ISSN 2319 – 7595

Volume 2, No.4, July – August 2013
International Journal of Information Systems and Computer Sciences
Available Online at http://warse.org/pdfs/ijiscs01242013.pdf

AN ADVANCED TECHNIQUE FOR RAPID DATA PROCESSING IN CLOUD COMPUTING USING X-QUERY

S.Praveen Kumar
spkmtech@gmail.com,
Dept Of IT,GIT,
GITAM UNIVERSITY

Dr. K.Singh
karansinghgbu@gmail.com,
School Of Information and Communication Technology,
Gautam Buddha University

ABSTRACT

XQuery was designed as a query language for XML data. The goal was to provide the expressive power of a query language like SQL and to support XML-specific operations such as navigation in hierarchical data. From the very beginning, an important feature of XQuery has been the capability to process untyped data. It can be concluded that XQuery tries to combine the features of existing programming languages like SQL, Java, or even PHP. XQuery allows to implement sophisticated applications in a single tier and with a single uniform technology, thereby avoiding impedance mismatches and improving flexibility and customizability. Like SQL, XQuery supports declarative queries and updates and is able to specify bulk queries and updates which are best executed inside a database. XQuery provides a special kind of architectural flexibility in the sense that XQuery runs on all application tiers. It runs in the database layer as it has been implemented by all major database vendors as part of their database products. Furthermore, XQuery runs in the middle-tier

Keywords: X-Query, Cloud Computing, Database, Xml.

1.CLOUD DATABASES

Massive growth in digital data, changing data storage requirements, better broadband facilities and Cloud computing led to the emergence of cloud databases .Cloud Storage, Data as a service (DaaS) and Database as a service (DBaaS) are the different terms used for data management in the

Cloud. They differ on the basis of how data is stored and managed. Cloud storage[1] is virtual storage that enables users to store documents and objects .It is evident that storage plays a major part in the data center and for cloud services. The storage virtualization plays a key part in the dynamic infrastructure attribute of Cloud Computing.

Currently Cloud platforms have very little support for database design related virtualization enhancements. But in future designing databases specific for Cloud especially for private clouds in large enterprises is a sure possibility. In this context the distributed databases are important when you design database applications which need to be delivered using Cloud platform. Cloud database has both financial and security advantages over traditional storage models.

SQL (commonly expanded to Structured Query Language) is the most popular computer language used to create, modify, retrieve and manipulate data from relational database management systems. The language has evolved beyond its original purpose to support object-relational database management systems.

GQL has some similarities with SQL but has very limited query expressions in order to provide for scalable processing. Big Integrator can process queries to such data sources with limited back-end query languages support. The absorber and finalizer for big table data sources know the limitations of GQL and will pre and post-process those operations that cannot be processed by the data sources. For this, Big Integrator generates integrating execution plans containing calls to relational databases, big table data stores, and local operators. The Big Integrator system provides query capabilities over combined cloud-based and relational databases.

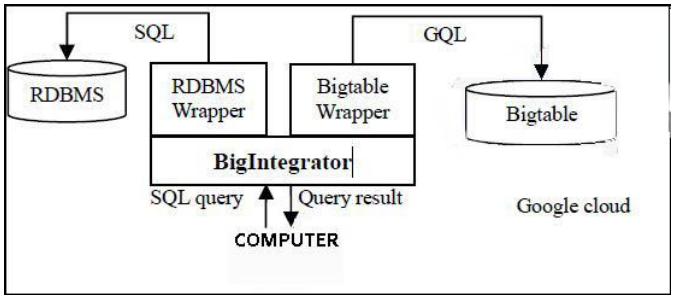

Figure 1: Big Integrator Architecture

In some of the databases[2] xml data is stored and retrieved much more than the normal textual data. Fetching xml data by using sql commands is typical and sophisticated. We use a language called XQuery programming language in order to fetch the xml data specifically.

