A Query Optimization Assistant for XPath

Haris Georgiadis
AUEB
harisgeo@aueb.gr

Minas Charalambidis
AUEB
minchar86@gmail.com

Vasilis Vassalos
AUEB
vassalos@aueb.gr

ABSTRACT

We demonstrate a generic and extensible cost-based optimization and execution system for XPath queries, named GeCOEX, using a comprehensive suite of query analyzing and administrative tools, named QuOAX. GeCOEX supports many different physical operator implementations and XML storage engines and is agnostic to the underlying physical data model. Its optimizer is the first generic cost-based optimizer for XPath queries that *always* picks the *cheapest* estimated plan, among a very large number of possible plans, for a wide range of XPath queries and different datasets in a *very small fraction* of the time required for efficient execution. The QuOAX suite provides administration tools that allow the user to add new – or deactivate already deployed – physical operator implementations, physical operator cost models and rewriting rules and also to make use of different XML storage and XML statistics estimators. QuOAX also provides query plan analysis and visualization tools that allow users to visualize the physical plan chosen by the optimizer or all possible generated physical plans for a given query and to execute any of those plans. QuOAX helps users to i) easily test new XPath processing techniques, comparing them directly with existing ones and identifying the situations to which they show promise, ii) improve the effectiveness of the optimizer and iii) find out the appropriate access methods or indices that are beneficial for a specific workload.

Categories and Subject Descriptors: H.2.1 **[Information Systems]**: Physical Design: *Access methods,* H.2.3 **[Information Systems]**: Systems: *Query processing,* D.2.9

General Terms: Performance, Experimentation

Keywords: XPath, XML, Cost-based Optimization, Algebraic rewritings, Cost Models

1. INTRODUCTION AND MOTIVATION

There has been a lot of research in the area of XML query processing [1][2][4][3][6][5][7]. Many of the proposed techniques have proven to be very good for specific query and data set characteristics but are often intertwined with the existence of specific auxiliary data structures [3][7] and XML encodings [3][6]. These characteristics, together with the coarse granularity of many of these techniques, make it hard to take full advantage of their benefits for more complex querying tasks or with arbitrary databases: in such scenarios, it is either not immediately clear which technique would perform best, and we need a framework for predictive comparison and evaluation of each technique, or it would be best to combine techniques. In other words, what is needed is a query optimization and execution system for XPath, together with powerful administrator or power user tools to control and visualize the process.

We will demonstrate a powerful Generic and Extensible Cost-based Optimization and Execution system for XPath (GeCOEX), complete with a suite of graphical tools (Query Optimization Assistant for XPath - QuOAX) that both allow an inside look into the workings of the optimizer and query executor and allow an administrator or power user to easily tune and extend the entire system. The system architecture is shown in Figure 1 and is discussed in the next section.

The GeCOEX system is based on the framework presented in [8] and [9]. GeCOEX uses a logical XPath algebra and a set of rewriting rules that together can algebraically capture many XPath processing strategies. The core of the optimizer is a cost-based plan selection algorithm for XPath queries, named PSA. The optimizer is independent from the underlying physical data model and storage system and the available logical operator implementations, depending only on the implementation of a set of APIs. We implemented and will demonstrate different implementations of these APIs, including access methods and statistics estimators as well as a large pool of physical operator implementations. Some of the implementations correspond to well-known XPath processing techniques and algorithms from the literature while others are novel, presented and evaluated in [9].

The cost-based optimizer of GeCOEX *always* picks the *cheapest* estimated plan, among a very large number of possible plans, for a wide range of XPath queries and different datasets in a *very small fraction* of the time required for efficient execution. Experimental evaluation of the effectiveness of the overall GeCOEX system has shown that the execution time of the chosen plan is within *12%* of the optimal execution time in *all but one* of the queries in the tested workloads [8].

QuOAX allows the user to easily deploy and/or parameterize the components of GeCOEX. The user can visualize query plans (either the one suggested by the optimizer or the set of all possible generated plans) for a given query in a comprehensive way. A query plan is represented as a tree that shows the user the logical operators composing the plan and their cardinality estimations, the specific physical operators for each one of them along with their cost estimations and, finally, the total cost and cardinality estimations for the entire physical plan. The user can choose to execute one or many physical plans and see the execution time for each one of them, or "edit" a plan, replacing an operator implementation with a different one from the pool of available operators.

