IBM® DB2® for Linux®, UNIX®, and Windows®

Best Practices
Managing XML Data

Matthias Nicola
IBM Silicon Valley Lab

Susanne Englert
IBM Silicon Valley Lab

Last updated: January 2011

Executive summary

This paper provides principles and guidelines for using DB2® pureXML® to solve business problems effectively and to achieve high performance when managing XML data in enterprise applications. The examples illustrating the best practices are based on a real-world financial application scenario and demonstrate how to implement the guidelines. The examples can be easily adapted to other types of XML applications. The paper covers the following areas:

- Storage options for XML data to improve performance and storage efficiency
- Techniques for adding XML data into a DB2 database
- Techniques for querying and updating XML documents efficiently
- Techniques for using indexes over XML data with queries effectively
- Techniques for efficiently maintaining and monitoring an XML database
- Techniques for developing efficient pureXML applications

Why XML

XML provides a neutral, flexible way of exchanging data among different systems, applications, and organizations. With XML, data is maintained in an extensible self-describing format to accommodate ever-evolving business needs. XML documents use tags to describe the data values they contain, and the nesting of tags to express hierarchical relationships between the data items. XML can describe very structured data and enforce the structure through XML schemas, but can also describe semi-structured data which is prevalent in content-oriented applications.

Service oriented architectures (SOA), enterprise application integration (EAI), enterprise information integration (EII), Web services, the enterprise message bus (ESB), and standardization efforts in many industries all rely on XML as an underlying technology for data exchange.

Organizations as well as entire industries have standardized XML schemas to promote exchange of data and are evolving those schemas to meet changing business needs. These efforts include ACORD in the insurance industry, FpML® and FIXML in the financial industry, RosettaNet in supply chain management, ARTS in the retail business, HL7 in healthcare, XBRL for business reporting, and DITA for authoring, managing, and publishing documentation in print and on the Web.

Such industry-specific initiatives, as well as regulatory requirements, are driving the deployment of XML. As more business transactions are conducted through Web-based interfaces and electronic forms, government agencies and commercial enterprises bear greater responsibility for preserving the original order, request, claim, trade, or submission. XML provides a straightforward means of capturing and maintaining the data associated with these electronic transactions. Indeed, XML documents frequently represent transaction records in message-based transaction processing systems.

Pros and cons of XML and relational data

As a self-describing data format, XML allows diverse data (with or without an XML schema) to be stored in a single document or row without sacrificing the ability to search or aggregate portions of that data. Applications can evolve their XML schemas without causing any changes to the underlying database schema. While the flexibility of XML means that examining and interpreting XML data can consume more CPU and I/O resources than if the same data was stored in relational form, factors such as schema complexity might make storing the data in relational form impractical.

With more rigid schema definitions, the relational model requires significantly less interpretation and allows for more optimized data operations. As such, it can provide very high performance but might fail to meet application requirements for schema flexibility. The relational data model is very suitable for applications with stable data structures and predictable access patterns. XML is often more suitable for applications with complex and variable data structures, and for combining structured and unstructured information.

In some cases, XML offers performance benefits over relational models precisely because of its flexibility. Relational databases often require normalization to fit business data into flat, tabular structures. This normalization of complex business data requires transformation when data is stored and retrieved, and often leads to multi-way join queries in relational databases. XML can be a more natural representation of complex business objects with all relevant relationships represented in a single document. The hierarchies within an XML document are essentially pre-computed joins between related data items.

Another consideration in choosing a data model is the application using the data. Even if data originates in XML, if subsequent processing of that data depends on the data being stored in a tabular format—for example, when applying relational online analytical processing (OLAP) to data in a data warehouse—then storing the data in relational format instead of XML might be the correct choice.

XML solutions to relational data model problems

The storage data model should match, to the greatest extent possible, the highest value and most critical usage model for your data. If the data being modeled is naturally tabular, it is typically better to represent it in relational format that as XML. However, there are cases where the relational model is not necessarily the best choice and sometimes even poor choice to hold your data. The following are some situations where an XML representation tends to be more beneficial than the relational format:

When the schema is volatile

Problem with relational data: If the schema of the data changes often, then representing the data in relational form results incurs cost and overhead of changing the relational schema. While some forms of schema modification are relatively painless in relational databases, such as adding a new column to a table, other forms are more involved, such as dropping a column or changing the type of a column. Still other forms of schema modification are downright difficult, such as normalizing one table into multiple tables. Changing the tables then means that applications need to change the SQL statements that access them.

Solution with XML data: Portions of the schema that are volatile can be expressed as a single XML column. The self-describing and extensible nature of XML allows seamless handling of schema variability and evolution. Changes in the XML document format are accommodated without changing tables or columns in the database and typically without breaking existing XML queries.

When data is inherently hierarchical in nature

Problem with relational data: Data that is inherently hierarchical or recursive is often difficult to represent in relational schemas. Examples include bill of materials, engineering objects, or biological data. A bill of materials explosion can be stored in a relational database but reconstructing it in parts or in full might require recursive SQL.

Solution with XML data: Since XML is a hierarchical data model, it is a much more natural representation for inherently hierarchical business data. Using XML allows simple, navigational data access to replace complex set operations if the same was represented in tabular format.

When data represents business objects

Problem with relational data: If application data represents business objects, such as insurance claim forms, then it is often beneficial to keep the data items that comprise a particular claim together, instead of spreading them over a set of tables. This is particularly true when the individual data items of a claim form have no valid business meaning by themselves and can only be interpreted in the context of the complete form. Normalizing the claims across dozens of relational tables means that applications deal with a complex and unnatural fragmentation of their business data. This increases complexity and the chance for errors.

Solution with XML data: XML allows you to represent even complex business objects as cohesive and distinct documents while still capturing all of the relationships between the data items that comprise the business object. Representing each claim form (business object) as a single XML document in a single row of a table provides a very intuitive storage model for the application developer and allows rapid application development.

When objects have sparse attributes

Problem with relational data: Some applications have a large number of possible attributes, most of which are sparse, that is attributes applicable to very few objects. A classic example is a product catalog where the number of different product attributes is huge, including: size, color, weight, length, height, material, style, weave, voltage, resolution, water resistance, and a nearly endless list of other properties. For any given product, only a subset of these attributes is relevant. One possible relational approach is to store this data is to have one column per attribute, which means a very large percentage of the cells in the table contains NULL values. This is undesirable and can be inefficient. A different relational approach for such sparse data is a 3-column table that stores several name/value pairs for each product ID. This means the attribute names aren't column names but values in a VARCHAR column. This prevents relational database systems from accurately estimating constraint selectivity and generating efficient query plans. Also, defining and enforcing constraints, such as uniqueness for a certain attribute, is extremely difficult.

Solution with XML data: The beauty of XML is that elements and attributes can be optional, so they are simply omitted if they don't apply for a specific product. Neither NULL values nor name/value pairs are needed. The XML schema can define a very large number of optional elements, but only few of them are used for any given object. While every row in a relational table has to have the exact same columns, XML documents in an XML columns can have different elements from one row to the next. Also, an XML index for an optional element will be very small if this element appears only in a small percentage of the documents (rows). This is a clear advantage over relational indexes which have exactly one entry per row.

When data needs to be exchanged

Problem with relational data: If you export a set of rows from a relational table and send them to another application or organization, the recipient cannot interpret the data without additional metadata that describes the columns. This is particularly true if your relational schema has changed since the last time you sent data.

Solution with XML data: XML data is self-describing. The XML tags are metadata which describe the values that they enclose.

Benefits of DB2 pureXML over alternative storage options

Since XML has become increasingly critical to the operations of an enterprise, XML documents are assets that need to be shared, persisted, searched, secured, and updated with full transactional consistency. Depending on its use, XML data might also need to be transformed, audited, and integrated with other data. To meet these requirements, storing XML data in its native hierarchical format in a DB2 database has several advantages, including:

- Retaining awareness of the internal structure of the XML data. This has advantages over storing the XML documents as character or binary large objects (CLOBs or BLOBs) in the database. In particular, you can easily query XML data using XQuery, XPath, and SQL/XML to take advantage of the XML structure, and you can enhance query performance by creating indexes over XML data. Additionally, you can easily update, transform, and publish XML data using SQL, XQuery and XSLT.

- Maintaining the hierarchical and flexible nature of XML data. This has advantages over decomposing (shredding) the XML documents into relational tables where an administrator maps XML elements and attributes to relational columns. After shredding, XML document values are stored in these tables without their original tags. Shredding often requires a large number of tables and often it is too complex to be practical. Queries over decomposed XML documents can require complex SQL joins that tend to be difficult to develop and tune. Changes to the XML schema often break the mapping to the relational database schema. This is incurs costly and time-consuming maintenance which defeats the flexibility for which XML is typically chosen. This is why DB2 pureXML allows you to use a single XML column to store and query XML documents that are based on different XML schemas, or different versions of an evolving XML schema.

- Integration of XML documents with relational data in a single database. This has advantages over storing relational data in one database and XML documents in a separate, XML-only database. Such an approach requires skills and labor to operate and maintain two database systems instead of just one. Also, combining data from the two databases usually requires extra logic in the application that is often difficult and inefficient. When you store both XML and relational data in a single DB2 database, you can combine both types of data in queries, perform joins between them, and even convert one to other as needed. This can potentially be more cost-efficient and provides better performance than using two separate databases.

Best practices for DB2 pureXML: Overview

The DB2 pureXML feature offers sophisticated capabilities for storing, indexing, validating and querying XML data – fully integrated with the DB2 relational data management features. This document describes principles for using pureXML in an effective and efficient manner. The goal of this document is not to be an introduction to the pureXML features or how they work. Instead, this document provides guidelines to help achieve higher pureXML performance, as well as examples of how to deploy the pureXML functionality to solve specific business problems effectively.

