Bulletin of the Technical Committee on

Data Engineering

December 2008 Vol. 31 No. 4

IEEE Computer Society

Letters

The TC on Data Engineering
Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TC on Data Engineering web page is
`http://tab.computer.org/tcde/index.html`.

The Data Engineering Bulletin
The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at
`http://tab.computer.org/tcde/bull_about.html`.

Letter from the Editor-in-Chief

International Conference on Data Engineering

ICDE (the International Conference on Data Engineering) is the flagship database conference of the IEEE. The 2009 ICDE will be held in Shanghai, China at the end of March. I would encourage readers to check the ”Call for Participation” on the back inside cover of this issue of the Bulletin for more details. ICDE has become not only one of the best database conferences, but one of the largest as well. I attend this conference every year and always find my time well spent. Not only is the research program first-rate, but there is an industrial program, demos, and workshops as well.

Self-managing Database Systems

The current issue contains, starting on page two, a report from the Working Group on Self-Managing Database Systems of the Technical Committee on Data Engineering. This is the sole working group of the TCDE. The report is on the group's third workshop, which has been held in concert with ICDE for the past three years, most recently at ICDE in Cancun, Mexico. The area of self-managing databases is not only important but also a very active area of research. The result is that the working group's workshops have seen high quality papers and active participation. The report is well worth reading.

The Current Issue

The database community has witnessed, over the past several years, a substantial number of papers exploring query processing as it applies to the XML data model. There continues to be research in this area and research papers, though they are no longer in the same numbers as earlier. This suggests that XML technology is maturing. So where has this left us?

The current issue explores the still very active XML and XQuery technological evolution not only in the research community but also in the commercial world. Indeed, both in the XQuery standards area and in the deployment commercially of support for it, XML technology is increasingly pervasive. The technology being deployed, while originally fairly limited, is now quite robust and general. The issue is the joint work of Vassilis Tsotras, one of my appointed editors, and Mike Carey. (Mike is doing an encore with this issue as I remember well when he was a Bulletin editor in 1987 and I was a Bulletin author for an issue he organized.) Both Vassilis and Mike are deeply involved in the XML area. Their knowledge of the area, the ongoing work, and the people doing it, has enabled them to cover the XML field very comprehensively. If you want to know about research in the XML area, and what commercial products are capable of, you will want to read (and save for reference) this issue. Thanks to both Vassilis and Mike for their fine job on an important topic.

David Lomet
Microsoft Corporation

Report: 3rd Int'l Workshop on Self-Managing Database Systems (SMDB 2008)

Introduction

Information management systems are growing rapidly in scale and complexity, while skilled database administrators are becoming rarer and more expensive. Increasingly, the total cost of ownership of information management systems is dominated by the cost of people, rather than hardware or software costs. This economic dynamic dictates that information systems of the future be more automated and simpler to use, with most administration tasks transparent to the user.

Autonomic, or self-managing, systems provide a promising approach to achieving the goal of systems that are increasingly automated and easier to use. But how can that be achieved? The aim of this workshop was to present and discuss ideas toward achieving self-managing information systems in an intimate, informal, and interactive environment.

SMDB 2008 was the second workshop organized by the Workgroup on Self-Managing Database Systems (`http://db.uwaterloo.ca/tcde-smdb/`) of the IEEE Computer Society's Technical Committee on Data Engineering. The Workgroup, which was founded in October 2005, is intended to foster research that enables information management systems to manage themselves seamlessly, thereby reducing the cost of deployment and administration.

Workshop Overview

The workshop was conducted in Cancun, Mexico on April 7, 2008, prior to the start of the International Conference on Data Engineering. The workshop's program committee consisted of the members of the SMDB Workgroup's Executive Committee plus four other well-known researchers who are leaders in the area. In response to the Call for Papers, the program committee received 19 submissions. Each paper was reviewed by 3 program committee members. Six papers were accepted to the Workshop, resulting in an acceptance rate of 32%. In an effort to make the Workshop as inclusive as possible, 4 more submissions were accepted as poster papers and given a shorter presentation time at the end of the workshop, for an overall acceptance rate of 53%. This year, we added an invited keynote speaker and a panel session featuring four distinguished researchers to summarize and comment upon the Workshop's presentations and discussions. The average attendance at the Workshop throughout the day was 40 participants.

Technical Program

The technical program was organized into 4 sessions: Welcome and Keynote Talk, Self-Healing and Self-Optimization, Physical Design and Virtualization, Poster Papers, and Panel and Wrap-up. Due to a travel delay, the keynote speaker presented after the second and third sessions. Links to the slides presented for each talk can be found in "Workshop Program" under SMDB 2008 at the Workgroup's web page (`http://db.uwaterloo.ca/tcde-smdb/`).

The first session contained three papers on ways to enable self-healing and self-optimization of databases. Nehme [1] advocated a comprehensive approach to self-managing systems, in which all aspects of systems management, performance, risk assessment, and availability are described and managed as part of a unifying self-healing framework. Most modern DBMSs have hundreds of configuration parameters, so it's impossible to evaluate all combinations of possible values. Debnath et al. [2] devised a practical approach to determining good configurations by exploiting a Plackett and Burman methodology that ranks queries based upon how sensitive they were to the extrema of the factorial design of all configuration parameters. Yellin et al. [3] extended traditional control theory concepts of "flux" to automatically balance the load of processors performing a join that is partitioned among them, taking into account the cost to response time of changing the partitioning on the fly when the load on some of the processors is perturbed. By adding heuristics to limit the frequency of adaptation, they were able to reduce the number of specious changes caused by overly-adapting, as observed in the traditional “flux” technique.

The papers of the second session dealt with problems of physical database design and the increasing use of virtualization. Malik et al. [4] addressed the problem in the SkyQuery sky survey database of widely-varying ad hoc queries to tables having hundreds or even thousands of columns, few of which are referenced in any given query. Their solution was an adaptive, on-line vertical partitioning algorithm that improved upon an existing, off-line vertical partitioning algorithm (Autopart) and exploited some structure in the problem to prune the large solution space to a computationally tractable size. Minhas et al. [5] measured how much overhead the Xen hypervisor introduces when running a database (Postgres) workload. In a head-to-head comparison between a virtualized and “bare” operating system, the authors found significant overhead (tens of percent) introduced by virtualization when the buffer pool was warmed, but was much smaller (around 6%) when the buffer pool was cool. In fact, in some cases the I/O wait time was even lower with Xen, because pre-fetching in Xen’s Dom0 is better than that in Postgres! Tata et al. [6] argued that physical design advisors might better be located on clients, exploiting server-based advisors if available but also dealing with the common case that either such server-based tools are unavailable or the prerequisite information for running them might not be available, e.g., before the data is loaded. They suggested ways to glean useful physical design information from what limited schema, data, statistics, and/or workload is available when design decisions must sometimes be made.

After lunch, John Wilkes of HP presented his keynote talk, “Utility functions, prices, and negotiation”, which addressed the problem of designing in a principled way meaningful Service Level Objectives (SLOs), an important part of Service Level Agreements (SLAs). Wilkes described a technique that exploits the concept of utility functions, which measure some degree of “goodness” to those involved. In the 2-dimensional space of pricing vs. service outcome (e.g., throughput), utility indifference curves define contours of equal utility. The consumer chooses one such indifference curve, below which defines a “minimal acceptable utility” region. Similarly, the service provider chooses a (usually different) utility indifference curve, above which defines a region in which he is comfortable. Any intersection of the two regions defines an area within which negotiation on the SLO is possible. That negotiation, however, is much harder to characterize rationally or to quantify, because people sometimes react irrationally, are often averse to losses, and tend to overweight rare, extreme events.

The poster session contained the four poster papers with shorter presentations. Furtado et al. [7] described a prototype of a DBMS (based upon PostgreSQL) that uniformly reduced response times by up to 56% by continuously monitoring usage and adapting to meet quality of service (QoS) objectives. Voigt et al. [8] addressed the problem of off-line but dynamic physical database design, i.e., taking the order of arrival of queries into consideration. They modeled each query and a corresponding configuration (set of indexes) as a node in a state graph, which is huge but easily solved. To avoid the pitfall of over-fitting to a particular workload and the exact order of arrival, they simply limited the number of possible transitions. Sharaf et al. [9] described ASETS, a self-managing transaction scheduler that is formed as a hybrid between the optimal algorithm for low utilization and the optimal algorithm for high utilization. The combined algorithm uses an SLA to calculate a deadline for each transaction, then puts it on one of two ordered lists, depending upon how tardy it is or how much slack it has to make its deadline. Finally, Rizvi et al. [10] gave an overview of IBM’s Balanced Warehouse (virtual) appliance for Business Intelligence workloads, which is composed of Balanced Configuration Units (BCUs), each a pre-configured, pre-tested unit that can deliver the performance required and allow incremental growth, but runs on non-proprietary hardware.

The last session had a panel format, with four distinguished panelists: Surajit Chaudhuri of Microsoft Research, Guy Lohman of IBM Almaden Research Center, Ken Salem of the University of Waterloo, and our keynote speaker, John Wilkes of HP. Each tried to respond to five questions in light of the day’s presentations:

1. **Is completely self-managing achievable?** What are the biggest roadblocks to that, both technically and in gaining the trust of the user to enable the DBMS in “autopilot mode”? How do we avoid making life

worse for the administrator by adding more things that can go wrong? Is "less really more", i.e., is the only way to get simplified management by making things go away rather than have wizards set dials / thresholds?

2. **Can administration be standardized the way SQL querying has been standardized?** The success of relational DBMSs has been significantly helped by the standardization of SQL, but administration remains very different from one vendor to the next. Can / should administration be standardized somehow? Would this facilitate the emergence of client-side tools?

3. **How do we know when we've succeeded?** We're used to measuring performance, but how do we measure self-managing or ease of use? If we can measure it, how would we benchmark it? What aspects of self-managing can be realistically included in such a benchmark (i.e., is it possible to test automatic recovery from realistic failure modes)?

4. **How can self-managing tools function with incomplete information**, e.g. how can we initially configure a system without having a workload and/or database statistics? Are existing tools too sensitive to the workload, anyway? How do we reduce the overhead that these management tools and their information needs impose on the DBMS?

5. **Self-managing DBMS: who cares?** DBMS are not deployed in isolation. If self-managing DBMS are challenging, can we hope for a self-managing stack? If not, should we bother with self-managing DBMS? If so, how should DBMS fit in with end-to-end self-management?

Chaudhuri enumerated all the reasons why self-managing is so hard to solve (e.g., large search spaces of possible configurations, difficulty of diagnosing problems automatically, the limitations of query optimizers as modeling tools, ...), but also listed areas in which advances have been made, notably memory management, index selection, enabling "what if?" analysis, and establishing some fundamental principles. However, he warned that a unifying theory of self-managing was unlikely in the near term, and that progress would likely continue to be made incrementally on individual problems. He also noted that robustness of self-managing tools is extremely important to establish trust with users.

Lohman said that users certainly care about self-managing, but they don't trust features that aren't on by default, and the loss of trust due to a failure is hard to recover, quoting an actual incident with early automated Bay Area Rapid Transit trains. He was somewhat skeptical that complete self-managing was possible, due to the complexity (and hence brittleness) of our models, but clearly great progress is still being made. He cautioned that we too often rely on performance measures because they are familiar, rather than real measures of self-managing or ease of use, which have yet to be devised. Finally, standardization in the administration area remains elusive, because the data definition language (and its underlying storage model), unlike the query portions of SQL, were never standardized and hence have diverged. He concluded that we have succeeded to a degree, but are farther from our goal than we like to admit.

Salem emphasized that databases, while an important part of the problem, do not exist in a vacuum, but are part of a much larger ecosystem that includes hypervisors, operating systems, application servers, etc. What gets deployed are complete systems, not components, and these must be managed and tuned together as a system. The database is not the center of the universe.

Wilkes stressed the importance of trust, and the difficulty of earning it from humans, who aren't always rational. He noted that people are far better at dealing with exceptions and approximations than are machines, and systems can often ignore useful information. He felt that policies (rules) were not the answer, because there are too many of them that would need to be written. People will accept and trust automation when the benefits exceed the cost, and the worst case disasters are no worse than what would happen with a person in charge. Trust only comes from reassurance that the system will always "do the right thing", and only then will the human give up control. Be sure never to take away that control without the human's permission, explain your automated decisions, and be wary of machine learning, which can be prone to inconsistencies, he advised.

Summary

Once again, the Workshop on Self-Managing Database Systems was extremely successful. Not only was attendance a bit higher than the previous year – despite the lure of Cancun's beach! – but so was participation through probing questions and lively discussion. The high quality of the papers and the enthusiastic interaction in the workshop demonstrate the vitality of research in self-managing information management systems.

The Workgroup on Self-Managing Database Systems would like to thank the participants and the organizers of the Workshop. They encourage anyone interested in making systems easier to manage to participate in the 2009 Workshop on Self-Managing Database Systems, which will be part of the International Conference on Data Engineering in Shanghai, China next spring.

References

[1] Rimma V. Nehme: Database, heal thyself, Proceedings of ICDE Workshops (SMDB 2008), pp. 4-10, Cancun, Mexico, 2008.

[2] Biplob K. Debnath, David J. Lilja, Mohamed F. Mokbel: SARD: A statistical approach for ranking database tuning parameters, Proceedings of ICDE Workshops (SMDB 2008), pp. 11-18, Cancun, Mexico, 2008.

[3] Daniel M. Yellin, Jorge Buenabad Chávez, Norman W. Paton: Probabilistic adaptive load balancing for parallel queries, Proceedings of ICDE Workshops (SMDB 2008), pp. 19-26, Cancun, Mexico, 2008.

[4] Tanu Malik, Xiaodan Wang, Randal C. Burns, Debabrata Dash, Anastasia Ailamaki: Automated physical design in database caches, Proceedings of ICDE Workshops (SMDB 2008), pp. 27-34, Cancun, Mexico, 2008.

[5] Umar Farooq Minhas, Jitendra Yadav, Ashraf Aboulnaga, Kenneth Salem: Database systems on virtual machines: How much do you lose?, Proceedings of ICDE Workshops (SMDB 2008), pp. 35-41, Cancun, Mexico, 2008.

[6] Sandeep Tata, Lin Qiao, Guy M. Lohman: On common tools for databases - The case for a client-based index advisor, Proceedings of ICDE Workshops (SMDB 2008), pp. 42-49, Cancun, Mexico, 2008.

[7] João Pedro Costa, Pedro Furtado: Poster session: Towards a QoS-aware DBMS, Proceedings of ICDE Workshops (SMDB 2008), pp. 50-55, Cancun, Mexico, 2008.

[8] Hannes Voigt, Wolfgang Lehner, Kenneth Salem: Poster session: Constrained dynamic physical database design, Proceedings of ICDE Workshops (SMDB 2008), pp. 63-70, Cancun, Mexico, 2008.

[9] Mohamed A. Sharaf, Shenoda Guirguis, Alexandros Labrinidis, Kirk Pruhs, Panos K. Chrysanthis: Poster session: ASETS: A self-managing transaction scheduler, Proceedings of ICDE Workshops (SMDB 2008), pp. 56-62, Cancun, Mexico, 2008.

[10] Haider Rizvi, Joyce Coleman, Sam Lightstone: Poster session: Simplifying business intelligence with a hybrid appliance: the IBM balanced warehouse, Proceedings of ICDE Workshops (SMDB 2008), pp. 71-78, Cancun, Mexico, 2008.

Anastassia Ailamaki, Shivnath Babu, Pedro Furtado, Sam Lightstone, Guy Lohman, Pat Martin,
Vivek Narasayya, Glenn Pauley, Ken Salem, Kai-Uwe Sattler, Gerhard Weikum
EPFL, Duke, U.Coimbra, IBM, IBM, Microsoft, Sybase, U.Waterloo, Ilmenau U.Tech, MPI

Letter from the Special Issue Editors

The widespread use of XML for describing and exchanging data on the web, together with increasing quantities of XML data in the enterprise world, make it crucial to have an efficient query capability for XML data. In anticipation of an XML-rich world, the W3C XML Query Working Group has given us XQuery, a declarative query language designed specifically for XML. XQuery 1.0 became an official W3C standard (a Recommendation) in January 2007. Today, XQuery is gaining traction in industry; most major relational database systems, including products from IBM, Oracle, and Microsoft, now support an XML data type and include XQuery support for querying the contents of XML columns of tables. In middleware, leading enterprise service buses support both XQuery and XSLT for data transformation and routing, and several information integration products are based on XML and XQuery as well. XML, and to an extent XQuery, is also starting to penetrate key industry sectors (such as publishing, government, and pharmaceuticals) that have heavy technical document management requirements.

At the same time, XQuery itself is evolving. The W3C XML Query working group has several XQuery extension activities in progress. These include XQuery 1.1, extending the capabilities of XQuery with features such as grouping and aggregation; the XQuery Update Facility, adding XML update functionality to XQuery; the XQuery Full-Text Extension, adding content-based query capabilities to XQuery for text-heavy XML data; and, last but not least, the XQuery Scripting Facility, adding the ability to mix procedural-style control and side effects with the declarative query capabilities of XQuery in support of complex, XML-centric applications.

These developments make this a good time to take a look at the current state of the art in the XQuery processing world, both from academic and industrial perspectives. That is the purpose of this special issue of the *Data Engineering Bulletin*. This issue presents a snapshot of the current state of both the art and the practice of XQuery processing. Due to space limitations and busy potential contributors, the snapshot is of course incomplete, but we feel we have captured an interesting range of XQuery processors. When considering applications of XQuery, one finds a broad range of potential use cases that range from file processing and data transformation to message processing, to data integration, and to data and/or document storage, management, and querying. We have attempted to cover the full range in assembling this special issue.

The first two articles discuss systems (*Pathfinder* and *TIMBER*) developed in academia, using two very different approaches, one based on a relational implementation and one on native XML storage. The next three articles describe server-side XQuery processors. The articles on IBM's DB2 *pureXML* and Oracle's *XML DB* deal with XQuery in relational data servers, while the third examines XQuery in a native XML server (*MarkLogic Server*) aimed at content-oriented XML use cases. Next in this special issue are two articles about middleware XQuery processors, namely, the XQuery engine from BEA's *AquaLogic Data Services Platform* and the *DataDirect XQuery* engine, each of which use XQuery for information integration. The final article is about XQuery processing in *Saxon*, a leading open source XQuery engine.

We hope that this special issue will serve as a starting point for further academic and industrial contributions, as XQuery's increasing acceptance and ongoing evolution provide a fertile ground for interesting new research. We would like to thank the articles' authors, all experts in the field, for their timely efforts in assembling their excellent contributions for this special issue. Special thanks go to Marcos Vieira at UC Riverside for editorial assistance with the issue.

Michael J. Carey and Vassilis J. Tsotras
UC Irvine and UC Riverside

Pathfinder: XQuery Off the Relational Shelf

Torsten Grust   Jan Rittinger
Universität Tübingen, Germany
`torsten.grust@uni-tuebingen.de`
`jan.rittinger@uni-tuebingen.de`

Jens Teubner
ETH Zürich, Switzerland
`jens.teubner@inf.ethz.ch`

Abstract

The Pathfinder project makes inventive use of relational database technology—originally developed to process data of strictly tabular shape—to construct efficient database-supported XML and XQuery processors. Pathfinder targets database engines that implement a set-oriented mode of query execution: many off-the-shelf traditional database systems make for suitable XQuery runtime environments, but a number of off-beat storage back-ends fit that bill as well. While Pathfinder has been developed with a close eye on the XQuery semantics, some of the techniques that we will review here will be generally useful to evaluate XQuery-style iterative languages on database back-ends.

1 The Rectangularization of XQuery: Purely Relational XML Processing

If you zoom back in time to dig for the semantic roots of XQuery [5], you will find that the language's core construct, the `for–let–where–order by–return` (FLWOR) block is one particular incarnation of a very general idea: the *comprehension* [26]. Many language-related concepts may be uniformly understood in comprehension form, but comprehensions provide a particularly concise and elegant way to express iteration over collections of objects—in the case of XQuery: finite, ordered sequences of XML nodes and atomic values (or *items*) [1]. Any program or query expressed in comprehension form is subject to a number of useful equivalence-preserving rewriting rules (the *monad laws*) and so is XQuery's FLWOR block. Once you look closely, a wide range of seemingly XQuery-specific optimizations realized by compilers and interpreters today, *e.g.*, `for` loop fusion or unnesting, in fact put the monad laws to work.

The family of programming and query languages whose semantic core may be cast in comprehension form is large. Among its members, specifically, is SQL, *the* relational database language. This observation sparked a whole line of work that we will review in the following pages:

> Exploit the common semantic ground of XQuery and SQL and try to turn relational database systems (*i.e.*, processors for strictly tabular, or rectangular, data) into efficient and scalable XQuery processors.

XQuery processors of this type should be able to benefit from the 30+ years of research and engineering experience that shaped relational database technology. This is, in fact, what we repeatedly observed in the course of

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

the *Pathfinder* project (initiated in late 2001), an effort to construct a *Purely Relational XQuery Processor* [25]. As of today, *Pathfinder* can compile XQuery expressions into code (different variants of table algebras or SQL) ready for consumption by relational-style database back-ends. The back-ends evaluate this code against tabular encodings of XML instances and item sequences and thus *act like* XQuery processors. We have found the resulting systems to exhibit runtime performance characteristics that often surpass specifically-built "native" XQuery engines. On top of that, standing on the shoulders of relational giants provides stability, scalability, instant and wide availability as well as the seamless coexistence of XML instances and tabular data.

Input to this purely relational approach to XQuery processing are relational encodings of the XML data model, *i.e.*, ordered unranked trees of nodes of several kinds. We thus start our tour of the *Pathfinder* technology in 2 with a brief review of suitable tabular encodings of XML and then see how XPath location steps may be efficiently evaluated in terms of (self-)joins over the resulting tables. We turn to XQuery's dynamic semantics in 3 and sketch *loop lifting*, a compilation strategy that derives efficient set-oriented execution plans from nested FLWOR blocks. 4 shows how a purely relational account of the XQuery semantics can provide insights and optimization hooks that would be hard to find and formulate on the XQuery language level. Different kinds of database systems have already been turned into *Pathfinder* back-ends. 5 discusses selected systems and how they fare in their new role as XQuery runtime environments. Finally, as we said, the comprehension construct can explain aspects of a large family of languages: 6 sheds some light on how other recent programming and query language proposals with an iterative core could be "rectangularized"—and thus put on top of relational back-ends.

2 How Many Rows Does Your Tree Have? (Tabular XML Encoding)

The performance of any RDBMS-backed solution depends critically on how its data is represented in the relational format, tables of tuples. A purely relational XQuery processor is no different in this respect and the choice of a good relational *tree encoding* is an important factor to the functioning of a relational XQuery setup. Two principal features must be provided by the XML-to-tables translation, both dictated by the semantics of XQuery:

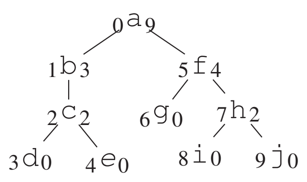
(i) It has to maintain XML *document order* and XQuery's concept of *node identity*. More explicitly, we expect the existence of a *surrogate* γ_v for each node v such that $\gamma_{v_1} = \gamma_{v_2}$ iff v_1 `is` v_2 and $\gamma_{v_1} < \gamma_{v_2}$ iff v_1 `<<` v_2.

(ii) Efficient mechanisms must exist that implement core operations on XML data. In particular, given a node surrogate γ_v, there must be a way to compute all surrogates for the node sequence $v/ax::nt$, where ax and nt are axis and node test of an XPath location step, respectively.

A variety of encodings has been published which provide both features, including ORDPATH [21], dynamic intervals [3], or XPath accelerator [14]. *Pathfinder* uses a variant of the latter, which we illustrate in a moment. As a drop-in replacement, the others could be plugged into *Pathfinder* seamlessly.

Pathfinder's relational XML storage, represents documents as a five-column table as shown in Figure 1 on the right for the XML instance

```
<a><b><c><d/>e</c></b>
<f>g<h><i/><j/></h></f></a>    .
```



(a) XML document tree.

pre	size	level	kind	prop
0	9	0	elem	a
1	3	1	elem	b
2	2	2	elem	c
3	0	3	elem	d
4	0	3	text	e
5	4	1	elem	f
⋮	⋮	⋮	⋮	⋮

(b) Relational tree encoding.

Figure 1: XML document tree, annotated with pre(·) and size(·) information (left/right), and resulting tree encoding.

For each node v, the table holds v's rank in a preorder tree traversal, $\mathsf{pre}(v)$, the number of descendants below v, $\mathsf{size}(v)$, its distance to the tree root, $\mathsf{level}(v)$, and the two columns kind and prop to represent XML information set characteristics of v, *i.e.*, its XML node type (one of elem, text, attr, $\ldots$) its tag name or typed value (depending on $\mathsf{kind}(v)$, refer to [14] for details). It is easy to see that $\mathsf{pre}(v)$ provides a suitable implementation for γ_v.

Evaluating XPath. Based on $\mathsf{pre}(\cdot)$, $\mathsf{size}(\cdot)$, and $\mathsf{level}(\cdot)$ properties, all twelve XPath axes can be characterized in a concise and machine-friendly manner. For axis `descendant`, *e.g.*, we have

$$\begin{array}{c} v' \in v/\mathtt{descendant} \\ \Leftrightarrow \\ \mathsf{pre}(v) < \mathsf{pre}(v') \leq \mathsf{pre}(v) + \mathsf{size}(v) \end{array} \quad . \tag{1}$$

Range expressions of this kind lend themselves to the use of *B-tree indexes* for efficient XPath evaluation. And, in fact, with appropriate index support, a relational XPath evaluation setup can outperform industry-strength "native" XML processors by significant margins [11].

The relational system plays its trump by organizing and indexing the relational tree encoding in the way that fits any given XPath query workload best. In [11], we found that *partitioned B-trees* form a particularly interesting class of indexes for XPath processing. The physical layout of a partitioned $\langle \mathsf{level}, \mathsf{pre} \rangle$ index, for instance, ideally matches the access pattern of an XPath `child` step. In addition, since a k-step XPath expression compiles into a k-way self-join at the relational end, the system's optimizer can gear the *order* of these joins to its liking [7]. This way, an *off-the-shelf* RDBMS solves formerly challenging problems in a purely mechanical way. This includes rewriting into forward-only plans [20] or top-down *vs.* bottom-up XPath evaluation [17].

Tree Awareness with Staircase Join. With a suitable tree encoding and the right selection of indexes, we enabled the relational system to act as an efficient tree processor, even though the system remained entirely unaware of the tree-structure that the encoded data originated from. Additional performance gains can be achieved by *injecting* such awareness into the RDBMS kernel.

Staircase join [13] is such an injection that can significantly improve relational XPath performance with only a local change to the RDBMS kernel. While evaluating an XPath location step, staircase join provides tree awareness by

- *pruning* nodes from the context set whose result nodes are already produced by other context nodes,
- *partitioning* the document space to (a) guarantee a duplicate-free result, sorted in document order and (b) achieve a strictly sequential, hence cache-efficient, access pattern to the underlying storage, and
- *skipping* parts of the document table which are early discovered (based on knowledge about the tree-origin) to not contain any result candidates.

Injecting staircase join into a main memory-oriented database system [13] or a traditional disk-based system [16] took only little changes to the systems' code. The change in runtime performance, however, was significant: we observed speed-ups of several orders of magnitude on both systems.

Types are Data. Other than traditional database query languages, XQuery blurs the distinction between data and its type. XML Schema types, *e.g.*, can be used as node tests in XPath location steps, just like tag names or node kinds. Likewise, the *runtime type* of arbitrary XQuery items can dynamically be inspected using the `instance of` and `typeswitch` constructs just like the item's value. A relational encoding for XQuery type annotations types, therefore, is called for.