The capabilities of the QuOAX suite can help the user test existing or new XML *processing* or *storage/encoding techniques* and strategies, since it is easy to:

- compare the performance of different XPath processing techniques (represented by different physical operator implementations) over different XML storage systems
- evaluate the impact of implementing a different XML access method (possibly based on new auxiliary structures and indices or on more efficient algorithms), as a result of extending the underlying XML storage system or deploying a completely different one.
- estimate the impact of rewriting rules (by altering or extending the pool of available rewriting rules)

The user can also use the QuOAX suite in order to increase the quality and the effectiveness of the GeCOEX *optimizer*. In particular, the user can use a query workload and measure the effectiveness of the optimizer by comparing the execution times of the query plans selected by the optimizer with those of the actual best plans (after executing all plans that can be generated from each query, a functionality provided automatically by QuOAX). This way the user can decide whether modifications on specific components improve the effectiveness of the optimizer and, as a result, the efficiency of the entire system. These components are: XML statistics estimators, cost models of the deployed physical operator implementations and cost models of the primitive access method implementations for a specific XML storage engine (as shown in Figure 1).

Finally, the user can use QuOAX in order to decide whether the creation of a specific index (for example a value or path index) would be beneficial for a given query, without having to actually create the index. The user can define indexes and see whether the optimizer makes a different choice and, if so, what is the estimated cost reduction.

2. ARCHITECTURE

GeCOEX system consists of three basic components: the *Query Parser*, the *Physical Plan Selector or Executor* and the *Physical Plan Executor*, as illustrated in Figure 1. Independence from the XML Storage System implementation is achieved via the *XPA API*. An input XPath expression is parsed by the query parser, which generates a logical plan as its algebraic representation in XPAlgebra [8], our navigation-based XPath algebra.

Using this initial logical plan, the Physical Plan Selector generates the best physical plan using an efficient plan selection algorithm called PSA [8], or the Physical Plan Generator generates all possible physical plans (based on a naïve algorithm called GAPH). Both PSA and GAPH make use of the available *Rewriting Rules* for logical transformations, access all needed statistics from the *Database Statistics* interface of the XPA and retrieve the costs of physical operators from their *Descriptors.*

The *Query Execution Subsystem* can support multiple XPath processing and XML storage methods. In order to use a different XML Storage System, one only has to provide the implementation of an XPA driver. This is due to the fact that Physical Operators do not have direct access to the underlying XML Storage System. Instead, they make use of a series of primitive access methods (abrv. *PAMs*), available through the *Primitive Access Methods interface* of the XPA API. The cost model provided by a physical operator *Descriptor* relies on the cost models of any *PAM* calls made by the respective operator. These are available to the *Descriptor* through the above mentioned XPA API, which stands as an abstraction layer between the XPA Driver used and the rest of the system. Using a new Storage System requires implementing the *PAMs* and defining their cost models in an XPA driver.

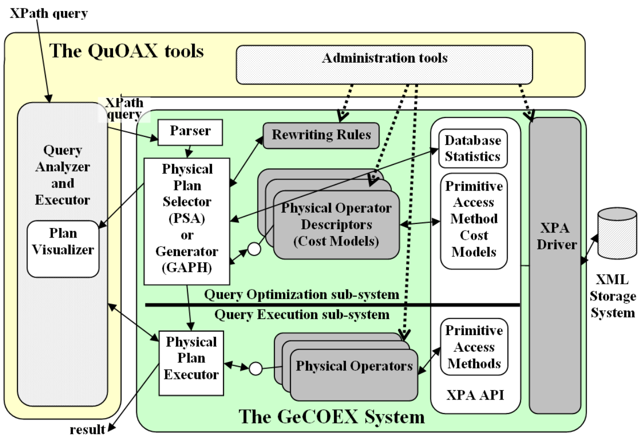


Figure 1 System Architecture

XPalgebra and Rewriting Rules: XPAlgebra is our generic sequence-based logical algebra for a large subset of XPath that includes forward and backward axes and non-positional predicates involving conjunctive boolean expressions that don't involve comparisons between paths. XPAlgebra follows XPath 2.0 semantics. XPalgebra operators return either a sequence of nodes (*sequence operators*) or a boolean value (*boolean operators*). We can think of sequence operators as capturing navigation steps of the main navigation of an XPath expression. Boolean operators capture predicate clauses. An XPAlgebra expression can be represented graphically as a tree read bottom-up. Sequence operators are linked with simple lines, lines crossed by arrows show attached boolean operators to *filter* or to other boolean operators. There is a straightforward algorithm for translating an XPath expression to its algebraic correspondence in XPalgebra which can be considered as a logical query plan. More information about XPalgebra can be found in [8].