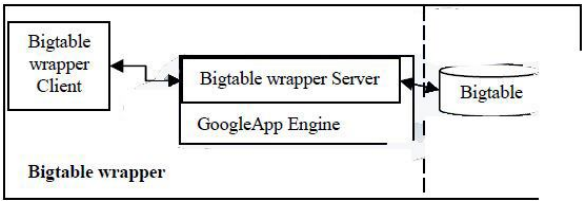

Figure 2: Big table wrapper Architecture

XML is useful because it reduces cost by increasing the flexibility of data management in various ways. Technically, XML is a universal syntax to serialize data. It is universal for two reasons. First, XML is platform-independent; i.e., XML works on all hardware and operating systems. Second, XML is based on UNICODE so that it supports all languages and alphabets. The first kind of flexibility XML provides is to dissociate schema from data. This way, data can exist first and schema can be added later in a pay-as-you-go manner.

The second kind of flexibility arises because XML is able to represent a large spectrum of data, from totally unstructured, semi structured to totally structured data. Furthermore, XML is able to represent data, meta-data, and even code that operate on the data and meta-data. This kind of flexibility has, for example, made XML the data format of choice for configuration data.

All these advantages have led to a wide adoption of XML; clearly, XML is here to stay. However, XML is also heavily criticized and many application developers avoid the use of XML whenever they can. First, XML is perceived to be slow, big, and clumsy. That is, XML data is typically much larger than the equivalent data represented in a proprietary format.

As for XML, the goal of XQuery is to reduce cost. What XML is for the representation of data, XQuery is for the processing of data and development of data-intensive applications . Again, the magic lies in increased flexibility. The first kind of flexibility provided by XQuery is that XQuery operates on any kind of data. Naturally, XQuery is able to process XML data. However, as stated in the previous section, XQuery is just as well able to process JSON, EDIFACT, CSV, or data stored in a relational database. The XQuery processing model specifies that XQuery expressions operate on instances of the XDM data model and these instance can be generated from any kind of data source. Secondly, XQuery inherits all the flexibility provided by XML.

XQuery provides a special kind of architectural flexibility in the sense that XQuery runs on all application tiers. XQuery runs in the database layer as it has been implemented by all major database vendors as part of their database products

2..INTRODUCTION

XQuery is more than ten years old. Its origins go back to the QL 1998 workshop held in Boston. Even though the W3C only recently released the XQuery 1.0 recommendation, the first public working drafts were published in 2001.

Originally, XQuery was designed as a query language for XML data. The goal was to provide the expressive power of a query language like SQL and, in addition, to support XML-specific operations such as navigation in hierarchical data. From the very beginning, an important feature of XQuery has been the capability to process untyped data. Furthermore, XQuery[3] has been designed to support the processing of data on the fly or of data stored in the file system; it is not necessary that the data be stored in a database.

A recent trend which potentially changes the adoption of XQuery is that XQuery is being extended by a number of additional features. These features go beyond message transformation and XML query processing for which XQuery was initially designed. Furthermore, the XQuery Scripting Facility and extended features such as the processing of windows for streaming data are under development with all these extensions. The purpose of this paper is to revisit the advantages of XQuery and clarify some of the myths about XQuery which were created in the early days of XQuery when indeed XQuery was just a query language.

The goal of XQuery is to reduce cost. What XML is for the representation of data, XQuery is for the processing of data and development of data-intensive applications . Again, the magic lies in increased flexibility. The first kind of flexibility provided by XQuery is that XQuery operates on any kind of data. Naturally, XQuery is able to process XML data. However, XQuery is able to process JSON, EDIFACT, CSV, or data stored in a relational database. The XQuery processing model specifies that XQuery expressions operate on instances of the XDM data Model and these instance can be generated from any kind of data

and from any kind of data source. Secondly, XQuery inherits all the flexibility provided by XML.

XQuery provides a special kind of architectural flexibility in the sense that XQuery runs on all application tiers. XQuery runs in the database layer as it has been implemented by all major database vendors (e.g., IBM, Microsoft, and Oracle) as part of their database products.

One particularly valuable advantage of XQuery is that XQuery makes it much easier to customize enterprise Web applications. The same application code can be applied to data in different schemas by using XQuery's flexible data model and the "schema-later" approach of XML. For instance, if one variant of the application added a field to a specific business object, then all the existing code is still applicable to the extended (as well as the original) business object. As a result, XQuery and XQuery database are naturally multi-tenant and do not require heavy weight-lifting as is necessary to implement multi-tenancy in relational database systems.