The type system of XQuery, incidentally, has a structure that we already know how to deal with efficiently. All XML Schema types relate to each other in a *tree shape*. Pairs of pre and size values ("type ranks") are a

```
for $_ in (0) return
  for $x in (5,6,7) return
    if ($x mod 2 ne 0)
    then ("odd",xs:string($x))
    else "even"
```

(a) Sample XQuery FLWOR block.

iter	pos	item
1	1	5
1	2	6
1	3	7

(b) `(5,6,7)`

iter	pos	item
1	1	"odd"
1	2	"5"
2	1	"even"
3	1	"odd"
3	2	"7"

(c) $(\underbrace{\texttt{"odd","5"}},\underbrace{\texttt{"even"}},\underbrace{\texttt{"odd","7"}})$

Figure 2: An XQuery FLWOR block with the loop-lifted representations of its binding sequence and result. The gray outer pseudo loop establishes a single-iteration context for the top-level item sequence `(5,6,7)`.

logical way to account for that, with pre(·) as a concise implementation for type annotations. But the virtue of using type ranks is not only their support for tree navigation. As we showed in [23], type ranks enable interesting, database-style evaluation strategies for queries on types. *Type aggregation*, *e.g.*, can accelerate the processing of `instance of` or `typeswitch` clauses with sequence-valued input. Type-constrained XPath expressions can profit from relational indexes over columns that encode type information, or even from combined type/data indexes.

3 Drawing Independent Work from Lack of Side Effects

Pathfinder's main XQuery compilation strategy, dubbed *loop lifting*, revolves around the XQuery FLWOR block as the main language construct: *any* subexpression e is considered to be iteratively evaluated in the scope of its innermost enclosing `for` loop (if e is a top-level expression, we install a pseudo single-iteration loop `for $_ in (0) return` e such that variable `$_` does not occur free in e). In line with the comprehension notion, the FLWOR block

$$\texttt{for \$x in (}v_1\texttt{,}v_2\texttt{,}\dots\texttt{,}v_n\texttt{) return } e$$

describes the n-fold *side effect-free* evaluation of e under unmodifiable bindings of `$x` to items v_i. The result will be `(`$e[v_1/\$x]$`,` $e[v_2/\$x]$`, ..., `$e[v_n/\$x]$`)` (note that the resulting sequence will be flat according to the XQuery data model). Since the individual evaluations of loop body e cannot interfere, the system may perform the evaluation in any order or even in parallel [4, § 4.8.2]. This leads to a significant load of *independent work*, the principal source of set-orientation and potential parallelism in *Pathfinder*-generated query plans.

The "Great Invariant." To implement this idea on a relational back-end, *Pathfinder* compiles an XQuery subexpression e into an algebraic plan fragment that, at runtime, will yield a ternary table encoding *e's result for all its n iterated evaluations* [6, 12]. These tables uniformly adhere to the schema iter|pos|item in which a row $\langle i, p, v\rangle$ indicates that, in the ith iteration, the evaluation of e returned a sequence in which item v occurs at position p—in a sense, we obtain a fully unrolled representation of the result of e's enclosing `for` loop.

Consider the sample XQuery FLWOR block in Figure 2a. The top-level binding sequence `(5,6,7)` is evaluated once only while the inner `for` loop body undergoes three individual evaluations and thus contributes three subsequences (marked by $\underbrace{\quad}$ in Figure 2c) to the final result (to illustrate: row $\langle 3, 1, \texttt{"odd"}\rangle$ indicates that the third iteration contributes a sequence with item `"odd"` at position 1).

This "great invariant" drives the design of the whole compiler and enables a truly compositional style of translation from XQuery to relational algebra—prepared to cope with `for` loop nesting hierarchies of arbitrary depth. Loop-lifted algebraic plans diverge from the classical σ-π-$\bowtie$ pattern emitted by SQL compilers: instead, the plans exhibit a narrow "stacked" shape [7] reflecting the orthogonal expression nesting that is typical for a functional language like XQuery. Figure 3 sketches the plan shape for XMark Query Q8 [22] (each box denotes an algebraic operator, see 5). The resulting plans

(*i*) are truly set-oriented, *e.g.*, the algebraic plan for `$x mod 2` evaluates the subexpression for *all* bindings of `$x`, in some order the database back-end sees fit,

(*ii*) offer a range of effective optimization hooks (see 4), and

(*iii*) are sufficiently versatile to embrace a family of further iterative languages (6).

4 Relational Insights Into XQuery Affairs

The previous two sections invested considerable effort to "rectangularize" XQuery, press it into some shape that is digestible by a relational back-end. This section explores how we can benefit from a relational formulation of XQuery problems thanks to advanced and well-understood optimization techniques.

A particular example of such well-understood optimization techniques is the early dismissal of irrelevant information from the processing pipeline, also known as *selection* and *projection pushdown*. The latter idea, the disposal of columns not inspected by any upstream operator, has interesting consequences when applied to a loop-lifting compiler.

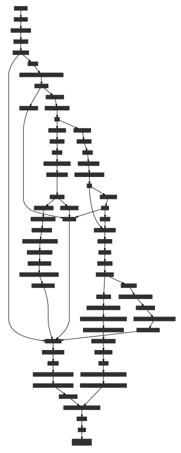


Figure 3: Plan shape model.

And Order is Data, Too. With *Pathfinder*, XQuery's various notions of order are encoded *in the data* (*i.e.*, the surrogates γ_v reflect document order, the columns iter and pos reflect iteration and sequence order): generated algebraic plans do *not* rely on some prescribed physical row order. Yet, the computation of encoded order information ultimately may still enforce such row order—and therefore incur a significant cost. With order made explicit on the data level, however, we now have a handle to *control* the dependence on ordered processing. By *projecting out* order-encoding columns, operators that were previously needed to ensure physical order will automatically be eliminated by *Pathfinder*'s optimizer.

In [10], we demonstrated how this effects in execution plans that can exploit opportunities to process (sub-) queries in an *unordered* fashion (*e.g.*, in the scope of XQuery's `unordered{}`), an opportunity that proved hard to discover by traditional query analyses on the level of XQuery. For XMark benchmark queries, *e.g.*, this led to a many-fold speed-up even when the dependence on order is not apparent in the source query.

Dependable Cardinality Forecasts for XQuery. Finding the most efficient execution plan for a given query often depends on the availability of accurate *result size estimates*. Though fairly well understood in the context of XPath, the problem of computing such estimates proved notoriously hard to solve for the complete XQuery language. The problem gets tangible once we look at it in the "rectangular" world. Relational equivalents for XQuery expressions provide the necessary fabric to connect existing work on XPath estimation with traditional relational techniques, such as the ones known from System R or different flavors of data statistics (*e.g.*, histograms) [24].

The outcome is a cardinality estimator for *arbitrary* XQuery (sub-)expressions whose accuracy we demonstrated for a wide range of different XQuery workloads [24]. And, most importantly, the estimator shows a high *robustness* with respect to intermediate estimation errors. Rather than piling up such errors during the estimation process, we found it often to recover gracefully and still come up with a meaningful estimate for the overall expression.

5 Off-the-Shelf and Off-Beat XQuery Runtime Environments

Loop-lifting turns the input XQuery expression into an algebraic plan solely operating at the table level. Plans are, generally, DAG-shaped (Figure 3) owing to a common sub-plan analysis stage installed in *Pathfinder*'s compilation pipeline. No traces of variable binding and reference, XPath traversal and node construction, explicit `for` iteration, conditionals (`if` or `typeswitch`), and similar source language features are left. This renders a wide range of set-oriented execution environments suitable runtime environments for XQuery, most of which have not originally been designed to act as XML processors.

Running XQuery on Off-the-Shelf SQL:1999 RDBMSs. The data-embedded order representation (4) makes off-the-shelf RDBMSs perfectly valid compilation targets and execution platforms for *Pathfinder*-generated code. A SQL:1999 code generator is, in fact, among the most advanced code generators available for *Pathfinder* today [8]. The compiler emits a no-frills table algebra dialect in which (groups of) operators have straight-forward SQL equivalents. Its row numbering operator ϱ, for example, has its direct correspondence in the SQL:1999 clause `ROW_NUMBER() OVER(PARTITION BY` $\cdots$ `ORDER BY` $\cdots$ `)`. *Pathfinder*'s SQL code generator implements a greedy template instantiation strategy—much like programming language compilers—that identifies plan sections whose semantics may be expressed in terms of a single SQL `SELECT-FROM-WHERE` block. The resulting SQL fragments are reasonably "good-natured", *e.g.*, all `UNION` operations are over disjoint tables, nested queries in `FROM` clauses are uncorrelated, and most occurring `JOIN` operations are equi-joins that implement the behavior of nested `for` iteration scopes.

The code generator introduces plan section boundaries,

(*i*) voluntarily, to share runtime evaluation effort if a (sub-)plan's output is input to more than one upstream branch in the plan DAG, or

(*ii*) by necessity, whenever the plan's stacked shape and the occurrence of a row numbering or duplicate removal operation does not allow to further grow the current SQL block.

The generated sequence of SQL code pieces are assembled into a *common table expression* (`WITH` $\cdots$) to jointly realize the semantics of the input XQuery expression on an off-the-shelf SQL back-end.

Lab experiments have shown how this approach turns SQL RDBMSs, hosting rectangularized XML instances as described in 2, into capable XQuery processors that do not stumble if document sizes get large [8, 11]. Quite the contrary: for queries against XMark instances beyond 100 MB size, we have seen IBM DB2 V9—running on loop-lifted SQL code—outperform its own built-in native XQuery processor *pureXML*$^{\text{TM}}$ [7].

Off-Beat XQuery Targets. If the underlying database back-end *does* operate over deterministically ordered tables, embedding order in the data appears wasteful: most perceivable implementations of ϱ lead to blocking sort operations in the final physical query execution plans. *Pathfinder*'s code generator for *MonetDB*, CWI Amsterdam's extensible database kernel tuned for in-memory operation [15], exploits explicit control over physical row order [2]. The narrow iter|pos|item tables that are pervasive in loop-lifted plans (3) prove to be a good match for the strictly column-oriented data and query model realized by *MonetDB*. The openness of the *MonetDB* kernel permits the injection of an implementation of staircase join that can particularly benefit from *MonetDB*'s ability to address rows, *i.e.*, encoded XML nodes, by document order rank [13]. *Pathfinder* plus *MonetDB* is distributed as *MonetDB/XQuery* [19]—a purely relational implementation of an XQuery compiler and runtime environment that can process Gigabyte-range XML instances in interactive time [2].

6 Compiling More Iterative Languages

Turning More Semantics into Data? The past few years with *Pathfinder* have taught us that RDBMSs can be turned into efficient processors for "alien" (*i.e.*, non-relational) languages if relevant pieces of the language's semantics are cast into data. To understand XQuery, in particular, we introduced relational representations of XML node identity and document order, XPath axes semantics, type annotations, sequence order, and nested `for` iteration.

```
concat [ ifodd xthen ["odd", show x]else["even"]|x < −[5, 6, 7] ]        (Haskell)
[5, 6, 7].collect {|x|x % 2! = 0? ["odd", x.to_s] : "even"}.flatten   (Ruby)
```

Figure 4: Haskell and Ruby equivalents of the XQuery FLWOR block of Figure 2a.

This recipe should be applicable to more languages, especially if their core iteration construct may be understood in terms of comprehensions and thus loop lifting. Comprehensions are indeed to be found, under varying coats of syntactic sugar, in a large family of languages. Among these are the programming languages Haskell and Ruby (Figure 4) [9], or Microsoft's LINQ [18]. A rectangularization of the relevant aspects of the language's semantics—*i.e.*, data types like ordered lists and dictionaries, or constructs like conditionals, variable assignment, and reference—plus loop lifting enables a *relational database engine to seamlessly participate in program evaluation*. Programmers continue to use their language's very own syntax, idioms, and functions—the system is in charge to decide *where* the computation described by a given program fragment will take place: on the heap or inside the relational database back-end. Programs that touch and move huge amounts of data, think *Computational Science*, will benefit the most from this support off the relational shelf.

References

[1] Scott Boag, Don Chamberlin, Mary Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language. W3C Recommendation, January 2007. `http://www.w3.org/TR/xquery/`.

[2] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. of the 25th ACM SIGMOD Int'l Conference on Management of Data*, Chicago, USA, June 2006.

[3] David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Özsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *Proc. of the 22nd ACM SIGMOD Int'l Conference on Management of Data*, pages 623–634, San Diego, CA, USA, June 2003.

[4] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kris Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. W3 Consortium, January 2007. `http://www.w3.org/TR/xquery-semantics/`.

[5] Peter Fankhauser, Mary Fernández, Ashok Malhotra, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 Formal Semantics. W3C Working Draft, June 2001. `http://www.w3.org/TR/2001/WD-query-semantics-20010607/`.

[6] Torsten Grust. Purely Relational FLWORs. In *Proc. of the 2nd ACM SIGMOD Int'l Workshop on XQuery Implementation, Experience and Perspectives (XIME-P)*, Baltimore, MD, USA, June 2005.

[7] Torsten Grust, Manuel Mayr, and Jan Rittinger. XQuery Join Graph Isolation. In *Proc. of the 25th Int'l Conference on Data Engineering (ICDE)*, Shanghai, China, March 2009.

[8] Torsten Grust, Manuel Mayr, Jan Rittinger, Sherif Sakr, and Jens Teubner. A SQL:1999 Code Generator for the Pathfinder XQuery Compiler. In *Proc. of the 26th ACM SIGMOD Int'l Conference on Management of Data*, Beijing, China, June 2007.

[9] Torsten Grust and Jan Rittinger. Jump Through Hoops to Grok the Loops. In *Proc. of the 5th Int'l ACM SIGMOD Workshop on XQuery Implementation, Experience, and Perspectives (XIME-P)*, Vancouver, Canada, June 2008.

[10] Torsten Grust, Jan Rittinger, and Jens Teubner. eXrQuy: Order Indifference in XQuery. In *Proc. of the 23rd Int'l Conference on Data Engineering (ICDE)*, Istanbul, Turkey, April 2007.

[11] Torsten Grust, Jan Rittinger, and Jens Teubner. Why Off-The-Shelf RDBMSs are Better at XPath Than You Might Expect. In *Proc. of the 26th ACM SIGMOD Int'l Conference on Management of Data*, Beijing, China, June 2007.

[12] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL Hosts. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, Toronto, Canada, September 2004.

[13] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. of the 29th Int'l Conference on Very Large Databases (VLDB)*, pages 524–535, Berlin, Germany, September 2003.

[14] Torsten Grust, Maurice van Keulen, and Jens Teubner. Accelerating XPath Evaluation in Any RDBMS. *ACM Transactions on Database Systems (TODS)*, 29(1):91–131, March 2004.

[15] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing Database Architecture for the New Bottleneck: Memory Access. *The VLDB Journal*, 9(3):231–246, December 2000.

[16] Sabine Mayer, Torsten Grust, Maurice van Keulen, and Jens Teubner. An Injection with Tree Awareness: Adding Staircase Join to PostgreSQL. In *Proc. of the 30th Int'l Conference on Very Large Databases (VLDB)*, pages 1305–1308, Toronto, Canada, September 2004.

[17] Jason McHugh and Jennifer Widom. Query Optimization for XML. In *Proc. of the 25th Int'l Conference on Very Large Databases (VLDB)*, pages 315–326, Edinburgh, Scotland, UK, September 1999.

[18] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Objects, Relations, and XML in the .NET Framework. In *Proc. of the 25th ACM SIGMOD Int'l Conference on Management of Data*, Chicago, USA, June 2006.

[19] MonetDB/XQuery. `http://www.monetdb-xquery.org/`.

[20] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking Forward. In *XML-Based Data Management and Multimedia Engineering, EDBT 2002 Workshops, Revised Papers*, pages 109–127, Prague, Czech Republic, March 2002.

[21] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. of the 23rd ACM SIGMOD Int'l Conference on Management of Data*, pages 903–908, Paris, France, June 2004.

[22] Albrecht Schmidt, Florian Waas, Martin Kersten, Mike Carey, Ioana Manolescu, and Ralph Busse. XMark: A Benchmark for XML Data Management. In *Proc. of the 28th Int'l Conference on Very Large Databases (VLDB)*, Hong Kong, China, 2002.

[23] Jens Teubner. Scalable XQuery Type Matching. In *Proc. 11th Int'l Conference on Extending Database Technology (EDBT)*, Nantes, France, April 2008.

[24] Jens Teubner, Torsten Grust, Sebastian Maneth, and Sherif Sakr. Dependable Cardinality Forecasts for XQuery. In *Proc. of the 34th Int'l Conference on Very Large Databases (VLDB)*, August 2008.

[25] The Pathfinder XQuery Compiler. `http://www.pathfinder-xquery.org/`.

[26] Philip Wadler. Comprehending Monads. In *Proc. of the 1990 ACM Conference on LISP and Functional Programming*, Nice, France, June 1990.

Querying XML in TIMBER

Yuqing Wu
Indiana University
yuqwu@indiana.edu

Stelios Paparizos
Microsoft Research
steliosp@microsoft.com

H. V. Jagadish
University of Michigan
jag@eecs.umich.edu

Abstract

In this paper, we describe the TIMBER *XML database system implemented at University of Michigan.* TIMBER *was one of the first native XML database systems, designed from the ground up to store and query semi-structured data. A distinctive principle of* TIMBER *is its algebraic underpinning. Central contributions of the* TIMBER *project include: (1) tree algebras that capture the structural nature of XML queries; (2) the stack-based family of algorithms to evaluate structural joins; (3) new rule-based query optimization techniques that take care of the heterogeneous nature of the intermediate results and take the schema information into consideration; (4) cost-based query optimization techniques and summary structures for result cardinality estimation; and (5) a family of structural indices for more efficient query evaluation. In this paper, we describe not only the architecture of* TIMBER, *its storage model, and engineering choices we made, but also present in hindsight, our retrospective on what went well and not so well with our design and engineering choices.*

The TIMBER system [10, 16] was developed at the University of Michigan, Ann Arbor, beginning 1999. It was an early native XML data management system. In this retrospective, we take stock of our work over the past nine years. Figure 1 provides an overview of the major system components. Secs. 1 through 4 describe the underlying algebra, query evaluation methods, query optimization, and indices, respectively. Sec. 5 mentions aspects of TIMBER not included in this article. Sec. 6 concludes with a retrospective view.

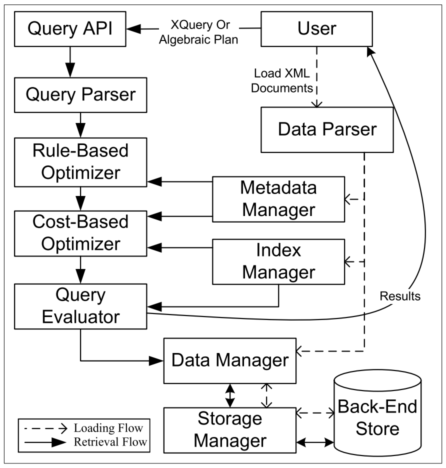


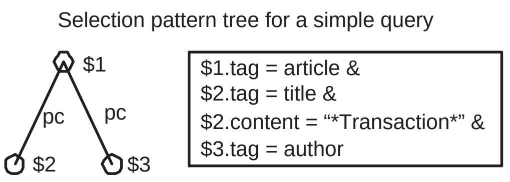
Figure 1: TIMBER Architecture: XML documents are parsed and nodes stored individually in the back-end store. Parsed queries, from multiple supported interfaces, go through a query optimizer to the query evaluator in a relatively standard overall database system architecture.

1 Algebra

Relational algebra has been a crucial foundation for relational database systems, and has played a large role in enabling their success. A corresponding XML algebra for XML query processing has been more elusive, due to the comparative complexity of XML, and its history.

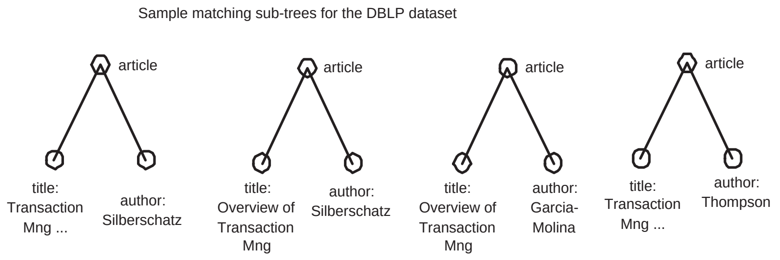
In the relational model, a tuple is the basic unit of operation and a relation is a set of tuples. In XML, a database is often described as a forest of rooted node-labeled trees. Hence, for the basic unit and central construct of our algebra, we chose an *XML query pattern* (or

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering



Selection pattern tree for a simple query
$1
pc
pc
$2
$3
$1.tag = article &
$2.tag = title &
$2.content = "*Transaction*" &
$3.tag = author

(a) Pattern tree for 'Select articles with some author and with title that contains Transaction'.



Sample matching sub-trees for the DBLP dataset
article
title: Transaction Mng ...
author: Silberschatz
article
title: Overview of Transaction Mng
author: Silberschatz
article
title: Overview of Transaction Mng
author: Garcia-Molina
article
title: Transaction Mng ...
author: Thompson

(b) Witness trees from the matching of the tree in Figure 2(a) to DBLP.

Figure 2: Pattern Tree and Witness Trees.

twig), which is represented as a rooted node-labeled tree. An example of such tree, we call it *pattern tree*, is shown in Figure 2(a). An edge in such tree represents a structural containment relationship, between the elements represented by the respective pattern tree nodes. The containment relationship can be specified to be either immediate (parent-child relationship) or of arbitrary depth (ancestor-descendant relationship). Nodes in the pattern tree usually have associated conditions on tag names or content values.

Given an XML database and a query pattern, the *witness trees* (pattern tree matchings) of the query pattern against the database are a forest such that each witness tree consists of a vector of data nodes from the database, each matches to one pattern tree node in the query pattern, and the relationships between the nodes in the database satisfy the desired structural relationship specified by the edges in the query pattern. The set of witness trees obtained from a pattern tree match are all structurally identical. Thus, a pattern tree match against a variegated input can be used to generate a structurally homogeneous input to an algebraic operator. Sample of witness trees can be found in Figure 2(b).

Using this basic primitives, we developed an algebra, called *Tree Algebra for XML* (TAX) [12], for manipulating XML data modeled as forests of labeled ordered trees. Motivated both by aesthetic considerations of intuitiveness, and by efficient computability and amenability to optimization, we developed TAX as a natural extension of relational algebra, with a small set of operators. TAX is complete for relational algebra extended with aggregation, and can express most queries expressible in popular XML query languages.

Ordering and Duplicates XML itself incorporates semantics in the order in which the data is specified. XML queries have to respect that and produce results based on this *document order*. XQuery takes this concept even further and adds an extra implicit ordering requirement. The order of the generated output is sensitive to the order the variable binding occurred in the query, the *binding order*. Additionally, a FLWOR statement in XQuery may include an explicit *ORDERBY* clause, specifying the ordering of the output based on the value of some expressions – this is similar in concept to ordering in the relational world and SQL.

Although XML and XQuery require ordering, many "database-style" applications could not care less about order. This leaves the query processing engine designer in a quandary: should order be maintained, as required by the semantics, irrespective of the additional cost; or can order be ignored for performance reasons. What we would like is an engine where we pay the cost to maintain order when we need it, and do not incur this overhead when it is not necessary. In algebraic terms, the question we ask is whether we are manipulating sets, which do not establish order among their elements, or manipulating sequences, which do.

The solution we proposed is to define a new generic *Hybrid Collection* type, which could be a set or a sequence or even something else. We associate with each collection an *Ordering Specification O-Spec* that indicates precisely what type of order, if any, is to be maintained in this collection.

Duplicates in collections are also a topic of interest, not just for XML, but for relational data as well. The more complex structure of XML data raises more questions of what is equality and what is a duplicate. Therefore there is room for more options than just sets and multi-sets. Our solution is to extend the *Hybrid Collection* type with an explicit *Duplicate Specification D-Spec*. Using our Hybrid Collections we extended our algebra [18] thus we were able to develop query plans that maintain as little order as possible during query execution, while producing the correct query results and managing to optimize duplicate elimination steps.

Tree Logical Classes (TLC) for XML XQuery semantics frequently requires that nodes be clustered based on the presence of specified structural relationships. For example the RETURN clause requires the complete subtree rooted at each qualifying node. A traditional pattern tree match returns a set of *flat* witness trees satisfying the pattern, thus requiring a succeeding grouping step on the parent (or root) node. Additionally, in tree algebras, each algebraic operator typically performs its own pattern tree match, redoing the same selection time and time again. Intermediate results may lose track of previous pattern matching information and can no longer identify data nodes that match to a specific pattern tree node in an earlier operation. This redundant work is unavoidable for operators that require a homogeneous set as their input without the means for that procedure to persist.

The loss of *Structural Clustering*, the *Redundant Accesses* and the *Redundant Tree Matching* procedures are problems caused due to the witness trees having to be similar to the input pattern tree, i.e. have the same size and structure. This requirement resulted in homogeneous witness trees in an inherently heterogeneous XML world with missing and repeated sub-elements, thus requiring extra work to reconstruct the appropriate structure when needed in a query plan. Our solution used *Annotated Pattern Trees (APTs)* and *Logical Classes (LCs)* to overcome that limitation.

Annotated Pattern Trees accept edge matching specifications that can lift the restriction of the traditional one-to-one relationship between pattern tree node and witness tree node. These specifications can be "-" (exactly one), "?" (zero or one), "+" (one or more) and "*" (zero or more). Figure 3 shows the example match for an annotated pattern tree. Once the pattern tree match has occurred we must have a logical method to access the matched nodes without having to reapply a pattern tree matching or navigate to them. For example, if we would like to evaluate a predicate on (some attribute of) the "A" node in Figure 3, how can we say precisely which node we mean? The solution to this problem is provided by our Logical Classes. Basically, each node in an annotated pattern tree is mapped to a set of matching nodes in *each* resulting witness tree – such set of nodes is called a *Logical Class*. For example in Figure 3, the gray circles indicate how the "A" nodes form a logical class for each witness tree. Using this techniques we extended TAX into our *Tree Logical Class* (TLC) algebra [19].

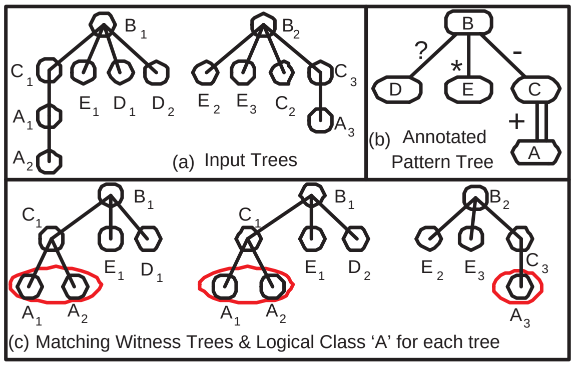

B1
B2
C1
A1
A2
E1
D1
D2
E2
E3
C2
C3
A3
(a) Input Trees
B
?
*
-
D
E
C
+
A
(b) Annotated Pattern Tree
(c) Matching Witness Trees & Logical Class 'A' for each tree

Figure 3: Sample Match for Annotated Pattern Tree

2 Query Evaluation

Data Storage The unit of storage in TIMBER is a node. For efficiency reasons, a node in the TIMBER *Data Manager* is not exactly the same as a DOM [22] node: there is a node corresponding to each element, with links to nodes corresponding to the first and last sub-elements; all attributes of an element node are clubbed together into a single node, which is then stored as a child node of that element node; the content of an element node, if any, is pulled out into a separate child node, in honor of interleaving of multiple sub-elements and text contents of mixed-type elements. We ignored all processing instructions and comments, which can be extended easily by creating nodes of those types.

