Executive summary

Keeping pace with rapidly changing requirements is a constant challenge for organizations today. One way to meet this challenge is by storing and processing XML natively in a service-oriented architecture (SOA). Doing so can help enterprises improve application development productivity and quality by eliminating the time-consuming mappings and schema evolution necessary for rapidly changing requirements, thus improving time to market and significantly lowering IT costs.

The breakthrough hybrid database server DB2® 9 for z/OS® integrates XML database technology into relational databases, providing unprecedented scalability and performance for both relational and XML data. The hybrid database server also helps improve performance and scalability by avoiding complex joins commonly seen in an object persistence solution with relational databases. This paper offers an overview of pureXML™ technology in DB2 9 for z/OS, its business value and technical feature details. In addition, for cross-platform development, it lists the commonalities and differences of the XML functionality between DB2 9.5 for Linux®, UNIX®, and Microsoft® Windows® and DB2 9 for z/OS.

© IBM 2006-2008

Table of Contents
Leveraging DB2 9 for z/OS pureXML Technology ........................................................... 1
DB2 9 for z/OS pureXML: an overview ............................................................................. 2
Usage scenarios of DB2 9 pureXML .................................................................................. 4
Business value of DB2 9 pureXML .................................................................................... 5
XML type and native XML storage .................................................................................... 6
Host language interfaces ..................................................................................................... 8
XML data encoding ............................................................................................................. 9
XML indexing ..................................................................................................................... 9
Searching and retrieving XML data .................................................................................. 11
XPath support ................................................................................................................... 13
Constructing XML ............................................................................................................. 15
Consuming web services from SQL .................................................................................. 17
SQL/XML with XPath versus XQuery .............................................................................. 18
Access Methods .................................................................................................................. 19
XML schema repository .................................................................................................... 20
Schema validation .............................................................................................................. 22
Annotated schema-based decomposition .......................................................................... 22
Utilities ............................................................................................................................... 22
Performance monitoring .................................................................................................... 23
Performance and cost factors ............................................................................................ 24
Commonalities and differences from DB2 9.5 for Linux, UNIX, and Windows (LUW) 25
Summary ............................................................................................................................ 26
For more information .......................................................................................................... 26

DB2 9 for z/OS pureXML: an overview
DB2 9 for z/OS XML support lets your applications manage XML data in DB2 databases as conveniently as relational data. You can store well-formed XML documents in their hierarchical form, retrieve all or portions of those documents, and generate new documents. Because the stored XML data is fully integrated into the DB2 database system, you can access and manage the XML data by leveraging DB2 functionality.

To efficiently manage traditional SQL data types and XML data, DB2 uses the existing mature and optimized data management infrastructure for XML storage, including the universal table spaces, the index structure, and the locking scheme. Application developers can use the new XML data type and powerful standard SQL/XML with XPath query language to develop efficient XML applications, while database administrators can use the same familiar utilities and tools to manage physical XML database objects and monitor performance.

XML document storage and retrieval
The XML column data type is provided for storage of XML data in DB2 tables and most SQL statements support the XML data type. This enables you to perform many common database operations with XML data, such as:
- Creating tables with XML columns.
- Adding XML columns to existing tables.
• Creating indexes on XML columns.
• Creating triggers on tables with XML columns.
• Inserting, updating, or deleting XML documents.

Alternatively, a decomposition stored procedure is provided that lets you extract data items from an XML document and store those data items in columns of relational tables, using an XML schema that has been annotated with instructions on how to store the data items.

You can use SQL to retrieve entire documents from XML columns, just as you retrieve data from any other type of columns. When you need to retrieve portions of documents, you can specify XPath expressions, through the standard SQL with XML extensions (SQL/XML). You can also use SQL/XML to construct new documents using relational and XML data.

**Application development**

Application development support of XML enables applications to combine XML and relational data access and storage using SQL. The following programming languages support the new XML data type:

- Assembler
- C and C++ (embedded SQL or DB2 ODBC)
- COBOL
- Java™ (JDBC and SQLJ)
- PL/I

In addition, the following development tools support the DB2 XML data type:

- IBM Data Studio
- Rational Data Architect and Rational Application Developer
- Microsoft .NET
- QMF and DataQuant
- SPUFI
- DB2 for z/OS Command Line Processor (CLP)

You can use these tools to manage XML schemas, generate or edit XML documents, testing queries, as well as construct web services for SQL statements and stored procedures without programming efforts to make DB2 as a web service provider. In addition, SQL statements can invoke DB2-supplied SOAPHTTP functions as web service consumers to use web service results directly in SQL, making web services much easier to consume.

**Database administration**

DB2 for z/OS database administration support for XML includes the following items:

*XML Schema Repository (XSR).*

The XSR is a repository for all XML schemas that are used to optionally validate XML documents stored in XML columns or decompose XML documents into relational tables.

*Utility and replication support.*

You can use DB2 for z/OS utilities and replication tools on XML objects. The utilities handle XML objects similarly to the way they handle LOB objects. For some utilities, you need to specify certain XML keywords.
XML indexing.
Indexing support is available for data stored in XML columns. The use of indexes over XML data can improve the efficiency of queries against XML documents. An XML index differs from a relational index in that a relational index indexes an entire column value, while an XML index indexes part of the data in a column. You indicate which parts of an XML column are indexed by specifying an XML pattern, which is a limited XPath expression.

Performance monitoring.
The accounting traces and performance traces and the same performance monitoring tools can be used for databases with XML data, thanks to the same infrastructure used for XML objects.

Usage scenarios of DB2 9 pureXML
XML data type can be used for the following scenarios:

Processing XML documents directly.
Enterprises are getting increasingly more XML data, in B2B or SOA deployment. The XML data type could be a good way to store what is exchanged, such as industry standard format in financial services (SEPA/UNIFI (ISO20022), ACORD, FIXML, FpML, MIMSO, XBRL, etc.), many other industries and government (DJXDM, HR-XML, HL7, ARTS, HIPAA, NewsML), and XForms, as well as enterprise internal data exchange standards.

Conducting business monitoring.
Efficient XML processing makes it possible to conduct data analysis of intermittent process status, with immediate access to arriving data via XML tools and SQL. XML can be used as the format to load data into data warehouses for business performance monitoring and decision making. Furthermore, with its extreme flexibility, XML can be used as the format to log events for auditing and regulatory compliance. Many different XML indexes can be created on a single XML column to serve diverse XML queries.

Generating XHTML or other format directly.
XHTML can be generated directly from SQL statements to improve performance and simplify systems. On the other hand, XML generated from SQL can also be transformed using XSLT into virtually any format you desire.