We are using a real-world application scenario as the setting for the best practices and examples in this paper. It's a scenario from the financial industry and deals with the management of "derivative trades" based on an XML format called FpML (Financial Products Markup Language). You don't need to be a financial expert to understand this scenario. Although we use this specific scenario, the best practices also apply to other XML applications, such as XML forms processing, order management systems, XML in health care and electronic patient records management.

The topics in this paper are roughly organized according to the typical life cycle of a database project. We start in this section by reviewing the data and tables that the application requires. Then we discuss the DB2 storage options for XML data in "Choosing the right storage options for XML data." After that, "Guidelines for adding XML data to a DB2 database" provides tips and techniques for adding XML data into a DB2 database. In "How to query XML data efficiently and effectively," we present guidelines and examples for querying XML data more efficiently. To improve query performance, best practices for defining and using XML indexes are given in "Usage guidelines for XML indexes." Guidelines for XML namespaces and XML updates are discussed in "Dealing with XML namespaces" and "Effectively updating XML data," respectively. The sections "Maintaining and monitoring an XML database" and "Developing pureXML applications" cover additional topics for database administrators (DBAs) and application developers. Finally, "Summary" concludes with a summary of the most important guidelines.

Sample scenario: derivative trades in FpML format

A “derivative trade” is a financial trade which is based upon (derived from) some other financial asset, such as a stock, an index, an interest rate, a currency, or other. In a derivative trade, two parties agree to exchange cash, depending on market conditions that affect the underlying asset. Typically, one party uses the trade to mitigate risk; the other party uses the trade to gain immediate income (through fees or premiums) or to speculate that future market conditions will provide profits. Consider the following example.

YourWorld Investments and MyGlobal Bank agree on a currency exchange derivative trade. They negotiate that on October 25, YourWorld will pay 71,900,000 Chinese Yuan to MyGlobal, and MyGlobal will pay 10,000,000 US Dollars to YourWorld. MyGlobal will profit from this trade if the value of the US Dollar declines below 7.19 CYN per USD between now and Oct 25. MyGlobal might be using this trade to hedge against the risk of a falling US Dollar. YourWorld might speculate that the US Dollar will gain in value, or collect up-front fees from MyGlobal for entering the trade.

What’s interesting about derivatives is that (a) there are many different types and variations, (b) the conditions of a particular trade are often individually negotiated and complex, and (c) the life span of a derivative can range from days to years and their conditions might change over time. The financial industry found that the flexibility and extensibility of XML was required to define a standard data format that could capture the high variability of derivatives. As a result, they developed FpML. FpML is essentially an XML Schema that defines how XML elements and attributes are used to describe derivative trades. The International Swaps and Derivatives Association, Inc (ISDA) manages the FpML standard on behalf of a community of investment banks that make a market in OTC derivatives. See [11] and [12] for more information on derivatives and FpML.

Sample data and tables

Our sample database for this paper consists of three tables, shown in figure 1.

```
create table trades (tradeId integer, tradedoc XML);
create table parties(partyInfo XML);
create table currencies(symbol char(4), name varchar(30), USDvalue double,
                        lastUpdated timestamp);
```

Figure 1: Definitions of sample tables

All table definitions and commands to populate them with sample data as well as and queries and other statements shown in this paper are available as a downloadable file at http://www.ibm.com/developerworks/db2/bestpractices/.

The table TRADES contains FpML documents and a trade ID for each document. Each trade document references the two trading parties using the values of their PARTYID elements in the document. More detailed information about the parties is stored in XML format in the PARTIES table. An XML-to-XML join between TRADES and PARTIES allows us to find detailed party information for a given trade, or vice-versa. Many FpML trades reference specific currencies by symbol. Additional currency information is available in relational form in the CURRENCIES table, as shown in figure 2. We will need XML-to-relational joins to relate trades to this currency information.

```
SYMBOL NAME                           USDVALUE   LASTUPDATED
------ ------------------------------ --------   -------------------
USD    US Dollar                         1.000   2008-02-05-15.15.57
EUR    Euro                              1.460   2008-02-05-15.15.59
GBP    Great Britain Pounds              1.960   2008-02-05-15.16.23
JPY    Japanese Yen                      0.009   2008-02-05-15.15.53
CNY    Chinese Yuan (Renminbi, RMB)      7.190   2008-02-05-15.16.13
```

Figure 2: Content of the CURRENCIES table

Our PARTIES table contains 3 rows, with one XML document per row. They describe the parties that are involved in the sample derivative trades. The following figure shows the XML documents for the three parties.

```
<Party>
   <PtyID>510026</PtyID>
   <ShortName>MIB</ShortName>
   <Name>MyGlobal International Bank</Name>
   <Status>Active</Status>
   <Address>
       <Street>498 Wall Street</Street>
       <City>New York</City>
       <Country>USA</Country>
   </Address>
   <Rating>
      <RatingDate>2006-05-16</RatingDate>
      <RatingValue>Baa1</RatingValue>
   </Rating>
</Party>

<Party>
   <PtyID>67781</PtyID>
   <ShortName>NVB</ShortName>
   <Name>National Village Bank</Name>
   <Status>Active</Status>
   <Address>
       <Street>1805 Back Street</Street>
       <PostalCode>EC3M 4TD</PostalCode>
       <City>London</City>
       <Country>UK</Country>
   </Address>
   <Rating>
      <RatingDate>2006-06-01</RatingDate>
      <RatingValue>Aa</RatingValue>
   </Rating>
</Party>

<Party>
   <PtyID>99114</PtyID>
   <ShortName>YWI</ShortName>
   <Name>YourWorld Investments</Name>
   <Status>Active</Status>
   <Address>
       <POBox>98765</POBox>
       <PostalCode>100027</PostalCode>
       <City>Beijing</City>
       <Country>China</Country>
   </Address>
   <Rating>
```

```
      <RatingDate>2007-01-15</RatingDate>
      <RatingValue>Aaa</RatingValue>
   </Rating>
   <Rating>
      <RatingDate>2005-04-21</RatingDate>
      <RatingValue>Aa</RatingValue>
   </Rating>
</Party>
```

Figure 3: Contents of the PARTIES table

Figure 4 shows the FpML document that represents the currency exchange trade between YourWorld Investments and MyGlobal International Bank that we described earlier. The trade starts with a tradeHeader element which labels the two trading parties as “party1” and “party2” and relates them to their respective trade IDs as well as the date of the trade. At the very bottom of the FpML document, “party1” and “party2” are mapped to the actual identifiers 510026 and 99114 which reference party information in the PARTIES table. The actual body of the trade is in the fxSingleLeg element. FX stands for foreign exchange. There are two elements called exchangedCurrency1 and exchangedCurrency2. The first describes the payment of CNY 71.9M from YourWorld to MyGlobal, the second shows the payment of USD 10M from MyGlobal to YourWorld.

For clarity, we have removed all namespaces from the FpML sample data. We will revisit namespaces later in this document.

```
<FpML>
  <trade>
    <tradeHeader>
        <partyTradeIdentifier>
           <partyReference href="party1"/>
           <tradeId tradeIdScheme=
                 "http://www.MyGlobal.com/trade-id">MyGlobal941</tradeId>
        </partyTradeIdentifier>
        <partyTradeIdentifier>
           <partyReference href="party2"/>
           <tradeId tradeIdScheme=
                 "http://www.YourWorld.com/trade-id">YWI0089</tradeId>
        </partyTradeIdentifier>
        <tradeDate>2001-10-23Z</tradeDate>
     </tradeHeader>
     <fxSingleLeg>
        <exchangedCurrency1>
           <payerPartyReference href="party2"/>
           <receiverPartyReference href="party1"/>
           <paymentAmount>
              <currency>CNY</currency>
              <amount>71900000</amount>
           </paymentAmount>
        </exchangedCurrency1>
        <exchangedCurrency2>
           <payerPartyReference href="party1"/>
           <receiverPartyReference href="party2"/>
           <paymentAmount>
              <currency>USD</currency>
              <amount>10000000</amount>
           </paymentAmount>
        </exchangedCurrency2>
        <valueDate>2001-10-25Z</valueDate>
        <exchangeRate>
           <quotedCurrencyPair>
              <currency1>CNY</currency1>
              <currency2>USD</currency2>
              <quoteBasis>Currency2PerCurrency1</quoteBasis>
           </quotedCurrencyPair>
           <rate>7.91</rate>
        </exchangeRate>
     </fxSingleLeg>
  </trade>
  <party id="party1">
      <partyId>510026</partyId>
  </party>
  <party id="party2">
     <partyId>99114</partyId>
  </party>
</FpML>
```

Figure 4: Example of an interest rate derivative in FpML format

Figure 5 shows another example of an FpML document. The document is "an overnight Term Deposit." National Village Bank pays a 3% fixed rate loan for a 25 million Euro deposit from YourWorld Investments. The loan starts on February 15, 2002 and matures on February 16, 2002. You can see that the general structure of the trade header and the party information is the same as in figure 4, while the body of the trade is quite different.

```
<FpML>
  <trade>
    <tradeHeader>
       <partyTradeIdentifier>
          <partyReference href="party1"/>
          <tradeId tradeIdScheme=
                  "http://www.YourWorld.com/trade-id">YWI7623</tradeId>
       </partyTradeIdentifier>
       <partyTradeIdentifier>
          <partyReference href="party2"/>
          <tradeId tradeIdScheme=
                  "http://www.NationalV.com/swaps/trade-id">69197</tradeId>
       </partyTradeIdentifier>
       <tradeDate>2002-02-14Z</tradeDate>
    </tradeHeader>
    <termDeposit>
        <productType>Overnight Term Deposit</productType>
        <initialPayerReference href="party1"/>
        <initialReceiverReference href="party2"/>
        <startDate>2002-03-15Z</startDate>
        <maturityDate>2002-03-16Z</maturityDate>
        <dayCountFraction>ACT/360</dayCountFraction>
        <principal>
           <currency>GBP</currency>
           <amount>35000000.00</amount>
        </principal>
        <fixedRate>0.03</fixedRate>
    </termDeposit>
  </trade>
  <party id="party1">
     <partyId>99114</partyId>
  </party>
  <party id="party2">
     <partyId>67781</partyId>
  </party>
</FpML>
```

Figure 5: Example of an "overnight Term Deposit" in FpML

These two sample documents (figure 4 and figure 5) already give you an idea of the diversity of FpML data. The FpML schema defines approximately 1800 different XML elements and over 600 types. Any given instance document contains only a fraction of those. However, a relational database schema without XML columns would require at least 400 to 500 tables to be able to represent any possible FpML document. The complexity would be staggering and is generally considered not manageable, which is why XML is required.