A second advantage of implementing a whole application in XQuery in a single tier is that code for, say, error handling and checking of integrity constraints need not be duplicated across tiers.

Like XML, XQuery is conceived by many to be slow and complicated. One of the goals of the authors of this paper is to address these concerns by building high performance XQuery processors and by providing best practices and examples that demonstrate the power and usefulness of XQuery as a programming tool.

XQuery is a family of recommendations of the W3C. It extends XPath and was co-designed with XSLT 2.0. As a formula, XQuery can be characterized as follows:

XQuery = Query + Update + Fulltext +
Scripting + Streaming + Libraries

XQuery is worth comparing to other programming languages. Database languages such as SQL typically cover the "Query", "Update", and potentially the "Fulltext" and "Streaming" aspects. General purpose programming languages like Java or C# cover the "Scripting" and "Libraries" aspects. XQuery does it all.

One noticeable omission in the XQuery family is a data definition language (DDL) which allows the specification of integrity constraints, the declaration of schemas, and the definition of a physical database design with indexes. SQL, obviously, provides such a DDL and such a DDL is also needed for XQuery applications.

In summary, it can be concluded that XQuery tries to combine the features of existing programming languages like SQL, Java, or even PHP. In this way, XQuery allows to implement sophisticated applications in a single tier and with a single uniform technology, thereby avoiding impedance mismatches and improving flexibility and customizability. Like SQL, XQuery supports declarative queries and updates; XQuery is able to specify bulk queries and updates which are best executed inside a database.

3. .X-QUERY PROCESSING TECHNIQUES

Architecture of an XQuery Processor
The XQuery specification specifies a processing model to evaluate XQuery programs. This processing model prescribes particular operations and interactions, but does not specify how to implement them.

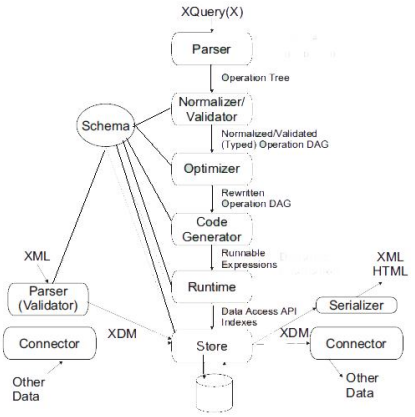

Figure 1 gives a generic architecture that most XQuery processors have adopted. This architecture is also related to the architecture used by most query processors of relational database systems and compilers/runtime systems of general purpose programming languages.

Implementation Variants

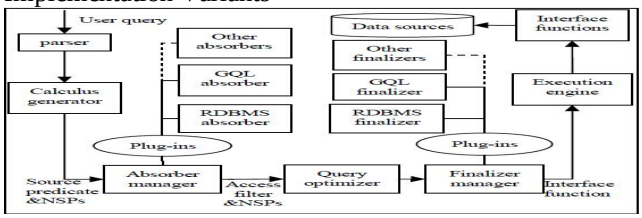

Figure 3: Representing Query Process in Big Integrator

This section gives an overview of implementation variants of theindividual components of an XQuery processor and outlines thecurrent best practices. XQuery canbe used in a wide spectrum of scenarios with varying requirementsand thus different design decisions. In general, most implementationscan be classified in one of the following three categories:lightweight, full, or relational. Lightweight implementations aretypically used for ad-hoc

transformations, embedding in other platformsor scripting. Full implementations are often used in the contextof native XML databases and are more concerned about complianceand index usages, whereas relation engines are XQuery implementationsbased on relational databases. Of course, this classificationmight not fit for all implementations, but it demonstrateswell the design space of implementations.

Given their more traditional (database) workloads, full implementationstypically prefer to use multiple stages and separate representationsof logical and physical plans. Again, rule-based optimization is common practice in this category of implementations.Furthermore, full XQuery implementations provideschema support, as annotating the query plan with schema-derivedinformation allows for optimizations such as general comparisonrewrites. Giventhe complexity of XML Schema, almost all implementations try to build on top of an existing library.
Schema information is particularly important in this regards, hencealmost all relational implementations rely on it and thus providethe necessary support. An extended version of the relational algebrais typically used as logical representation. The extension ismade for the XQuery specific operators such as path expressions.Stating those expressions explicitly simplifies the optimization process.Even though relational implementations try to use their existingcost-based optimizer, the relevant cost models for XQuery havenot yet reached a sufficient degree of maturity. Therefore, even forrelational implementations, most of the XQuery-specific optimizationsare still overwhelmingly rule-based instead of cost-based.

