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

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

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`