In semi-structured data, the essential of the structural properties is reflected by the containment relationship between an element and its sub-elements. Establishing parent-child (or ancestor-descendant) relationships among nodes are the center parts of XML queries, and a sub-tree rooted at certain nodes are frequently demanded as query results. As such, the determination of the containment relationships is at the core of XML query processing. In TIMBER, we facilitated such operation by associating a numeric vector (start, end, level) with each data node in the database. The start and end labels of a node are the pre-order and post-order traversal of the node in the XML tree structure. Together, they define a corresponding interval such that every descendant node has an interval that is strictly contained in its ancestors' intervals. The level label reflects the depth of the node in the document for establishing parent-child relationships between nodes. Formally,

- A node $n_1(S_1, E_1, L_1)$ is the ancestor of node $n_2(S_2, E_2, L_2)$ iff $S_1 < S_2 \wedge E_1 > E_2$.
- A node $n_1(S_1, E_1, L_1)$ is the parent of node $n_2(S_2, E_2, L_2)$ iff $S_1 < S_2 \wedge E_1 > E_2 \wedge L_1 = L_2 - 1$.

The (start, end, level) vector of each node is generated automatically by the system during the data parsing process and stored together as attributes with each node. An additional doc label is associated with each node, such that the vector (doc, start, end, level) serves as a logical identifier for each node in a TIMBER database. We chose to store the nodes in the order of their start key, e.g. in document order, such that nodes within a sub-tree are always clustered together, hence guarantee efficient access of all nodes in a sub-tree, given the root.

Structural Join Evaluation TIMBER includes access methods corresponding to all operators in the TLC algebra. TLC (and TAX) operators have two parts: pattern match for witness tree identification followed by the actual operator application to the matched witness tree. Therefore, efficient pattern matching is crucial.

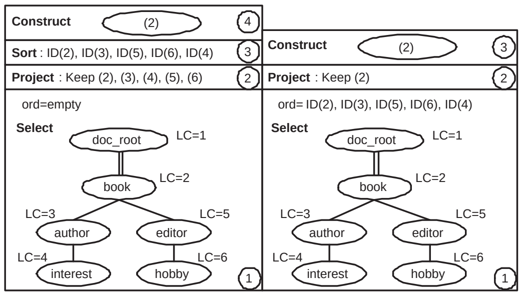
Pattern matching comprises two steps: first, value indices are used to look up matches for individual nodes in the pattern; then, *structural joins* are computed amongst these matched nodes. Structural join is the dominate operation in XML query evaluation both in terms of the frequency of usage and the cost. Consequently, efficient implementation of the structural join is critical to the efficient evaluation of XML queries in general.

Using the formulae of the containment relationship presented above, each structural join is represented as an ordinary relational join with a complex inequality join condition. Variations of the traditional sort-merge algorithm can be used to evaluate this join effectively, as suggested in [2, 26]. We exploited the tree structure of XML to do better. We have developed, and used in TIMBER, a whole *Stack-Tree* family of structural join algorithms.

The basic idea of the *Stack-Tree* algorithm is to take the two input operand lists, AList and DList, both ordered by the start position and merge them using a stack. It takes advantage of the fact that in a depth-first traversal of the database tree, every ancestor-descendant pair appears on a stack with the ancestor below the descendant, and perform a limited depth-first traversal, skipping over nodes that are not in either input candidate list (AList or DList). The output is a list of matching pairs, which satisfy the designated structural relationship, in ascending order of the *start key* of either the ancestor or descendent participating in the join. The sort order of the output is very important for pipelined query evaluation. The *Stack-Tree* algorithm is a non-blocking algorithm and can produce result as the join happens.

Small variations of the algorithms described above can be used if the desired structural join is a parent-child join rather than an ancestor-descendant join. Similarly, one can define semi-join, outer-join, and other variants. (Semi-joins, and left outer joins, in particular, seem to occur frequently in XML queries).

The algorithm requires an in-memory stack whose size is bounded by the maximum depth of the XML document. Even for the variant which requires the output to be sorted by the ancestor node, in which results has to be temperedly stored for each of the node in the stack, till the bottom of the stack is popped. Through careful list manipulation, we can perform this result-saving with limited memory buffer space and at most one additional I/O for any result page. The space and time complexity of the *Stack-Tree-Anc* algorithm is $O(|AList| + |DList| + |OutputList|)$. The I/O complexity is $O(\frac{|AList|}{B} + \frac{|DList|}{B} + \frac{|OutputList|}{B})$, where B is the blocking factor. (These asymptotic results apply to most other algorithms in the Stack-Tree family as well). Experiments show that these algorithms far outperform the navigation-based join algorithms, as well as the RDB implementation, in all cases.


(a) On the right the rewritten plan having pushed the *Sort* into the *Select*.

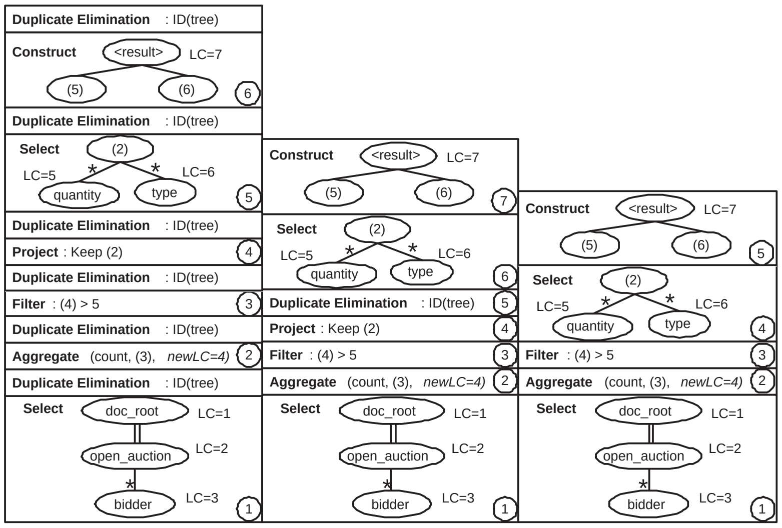

(b) Minimizing Duplicate Elimination procedures.

Figure 4: Order and Duplicate Rewrites.

3 Query Optimization

3.1 Algebraic Rewrites

In this section we demonstrate some of the advantages we got by using algebraic primitives to produce more efficient solutions. We discuss how we addressed grouping in XQuery and also show some algebraic rewrites that focus on smart placement of ordering and duplicate operations.

Grouping: While SQL allows for grouping operations to be specified explicitly, XQuery provides only implicit methods to write such queries. For example consider a query that seeks to output, for each `author` all the `titles` of `articles` he authored. A possible XQuery statement for this purpose (i.e. XQuery use case 1.1.9.4 Q4 from `http://www.w3.org/TR/xquery-use-cases`) would involve a nested FOR loop. A direct implementation of this query as written would involve two distinct retrievals from the bibliography database, one for `authors` and one for `articles`, followed by a join. Yet, one of our basic primitives of our algebra is a GROUPBY operator, thus enabling us to produce a smarter plan than the one dictated by XQuery. The power of the algebra allows for the transformation of the naïve join plan into a more efficient query plan using grouping – overcoming the XQuery nuances and making it similar to a relational query asking for the same information [17].

Duplicates and Ordering: As we discussed in Section 1, smart operation placement of ordering and duplicate elimination procedures can cause orders of magnitude difference in evaluation performance. We show two examples of such rewrites. Figure 4(a) shows how we optimize ordering. The rewrite takes advantage of our extended operations that use *Ordering Specification* annotations to push the `Sort` procedure into the original `Select`. Thus, the rewrite provides the cost-based optimizer with the means to efficiently plan the pattern tree match using the appropriate physical access methods, without having to satisfy a blocking `Sort` operation at the final step of the query plan. Figure 4(b) illustrates how the duplicate elimination procedures can be minimized. First, we naïvely force a duplicate elimination after every operation to produce the correct behavior. Then our technique detects and removes all redundant procedures by checking which operations will potentially produce duplicates. With the last step, we took advantage of our partial duplicate collections and manage to remove the duplicate elimination procedure completely. Details for both techniques can be found in [18].

3.2 Structural Join Order Selection

Join order selection is among the more important tasks of a relational query optimizer. Correspondingly, in an XML database, structural joins predominate. Every pattern match is computed as a sequence of structural joins, and the order in which these are computed makes a substantial difference to the cost of query evaluation. What's different from the relational engine is that (1) in the context of XML structural relationships can be as selective as value predicates; and (2) with the help of value indices and the structural join algorithms, structural join can be evaluated without accessing the original data. Therefore, it is not always a good idea to push selection predicates all the way down.

We proposed a set of algorithms for selecting the optimal join order for computing a pattern match [25]. The Dynamic Programming (DP) algorithm is capable of enumerating all possible evaluation plans and estimating their costs. This guarantees that the DP algorithm can select the optimal evaluation plan. However, the number of plans explored can be exponential in the size of the query pattern, making a full dynamic programming solution prohibitive. The Dynamic Programming with Pruning (DPP) algorithm explore only the most promising plans by pruning the costly plan in the early stage of plan enumeration.

A less expensive solution can be developed based on the following observation: by choosing an appropriate structural join algorithm, the results of a structural join can be output ordered by either of the two nodes involved in the join. No extra sorting is needed, and no blocking points created in the pipeline, if the *OrderBy* node in one join is a node involved in the next join. This leads to the finding that *any XML pattern matching can be evaluated with a fully-pipelined evaluation plan to produce results ordered by any node in the pattern tree*.

Contrary to the common wisdom in RDB query evaluation that a left-deep plan usually outperform bushy plans, the optimal evaluation plan for XML pattern matching can be a bushy plan. The Fully-Pipelined (FP) algorithm explores only these index-only plans, left-deep or bushy, in the join order selection process. Our experiments showed that not only can the FP algorithm select a very good (close to optimal) evaluation plan, it is itself also much more efficient than the DP and DPP algorithms.

3.3 Result Size Estimation

Query optimization techniques, as presented above, enumerates a subset of all the possible join plans and picks the one with the lowest cost to execute. To estimate this cost, we need an accurate estimate of the cardinality of the final query result as well as each intermediate result for each query plan. Even though the attributes participating in a join operation in RDB are often assumed to be independent, such assumption usually results into biased cardinality estimation in the context of XML, due to the fact that nodes *are* correlated, via parent-child or ancestor-descendant relationships that are natural to XML data.

The numeric start and end labels associated with each data node in the database define a corresponding interval between them. ignoreDescendant nodes have an interval that is strictly included. Taking the start and end pair of values associated with each node that satisfy a predicate, we constructed a two-dimensional histogram [23, 24]. Each grid cell in this *position histogram* represents a range of start position values and a range of end position values. The histogram maintains a count of the nodes satisfying the predicate that have start and end position within the specified ranges. Each data node is mapped to a point in this 2D space. Node A is an ancestor of node B iff node A is to the left of and above node B. Therefore, given the position histograms of two node predicate, the estimate of the join result of this two nodes can be computed by looping through each grid cell in the histogram of one node predicate and counting the number of nodes (in the other histogram) which can have the desired relationship with a node in that grid cell. The estimate can be represented in forms of a position histogram itself, which makes it possible to estimate the result sizes for complex query patterns.

4 Indexing

There is a rich history of work on index structures suited to specific purposes, in particular, the work done in the context of object-oriented systems, such as [4, 14], and more resent work on structural indices such as

DataGuide [8] and A(k)-index [13]. More importantly, we drew inspiration from the theoretical work that studies the properties of the XPath language, and found that suitable indices for XML should be those that (1) are based on the partition of XML components which corresponds to the partition induced by important sub-language of XPath; (2) label the partition to facilitate lookup; (3) organize the partitions in a way that facilitates retrieval of one or more such partitions; (4) support index-only plans for answering most XPath queries.

On the data side, seeing XML document D as a node-labeled tree, we formally define it as a 4-tuple (V, Ed, r, λ), with V the finite set of nodes, $Ed \subseteq V$ x V the set of edges, $r \in V$ the root, and $\lambda\colon V \to \mathcal{L}$ a node-labeling function into the set of labels $\mathcal{L}$. For a given pair of nodes m and n in an XML document D where m is an ancestor of n, we define its associated *label-path* to be the unique path between m and n, denoted $LP(m, n)$. Given a node n in D, and a number k, we define the k-*label-path* of n, denoted $LP(n, k)$, to be the label-path of the unique downward path of length l into n where $l = \min\{height(n), k\}$.[1].

We use the notion of label-paths to define $\mathcal{N}[k]$-equivalence such that two nodes are $\mathcal{N}[k]$-equivalent if the upward path of length k from them are identical. The $\mathcal{N}[k]$-*partition* of an XML document D is then defined as the partition induced by this equivalence relation. It immediately follows that each partition class C in the $\mathcal{N}[k]$-*partition* can be associated with a unique label-path, the label-path of the nodes in C, denoted $LP(C)$. On the other hand, a k-label-path p in an XML document D uniquely identifies an $\mathcal{N}[k]$-*partition* class, which we denote as $\mathcal{N}[k][p]$. In [7] we proved that the $\mathcal{N}[k]$-partition and the $\mathcal{A}[k]$-partition are the same. Similarly, we can define the $\mathcal{P}[k]$-*equivalent* relation between node pairs and the $\mathcal{P}[k]$-*partition* of node pairs in an XML document induced by the $\mathcal{P}[k]$-*equivalent* relationship.

XPath query language has been studied by many researchers. The XPath algebra, as proposed in [9], is defined as follows:

$$XPath\,algebra := \varepsilon|\emptyset|\downarrow|\uparrow|\ell|\lambda|E_1 \diamond E_2|E_1[E_2]|E_1 \cup E_2(X)|E_1 \cap E_2|E_1 - E_2$$

Where E_1 and E_2 are XPath algebra expressions. The *path semantics* of the algebra results into a set of node pairs, while the node semantics produces results in the form of node set. We focused our study on a few sub-algebras of XPath. The $\mathcal{D}$ algebra consists of the expressions in the XPath algebra without occurrences of the set operators, predicates ([]), or the $\uparrow$ primitive. The $\mathcal{D}^{[]}$ algebra consists of the $\mathcal{D}$ algebra plus predicates. More importantly, we studied a localized version of these sub-languages, e.g. $\mathcal{D}[k]$ and $\mathcal{D}^{[]}[k]$, restricting the length of the path to k.

The partition induced by a query language $\mathcal{F}$ under the path-semantics is defined as a partition of node pairs whereas two pairs are —em $\mathcal{F}$-equivalent to each other iff for any XML document D and any query expression in $f \in \mathcal{F}$, the node pairs are either together in $f(D)$, or together not in $f(D)$ [2].

We proved in [7] that the $\mathcal{P}[k]$-*partition* is the same as the $\mathcal{D}[k]$-*partition* of node pairs. In addition, we proved in [5] that every $\mathcal{D}^{[]}$ expression can be rewritten into sub-expressions in $\mathcal{D}[k]$, with the help of the inverse ($^{-1}$) operation, project operation and natural join operation to stitch the results of the sub-expressions together. Therefore, a proper index based on the $\mathcal{P}[k]$-*partition* of an XML document, with a modest k value, having the index entries featuring the (start, end, level) trio, is sufficient to support index-only evaluation plan for any XPath queries. Based on this theoretical result, we designed the $\mathcal{P}[k]$-Trie index, which uses the reversed label path as index key, and organizes the index entries in a trie structure. This index (1) has a reasonable size with a modest k; (2) is balanced, with k as the upper bound for the length of the search path; and (3) can answer queries of any length and with any arbitrary branching predicates with index-only plan. Our experiments showed that it outperformed the $\mathcal{A}[k]$-index by orders of magnitude.

[1] $height(n)$ denotes the height of node n in D.

[2] Similarly, we can define the partition induced by $\mathcal{F}$ under the node-semantics.

5 Other Contributions

Space constraints prevent us from describing contributions of the TIMBER project beyond the core components discussed above. In this section, we briefly mention some of these other efforts.

XML holds out the promise of integrating unstructured text with structured data. The challenge lies in developing query mechanisms that can marry the very different IR-style queries appropriate for text with the structured representations of logic used by databases. The TIX algebra [3] was an early effort at bringing these two together.

Uncertainty in databases has recently become a hot topic. Uncertainty is particularly important in the context of XML because of the nature of applications where information may be obtained from sources that are less uniformly structured, less under our control, and less reliable. The ProtDB [15] facility in TIMBER provides a natural model to represent probabilistic data in XML, and to query it efficiently.

One limitation of XML is that it requires all data to be organized in a strict hierarchy. Often, there isn't a single logical hierarchical structuring of the data. For examples, should publications be organized by year, by venue, or by author? Each may be more appropriate for some application scenarios, but XML requires that a single choice be made. TIMBER supports multi-color XML [11], where multiple hierarchies can be established, in different "color", on the same data. This multi-color facility is of particular value in a data warehousing context.

In addition to the macro benchmarks such as XMark [1], in developing an XML database system, we felt the need for a diagnostic benchmark. Traditional application level benchmarks included too many things in a single number so that it was hard for us to determine why performance was bad when we found it to be worse than we expected. We created MBench [20], an engineer's XML benchmark, for ourselves. MBench provides pairs of queries that differ in only one parameter value, thereby providing valuable information regarding what situations hurt performance.

6 Discussion and Conclusion

The heart of TIMBER is its algebra. Having this algebra allowed us to deal with a large subset of XQuery, including nesting, joins, grouping, and ordering, while at the same time enabling optimizations and set based processing. The heterogeneity of XML makes set-oriented processing difficult. The semantics of XQuery are defined in terms of a tuple-at-a-time nested loops structure, and this exacerbates the difficulty. The TIMBER family of algebras provide an elegant bridge across this divide.

Unfortunately, this was not one algebra, but rather a set of algebras. Since the algebra did not come before the query language, the algebra had to be extended to keep pace with language features and optimizations supported. This is as if there were SQL before relational algebra. And then we were to devise a sequence of algebras, RA, RA with grouping and aggregation, RA with cube and ROLAP support, and so on. While this was intellectually the right thing to do, this has kept one early algebra from becoming *THE* standard.

Knowing that the heart of our contribution would be at the algebra level, we consciously chose to focus on the upper layers of the database system, and use a data store for the lower layers. We chose to use Shore [6], because it was such a highly-regarded and widely used academic system. This turned out to be a mistake. For one thing, Shore was an academic project, and the code base was no longer supported by the time we began to use it. For another thing, sizes of main memory, and hence of "interesting" databases had grown substantially between the time Shore was implemented and the time TIMBER was implemented. We kept bumping up against Shore scaling barriers. Finally, a large part of the code in a storage manager such as Shore is devoted to transaction management. This was a feature we ended up never using in TIMBER. So we had a great deal of additional code to carry around without using. After several years, we switched to BerkeleyDB, and that addressed the first two problems above, but the third still remains.

In spite of the challenges mentioned above, Shore was a sufficiently robust engine, and the TIMBER code on top written well enough, that we were able to handle gigabyte size XML documents at a time when commercial native XML companies could only do a few megabytes at best. Since then, there has been significant commercial activity, and we believe many commercial engines, particularly those of relational vendors, will comfortably handle much larger sizes than this.

TIMBER is written in a multiplicity of languages, most importantly in C++ for the query evaluation engine and in C# for the parser and rule-based query optimizer. We were early adopters of Microsoft's Visual Studio .Net. Its cross-language development facilities worked as advertised for us, with only very minor glitches.

TIMBER code is written in a modular way, and source code is available for free download at [21]. We have had over a 1000 copies of TIMBER downloaded. However, we know anecdotally of at least some who downloaded the source but were unable to build a working executable. We believe we could have had many more users if only we could have constructed a smaller footprint system that was easier to build.

Neither TAX nor XQuery supported updates when we started TIMBER. We did build in some update facilities later, but these continue to feel like a retrofit. The weak support for updates once again highlights that transaction support is unnecessary.

In the document world, people are used to having thousands of documents, each relatively small. When XML is treated as a database, the entire database becomes one document. For the same total size of data, obtained as a product of these two, we could have one very large document or many small documents, or something in between. TIMBER consciously made an effort to support the former, knowing that this was a challenge for other native XML systems with a document processing orientation. This allowed TIMBER to shine, on the one hand, but also made comparisons harder.

In terms of a legacy, the stack-based family of algorithms is the one with the most significant impact among all parts of the TIMBER system. Since its introduction, the stack-based structural join algorithm has inspired a stream of work on structural join algorithms, query optimization techniques, indexing techniques, and result-size estimation techniques for XML. The original paper [2] has been cited 474 times according to Google Scholar to date, and has had dozens of researchers devise improvements.

In conclusion, TIMBER was a large systems project run on a shoe-string. The code is still available and is still being downloaded. It includes many novel ideas, and it certainly taught us a great deal about how to build a database system. However, the TIMBER system itself would have had much greater impact and use if we had found a way to bring it out sooner and smaller.

References

[1] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *VLDB*, pages 974–985, 2002.

[2] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proc. ICDE Conf.*, Mar. 2002.

[3] S. Al-Khalifa, C. Yu, and H. V. Jagadish. Querying structured text in an xml database. In *Proc. SIGMOD Conf.*, 2003.

[4] E. Bertino. An indexing technique for object-oriented databases. In *ICDE*, pages 160–170, 1991.

[5] S. Brenes, Y. Wu, D. Van Gucht, and P. Santa Cruz. Trie indexes for efficient xml query evaluation. In *WebDB*, 2008.

[6] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring up Persistent Applications. In *Proc. SIGMOD Conf.*, 1994.

[7] G. H. L. Fletcher, D. Van Gucht, Y. Wu, M. Gyssens, S. Brenes, and J. Paredaens. A methodology for coupling fragments of XPath with structural indexes for XML documents. In *DBPL*, pages 48–65, 2007.

[8] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[9] M. Gyssens, J. Paredaens, D. Van Gucht, and G. H. L. Fletcher. Structural characterizations of the semantics of XPath as navigation tool on a document. In *PODS*, pages 318–327, 2006.

[10] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C.Yu. TIMBER: A native XML database. *VLDB Journal*, 11(4), 2002.

[11] H. V. Jagadish, L. V. S. Lakshmanan, M. Scannapieco, D. Srivastava, and N. Wiwatwattana. Colorful xml: one hierarchy isn't enough. In *Proc. SIGMOD Conf.*, 2004.

[12] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *Proc. DBPL Conf.*, Sep. 2001.

[13] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, pages 129–140, 2002.

[14] C. Kilger and G. Moerkotte. Indexing multiple sets. In *VLDB*, pages 180–191, 1994.

[15] A. Nierman and H. V. Jagadish. Protdb: probabilistic data in xml. In *Proc. VLDB Conf.* 2002.

[16] S. Paparizos, S. Al-Khalifa, A. Chapman, H. V. Jagadish, L. V. S. Lakshmanan, A. Nierman, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native system for quering XML. In *Proc. SIGMOD Conf.*, Jun. 2003.

[17] S. Paparizos, S. Al-Khalifa, H. V. Jagadish, L. V. S. Lakshmanan, A. Nierman, D. Srivastava, and Y. Wu. Grouping in XML. *Lecture Notes in Computer Science*, 2490:128–147, 2002.

[18] S. Paparizos and H. V. Jagadish. Pattern tree algebras: sets or sequences? In *Proc. VLDB Conf.*, 2005.

[19] S. Paparizos, Y. Wu, L. V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *Proc. SIGMOD Conf.*, Jun. 2004.

[20] K. Runapongsa, J. M. Patel, H. V. Jagadish, Y. Chen, and S. Al-Khalifa. The michigan benchmark: towards xml query performance diagnostics. *Inf. Syst.*, 31(2):73–97, 2006.

[21] Timber Group at Univ. of Michigan. Timber system. `http://www.eecs.umich.edu/db/timber`.

[22] W3C DOM Working Group. Document Object Model. `http://www.w3.org/DOM/`.

[23] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating answer sizes for XML queries. In *Proc. EDBT Conf.*, Mar. 2002.

[24] Y. Wu, J. M. Patel, and H. V. Jagadish. Using histograms to estimate answer sizes for XML queries. *Information Systems*, 2002.

[25] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *Proc. ICDE Conf.*, Mar. 2003.

[26] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proc. SIGMOD Conf.*, 2001.

XQuery Rewrite Optimization in IBM® DB2®[*] pureXML™

Fatma Özcan
IBM Almaden Research Center
650 Harry Road, San Jose

Normen Seemann
IBM Silicon Valley Lab
555 Bailey Road, San Jose

Ling Wang
IBM Silicon Valley Lab
555 Bailey Road, San Jose

Abstract

In this paper, we describe XQuery compilation and rewrite optimization in DB2 pureXML, a hybrid relational and XML database management system. DB2 pureXML has been designed to scale to large collections of XML data. In such a system, effective filtering of XML documents and efficient execution of XML navigation are vital for high throughput. Hence the focus of rewrite optimization is to consolidate navigation constructs as much as possible and to pushdown comparison predicates and navigation constructs into data access to enable index usage. In this paper, we describe the new rewrite transformations we have implemented specifically for XQuery and its navigational constructs. We also briefly discuss how some of the existing rewrite transformations developed for the SQL engine are extended and adapted for XQuery.

1 Introduction

XML has emerged in the industry as the predominant mechanism for representing and exchanging structured and semi-structured information across the Internet, between applications, and within an intranet. Key benefits of XML are its vendor and platform independence and its high flexibility. With the proliferation of XML data, several XML management systems [7, 10, 17, 5, 4, 6, 12, 11, 14] have been developed over the last couple of years. All major database vendors have released XML extensions to their relational engines, in addition to many native XML management systems. XQuery [18] and SQL/XML [9] are the two industry-standard languages that are supported by these systems to query XML. Most of the current research now focuses on optimization of XQuery and SQL/XML in these XML management systems.

In this paper, we describe XQuery rewrite optimization within the context of *DB2 pureXML* [4], which is a hybrid relational and XML database engine that provides native XML storage, indexing, navigation and query processing through both SQL/XML [9] and XQuery [18], using the XML data type introduced by SQL/XML. *DB2 pureXML* stores XML data in columns of relational tables, as instances of the XQuery data model [19] in a structured type-annotated tree. By storing binary representation of type-annotated trees, *DB2 pureXML* avoids repeated parsing and validation of documents. *DB2 pureXML* [4] query evaluation run-time contains three major components for XML query processing: (1) XML navigation engine, (2) XML index run-time and (3) the XQuery function library. Additionally, several relational runtime operators have been extended to deal

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*]DB2 pureXML is a a trademark or registered trademark of International Business Machines Corporation.

with XML data. The XML navigation engine evaluates path expressions over the native store, by traversing the parent-child relationships in XML storage. It returns node references and atomic values to be further processed by the query run-time. Unlike other approaches in which every XPath step is modeled as separate operator [6, 16, 5], a single navigation operation in *DB2 pureXML* can evaluate multiple XPath expressions, consisting of multiple steps, as a whole. After parsing both SQL/XML and XQuery queries are mapped into a unified internal representation and optimized by the hybrid query compiler [4].

An important decision which impacted the whole XQuery compiler design is that *DB2 pureXML* does not require all XML documents in an XML column conform to a single schema, or to a collection of conforming schemas, and it does not implement static typing. Static typing is too restrictive for evolving schemas, as each document insertion or change in schema may result in recompilation of applications. As a result, XPath transformations that exploit schema information cannot be applied in *DB2 pureXML*. Instead, we focus on rewrites that optimize the general data flow in a complex XQuery or SQL/XML query. In this paper, we describe those rewrites that we developed for XQuery.

The rest of this paper is organized as follows: In Section 2, we provide an overview of how XQuery is modeled in *DB2 pureXML*, and then in Section 3 we describe rewrite transformation developed for XQuery. Finally, we conclude in Section 4.