Figure 2 (a) illustrates the graphical representation of the XPAlgebra expression that corresponds to the XPath query '/s/r/*/it[mb/m/to]//k' which is read bottom-up. The forward path (*fp*) operator $fp_{/s/r/*/it}$ takes as input the root singleton and returns all 'it' elements under '/s/r/*/it'. The filter operator *f* corresponds to the predicate of the XPath query. It keeps only those 'it' elements for which there is at least one 'to' descendant under relative path '/mb/m/to' (thus, the boolean forward path operator $Ъfp_{/mb/m/to}$ must return *true*), whose text node equals 'x' (thus, the boolean value filter operator $Ъvf_{text()='x'}$ also must return *true*).

To algebraically capture the large variety of plans, GeCOEX adopts a rewriting-based approach. We have defined a comprehensive set of rewritings that can produce, for each XPath query, a large set of logical plans capturing virtually all the important processing strategies for XPath at the logical level [8]. The plan illustrated in Figure 2 (b) derives from the one in Figure 2(a) by sequentially applying a series of rewriting rules. According to that plan (b), we can first evaluate all 'k' elements under '/s/r/*/it//k' (forward path operator $fp_{/s/r/*/it//k}$) and then filter them (filter operator *f*) keeping only those that have at least an 'it' ancestor (boolean ancestor operator $\text{Ъ}a_{it}$) which, in turn, has at least one 'to' descendant under relative path '/mb/m/to' (boolean forward parth $\text{Ъ}fp_{/mb/m/to}$) with a text node equaling 'x' (boolean value filter operator $\text{Ъ}vf_{text()='x'}$).

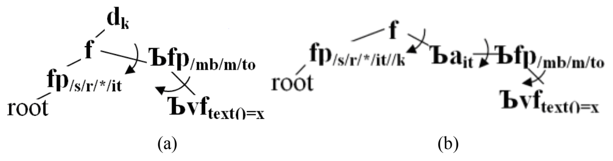


Figure 2. Logical Plans

Physical Operators: The GeCOEX system can capture a large variety of existing XPath query processing techniques. A total of 42 physical operators have been implemented, divided roughly into four 'families', each based on a proposed XPath processing technique: Sort-Merge-based [8] (inspired by the Sort-Merge join algorithm), Staircase Join [1], Lookups [8] (inspired by PPF-based XPath processing [3]), and PathStack [2]. The performance advantages of the Lookups and Sort-Merge-based families of operators compared to existing techniques have been experimentally evaluated in [9]. We will demonstrate use of all physical operators in query plans and their efficiency differences.

XPA drivers/XML Storage Systems: The GeCOEX system can work with any storage engine that implements the *XPA API*. We have developed five different versions of such a native XML storage system. In all versions, XML elements are stored in B-Tree structures. B-Trees indexes can be built on specific text or attribute nodes (*value-indices*). The implementation of *DBStatistics* uses the (stored) cardinalities of root-to-node paths (*RTN-path*) and statistics regarding attribute and text node values. The implemented storage systems differ in the labelling scheme used, in the inclusion or not of a root-to-node path index and in whether they keep a separate B-Tree per tag name [8].

The Query Oprimization Assistant (QuOAX): Figure 1 shows the two basic components of QuOAX, the *Administration tools* and the *Query Analyzer and Executor*. The user interacts with the former in order to configure GeCOEX by deploying an XPA driver corresponding to a specific XML storage system, by deploying or deactivating Physical Operators (and their Descriptors) or by altering the set of available rewriting rules. The user interacts with the Query Analyzer and Executor in order to feed the system with one or more XPath queries and see, either the best physical plans (as selected by the *Physical Plan Selector*) or all possible physical plans (as generated by the *Physical Plan Generator*), thanks to the *Plan Visualizer*. The user can chose one or more of these plans for execution. Together with the result, information such as execution times and cardinalities as well as cost and cardinality estimations is returned.