4. OPTIMIZATIONS

Standard XQuery-specific optimizations include join detection, constant folding, avoiding duplicate elimination, document orderingand node identifier operations[4]. Join detection allows replacing nested-loop joins (implied by the XQuery syntax and the ordering constraints) by more efficient join algorithms, which is particularly important for large data sets. Constant folding allows pre-computation of partial results and simplification of expressions.
Duplicate elimination and document ordering are implied by the orderednature of XDM and the semantics of many expressions, mostprominently path and set expressions. Since their implementationstend to be expensive and pipeline-breaking, it is therefore importantto only instantiate them when absolutely necessary. Since theytypically rely on XDM node identifiers, eliminating them also helps in avoiding to generate node identifiers, which is one of the mostexpensive operations at the store level.

Other notable optimizations are the elimination of non-forward axes (parent, pre-sibling etc.) of path expressions and optimizations regarding general comparison. Using only forward axes allows tremendous simplifications and optimizations for the store, in particular enabling streaming execution of data accesses. Optimization regarding general comparison is one of the most performancecriticalfactors. Given the syntax of XQuery, users tend to write general comparisons (e.g, =, <, <=) even though a simple comparision (e.g., eq, lt, gt) would have been sufficient. General comparisons are expensive in two ways: Expressing the type castingand existential quantification required by the semantics of generalcomparison leads to complex implementations that can cost up toorders of magnitudes more than simple values expressions. In addition ,general comparison is neither transitive nor reflexive, thusprohibiting many other optimizations and complicating the use ofindexes. XML Schema often provides the information needed torewrite general comparisons into value comparisons.

5. RUNTIME

Runtime implementations differ mainly in the following techniques ,independent of the targeted use-case: iterator model (pull vs. push), runtime operators (relational vs. XQuery expressions),and internal data model representation (tokens vs. items).Iterator Model. Most runtime implementations follow a pullbased iterator model. Iterators allow for lazy evaluation and streaming execution, so that the runtime [5] can dealwith recursive function calls and data streams - also infinite datain the case of stream queries. Since materialization of intermediate results is avoided, the required memory footprint for processingis minimized. Unfortunately, the depth and nesting of XQueryoperator trees often cause bad cache behavior and a high numberof function calls between iterators. This is particularly bad if afine-grained internal data representation, such as tokens, is used in combination with a lazily evaluated iterator model.Therefore some engines forego the iterator model and compilethe code into native code or their own virtual machine code. Consequently, such runtimes are rather pushbasedand apply a single operation at the time on the whole inputdata set before they forward the complete result to the next operations.

Such an approach increases the cache locality and better utilizesthe pipeline architecture of modern CPUs. On the other hand,this requires materializing the intermediate results, thus increasingthe memory footprint and prohibiting lazy evaluation. Saxon is anexample for an engine positioned between those extremes: It partlycompiles the iterator tree into Java code and mixes push and pulldepending on optimization heuristics. Oracle also performs thismixture of push and pull, using different operator implementationsfor different requirements.

Internal Data Model: Finally, the internal data model representation ,i.e. the representation of the smallest object transferred between the operators, varies from items to tokens.The item representation is closest to the XQueryData Model. Items may represent a [6]singleatomic value such as a string or even a complete XML tree/document, whereas tokens are of smaller granularity and can becompared to (typed) events generated by a SAX parser. Althoughtokens are fast to generate by a suitable parser and allow for lazyevaluation even inside an XML item, they often require invokingevery iterator several times to produce the content of single item of the result set. Hence, a more coarse-grained model, such as items,is often superior to tokens, reducing the required amount of functioncalls to generate a result.