2 XQuery Compilation in *DB2 pureXML*

DB2 pureXML provides a hybrid compiler, supporting both XQuery and SQL/XML queries. It contains several modules: two parsers, one for XQuery and one for SQL/XML, a global semantics module, a rewrite module, a cost-based optimizer module, and a code-generation module, executed in this order. XQuery and SQL/XML queries are first parsed using their respective parsers. The output of the parsers is a unified internal representation, i.e. the QGM (Query Graph Model) graph. The rest of the processing is common for both languages. The rewrite module contains a rule-based transformation engine [15], as well as several transformations that are applied before or after the rule-based engine. It applies algebraic transformation to the QGM graph. The cost-based optimizer translates the final QGM produced by the rewrite module into query execution plans and choses the optimal one. The focus of this paper is the rewrite module. But, in this section, we will start with an overview of basic QGM[15] and its extensions to XQuery, necessary to understand the rewrites.

In its simplest form, a QGM graph consists of operations (nodes) and quantifiers (arcs) which represent the data flow between operations. QGM supports arbitrary table operations, where the inputs and outputs are tables. Examples of operations include SELECT, GROUP BY, UNION, and etc. The SELECT operation node in QGM roughly represents a SPJ query block and handles restriction (selection), projection, as well as joins. Each operation consumes a set of input columns through its input quantifiers, and produces a set of output columns. Quantifiers range over operation nodes or base tables, and carry the input columns. There are two types of quantifiers: *ForEach* and *Any/All*. The expression within an operation node is applied to each tuple input by a *ForEach* quantifier. *Any/All* quantifiers are used to express universally (or existentially) qualified predicates.

XQuery [18] includes similar constructs to iterate over XML sequences, apply predicates and sort data. We exploit many existing features of QGM to model these XQuery features and introduce new entities to represent and manipulate XPath expressions and XML sequences. In general, the result of every XQuery expression is a sequence of items. Since XQuery sequences, i.e. XQDM (XQuery data model) [19] is represented as a column in *DB2 pureXML*, any sub QGM-graph that is created to represent a specific XQuery expression produces a table with a single row and a single column of type XML. FLWOR and quantified expressions define new variables that are in scope within their respective expressions. To keep track of these variable scopes, we model FLWOR and quantified expressions as scalar sub-queries, with explicit QGM operation nodes defining the query blocks. The rest of the XQuery expressions that we support are represented as scalar functions; they either have run-time counterparts that implement them, or they are expanded into detailed QGM operations later in the compiler.

Some XQuery expressions consume a sequence as a whole (such as functions), while others require iterating through the items in a sequence. We need to model these different ways of how XML data is flown into various XQuery expressions. For XQuery, we have introduced two new kinds of *ForEach* quantifiers, *FOR* and *LET*. A LET quantifier aggregates the output of an operation node into an XML sequence, whereas a FOR quantifier unnests XML sequences output by an operation node and iterates over every single item. For example, if an operation node produces a table with two rows containing $\{a, b, c\}$ and $\{d, e\}$, then the output of a LET quantifier is a single row that contains all items, i.e. $\{a, b, c, d, e\}$, whereas the output of a FOR quantifier is a table with five rows, each row containing a single item.

2.1 Representation of XPath Expressions

XPath [18] expressions consist of a series of steps, where each step either expresses navigation, or contains another XQuery expression, such as an XQuery built-in function, a FLWOR or a quantified expression, or a node constructor. The focus of earlier research has been on efficient representation and execution of XPath expressions, which contain only navigational steps. Most systems [16, 5, 6, 17, 14, 11, 7], represent and execute each step separately as selections. In other words, they normalize [20] XPath expressions into explicit FLWOR blocks, where iteration between steps and within predicates is expressed explicitly. Some provide indexes for efficient access to individual nodes. But, they all require structural joins [2] to establish parent-child (or ancestor-descendant) relationships.

In *DB2 pureXML*, we support XPath expressions, with its full generality and allow any XQuery expression in an XPath step or predicate. In general, we do not normalize XPath expressions, except in some certain cases. Instead, we represent XPath expressions, which may contain many steps and branches, as a pattern tree which computes a single variable binding. As the XML navigation engine of *DB2 pureXML* holistically computes an XPath expression, we do not need to model each step separately and we do not need structural joins to combine the results. Later, rewrites combine multiple XPath expressions into a single pattern tree, which computes multiple variable bindings.

We introduce a new operation, namely the *ExpBox*, to represent XML navigation. An ExpBox contains an annotated pattern tree, and produces tuples of XQDM bindings. A pattern tree is a tree representation of many co-mingled XPath expressions. A pattern tree node represents an XPath step and has three or more positional children. The first child of a pattern tree node represents the axis, the second one is either a name, a kind, or a wildcard test ”*”, and the third child represents the predicate. The rest of the children of a pattern tree node represents the next steps, and are other pattern tree nodes. Pattern tree nodes are annotated with flags to capture various properties. The *isExtraction* flag is set to *true*, if the pattern tree node computes a variable binding that needs to be extracted and returned to the run-time engine for further processing. The *isFor* flag is set to *true* if the pattern tree node represents the last step of a FOR binding. A pattern tree node can be marked as a FOR even if it does not represent an extracted variable binding. When XPath expressions are merged to eliminate unnecessary extractions, we need to remember the last step of a FOR binding so that navigation run-time can apply the correct duplicate elimination and document order rules. The *EmptyOnEmpty* flag signals when an empty sequence needs to be created if there is no qualifying node.

2.2 Representation of FLWOR Expressions

The FOR and LET bindings in a FLWOR expression produce a tuple stream, which is then filtered by the **where** clause, and the **return** clause is invoked for each surviving tuple. We model the FLWOR expression by using two SELECT operations. The lower one computes the FOR and LET bindings and applies the **where** clause predicates. We create a sub-graph for each binding and create either a FOR or a LET quantifier over it. These FOR and LET quantifiers, which provide the tuple stream as input to the lower SELECT node, reflect the join semantics of the FLWOR expression. Its output is fed to another SELECT operation, which is used to model the

return clause and the **order by** clause, if present. Later in query rewrites these two select boxes may be merged depending on the properties of the expressions in the **order-by** and **return** clauses.

3 XML Rewrites

The rule-based rewrite engine of DB2 provides several rewrite transformations for relational data [15]. Some of these rewrites are also applicable to XQuery, as they optimize the data flow in QGM by minimizing the number of operations and the length of the data flow, and both SQL and XQuery are modeled with QGM. For example, there is a rewrite which merges SELECT operation nodes. This rewrite is extended to deal with the new quantifier types, which are introduced for XQuery. This rewrite enables unfolding of nested FLWOR blocks, and minimizes the QGM graph significantly. There are other rewrites which would not be applicable and those are blocked for XQuery operations.

In this section, we focus on the new set of rewrite transformations introduced for XQuery, namely rewrites for optimizing XPath expressions and the new LET and FOR quantifiers. The main goal of these new rewrites is to consolidate XPath expressions into the least number of navigation operation nodes possible, as well as to bring comparisons into XPath expressions and close to the table access to enable XML index usage.

DB2 pureXML supports value indexes defined by XPath expressions. These indexes are used to answer XPath expressions which contain value or general comparisons. *DB2 pureXML* employs XML indexes to eliminate documents that do not satisfy XPath predicates, and uses XPath query containment algorithms of [3] to decide whether an index is eligible.

Most of the new rewrites work as part of the rule-based engine, but we also provide some transformations that are outside. If the transformation can fire multiple times and interacts with other rewrites to enable them or is enabled by them, we implement it as part of the rule-based engine. Otherwise, it is implemented as a one-time only transformation. The rewrites that are part of the rule-based engine work on one aspect, such as a quantifier or an operation node, of the QGM graph at a time and collectively simplify the QGM graph.

In addition to these rewrites, we also provide a separate rule-based transformation engine just for XPath expressions. The transformations in this set work on a single XPath expression, usually one XPath step at a time. These transformations include rules that normalize XPath expressions by eliminating parent axes, converting multiple predicates on a step into a conjunction when possible, among others.

Note that *DB2 pureXML* does not support static typing, but type information is important in query optimization. Type information can be derived from two places: from the XML schema against which the document has been validated, and from the signatures of the applied functions and operators. For example, fn:count() function always returns a single integer, and fn:data() function always generates an atomic type. We use the return data types of functions and operators, as well as literals, to infer the data type of an operation. We exploit type information both in index matching, as well as in some rewrites. For example, the FOR2REG rewrite, which is explained below, will fire if the data type of the XML column is a singleton.

In the following, we describe the general conditions under which the rewrites will fire. The actual rules contain more details, which we omit here due to space limitations.

3.1 LET and FOR Quantifier Rewrites

As discussed earlier, a LET quantifier requires aggregating the results of the operation node it ranges over, so it is translated into a group-by operation, and it is blocking. A FOR quantifier, on the other hand, needs to iterate over the results of the operation node it ranges over, and it is translated into an UNNEST operation. It desirable to eliminate both kinds of operations, if possible. We provide rewrites which tries to convert a LET quantifier into a FOR and a FOR quantifier into a regular (REG) quantifier. The first condition we check for both rewrite is that the operation node that the quantifier ranges over is not a common subexpression.

In its simplest form, we can convert a FOR quantifier into a REG quantifier if we can prove that the operation node it ranges over produces one singleton sequence. To prove this property, we may have to trace the computation back several operations. Converting a LET quantifier into a FOR is more involved and requires more properties to be proved. We check separate conditions depending on the operation node the LET quantifier ranges over. If it is a SELECT operation, then we check whether there is a subsequent FOR or a LET quantifier that obliterates this LET step, ensuring that there is no operation in between that requires to consume the output of the LET quantifier as a single sequence. If the operation node is an ExpBox, i.e. an XPath expression, then we need to prove that this XPath expression is *input independent*. We say that an XPath expression is *input independent* if its context sequence contains distinct nodes, and the subtrees pointed to by the nodes in the context sequence do not overlap. This will be true when the context column is a base table column, or the XPath expression consists of only navigational steps, and does not contain any descendant axis or positional predicates.

3.2 XPath Merging

There are two forms of XPath merging: one rewrite transformation which is part of the rule-based engine, and another one that is applied after all rewrites. The first one merges two XPath expressions, $xpath_1$ and $xpath_2$, if 1-) $xpath_1$ computes the context of $xpath_2$, 2-) there is no predicate on the first step, i.e. the context step, of $xpath_2$, 3-) the output of $xpath_1$ is only used in $xpath_2$ as the context, and 4-) $xpath_1$ and $xpath_2$ are compatible in their distinctness properties. When we merge these two XPath expressions, we create a new ExpBox containing the XPath expression that is the concatenation of $xpath_1$ and $xpath_2$, without its context step, and we mark the quantifier ranging over this new node same as the quantifier ranging over $xpath_2$. Note that if $xpath_1$ is a FOR binding, then we need to be careful to produce the correct set of results. For example, suppose $xpath_1$ is a FOR binding and produces \$i as $\$doc//customer$ and $xpath_2$ is a LET binding given by $\$i/accountId$. If the document has multiple customers, the final output should be a set of account id's for each customer. When we merge the two XPath expressions into $\$doc//customer/accountId$ and mark the final output as a LET binding, we also mark the intermediate $customer$ step as a FOR step, so that our navigation run-time produces the correct output.

The second transformation takes as input the resulting QGM after all the rewrites have been applied. It first computes a dependency graph among the XPath expressions in a query block, i.e. a SELECT operation node. Next, the algorithm partitions the set of XPath expressions within the same query block that are over the same document into clusters, by taking into account the interactions with other operations in the query so as not to sacrifice an optimal execution plan. Finally, it merges the XPath expression within the same cluster, as long as the resulting dependency graph is acyclic. This transformation produces expressions which compute multiple bindings. The details of this rewrite can be found in [1].

3.3 Resetting EmptyOnEmpty Flag

A let-clause binds its variable to the result of the associated expression, even when the result of the expression is an empty sequence. As all values of the LET bindings need to be returned, we cannot use an XML index to compute the expression in a LET binding, unless we can prove certain properties. We introduce a new quantifier flag, called *EmptyOnEmpty*, which signals that the quantifier needs to produce an empty sequence, even if the operation node it ranges over produces no results. When we first parse an XQuery expression, we create a LET quantifier over all XQuery expressions, and over LET bindings, because all XQuery expressions have implied LET semantics [18]. Later, we provide a rewrite transformation which tries to reset this flag, enabling both index usage and several other rewrites, most notably the one that merges SELECT boxes.

In general, we can reset the *EmptyOnEmpty* flag when there is a **where** clause predicate which eliminates the empty sequence, and there are no other consumers of that LET binding. Moreover, there are two other XQuery operations which discard the empty sequences, iterators, such as FOR clauses, and sequence concatenation. If we prove that the empty sequence is to be discarded later on due to one of these operations, we can reset the *EmptyOnEmpty* flag.

3.4 Local Predicate Pushdown into XPath Expressions

Similar to pushing down selections in a relational query, we provide a rewrite which tries to push down *local* predicates into base column accessing XPath expressions to filter out unqualified data as early as possible. We consider a predicate to be *local*, if it accesses only one document. Moreover, an XPath such as $\$doc/cid$ can also be considered as a local predicate by converting it into $\$doc[cid]$, and can be pushed down to its context XPath. We call this *XPath pushdown*.

3.4.1 XPath Pushdown

XPath itself can be considered as a local predicate, as navigation steps are also existential tests. A set of rewrite rules together implement XPath push down. This set mainly includes (1) rules to push down XPath through operations such as SELECT and UNION, base tables, and XML element construction, and (2) *XPIMPLY* rule, which converts XPath into a local predicate.

An XPath can be pushed down if the following conditions hold: 1-) The XPath expression consists of only navigational steps, and does not have any steps containing functions, such as $\$doc/a/fn : concat(b, c)/d$. Note that functions in predicates do not block this rewrite. 2-) There are no common subexpressions along the path where the XPath expression will be pushed down. 3-) The XPath expression is *input independent*. 4-) The target operation node does not have any sorting requirements.

During push down, each rule pushes down the XPath expression through one operation node at a time. The rule engine remembers the current pushable position and makes a new copy of the pushable XPath expression. It then recursively calls the next rewrite to further push down the XPath expression. This way we try to reach to the base table level, where we can enable index matching. Once the rule engine locates the operation node where the XPath expression cannot be pushed down any further, *XPIMPLY* rule fires and converts an XPath expression of the form $\$d/steps$ into $\$d[steps]$, provided that there is no other consumer for this XPath expression.

3.4.2 Local Predicate Pushdown

This rewrites pushes down a **where** clause predicate into an XPath expression. Consider the following query: *Query I:* $for\ \$c\ in\ db2 - fn : xmlcolumn(\text{"}T2.DOC\text{"})/c,\ \$a\ in\ \$c/a\ where\ \$c/d = 5\ return\ \$c.$

The predicate $\$c/d = 5$ in this query can be pushed down into the first XPath expression, and rewritten as: *Query II:* $for\ \$c\ in\ db2 - fn : xmlcolumn(\text{"}T2.DOC\text{"})/c[d = 5],\ \$a\ in\ \$c/a\ return\ \$c.$

In general, a **where** clause predicate can be pushed down into the context XPath expression if: 1-) It is a *local* predicate, containing general and value comparisons, connected with conjunction and/or disjunction, 2-) It is not a predicate on an aggregation result, and 3-) The target XPath expression is a FOR binding. This rewrite does not work only in a single query block. Instead, when we locate such a candidate predicate, we disconnect it from its current SELECT operation node, and try to push it down as many query blocks as possible. This rewrite helps consolidate XPath expressions, and may enable merging of further XPath expressions. For example, for *Query II*, XPath merging rule will fire at some point, and merge the two XPath expressions, consolidating the whole query into a single XPath expression.

3.5 Join Pull up (Simple Decorrelation)

Consider *Query I* below, which contains an XPath expression with a correlated variable, expressing a join. There are several problems with this query: 1-) The join order is fixed due to the correlation, 2-) Only nested-loop join method can be used, and 3-) Only an index on T1 can be used, and any index on T2 cannot be exploited, because the XPath expression on T2 needs to be executed first.

Query I:

```
for $i in db2-fn:xmlcolumn(”T2.DOC”)/c,
$j in db2-fn:xmlcolumn(”T1.DOC”)/a[b=$i/d]
return $j
```

Query II:

```
for $i in db2-fn:xmlcolumn(”T2.DOC”)/c,
$j in db2-fn:xmlcolumn(”T1.DOC”)/a
where $j/b=$i/d
return $j
```

To address these problems, we provide a rewrite, called *join pull up*, which pulls up join conditions embedded in XPath expressions into the **where** clause, decorrelating the query. For example, *Query I* will be converted into *Query II*. This enables the optimizer to consider using both join orders, all join methods, as well as both indexes on T1 and T2. In general, a join predicate can be pulled up when all of the following conditions hold: 1-) The quantifier ranging over the ExpBox containing the join predicate, is either a FOR quantifier, or a LET quantifier, which does not have the *EmptyOnEmpty* flag set and which is not consumed anywhere else. 2-) The join predicate is either a general or a value comparison. 3-) It is the last predicate of a predicate sequence. 4-) It is a predicate, which maybe connected by a conjunction. Given a predicate of form $xp[prd1AND(prd2ORprd3))$, only $prd1$ is considered for pull up.

3.6 Query Decorrelation Rewrites

A correlation is a reference to a variable that has been defined in a previous or enclosing query block. Correlated subqueries are quite common in XQuery. For example, most grouping queries in XQuery are expressed using correlation. Although this a natural way of writing queries, it provides several performance bottlenecks: It severely limits the optimizer choices, because the correlation imposes a partial join order, and only a nested-loop join method can be used. Moreover, in a parallel environment correlation creates a synchronization point, and becomes a bottleneck in the data flow.

As we discussed earlier, *DB2 pureXML* query compiler already employs a variety of simplifying rewrite transformations, which may decorrelate some of the simple cases, such as join-pull up rewrite. However, only the magic decorrelation rewrite [13] addresses the most general problem. The magic decorrelation algorithm is closely entwined with the magic sets rewrite [8]. For convenience, we highlight the aspects of these rewrites that need to be revisited for XML processing.

When magic processing a subquery that contains the correlation variable, we generate a magic operation node as a SELECT DISTINCT operation, joining all the eligible quantifiers. Eligible predicates are pushed to form a semi join under an adornment node, effectively filtering the data stream. The adornments consists of the set of conditioned, bound, and free variables which are determined by the pushed predicates. Simply put, the magic sets rewrite generalizes local predicate pushdown to join predicates. Enforcing distinctness in the magic node is important so that we do not increase the total cardinality. Magic decorrelation rewrite [13] extends the magic sets to correlations. In this case, the magic node flows all to-be-decorrelated columns.

The main challenge in decorrelating an XML-typed variable reference is enforcing distinctness in the magic node. There are different ways in which XML data can be compared. One natural way is to employ the $fn:data()$ function to retrieve a comparable value. However, this approach can be costly since we potentially deal with large XML-structures. Another way is to use node id’s (which are comparable) to perform equality comparisons and GROUP BY operations. However, XML type can contain both nodes and atomic values, which do not have id’s. If we can prove that the XML type only contains nodes, we can use the id-based approach. But, in the general case a better solution is to ensure that we do not have to enforce distinctness. We can achieve this by adding keys to the magic node and to the list of to-be-decorrelated columns during decorrelation. We can obtain keys from descendant nodes as follows: For base tables, we pull up any key defined on the table. If no such key exists, we can use the record identifiers of the base tables. For a node which enforces distinctness, we can pull up all of its output columns. For any join node, we can pull up keys from every join operand. If we can determine and add such keys, then we do not have to enforce distinctness on the magic node, and we do not add any GROUP BY columns or equality predicates using any XML-typed columns. Naturally, we cannot always determine such keys. However, we observed that this approach is better-suited and more flexible for a majority of queries.

4 Conclusion

In this paper, we described XQuery compilation and algebraic rewrite optimization within the context of *DB2 pureXML*, a hybrid relational and XML database engine. We focused on rewrites whose main goal was to consolidate the XPath expressions in the query into the least number of possible navigation operations and enable index usage. We provide other rewrites, which are needed to simplify the QGM graphs generated for XQuery and SQL/XML, in addition to the rewrites we described here. We omit those due to space limitations.

References

[1] A. Balmin and F. Özcan and A. Singh and E. Ting. Grouping and optimization of XPath expressions in DB2 pureXML. In *Proc. of SIGMOD*, 2008.

[2] S. Al-Khalifa et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proc. of ICDE*, 2002.

[3] A. Balmin, F. Özcan, K. Beyer, R. Cochrane, and H. Pirahesh. A framework for using materialized XPath views in XML query processing. In *Proc. of VLDB*, Toronto, Canada, 2004.

[4] K. Beyer et al. System RX: One part relational, one part XML. In *Proc. of ACM SIGMOD*, pages 347–358, 2005.

[5] C. Re and J. Simeon and M.F. Fernandez. A Complete and Efficient Algebraic Compiler for XQuery. In *Proc. of ICDE*, 2006.

[6] D. Florescu et al. The BEA Streaming XQuery Processor. *VLDB Journal*, 13(3):294–315, 2004.

[7] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *Proc. of VLDB*, Toronto, Canada, 2004.

[8] I. S. Mumick and H. Pirahesh. Implementation of Magic-sets in a Relational Database System. In *Proc. of SIGMOD*, pages 103–114, 1994.

[9] International Organization for Standardization (ISO). Information Technology-Database Language SQL-Part 14: XML-Related Specifications (SQL/XML), ANSI/ISO/IEC 9075-14:2006.

[10] H. V. Jagadish et al. TIMBER: A Native XML Database. *VLDB JOurnal*, 11(1):274–291, 2002.

[11] Z.H. Liu, M. Krishnaprasad, and V. Arora. Native XQuery Processing in Oracle XMLDB. In *Proc. of SIGMOD*, pages 828–833, 2005.

[12] M. J. Carey. Data Delivery in a Service-oriented World: The BEA AquaLogic Data Services Platform. In *Proc. of SIGMOD*, pages 695–705, 2006.

[13] P. Seshadri, H. Pirahesh and T. Y. C. Leung. Complex Query Decorrelation. In *Proc. of ICDE*, 1996.

[14] S. Pal et al. XQuery Implementation in a Relational Database System. In *Proc. of VLDB*, 2005.

[15] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proc. of SIGMOD*, pages 39–48, 1992.

[16] J. Shanmugasundaram et al. Querying XML Views of Relational Data. In *Proc.of VLDB*, pages 261–270, Roma, Italy, September 2001.

[17] T.Fiebig et al. Anatomy of a Native XML Base Management System. *VLDB JOurnal*, 11(4), 2002.

[18] *XQuery 1.0: An XML Query Language*, January 2007. W3C Recommendation, See `http://www.w3.org/TR/xquery`.

[19] *XQuery 1.0 and XPath 2.0 Data Model*, January 2007. W3C Recommendation, See `http://www.w3.org/TR/xpath-datamodel`.

[20] *XQuery 1.0 Formal Semantics*, January 2007. W3C Recommendation, See `http://www.w3.org/TR/query-semantics`.

Towards a Unified Declarative and Imperative XQuery Processor

Zhen Hua Liu, Anguel Novoselsky, Vikas Arora
Oracle Corporation
{zhen.liu,anguel.novoselsky,vikas.arora}@oracle.com

Abstract

Since the birth of XML, the processing of XML query languages like XQuery/XQueryP has been widely researched in the academic and industrial communities. Most of the approaches consider XQuery as a declarative query language similar to SQL, for which the iterator-based (stream-based), lazy evaluation processing strategy can be applied. The processing is combined with XML indexing, materialized view, XML view query rewrite over source data. An alternative approach views XQuery as a procedural programming language associated with eager, step-based evaluation, where each expression is fully evaluated by the end of the corresponding expression execution step. Usually, this approach uses a virtual machine running byte-code for compiled programs. In this paper, we share our experience of building a unified XQuery engine for the Oracle XML DB integrating both approaches. The key contribution of our approach is that the unified XQuery processor integrates both declarative and imperati ve XQuery/XQueryP processing paradigms. Furthermore, the processor is designed with a clean separation between the logical XML data model and the physical representation so that it can be optimized with various physical XML storages and data index and view models. We also discuss the challenges in our approach and our overall vision of the evolution of XQuery/XQueryP processors.

1 Introduction

The original XQuery is more SQL-like declarative query languages and this is why XQuery is initially adopted as a language for querying persistent XML data in database environments. The latest XQuery extensions - XQuery Update Facility and especially XQueryP [2, 15] pushes the XQuery evolution far beyond what the original goal is. Currently, XQuery/XQUpdate/XQueryP is capable of not only for querying but also for transforming, updating and manipulating both persistent and transient XML data in variety of environments. This means that there is no need to embed XQuery with a host procedural programming languages, such as Java/C to build large scale XML applications.

Similar to that of SQL/PSM or Oracle PL/SQL, XQueryP combines a hybrid imperative-declarative processing paradigm with a single XML data model. This way, users can use declarative query constructs to do 'finding the needles in the haystack' type of data search efficiently by leveraging index built over large volume of XML data collections. Meanwhile users can use imperative programming constructs in order to do data transformation operations efficiently leveraging classical imperative language processing paradigms. Therefore, the design of XQuery/XQueryP processors needs to embrace both SQL like declarative language processing paradigm and

Java/C like imperative language processing paradigm. This is the primarily design philosophy that we follow to build an integrated XQuery/XQueryP processor in Oracle.

Furthermore, we take into account XML shapes and characteristics - the XML data can have different physical representations: persistent XML data with different storage, index and materialized views, XML view over relational data, transient based XML data stream, in memory XML DOM tree structures, stream of SAX events. Therefore, we don't want to design an XQuery/XQueryP processor that is hardwired to a limited number of physical XML representation forms. Instead, our second design paradigm is to keep a clean separation between program logic and XML representation so we can apply both XML representation independent and XML representation dependent optimizations on an XQuery program. This is similar to that of the well known compiler design principle that separates target independent and target dependent optimization.

2 Oracle XQuery/XQueryP Product and Requirement Overview

Riding with the XML/XQuery technology wave, Oracle XMLDB [3] supports XML, XQuery/XPath/XSLT and SQL/XML [1] processing in Oracle DBMS database server. In addition, Oracle also supports XDK package, where XQuery/XSLT processors can be used embedded as libraries to build standalone application independent of database server.

The XML processing in Oracle XML DB is based on the XMLType datatype, which is the native datatype in SQL/XML. XQuery can be invoked directly or embedded in SQL/XML to query, update both persistently stored XML documents in tables and XML views over relational data. Since XML in general is falling into data centric XML and document centric XML category, therefore, there is no-one-size-fit-all XML storage/index solution. So Oracle XMLDB supports object relational storage for data centric XML [5], binary XML with XMLIndex for document centric XML [3]. For users who truly want text fidelity, storing XML as CLOB is also available.

Beyond the physical representation for XML storage itself, there are also use cases where XML content is generated dynamically using XQuery to define query XML views over relational data, for example, generation of hierarchical based XML reports over relational data, or RSS data generation from relational data. Furthermore, the SQL/XML standard integrates both XQuery and SQL together so that users can have a dual XML and relational view of the underlying data. That duality allows using XQuery to query relational data and using SQL to query XML by leveraging XMLTable construct in SQL/XML. So the requirements of XQuery processing in Oracle XMLDB are to build a tightly integrated XQuery and SQL engine that can optimize queries over a variety of physical XML storage and view representations while leveraging different indexes and materialized views. To facilitate this, the Oracle XMLDB XQuery processor compiles XQuery and SQL into the same underlying iterator based query execution plans [18] so that the stream-based lazy evaluation model is fully shared and queries are globally optimized across all storage forms [6].