Using XML as a versatile data model.
Even when the data you handle is not in XML, you can still take advantage of the flexibility of the XML column type, to support versatile schemas and enable end-user customizable applications, which is very difficult to reflect in a relational schema. You can use an XML column to support object persistence (single XML column v.s. many tables) and avoid mapping of complex XML into many relational tables. XML is particularly good at representing sparse attribute values where there are many attributes but only a few of them are applicable to each case.

Migrating from legacy data model.
XML data type is a much better match for legacy data models, such as hierarchical and network models, compared with relational model. For example, there is no need to
introduce an artificial key when you move a parent-child relationship from hierarchical model to XML, as it is natural to represent that relationship directly.

**Business value of DB2 9 pureXML**

As the first hybrid data server for the industry, DB2 9 lets you store XML data in its pure, native form. Before IBM introduced pureXML technology, there were only a few options to store XML data, including:

*Storing XML as files in file systems.*

Storing XML data as files has the advantage of preserving original documents. But it provides no database ACID (Atomicity, Concurrency, Isolation, and Durability) properties, and other database processing capabilities, such as indexing.

*Decomposing, or shredding XML into relational or object relational form.*

The decomposition approach is commonly used for regularly structured data. It has the advantage of not requiring XML functionality in databases. However, there are many disadvantages:

- Mapping from XML to tables can be complex and fragile, and mapping must be predefined.
- It may need artificial keys to keep the parent-child relationship.
- It could be difficult to reconstruct the XML data, often requiring many joins, with potential poor performance.
- Decomposition typically applies to a single schema, and changes are usually limited. If the schema changes, the changes of the mapping could be tedious, and schema evolution may require database outages.
- Queries are in SQL or through XPath or XQuery to SQL transformation. It is usually less productive in coding, and queries can be difficult to understand, diagnose, and explain.

*Storing XML in large object (LOB) or VARCHAR columns.*

The LOB or VARCHAR storage is the simplest to store XML in a database. For search purposes, commonly searched portions may be extracted into relational tables. It has the advantage of preserving the original documents, and is perfect for XML data that is never searched. However, it has many shortcomings:

- Without additional indexing, XML documents must be parsed for searching every time in either the database server or the client, which is prohibitively expensive.
- When portions are extracted in relational tables with indexes for fast search, it is tedious and inefficient to keep the two in sync when there are updates to XML.
- It is expensive to retrieve portions of a document.

Other vendors also provide LOB-based native XML storage, which stores post-parse binary representation in a BLOB. This approach suffers some disadvantages of being hard to retrieve portions, and inefficient to update. Furthermore, if the binary format needs to be converted to a relational representation for query processing either on the fly or persistently, it could be costly in either processing time or storage. And it becomes prohibitively expensive.
DB2 9 pureXML technology features native hierarchical storage, and native XML operators for query processing. Compared with decomposition or LOB approach, and other vendors’ relational-based technology, it has the following advantages:

**Data model and storage:**

- Offers a compact value-based hierarchical storage.
- Can directly represent flexible hierarchical structures with explicit parent-child relationship.
- Avoids relational normalization and joins that are necessary to re-assemble the normalized tables.
- Provides node-level XML indexing for query performance.
- Delivers schema flexibility with no schema restrictions on XML columns, and therefore, schema can evolve freely.
- Provides a smoother transition and opportunity for best mix by managing both relational data and XML data together.
- Leverages mature existing infrastructure for reliability, availability and scalability.
- Lets DBAs use the same familiar tools to administrate XML objects.

**Query languages and processing:**

- Supports the standard declarative XML query languages SQL/XML and XPath. It provides high productivity in developing applications to process XML data that mapping approaches cannot achieve.
- Delivers native optimized operators and access methods that provide unprecedented performance and scalability.
- Eliminates impedance mismatch between applications and databases when applications are processing XML data to help drastically improve productivity.
- Enables new programming paradigms and software architectures that are heavily based on XML.

Overall, DB2 9 can help significantly improve productivity of XML application development by eliminating tedious mappings and database schema evolution, and improve efficiency in storage and query processing, and offer all the properties a database server provides for XML data. The result is shorter time to market, easier maintenance for rapidly changing business needs, high performance and scalability, and new business insight into otherwise unexploited XML data. The next few sections offer more detailed information on the specific technical features of DB2 9 for z/OS.

**XML type and native XML storage**

DB2 9 introduced XML as a first-class SQL type and enables you to create a table with one or more XML columns. For example, the following statement creates a table with an XML column, POXML:

```sql
CREATE TABLE BASICS.PURCHASEORDERS (  PONUMBER VARCHAR(10) NOT NULL,  PODATE DATE,  POSTATUS CHAR(1),  POXML XML)  IN DATABASE SALESDB;
```
You can also alter an existing table to add one or more XML columns:

```
ALTER TABLE BASICS.PURCHASEORDERS ADD COLUMN INVOICEXML XML;
```

For a table containing one or more XML columns, DB2 adds a hidden column of BIGINT type named `DB2_GENERATED_DOCID_FOR_XML` in the table (called a base table), and creates a separate XML table space and an internal XML table for each XML column. The internal XML table consists of three columns (DOCID, MIN_NODEID, XMLDATA). The XML table space always uses 16 KB page size, and it is a partition-by-growth table space for a simple, segmented, or partition-by-growth base table space, and a partitioned table space for a partitioned base table space. The `XMLDATA` column, with a VARBINARY type, contains the hierarchical storage for XML data model. There are one or more rows in the internal XML table for an XML document depending on the document size. An implicit NODEID index is used to link multiple rows for a same document. The two columns, DOCID and MIN_NODEID, are used for clustering.

You can store well-formed XML documents into an XML column, and there is no XML schema constraint and no length limit associated with an XML column in DB2 9. You can insert an XML document using the INSERT statement with a string literal, a host variable or parameter marker, another column, a file, or an XML expression for an XML value. You can also use the LOAD utility to load XML data.