Our TRADES table contains five FpML documents, including the two above (figure 4 and figure 5).

The complete sample data is available for download in a DB2 Command Line Processor (CLP) script as part of a downloadable file at http://www.ibm.com/developerworks/db2/bestpractices/.

Choosing the right storage options for XML data

Configuring the storage options correctly is important for maximizing the performance of a DB2 database. In this section, we discuss the type of table space for the database, the page size for the XML data stored in the database and the method of storing the XML data in the database.

The storage space consumption of XML data in a DB2 table space is 0.7x to 1.5x of the original XML data in text format in the file system. To get a more precise estimate, you can insert 1000 or more representative documents into an empty table and use a DB2 table snapshot to see the number of pages used for that table.

Selecting table space type and page size for XML data

DMS (database managed space) table spaces typically provide higher performance than SMS (system manage space) table spaces (operating system managed table spaces). This is true for relational data, and even more so for XML read and write access. Newly-created table spaces are DMS by default. The use of DMS table spaces with automatic storage is recommended so that DMS containers grow as needed without manual intervention. If an XML document is too large to fit on a single page in a table space, the DB2 software (referred to as simply DB2) splits the document into multiple regions which are then stored on multiple pages. This is transparent to your application and allows DB2 to handle XML documents up to the bind-in limit of 2GB per document.

Generally, the lower the number of regions (splits) per document, the better the performance, especially for insert and full-document retrieval. If a document does not fit on a page, the number of splits per document depends on the page size (4 KB, 8 KB, 16 KB, or 32 KB). The larger the page size of your table space, the lower the number of potential splits per document. For example, suppose a given document gets split across forty 4 KB pages. Then the same document might get stored on only twenty 8 KB pages, or ten 16 KB or five 32 KB pages, respectively. If the XML documents are significantly smaller than the selected page size, no space will be wasted since multiple small documents can be stored on a single page.

Most XML applications perform better using16 KB or 32 KB pages. 16 KB pages can provide good performance if most documents are quite small (for example, less than 4 KB) so that several documents fit on a page. Larger documents are better served by 32 KB pages. For our FpML scenario, we use 16 KB pages. 16 KB pages can also be a good compromise if you use a single page size for XML and relational data, or for data and indexes, and you find that 32 KB page are detrimental for efficient access to relational data or indexes. In that case however, you can consider using different page sizes.

Different table spaces and page size for XML and relational data

Our CREATE TABLE statements in figure 1 place XML data and relational data of a table into in the same table space. This means they use the same page size and are buffered in the same buffer pool. Within the table space, the relational data is stored in base table data pages, while the XML data is stored separately in XML Data Area (XDA) pages. Storage is separated because XML documents, like large objects (LOBs), can be too large to fit within a single row on a data page of the table. This default layout can provide good performance for most application scenarios. You do have the option of storing XML documents together with relational data in the data pages; this option, called "base table inlining", is described in the following section. However, by default, relational and XML data are stored separately.

If you have done a performance analysis and find that you need a large page size for XML data but a small page size for relational data or indexes, you can use separate table spaces to achieve this. When you define a table, you can direct "long" data into a separate table space with a different page size. Long data includes LOB and XML data.

The following example defines two buffer pools and two table spaces, one each with 4 KB and 32 KB pages. A table space always requires a buffer pool with a matching page size. Our three tables are assigned to table space relData with 4 KB pages. All of the columns are stored in that table space, except the XML columns. They are stored on 32 KB pages in table space xmlData.

```
create bufferpool bp4k pagesize 4k;
create bufferpool bp32k pagesize 32k;

create tablespace relData
pagesize 4K bufferpool bp4k;

create tablespace xmlData
pagesize 32K bufferpool bp32k;

create table trades (tradeId integer, tradedoc XML)
     in relData
     long in xmlData;

create table parties(partyInfo XML) in relData long in xmlData;

create table currencies(symbol char(4), name varchar(30), USDvalue double,
                        lastUpdated timestamp) in relData;
```

Figure 6: Sample tables with XML and relational data in different table spaces with different page sizes

Unless explicitly specified, new table spaces are created as DMS with large row IDs. This means that a table space with 4 KB pages can grow to 2 TB and supports up to 2335 rows per 32 KB page. You generally do not have to choose a smaller page size due to limitations on the number of rows per page.

Inlining and compression of XML data

A storage option with potential performance advantages for XML data is "inlining," which stores XML data physically together with the relational data in the same row.

If some or all of your XML documents are small enough to fit into their corresponding row on the base table page in the table object, they can be inlined into the relational row.

This provides more direct access to the XML data and avoids the redirection to the XDA object. If some documents in the XML column are still too large to be inlined, they are stored "outlined" in the XDA object as usual. Inlining can reduce the size of the regions index dramatically, since inlined documents do not require any regions index entries. They always consist of a single, inlined region. Inlining is requested as part of the XML column definition; you specify the maximum length of XML documents that are to be inlined, as shown in figure 7.

```
create table trades (tradeId integer, tradedoc XML inline length 16000)
     in relData;
```

Figure 7: Inlined XML storage definition

In this example, the XML column is defined with the option INLINE LENGTH 16000. This means that any document that can be stored in 16000 bytes or less will be inlined. The size specified here refers to the size of the document after XML parsing in DB2, not the size of the textual XML document in your file system. The inline length must be smaller than the page size minus the size of the other columns in the table.

If you have only one XML column in the table, the easiest way to inline your documents is to set the inline length to the largest value possible. This value is roughly the page size for the table minus the total byte count of the relational columns in the row. DB2 will not let you define an inline length that is too long. Choosing a large inline length will not waste any space, since only the space needed to store the document is actually used. Specifying a large inline length might, however, prevent you from adding new columns to the table in the future.

DB2 Version 9.7 includes new functions called ADMIN_EST_INLINE_LENGTH to help you decide on an appropriate inline length for a particular column and ADMIN_IS_INLINED to see which documents in a column are already inlined. For example, the following statement provides the estimated required inline length for documents in the TRADEDOC column and indicates whether they are already inlined. The output is 1 or 0 to indicate inlined or not inlined, respectively.

```
select tradeId, admin_est_inline_length(tradedoc), admin_is_inlined(tradedoc) from
trades
```

Figure 8: Displaying inline length and inline status for XML documents

Inlined XML data always resides in the same table space as the relational columns of the table and cannot be stored on a different page size or in a separate table space.

Inlining XML documents can improve the performance of table scans and list-prefetch index scans because it enables the documents to benefit from relational data page prefetching. Queries that may have been I/O bound without inlining might be significantly faster with it, since DB2 spends less time waiting for physical I/O. If a large percentage of your documents are inlined, then the DB2 internal regions index is very small, which saves disk space and can further contribute to better query performance. In

DB2 9.5 only, inlining is also a prerequisite for table compression, which often results in considerable performance improvement.

However, inlining also significantly increases the row size on your data pages. This in turn decreases the number of rows stored per page. Queries that only access the relational columns of the table now need to read a much larger number of pages than without inlining. This can lead to more I/O and lower performance for these queries than if XML base table inlining was not used. If your queries typically always touch the XML column, then this does not affect you. However, if you have many queries that don't involve the XML column, then it might be better not choose to store it inlined.

Another important storage option for tables that include XML data is table compression. Compressing XML data can provide a tremendous performance boost if your system is I/O-bound rather than CPU-bound. In addition to reducing the amount of physical disk space required to store the data, your buffer pool can hold more data, potentially improving buffer pool hit ratios.

In DB2 9.5 only, inlining is required for XML documents to be compressed. Compression of both the relational and the XML data in a row is achieved by using the INLINE LENGTH option for the XML column and the COMPRESS YES attribute for the table, as shown in Figure 9.

```
create table trades (tradeId integer, tradedoc XML inline length 16000)
     in relData compress yes;
```

Figure 9: Inlined and compressed XML storage definition

In DB2 9.7, inlining and compression are independent of each other. That is, tables containing XML data can be compressed even if the XML data is not inlined, as shown in Figure 10. Compression and inlining can still be combined, as shown above.

```
create table trades (tradeId integer, tradedoc XML)
     in relData compress yes;
```

Figure 10: Compression of a table with XML data that is not inlined in DB2 9.7

It is not uncommon to compress XML data by 60 to 70 percent. The following statement can be used to check the compression ratio of the TRADES table:

```
select tabname,pages_saved_percent,bytes_saved_percent
from table(sysproc.admin_get_tab_compress_info('MYSCHEMA','TRADES','ESTIMATE'))
      as t
```

Figure 11: Administration function to check the data compression ratio

In summary, you should almost always use table compression with tables having XML columns unless the workload accessing them is already CPU-bound. Inlining of a particular XML column makes sense for I/O-bound workloads in which the majority of statements involve that XML column.

Guidelines for adding XML data to a DB2 database

The first challenge in our sample scenario is to insert large amounts of FpML trades into DB2. The following sections describe techniques for efficiently adding XML data into a DB2 database.

Inserting XML documents with high performance

DB2 provides three options for moving XML data into a DB2 table: insert, import and load. The load support for XML is available since DB2 9.5. Insert and import have similar characteristics from a performance and tuning point of view because the import utility actually executes a series of inserts.