The XML storage systems along with their corresponding XPA drivers, the entire GeCOEX system and all the physical operators, are already implemented (in Java). Berkeley DB Java Edition has been used as B-Tree implementations in the XML storage systems. The QuOAX is implemented as a Java Swing application.

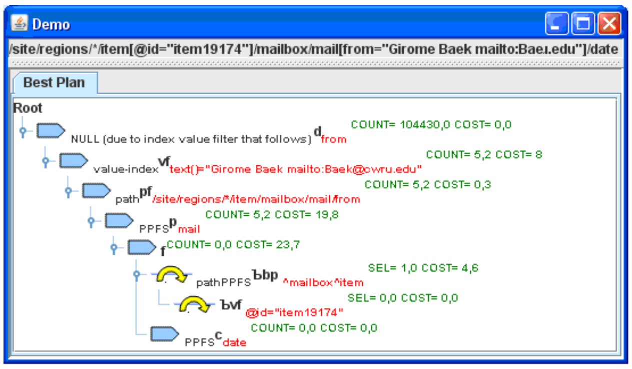
3. DEMONSTRATION SCENARIO

We will demonstrate the GeCOEX system and QuOAX tools with a variety of real and synthetic large XML documents (hundreds of MB to a few GB), using some of the five XML storage systems, and queries relevant to each document.

At first, we will use the QuOAX Administrative tools to configure the GeCOEX System. We will deploy GeCOEX in real time on top of a particular storage system by registering the appropriate XPA driver. Using a user-friendly window-based wizard we will continue with deploying all the 42 available physical operators.

Next, we will use the QuOAX query analyzer and executor tool and analyze and run an XPath query from our workload. We select first to use the Physical Plan Selector (PSA algorithm) to find the (estimated as) cheapest physical plan (Figure 3). Demo users will be able to see details about the chosen plan. The graphical representation of the plan is top-down starting from root, as shown in Figure 2. Blue pentagons are used for sequence operators (performing basic XPath navigation) and yellow arrows for boolean operators (corresponding to predicate clauses). A user will be able to execute a plan tree and be informed about the elapsed time, the number of result nodes as well as cost and cardinality estimations. The QuOAX query analyzer allows for easy comparison of different physical operators that implement the same logical operator. In the demonstration, we will select two different physical operators for a specific operator of a plan and we will re-execute the query plan. The plan will be executed twice, one for each of the selected physical operators, so we will be able to compare the performance of the specific operators by comparing the elapsed times.

In the demonstration we will also use the QuOAX administration tools to declare a specific value index (without actually building it). Giving the same query to the query analyzer, we will see whether a new physical plan is selected. If so, we will build the index and execute the selected plan to see the actual performance gain.



```
/site/regions/*/item[@id="item19174"]/mailbox/mail[from="Girome Baek mailto:Baeı.edu"]/date
Root
  NULL (due to index value filter that follows) d from        COUNT= 104430,0 COST= 0,0
    value-index vf text()="Girome Baek mailto:Baek@cwru.edu"  COUNT= 5,2 COST= 8
      path pf /site/regions/*/item/mailbox/mail/from          COUNT= 5,2 COST= 0,3
        PPFS p mail                                           COUNT= 5,2 COST= 19,8
          f                                                   COUNT= 0,0 COST= 23,7
            pathPPFS Ъbp ^mailbox^item                        SEL= 1,0 COST= 4,6
              Ъvf @id="item19174"                             SEL= 0,0 COST= 0,0
            PPFS c date                                       COUNT= 0,0 COST= 0,0
```

Figure 3. The estimated as cheapest physical plan

Going back to the beginning of query execution, we will follow the option of selecting the Physical Plan Generator in order to see all physical plans, generated by the GAPH algorithm: plans will appear grouped by the logical plan they belong to. As illustrated in Figure 5, for each distinct logical plan, a separate top-level tab is created containing all corresponding physical plans in separate sub-tabs. For each physical plan, the total cost and cardinality estimations are shown in the right-text area. From this point during the demonstration we will choose to execute a specific physical plan or all physical plans. In the latter case, a spreadsheet will be printed (and also appear in a separate window) with the following columns: *Plan id*, *Estimated Cost*, *Execution time1*, *Execution time2*, *Estimated Cardinality*, and *Count*. The *Plan id* is the unique name of its plan. For example plan 26.38 is the 38th physical plan that derives from the 26th logical plan. The