The requirement of XQuery/XQueryP processor, which we refer as XVM (XQuery Virtual Machine) [7], in XDK is expected to work standalone without any prior knowledge of physical XML representation forms. It uses XQDOM API, which is DOM API extended with PSVI and XQDM constructs as logical API to manipulate XML. The implementation of the API can be efficient and scalable depending on the physical XML representation without materializing a DOM in memory object. Users are able to do full-blown XML programming with intensive procedural logics, for example, extensive usage of XQuery modules, user defined XQuery functions, variable assignments, procedural loops etc. When XVM is embedded into Oracle XMLDB database server, it compiles database stored XQuery modules, user defined functions and XQueryP sequential expressions (aka statements) into machine independent byte-code and provides a virtual machine environment to execute it. Pure query expressions are 'pushed down' to the DB XQuery processor to be executed by leveraging database index, materialized views and various join strategies (hash join, merge join in addition to nested loop join) and parallel query processing capability from the DB iterator engine. The 'pushed down' expression results are fetched by XVM in a form of an iterator data object.

3 DB-based Iterator XQuery Processing

3.1 XML Extended Relational Algebra (XERA)

Although in principal, an iterator based XQuery engine can be built from scratch in the Oracle database server, it is actually much more effective to leverage the mature iterator based SQL engine [18] in Oracle database server. This allows us to handle not only pure XQuery but also hybrid SQL/XML query and to do cross-language optimizations between SQL and XQuery handling physcial XML data and index stored relationally [4, 6]. Furthermore, it is important for the XQuery/XQueryP processor to leverage existing SQL compilation and execution infrastructure instead of re-inventing the wheel again. Note, however, we are NOT translating XQuery to SQL, instead, we compile XQuery and SQL into the same underlying compile-time structures and build iterator based execution plan. The challenge is that there is semantic difference between XQuery and SQL so that we need to bridge the semantic gap between the two languages by leveraging SQL extensibilit y framework [8] to derive XML Extended Relational Algebra (XERA) [4]. The key points to support XERA in Oracle database server are described below.

- Add XMLType that models XQuery Data Model (XQDM) as new native datatype in SQL.
- Add new SQL table function that can iterate each XQDM item and node as native SQL iterator.
- Add XQuery type in XQDM run time data so that dynamic type checking is feasible.
- Add XQDM manipulation operators that support XQuery constructs and XQuery Functions and Operators as native XQSQL operators. These XQSQL operators can be executed iteratively under the SQL table function iterator.
- Add new XQDM based user defined aggregates to support aggregate functions over XQDM.
- Define various algebra rules among these new XQSQL operators, aggregators and table functions with existing SQL operators and aggregators so that they can be algebraically optimized when they are composed together.

3.2 Physical XML storage/index independent optimization

The key XML physical storage/index/view independent optimization is described below:

- We do static type analysis of XQuery to eliminate as much dynamic type checking as possible and compile expensive type polymorphic operators into efficient compile-time type determined operators as much as we can.
- Similar to that of SQL view merge [9], we merge nested FLWOR expression in for clauses to its parent FLWOR clause.
- Similar to that of SQL EXISTS/NOT-EXISTS subquery un-nesting to semi-join and anti-join [10], we merge existence and not-existence check based XQuery expression into semi-join and anti-join.
- We perform operator normalization, cancellation and reduction based on algebra rules [11, 6]. This is particularly important to cancel XQDM aggregation with its table function iteration.

3.3 Physical XML storage/index dependent optimization

As XPath traversal is typically the unit of optimization from physical XML layer, we add XQSQL XQPath() operator that processes a sequence of XPath steps [4]. For XML with object relational storage or XML view over relational data, the XQPath() operator is rewritten into joins of the underlying relational tables [11]. For binary XML with XMLIndex, the XQPath() operator is rewritten into path index lookup which becomes the self-joins of the XMLIndex path tables [4]. For binary or CLOB XML storage without XMLIndex, XQPath() can be executed iteratively using XPath ierator. This rewrite and optimization process can be carried inside out through nested SQL query blocks with view merge, subquery un-nesting and operator algebraic reduction [11] (constructor and destructor simplification) and produces native query over the underlying XML storage and index relational tables so that efficient physical join and group processi ng and parallel execution strategies can be explored extensively.

Beyond path traversal leveraging index usage, in general, a XAM based approach of matching XML index or materialized view pattern shall be followed [13]. One of the common XAM patterns is the XPath with branching predicate twig pattern, which we call mater-detail twig pattern, that is commonly used in practices observed from our customer XML usecases. We index such pattern via the XMLTable based structured XMLIndex [12]. Such master-detail twig pattern is evaluated by probing the relational tables constructed by the XMLTable based structured XMLIndex.

4 VM-based procedural XQuery & XQueryP Processing

Contrary to the iterator-based stream evaluation of XQuery, XVM treats XQuery and XQueryP as general programming languages. XVM compiler (XCompiler) compiles XQuery expressions into a set of RISC style virtual machine byte-code instructions that compute the result of an expression from each of the sub-expressions in bottom up fashion with the help of stack where the results from sub-expressions are pushed and operands are popped. During run time, a virtual machine environment is created to run the byte code.

4.1 XVM Instructions

XVM instructions are classified into the following groups based on their tasks:

- **XPath step** instructions

  Execution of these instructions calls the proper node navigation methods in the XQDOM interface.

- **XML node construction** instructions

  Execution of these instructions calls the proper node construction methods in the XQDOM interface.

- **Arithmetic and Comparison** instructions

  By default, these instructions are type polymorphic, that is, they do arithmetic and comparison based on the type of the operands. However, the XCompiler can generate non-polymorphic instructions when XCompiler can determine the types of these operands via static type analysis.

- **Data transfer (load, store, push, pop)** instructions

  These instructions move XVM sequence objects between XVM main and context stacks.

- **Type checking and type conversion** instructions

  These instructions implements XQuery run time type checking and value casting.

- **Control transfer (branch and loop)** instructions

  These instructions move XVM sequence objects between XVM main and context stacks.

- **Function call** instruction

  These instructions call XQuery functions. Both built-in XQuery functions and operators and user-defined functions are invoked this way.

- **Iterator-executer-function-call CISC** instruction

  This instruction pushes down XQuery expression to be evaluated by iterator-based XQuery processor.

4.2 XVM Execution Model

The XVM execution architecture is quite simple. There is a set of functions, one for each instruction, implementing the instruction semantics. The XVM main loop moves the instruction pointer over byte-code instructions and calls the corresponding function. The default instruction pointer step is one instruction. Only instructions like 'branch' or 'call' can change the instruction pointer according to their operand values. Each instruction takes its operands from the VM-stack and pushes back the result. When a function is called is activated, the corresponding function stack frame is pushed into the XVM-stack. The frame contains the return address, current stack pointers, current node, a descriptor address plus (if needed) slots for parameters and local variables.

To avoid as much dynamic memory allocation as possible, XVM takes advantage of the nature of stack based expression computing. XVM models XQuery data model items as data objects on the pre-allocated stack. (The stack can grow during run time by dynamically allocating stacks segments, however, the frequency of dynamic memory allocations is significantly reduced). One special kind of the XVM data is the iterator data object, which delivers the data through an iterator interface. Another special kind of data object is the XML node-set object that stores the XML node references. Although XVM uses the XML node reference to perform XQDOM operations, the content of the XML node reference and the implementation of XQDOM interface is completely opaque to XVM. This allows XVM to work with different physical XML forms. When XVM runs inside Oracle database server, there are various optimizations, such as scalable and pageable DOM implementations, to implement the XQDOM interface.

4.3 XQuery Module Handling

XVM supports both static and dynamic linking of XQuery modules. For small size XQuery applications, XVM compiles all modules with the main query body and generate one composite executable byte-code module. However, for large-scale XQuery applications that involve libraries of modules, a dynamic linking mode is used. In this mode, all XQuery modules are compiled separately and their byte-code has header containing tables for imported and exported entities like top-level functions, variables etc. All external references are resolved by name and module id, quite like references in Java classes. In run time, when XVM executes an instruction that refers to unresolved imported entity, it checks if the corresponding module is loaded. If the module is not loaded, the XVM loads it and allocates a table for the module external references. As it was said earlier, the external references are resolved lazily on demand.

5 Integrated Iterator & procedural Processing

5.1 Rationale of integration

There is a trade-off between processing iterator based lazy evaluation model versus procedural oriented eager evaluation model. The lazy evaluation model scales with large data size but does not scale with large program size because the iterator execution tree with all of its intermediate computational states have to be maintained. The eager evaluation strategy scales with large program size but not with large data size because intermediate results have to be materialized. Both of which, however, can be maintained and overflowed to disk if necessary. Eager evaluation strategy is more efficient than lazy evaluation when all intermediate results are needed to determine an answer. However, eager evaluation strategy is sub-optimal if only partial results are needed. Therefore, this results in our unique design principles of combining both eager and lazy strategies to compile and execute XQuery in Oracle XML database server.

5.2 XQuery Expression Push Down

XCompiler compiles sequential XQuery expressions into RISC like XVM instructions, whose execution sequence behaves as classical programming languages where each step finishes its step execution, computes the result and applies changes before executing the next one. However, for non-sequential XQuery expressions, the XCompiler is also able to compile them into an iterator based CISC type of instruction that is associated with an iterator query plan which can then be serialized as part of the data segments of the byte code. When XCompiler is invoked in the Oracle database server to process XQuery, XCompiler invokes DB-based XQuery compiler to decide if any non-sequential XQuery expression fragments can be optimized and executed efficiently considering the physical characteristics of the input XML. As discussed in section 3, the DB-based XQuery processor is able to compile XQuery into XERA and then optimize it based on the physical XML storage and index forms to generate iterato r-based query execution plan.

During XVM execution time, execution of the iteraor-based CISC instruction by XVM first de-serializes the query execution plan that is prepared by the DB-XQuery compiler from byte code data segment, then executes the query plan by calling DB-XQuery executor. As discussed in section 3, the DB XQuery executor is an integrated XQuery-SQL processor that uses index and stream evaluation to efficiently execute the query plan with large XML data sizes. The result of the DB XQuery executor is stored in XVM iterator data object and is consumed by XVM in an iterator fashion.

Therefore, the overall integrated XQuery and XQueryP execution model is that the XVM drives the execution of a sequence of sequential XQuery expressions. Each sequential expression, like an imperative statement, may change the execution environment and cause visible side effects, for example, changing the persistent XML or changing the value of global or local variables. When evaluating each sequential expression, different query fragments of the expression can be pushed down to a DB-based XQuery processor that evaluates the query fragment efficiently by using index, parallel query processing technique that DB-based XQuery processing is specially designed for.

One of the key aspects of XQuery expression pushing down is to make data search and operation as close to that of the data source as possible so that index search can be used and algebraic based constructor and destructor optimization can be applied. However, data search and operation may be separated from the data source access due to the presence of XQuery user defined functions or XQuery variable accesses. Therefore, inlining XQuery user defined function and variable access so that data operation can be optimized with its input data source is critical. However, not every user defined function is inlineable semantically. Furthermore, inlining user defined function call may not always be optimal. Therefore, the XCompiler analyzes the XQuery expressions to see if the inling may produce an optimal plan heustically. It does data flow analysis by starting with the XML input data source expressions and to see if inlining inlineable functions which consume the result of the XML i nput data source expressions can produce an optimal plan. Since the physical XML input data source information is maintained by the Oracle database server, so for each XQuery expression with inlined functions, XCompiler actually invokes DB XQuery compiler to see if the inling is able to produce optimal query plans. If it is, then the inline decision is made so that XQuery expression is pushed down to the DB XQuery processor for evaluation during run time.

Another optimization resulting from function inlining is that a function taking generic parameter type, such as item()* , its body can be optimized when more precised argument type is available during function inline time. This is generally known as function specialization and partial evaluation technique [14]. When a generic function body expression is cloned and substituted with its specific argument expression, more optimization is achieved.

6 Challenges

XQuery and XQueryP processing can naturally leverage many research ideas and techniques from database and procedural programming language processing. However, unlike the past approaches, such as SQL-PSM or Oracle PL/SQL, where the demarcation between query processing logic and procedural programming logic explicitly separated by the language itself, such demarcation is blurred in XQuery/XQueryP. Users have tremendous freedom to write XQuery and XQueryP logic in whatever way they feel is natural for them. It is then up to the compiler optimizer to figure out user's intensions and to find a proper way to optimize and translate them for the target environment [16]. Compiler may decide to convert a sequential expression into a query or a FLWOR expression into a sequential loop statement. For example, users can write procedure loop iterating through the sequence with conditional logic on testing each item of the sequence within the loop. In a case of non-database ta rget environment the compiler will keep the loop as is and apply the typical loop optimizations only. On the other hand a DB-query oriented optimization may convert such a procedural loop into a FLWOR expression that may leverage index to avoid looping each item of the sequence. A recursive XQuery function call that traverses an XML subtree can be optimized into //node() XPath. Sequential looping expression that does aggregation of input items can be optimized into pre-defined XQuery aggregation functions, such as sum(), avg() etc.

For optimizing pure XQuery without user defined function calls and modules, the challenge of leveraging cost-effective XML indexing to process the query is required. In SQL, writing a semantically equivalent query in different ways may result in tremendous performance difference. This depends on how many different ways to express the same query and how sophisticated query normalization, transformation and rewrite techniques the underlying optimizer is equipped with. In XQuery language, number of ways to express an equivalent query is significantly larger compared with SQL. Therefore, it is challenging to build XQuery optimizer that is query form agnostic without user hints.

Handling XQuery user defined function call is another challenge. Traditionally in procedural programming language, user defined function call is fully completed and return value is materialized before returning of the function. This is the most efficient way when the size of the result set is not large. In SQL, the concept of pipelined function [17] is introduced to cope with user defined SQL function that can return a collection of data. The return result of such pipelined function is fetched set at a time through an iterator, effectively streaming evaluation of the function body. However, in XQuery, any user defined function that returns a sequence can be subject to streaming evaluation. Executing every user defined XQuery function in streaming manner causes proliferation of function execution states and closure and is not scalable with respect to program size. Therefore deciding what user defined function shall be executed in streaming fashion is left as an exercise to the optimizer.

7 Conclusion

In conclusion, we believe that the best XQuery processing solution is the one, which finds the right balance between query and procedural optimizations, and we believe that there is still a long way to go till a processor that implements such a solution appears. Meanwhile it is probably beneficial to define different XQuery subsets specialized for efficient use case processing. For example, in backend database settings, XQuery shall be modeled and processed more towards declarative query language whereas in application mid-tier and settings, XQuery shall be modeled and processed more towards imperative programming language. In practice, this also allows users to write much more efficient programs by following best practices to separate data query from data transformations, and verifying that data query execution plans for XQuery executed by the database are optimal. Database queries are expected to find the needle in the haystack and should be index driven where possible. Thi s is particularly important when XQuery is used to locate XML document or fragment within large document collection. Our experiences of supporting XQuery and XQueryP applications have actually shown that following such disciplined approach of separating query from command [19] when writing XQuery and XQueryP programs gives guaranteed predictable performance with improved productivity to users when building large scale XML applications.

References

[1] I. O. for Standardization (ISO), *Information Technology - Database Language SQL - Part 14: XML - Related Specifications (SQL/XML)*.

[2] D. D. Chamberlin, M. J. Carey, D. Florescu, D. Kossmann, J. Robie, *Programming with XQuery*, in XIME-P, 2006.

[3] R. Murthy, Z. H. Liu, M. Krishnaprasad, S. Chandrasekar, A. Tran, E. Sedlar, D. Florescu, S. Kotsovolos, N. Agarwal, V. Arora, V. Krishnamurthy, *Towards an enterprise XML architecture*, in SIGMOD Conference, pp. 953–957, 2005.

[4] Z. H. Liu, T. Baby, S. Chandrasekar, Hui Chang, *Towards a Physical XML independent XQuery/SQL/XML Engine*, in VLDB, pp. 1356–1367, 2008.

[5] R. Murthy, S. Banerjee, *XML Schemas in Oracle XML DB*, in VLDB, pp. 1009–1018, 2003.

[6] Z. H. Liu, M. Krishnaprasad, V. Arora, *Native Xquery processing in oracle XMLDB*, in SIGMOD Conference, pp. 828–833, 2005.

[7] A. Novoselsky, Z. H. Liu, *XVM - A Hybrid Sequential-Query Virtual Machine for Processing XML Languages*, in PLAN-X, 2008.

[8] M. Stonebraker, *Inclusion of New Types in Relational Data Base Systems*, in ICDE, pp. 262–269, 1986.

[9] M. Stonebraker, *Implementation of Integrity Constraints and Views by Query Modification*, in SIGMOD Conference, pp. 65–78, 1975.

[10] W. Kim, *On Optimizing an SQL-like Nested Query*, in ACM TODS, 7(3):443–469, 1982.

[11] M. Krishnaprasad, Z. H. Liu, A. Manikutty, J. W. Warner, V. Arora, S. Kotsovolos, *Query Rewrite for XML in Oracle XML DB*, in VLDB, pp. 1122–1133, 2004.

[12] Z. H. Liu, M. Krishnaprasad, H. J. Chang, V. Arora, *XMLTable Index An Efficient Way of Indexing and Querying XML Property Data*, in ICDE, pp. 1194–1203, 2007.

[13] A. Arion, V. Benzaken, I. Manolescu, *XML Access Modules: Towards Physical Data Independence in XML Databases*, in XIME-P, 2005.

[14] Partial Evaluation: `http://en.wikipedia.org/wiki/Partial_evaluation`

[15] V. R. Borkar, M. J. Carey, D. Engovatov, D. Lychagin, T. Westmann, W. Wong, *XQSE: An XQuery Scripting Extension for the AquaLogic Data Services Platform*, in ICDE, pp. 1229–1238, 2008.

[16] D. Florescu, Z. H. Liu, A. Novoselsky, *Imperative Programming Languages with Database Optimizers*, in PLAN-X, 2006.

[17] Table function: `http://www.psoug.org/reference/pipelined.html`

[18] G. Grafe, *Query Evaluation Techniques for Large Databases*, in ACM Computing Surverys, 25(2):73–170, 1993.

[19] `http://en.wikipedia.org/wiki/Command-query_separation`

Big, Fast XQuery: Enabling Content Applications

Mary Holstege
Mark Logic Corporation

Abstract

Increasingly, companies recognize that most of their important information does not exist in relational stores but in documents. For a long time, textual information has been relatively inaccessible and unusable. Database applications allow relational data to be used and re-used; the architecture of relational database systems allow such applications to function even in the face of large amounts of data. XML [10] and XQuery [8] now allow the creation of a new kind of application that unlocks content in a similar way: a content application. In this paper, we examine the technologies that enable content applications to operate at scale in the context of MarkLogic Server [2].

1 Content Applications

Database applications built on top of relational database management systems use SQL to select specific pieces of data, join them against other data, and reassemble them into new views. It is this flexible, granular reuse of data makes relational databases powerful tools. Relational databases, however, are less useful for dealing with content which is arranged not in regular typed fields but in complex hierarchical documents consisting of running text.

Documents are often described as "unstructured" or "semi-structured" but the problem with documents, from a relational point of view, is not that there is too little structure, but that there is too *much*. Consider a medical document that describes the course of treatment for a patient, with procedures, observations, and actions indicated. Part of such a document, using XML markup, is shown in Figure 1.

The document has sections, which have titles and content. The content has running text which is interspersed with markup for things such as instruments, actions, and observations. Some of these are nested in other markup. The relation of an instrument, say, to the section's content is not simple: the order relative to other entities and other chunks of text is crucial and defines the narrative. While a relational model for this information is certainly possible, it is difficult and loses the narrative coherence of the original. It is easy to represent this narrative structure using XML markup, however. The more semantically rich and detailed the markup becomes, the harder it gets to map into a relational model, and, crucially, the harder it becomes to further enrich the mapped structure.

A content application is to textual content as a database application is to relational data. It uses a query language to select specific pieces of documents, join them against other document pieces, and reassemble them into new documents. A content application goes beyond simple text search ("get me the document that contains the phrase 'important symptom"') into fine-grained selection and assembly based both on full-text operators and

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

```
...
  <section>
    <section.title>Procedure</section.title>
     <section.content>
      The patient was taken to the operating room where she was placed
      in supine position and
      <anesthesia>induced under general anesthesia.</anesthesia>
      <prep>
        <action>A Foley catheter was placed to decompress the
        bladder</action> and the abdomen was then prepped and draped
        in sterile fashion.
      </prep>
      ...
      The fascia was identified and
      <action>#2 0 Maxon stay sutures were placed on each side of
       the midline.
      </action>
      <incision>
        The fascia was divided using
        <instrument>electrocautery</instrument>
        and the peritoneum was entered.
      </incision>
      <observation>The small bowel was identified.</observation>
'''
```

Figure 1: Simple Medical Document

operators over the hierarchical narrative of the document. A query such as "show me all procedures involving important symptoms where no anesthesia occurs before the first incision" entails full-text ("important symptoms, "procedure"), ordering (phrase "important and "symptoms", "occurs before"), and hierarchy (section's content's incisions). A relational model suited to answering such a question would be ill-suited to reconstructing the original narrative (the entire procedure): the reconstruction would be an immensely complex join.

With increasing amounts of content being created natively in XML or being readily convertible to XML the time is ripe for complex scalable content applications built on XML. What that calls for is a language designed for effective XML processing and a system architecture optimized for content.

1.1 Characteristics of Content Applications

Content applications vary widely, but some general trends can be identified:

- Individual documents may to be relatively large. Documents as large as 10 megabytes are common; those running to gigabytes are not unknown.
- The number of documents may also be large, and can run into tens of millions of documents or more. However, the number of documents is usually much smaller than this.
- Content bases are frequently created from large amounts of existing content which needs to be loaded and indexed in bulk.
- In general, update is less frequent than selection, but for Web 2.0 style content applications, being able to add annotations and metadata to the content base is also important. Frequently the content-loading and content-cleaning (update-intensive) and content-access (update-light) phases in the life-cycle of a content application are distinct, so optimization does not need to focus on maintaining fast query during periods of heavy update. This differs from on-line transactional database applications.
- It is important to be able to select small pieces of documents based on full-text, ordering, and hierarchical criteria. Full-text searching brings in linguistic knowledge for stemming, tokenization, and thesauri, as

well as relevance calculations to determine which matches are better than others. Multi-lingual content bases, documents, or even paragraphs are not uncommon.

- Document schemas may be complex, fluid, or unknown. As content applications evolve, new markup may be incrementally added to enrich the content and enable new kinds of queries.
- Content frequently arrives that does not adhere to the defined document structure, in need of clean-up. Clean-up transformations may be very complex, involving hierarchical restructuring.
- Content applications do evolve: to gain business advantages over competitors, to provide more accurate selections or better context for information, to integrate new content sources, and so on. Evolution of a content application often leads to an evolution of the content and vice versa: this in turn implies that flexible schemas and some ability to update content in place are important.
- Content applications are amenable to end-to-end XML processing: content can readily be encoded and stored in XML, XML (especially XHTML) can be directly rendered in a browser, and given an XML-oriented processing language, selected pieces of XML can be operated on to provide appropriate business logic.

Given these general characteristics of content applications, Section 2 reviews some aspects of XQuery that suit it to building content applications, and and Section 3 looks at how the architecture of the MarkLogic Server supports building scalable content applications.

2 Query Language

To enable content applications, the query language needs to understand large, hierarchically structured documents containing human text. It needs to permit highly granular selection based on both the hierarchy and the ordering of children (both of which are highly salient features of content). The XQuery family of specifications allows documents marked up with XML to do just this.

- XML-aware

  XQuery[8] was defined specifically as a query language for XML content. Simple path expressions can be used to select based on XML structure (both hierarchical and sequential) and XML result structures can be easily generated, often using an XML syntax.
- Full-text

  The query language for content applications needs to be able to perform full-text operations, not just string matching, to take into account language-sensitive operations such as stemming and tokenization. The proposed Full-Text extensions to XQuery[9] will provide such functionality.
- Update

  Although content applications typically focus on selection more than updating, the ability to update content or to add annotations or metadata is important to many content applications. Update operations also provide for performing content cleanup and augmentation in place. The proposed Update extensions to XQuery[7] will define update operations.
- Extension Functions

  Depending on the application, specialized capabilities may be required, such as security-related operations, or trigonometric functions. Fortunately, XQuery provides an extension mechanism through function libraries. XQuery itself defines a large selection of built-in functions and operators[6] for basic data manipulation.

- Optimization-friendly

  XQuery is a functional language without side-effects in which most functions are deterministic and stable within a single query, allowing optimizers to freely reorder expressions and avoid recomputing expressions. [1] The proposed update facility uses snapshot semantics which preserves this aspect of the language: the exact order in which updating operations are performed makes no difference to the result of the expression, so an optimizer is free to reorder expressions.

  The XQuery language design also enables lazy evaluation: if a path expression results in a million node sequence, only those nodes in the sequence that make a material difference to the final result need be fetched. Since full-text searches on large content bases can frequently produce such large results, the ability to notice that only the first ten results are being rendered and returned as a result of the overall query can lead to tremendous savings in effort.

- As Typed As You Want To Be

  Another interesting feature of XQuery that is particularly useful for content applications is that, while operations *may* be strongly typed, they *need* not be. An XQuery program can require a variable, function parameter, or return value to be of a specific named type that is declared in an XML Schema, or it can allow it to be anything at all. More flexibility is possible: a "type" constraint could be the requirement that the item be some kind of XML node, or a specific kind of XML node — an element for example — or an element with a particular name, whether defined in an XML Schema or not.

  Content frequently does not arrive perfectly conformant to some schema, and it may evolve over time. It is a great benefit to content applications to be able to use the same tools tools to perform the initial clean-up and evolution of content as are used to process normalized content. XQuery programs with different degrees of typing can be applied at different stages of the process. Alternatively, loosely typed XQuery programs can be used to process content without having to normalize it at all.

3 System Architecture

Query language functionality is only part of the puzzle for enabling content applications. Efficient execution of the query language at scale is important for real-world content applications. The architecture of the MarkLogic Server[2] enables such efficient execution at scale, by optimizing for the characteristics of content applications and taking advantage of the opportunities afforded by the features of XQuery. As [4] and [5] point out, special-purpose databases tuned for particular kinds of problems can easily out-perform general purpose relational databases in their problem domain by factors of 10 or more.

The MarkLogic Server architecture divides processing into two fundamental parts: evaluation and data access. Typically, scaling is accomplished by distributing the evaluation to one set of hosts called E-nodes ("evaluation nodes") and data access to another set of hosts called D-nodes ("data nodes"). The E-node and D-node functionality can be also be combined into a single host. Load balancers and caching proxies can be used to reduce and distribute the load across E-nodes.

3.1 D-nodes

D-nodes store the XML documents along with indexes to enable efficient access to those documents. D-nodes respond to requests from E-nodes to locate, fetch, or update documents under their control.

[1]There are some exceptions, such as the fn:trace() and fn:error() functions, as well as vendor extension functions to perform HTTP requests, compute random numbers, or report execution times, and the like. Optimizers are nevertheless left with a fairly free hand.