6..GENERAL RUNTIME OPTIMIZATIONS

Independent of the choiceof the internal representation, the iterator model and the runtimeoperators, the optimizations of general comparison, numericaloperations and FLWOR expressions are applicable to all architectures :As for the compiler, general comparison is also an issue insidethe runtime. The optimizer is often not able to substitute generalcomparison by value comparison. As mentioned before, generalcomparison is especially complex because of the applied rules specified in the specification. Thus, spending time to optimize the general comparison quickly pays off. The standard approach is tooptimize the implementation for the common case, i.e, a simple value comparison, with exceptions and fall-back mechanisms forthe less likely truly general comparison situation.

General Design Principles. Certain methods and aspects are common for all types of stores. First of all, since XDM mandatesthat all nodes must have a way to identify them, implementationsof node identifiers need to be provided. It is now common practice to also express structural constraints such as document order and parent/child-relationships in the node identifiers to simplify pathexpressions, set operations, duplicate elimination and document ordering. For updatable stores, Ordpath is the state-of-the-art method, for read-only stores Dewey IDs are used, which bothencode the document structure in a compact way. Therefore, all operationrequiring structural constraints can be supported efficiently.

Generating and maintaining node identifiers is expensive, both interms of computational cost and memory overhead. As outlinedabove, in many use cases, an optimizer can decide to avoid generatingthem.Furthermore, the store is responsible for generating the internaldata representation of XDM. Parsing and object creation have beendetermined as the major cost drivers. This is of particular concernif only fragments of the documents are needed and/or in the case ofone-time transformations and streaming. Hence, document projection, comparable to projection push-downs, is one method tospeed up the processing of document parsing and at the same time minimizing the work for the runtime as much as possible. In the same region of optimizations regarding the store is theuse of object pools and dictionaries for namespaces, elements andstrings. The latter allows performing comparisons based on pointers instead of, for example, the string representation for elementnames.

Usage-specific Implementations: The main differentiatorsfor storage implementations are the usage scenarios and the supportedfunctionality. Since several XQuery engines will cover arrange of usage scenarios and functionalities, they provide multipledifferent store implementations, and choose the most appropriateone depending on the required workload.

Among the different sets of supported functionality, the differencebetween read-only and updatableXDM stores is most important: updatablestores need to provide facilities to support snapshot semanticsrevalidation support iftyped XDM is used, updatable node identifiers and thepossibility to push the XDM updates to external data.Beyond updateable/non-updateable, the store implementations can be categorized in several dimensions: First, stores can be dividedinto in-memory and persistentstores. Second, the storing techniquecan be roughly grouped into binary XML stores relational edge-stores, andhybrid relation/XML binary stores. The storing techniquesitself may be split according to the various ways of shreddinginto relational tables or the different XML binary encodings. Comparing all these approaches is beyond the scopeof this paper.

Indexing: Indexes play a similarly important role for XQuery asfor SQL engines. However, data types and general comparisonoften complicate the use of indexes in queries, especially temporaryad-hoc indexes. It is therefore essential to have a clean indexinterface to provide the necessary information for such optimizations.Three types of indexes are important for XQuery: structural,value and full-text indexes. While full-text indexes are rarely implemented, structural and value indexes can be found in most implementations. Value indexes particularly vary in the way they arecreated. An approach implemented in XQRL indexes certain pathexpressions, thus creating one value index perpath. Alternatively , e.g., in DB2, structural and value indexes are combined. For the actual index structure hash tables or B-Trees are the common approaches.

REFERENCES

[1]. Cloud Computing & Databases by Mike Hogan, CEO ScaleDB Inc.

[2]. Minpeng Zhu and Tore Risch "Querying Combined Cloud-Based and Relational Databases".
[3]. XQuery Reloaded by Roger BamfordVinayakBorkar Matthias Brantner Peter M. Fischer.
[4]. International Journal of Database Management Systems (IJDMS), Vol.3, No.1, Information Retrieval Using Xquery Processing Techniques by E.J.Thomson Fredrick1 And G.Radhamani.
[5]. Implementing an interpreter For fuzzy xquery language Pannipa sae ueng1, wiphadawettayaprasit.
[6]. Fuzzy Logic Based XQuery operations for Native XML Database Systems by E.J.Thomson Fredrick and Dr.G.Radhamani