For example, the following INSERT statement inserts a string literal XML document into an XML column. Notice that XML documents are case-sensitive.

```
INSERT INTO BASICS.PURCHASEORDERS VALUES
('2006040001', CURRENT DATE, 'A',
'<xml version="1.0" encoding="UTF-8">
<purchaseOrder orderDate="1999-10-20">  
  <shipTo country="US">  
    <name>Alice Smith</name>  
  </shipTo>  
  <billTo country="US">  
    <name>Robert Smith</name>  
  </billTo>  
  . . .  
  </purchaseOrder>
',  NULL);
```

During the INSERT or LOAD process, XML values in string format are parsed and converted into the internal representation for storage. By default, insignificant whitespace is stripped during this process. If you need to preserve whitespace in a document, you need to invoke `XMLPARSE()` function explicitly and specify the `PRESERVE WHITESPACE` option as follows:

```
INSERT INTO BASICS.PURCHASEORDERS VALUES
('2006040001', CURRENT DATE, 'A',
XMLPARSE(DOCUMENT CAST(? AS CLOB(100K)) PRESERVE WHITESPACE),
```
In case you need to preserve some of the whitespaces but not all, you need to use an attribute called `xml:space` with a value “preserve” on the elements in your document that you want whitespace to be preserved, which is a W3C XML standard mechanism, while using `strip whitespace` option for the parsing.

You can update an XML column with a new document. For example, the following `UPDATE` statement replaces an existing purchase order with a new one and stores an invoice at the same time.

```sql
UPDATE BASICS.PURCHASEORDERS SET
    POXML = :new_poxml, INVOICEXML = :invoicexml
WHERE PONUMBER = '2006040001';
```

You can also delete a row with XML just as a regular column. For example, the following `DELETE` statement deletes a row with the given PONUMBER.

```sql
DELETE FROM BASICS.PURCHASEORDERS
WHERE PONUMBER = '2006040001';
```

For searched `UPDATE` and `DELETE`, you can specify both relational predicates and XML predicates.

DB2 9 invokes z/OS XML System Services for high-performance parsing. You need to use z/OS R1.8 or later, or z/OS R1.7 with the PTF for XML System Services (APAR OA16303) installed.

You can also validate a document against an XML schema before insertion.

### Host language interfaces

DB2 9 provides XML host language interfaces for Assembler, C or C++ (embedded SQL or DB2 ODBC), COBOL, Java (JDBC or SQLJ), PL/I, and .NET. All the interfaces use string as the XML format. In host languages, XML host variables use a syntax that looks like a distinct type on LOB. For example, in C or C++, you can use the following host variable declaration:

```sql
EXEC SQL BEGIN DECLARE SECTION;
    SQL TYPE IS XML AS CLOB(1M) xmlPo;
EXEC SQL END DECLARE SECTION;
```

You can use the host variable in `INSERT` or `SELECT` as follows:

```sql
EXEC SQL INSERT INTO BASICS.PURCHASEORDERS VALUES ('200600001',
    CURRENT DATE, 'A', :xmlPo);
EXEC SQL SELECT POXML INTO :xmlPo
```
For INSERT, implicit XMLPARSE is invoked, while for SELECT, implicit XMLSERIALIZE applies. You can also use explicit XMLPARSE and XMLSERIALIZE to convert between string format and internal data model format. The following is an example of XMLSERIALIZE:

```sql
EXEC SQL SELECT XMLSERIALIZE(POXML AS CLOB(100K)) INTO :clobPo
FROM BASICS.PURCHASEORDERS
WHERE PONUMBER = '20060001';
```

Since XML data does not have a length limit, it is difficult to determine how much memory to allocate for a host variable to receive an XML value from DB2. In DB2 9, a new way of fetch XML and LOB data is introduced to allow for piece-by-piece fetch. The facility is the new option for the FETCH statement: FETCH WITH CONTINUE and FETCH CURRENT CONTINUE. Check SQL reference for details.

In JDBC, the standard interface methods `setString()`, `setCharacterString()`, `getString()`, `setBinaryStream()`, and `getBinaryStream()` etc. are expanded to support the XML type also. A new class `com.ibm.db2.jcc.DB2Xml` is also introduced to provide some XML specific methods.

**XML data encoding**

DB2 9 for z/OS supports XML columns in a table of any DB2-supported encoding. XML data is converted into UTF-8 at bind-in time before parsing if it is not already in UTF-8. Likewise, XML data is serialized into UTF-8 first internally at bind-out time and then converted into the encoding of the host variable or application encoding if necessary.

If an XML value is stored in a character host variable, the encoding of the host variable takes precedence over the encoding declaration inside the XML data. It is important to keep consistency between the real encoding and host variable encoding. Otherwise, the data may get corrupted or parsing may fail.

On the other hand, if an XML value is stored in a binary host variable, the encoding determination process as specified by W3C for XML will apply, which includes Byte Order Mark (BOM) or internal encoding declaration.

Since XML character data are stored in UTF-8 internally in DB2 9, using UTF-8 database and application encoding, or UTF-8 encoding in binary host variables for XML data, can avoid the encoding conversion overhead and potential data loss problem during the bind-in and bind-out processes.

**XML indexing**

When an XML column is included in a table, two indexes are implicitly created by DB2 as part of the storage scheme and to facilitate access methods: a DOCID index on the base table and a NODEID index on the internal XML table. In addition to these XML related index objects, you can create specific XML indexes on XML columns using
XPath expressions. The XML indexes supported in DB2 9 are value indexes. A value
index maps node values to nodes, identified by NODEIDs and RIDs of records in which
the nodes reside. For example, the following example creates an XML index on the
POXML column of table BASICS.PURCHASEORDERS using XML pattern
'//purchaseOrder/items/item/desc', which identifies all the descriptions of items
within purchaseOrder. Notice that XPath expressions are case-sensitive.

CREATE INDEX IXPO ON BASICS.PURCHASEORDERS(POXML) GENERATE KEYS USING
XMLPATTERN '//purchaseOrder/items/item/desc' AS SQL VARCHAR(100);

Only string and numerical data types are supported for XML indexes in DB2 9 for z/OS,
which use SQL VARCHAR(n) or DECFLOAT correspondingly. An XML index is logically
created on an XML column of a base table, but is physically on the implicitly created
XML table, which is reflected in the catalog information and database object description.

The XML pattern is a limited subset of XPath expressions that do not have any predicate.
Only element, attribute, or text nodes are allowed for indexing in DB2 9. An indexed
element node can have sub-elements, in which case the concatenated text is used for
the key value, but there is no composite key supported.

An XML index is different from indexes on columns of other types in that it may have
zero or more index entries for each document, depending on the XML pattern specified
and the document value. For example, for the index created above, there are two entries
for the document illustrated below. They are Lawnmower and Baby Monitor.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    ...
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    ...
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <desc>Lawnmower</desc>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <desc>Baby Monitor</desc>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>2003-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```
XML indexes are used in accelerating the query processing for the `XMLEXISTS()` predicate and `XMLTABLE()` function, but not `XMLQUERY()`.