The advantages of the load utility are the same for XML as for relational data: The data does not get logged and parallelism is automatically used to increase performance. DB2 determines a default degree of parallelism based on the number of CPUs and table space containers. On the other hand, benefits of import and insert include that data can be added to the table while other transactions are in progress, all constraints are verified, and triggers are fired.

In some cases it can be useful to load XML data into a staging table and to populate the actual target table(s) by performing ETL (extract, transform, and load) operations with the staging table as input. For example, a staging table can be helpful if values and fragments within each XML document are used to populate multiple hybrid tables based on specific business logic.

Whether you use insert, import or load, the following performance guidelines apply:

- As a key prerequisite, be sure to use DMS table spaces with a large page size, such as 16KB or 32KB.
- Even if you have not defined any indexes on the target table, the DB2 pureXML storage mechanism transparently maintains regions and path indexes for efficient XML storage access. Thus, you should provide sufficient buffer pool space to support index reads.
- If you have multiple user-defined XML indexes, it is typically better to define them before any bulk insert rather than afterwards. During insert, each XML document will be processed only once to generate index entries for all XML indexes. However, if you issue multiple CREATE INDEX statements later, all documents in the XML column will be read multiple times.

Additionally, consider the following guidelines for insert and import operations:

- Increasing the log buffer size (LOGBUFSZ) and the log file size (LOGFILSIZ) helps insert performance. This is particularly important for XML inserts since the data volume per row tends to be a lot bigger than for relational data. A fast I/O device for the log is recommended.

- If you use import, a small value for the COMMITCOUNT parameter tends to hurt performance. Committing every 100 rows or more will perform better than committing every row. You can also omit the COMMITCOUNT parameter and let DB2 commit as often as appropriate.

- The ALTER TABLE…APPEND ON statement enables append mode for the table. New data is appended to the end of the table instead of searching for free space on existing pages. See the DB2 documentation for further details and guidelines.

Splitting large XML documents into smaller pieces

If a large number of XML documents need to transmitted or moved, it's quite common to combine them into a single large XML document. It's often easier to handle a single large file than thousands of small files. Figure 12 shows the structure of a compound document, a single XML document that uses the root element <alltrades> to encapsulate a sequence of FpML documents.

When you receive a compound document, you will want to split it into its individual trades and insert one trade document per row in the TRADES table. This will help provide better query and update performance than storing the large compound document in one piece.

```
<alltrades>
  <FpML>
    <trade>
     ...
    </trade>
  </FpML>
  <FpML>
    <trade>
     ...
    </trade>
  </FpML>
 ...
  <FpML>
    <trade>
     ...
    </trade>
  </FpML>
<alltrades>
```

Figure 12: Structure of a compound XML document

DB2 can add a single XML document up to 2GB in size and you can split the 2GB document into smaller documents, as shown in figure 13. The XMLTABLE function in this insert statement produces one row per trade (per <FpML> element) in a single column of type XML. The "?" in the passing clause denotes a parameter marker that provides the XML document as input to the statement. The cast (? as XML) is used to cast the parameter marker to type XML.

The XML data model requires a parsed, well-formed XML document to have one document node (as a parent of the single root element of the document). This document node is not visible in the textual (serialized) representation of an XML document. The trade sub-trees extracted from the input document do not have document nodes, and hence cannot be inserted as well-formed documents. Therefore, the document{} constructor creates a document node for each extracted trade.

```
INSERT INTO trades(tradedoc)
SELECT doc FROM
  XMLTABLE ('$d/alltrades/FpML' passing cast(? as XML) as "d"
    COLUMNS
      doc XML PATH 'document{.}') AS X
```

Figure 13: Split the large input document into individual trade documents

The TRADES table also has a TRADEID column of type integer. This column does not get populated by the above insert. Depending on your exact application requirements, you could define the TRADEID column as an automatically generated identity column. If the table with the XML column also contains relational columns, it can be advantageous to populate these with values extracted from the XML document. This is discussed in the next section.

Storing trades in a hybrid fashion

In figure 14 you see a second version of the TRADES table which keeps the trade date, the trade type and the IDs of the involved parties in relational columns. This is in addition to the full FpML document in the XML column, and therefore called hybrid storage.

Extracting selected XML element values into relational columns in the same row as the XML document can serve a variety of purposes. Relational columns allow easy SQL-only access to important or frequently accessed data items, the definition of primary key, foreign key or other constraints, and the definition of multi-column (composite key) relational indexes. The extraction of XML element or attribute values into relational columns can also be automated with triggers [1] or with UDFs and generated columns [3].

```
create table trades2 (trdDate Date, trdType varchar(20), partyId1 integer,
                      partyId2 integer, tradedoc XML);

insert into trades2
select *
from xmltable('$d/FpML' passing cast(? as XML) as "d"
  columns
    tradeDate    date          path     'trade/tradeHeader/tradeDate',
    tradetype    varchar(20)   path     'trade/*[2]/local-name(.)',
    partyId1     integer       path     'party[@id="party1"]/partyId' ,
    partyId2     integer       path     'party[@id="party2"]/partyId',
    doc          XML           path     'document{.}')
     )  as t;
```

Figure 14: Hybrid storage of FpML trades

The INSERT statement in figure 14 extracts exactly those items from the document for which we have relational columns. Hence, these values are stored redundantly in the relational columns and within the XML document in the XML column.

To ensure consistency between the XML column and the relational column you can wrap the INSERT statement in a stored procedure that only takes the FpML document as an input parameter. If all inserts are performed using that stored procedure, the data in each row will remain consistent. The same can be done for updates. Inserts and updates that only touch the relational columns should be avoided.

XML Data in DPF databases

DB2 Version 9.7 and higher allows you to store XML data in a database that uses the Database Partitioning Feature (DPF). XML columns can be added to a partitioned table with the following restrictions:

- The distribution key of the table cannot include XML columns
- XML indexes defined on a hash-partitioned table cannot be unique.

The restriction on unique indexes stems from the DPF rule that any unique index must contain all columns of the distribution key. Since the distribution key cannot include XML columns, it follows that no unique index can be defined on an XML column. If your table includes both XML and relational columns, you can create a (possibly compound) distribution key on the relational columns. However, if your data is purely XML, then you will need to create a relational distribution key before inserting the data into a partitioned table. There are two ways to do this.

The first and simplest option is to add a generated column to the table and define that column as the distribution key, as shown here:

```
create table parties(id INT generated always as identity, partyInfo XML)
             distribute by hash(id);
```

Figure 15: Partitioned table with XML column and generated distribution key

If you omit the "`distribute by hash`" clause, the first column in the table is used as the distribution key by default.

If the XML data contains an element that is frequently used in join predicates, then a second option is often better. In this case, you may want to extract the value of this element into a relational column. The extracted column can be used to write queries with relational (rather than XML) join predicates and might be a good choice for the table distribution key. This is especially true if the table is to be co-located with another table to which it is frequently joined.

For example, suppose that you would like to create a table similar to the “trades2” table defined in Figure **14** above in a DPF database, and have decided that the `partyId1` column would be a good distribution key. You could define and populate the table as follows:

```
create table trades2 (partyId1 integer, partyId2 integer, tradedoc XML)
             distribute by hash(partyId1);

insert into trades2
select *
from xmltable('$d/FpML' passing cast(? as XML) as "d"
  columns
    partyId1     integer       path     'party[@id="party1"]/partyId' ,
    partyId2     integer       path     'party[@id="party2"]/partyId',
    doc          XML           path     'document{.}')
     )  as t;
```

Figure 16: Use of an extracted XML value as a DPF distribution key

The extraction of XML values into relational columns can also be automated with triggers [1]**Error! Reference source not found.** or with UDFs and generated columns [3]. Inserts into a partitioned table that require extraction of an XML value in order to populate the relational distribution key will utilize the coordinator partition more heavily than similar inserts not requiring XML extraction. When XML data is inserted into a partitioned table having a distribution key that does not depend on the XML data, parsing of the XML data need not happen at the coordinator. Parsing can be delayed so that it takes place at the partition into which the data is inserted. However, if the value of the distribution key depends on extraction from XML, the XML document must be parsed at the coordinator partition performing the inserts.

Ensuring XML data quality at insert time

The preferred way to ensure data quality for XML documents is to register an appropriate XML schema in DB2 and to use it for document validation upon insert or update. In many XML applications, including our FpML scenario, XML schemas keep evolving over time. To validate a variety of incoming FpML documents you might need to register multiple versions of the FpML schema in DB2, such as versions 4.1, 4.2, 4.3 and eventually FpML 5.0.

Remember that validating XML documents against a schema is optional in DB2. There is no disadvantage in terms of performance or functionality if you don’t validate.

If the XML documents have already been validated in the application layer, of if they come from a trusted source, then validation in DB2 might not be required. In that case, you might want to insert without validation and save the extra CPU consumption that validation incurs.

If you choose to validate at the DB2 level, you can do so in several ways. One is to use the XMLVALIDATE function explicitly in insert statements, as shown in figure 17. This is very flexible and allows you to control validation on a per-document basis. Documents based on different versions of the FpML standard can be validated against the respective version of the FpML schema and still be stored in the same XML column.

```
insert into trades (tradeID, tradedoc)
    values (? , xmlvalidate(? according to xml schema ID matthias.fpml43))
```

Figure 17: Explicit validation of trade documents in insert statements

The drawback of this approach is that every insert statement in every application needs to use the XMLVALIDATE function and reference the proper XML schema identifier (which is assigned when a schema is registered). In other words, ensuring data quality in this manner is a distributed responsibility and there is no central control. However, DB2 has a couple of features that add more central control over the admissible XML data.