Estimated Cost is the cost estimation of the respective plan. *Execution time1* and *Execution time2* are the execution times of the respective plan in first and second time, respectively. The *Estimated Cardinality* and the *Count* are the cardinality estimation and actual cardinality (number of nodes actually returned) for the specific plan, respectively. The spreadsheet is very useful for evaluating the effectiveness of the PSA algorithm as well as of our end-to-end cost-based optimization and execution engine.

The effectiveness of PSA depends on whether it picks the cheapest estimated plan in the large plan space defined by the rewriting rules and the physical operators. To evaluate effectiveness we will compare the cost estimation of the estimated as cheapest plan with the cost estimate of the PSA-selected plan. In our experimental evaluation, described thoroughly in [8], for all queries in our query sets, and all datasets that we used, PSA picked the plan with the lowest estimated cost.

On measuring the effectiveness of GeCOEX, our end-to-end cost-based optimization and execution engine (including the robustness and precision of cost models and statistics estimation algorithms for access methods and physical operators), we will use the spreadsheet produced by QuOAX query analyzer and compare the following a) the average execution time of the plan selected by PSA b) the average execution time of the best plan c) the average execution time of the best plan among those corresponding to the *default* logical plan, and the execution time of the worst plan. Figure 4 illustrates a graph that summarizes the results of such an experiment; a total of 16 queries have been used on a 570MB dataset from the XMark benchmark [10]. In all our experiments, for the vast majority (over 80%) of queries PSA chooses a plan whose cost is less than 5% above the cost of the actual best plan [8]. When the *DBStatistics* implementation fails to give precise estimates due to inaccurate statistics (as in Q18), PSA makes less good choices.

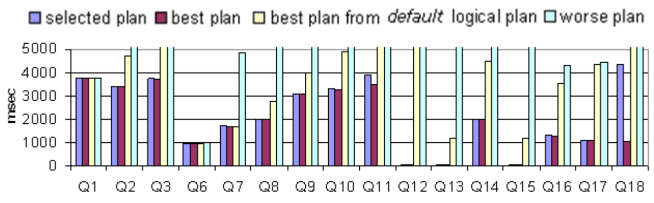
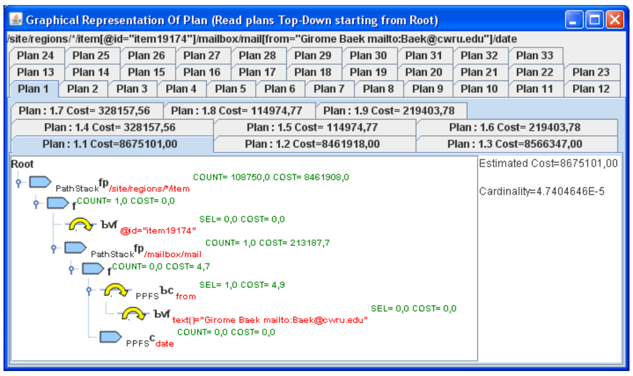


Figure 4. Evaluating the effectiveness of GECOEX

In the demonstration we will also be modifying system properties using the administrative tools to discuss with the audience the impact of these changes in plan generation.


Graphical Representation Of Plan (Read plans Top-Down starting from Root)

/site/regions/*/item[@id="item19174"]/mailbox/mail[from="Girome Baek mailto:Baek@cwru.edu"]/date

Plan 24 | Plan 25 | Plan 26 | Plan 27 | Plan 28 | Plan 29 | Plan 30 | Plan 31 | Plan 32 | Plan 33
Plan 13 | Plan 14 | Plan 15 | Plan 16 | Plan 17 | Plan 18 | Plan 19 | Plan 20 | Plan 21 | Plan 22 | Plan 23
Plan 1 | Plan 2 | Plan 3 | Plan 4 | Plan 5 | Plan 6 | Plan 7 | Plan 8 | Plan 9 | Plan 10 | Plan 11 | Plan 12