**Searching and retrieving XML data**

In addition to simple SELECT of columns and expressions of XML type, you can search on XML data using the `XMLEXISTS()` predicate. Except for NULL testing, `XMLEXISTS()` is the only predicate applicable to the XML type, and no direct comparison operators are available for XML at the SQL level. In `XMLEXISTS()`, you specify an XPath expression for a document to match. If the result of the XPath expression is an empty sequence, then `XMLEXISTS()` returns false. Otherwise, it returns true. For example, the following queries select POXML that has an item with “Shoe” as the description. The second example is to illustrate that you can pass an SQL expression into an XPath expression.

```sql
SELECT POXML
FROM BASICS.PURCHASEORDERS
WHERE XMLEXISTS('/purchaseOrder/items/item[desc = "Shoe"]' PASSING POXML);

SELECT POXML
FROM BASICS.PURCHASEORDERS
WHERE XMLEXISTS('/purchaseOrder/items/item[desc = $x]' PASSING POXML, 'Shoe' AS "x");
```

You can also pass in a host variable or parameter marker, or another column as an argument for the XPath expression. You can use `XMLEXISTS` predicate anywhere a predicate can be used except in the ON clause of an outer join.

You can extract portions of an XML document using the `XMLQUERY()` function with XPath as the first argument, and other optional arguments just as in `XMLEXISTS()`. The following example extracts the quantity elements of “Shoe” item from the purchase order.

```sql
SELECT XMLQUERY('/purchaseOrder/items/item[desc="Shoe"]/quantity'
PAS SING POXML)
FROM BASICS.PURCHASEORDERS
WHERE XMLEXISTS('/purchaseOrder/items/item[desc = "Shoe"]' PASSING POXML);
```

Functions `fn:data()` and `fn:string()` can be used to get the value of an element or attribute instead of the element or attribute itself. For example, if you want to get a list of quantities you can use the following query.

```sql
SELECT XMLQUERY('fn:data(//items/item[desc="Shoe"]/quantity)' 
                    PASSING POXML)
FROM BASICS.PURCHASEORDERS
WHERE XMLEXISTS('//items/item[desc = "Shoe"]' PASSING POXML);
```
The most powerful XML function is the XMLTABLE() function, which, together with the XMLCAST() function, is delivered through PTF UK33493 for DB2 9. XMLTABLE() is a table function on XML data, and it can be used to:

- **Create a relational view on XML data, and enable to apply SQL power to process XML data.** (multiple rows with multiple columns)
- **Iterate through a sequence, achieve some XQuery functionality together with SQL XML constructor functions.** (multiple rows with a single column)
- **Select multiple items from a document, and use them separately.** (single row with multiple columns)
- **Split a large document into smaller documents to store, together with regular columns.**

XMLQuery() function returns a sequence of items from an XML value. If you need to process each item, you have to use XMLTABLE() function. For example, if you need to generate a new document from a purchase order, taking some of the item information into the new document, you would need XMLTABLE() function to iterate through each item, construct a new element, then use XMLAGG() to concatenate them back. The following query illustrates how to select document components to generate a new document:

```sql
SELECT XMLELEMENT(NAME "invoice", ..., XMLAGG( XMLELEMENT(NAME "item", "@partnum", "productName", "USPrice") ), ... ) FROM PurchaseOrders PO, XMLTABLE( '//item' PASSING PO.POXML COLUMNS "@partnum" XML, "productName" XML, "USPrice" XML) AS X WHERE PO.ponumber = '200300001';
```

The following example shows how XMLTABLE() function can be used to split a large document into smaller ones with regular columns to identify the split documents.

```sql
INSERT INTO TB25MMF
SELECT T.RETURNID, T.DOCID, T.DOCNAME, XMLDOCUMENT( T.DOC) FROM XMLTABLE(XMLNAMESPACES(DEFAULT 'http://www.irs.gov/efile'), '$d/Return/ReturnData/*' passing ? as "d" COLUMNS RETURNID char(20) PATH './../ReturnHeader/ReturnId', DOCID varchar(50) PATH '@documentId' DOCNAME varchar(50) PATH '@documentName', DOC xml PATH '.') as T
```

The following query examples show how a relational view can be defined on XML documents and SQL expressions can be applied to the relational view for reporting and analytics:

```sql
CREATE VIEW ORDER_VIEW AS
SELECT PO.POIID, X.*
```
Due to its power, XMLTABLE() function can be found in most XML queries in real applications. DB2 optimizes XMLTABLE() processing by supporting predicate pushdown from relational predicates, eliminating unreferenced columns, and using XML indexes.

XMLCAST() is used to cast between an XML type and a relational type. Since casting from relational data to XML data is implicitly performed when passing them into XML functions, and casting from XML to relational is also done in the XMLTABLE() function, XMLCAST() is mostly used in casting an XML value into a relational type with XMLQUERY() function. The following example shows the difference of casting an xs:string to VARCHAR from XMLSERIALIZE().

Assuming the department name for department 123 is “Shipping & Receiving”, the result of the above query will be the following:

Shipping & Receiving

**XPath support**

In DB2 9 for z/OS, an XPath expression is used to identify portions of an XML document and is used in XMLEXISTS(), XMLQUERY(), XMLTABLE(), and XMLPATTERN of
CREATE INDEX. Our strategy is to provide the core XML query language features that are critical to the application development in DB2 9, and expand them into the full XQuery language in the follow-on releases. We adopt the equivalent of the core XPath 1.0 language constructs and data types. However, we follow XPath 2.0 semantics and make them compatible with XQuery, including the XQuery prolog for namespace declaration that is not part of XPath but is necessary for the language.

The supported primitive data types are the following:
- `xs:boolean`,
- `xs:integer`,
- `xs:decimal`,
- `xs:double`, and
- `xs:string`.

And the supported axes are the XQuery-required axes:
- **the forward axes and their abbreviated form**: child, attribute, descendant, descendant-or-self, self, `//`, `@` and
- **the parent axis and its abbreviated form** `(.)`.

In addition to the constructors, general comparison and operators for the data types, the following built-in functions are supported: `fn:abs`, `fn:boolean`, `fn:compare`, `fn:concat`, `fn:contains`, `fn:count`, `fn:data`, `fn:string-length`, `fn:normalize-space`, `fn:not`, `fn:round`, `fn:string`, `fn:substring`, `fn:sum`. And the following additional XPath functions were delivered through PTF UK34342 post GA: `fn:distinct-values`, `fn:max`, `fn:min`, `fn:upper-case`, `fn:lower-case`, `fn:translate`, `fn:tokenize`, `fn:matches`, `fn:replace`, `fn:position`, `fn:last`, `fn:name`, `fn:local-name`.

Note that positional predicate is now supported with the `position()` function, as well as its short notation, such as `/purchaseOrder/items/item[2]`. For string search, `fn:contains` provides simple substring search, while `fn:matches` provides powerful regular expression pattern search.