You can define check constraints to ensure that XML documents are rejected if they have not been properly validated with the XMLVALIDATE function. The first check constraint in figure 18 allows the insertion of XML documents only if they are successfully validated with the XMLVALIDATE function against any XML schema that is registered in DB2. Inserts without successful validation are rejected. The second check constraint is stronger because it only allows insertion of documents which have been validated against any of the schemas with the schema identifiers listed in the IN clause. The two check constraints are equivalent if the only XML schemas registered in the database are matthias.fpml41, matthias.fpml43, matthias.fpml50.

```
alter table trades add constraint chkvalidated1
      check (tradedoc is validated)

alter table trades add constraint chkvalidated2
      check (tradedoc is validated according to xmlschema id
                     IN (matthias.fpml41, matthias.fpml43, matthias.fpml50)
```

Figure 18: Check constraints that reject non-validated document

The check constraints themselves do not perform or cause any schema validation. They only check whether a document that is to be inserted (or updated) has been properly validated.

However, you can use a BEFORE trigger to automatically force document validation (figure 19). To avoid the overhead of double validation, you can validate either explicitly in insert statements or with a trigger, but not both.

```
create trigger tr1 before insert on trades
referencing new as n
for each row mode db2sql
begin atomic
set (n.tradedoc) = xmlvalidate(n.tradedoc according to
                                      xmlschema id matthias.fpml43 );
end
```

Figure 19: BEFORE Trigger to force document validation

If you have validated different documents in your database against different schemas, there might be situations when you want to find the schema that was used to validate a given document; or, you might want to find all documents that were validated against a certain schema. The scalar function XMLXSROBJECTID can do this for you. It takes an

XML document (or any piece of it) as input and returns the ID of the XML schema against which the document was validated. With the XML schema ID, you can find the XML schema in the DB2 XML schema repository (XSR). Consider the following examples:

```
select xmlxsrobjectid(tradedoc)
from trades
where xmlexists...;

select T.tradedoc
from trades T, syscat.xsrobjects X
where xmlxsrobjectid(T.tradedoc) = X.objectid
  and X.objectname = 'fpml43';
```

Figure 20: Using the function XMLXSROBJECTID

For further information on managing documents for multiple XML schemas, see [13].

How to query XML data efficiently and effectively

In this section, we'll talk about when to use SQL/XML or XQuery to write queries, with examples using our sample database. We'll discuss aspects of both types of queries that tend to be tricky, including use of the wildcard characters // and *, effectively limiting rows in SQL/XML queries, common errors with XMLEXISTS predicates, and dealing with XML attribute data. Other topics include writing joins involving XML data and tips on XMLTABLE queries.

Choosing SQL/XML vs. XQuery: What are the pros and cons of each?

You can express many queries in plain XQuery, in SQL/XML, or XQuery with embedded SQL. In certain cases you might find one of the options more intuitive to express your query logic than others. In general, the correct approach for querying XML data needs to be chosen on a case-by-case basis, taking the application's requirements and characteristics into account. However, we can summarize the following guidelines:

- **Plain SQL without any XQuery or XPath** is really only useful for full-document retrieval and operations such as insert, delete, and update of whole documents. Selection of documents must be based on non-XML columns in the same table.

- **SQL/XML with XQuery or XPath embedded in SQL statements** provides the broadest functionality and the fewest restrictions. You can express predicates on XML columns, extract document fragments, pass parameter markers to XQuery expressions, use full-text search, aggregation and grouping at the SQL level, and you can combine and join XML with relational data in a flexible manner. Most applications are well served by this approach.

- **XQuery** is a powerful query language, specifically designed for querying XML data. As such, it is a good option if your applications require querying and manipulating XML data only, and do not involve any relational data. This might sometimes be simpler and more intuitive. Also, if you are migrating from an XML-only database to DB2 and already have existing XQueries, you might prefer to stick with plain XQuery.

- **XQuery with embedded SQL** can be a good choice if you want to leverage relational predicates and indexes as well as full-text search to pre-filter the documents from an XML column which are then input to an XQuery. SQL embedded in XQuery also allows you to run external functions against the XML columns. But, if you need to perform data analysis queries with grouping and aggregations, you might prefer SQL/XML.

In the following sections, we'll show examples illustrating the advantages and disadvantages of each option. The examples refer to the sample documents in the derivative trades database introduced previously. We'll also give some tips on getting the output in the format you want.

SQL/XML (XQuery/XPath embedded in SQL)

SQL/XML is a part of the SQL language standard that defines an XML data type together with functions for querying, constructing, validating, and converting XML data. The standard includes numerous publishing functions allowing users to construct XML from relational data, as well as functions such as XMLQUERY, XMLTABLE and the XMLEXISTS predicate. These constructs allow users to embed XQuery or simple XPath expressions in SQL statements.

Use SQL/XML if:

- You have an existing SQL application and need to add some XML functionality here and there.
- You are an SQL fan and want to keep SQL as the primary language, because this is what you and your team are most familiar with.
- Your queries need to return data from relational columns and from XML columns at the same time.
- Your queries require full-text search conditions via DB2 Net Search Extender (NSE) or OmniFind® products.
- You want results returned as sets and missing XML elements represented with nulls.
- You want to use parameter markers.

Here are some examples using SQL/XML.

The XMLQUERY function is typically used in the select clause to extract XML fragments from an XML column, while XMLEXISTS is commonly used in the where clause to express predicates over XML data. Here's an example that shows how you can use SQL/XML to query XML and relational data in an integrated manner. The select clause retrieves data from both relational and XML columns, and the where clause contains both relational and XML predicates.

```
select tradeId, xmlquery('$i/FpML/trade/tradeHeader/tradeDate/text()'
                       passing tradedoc as "i") as tradeDate
from trades
where xmlexists('$i/FpML/party[partyId="510026"]' passing tradedoc as "i")
    and tradeId < 600;
```

Figure 21: SQL/XML to return both relational and XML data

This sample query uses XMLEXISTS to select trades made by partyId 510026 and applies XMLQUERY to return the trade date from the corresponding documents. The results are as follows:

```
     TRADEID    TRADEDATE
         123         2001-04-29Z
         456         2001-10-23Z
```

You can simplify the query by omitting the PASSING clause in the XMLEXISTS and XMLQUERY functions. You can use column names as variables in the XQuery expression, without the PASSING clause, as shown in figure 22. We will use this simplified syntax from now on.

```
select tradeId,
      xmlquery('$TRADEDOC/FpML/trade/tradeHeader/tradeDate/text()')
        as tradeDate
from trades
where xmlexists('$TRADEDOC/FpML/party[partyId="510026"]')
   and tradeId < 600;
```

Figure 22: Alternate formulation of query in figure 21 using simplified syntax

We can express the same query using the XMLTABLE function, shown in figure 23. In this format, we specify conditions to restrict the input data and to extract the output values that we are interested in. The XQuery expression in the XMLTABLE function identifies the trades involving partyId 510026 and the path expression in the COLUMNS clause ("trade/tradeHeader/tradeDate") returns their trade dates as a SQL DATE type. Its output is the same as that of the queries in figure 21 and figure 22. This syntax for XMLTABLE avoids a PASSING clause in the XQuery expression.

```
select t.tradeId, tx.tradeDate
from trades t,
    XMLTABLE('$TRADEDOC/FpML[party/partyId="510026"]'
      COLUMNS
      tradeDate   date   path 'trade/tradeHeader/tradeDate' ) as tx
where t.tradeId < 600;
```

Figure 23: XMLTABLE variant of query in figure 21

SQL/XML is good for grouping and aggregation of XML. The XQuery language does not provide an explicit group-by construct. Although grouping and aggregation can be expressed in XQuery using self-joins, it is quite awkward. You can use SQL/XML functions such as XMLTABLE or XMLQUERY to extract the data items from XML columns and then use familiar SQL concepts to express grouping and aggregation on top of that.

For example, let's find the number of trades in which each partyId is involved. Figure 24 shows how to use XMLTABLE to extract the partyId values for subsequent SQL aggregation and grouping. Using SQL group by and aggregation functions on the result of XMLTABLE functions is often more efficient than producing the same result in plain XQuery.

```
select T.partyId , count(*)
from trades,
   xmltable('$TRADEDOC/FpML/party/partyId'
      columns
      partyId    integer     path '.'   )  as T
group by partyId;
```

Figure 24: Grouping and aggregation using XMLTABLE

The results are as follows:

```
PARTYID     COUNT
----------- -----------
      67781           4
      99114           3
     510026           3
```

Plain XQuery

All of the DB2 major application programming interfaces (APIs) support XQuery as a first-class language, just like SQL. XQuery is useful for:

- XML-only applications that do not need to use SQL or relational structures.
- Migration from an XML-only database to DB2. Existing XQueries can often run with only minor changes in DB2. For example, the input data for an XQuery comes from the DB2 function db2-fn:xmlcolumn(), while other databases might call it collection(). In this case only a simple rename is needed.
- You might find XQuery convenient to express joins between two XML documents, as well as unions of XML values.

XQuery has some disadvantages, too:

- With plain XQuery you can’t use full-text search capabilities provided by the DB2 Net Search Extender (NSE). You need to involve SQL for full-text search.
- Plain XQuery does not allow you to call SQL user defined functions (UDFs) or external UDFs written in C or Java.
- Currently, DB2 does not provide a way to invoke a stand-alone XQuery with parameter markers. In order to pass parameters to XQuery, you must use SQL/XML to convert a parameter-marker ("?") into a named variable.

XQuery with embedded SQL

XQuery alone allows you to access XML data, and only XML data. This is perfectly fine if you are dealing with XML data only, but insufficient if your applications require combined access to XML and relational data. This is possible with SQL/XML, which embeds XQuery in SQL. Conversely, embedding SQL in XQuery opens up additional possibilities.

XQuery with embedded SQL allows you to use relational predicates to restrict the input to a particular XQuery and process only a subset of the XML documents. For this purpose, DB2 provides the function db2-fn:sqlquery() to invoke an SQL query from within XQuery. This function takes an SQL SELECT statement and returns an XML column as output. For example, the query in figure 25 does not consider all documents in the XML column TRADEDOC, but has an embedded SQL statement which pre-filters the XML documents by applying a predicate on the relational column TRADEID.

```
XQUERY
for $t in db2-fn:sqlquery("select tradedoc from trades t
                          where t.tradeId >= 789")/FpML/trade
where $t/termDeposit/principal/currency="EUR"
return $t/tradeHeader/tradeDate;
```

Figure 25: Restricting input to XQuery using a SQL predicate

A regular relational index on the TRADEID column of the TRADES table can help speed up the embedded SQL query.