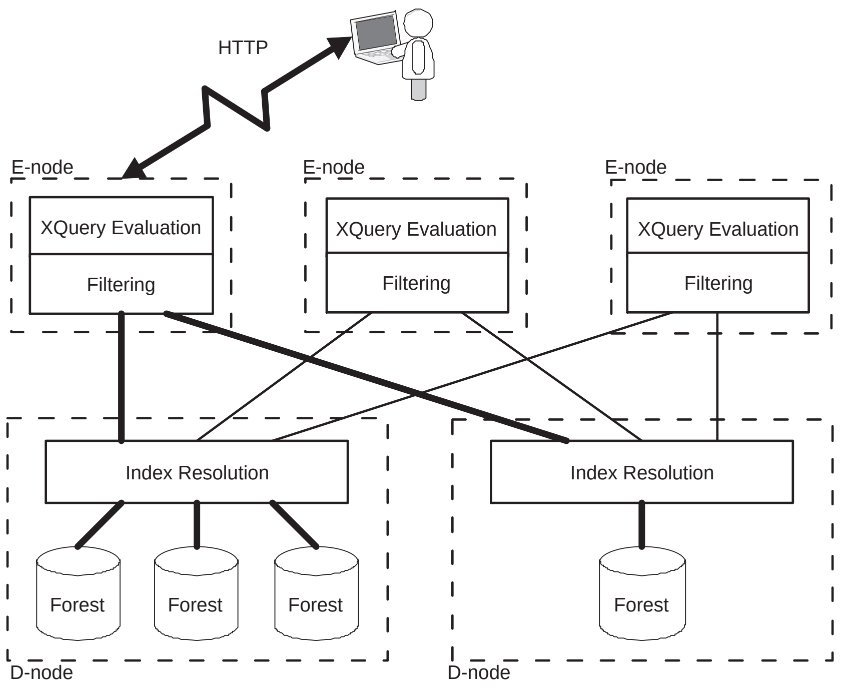


Figure 2: System Architecture

- Fragments

  Documents are broken into non-overlapping units of access called fragments. Fragments are the basic unit of operation in the system. Updates and selection from the databases occurs at the fragment level. Fragmentation choices involve making trade-offs between the expected number of fragments that may need to be fetched and processed to return the correct results of a query, and the size of fragments that must be filtered to produce the correct results or written to disk to process an update. These tradeoffs can be complex and how to balance them is beyond the scope of this paper.

- Forests

  A database may be broken into multiple forests, where each forest is under the control of a specific D-node. Distribution of data across multiple forests allows for greater concurrency and scaling.

- Inverted Indexes

  The forests managed by each D-node include universal indexes that map facts about documents to posting lists. Many kinds of posting lists exist: posting lists for each word, but also posting lists for structural facts, such as the presence of particular elements. The indexes are compressed inverted indexes[11]. Index settings control which specific kinds of posting lists are available in the indexes, and whether the lists record position details or just fragment identifiers.

  XQuery path expressions and full-text queries can be resolved against the indexes by intersections and unions of posting lists for the component facts. Such a result may not be accurate, however. For example, if only a simple word index without positions were available, the phrase query “simple example” could not be accurately resolved in the index. The best the index could do is return fragments that have postings for both “simple” and “example”. A secondary phase, called the filter, is responsible for weeding out the false matches. Index resolution can provide accurate answers in many cases: there is a tradeoff between index resolution accuracy and index size. Again, details of how to balance such trade-offs is beyond the scope of this paper.

- Managing Updates

  The fragments themselves are stored in memory-mapped compressed representations of the document trees. This tree data and the indexes are stored in "stands" in accordance with the principals of a log-structured file system[3]: new content is initially held in memory and then written out in a sequential fashion when a sufficient amount has accumulated. Journalling allows for recovery in the event of a crash. Once written, the tree data and indexes are never updated. Fragments are updated by writing new versions of those fragments to new stands and noting that the fragment in the old stand has been deleted. The server uses multi-version concurrency control[1] to increase the throughput for read operations. Each fragment has a timestamp associated with it. Read operations obtain the most recent version of the fragment with a timestamp preceding the current transaction's timestamp, and therefore always obtain a consistent snapshot of the content base. A periodic merge process creates new stands with any deleted fragments eliminated and the indexes merged to optimize access. This allows both updates and selection to be fast under the normal expected conditions of relatively few active stands and a relatively modest update load once the bulk of the content has been added. Initial loading can also be fast because sequential writes of data in bulk is faster than piecemeal random writes.

3.2 E-nodes

E-nodes are responsible for communicating with clients and for XQuery evaluation: parsing, static analysis, dynamic evaluation, and assembly and serialization of results. E-nodes include HTTP listeners that service requests to execute XQuery modules and return the results.

- Query Processor

  The query processor performs static analysis and rewrite optimization of the query. Any operations that access data are converted into index requests and sent to the D-nodes, with the results being filtered as necessary. The query processor relies on lazy evaluation of node sequences to avoid fetching or processing content unless it is required by the ultimate result of the query.

- Filter

  The filter iterates through the postings returned by the D-nodes and applies the specific match criteria to the selected fragments and returns the requested nodes.

- Application Server

  An E-node also operates as an application server. It accepts HTTP requests on configured ports. In addition to performing conventional serving of documents, requests for XQuery modules are serviced by executing the indicated module and returning the results. Direct execution of XQuery modules enables a rapid development methodology for web applications: XHTML can be generated directly from XQuery for consumption in a browser.

3.3 Summary: Basic Query Flow

Consider a simple query for a phrase within an element as part of a larger query that only makes use of the first ten hits.

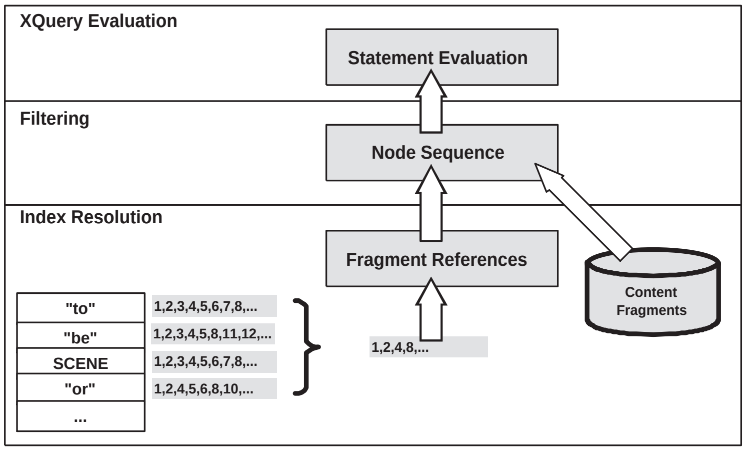


Figure 3: Query Evaluation

1. A client sends an HTTP request to an E-node. The application server accepts the request, locates, parses, and analyzes the appropriate XQuery module. For example: [2]

```
import module namespace my="http://marklogic.com/example"
   at "/MarkLogic/example.xqy";
for $result in cts:search( //SCENE, "to be or not to be" )[
    fn:position() = (1 to 10)
  ] return my:render-result($result)
```

2. The XQuery evaluator constructs an index query to be resolved by the indexes, based on the knowledge of available indexes. In this case, the query parser produces an index request such as:

```
AND(SCENE,"to","be","or","not")
```

3. The indexes combine posting lists to form a sequence of fragment references. Depending on the indexing options, index resolution may return "false positives", fragments identified by the index that do not match original criteria. Each D-node operates in parallel. Index resolution in this case examines the posting lists for the five terms, combines them into a single posting list that has references for all fragments that contain all five terms (fragments 1,2,4,8, etc. in the diagram).

4. The filter turns the sequence of fragment references into a sequence of nodes matching the original criteria. The first fetches each candidate fragment in turn and selects nodes in the fragment that actually meet the criteria (all the words in the phrase appearing in the appropriate order within a `SCENE` element). A fragment containing, for example, the phrase "not to be seen or heard" would be returned from the index resolution, but would not meet the original criteria and would be skipped by the filter.

5. XQuery is evaluated to render the result nodes. Lazy evaluation of the node sequence causes fragments to be fetched and filtered only as needed. In this case the filter only fetches as many candidate fragments are required to return ten `SCENE` element nodes to pass to the `my:render-result` function. If the `ACT` element were the root of the fragment, the entire act would be fetched for filtering, but only the matching

[2]The XQuery Full-Text extension defines the operator `ftcontains` which can be used to test whether a particular node matches some full-text criteria. A common case is to return the sequence of matching nodes, generally ordered by decreasing score. This is what the `cts:search` extension function does.

`SCENE` elements would be returned. If the `ACT` has ten matching `SCENE` elements, only that one fragment would be fetched.

6. The application server constructs an appropriate HTTP response and returns it to the client.

Caches at various level short-circuit some of these operations.

4 Conclusions

The divide between "content" and "data" is not a hard and fast one. However, content applications do tend to have different characteristics than relational database applications. Representing content with XML, operating on it with XQuery, and executing on an architecture optimized for such operations can open up the possibility manipulating large content bases at a fine-grained level to create new and interesting applications. It provides for a middle path between simply identifying documents that meet certain full-text criteria on the one hand, and losing the overall complex hierarchical and narrative flow of documents on the other.

References

[1] Philip A. Bernstein and Nathan Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

[2] MarkLogic Server 4.0, 2008. `http://marklogic.com/product/marklogic-server.html`.

[3] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10:1–15, 1992.

[4] Michael Stonebraker and Uğur Çetintemel. 'One Size Fits All': An Idea Whose Time Has Come and Gone. In *International Conference on Data Engineering (IDCE)*, 2005.

[5] Michael Stonebraker *et al.* One Size Fits All? — Part 2: Benchmarking Results. In *Conference on Innovative Data Systems (CIDR)*, 2007.

[6] Ashok Malhotra *et al.* (editors). XQuery 1.0 and XPath 2.0 Functions and Operators. W3C Recommendation, W3C, January 2007. `http://www.w3.org/TR/2007/REC-xpath-functions-20070123/`.

[7] Don Chamberlin *et al.* (editors). XQuery Update Facility 1.0. Candidate Recommendation, W3C, March 2008. `http://www.w3.org/TR/2008/CR-xquery-update-10-20080314/`.

[8] Jérôme Siméon *et al.* (editors). XQuery 1.0: An XML Query Language. W3C Recommendation, W3C, January 2007. `http://www.w3.org/TR/2007/REC-xquery-20070123/`.

[9] Sihem Amer-Yahia *et al.* (editors). XQuery and XPath Full Text 1.0. Candidate Recommendation, W3C, May 2008. `http://www.w3.org/TR/2008/CR-xpath-full-text-10-20080516/`.

[10] Tim Bray *et al.* (editors). XML 1.0 Recommendation. Fourth Edition, W3C, August 2006. `http://www.w3.org/TR/1998/REC-xml-20060816`.

[11] Justin Zobel and Alistair Moffat. Inverted Files for Text Search Engines. *ACM Computing Surveys*, 38, July 2006.

Experiences with XQuery Processing for Data and Service Federation

Michael Blow, Vinayak Borkar, Michael Carey, Daniel Engovatov,
Dmitry Lychagin, Panagiotis Reveliotis, Joshua Spiegel, Till Westmann [*]

Abstract

In this paper, we describe our experiences in building and evolving an XQuery engine with a focus on data and service federation use cases. The engine that we discuss is a core component of the BEA AquaLogic Data Services Platform product (recently re-released under the name Oracle Data Service Integrator). This XQuery engine was designed to provide efficient query and update capabilities over various classes of enterprise data sources, serving as the data access layer in a service-oriented architecture (SOA). The goal of this paper is to give an architectural overview of the engine, discussing some of the key implementation techniques that were employed as well as several XQuery language extensions that were introduced to address common data and service integration problems and challenges.

1 Introduction

The advent of relational databases in the 1970's ushered in a productive era in which developers of data-centric applications could work more efficiently than ever before. Instead of writing procedural programs to access and manipulate data, declarative queries could accomplish the same tasks. With physical schemas hidden by the relational model, developers spent less time worrying about performance, as physical changes no longer implied program changes. Simplified views could be defined and used with confidence because rewrite optimizations ensured that queries over views are just as performant as queries over base data. The relational revolution was a huge success and led to many commercial database products. Almost every enterprise application developed in the past 15-20 years uses a relational database for persistence, and all enterprises run major aspects of their operations on relationally-based packaged applications like SAP, Oracle Financials, PeopleSoft, Siebel, Clarify, and SalesForce.com.

Today, developers of data-centric enterprise applications face a new challenge. There are many different relational database systems (Oracle, DB2, SQL Server, MySQL, ...) and a given enterprise is likely to have a number of different relational databases within its corporate walls; information about key business entities like customers or employees commonly exists in multiple databases. Also, while most "corporate jewels" are stored relationally, they are often relationally inaccessible because the applications enforce the business rules

[*]Work was done while authors were at BEA Systems, Inc. Current affiliations: Michael Blow, Dmitry Lychagin, and Joshua Spiegel - Oracle Corporation. Vinayak Borkar - Black Titan Software, LLC. Michael Carey - University of California, Irvine. Daniel Engovatov - Stanford University. Panagiotis Reveliotis - Composite Software, Inc. Till Westmann - SAP

and control the business logic. Meaningful access must thus come through the "front door" via application APIs. Because of this, developers of new applications face a major integration challenge: bits and pieces of a given business entity will live in a mix of relational databases, packaged applications, files, legacy mainframe systems, and/or home-grown applications. New "composite" applications need to somehow be created from these parts.

Composite application development is the goal of the service-oriented architecture (SOA) movement [2]. XML-based Web services are one piece of the puzzle, providing physical normalization for intra- and inter-enterprise service invocations and data exchange. Web service orchestration languages [3] are another piece of the puzzle, but are procedural by nature. At BEA, we felt that a declarative approach was needed for creating *data services* [8] for use in composite applications. We chose to ride the wave created by Web services and the associated XML standards, using XML, XML Schema, and XQuery to knit together a standards-based foundation for data services development [9, 10]. The BEA AquaLogic Data Services Platform (ALDSP), introduced in mid-2005, has XQuery in the leading role as the language for accessing and composing information from sources including relational databases, Web services, packaged applications, and files. This paper reviews the ALDSP XQuery implementation and some of the key challenges that we addressed during its development.

2 Background

The types of data models employed by enterprise data sources span from semi-structured to fully-structured, from flat to hierarchical to graph-based, and from untyped to loosely-typed to strictly-typed. For example, relational databases contain structured, flat data while XML documents contain semi-structured, hierarchical data. Some backend sources may require input or provide output in the form of flat, structured data (e.g. stored procedures), or hierarchical, semi-structured data (e.g. Web services). Given the vast heterogeneity found in enterprise data models, a data federation approach should support access to as many different kinds of data sources as possible and employ a rigorous yet versatile data model and type system.

In our approach, the XML data model [11], XML Schema [4, 5], and the XQuery language [13] serve as a solid foundation for integrating diverse data sources. XML provides a flexible way of describing many different types of data representations, while XML Schema offers a standard facility for the formal definition of both simple and complex, hierarchical types. The combination of XML Schema types and the concept of sequence type, specified by the XQuery type system, facilitates the specification of data models that go beyond document types, admitting collections of heterogeneous, arbitrarily shaped data items, and providing additional constructs for advanced usages [12].

XQuery has been specifically designed to query XML documents while paying a lot of attention to many details of XML-centric data processing. XQuery supports both typed and untyped data, focusing on structured as well as semi-structured use cases [14]. The language itself is declarative, enabling many rewriting and optimization opportunities for the compiler and runtime engine, many of which have been extensively researched over the past years (e.g., [6, 7]). XQuery is relatively easy to use, with simple constructs for node construction, XPath-based navigation, and flexible FLWOR expressions for joining and ordering of XML data. While currently focusing on declarative query processing, the language roadmap includes the XQuery Update Facility extension [15], for handling data modifications in a declarative fashion, as well as the XQuery Scripting Extension [16], for imperative programming when strict evaluation order is needed and side-effects may be present. The XQuery language has an active community of users and is gaining adoption across many commercial software vendors. All these factors make it an excellent language choice for building a complex data federation system.

Figure 1 illustrates how a complex data federation problem of assembling a single view of customer information is easily accomplished in an XQuery-capable system. It demonstrates a scenario where the data is assembled from three different data sources: two relational databases containing customer information along with the orders, and a Web service used to obtain the credit rating. Access to relational tables is modeled via

```
declare namespace db_customer = 'urn:CUSTOMER';
declare namespace db_order = 'urn:ORDER';
declare namespace websrv_credit_check = 'urn:CREDIT_CHECK';

declare function getProfile() as element(customer_profile)*
{
  for $customer in db_customer:CUSTOMER()
  return
    <customer_profile>
      <customer_id>{ data($customer/cid) }</customer_id>
      <name>
        <first>{ data($customer/first_name) }</first>
        <last>{ data($customer/last_name) }</last>
      </name>
      <credit_rating>{
        let $ssn := data($customer/ssn)
        return websrv_credit_check:GET_CREDIT_RATING($ssn)
      }</credit_rating>
      <orders>{
        for $order in db_order:ORDER()
        where $order/customer_id eq $customer/cid
        order by $order/order_date descending
        return
          <order>
            <order_id>{ data($order/order_id) }</order_id>
            <date>{ data($order/order_date) }</date>
            <total>{ data($order/total_amount) }</total>
          </order>
      }</orders>
    </customer_profile>
};
```

Figure 1: XQuery example

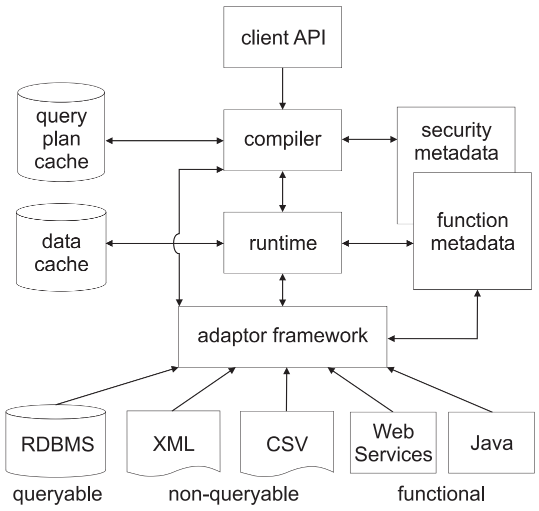


Figure 2: Overview of the ALDSP engine architecture

XQuery function calls (db_customer:CUSTOMER() and db_order:ORDER()), as is a parameterized invocation of the Web service (websrv_credit_check:GET_CREDIT_RATING()). Note that, due to the usage of XML, the result has a natural nested structure, allowing for convenient client data consumption and simple bindings to other programming environments and data models, such as Service Data Objects (SDO) [17] and the Java Architecture for XML Binding (JAXB) [18].

3 XQuery Language Extensions

While our experience has shown XQuery to be an excellent choice for a data federation language, we also found it necessary to extend the language in certain ways in order to support advanced querying capabilities and to make existing features easier to use. This section describes some of the language extensions that have been implemented in ALDSP for these purposes.

• **Metadata.** In ALDSP, enterprise information sources are modeled as external XQuery functions whose actual implementations are transparently provided by the system. Early in the design of ALDSP we were faced with the need to capture and store metadata pertaining to external data sources. The solution we adopted was to extend XQuery prolog declarations with a flexible concept of annotations, which are XML fragments augmenting either an individual function declaration or a whole prolog/module in general [19]. They are defined using "pragma" directives that either precede a function declaration or appear at the beginning of a module/prolog definition:

```
(::pragma name <XML_content/> ::)
```

As the content of an annotation is XML, it can easily hold various kinds of information. One of the usages of annotations in ALDSP is to describe data source binding properties such as relational database connectivity configurations, Web service definition and endpoint locations, delimited file format properties, etc. Over time, ALDSP's usage of annotations has evolved to store many other details of a function/prolog configuration in the product, such as function visibility scope, modeling kinds, update configuration information, and key specifications. In retrospect, this powerful annotation framework minimized the overall number of artifacts in the system and allowed us to quickly introduce new concepts and features as ALDSP evolved.

• **Optional node constructors.** Renaming elements and attributes is a common operation performed in queries that integrate data. In the following example, an XQuery expression is used to rename the customer's "last_name" element to "last", creating a new element with the new name and copying the typed value of the input element:

```
<last>{ data($customer/last_name) }</last>
```

Per XQuery semantics, this expression calls for the construction of an empty element in the event that the input is the empty sequence. But what if the user wants to create the new element with the new name only if the input is non-empty? One can express that logic in XQuery 1.0 as follows:

```
if (exists(data($customer/last_name))) then <last>{ data($customer/last_name) }</last> else ()
```

Given the occurrence frequency of this sort of scenario in data integration use cases, a less verbose approach was required. We extended the direct element and attribute constructors of XQuery with a ? modifier, so the same logic can be expressed as follows in ALDSP:

```
<last?>{ data($customer/last_name) }</last>
```

To optionally create attributes based on the input, one would write

```
<customer last?="{ data($customer/last_name) }" />
```

• **Group by.** Grouping data is an important operation in query processing but, unfortunately, the standard XQuery 1.0 provides no concise way to do so. In our XQuery engine, we added a GROUP BY clause to the FLWOR expression [1]. The following query constructs sequences of customer names grouped by their zip codes.

```
for $customer in db_customer:CUSTOMER()
group $customer as $c-group by $customer/zip_code as $zip
return <group zip="{ $zip }">{ $c-group/last_name }</group>
```

• **XQSE.** Although any computation can be expressed in XQuery, some processing is easier to express in an imperative manner (like in Java, C++, etc). This is also relevant when the steps in a program have side effects beyond the state of the program itself, as XQuery is a side-effect free language. We introduced the XQuery Scripting Extension (XQSE), described in detail in [20], to overcome this limitation of XQuery. XQSE is a proper superset of XQuery based on statements. XQuery expressions can be used anywhere in an XQSE program where an expression is expected. Some of the constructs supported in XQSE are "while" and "iterate" loops, variable assignment with "set" statements, conditional "if" statements, and "try/catch" based error-handling, which is commonplace in popular programming languages.

• **Typing extensions.** The XQuery standard includes an optional feature for statically typing expressions. We found it necessary to extend the XQuery type inferencing rules to meet users' requirements, as requiring query writers to explicitly request revalidation on node construction in order to stay in the typed world was producing a poor user experience. To work around this issue, we implemented a structural form of type inferencing; types in ALDSP are represented by their structure rather than by their schema type name. This is also absolutely essential for view unfolding, which needs to preserve type information through the process of node construction and subsequent drill-down [21].

4 Implementation Techniques

Figure 2 gives an overview of the ALDSP query engine. Queries are submitted for execution through the client API, compiled and optimized, then evaluated by the runtime subsystem, utilizing the adaptor framework for external data source connectivity. Assisting in query processing are metadata context providers, which keep track of various configuration parameters and other properties, as well as caching components for improving overall system performance.

Efficient query execution is crucial in data integration scenarios. Our experience has shown that layers of XQuery functions are quite common in federated data views. In ALDSP, users start with XQuery functions representing physical data sources, then create functions for logical transformations, and finally specialize them for publishing through client APIs. User-defined XQuery functions can be reused in each step during this process, selection predicates can be applied at various layers, and code reuse could potentially result in subparts of a function not being required for a final result. The ALDSP engine performs efficient query evaluation by using standard optimization techniques such as function inlining, unnesting, dead code elimination, and many others [21]. All non-recursive functions are inlined in the beginning of the rewriting process, thus enabling the optimizer to have a global view of the whole query. Subsequent optimization stages rely on this global view to rewrite parts of the plan to a more efficient form, eliminate expressions that were determined to be unnecessary for the result, and choose optimal implementations for runtime operators.

Another important feature of our engine is the inclusion of relational operators in its core XML query algebra. During the query compilation phase, these operators enable well-known relational optimizations such as join reordering, predicate pushdown, transitive condition inference, and many others. At runtime, relational operators are evaluated on tuple streams in a traditional (relational database like) manner. Efficient join processing is vital to overall system performance. The ALDSP query compiler detects inner, outer, and semi-joins patterns in XML queries and the execution engine implements them using well-known join algorithms. When it comes to combining data from relational sources, ALDSP employs a distributed join method internally called *clustered parameter passing join*. It significantly reduces the number of accesses to the underlying database sources and leads to a very efficient query evaluation. Grouping and aggregation operations require special attention in data integration use cases and have always been at the focus of ALDSP query processing. First of all, as described in the previous section, ALDSP introduces an additional "group-by" clause in the FLWR expression, which is backed up by optimizer and runtime support. During query compilation the optimizer may choose to split group-by into two operations: clustering and pre-clustered grouping. Clustering is a weaker form of sorting which may be merged with adjacent order-by clauses or eliminated altogether if the optimizer can prove that the input is already clustered on the required field. The grouping operation is then executed in a streaming fashion on pre-clustered input.

Relational database systems play a central role in the information federation architecture, typically storing most of the enterprise data. For this reason, the ALDSP engine specifically focuses on optimizing database access patterns. We designed and implemented ALDSP's XQuery to SQL translation framework to identify XQuery subexpressions and patterns that can be translated into equivalent SQL queries and pushed down to underlying database sources for native execution. A key feature of the SQL generator is its broad support of different SQL dialects found in modern database systems, which is also customizable by users. The XQuery to SQL translation process is driven by the ALDSP query optimizer. First of all, it relies on the join identification performed in previous optimization stages. Using join blocks in the plan, the optimizer then re-arranges expressions to maximize SQL-able fragments. Finally, there's a SQL text generation stage which emits SQL queries and replaces XQuery fragments with database invocation expressions which will be executed at runtime. The key problem we faced at this stage is how to preserve the semantic equivalence between a generated SQL statement and the actual XQuery expression given by the user. Unfortunately, we found that in some cases preservation may not be possible or may lead to highly suboptimal query execution plans. In these relatively rare cases, the query optimizer is designed to prefer overall query performance over adhering exactly to precise XQuery semantics, while also providing query architects with flexible mechanisms to control which parts of the query are executed by the underlying databases and which are evaluated in the middle tier by the ALDSP engine. An example of such a semantic mismatch is when a database does not properly distinguish between an empty string and a NULL value, or if it has some special rules for string comparison operations on certain character data types.

The major challenge in executing queries efficiently in the middleware is to avoid data materialization, as it usually impacts performance negatively. The ALDSP runtime engine meets this challenge by processing data in a streaming fashion, thus preventing materialization whenever possible. XML data is represented as a stream of small tokens, each corresponding to a part of an XML data item [22]. These tokens flow through the runtime system and are discarded as soon as possible. The ALDSP's internal XML data model extends the XQuery Data Model with support for tuple tokens which serve as typed containers for various data items. Having tuple tokens greatly simplifies implementation of joins and grouping operators, at the same time natively matching relational data obtained from back-end database systems during query execution. In cases when large data sets are unavoidable during query execution, the ALDSP runtime supports such time-tested memory management techniques as external merge sorting and secondary storage join operators.

5 Updates

We now turn our attention to the ALDSP update model. ALDSP's API enables a client to execute a query, operate on the results, and then submit the modified data back to persist the changes. Changes on the client side are transmitted using Service Data Objects (SDO) [17]. On the server side we have extended the XQuery Data Model (XDM) with an SDO-like ability to carry changes. The result, eXtended XDM, or XXDM for short, is a proper superset of XDM in terms of information content. In other words, XXDM can model everything that XDM can model, and it can also model changes to XDM instances.

XXDM nodes share the same data model attributes as XDM nodes (see [11]) and have an additional attribute called "state" which is used to indicate if the node has been changed or not, and if so, how. This state attribute can have one of four values: CREATED, DELETED, MODIFIED, or NONE. A newly created XXDM node has a value of CREATED, a node to be deleted has a value of DELETED, a node that has been modified has a value of MODIFIED, and a node that has not been altered has a value of NONE. Like nodes, atomic values have state as well but their attribute may not have a value of MODIFIED. Modified atomic values are represented by a DELETED value (the old value) followed by a CREATED value (the new value). We use this finer-grained indicator for modification of simple content so that changes in sequences of atomic values can be captured more efficiently.

XXDM is similar, at least abstractly, to the pending update list (PUL) concept in the XQuery Update Facility (XUF) [15]. While conceptually related, the goal of XXDM is different. The PUL is used to explain the semantics of various XUF constructs, and is used only implicitly for that purpose. In contrast, XXDM is a concrete extension to XDM that provides programmatic access to data and changes.

Changes to a result set need to be translated to the underlying data sources, and ALDSP provides the user with two tools for doing this: automatic update maps and XQSE (see Section 3). Update maps are an internally generated description of how to map values from target to source. ALDSP generates them automatically by analyzing the XQuery source for a data service definition and essentially inverting the query. The mapping is described using an internal language that the user can inspect, fix, and augment using a graphical editor. For cases where the update map is insufficient or unavailable, the XQSE scripting capabilities can be used to decompose the changes manually. For this purpose, ALDSP provides a built-in library of mutator functions for working with XXDM instances. XQSE can also be used in combination with update maps, allowing the user to inject complex business logic or error handling without having to hand code the basic "mapping" logic. We refer the reader to [23] and [24] for more information.