One important difference of general comparison in XPath/XQuery from SQL comparison is that general comparison is between two sequences and has existential semantics, i.e. as long as one pair from the two sequences produces the true result, the comparison is true.

Because in DB2 9 for z/OS all stored XML documents are untyped, sometimes you need to use type casting (constructors for primitive types) for the correct semantics. For example, no cast is needed for the following query: "Find all the products in the Catalog with RegPrice > 100", assuming we have an XML column named `XCatalog`.

```xml
XMLQUERY('/Catalog/Categories/Product[RegPrice > 100]' PASSING XCatalog)
```

Similarly, there is no need for cast in the following query: "Find all the products with more than 10% discount in the Catalog".
However, we need a cast for the following query: “Find all the products on sale in the Catalog”, as comparison $\text{RegPrice} < \text{SalePrice}$ will become an untypedAtomic comparison if the cast to $\text{xs:double}$ is not specified:

$$\text{XMLQUERY}('/\text{Catalog}/\text{Categories}/\text{Product}[\text{RegPrice} * 0.9 > \text{SalePrice}]$'

PASSING XCatalog)

In the above latter two examples, there is a potential cardinality problem that may cause an error. For the expression $\text{RegPrice} * 0.9$ to work, there can only be one $\text{RegPrice}$ element under a $\text{Product}$ element. Likewise, $\text{xs:double()}$ only takes one item, and it will cause an error if there are multiple $\text{SalePrice}$'s for a $\text{Product}$ element. It is important in XPath programming to avoid cardinality errors. To avoid these potential errors, we can code the two XPath expressions as follows, respectively:

$$\text{XMLQUERY}('/\text{Catalog}/\text{Categories}/\text{Product}[\text{RegPrice} > \text{xs:double}(\text{SalePrice})]$'

PASSING XCatalog)

These two XPath expressions use a feature that is not available in XPath 1.0, i.e. to return a sequence of atomic values from the last step of a path expression.

**Constructing XML**

In DB2 V8, the following XML publishing functions, including XML constructors and other functions, were introduced to construct XML data from relational data: $\text{XMLELEMENT}$, $\text{XMLATTRIBUTES}$, $\text{XMLNAMESPACES}$, $\text{XMLFOREST}$, $\text{XMLCONCAT}$, and $\text{XMLAGG}$. Since there was no external XML data type, the $\text{XML2CLOB}$ function must be used to get the data out of the DB2 server as a CLOB. These functions provide convenient and high-performance alternative to XML construction in applications.

In DB2 9, these functions are extended to handle binary data types using $\text{HEX}$ or $\text{BASE64}$ encoding, and take null handling options. New constructors are added to make the constructor set complete, these include: $\text{XMLTEXT}$, $\text{XMLPI}$, $\text{XMLCOMMENT}$, and $\text{XMLDOCUMENT}$. Since the functions return the XML data type that is now a first-class SQL type, there is no need to use the $\text{XML2CLOB}$ function any more.

These functions also take the XML data type as input, you can use them to construct new documents from portions of existing documents extracted by the $\text{XMLQUERY}$ and $\text{XMLTABLE}$ functions. Here is an example of constructing XML from relational data:

```
SELECT XMLDOCUMENT(  
    XMLELEMENT(NAME "hr:Department",  
        XMLNAMESPACES('http://example.com/hr' as "hr"),
```
Notice that you can specify ordering inside the `XMLAGG()` function. A sample result may look like the following with formatting spaces inserted for easy reading:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<hr:Department xmlns:hr="http://example.com/hr" name="Shipping">
   <!-- names in alphabetical order -->
   <hr:emp>Lee</hr:emp>
   <hr:emp>Martin</hr:emp>
   <hr:emp>Oppenheimer</hr:emp>
</hr:Department>
```

Rational Data Architect provides a tool in assisting generation of the SQL queries that generate XML from relational data.

The following is another example to construct a new document, an invoice, from existing data, a purchase order, illustrating how SQL/XML constructors with `XMLQUERY()` can be used to achieve some of the XQuery functionality.

```sql
SELECT XMLDocument(
   XMLElement(NAME "invoice",
      XMLAttributes( '12345' as "invoiceNo"),
      XMLQuery ('/purchaseOrder/billTo' PASSING poxml),
      XMLElement(NAME "purchaseOrderNo", PO.ponumber),
      XMLElement(NAME "amount",
         XMLQuery ('fn:sum(/purchaseOrder/items/item/xs:decimal(USPrice))' PASSING poxml))
   )
) AS "dept_list"
FROM employees e
GROUP BY dept;
```

The result may look like this (formatted for ease of reading):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<invoice invoiceNo="12345">
   <billTo country="US">
      <name>Robert Smith</name>
      . . .
   </billTo>
   <purchaseOrderNo>200600001</purchaseOrderNo>
   <amount>188.93</amount>
</invoice>
```
Consuming web services from SQL

DB2 provides two UDF functions to invoke web services directly from within SQL statements: SOAPHTTPNV and SOAPHTTPNC, and they are replacing SOAPHTTPV and SOAPHTTPC. They differ in the return types. SOAPHTTPNV returns VARCHAR(32672), while SOAPHTTPNC returns CLOB(1M).

For example, the following is a simple example to get the conversion rate from EUR to USD from a public web service.

```sql
SELECT DB2XML.SOAPHTTPNV(
    'http://www.webservicex.net/CurrencyConvertor.asmx',
    'http://www.webserviceX.NET/ConversionRate',
    '<?xml version="1.0" encoding="utf-8"?>
    <soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
    <ConversionRate xmlns="http://www.webserviceX.NET/">
    <FromCurrency>EUR</FromCurrency>
    <ToCurrency>USD</ToCurrency>
    </ConversionRate>
    </soap:Body>
    </soap:Envelope>"
) FROM SYSIBM.SYSDUMMYU;
```

This is the result on August 13, 2008 (formatted for ease of reading):

```xml
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <soap:Body>
    <ConversionRateResponse xmlns="http://www.webserviceX.NET/">
    <ConversionRateResult>1.492</ConversionRateResult>
    </ConversionRateResponse>
    </soap:Body>
</soap:Envelope>
```

With the XML processing capability in SQL/XML, the returned XML response from web services can be directly used for further computation in the same SQL statement using the XMLPARSE() result as input to XMLQUERY() or XMLTABLE() functions. The following is an example of converting EUR1,000 to USD with the conversion rate extracted from the same web service response.

```sql
SELECT 1000 * XMLCAST(
    XMLQUERY
    ('$d//*:ConversionRateResult' PASSING XMLPARSE (DOCUMENT DB2XML.SOAPHTTPNV(
        'http://www.webservicex.net/CurrencyConvertor.asmx',
        'http://www.webserviceX.NET/ConversionRate',
        '<?xml version="1.0" encoding="utf-8"?>
        <soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soap:Body>
    <ConversionRateResponse xmlns="http://www.webserviceX.NET/">
    <ConversionRateResult>1.492</ConversionRateResult>
    </ConversionRateResponse>
    </soap:Body>
</soap:Envelope>")) FROM SYSIBM.SYSDUMMYU;
```
It returned result: 1492.00000.