DB2 can use XML and relational indexes at the same time, such as relational indexes for the embedded SQL statement plus an XML index for the XML predicate $t/termDeposit/principal/currency=”EUR”.

XML query results – getting what you want

Depending on how you write a specific query, DB2 might deliver the query results in different formats. For example, plain XQuery returns items (such as elements or document fragments) in the result set as one item per row, even if multiple items come from the same document (row) in the database. On the other hand, the SQL/XML function XMLQUERY might return multiple items in a single row. Let's look at some examples. The queries in figure 26 (SQL/XML) and figure 27 (XQuery) both ask for the partyIds in each trade document. The query in figure 26 returns one row for each trade document in the table, and each row contains both of the partyId elements in the document. PartyIds from the same document cannot be returned in separate rows because this is an SQL select statement and cannot produce more than one output row per qualifying input row:

```
select XMLQUERY('$TRADEDOC/FpML/party/partyId') from trades;
```

Figure 26: One row per document

The result set is the following:

```
<partyId>510026</partyId><partyId>67781</partyId>
<partyId>510026</partyId><partyId>99114</partyId>
<partyId>99114</partyId><partyId>67781</partyId>
<partyId>510026</partyId><partyId>67781</partyId>
<partyId>99114</partyId><partyId>67781</partyId>
```

The query in figure 27, on the other hand, returns each partyId as a separate row. This is because it's an XQuery which returns a sequence of items even of multiple items originate from the same document:

```
XQUERY db2-fn:xmlcolumn("TRADES.TRADEDOC")/FpML/party/partyId;
```

Figure 27: One row per partyId

The result set is the following:

```
<partyId>510026</partyId>
<partyId>67781</partyId>
<partyId>510026</partyId>
<partyId>99114</partyId>
<partyId>99114</partyId>
<partyId>67781</partyId>
<partyId>510026</partyId>
<partyId>67781</partyId>
<partyId>99114</partyId>
<partyId>67781</partyId>
```

Even though the result of the query in figure 27 is usually easier to consume by an application (that is one XML value at a time), one disadvantage is that you don't know which partyId elements come from the same trade document. The output of the query in figure 26 preserves this information by presenting the partyIds in pairs. If the application uses an XML parser to handle each XML result row from DB2, the first result row from figure 26 would be rejected by the parser because it is not a well-formed document (it lacks a single root element). To solve this, you can add a single root element as shown in figure 28:

```
select XMLQUERY('<partyIdList>{$TRADEDOC/FpML/party/partyId}</partyIdList>')
from trades;
```

Figure 28: Adding a root element

This changes the query result such that each result row is a well-formed XML document. A sample result row (of a total of 5) is:

<partyIdList><partyId>99114</partyId><partyId>67781</partyId></partyIdList>

If you prefer to get each partyId in a separate row, this can be achieved with the XMLTABLE function shown in figure 29:

```
select X.* from
trades,
    XMLTABLE('$TRADEDOC/FpML/party'
       COLUMNS
       PartyId    Varchar(20)  PATH 'partyId') as X;
```

Figure 29: One partyId value per row

The result set is the following:

```
510026
67781
510026
99114
99114
67781
510026
67781
99114
67781
```

If you need to combine relational and XML data, then SQL/XML is the best choice in most cases. In particular, SQL/XML is the option that allows parameter markers against XML data. If you have XML-only applications, stand-alone XQuery is a powerful choice, and can be augmented with embedded SQL to allow full-text search and invocation of UDFs.

Join queries involving XML data

The SQL/XML predicate XMLEXISTS makes it easy to join XML data and relational data. The following example selects the trades that are term deposits and retrieves their principal amounts using an XMLTABLE function. It also retrieves the currency symbol of these trades and uses it to join with the relational SYMBOL column of the CURRENCIES table. Finally, it multiplies the deposit's principal value by the currency value for that symbol to arrive at a USD amount for the principal. This query uses XMLEXISTS to perform a join between a relational value (CURRENCIES.SYMBOL) and an XML value (/FpML/trade/termDeposit/principal/currency). The relational column SYMBOL is referenced as $SYMBOL in the XMLEXISTS predicate:

```
select t.tradeid, c.symbol, c.USDvalue * tx.amount as principal_value_USD
from trades t, currencies c,
xmltable('$TRADEDOC/FpML/trade/termDeposit/principal'
      COLUMNS
      amount  double  path 'amount'
  ) as tx
where
xmlexists('
      $TRADEDOC/FpML/trade/termDeposit/principal[currency = $SYMBOL]');
```

Figure 30: Joining relational and XML data with SQL/XML

The following is the result of the query in figure 30:

```
TRADEID     SYMBOL PRINCIPAL_VALUE_USD
----------- ------ ------------------------
        790 EUR      +3.65000000000000E+007
        791 EUR      +7.30000000000000E+007
        789 GBP      +6.86000000000000E+007
```

If the relational column SYMBOL in the currencies table is defined as a fixed length character string, (for example, char(4) rather than varchar(4)), we need to remove any trailing blanks for the join to work. We can do so by writing the predicate as:

[currency = fn:normalize-space($SYMBOL)]

To accomplish this join, we passed the currency symbol into the XMLEXISTS predicate such that the actual join condition is an XQuery predicate. Conversely, we can also extract the principal currency from the XML data into the SQL context such that the join condition is an SQL predicate:

```
select t.tradeid, c.symbol, c.USDvalue * tx.amount as principal_value_USD
from trades t, currencies c,
xmltable('$TRADEDOC/FpML/trade/termDeposit/principal'
     COLUMNS
     amount   double   path 'amount'
  ) as tx
where
  c.symbol = XMLCAST(
         XMLQUERY('$TRADEDOC/FpML/trade/termDeposit/principal/currency')
         as char(3));
```

Figure 31: Join using XMLCAST of XML data and a relational join predicate

Typically, the query in figure 30 is preferable over the one in figure 31.

This is because the XMLCAST function expects a single input value. It fails in situations where the path expression inside XMLQUERY returns more than one item. This could happen if the path expression was too general – something like $TRADEDOC/FpML/trade//currency, for example, which would return multiple currency symbols for trade documents describing foreign exchange transactions. However, XMLCAST can be a very good option for joins between XML values that occur only once per document and relational values, because it allows the use of a relational index (in this case, on CURRENCIES.SYMBOL). A relational index cannot be used in figure 30 because the join condition is not a relational predicate but an XQuery predicate.

The query in figure 32 joins the XML columns of the PARTIES and TRADES tables to return the actual names of parties involved in the trades. It casts the resulting values to a SQL VARCHAR type in order to be able to use SELECT DISTINCT and eliminate duplicate party names.

```
select distinct xmlcast(xmlquery('$PARTYINFO/Party/Name')as varchar(30))
from parties p, trades t
where XMLEXISTS('$PARTYINFO/Party[PtyID = $TRADEDOC/FpML/party/partyId]');
```

Figure 32: Joining XML data with SQL/XML

The same join can also be expressed in plain XQuery with two nested FOR clauses. Intuitively you can think of them as a nested-loop join between the two tables.

```
XQUERY distinct-values(
for $tdoc in db2-fn:xmlcolumn("TRADES.TRADEDOC")/FpML
    for $pty in db2-fn:xmlcolumn("PARTIES.PARTYINFO")/Party
where $tdoc/party/partyId=$pty/PtyID
return $pty/Name);
```

Figure 33: Join between two XML documents

Join queries involving XML data in partitioned tables (DPF)

The join techniques shown in the previous section can be used without modification if the tables being joined happen to be partitioned. However, in DPF, the performance advantages of using relational –as opposed to XML- join predicates are greater than in single-partition DB2 instances.

In both DPF and single-partition DB2 instances, the number of possible join methods is greater if relational join predicates are used. In particular, hash join and merge join are available only to relational joins. The greater variety of join strategies available with relational joins may offer better query execution plans and thus higher performance

For DPF joins in particular, relational joins offer a greater choice of data redistribution strategies between partitions than do XML joins. In joins between tables that are not co-located, potentially matching rows must be moved before the join so that they reside on the same partition. Relational joins between two tables offer three such strategies:

- Broadcast a copy of one table to all of the partitions of the other
- Redistribute one table using hashing of the join key so that rows with a given join key value are sent to the partition of the other table that has potentially matching rows because it is already distributed on the join key
- Redistribute both tables by hashing on the join key so that matching rows end up on the same partition

XML joins, on the other hand, are limited to the first ("broadcast") redistribution strategy. If one of the tables being joined is small, it will typically be chosen to be broadcast and this may not be a problem. However, if both joined tables are large, broadcasting moves more data than the other two options.

Because of the greater variety of join methods and redistribution strategies available, you should strongly consider extracting likely join keys into relational columns from XML data when it is inserted, and using these relational columns rather than their equivalent XML elements when writing join predicates.

*Properly using * and // in XQuery and SQL/XML queries*

The wildcard characters * and // need to be used with care, as they can have performance implications and might return ambiguous results. Whether you use XQuery or SQL/XML there are several different path expressions that would retrieve the maturity dates or principal amounts for certain trades. For example, both

/FpML/trade/termDeposit/principal/amount as well as //principal/amount return the principal amounts for term deposit trades.

For better performance, it's best to use the fully specified path rather than * or // because DB2 can navigate directly to the desired elements, skipping over non-relevant parts of the document. If you ask for //amount instead of /FpML/trade/termDeposit/principal/amount, you ask for amount elements anywhere in the document. This requires DB2 to navigate down into every branch of the document at every level for principal/amount elements, which is avoidable overhead.

