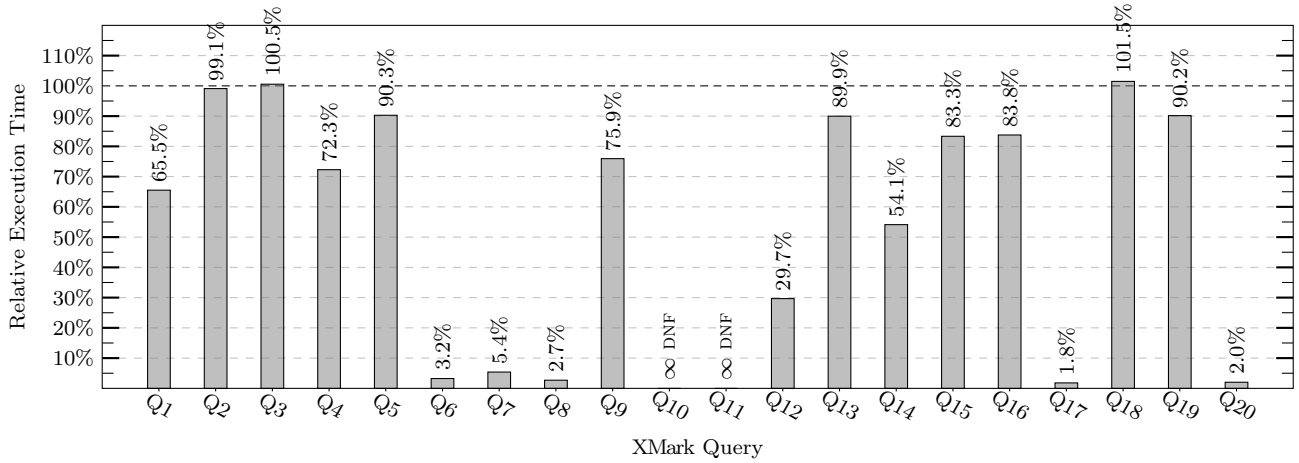


Table 5: Impact of statistical views on the XQuery evaluation performance for the XMark benchmark query set. Without statistical views, Queries Q10 and Q11 (marked with ∞ DNF) did not finish evaluation in acceptable time; with statistical views these queries executed in 4.8 h and 4 min, respectively.

```
<OPTGUIDELINES>
  <NLJOIN>
    <NLJOIN>
      <ACCESS TABLE='a4'/>
      <IXSCAN TABLE='a3' INDEX='gp'/>
    </NLJOIN>
    <IXSCAN TABLE='a2' INDEX='kpsv'/>
  </NLJOIN>
</OPTGUIDELINES>
```

With this optimization guideline in place, DB2 generates the query execution plan of Figure 4(b), even if the statistical view `docaccess` has not been defined.

Extensive use of optimization profiles can turn the DB2 query processor into a programmable relational algebra engine. While we do not plan to follow this route and instead focus on improvements in *Pathfinder*'s SQL code generator and proper XML document statistics, scenarios in which *Pathfinder*-generated optimization profiles lead to even better execution plans are conceivable.

3. BEYOND DB2—BEYOND XQUERY

With the advent of *Pathfinder*'s SQL code generator, there is now little doubt that DB2's SQL query processor can be turned into an XQuery processor that need not stumble once XML input sizes grow large (beyond 100 MB and towards 1 GB, say). The DB2 kernel infrastructure has already repeatedly proven its versatility and can process tree-shaped non-uniform XML data efficiently, given that the system has learned about the input documents' structure.

While this Ph.D. project will further study how we can compensate for DB2's lack of tree-awareness, we will try to significantly broaden the scope of this work. In one dimension, this entails code generation for RDBMS backends other than DB2—*Pathfinder*'s internal algebraic primitives are sufficiently simple and generic to drive a number of database systems (among these, currently, CWI's column store MonetDB, kdb+ from Kx Systems, and Microsoft® SQL Server). In a second dimension, there are few details of the *Pathfinder* code generation process that are *truly* XQuery-specific. We plan to embrace other "nested-loop languages", *e.g.*, LINQ or fragments of RUBY, and thus turn database systems into efficient and scalable runtime systems for these languages.

Acknowledgments. I would like to thank my Ph.D. advisor Torsten Grust and Jan Rittinger for their insightful feedback.

4. REFERENCES

[1] R. Bayer and K. Unterauer. Prefix B-Trees. *ACM TODS*, 1977.

[2] S. Boag, D. Chamberlin, and M. Fernández. XQuery 1.0: An XML Query Language. W3 Consortium, 2007. `http://www.w3.org/TR/xquery/`.

[3] DB2 9 for Linux, UNIX and Windows Manuals, 2007. `http://www.ibm.com/software/data/db2/udb/`.

[4] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *Proc. VLDB*, August 1997.

[5] G. Graefe. Sorting and Indexing with Partitioned B-Trees. In *Proc. CIDR*, January 2003.

[6] T. Grust, M. Mayr, J. Rittinger, J. Teubner, and S. Sakr. A SQL:1999 Code Generator for the Pathfinder XQuery Compiler (Demo Paper). In *Proc. SIGMOD*, June 2007.

[7] T. Grust, J. Rittinger, and J. Teubner. Why Off-The-Shelf RDBMS are Better at XPath Than You Might Expect. In *Proc. SIGMOD*, June 2007.

[8] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. VLDB*, September 2004.

[9] Y. E. Ionnadis. The History of Histograms (abridged). In *Proc. VLDB*, August 2003.

[10] J. Melton and A. R. Simon. *SQL:1999 – Understanding Relational Language Components*. Morgan Kaufmann, 2002.

[11] A. R. Schmidt, F. Waas, M. L. Kersten, I. Manolescu, M. J. Carey, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. VLDB*, August 2002.