**SQL/XML with XPath versus XQuery**

One of the key differences of XPath from XQuery is that XPath can only select data from a document and return as a sequence, while XQuery can construct a new document from an existing document or documents. With the delivery of `XMLTABLE()` function in SQL/XML, DB2 9 for z/OS is able to produce sophisticated XML query results that are comparable with XQuery. And leveraging SQL capabilities on the XML data, it can even provide functionality that is beyond XQuery. We use the following a few examples to illustrate how an XQuery can be converted into an SQL/XML query.

The first example is a simple FLWOR expression that does not have any constructors or order by involved. In this case, the FLWOR expression is actually an XPath expression.

```sql
XQuery (not supported in DB2 9 for z/OS):
for $x in /purchaseOrder/items/item
where contains($x/desc, "Shoe")
return $x
```

Equivalent XPath (supported in DB2 9 for z/OS):
/\purchaseorder/items/item[contains(desc, "Shoe")]

The second example is from W3C XQuery use cases, which involves a constructor on the outside. Assuming we have a table DOCUMENTS(DOCURI, DOC), we can use SQL/XML constructors and XMLQUERY function to achieve the same functionality. Note that `name()` and `local-name()` return the same result in this case as there is no namespace.

```sql
XQuery (not supported in DB2 9 for z/OS):
<results>
  {
    for $t in doc("books.xml")//@chapter | section/title
      where contains($t/text(), "XML")
      return $t
  }
</results>
```

Equivalent SQL/XML (supported in DB2 9 for z/OS):
The last example is also from W3C XQuery use cases, which involves constructor on the outside and inside of FLWOR, and an order by clause. Assuming we have a table DOCUMENTS(DOCURI, DOC), we can use SQL/XML constructors and XMLTABLE() and XMLAGG functions to achieve the same functionality:

```
XQuery (not supported in DB2 9 for z/OS):
<bib>
  { 
    for $b in doc("http://bstore1.example.com/bib.xml")//book 
    where $b/publisher = "Addison-Wesley" and $b/@year > 1991 
    order by $b/title 
    return 
      <book> 
        { $b/@year } 
        { $b/title } 
      </book> 
  }
</bib>
```

Equivalent SQL/XML (supported in DB2 9 for z/OS):
```
SELECT XMLDOCUMENT( 
  XMLELEMENT(NAME "bib", 
    XMLAGG( 
      XMLELEMENT(NAME "book", @year, title) 
      ORDER BY XMLCAST("title" as VARCHAR(100)) 
    ) 
  ) 
FROM DOCUMENTS, 
XMLTABLE('/bib/book[publisher = "Addison-Wesley" 
    and @year > 1991]' PASSING DOC 
COLUMNS 
  "@year" XML, 
  "title" XML) AS X 
WHERE DOCURI = 'http://bstore1.example.com/bib.xml';
```

In addition, SQL DATE and TIMESTAMP can be used as the result types of XMLTABLE() function so that these types can be handled in XML data with SQL functionality.

**Access Methods**

DB2 9 introduces several new access methods for XML data. The basic access method is the so-called DocScan. It traverses XML data and evaluate XPath expressions using our patent-pending streaming XPath evaluation algorithm, called QuickXScan. However, there is no new access type indicator for DocScan in the PLAN_TABLE as it is part of R-Scan if there is a predicate on an XML column involved.
Three new access type indicators are introduced for XML index-based access. Similar to RID list access, ANDing, and ORing, they include:

- **DocID list access (DX).**
- **DocID list ANDing (DI for DocID list Intersection).**
- **DocID list ORing (DU for DocID list Union).**

As mentioned earlier, XML indexes are only used for the XML EXISTS predicate and XMLTABLE function evaluation. For example, to evaluate predicate

```sql
XML EXISTS('/Catalog/Categories/Product[RegPrice > 100]' PASSING XCatalog)
```

If you have an XML index, IX1, on the XCatalog column created with the XML Pattern and type as follows:

```sql
CREATE INDEX IX1 ON MYTABLE(XCATALOG) GENERATE KEYS USING XMLPATTERN
'/Catalog/Categories/Product/RegPrice' as SQL DECFLOAT
```

DB2 9 will use this index for DocID list access (DX) for the predicate. It will get unique DocID list from the XML index, then convert the list to a RID list using the DOCID index on the base table, and access the base table and XML table. It will then re-evaluate the predicate on the document using QuickXScan. Because DB2 9 always re-evaluates XML EXISTS predicate, the XML pattern of an XML index does not have to exactly match with an XPath expression to apply the index. Similar for the XMLTABLE() function, XML indexes only provide filtering, and QuickXScan is used to evaluate the function.

Here is another example of using DocID list ANDing (DI) to evaluate a predicate:

```sql
XML EXISTS('/Catalog/Categories/Product[RegPrice > 100 and Desc = "Shoe"]' PASSING XCatalog)
```

Assuming two indexes on the XCatalog column with XML Patterns and data types: one is the same as the above IX1, and the other is the following IX2:

```sql
CREATE INDEX IX2 ON MYTABLE(XCATALOG) GENERATE KEYS USING XMLPATTERN
'/Catalog//Desc' as SQL VARCHAR(50);
```

Indexes IX1 and IX2 will be used to get two DocID lists and then DocID list ANDing (DI) will be applied to get a unique DocID list. DB2 9 will then convert the DocID list to a RID list via the DOCID index and access the base table and evaluate the predicate through QuickXScan.