Plan : 1.7 Cost= 328157,56 | Plan : 1.8 Cost= 114974,77 | Plan : 1.9 Cost= 219403,78
Plan : 1.4 Cost= 328157,56 | Plan : 1.5 Cost= 114974,77 | Plan : 1.6 Cost= 219403,78
Plan : 1.1 Cost=8675101,00 | Plan : 1.2 Cost=8461918,00 | Plan : 1.3 Cost=8566347,00

```
Root
  PathStack fp /site/regions/*/item          COUNT= 108750,0 COST= 8461908,0
    f COUNT= 1,0 COST= 0,0
      Ъvf @id="item19174"                    SEL= 0,0 COST= 0,0
      PathStack fp /mailbox/mail             COUNT= 1,0 COST= 213187,7
        f COUNT= 0,0 COST= 4,7
          PPFS Ъc from                       SEL= 1,0 COST= 4,9
            Ъvf text()="Girome Baek mailto:Baek@cwru.edu"   SEL= 0,0 COST= 0,0
          PPFS c date                        COUNT= 0,0 COST= 0,0
```

Estimated Cost=8675101,00

Cardinality=4.7404646E-5

Figure 5. All physical plans generated by GAPH

Altering the pool of physical operators: We will deactivate a family of physical operators in order to see how the lack of those operators affects the optimizer's plan selection. We will show that by altering the pool of physical operators may cause the optimizer to select physical plans of completely different navigational structure and that PSA correctly generates the best plans. Figure 6 illustrates the results of such an experiment; we executed a typical XPath query (of the form p1[p2]p3[p4]p5, where p1-p5: forward paths) first with our default pool of physical operators, and second after removing the Lookup (LU) family from the pool. PSA correctly generates the best plans in both cases, which happen to have completely different navigational structures, shown in Figure 6 as $L^{default\text{-}pool}$ and $L^{without\text{-}LU}$, respectively. For both pools of operators, the GECOEX optimizer selected a plan that is less than 4% above the cost of the best plan.

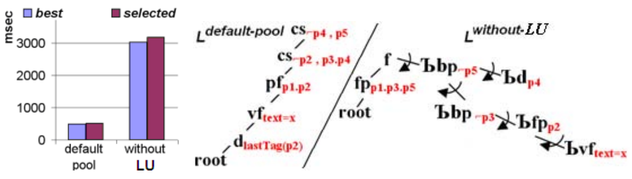


Figure 6. Gain from using plan suggested for each pool

We will also demonstrate how a new physical operator should be implemented and how can be deployed to the system.

Altering the set of rewriting rules: We will use the administrative tools to deactivate a couple of rewriting rules. We will then return to the query analyser to see whether the lack of those rewriting rules will affect optimiser's decision for a query.

Deploying other XML storage systems: We also plan to show how one can build an XPA driver for a specific XML storage system (which methods of the XPA API must be implemented) and demonstrate how the new driver can be deployed using QuOAX. We will then deploy a different XPA driver and, returning to the query analyser, show that the different XML storage system (thus, different *PAM* implementations and, in turn, different cost models for these *PAMs*) can lead the optimizer to select completely different physical plans.

REFERENCES

[1] T. Grust, M. van Keulen, J. Teubner: Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. VLDB 2003

[2] N. Bruno , N. Koudas , D. Srivastava, Holistic twig joins, SIGMOD Conference, 2002

[3] H. Georgiadis, V. Vassalos: Xpath on steroids: exploiting relational engines for XPath performance. SIGMOD Conference 2007

[4] K. S. Beyer, R. Cochrane, et al: System RX: One Part Relational, One Part XML. SIGMOD Conference 2005

[5] S. Paparizos, S. Al-Khalifa et al.: TIMBER: A Native System for Querying XML. SIGMOD Conference 2003.

[6] J. Lu, Tok W. Ling, C. Y. Chan, T. Chen: From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. VLDB 2005

[7] H. Jiang, H. Lu, W. Wang, B. C. Ooi: XR-Tree: Indexing XML Data for Efficient Structural Joins. ICDE 2003

[8] H. Georgiadis, M. Charalambides, V. Vassalos: Cost based plan selection for xpath. SIGMOD Conference 2009

[9] H. Georgiadis, M. Charalambides, V. Vassalos: Efficient physical operators for cost-based XPath execution. EDBT 2010

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