Note that * and // can also lead to undesired or unexpected query results. For example, consider the trade document in figure 4 which describes a currency exchange transaction with distinct amount elements for the two currencies involved. For this document, the path //amount would return both of the following elements without distinguishing between them:

- …/fxSingleLeg/exchangedCurrency1/paymentAmount/amount
- …/fxSingleLeg/exchangedCurrency2/paymentAmount/amount

In particular, one amount is in Chinese Yuan and one amount is in U.S. dollars, but this information is lost to an application that asks simply for //amount.

In summary, you should try to specify paths to desired elements as completely as possible in order to help obtain better performance and unambiguous results.

Limiting returned rows and avoiding empty rows in SQL/XML queries

In this section, we show how to correctly apply predicates in SQL/XML queries so that they filter out rows in the way that we intend. Suppose that you would like to display the trade dates for all trades involving partyId 510026 for the document shown in figure 5, and write the following query:

```
select xmlquery('
          $TRADEDOC/FpML[party/partyId="510026"]/trade/tradeHeader/tradeDate')
from trades;
```

Figure 34: XMLQUERY with predicate

This query is not what you want, for multiple reasons:

1. It returns the following result set, which has as many rows as there are rows in the table and includes trades made by parties other than the desired one. This is because the SQL statement has no WHERE clause and therefore cannot eliminate any rows.

```
<tradeDate>2001-04-29Z</tradeDate>
<tradeDate>2001-10-23Z</tradeDate>
<tradeDate>2001-04-29Z</tradeDate>


  5 record(s) selected
```

For each row in the table that doesn't match the predicate, a row containing an empty XML sequence is returned. This is because the XQuery expression in the XMLQUERY function is applied to one row (document) at a time and never removes a row from the result set, only modifies its value. The value produced by that XQuery is either the tradeDate element if the predicate is true, or the empty sequence otherwise. These empty rows are semantically correct (according to the SQL/XML standard) and must be returned if the query is written the way it is.

2. Performance of this query will not be good. First, an index which might exist on /FpML/party/partyId cannot be used because this query is not allowed to eliminate any rows. Second, returning many empty rows makes this query needlessly slow.

To help resolve the performance issues and get the desired output, you should use the XMLQUERY function in the SELECT clause only to extract the trade dates, and move the search condition, which should eliminate rows, into an XMLEXISTS predicate in the WHERE clause. This will allow index usage, row filtering, and avoids the overhead of empty results rows. Writing the query in the following way causes only the desired rows to be retrieved.

```
select xmlquery('$TRADEDOC/FpML/trade/tradeHeader/tradeDate' )
from trades
where xmlexists('$TRADEDOC/FpML[party/partyId="510026"]');
```

Figure 35: XMLQUERY predicate with XMLEXISTS to filter out rows

The result set is the following:

```
<tradeDate>2001-04-29Z</tradeDate>
<tradeDate>2001-10-23Z</tradeDate>
<tradeDate>2001-04-29Z</tradeDate>
  3 record(s) selected
```

In summary, predicates in the XMLQUERY function are only applied within each XML value, so they never eliminate any rows. Document- and row-filtering predicates should go into the XMLEXISTS function.

However, it is still possible to make errors inside the XMLEXISTS function that lead to unexpected results. For example, a common error is to write the previous query without square brackets in the XMLEXISTS function:

```
select xmlquery('$TRADEDOC/FpML/trade/tradeHeader/tradeDate ')
from trades
where xmlexists('$TRADEDOC/FpML/party/partyId = "510026"');
```

Figure 36: Forgotten brackets in the predicate

This produces the same result as the query in figure 34, including all the unwanted empty rows. Without brackets, the XMLEXISTS predicate always evaluates to true. Hence, no rows are eliminated. This is because for a given row, the XMLEXISTS predicate evaluates to false only if the XQuery expression inside returns the empty sequence. However, without the square brackets the XQuery expression is a Boolean expression which always returns a Boolean value and never the empty sequence. Note that XMLEXISTS checks for the existence of a value and evaluates to true if a value exists, even if that value happens to be the Boolean value "false". This is the correct behavior according to the SQL/XML standard, although it's probably not what you intended to express.

Similarly, be careful not to make this same mistake when using two or more predicates, as shown in figure 37

```
select xmlquery('$TRADEDOC/FpML/trade/tradeHeader/tradeDate ')
from trades
where xmlexists('
           $TRADEDOC/FpML/party[partyId="510026"] and
           $TRADEDOC/FpML/trade/termDeposit[startDate="2002-02-14Z"]');
```

Figure 37: Incorrect use of “and”

This query uses square brackets, so what's wrong with it? The XQuery expression is still a Boolean expression because it's of the form "exp1 and exp2". So again, all rows are returned. Here is the proper way of writing this query to filter rows and allow for index usage.

```
select xmlquery('$TRADEDOC/FpML/trade/tradeHeader/tradeDate ')
from trades
where xmlexists('$TRADEDOC/FpML[party/partyId="510026" and
                 trade/termDeposit/startDate="2002-02-14Z"]');
```

Figure 38: Corrected

In summary, don't use Boolean predicates in XMLEXISTS. Put predicates in square brackets, including any "and" and "or". For further guidelines and XML query examples, see [1].

Writing XMLTABLE queries

The XMLTABLE function has a wide variety of uses. It is especially helpful when combining XML data with relational data, or for creating relational views of XML data. Here’s an XMLTABLE example that retrieves basic information about each trade from the TRADES table.

```
Select t.*
from trades, xmltable('$TRADEDOC/FpML'
columns
tradeType   varchar(20)   path  'trade/*[2]/local-name(.)',
tradeDate   date          path  'trade/tradeHeader/tradeDate',
partyId1    integer       path  'party[@id="party1"]/partyId',
partyId2    integer       path  'party[@id="party2"]/partyId'
)  as t;
```

Figure 39: An XMLTABLE query

The query produces the following output:

```
TRADETYPE            TRADEDATE  PARTYID1    PARTYID2
-------------------- ---------- ----------- -----------
bulletPayment        04/29/2001      510026       67781
fxSingleLeg          10/23/2001      510026       99114
termDeposit          02/14/2002       99114       67781
termDeposit          04/29/2001      510026       67781
termDeposit          03/26/2002       99114       67781
```

Recall that the XMLTABLE function is used in the FROM clause of the SELECT statement together with the table (TRADES) that it operates on. The XMLTABLE function is implicitly joined with the table TRADES and applied to each of its rows. The XMLTABLE function contains one row-generating XQuery expression and, in the COLUMNS clause, one or multiple column-generating expressions. In figure 39, the row-generating expression is the XPath $TRADEDOC/FpML.

The row-generating expression is applied to each XML document in the XML column and produces one or multiple FpML elements (subtrees) per document. The output of the XMLTABLE function contains one row for each FpML element. Hence, the output produced by the row-generating XQuery expression determines the cardinality of the result set of the SELECT statement.

The COLUMNS clause transforms XML data into relational data with a column having a name and a SQL data type. In figure 39, the returned rows have 4 columns each – tradeType, tradeDate, partyId1, and partyId2. The row-generating expression provides the context for the column-generating expressions. So typically, you can append the column-generating expressions to the row-generating expression to get an intuitive idea of what a given XMLTABLE function returns in its columns.

Remember that the path expressions in the COLUMNS clause must not return more than one item per row. If a path expression returns a sequence of two or more items, the XMLTABLE execution will fail, as it is not possible to convert a sequence of XML values into an atomic SQL value. This scenario is discussed later.

Generating XMLTABLE rows for a subset of your data

By default, the XMLTABLE row-generating expression is applied to each XML document in the specified column. However, you might want produce rows only for a subset of the documents based on some filtering predicate. For example, suppose you want to modify the XMLTABLE query in figure 39 so that it produces rows only for trades involving partyId 99114. You can use a WHERE clause to express this restriction in two ways, shown in figure 40. One of the two possible WHERE clauses is commented out because it is not good practice. If you define the filtering predicates on the columns generated by XMLTABLE, the XMLTABLE function must still produce rows for all of the documents, even if they don't satisfy the predicates. The predicates are applied after XMLTABLE has done its work, which is not good for performance.

```
select t.*
from trades, xmltable('$TRADEDOC/FpML'
columns
tradeType    varchar(20)   path     'trade/*[2]/local-name(.)',
tradeDate    date          path     'trade/tradeHeader/tradeDate',
partyId1     integer       path     'party[@id="party1"]/partyId',
partyId2     integer       path     'party[@id="party2"]/partyId'
)  as t

-- not good:  where partyId1 = 99114 or partyId2 = 99114

              where xmlexists('$TRADEDOC/FpML/party[partyId=99114]');
```

Figure 40: Restricting XMLTABLE with XMLEXISTS

The better approach is to use XMLEXISTS to express the filtering condition on the underlying XML column. This restricts the documents which are input to the XMLTABLE function. Alternatively, the query in figure 41 applies the XML predicate in the row-generating expression itself, in which case the XMLEXISTS is not needed. It returns the same result and performs equally well.

```
select t.*
from trades, xmltable('$TRADEDOC/FpML[party/partyId=99114]'
columns
tradeType    varchar(20)   path  'trade/*[2]/local-name(.)',
tradeDate    date          path  'trade/tradeHeader/tradeDate',
partyId1     integer       path  'party[@id="party1"]/partyId' ,
partyId2     integer       path  'party[@id="party2"]/partyId'
)    as t;
```

Figure 41: Restricting XMLTABLE in the row-generating expression

Dealing with duplicate paths in XMLTABLE queries

The path expressions in the COLUMNS clause must not produce more than one item per row, otherwise the XMLTABLE function will fail. However, there are reasonable situations where multiple items might be produced, and you might need to change the query to deal with them. For example, consider the documents in the PARTIES table shown in figure 3. A party document can contain one or multiple elements describing its rating and date. If you wanted to query this data and return a relational table with basic information about each party, you might write the following query:

```
select p.*
from parties, xmltable('$PARTYINFO/Party'
columns
PartyID    integer       path 'PtyID',
ShortName  varchar(10)   path 'ShortName',
Rating     varchar(6)    path 'Rating/RatingValue') as p;
```

Figure 42: XMLTABLE that will return an error due to multiple Ratings

For the sample documents that are used in this article, this query fails with the following error message:

```
SQL16003N An expression of data type "( item(), item()+ )" cannot
be used when the data type "VARCHAR_6" is expected.
```

This message means that the query is trying to cast an XML sequence of multiple items to a single VARCHAR value. A value of data type "(item(), item()+)" means the value is an item followed by one or more additional items. In simpler terms, this means that the value is a sequence of two or more items. This happens because the path expression "Rating/RatingValue" returns two RatingValue elements for "YourWorld Investments".