**XML schema repository**

W3C XML Schema specifies a target namespace and optional schema location, both URIs, to identify an XML schema. For example, a target namespace could be

However, it is not required to have the schema online in the specified URIs. In addition, it is not recommended to get a schema online automatically by applications. Therefore you need to register XML schemas into DB2 XML Schema Repository (XSR) before you can use them in XML schema validation or annotated schema-based decomposition. A set of stored procedures are provided for managing XML schemas. When you register an XML schema, you specify an SQL ID for it. The stored procedures are the following:

- `XSR_REGISTER(rschema, name, schemalocation, xsd, docproperty)`
- `XSR_ADDSCHEMADOC(rschema, name, schemalocation, xsd, docproperty)`
- `XSR_COMPLETE(rschema, name, schemaproperties, isUsedForDecomp)`
- `XSR_REMOVE(rschema, name)`

Assuming you have a schema with schema location 'http://www.n1.com/order.xsd' and it also uses two other schema documents 'http://www.n1.com/lineitem.xsd' and 'http://www.n1.com/parts.xsd' by include or import, and you want to identify this schema using SQL ID ORDERSCEMA, you can use the following stored procedure call sequence, with the root schema document first:

```
XSR_REGISTER('SYSXSR', 'ORDERSCEMA', 'http://www.n1.com/order.xsd', :xsd, :docproperty)
XSR_ADDSCHEMADOC('SYSXSR', 'ORDERSCEMA', 'http://www.n1.com/lineitem.xsd', :xsd, :docproperty)
XSR_ADDSCHEMADOC('SYSXSR', 'ORDERSCEMA', 'http://www.n1.com/parts.xsd', :xsd, :docproperty)
XSR_COMPLETE('SYSXSR', 'ORDERSCEMA', :schemaproperty, 0)
```

In each call, :xsd contains one of three corresponding XML schema documents. At the `XSR_COMPLETE` call, DB2 9 will compile the schema into a binary format. When the schema is used, its binary format is loaded to achieve high performance. Any errors will be reported during the compilation time also. It invokes Java XML parser so you need Java JDK 1.5 or above installed with DB2 9 server, and Java stored procedures configured.

Alternative ways to register XML schemas are to use IBM Data Studio or Command Line Processor (CLP). In CLP, the following command can achieve the same as the previous stored procedure steps, assuming the corresponding files contain the schema documents:

```
REGISTER XMLSCHEMA http://www.nl.com/order.xsd
 FROM file://C:/xmlschema/order.xsd
 AS SYSXSR.ORDERSCEMA
 ADD http://www.nl.com/lineitem.xsd
 FROM file://C:/xmlschema/lineitem.xsd
 ADD http://www.nl.com/parts.xsd
 FROM file://C:/xmlschema/parts.xsd
 COMPLETE;
```
Schema validation

To validate an XML document against a registered schema, you invoke the DB2-supplied `DSN_XMLVALIDATE()` UDF. The `DSN_XMLVALIDATE()` UDF works as the standard `XMLVALIDATE()` SQL function except that it does not retain type annotations after schema validation. For example, you can validate XML data during INSERT:

```
INSERT INTO BASICS.PURCHASEORDERS VALUES
(‘2006040001’, CURRENT DATE, ‘A’,
XMLPARSE(DOCUMENT
    SYSFUN.DSN_XMLVALIDATE(:lobPO, ‘SYSXSR.ORDERSCHEMA’)),
NULL);
```

DB2 9 invokes a new high-performing schema validation parser (XLXP-C) for validation. However, schema validation is still more costly than simple parsing only.

Annotated schema-based decomposition

If you want to decompose (shred) an XML document and store the data in regular SQL columns and XML columns of relational tables, you can use a new stored procedure `XDBDECOMPXML` to achieve this. You need to edit schema documents and add annotations to specify how you want a document to be decomposed. The IBM Data Studio provides a tool to assist the annotation. If you decompose XML data into pure relational data without XML, you are no longer taking the advantages of pureXML technology. For details of annotations and the `XDBDECOMPXML` stored procedure, refer to DB2 9 XML Guide.

XML decomposition for XML files can also be achieved by a DECOMPOSE XML command in CLP.

Utilities

All DB2 utilities are enhanced to handle XML data type and XML related database objects properly. Here is a list of utility features and restrictions related to XML:

- **CHECK DATA**: checking consistency between a base table space which contains XML columns and its XML table spaces.
- **CHECK INDEX**: checking of the DocID, NodeID and XML value indexes.
- **CHECK LOB**: adds error checking to disallow processing of XML table spaces.
- **COPY INDEX**: support taking full image copies and concurrent copies of DocID, NodeID and XML value indexes.
- **COPY TABLESPACE**: support taking full, incremental image copies and concurrent copies of XML table spaces.
- **COPYTOCOPY**: support the replication of image copies of XML table spaces, DocID, NodeID and XML value indexes.
- **EXEC SQL**: adds error checking to disallow cross loading of tables with XML columns.
- **LISTDEF**: implements a new XML keyword for constructing lists with or without XML objects.
- **LOAD**: supports loading of tables with XML columns. Loading from file references is support.
- **MERGECOPY**: supports merging of image copies of XML table spaces with existing function.
- **QUIESCE TABLESPACESET**: includes XML table spaces and index spaces in the set of quiesced objects.
- **REAL TIME STATISTICS**: gathers existing statistics on the new XML objects.
- **REBUILD INDEX**: supports rebuilding of DocID, NodeID and XML value indexes.
- **RECOVER INDEX**: supports recovery of DocID, NodeID and XML value indexes and will include XML objects during consistency checking of point-in-time recoveries.
- **RECOVER TABLESPACE**: supports recovery of XML table spaces and includes XML objects during consistency checking of point-in-time recoveries.
- **REORGL INDEX**: supports reorganization of DocID, NodeID and XML value indexes.
- **REORGL TABLESPACE**: supports reorganization of XML table spaces and of base table spaces with XML columns with some restrictions.
- **REPORT TABLESPACESET**: includes XML table spaces, DocID, NodeID and XML value indexes in the set of reported objects.
- **RUNSTATS INDEX**: processes base table DocID indexes normally, collects basic statistics for NodeID indexes and XML value indexes.
- **RUNSTATS TABLESPACE**: processes the base table space DocID column normally and collects basic statistics for all XML table space columns.
- **UNLOAD**: supports unloading of tables containing XML columns. Unloading to file references is supported, but unloading of XML data FROMCOPY is not supported.

All the standalone utilities also support XML objects. Since an XML table space contains string IDs in XML data that are in the dictionary (i.e. SYSIBM.SYSXMLSTRINGS catalog table) specific to each catalog, an XML table space copied by DSN1COPY cannot be moved to another DB2 subsystem.

The database operation and recovery are similar to that of a database with LOB data. The following provides guidelines.