6 Conclusion

In this paper we have explained how we utilized XQuery at BEA as the core technology for a modern information integration product (ALDSP, now called ODSI – for Oracle Data Service Integrator). We discussed how we implemented the full XQuery language in that context at BEA, covering some of the techniques used to ensure efficiency and some problems that we faced along the way. Key techniques included the use of efficient and streamable internal data formats, much like those in commercial relational query engines, and a strong focus on delegating query processing to the underlying data containers whenever possible. We also briefly described how ALDSP handles updates. Based on our experiences to date with XML and XQuery, as well as with the diversity of enterprise data sources, we are very optimistic about the future of XML and XQuery as the “right” fit for information integration in the SOA era.

References

[1] V. Borkar, M. Carey, *Extending XQuery for Grouping, Duplicate Elimination, and Outer Joins*, in XML Conference and Expo., Nov, 2004.

[2] M. Huhns, M. Singh, *Service-Oriented Computing: Key Concepts and Principles*, in IEEE Internet Computing (9):2, 75–81, 2005.

[3] M. Singh, M. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*, Wiley, 2005.

[4] World Wide Web Consortium, *XML Schema Part 1: Structures Second Edition*, http://www.w3.org/TR/2004/WD-xquery-20040723/, 28 Oct, 2004.

[5] World Wide Web Consortium, *XML Schema Part 2: Datatypes Second Edition*, http://www.w3.org/TR/2004/WD-xquery-20040723/, 28 Oct, 2004.

[6] V. Braganholo, S. Davidson, C. Heuser, *From XML View Updates to Relational View Updates: Old Solutions to a New Problem*, in Proc. of the Int’l Conference on Very Large Data Bases, 276–287, 2004.

[7] Y. Papakonstantinou, V. Borkar, M. Orgiyan, K. Stathatos, L. Suta, V. Vassalos, P. Velikhov, *XML Queries and Algebra in the Enosys Integration Platform*, in Data & Knowledge Engineering, (44):3, 299–322, 2003.

[8] M. Carey, *Data Services: This is Your Data on SOA*, in Business Integration Journal, Nov/Dec, 2005.

[9] V. Borkar, M. Carey, N. Mangtani, D. McKinney, R. Patel, S. Thatte, *XML Data Services*, in International Journal of Web Services Research, (3):1, 85–95, 2006.

[10] M. Carey, the AquaLogic Data Services Platform Team, *Data Delivery in a Service-Oriented World: The BEA AquaLogic Data Services Platform*, in Proc. of the ACM SIGMOD Int’l Conference on Management of Data, 695–705, 2006.

[11] World Wide Web Consortium, *XQuery 1.0 and XPath 2.0 Data Model (XDM)*, http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/, 23 Jan, 2007.

[12] World Wide Web Consortium, *XQuery 1.0 and XPath 2.0 Formal Semantics*, http://www.w3.org/TR/2007/REC-xquery-semantics-20070123/, 23 Jan, 2007.

[13] World Wide Web Consortium, *XQuery 1.0: An XML Query Language*, http://www.w3.org/TR/2007/REC-xquery-20070123/, 23 Jan, 2007.

[14] World Wide Web Consortium, *XML Query Use Cases*, http://www.w3.org/TR/2007/NOTE-xquery-use-cases-20070323/, 23 Mar, 2007.

[15] World Wide Web Consortium, *XQuery Update Facility 1.0*, http://www.w3.org/TR/2008/CR-xquery-update-10-20080801/, 28 Aug, 2008.

[16] World Wide Web Consortium, *XQuery Scripting Extension 1.0*, http://www.w3.org/TR/xquery-sx-10/, 28 Mar, 2008.

[17] Adams et al., *Service Data Objects For Java Specification*, in Service Data Objects For Java Specification, Ed. 2.1, 2006.

[18] S. Vajjhala, J. Fialli, *The Java Architecture for XML Binding (JAXB) 2.0*, http://jcp.org/en/jsr/detail?id=222, 19 Apr, 2006.

[19] P. Reveliotis, M. Carey, *Your Enterprise on XQuery and XML Schema: XML-based Data and Metadata Integration*, in Proc. of the Int'l. Workshop on XML Schema and Data Management (XSDM), 2006.

[20] V. Borkar, M. Carey, D. Engovatov, D. Lychagin, T. Westmann, W. Wong, *XQSE: An XQuery Scripting Extension for the AquaLogic Data Services Platform* in Proc. of the Int'l Conference on Data Engineering, 1229–1238, 2008.

[21] V. Borkar, M. Carey, D. Lychagin, T. Westmann, D. Engovatov, N. Onose, *Query Processing in the AquaLogic Data Services Platform* in Proc. of the 32nd Int'l Conference on Very Large Data Bases, 1037–1048, 2006.

[22] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. Carey, A. Sundararajan, *The BEA Streaming XQuery Processor*, in The VLDB Journal, (13):3, 294–315, 2004.

[23] V. Borkar, M. Carey, D. Lychagin, R. Preotiuc-Pietro, P. Reveliotis, J. Spiegel, T. Westmann, *XDM + SDO = XXDM: Getting Change Back From XDM*, in Proc. of the Int'l Workshop on XQuery Implementation, Experience and Perspectives <XIME-P/>, 2008.

[24] M. Blow, V. Borkar, M. Carey, C. Hillery, A. Kotopoulis, D. Lychagin, R. Preotiuc-Pietro, P. Reveliotis, J. Spiegel, T. Westmann, *Updates in the AquaLogic Data Services Platform*, in Proc. of the Int'l Conference on Data Engineering, 2009.

Data Aggregation, Heterogeneous Data Sources and Streaming Processing: How Can XQuery Help?

Marc Van Cappellen, Wouter Cordewiner, Carlo Innocenti
XML Products, DataDirect Technologies

Abstract

Software infrastructures and applications more and more must deal with data available in a variety of different storage engines, accessible through a multitude of protocols and interfaces; and it is common that the size of the data involved requires streaming-based processing.

This article shows how XQuery can leverage the XML Data Model to abstract the data physical details and to offer optimized processing allowing the development of highly scalable and performant data integration solutions.

1 Introduction

Data access has always been a hot topic. The variety of interfaces available for querying, creating and updating data is impressive and constantly growing. JDBC, ODBC, ADO .NET are the typical basic interfaces you will deal with when working with relational data sources; but don't ignore also Object Relational Mapping systems, like Hibernate [4] for example. If you need to deal with XML, you will most likely hear about DOM, SAX and StAX interfaces; or maybe object to XML mapping, like JAXB [3], for example. Things get even more difficult when dealing with different data formats, like Electronic Data Interchange messages (EDI) or even flat files. The choice about which data access solution to use in which scenario becomes more complicated when your application needs to deal with multiple, physically varied data sources, which is a typical problem especially when dealing with SOA [1].

SOA (Service Oriented Architecture) [8] has been around for a number of years earning acceptance as a solid approach for systems management - one that allows for the broad reuse of existing software assets, provides a sound architectural model for the federation of disparate IT systems, and supports the automation of abstract business processes via a range of programming paradigms.

But how does data management fit in? Guidelines for service-oriented data access and management techniques are sparse. Those that are available have typically been formulated by SOA experts, not data management experts. As a result, different understandings of the same problems turn into a constant source of confusion and headaches.

Most SOA data management solutions currently in use rely on traditional, well defined interfaces including ODBC, JDBC, OCI, ADO.NET, OLE DB, and others. All of these interfaces share similar concepts, but most of them fail to capture the differences between traditional data access architecture characteristics (tightly coupled,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

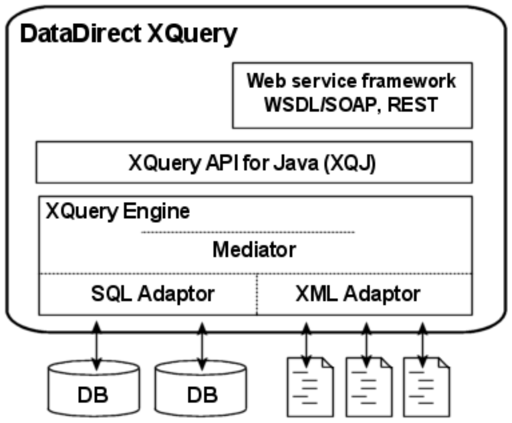


Figure 1: Overview of the DataDirect XQuery engine architecture

complex state machine, connection based, and relational model driven) and characteristics associated with SOA (loosely coupled, stateless, message-centric, and typically XML-based data interchange).

XQuery [10], the XQuery for Java API [6], and Web services [9] provide a great way to bridge the data disparity with service-orientation. XQuery still exposes an interface against which users submit queries and from which they process results, but at the same time it is easily embedded in an application or exposed via a Web service, and it further provides abstraction between the consumer of the data and the physical details about how the data is stored. XQuery is designed to give language implementations the possibility to execute queries against heterogeneous data sources, interpreting (but not necessarily materializing) all of them as XML.

XQuery is based on an XML data model, providing smooth integration in todays Web service-centric infrastructures. When you consider the service-orientation design paradigm, it becomes evident that XQuery features are very much in alignment with the goals of service-oriented computing.

Making XQuery work efficiently against heterogeneous data sources presents peculiar challenges. DataDirect XQuery is an XQuery implementation that was first released in year 2005. DataDirect XQuery's design emphasizes performance and scalability across heterogeneous data sources, with a strong focus on relational data and large XML documents. This paper reviews some of the techniques DataDirect XQuery uses to optimize access to relational and XML data sources.

2 Background

Figure 1 describes the high level architecture of the DataDirect XQuery engine. The engine is accessed through either a Web service framework [2], which allows easy deployment with most application servers, or through a standard API, the XQuery API for Java (XQJ) [6]; that's the interface that can be used to access DataDirect XQuery directly from a client application. The XQuery engine itself is split in three logical components:

- Mediator
  The Mediator is the component that takes care of decomposing an XQuery based on which data sources are being accessed, and of merging the result back from the various data sources.

- SQL adaptor
  The SQL adaptor is responsible for handling the parts of the query that are dealing with relational databases and for pushing the burden of the query to the database itself.

- XML adaptor
  The XML adaptor is responsible for handling the parts of the query that are dealing with XML data

sources, implementing a variety of optimization techniques that allow the process to be highly scalable and performant.

Each of these logical components covers a critical role in making the DataDirect XQuery engine able to deal with heterogeneous data sources. There are several general optimization tasks that are accomplished by the Mediator and that you can probably find described in the context of various XQuery implementations; but some of the most sophisticated optimizations occur at the data source adaptor level. The following sections focus on the specific techniques implemented by the SQL and XML adaptor.

3 XQuery and relational data sources

XQuery and SQL support different operations on very different data models; XQuery works against the XQuery Data Model (XDM) [11], while SQL works against a Relational Data Model; XQuery is designed to make navigation of XML hierarchical structures easy, while SQL focuses more on the task of joining multiple related tables and creating projections on the result. And that's just scratching the surface in terms of differences.

Some XQuery implementations just materialize entire relational tables as XML; others issue the same SQL regardless of the database involved; others yet rely on the least common denominator functionality of the least capable JDBC drivers, which limits performance significantly; and many perform most XQuery functions in the XQuery engine instead of evaluating them in the database.

DataDirect XQuery has been designed to optimize performance and scalability when dealing with all its supported data sources, especially relational databases and large XML documents. Before taking a look at specific XQuery examples and how DataDirect XQuery executes them, let's take a look at the main high level techniques implemented by the SQL adaptor:

- Minimal data retrieval
  Moving data is expensive. In DataDirect XQuery, the generated SQL is as selective as possible, retrieving only the data needed to create the results of a query. It is not unusual that in some cases DataDirect XQuery fetches only part of a single row where other XQuery implementations return an entire table

- Leverage the database
  DataDirect XQuery pushes down into the database operations that can be performed in SQL; that way the relational query optimizer can leverage indexes and other structures. The performance gains this brings are particularly important for joins, Where and Order By clauses, and SQL functions. This also reduces data retrieval, since data need not be retrieved for operations to be done in the database.

- Optimize for each database
  Today's relational databases support significantly different dialects of SQL, and even when two databases support the same operation, their performance may be quite different. Most databases have enough functionality to support XQuery efficiently, but the constructs needed to do this are different for each database. Some XQuery implementations support only one database; others generate the same SQL regardless of the database involved, which results in poor performance. In contrast, DataDirect XQuery uses a different SQL adaptor for each database, generating SQL specifically optimized for that database.

- Support incremental evaluation
  In many applications, results are returned to the user as soon as they are available, displaying the first results well before the entire query has been performed. Many XML applications are based on streaming architectures. DataDirect XQuery uses lazy evaluation so that streaming APIs can retrieve data as soon as it is available. As data is needed, the engine retrieves it incrementally from JDBC result sets. Because there is no need to have the entire result in memory at one time, very large documents can be created.

- Optimize for XML hierarchies
  Because XML construction is hierarchical, DataDirect XQuery uses SQL algorithms that optimize retrieving data for building hierarchies. For instance, the XQuery engine makes extensive use of merge-joins when building hierarchical documents.

- Give the programmer the last word
  Every SQL programmer knows that occasionally hints are needed to get optimal performance for a specific query. This is also true in XQuery, so DataDirect XQuery allows programmers to influence the SQL it generates. This can significantly improve performance in some cases.

The following sections illustrate examples of how XQueries defined against relational databases are translated into SQL by the DataDirect XQuery's SQL Adaptor. Most of the generated SQL shown in this paper is for Oracle 11g - SQL generated for other databases may look significantly different.

The following examples all assume a simple database structure made of two tables, *HOLDINGS* and *USERS*, which contain information about how many and what kind of stock holdings users of the system own.

3.1 Querying data

To minimize data retrieval, DataDirect XQuery generates very selective SQL, returning only the data that is needed for a given XQuery. To avoid retrieving rows that are not needed, the conditions in Where clauses and predicates are converted to Where clauses in the generated SQL. To avoid retrieving columns that are not needed, the generated SQL specifies the columns actually needed to evaluate the XQuery.

3.1.1 Where clause pushdown

Consider the following XQuery, which retrieves all *holdings* for less than 10,000 shares; the XQuery can be easily written in two different ways, one using the Where clause, the other using straight XPath predicates.

```
for $h in collection('HOLDINGS')/HOLDINGS
where $h/SHARES < 10000
return $h
```

```
collection('HOLDINGS')/HOLDINGS[SHARES < 10000]
```

For both XQueries, the SQL query generated by DataDirect XQuery fetches and returns only the rows that are actually to compose the XQuery result:

```
SELECT ALL nrm4."USERID" AS RACOL1, nrm4."SHARES" AS RACOL2, nrm4."STOCKTICKER" AS RACOL3
FROM "MYDB"."HOLDINGS" nrm4
WHERE nrm4."SHARES" < 10000
```

3.1.2 Projection pushdown

The following XQuery retrieves first and last name for each user older than 40. The XQuery is similar to the example shown in the previous section, but this time instead of returning the whole row meeting the selection criteria, the query only needs to retrieve two fields:

```
for $user in collection('USERS')/USERS
where $user/AGE > 40
return <user>{$user/FIRSTNAME, $user/LASTNAME}</user>
```

In this case the DataDirect XQuery engine needs to push to the SQL engine the Where clause, and the selection of two specific columns:

```
SELECT ALL nrm5."FIRSTNAME" AS RACOL1, nrm5."LASTNAME" AS RACOL2
FROM "MYDB"."USERS" nrm5
WHERE nrm5."AGE" > 40
```

3.1.3 Join pushdown

Relational databases are designed to optimize joins, so DataDirect XQuery leverages the database when an XQuery join involves SQL data. Performing all the joins in the database typically results in a dramatic performance gain.

Consider the following XQuery, which retrieve all users and stock holdings for each user:

```
for $u in collection('USERS')/USERS,
    $h in collection('HOLDINGS')/HOLDINGS
where $u/USERID = $h/USERID
return <holding name="{$u/LASTNAME}">{$h/SHARES/text()}</holding>
```

The SQL generated by DataDirect XQuery pushes the resolution of the join operation to the database:

```
SELECT ALL nrm5."LASTNAME" AS RACOL1, nrm9."SHARES" AS RACOL2
FROM "MYDB"."USERS" nrm5, "MYDB"."HOLDINGS" nrm9
WHERE nrm5."USERID" = nrm9."USERID"
```

There are of course multiple ways to express the same join condition in XQuery; for example, in this case the same condition could have been expressed using an XPath predicate, like in:

```
for $u in collection('USERS')/USERS, $h in collection('HOLDINGS')/HOLDINGS[USERID = $u/USERID] return ...
```

DataDirect XQuery is able to capture the multiple ways to express the same queries and it will push down the same SQL.

4 XQuery and XML data sources

While optimizing XQuery when working against relational data sources is mostly a matter of issuing the *best* SQL to the server and to lazily fetch results, when querying XML data sources the XQuery engine needs to deal with the physical task of analyzing and filtering the data. XML data sources include:

- XML documents
- Web service call results (typically SOAP responses)
- Office Open XML (OOXML) or OpenDocument format (ODF) documents
- Comma Separated Value (CSV) files, Tab Delimited files or other flat file formats
- Electronic Data Interchange (EDI) messages streamed to XML

Software architects often tend to underestimate the challenges offered by querying XML documents; that's why often that operation becomes the bottleneck of complex systems. What may start as an application designed to deal with a few relatively small XML documents can easily need to scale up to handle hundreds of XML documents per second, or XML documents that grow to be several Gigabytes in size.

DataDirect XQuery has been optimized to handle data sources in a highly scalable and performant way. The engine's XML adaptor implements several techniques to accomplish that task, like general execution tree optimizations (including function inlining, detecting loop invariants, etc.), in-memory indexing and more; but two major techniques stand out: document projection and streaming processing.

4.1 XML document projection

XML document projection is a clever idea introduced originally by Amelie Marian and Jerome Simeon [7]. The basic idea behind document projection is: given an XML document that represents several details for each *item*, if my XQuery only needs to retrieve/query a couple of attributes for each *item*, why should the XQuery engine materialize in memory the whole *item* elements?

Consider this simple XML document, describing a few objects available for auction:

```
items.xml:
  <ITEMS>
    <ITEM>
      <ITEMNO>1002</ITEMNO>
      <DESCRIPTION>Motorcycle</DESCRIPTION>
      <OFFERED_BY>U02</OFFERED_BY>
      <START_DATE>1999-02-11T00:00:00</START_DATE>
      <END_DATE>1999-03-25T00:00:00</END_DATE>
      <RESERVE_PRICE>500</RESERVE_PRICE>
    </ITEM>
    <ITEM>
      <ITEMNO>1003</ITEMNO>
      <DESCRIPTION>Bicycle</DESCRIPTION>
      <OFFERED_BY>U02</OFFERED_BY>
      <START_DATE>1999-02-11T00:00:00</START_DATE>
      <END_DATE>1999-03-25T00:00:00</END_DATE>
      <RESERVE_PRICE>200</RESERVE_PRICE>
    </ITEM>
  </ITEMS>
```

And now consider this XQuery that retrieves the auction end date for a specific *ITEM* in the XML document above:

```
  for $i in doc('items.xml')/ITEMS/ITEM
  where $i/ITEMNO eq '1002'
  return $i/END_DATE
```

The XQuery only needs two pieces of information for each *ITEM* in the source XML document: *ITEMNO* to resolve the search criteria and *END_DATE* to return the required result. The only parts of the input XML document that are instantiated in memory are the ones highlighted in the following XML fragment:

```
<ITEMS>
  <ITEM>
    <ITEMNO>1002</ITEMNO>
    <DESCRIPTION>Motorcycle</DESCRIPTION>
    <OFFERED_BY>U02</OFFERED_BY>
    <START_DATE>1999-02-11T00:00:00</START_DATE>
    <END_DATE>1999-03-25T00:00:00</END_DATE>
    <RESERVE_PRICE>500</RESERVE_PRICE>
  </ITEM>
  <ITEM>
    <ITEMNO>1003</ITEMNO>
    <DESCRIPTION>Bicycle</DESCRIPTION>
    <OFFERED_BY>U02</OFFERED_BY>
    <START_DATE>1999-02-11T00:00:00</START_DATE>
    <END_DATE>1999-03-25T00:00:00</END_DATE>
    <RESERVE_PRICE>200</RESERVE_PRICE>
  </ITEM>
</ITEMS>
```

DataDirect XQuery statically analyzes an XQuery and generates a *projection tree*; in the example above, the projection tree can be expressed as:

```
  +-step axis="self" test="document-node()"
    +-step axis="child" test="ITEMS"
      +-step axis="child" test="ITEM"
        +-step axis="child" test="ITEMNO"
          +-step axis="descendant" test="node()"
        +-step axis="child" test="END_DATE"
          +-step axis="descendant" test="node()"
```

The projection tree is used in DataDirect XQuery as part of the content handler which processes the XML parser events, ensuring that only the necessary XML parts included in the projection tree are actually materialized in memory.

The process is typically not as simple as the one described in the example above; just think, for example, about the necessary steps needed to handle expressions like *//ITEM* (path reduction) or *../ITEM* (parent axis). But the benefits in terms of performance and scalability when dealing with large XML documents are often impressive, even when *document streaming* (described below) is not available.

4.2 XML document streaming

Processing XQuery in streaming fashion is the ultimate solution in terms of querying XML documents in a scalable way. In the ideal case, when running an XQuery against one (or more) XML document(s) in streaming mode the amount of memory required by the XQuery engine doesn't grow proportionally to the size of the input(s). That allows XQuery to run against XML documents much larger than the physical memory available on a workstation, even when XML document projection can't help.

Consider an XML document similar to the one discussed above in the context of XML document projection, where this time the number of listed *ITEM* elements is in the order of millions. When DataDirect XQuery analyzes the following XQuery, it creates the projection tree and it knows it can avoid materializing in memory several sub-elements for each analyzed *ITEM* element:

```
<MYITEMS> {
  for $i in doc('items.xml')/ITEMS/ITEM
  where $i/OFFERED_BY eq 'U02'
  return
    <ITEM>{$i/ITEMNO, $i/DESCRIPTION, $i/RESERVE_PRICE}</ITEM>
} </MYITEMS>
```

But still, there is a large amount of information that would need to be stored in memory to execute the XQuery in a traditional manner (with no streaming processing); and the amount of required memory would indeed be proportional to the size of the input XML document. Thanks to the XML document streaming technique, DataDirect XQuery is able to process the XQuery described above in streaming fashion, which means that only one *ITEM* per time is actually materialized in memory and discarded when no more needed.

It's worth noting that XML document projection and streaming are two complementary implementation techniques, which implies that when an XQuery is processed in streaming fashion, XML document projection still takes place, limiting the amount of data temporarily materialized by the streaming engine.

When document streaming is used in conjunction with one of the streaming interfaces to consume the result, like StAX [5] for example, which is supported by the XQuery API for Java standard, the whole XQuery processing works in a purely streaming fashion, with the XQuery engine consuming parts of the input XML document on demand based on the way the client application is consuming the XQuery result.

Thanks to these XQuery processing techniques, applications are able to process XML documents in the range of several dozens of Gigabytes without incurring in scalability issues.

5 Mixing data sources together

In the previous sections we have discussed several techniques implemented by DataDirect XQuery to optimize processing of XQuery when working against relational or XML data sources. But it is common that applications need to deal with data which is not available in a single format; and that's the context where dealing with a single query language, data model and interface which covers heterogeneous data sources becomes fundamental.

Think about a scenario, for example, where a list of auctioned *ITEM*s is available in an XML document, as described in 4.1, but details about the person who's offering the *ITEM* are available in a *USERS* table hosted on a relational database, including information about the user id, name, address and email. Now think about the need of creating an application that given a user's email address retrieves all the items that are being auctioned by that user.

Thanks to XQuery, that task can be solved by a single, simple query. Note how the XQuery author doesn't need to worry about different data models or different interfaces to the underlying physical data sources:

```
<ITEMS> {
  let $user := collection("USERS")/USERS[USERID = "U02"]
  for $i in doc('items.xml')/ITEMS/ITEM
  where $i/OFFERED_BY eq $user/USERID
  return
    <ITEM>{$user/NAME, $i/ITEMNO, $i/DESCRIPTION, $i/RESERVE_PRICE}</ITEM>
} </ITEMS>
```

Thanks to the optimization techniques discussed above about how DataDirect XQuery handles relational and XML data sources, the query above will take full advantage of the performance capabilities of the database engine hosting the *USERS* table, and of the document projection and streaming processing in dealing with the *items.xml* XML document.

The application consuming the result is shielded from the physical origin of the data returned by the XQuery; even if the result mixes information stored in a relational database and in an XML document, the client application doesn't need to know about that, and it is able to access the returned data through the standard interfaces exposed by the XQuery API for Java.

6 Conclusions

In this paper we have discussed how XQuery can be useful in providing data services which accomplish data integration tasks across heterogeneous data sources. In order to succeed in that task, XQuery implementations must be optimized to deal with the peculiarities of the various supported data sources. DataDirect XQuery implements a variety of techniques when dealing with relational databases and XML documents; those include the ability to push SQL to the relational engine, to minimize the amount of data retrieved from the database, to leverage XML document projection and XML document streaming to handle large XML documents in an efficient and scalable way. Thanks to these techniques XQuery is an excellent technology for simplifying and streamlining data access in the context of traditional and SOA applications.

References

[1] Jason Bloomberg and John Goodson. Best Practices for SOA: Building a Data Services Layer. *SOA World Magazine*, May, 2008.

[2] DataDirect Technologies. DataDirect XQuery Web Service Framework. *http://www.xquery.com/examples/web-service-example/xquerywebservice/*.

[3] S. Vajjhala and J. Fialli. The Java architecture for XML binding (JAXB) 2.0. *http://jcp.org/en/jsr/detail?id=222*.

[4] Red Hat Middleware, LLC. Java Persistance with Hibernate. *http://www.hibernate.org/397.html*.

[5] Java Community Process. JSR 173: Streaming API for XML (StAX). *http://jcp.org/en/jsr/detail?id=173*.

[6] Java Community Process. JSR 225: XQuery API for Java (XQJ). *http://jcp.org/en/jsr/detail?id=225*.

[7] A. Marian and J. Simeon. Projecting XML Documents *Bell Laboratories*, France, 2003.

[8] M. Huhns and M. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 1(9):75–81, 2005.

[9] WWW Consortium. Web Services. *http://www.w3.org/2002/ws/*.

[10] WWW Consortium. XQuery 1.0: An XML Query Language. *W3C Recommendation*, 23 Jan 2007.

[11] WWW Consortium. XQuery 1.0 and XPath 2.0 Data Model (XDM). *W3C Recommendation*, 23 Jan 2007.

Ten Reasons Why Saxon XQuery is Fast

Michael Kay
Saxonica Limited
Reading, Berks, UK
mike@saxonica.com

Abstract

This paper describes the internal features of the Saxon XQuery processor that make the most significant contribution to its speed of execution. For each of the features, an attempt is made to quantify the contribution, in most cases by comparing performance achieved when the feature is enabled or disabled.

1 Introduction

Saxon [1, 2] is an implementation of XQuery written in Java. It implements the XQuery 1.0 specification [3] in full, with the exception of the static typing feature (see [3], section 5.2.3), but including support for schema-aware processing. It also implements the XQuery Update specification [4], which is currently a W3C Candidate Recommendation.

Saxon also implements XSLT 2.0 [5], XPath 2.0 [6], XML Schema 1.0 [7], and a significant subset of the new features in the draft XML Schema 1.1 specification [8]. In fact Saxon started life as an XSLT processor, and was later adapted to handle XQuery as well. The two languages are implemented as different syntax front-ends to the same run-time engine; both compilers generate the same code and at run-time there is essentially no knowledge of whether the code originated as XSLT or XQuery.

Saxon is available in several versions. The open-source product, Saxon-B, implements all the mandatory features of the W3C specifications. The commercial version of the product, Saxon-SA, provides additional optional features, including schema processing, schema-aware XSLT and XQuery processing, and XQuery Update, as well as a number of performance-oriented features including a more advanced query optimizer, support for streamed query execution, document projection [9], and Java code generation.

Saxon is released on both the Java and .NET platforms. The code is written in 100% pure Java. The .NET version is created by cross-compiling the Java bytecode into .NET IL code, using the open-source IKVMC cross-compiler [10]. The version described in this paper is Saxon-SA 9.1 on the Java platform, unless otherwise specified.

Saxon has been under development for over ten years, and the size of the code base is now some 180,000 non-comment lines, excluding test material and tooling. The development objectives for Saxon are, in order of priority: **(1)** Rigorous standards conformance; **(2)** Reliability; **(3)** Usability (primarily of interfaces and error messages); and **(4)** Performance.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

While this paper is concerned with performance, it is important to note at the outset that performance goals are never achieved by sacrificing the higher-priority objectives. In practice, while the objectives are sometimes in conflict, it has in nearly all cases proved possible to achieve the required performance without compromising other goals. For an example see [11].

It is not the intention of this paper to compare the performance of Saxon with other XQuery processors. It is impossible to do this objectively when one knows one product much better than the others. A number of papers have been published describing comparative benchmarking of different XQuery processors [12, 13, 14]. Independent benchmarks can be frustrating for a vendor because they exhibit a lack of specialized knowledge on how to get the best possible results from one's own product; also in the case of Saxon, they often use the open-source version rather than the higher-performance commercial version. Nevertheless, the overall conclusion from these independent studies is that Saxon performance, while not always in pole position, is comfortably near the front of the field.

Another problem with benchmarks is that performance is not a one-dimensional objective. Some users are interested in the throughput of a transaction processing workload that handles thousands of small messages per second using the same queries. Others are interested in the elapsed time for processing very large documents. Some users generate queries on-the-fly, in which case query compile time can be as important as execution time. Some workloads are dominated by the cost of parsing source documents, some by serialization of results, others by the computational cost of the query itself. A well-rounded product needs to satisfy all its users, not just to optimize its score in a synthetic benchmark.

2 The Architecture of Saxon

There is no space in this paper to give a detailed account of the internal architecture of the Saxon product. An article [15] was published some years ago, and although it describes the product from an XSLT rather than XQuery perspective, the broad picture remains valid today.

It should be noted that Saxon is not a database product. Its raw material is XML held in unparsed form in filestore, or sent over the wire. This means that Saxon does not have the luxury of maintaining persistent indexes or collecting statistical data for use by its optimizer; it has to take the data as it comes. When a query is schema-aware, Saxon is able to take schema information into account when compiling a query, but the general rule is that queries are compiled with no knowledge of what will be found in instance documents.

Like every other implementation, the Saxon XQuery processor has compile-time and run-time processing phases. Broadly, the compiler works by creating an expression tree as the output of the parsing phase. It then performs type checking, which labels nodes in the tree with the results of static type inferencing, and adds additional operators to the tree to perform run-time type checking or conversion where required. Saxon works on the principle of *optimistic static type checking*, which means that a compile-time error is reported only if the inferred static type of an expression is disjoint with the required type; if the static type overlaps but is not subsumed by the required type, then additional code is generated to perform run-time type checking. Following type-checking, the next phase is optimization; this examines the tree for constructs that can be rewritten and replaced by alternative, hopefully more efficient equivalents. The optimization phase is optional, and where compile-time performance is critical it can safely be omitted or performed less aggressively.

The final optimized expression tree can then be used in two ways: it can be interpreted by the run-time execution engine, or it can be used as input to the Java code-generator. This generates Java byte code to execute the query directly (currently via Java source code as an intermediate form), and the byte code is then executed by the Java VM in the normal way. The byte code, of course, still makes many calls on a precompiled Saxon run-time library.

Saxon does not include its own XML parser; it can work with a variety of third-party parsers (both push and pull). It does however include its own schema processor and validator: close integration between the schema processor and the XQuery engine was considered essential for high performance.

3 Performance Features

In this central section of the paper we examine a number of features implemented in Saxon whose aim is to improve query performance, and we attempt to quantify the impact of each feature.

3.1 The TinyTree and the NamePool

The XML document used as input to a query may be stored in a variety of ways; what these have in common is that they all implement the abstract Java interface `NodeInfo`. `NodeInfo` is essentially at the same level as the abstract XDM model described by W3C [16]; it differs however in that it offers direct support for the thirteen XPath axes (child, descendant, ancestor, following-sibling, etc). This allows each `NodeInfo` implementation to optimize the way it navigates each axis; and in the case of models that create node objects on demand, it also means that nodes are created only where the caller actually requires them, and not for intermediate nodes that end up being skipped.

There are two native implementations of the `NodeInfo` interface in Saxon: the linked tree, which is a conventional "object-per-node" tree structure in which parent nodes contain a list of their children, and the TinyTree, which we will describe in this section. There are also a number of implementations of `NodeInfo` that wrap external object models including DOM [17] (both Java and Microsoft versions), JDOM [18], DOM4J[19], and XOM [20]. A number of vendors integrating Saxon into other applications have written `NodeInfo` implementations to access other data sources.

The TinyTree structure is unashamedly inspired by the DTM model in Xalan [21], though it does not mimic the design at a detailed level. There are also some similarities with Intel's "record representation" [22], though a significant difference is that Saxon's structure represents nodes in the tree, whereas Intel's represents events in the parse stream.

The TinyTree represents a document using six principal arrays of integers. These arrays contain one entry for each node (other than attribute and namespace nodes), and are indexed by node number. They contain respectively: the node kind (for example element, text, comment), the *name code* (see below), the depth in the hierarchy, a next sibling pointer (which for the last sibling points back to the parent node), and two overlaid values which in the case of elements point to the first attribute and the first namespace node, and for other kinds of nodes are pointers to the textual content in a text buffer (or in the case of a whitespace-only text node, a representation of the actual whitespace compressed using run-length encoding). The total size of these six integers is 19 bytes per node. Attributes and namespaces are represented in separate but similar sets of arrays.

Additional arrays are allocated when needed. The first time a reverse axis such as preceding-sibling is used on a particular document, an array containing prior sibling pointers is created and populated. If the document is schema validated, an additional array is allocated to hold the type annotations produced as a result of the validation process (again as integers, using the name code of the type name).

The TinyTree is designed to be compact without sacrificing speed of access. In particular, it avoids the heavy overhead of using one Java object for each node in the tree; instead, `NodeInfo` instances are allocated as transient objects on demand, and are garbage collected when no longer needed. Modern Java VMs make garbage collection of short-lived objects a highly efficient operation. The TinyTree is also optimized for read-only access. It makes it very efficient to compare nodes for document order, a common operation in XPath. This structure does not support XQuery Update; for that, a mutable linked tree must be used.

Names of elements and other nodes are represented using an integer name code, which can be translated into a fully qualified name by reference to the `NamePool` holding all the names, which is used to allocated new codes. The name code contains a unique identifier for the URI/local-name pair in 20 bits, with a further 10 bits used to represent the prefix; this imposes a limit of a million or so URI/local-name pairs, which as far as I know has never caused a problem, and a limit of 1024 distinct prefixes for each URI, which does occasionally cause problems for pathological applications; but we can live with that. The essence of the approach is that the same `NamePool` is used at compile time and at document parsing time, which means that the compiler can generate code that searches for named nodes using an integer comparison rather than a string comparison.

The primary motivation for the TinyTree is to reduce memory occupancy and building time for large documents without sacrificing access speed, while the main driver for the use of integer name codes is to improve the speed of matching nodes by name. We can evaluate both effects by comparing the TinyTree with both the Saxon linked tree (which uses an object per node, but with integer name codes) and with the DOM (which uses an object per node, and string comparison for names.) To do this I took the 100Mb version of the XMark dataset [23], modified to use a namespace to make it more typical, and ran the query `count(/ns:site//ns:from)` against it. This gave the results shown in Table 1:

Table 1: TinyTree performance

	TinyTree	Linked Tree	DOM
Build time	5136ms	7933ms	8332ms
Memory used	327Mb	370Mb	796Mb
Query time	35ms	226ms	10603ms

This was run with whitespace stripped from the tree, which makes a significant difference to the figures. The DOM used was the Xerces implementation bundled in JDK 1.5. It can be seen that although the TinyTree beats the linked tree on both time and space, the most noticeable gain is in search speed.

3.2 Pull/Push Pipelining

Pipelining is well established as an execution strategy for functional languages as well as for relational databases. The essence of the approach is that an operator that nominally takes a sequence as input and produces a sequence as output (for example the filter operator represented in XPath by the syntax A[B]), should read its input one item at a time and deliver its output to the parent operator one item at a time. This is a description of a pull pipeline: it is driven by read operations issued by the ultimate consumer of the data. Equally valid is a push pipeline, controlled using write operations issued by the supplier of the data.

Saxon uses a combination of pull and push pipelines, and choosing the right kind of pipeline at each stage appears to make a significant difference to performance.

Pull pipelines are used primarily for evaluating XPath expressions, that is, when reading from the source document. Push pipelines are used primarily when constructing documents (both the initial source document and the result document), and also when serializing. Saxon's schema validator is a complex push pipeline, as is the XML serializer. This split between pull and push was very natural in an XSLT 1.0 processor, where there is a clean split in which XPath expressions read the input and XSLT instructions write the output. In XQuery (and for that matter in XSLT 2.0), the two kinds of operation can be composed in arbitrary ways. Nevertheless there are two very different kinds of operation; and it remains true that many queries are "single-phase" in the sense that they only read nodes from the initial query input and only write nodes to its final output.

Feeding data from a pull to a push pipeline is easy: a program owning the control loop reads from the first pipeline and writes to the second. Doing the opposite is more challenging. In the absence of a language with intrinsic coroutine support, there can only be one control loop. Two solutions are available: either break the pipeline by building the intermediate sequence in memory, or use multiple threads. Both involve overheads. Saxon uses both techniques, though multiple threads are used only in one very specific situation, to support streamed processing where the source document is not built into an in-memory tree. So one of the main design aims is to use pull and push where appropriate, but while minimizing the need to switch from one to the other.

```
for $i in distinct-values(
          /site/people/person/profile/interest/@category)
let $p := for  $t in /site/people/person
          where  $t/profile/interest/@category = $i
          return <personne>
                 ...
                 </personne>
return <categorie><id>{$p}</id></categorie>
```

Figure 1: The XMark query $q10$

To achieve this, Saxon divides query operators into three categories:

- Simple read expressions are always executed in pull mode. These include path expressions and filter expressions, sequence concatenation, union/intersection, and function calls such as `subsequence`, `insert`, or `index-of`.
- Node constructors are generally executed in push mode: they write events to an output pipeline. This works especially well when the output is sent straight to a serializer; in this situation there is no need to materialize the constructed tree in memory. These instructions are also able to operate in pull mode (to deliver events on demand to a client), but this is only done if the application that fires off the query explicitly asks for the query result in this form.
- Other expressions, notably FLWOR expressions, conditional expressions, and function calls can operate in either push or pull mode. In general they operate in the same mode as their caller, so if they are invoked during tree construction they will push, and if invoked in the middle of a path expression they will pull. This means that a function body may execute in either mode depending on the context of the caller.

How can we evaluate the effectiveness of this strategy? As an illustration, XMark query $q10$ (see Figure 1), after rewriting by the Saxon optimizer to inline the unnecessary variable `$p`, is a classic one-phase query; run with default options it takes 1926ms, but if we force it to run in pull mode it takes 3456ms, largely because the result document is materialized in memory before being serialized. This query contains a FLWOR expression (for `$t`) that is logically inside an element constructor, and is therefore evaluated in push mode. If we artificially force the FLWOR expression into pull mode (by a tweak to the Saxon code), the execution time becomes 2720ms. Forcing the variable `$p` to be materialized rather than being pipelined also affects the performance adversely, this time to 2398ms. These figures should be sufficient to illustrate that the impact of pipelining decisions can be significant, though they do not prove, of course, that Saxon always gets it right.

3.3 Path Expressions

Path expressions in Saxon are evaluated using a nested loop strategy. A path expression such as `x/y/z` finds all the `x` children of the context node; for each of these it finds all the `y` children, and for each of these it finds all the `z` children. In case this seems obvious, it is not the strategy that all products use, and some researchers have expressed surprise that it should perform so well.

Because of pull pipelining, it is actually an inverted nested loop: the client requests the next `z` element, which might cause the next `y` to be found, and so on. Neither the final node sequence delivered by the path expression nor any intermediate results are materialized in memory.

The main optimization carried out by Saxon is to eliminate sorting wherever possible. The semantics require that the results of each “/” operator, and indeed the results of each axis step, are sorted into document order with duplicates eliminated. In practice such sorting is very rarely needed because the nested-loop evaluation in many cases delivers results already sorted and deduplicated. Saxon goes to considerable trouble to avoid unnecessary sorting. Furthermore, even when the evaluation strategy delivers nodes in the wrong order, the consumer of the results might not care: for example given the expression `exists(x//y//z)`, sorting the node sequence will not affect the outcome.

The main aspect of the analysis is determining combinations of axis steps that are "naturally sorted". This is the case for any sequence of child axis steps. It is also true for an expression such as `a/b//c`, but not (perhaps surprisingly) for `a//b/c`. There is no space here to give the rules in detail.

One case that often causes difficulty is a path such as `$x/a/b/c` that starts with a variable reference. Here, if `$x` is a singleton node sequence, or any sequence that is sorted, contains no duplicates, and contains no node that is an ancestor of any other, then the entire path will be "naturally sorted", making sorting unnecessary. This can sometimes be determined by static analysis, but failing this, Saxon generates conditional code to test at run-time whether `$x` is a singleton, and thus avoids the sort in this common case.

For the query `/site//keyword`, which returns around 70,000 nodes on the XMark 100Mb database, eliminating the sort reduces the TinyTree execution time from 107ms to 60ms. When running against a DOM, where sorting into document order is more expensive, the saving is more dramatic: against the 10Mb database, run time reduces from 1300ms to 290ms; for 100Mb, the query does not even complete without this optimization.

3.4 Join Optimization

Saxon-SA optimizes joins by constructing hash indexes and then using them to support fast filtering of indexed sequences. The optimizer does not actually recognize the concept of a join. What it does is firstly, to break up the condition in the `where` clause of a FLWOR expression and distribute it among the input sequences read by the expression, thereby turning them into XPath filter expressions; and then (independently) it identifies filter expressions that are likely to benefit from indexing.

Two kinds of index are used: indexed documents, and indexed variables. Wherever possible, an index is attached to a document node, which allows it to be reused whenever that document is searched, even in a different query. Where this is not possible, the contents of a variable can be indexed: such an index dies when the variable goes out of scope.

Join optimization is widely discussed in the database literature. A significant difference for Saxon is that there are no pre-existing indexes: any index that is required must be created within the query. Nevertheless, impressive savings are possible in the right circumstances. For example, Table 2 shows the performance of XMark query $q9$ against databases of different sizes using Saxon-B (without join optimization) and Saxon-SA (with).

Table 2: Join optimization

	1Mb	10Mb	100Mb
Saxon-B	41ms	3612ms	381543ms
Saxon-SA	3ms	26ms	246ms

It is plain here that Saxon-SA performance is linear while Saxon-B is quadratic.

3.5 Miscellaneous Rewrites

Further compile-time expression rewrites done by the Saxon-SA optimizer include the following:

- Replace `count(X)=0` by `empty(X)`. This takes advantage of the fact that when `X` is pipelined, the latter expression can exit as soon as it sees the first item in the sequence; there is no need to compute the count.

- Constant folding: constant subexpressions are evaluated at compile time.
- Variable inlining: when a variable is only referenced once, and not in a loop, the reference is replaced by the initializing expression
- Function inlining: calls to non-recursive functions of modest size are replaced by the function body. This often enables further optimization of the new expression.
- Loop lifting: expressions within a repeatedly-evaluated subexpression (for example a filter predicate, or the return clause of a FLWOR expression) that do not depend on the loop variables are moved outside the loop, but taking care to ensure that they are not executed if the loop is iterated zero times.
- Global variable extraction: expressions within a function body that do not depend on the function arguments are promoted to global variables.
- Compound if/then/else expressions acting as switch statements, testing the value of one expression against a range of constant values, are recognized and supported by hashing.

The benefits achieved by these rewrites are highly variable. In each case it is easy to find example queries where the rewrite gives an order-of-magnitude improvement. It is less easy to quantify how many queries benefit from each rewrite. Very often these rewrites are most effective in combination: one apparently minor rewrite simplifies the expression sufficiently to enable another more powerful one, in particular, the join optimizations discussed in the previous section.

3.6 Schema-Aware Processing

Schema-aware processing allows a query to be compiled with knowledge of the schema that a source document will conform to.

The major benefits of schema-aware processing are usability and reliability: it enables easier debugging of queries, and increases the likelihood that a query that is put into production with inadequate testing (as many are) will turn out to be bug-free.

The effect of schema-aware processing on performance is in fact mixed. For some applications, the overhead of performing schema validation on the input outweighs any savings achieved through greater intelligence in the query execution plan. There are also cases where manipulating the document as raw text turns out to be faster than processing it as typed content.

An example where schema-aware processing has a negative effect on performance is in XMark query $q11$, which is dominated by the predicate

```
where $p/profile/@income > (5000 * $i)
```

If the attribute `@income` is typed as `xs:decimal`, and if `$i` is also `xs:decimal`, which will happen if the schema for the XMark database is written to use `xs:decimal` for money amounts, then this will involve a decimal comparison; whereas without schema-awareness, the comparison will use double-precision floating point. In Java, on a typical platform, double arithmetic is much faster than decimal, because it is supported in the hardware. A user who is aware of this problem can work around it, but by default, the query will run more slowly.

On the other hand, knowledge of the paths that exist in the source data can sometimes be exploited to great advantage. The XMark benchmark queries tend to be written with full paths, such as

```
let $ei := $site/people/person/creditcard
```

but real users are often less patient, and write

```
let $ei := $site//creditcard
```

Given sufficient type information, Saxon-SA will rewrite the abbreviated path to use the step-by-step form, which can greatly reduce the number of nodes that need to be searched.

With schema-aware processing, the second query takes around 6ms on the 100Mb XMark dataset; without schema-awareness, it takes 51ms. However, schema validation increases the parsing time for the source document from 5s to 15s.

3.7 Streaming

Saxon-SA provides the ability to execute certain queries in streaming mode. This is not done as an automatic optimization, but must be explicitly requested using a pragma. In this mode, simple expressions can be evaluated without first building a tree in memory.

This does not make the query itself run faster, but it saves the cost of building the tree, and of course it enables source documents to be processed that are too large to fit in memory (transforming a 20Gb document has been timed at 50min [24]).

Streaming is a natural extension of pipelining: it pipelines together the operations of parsing and query evaluation, removing the need to materialize the intermediate data, that is, the tree representation of the source document.

For a query such as `count(//person)` on 100Mb of input, the execution time including parsing is around 5s with streaming, 5.6s without. The big difference is that with streaming, memory is reduced from 450Mb to 1.7Mb. So the effect is not so much on the speed of the query, as on its scalability. This illustrates the message that performance cannot be considered a one-dimensional property.

Speed improves greatly when the file is not read to completion. The query `exists(//africa)` on the same data takes just 180ms with streaming, 5.6s without.

3.8 Document projection

Document projection (see [9]) is a technique for building a tree containing only that subset of the source document that is needed to execute a query, as determined by static analysis of the query. As with streaming, the technique is only suitable where the document is being parsed in order to execute one query that is known in advance, but unlike streaming, it works with any query.

At present Saxon never does document projection automatically, only on request. The main reason for this is that the risk of bugs is considered high, since it relies on inferencing about the access paths used by every single construct in the language.

Document projection, like streaming, has more effect on memory usage than on execution time: with XMark, it reduces the tree size by 90% or more for 15 out of 20 queries, but only two are speeded up by more than 25% ($q6$ by 75%, and $q7$ by 95%).

3.9 Java code generation

Saxon-SA offers the option to generate Java bytecode representing the logic of the query, as an alternative to interpreting the query execution plan. (This is currently done indirectly, via generation of Java source.) The generated code may be executed from the command line, via an API, or as a Java servlet. Many operations, of course, are still handled by calls to the run-time library, the same library that the interpreter uses.

The speed-up obtained by compilation is not as great as one might expect: 25% is typical. For XMark (10Mb), the biggest improvement (54%) is to the slowest query, $q11$, from 3344ms to 1541ms. The saving appears to be greatest for queries dominated by arithmetic or string manipulation – simple path expressions show very little improvement over the interpreter. This suggests that an equally effective (and more convenient) strategy might be to do just-in-time compilation of a few selected subexpressions.

I experimented at one time [25] with generating code for path expressions that was committed to a particular tree model such as the TinyTree, rather than working generically on any tree model. The results were not encouraging, so the experiment was abandoned. Part of the reason is that evaluating path expressions is already very fast.

3.10 Methodology

I said I would give ten reasons why Saxon is fast, and the first nine have been technical characteristics of the delivered product. The final reason is deeper, and relates to the engineering discipline used to develop the software. Here are a few lessons learnt from the experience of developing Saxon over a period of ten years:

- Investigate every user-supplied performance problem in depth. There is no better raw material for understanding how the code behaves, and without such understanding there can be no improvement.
- Optimize the code that typical users write, whether it is well-written code or not. Try to educate users on how to write code that works well on your product, but recognize that you will only reach a small minority.
- Never make performance improvements to the code without measuring the impact. If you cannot measure a positive impact, revert the change (easily said, but psychologically very difficult when you've put a lot of effort in). Keep records of what you learnt in the process.
- Avoid performance improvements that rely on user-controlled switches. Most users (including people who publish comparative benchmarks) will never discover the switch exists; of the remainder, a good number will set the switch sub-optimally.
- Remember that every optimization you make to your code is likely to require a substantial investment in new test material, and even then, is likely to result in several new bugs escaping into the field. Do not do it unless the gain is worth it.
- Maintain a set of performance regression tests to ensure that performance gains made in one release are not lost in the next.
- Separately, maintain tests to show that query optimizations are taking place as intended. In Saxon this is done by outputting an XML representation of the query execution plan for test queries, and checking assertions about these plans expressed as auxiliary queries.

For the other nine ways of achieving good performance in Saxon, I have tried to quantify the benefit. For this tenth cause, I am afraid I cannot do so – I do not have anything to compare with.

4 Conclusions

In this paper I have presented ten characteristics of the Saxon XQuery implementation that contribute to its performance, and for most of these, I have attempted to quantify the size of that contribution for some selected queries.

Few of these mechanisms are unique to Saxon; what makes Saxon distinctive is the deployment of a balanced portfolio of techniques to deliver efficient query execution over a variety of user workloads, coupled with a determination to place other qualities of the product (standards conformance, reliability, usability) ahead of raw performance. In a crowded marketplace with over 50 XQuery implementations competing for user attention, I believe it is this balanced approach that has led many users to make Saxon their preferred choice.

References

[1] http://saxon.sf.net/

[2] http://www.saxonica.com/

[3] http://www.w3.org/TR/xquery/

[4] http://www.w3.org/TR/xquery-update-10/

[5] http://www.w3.org/TR/xslt20/

[6] http://www.w3.org/TR/xpath20/

[7] http://www.w3.org/TR/xmlschema-1/

[8] http://www.w3.org/TR/xmlschema11-1/

[9] Amélie Marian and Jérôme Siméon, *Projecting xml documents*, in Proc. of 29th International Conference on Very Large Data Bases, 2003, pp. 213–224.

[10] http://www.ikvm.net/

[11] http://saxonica.blogharbor.com/blog/_archives/2006/8/13/2226871.html

[12] http://ilps.science.uva.nl/Resources/MemBeR/other-benchmarks.html

[13] http://gemo.futurs.inria.fr/events/EXPDB2006/PAPERS/Afanasiev.pdf

[14] http://portal.acm.org/citation.cfm?id=1324679

[15] http://www.ibm.com/developerworks/library/x-xslt2/

[16] http://www.w3.org/TR/xpath-datamodel/

[17] http://www.w3.org/DOM/

[18] http://www.jdom.org/

[19] http://www.dom4j.org/

[20] http://xom.nu/

[21] http://xml.apache.org/xalan-j/dtm.html

[22] K. Jones, J. Li, and L. Yi, *Building a C++ processor for large documents and high performance*, in Extreme Markup Languages, 2007.

[23] http://www.xml-benchmark.org/

[24] http://saxonica.blogharbor.com/blog/_archives/2007/9/25/3252121.html

[25] http://saxonica.blogharbor.com/blog/_archives/2006/7/24/2157486.html

25th IEEE International Conference on Data Engineering (ICDE 2009)

29 March – 4 April, 2009 Shanghai, China

Data Engineering refers to the use of engineering techniques and methodologies in the design, development and assessment of information systems for different computing platforms and application environments. The **25th International Conference on Data Engineering** provides a premier forum for sharing and exchanging research and engineering results to problems encountered in today's information society. The conference programme will include research papers on all topics related to data engineering, including but not limited to:

Approximation and uncertainty in databases	**Social information management, annotation and data curation**
Probabilistic databases	**Query processing and query optimization**
Data integration	**Database tuning, and autonomic databases**
Metadata management and semantic interoperability	**Scientific, biomedical and other advanced applications**
Data mining and knowledge discovery	**Spatial, temporal and multimedia databases**
Data privacy and security	**Transaction and workflow management**
Data streams and sensor networks	**Ubiquitous, mobile, distributed, and peer-to-peer databases**
Data warehousing, OLAP and data grids	**Web data management**
Database user interfaces and information visualization	**XML data management**
Personalized databases	**Database architectures**

Accepted contributions at ICDE 2009 will make efforts (1) to expose practitioners to the most recent research results, tools, and practices that can contribute to their everyday practical problems and to provide them with an early opportunity to evaluate them; (2) to raise awareness in the research community of the difficult data & information engineering problems that arise in practice; (3) to promote the exchange of data & information engineering technologies and experiences among researchers and practitioners; and (4) to identify new issues and directions for future research and development in data & information engineering.

AWARDS

An award will be given to the best paper submitted to the conference. A separate award will be given to the best student paper. Papers eligible for this award must have a (graduate or undergraduate) student listed as the first and contact author, and the majority of the authors must be students.

INDUSTRIAL PROGRAM

ICDE 2009 will include an industrial track covering innovative commercial implementations or applications of database or information management technology, and experience in applying recent research advances to practical situations. Papers will describe innovative implementations, new approaches to fundamental challenges (such as very large scale or semantic complexity), novel features in information management products, or major technical improvements to the state-of-the-practice.

PANELS

Conference panels will address new, exciting, and controversial issues, being provocative, informative, and entertaining.

DEMONSTRATIONS

Presented research prototype demonstrations will focus on developments in the area of data and knowledge engineering, showing new technological advances in applying database systems or innovative data management/processing techniques.

TUTORIALS

ICDE 2009 will host tutorials, relevant to the conference topics. Tutorials can be single-session (1.5 hour) or for double-session (3 hour).

WORKSHOPS

The following workshops will be hosted by ICDE 2009:

- **DBRank: Third International Workshop on Ranking in Databases**
- **First IEEE Workshop on Information & Software as Services (WISS'09)**
- **Fourth International Workshop on Self-Managing Database Systems (SMDB 2009)**
- **Management and Mining of UNcertain Data (MOUND)**
- **Modeling, Managing, and Mining of Evolving Social Networks (M3SN)**
- **Second International Workshop on Data and Services Managementin Mobile Environments (DS2ME 2009)**

For more information, visit http://i.cs.hku.hk/icde2009/