- **To recover base table space, take image copies of all related objects**
  - Use LISTDEF … XML or ALL to obtain a list of related objects including XML objects
  - QUIESCE is no longer needed in DB2 9 for consistent point in time recovery
- **COPYTOCOPY** may be used to replicate image copies of XML objects.
- **MERGECOPY** may be used to merge incremental copies of XML table spaces.
- **Point in Time recovery (RECOVER TORBA, TOLOGPOINT)**
  - All related objects, including XML objects must be recovered to a consistent point in time
- **Optional**: **CHECK INDEX** and **CHECK DATA** utilities to validate base table spaces with XML columns, XML indexes and related XML table spaces.
- **If there is an availability issue with one object in the related set, availability of the others may be impacted.**

### Performance monitoring

Since native XML support in DB2 9 is built on top of regular table spaces, index structures, and buffer pools, there are no special changes in DB2 Performance Expert
other than the new XML document lock (type x'35'). XML performance problems can be analyzed through accounting traces and performance traces as usual.

Some configuration information may help you. DB2 9 introduces a new load module DSNNXML in the DBM1 address space for most of native XML processing. Implicit or explicit XMLPARSE invokes z/OS XML System Services within the same address space. Stored procedures are used for schema registration, and XML schema validation invokes a user-defined function supplied by DB2.

Performance and cost factors

Overall performance involving XML data depends on many factors. Let us look at storage format, XML parsing, data ingestion, XML indexing, and XML query processing. We have designed a very compact XML storage. In addition, the standard table space compression works very well on the XML storage format. Insignificant whitespace in XML documents is stripped by default during insertion. The base storage for XML data ranges from about 0.3 to 1.5 times the original document size, depending on both the whitespace and compression options. Measurements also show that the CPU overhead of compression is minimal, and compression improves elapsed time slightly for queries.

There was a belief that XML parsing is expensive. It is no longer true for z/OS XML System Services, which provides extremely high-performing XML parsing service. XLXP-C validating parser, which DB2 invokes for schema validation, is high-performing, too. For simple insertion, parsing takes about 15% to 30% of the total CPU time. Validation is 2-3 times more expensive still. Compared with LOB insertion, insertion of small XML documents actually is faster than LOBs. On average, it is observed that LOB insertion is about 70% of the XML insertion CPU time across various sizes up to 10MB. In batch mode, a single thread on z9 with z/OS R1.8 can insert 4 millions documents of 10KB in size per hour or 1100 docs/sec.

XML indexes add 10-20% CPU time each to base insertion, if it has index keys generated. And time varies based on the XPath pattern, the more steps, the more expensive. Also descendant axis (//) and wild card (*) are also expensive to evaluate.

The LOAD utility uses insert for XML data. However, it is more efficient to use LOAD than INSERT with the LOG NO option. Also rebuild index is better than create index over existing data.

Simple select of XML data is extremely fast. For example, in batch mode, a single thread on z9 can select over 9 millions documents of 10KB in size per hour or 2580 docs/sec.

Regarding the memory usage, typical relational columns have a fixed maximum length so query preparation allocates memory for the columns. However, for XML, dynamic storage allocation is used extensively, and usually it requires more storage to handle same amount of data.

XML query performance depends on the data touched and retrieved, and whether indexes can be used, and the complexity of the operations involved in queries. For most OLTP queries, indexes should be used to speed up the query processing. Depending on the query complexity and data volume, query throughput could range from a few hundreds to a couple of thousands per second with 5 CPUs on z9. We are continually
working on improving query performance. Since XML indexes are used for XML EXISTS and XMLTABLE with re-evaluation of the predicates by whole document scan, large documents that contain extra irrelevant data after index filtering is not good for query performance.

Compared with decomposition, using native XML is very favorable, as typically decomposition involves insertion into many tables with many rows, while insertion of XML is one shot. For queries, depending on the cases, relational queries could be faster if only a small portion of the data are searched on and returned. On the other hand, XML queries could be faster if the equivalent relational queries need to join a large number of tables, and return significant amount of document data. Especially, if the decomposition and joins are only to normalize and denormalize the data, no added value in the logic, then XML could be faster and is the right type to use.

Regarding the cost, all the XML processing except for the schema validation is inside DB2 engine and is eligible for zIIP re-direction, including XML parsing, if DB2 is under SRB mode, i.e. invoked through DRDA. However, it does not change the percentage of re-direction. In addition, XML parsing using z/OS System Services is 100% re-directable to zIIP or zAAP depending on the mode it's in (SRB to zIIP, TCB to zAAP). We've observed up to 15-50% CPU re-direction for INSERT or LOAD depending on the document size in the lab testing. The larger the documents, the more re-direction is possible.

Commonalities and differences from DB2 9.5 for Linux, UNIX, and Windows (LUW)

DB2 9 for z/OS XML features are a compatible subset of that of DB2 9.5 for LUW. The commonalities include:

- SQL XML data type and DDL, although there are well-known platform-specific options for databases in DDL.
- Standard-conforming SQL/XML language with XML query languages (XPath on z/OS, XQuery on LUW).
- Indexing: VARCHAR(n) and numerical (DECFLOAT on z/OS and DOUBLE on LUW). LUW also supports VARCHAR(HASHED), DATE and TIMESTAMP.
- XML columns are supported in tables of any supported encoding.
- XML Schema Repository, and schema validation (UDF on z/OS v.s. Built-In Function on LUW).
- INSERT/UPDATE/DELETE: same statements, LUW supports XQuery update transform expression.
- Host language interfaces: PL/I and assembler in z/OS in addition to C/C++, COBOL, Java, and .NET etc.
- Annotated schema decomposition.
- Text search.

The following are z/OS-specific:

- XPath in SQL/XML only while XQuery is supported in both embedded and top-level on LUW.
- XML columns are supported in partitioned table spaces and data sharing environment.
- Inline XML is not available on z/OS.
- Compression is supported for XML table spaces.
- **REORG** and many utilities are supported for XML objects.
- Latest parsers are used to provide unprecedented performance.

**Summary**

In this whitepaper, we have discussed the business value DB2 9 pureXML brings and some details of the XML features. The flexibility of XML schema and declarative and efficient XML query languages eliminate the bottleneck of mapping and schema evolution, improve productivity and quality of application development, and significantly accelerate time-to-market. It can also improve the system performance in processing XML, together with the unparalleled System z reliability, availability, and scalability. DB2 9 pureXML marks a new era of database application development, and leads the trend in enterprise XML data management.

**For more information**

To learn more about DB2 for z/OS, visit

For more technical information about DB2 for z/OS, visit information center
http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp

**DB2 9 XML Guide**

“**DB2 9 and z/OS XML System Services Synergy Update,**” Whitepaper

For **Command Line Processor (CLP)** see

For IBM Data Studio free download, visit

DB2 for z/OS XML related questions, send emails to
db2zxml@us.ibm.com.