You can deal with this issue in several ways. If you only want to see only the first <Rating> element, you can use a positional predicate to explicitly return the RatingValue of the first occurrence of a Rating element only:

```
select p.*
from parties, xmltable('$PARTYINFO/Party'
columns
PartyID     integer       path 'PtyID',
ShortName   varchar(10)   path 'ShortName',
Rating      varchar(6)    path 'Rating[1]/RatingValue') as p;
```

Figure 43: Return only the first Rating element

The query returns the following output:

```
PARTYID     SHORTNAME  RATING
----------- ---------- ------
     510026 MIB        Baa1
      67781 NVB        Aa
      99114 YWI        Aaa
```

Similarly, if you always want to return the first two rating elements, you could explicitly define columns for both of them. If a party has only one rating, then the second column (RatingPrior) will be null, unless you define a default value for it. Figure 44 shows how:

```
select p.*  from parties, xmltable('$PARTYINFO/Party'
columns
PartyID        integer        path 'PtyID',
ShortName      varchar(10)    path 'ShortName',
Country        varchar(10)    path 'Address/Country',
RatingRecent   varchar(6)     path 'Rating[1]/RatingValue',
RatingPrior    varchar(6) default 'NONE'  path 'Rating[2]/RatingValue') as p;
```

Figure 44: Return the first two ratings, with a default in case the second is not present

The following is the output for the query:

```
PARTYID     SHORTNAME  RATINGRECENT RATINGPRIOR
----------- ---------- ------------ -----------
     510026 MIB        Baa1         NONE
```

```
      67781 NVB          Aa             NONE
      99114 YWI          Aaa            Aa
```

If the number of available ratings for each party varies a great deal, you might want to consider returning all of them, either as a VARCHAR containing a comma-separated list, or as an XML column containing a sequence. Figure 45 shows both options:

```
select p.*
from parties, xmltable('$PARTYINFO/Party'
columns
PartyID    integer       path 'PtyID',
Short      varchar(10)   path 'ShortName',
Country    varchar(10)   path 'Address/Country',
Rating_v   varchar(6)    path 'fn:string-join(
                               Rating/RatingValue/text(),",")',
Rating_X   XML           path 'Rating/RatingValue')
as p;
```

Figure 45: Return multiple ratings as a comma-separated-list or as an XML sequence

The following is the output for the query:

```
PARTYID SHORT RATING_V RATING_X
------- ----- -------- --------
510026 MIB    Baa1     <RatingValue>Baa1</RatingValue>
 67781 NVB    Aa       <RatingValue>Aa</RatingValue>
 99114 YWI    Aaa,Aa   <RatingValue>Aaa</RatingValue>
                           <RatingValue>Aa</RatingValue>
```

Further tips and techniques for the XMLTABLE function are provided in [1], [8] and [9].

Using relational views over XML data

It can be useful to define a view to expose XML data in relational format. Let's look at the following example where the SQL/XML function XMLTABLE is used to return values from the XML documents in the TRADES table in tabular format. The view extracts the trade ID, trade date and party ID information as relational columns as well as providing the original XML document.

```
create view tradesv (tradeId, tradeDate, partyId1, partyId2, tradedoc) as
select tradeId, t.*, tradedoc
from trades, xmltable('$TRADEDOC/FpML'
columns
tradeDate   date     path  'trade/tradeHeader/tradeDate',
partyId1    integer  path  'party[@id="party1"]/partyId',
partyId2    integer  path  'party[@id="party2"]/partyId'
) as t;
```

Figure 46: Relational view over XML data using XMLTABLE

Querying this view for all columns except the XML document TRADEDOC results in the following output:

```
TRADEID     TRADEDATE  PARTYID1    PARTYID2
----------- ---------- ----------- -----------
        123 04/29/2001      510026       67781
        456 10/23/2001      510026       99114
        789 02/14/2002       99114       67781
        790 04/29/2001      510026       67781
        791 03/26/2002       99114       67781
  5 record(s) selected.
```

We included the XML column INFO in the view definition to help query this view more efficiently. Suppose that you want to retrieve a tabular list of tradeIds, tradeDates and partyId pairs for a given partyId. Both of the following queries can do this. The query in figure 29 uses a predicate on the relational INTEGER column partyId1, which is exposed by the relational view TRADESV.

```
select tradeId, tradeDate, partyId1, partyId2 from tradesv
where partyid1 = 99114 or partyid2 = 99114;
```

Figure 47: Query over relational view with relational predicates

While this is a natural and appealing formulation, this query is very much like the one in figure 40 in that the filtering relational predicates on the XMLTABLE view cannot be applied to the underlying XML column or its indexes. A better way to write the query is to use an XMLEXISTS predicate against the document exposed by the TRADESV view:

```
select tradeId , tradeDate, partyId1 , partyId2 from tradesv where
XMLEXISTS ('$TRADEDOC/FpML/party[partyId="99114"]');
```

Figure 48: Query over relational view with XMLEXISTS predicate

This ensures that only the rows having a partyId 99114 are generated by the view, resulting in a shorter runtime, especially on a large data set.

Usage guidelines for XML indexes

The following sections describe guidelines for creating indexes over XML data and how to use the indexes effectively.

Defining indexes

Indexes are crucial for maximum performance of your XQuery and SQL/XML statements. DB2 allows you to define path-specific XML indexes on XML columns. This means you can index selected elements and attributes that are frequently used in predicates and joins. For example, the following index would help speed searches for trades having certain principal amounts:

```
create index tdpa on trades(tradedoc) generate keys using xmlpattern
'/FpML/trade/termDeposit/principal/amount' as sql double;
```

Figure 49: XML index on principal amounts

This statement will create an index over XML documents currently stored in the TRADEDOC column of the TRADES table. An index key will be created for each node that matches the specified XPath expression. Index keys are added and removed when documents are inserted, updated or deleted.

Remember that XML indexes can only be defined on a single XML column, cannot have composite keys, but might produce multiple keys per base table row. For example, an index defined on "/FpML/trade/fxSingleLeg/*/paymentAmount/amount" will create two keys for every currency exchange derivative, because each such document contains two amount elements. Remember also that XML indexes contain no entries for documents that don't contain the indexed paths. Thus, the index in figure 49 would not contain any keys for trade documents describing bullet payments or currency exchange transactions, as those don't contain the path "/FpML/trade/termDeposit/principal/amount".

Finally, remember that you can create unique XML indexes in a non-partitioned DB2 instance, but not on partitioned tables in DPF systems. The section "XML Data in DPF databases" explains why.

Like relational indexes, XML indexes consume space in the table space and must be maintained (kept current) when documents are inserted, updated or deleted. Thus, you need to balance the benefits they provide for queries against the extra resources required to store and maintain them. The size and the maintenance overhead of an XML index is directly related to the number of elements or attributes per document that match the XMLPATTERN of the index. The next section explains how to keep the size and cost of XML indexes as low as possible.

You can find more details on defining XML indexes in [1] and [6].We assume that you know the basics of specifying index paths using element and attribute names, the meaning of the "/", "//" and "*" symbols, and the necessity of including a SQL type for each index. In the following sections, we'll focus on best practices for index definitions.

Define indexes using fully specified paths whenever possible

Suppose that you need to ensure good performance for queries that ask for payment amounts in currency exchange trades. You know each currency exchange trade has the following two paths leading to a payment amount:

- /FpML/trade/fxSingleLeg/exchangedCurrency1/paymentAmount/amount
- /FpML/trade/fxSingleLeg/exchangedCurrency2/paymentAmount/amount

You decide to keep things simple and define a single index that indexes all paths ending in "paymentAmount/amount" as shown in figure 50.

```
create index tpa on trades(tradedoc) generate keys using xmlpattern
'//paymentAmount/amount' as sql double;
```

Figure 50: An index with wildcards

This index can be used for queries such as the one in figure 51, for the following reasons:

- The index path is more general than the one specified in the query.
- The data type of the literal value in the predicate (numeric) matches the type of the index.

```
select tradeid from trades t where
xmlexists('$TRADEDOC/FpML/trade/fxSingleLeg/*/paymentAmount[amount >
                                     1000000 and currency = "USD"]');
```

Figure 51: A query involving /paymentAmount/amount

However, the // in the index definition has similar drawbacks as described in "Join queries involving XML data in partitioned tables (DPF)" for queries. First, the // means that DB2 needs to perform more work looking for matching elements to index than if the path was fully specified and pointed to a specific element. Second, the // in the index pattern might select more nodes than you intended, which means that the index becomes larger than it needs to be. This can be avoided if you specify the path more precisely:

```
create index tfxpa on trades(tradedoc) generate keys using xmlpattern
'/FpML/trade/fxSingleLeg/*/paymentAmount/amount' as sql double;
```

Figure 52: A more specific index

Using XML index data types correctly

DB2 supports XML indexes with five different data types: DOUBLE, VARCHAR(n), VARCHAR HASHED, DATE and TIMESTAMP. The choice of the index data type is important, as it affects the potential use of the index in evaluating predicates. For example, the index in figure 53 can not be used by the query in figure 54.

```
create index thtd on trades(tradedoc) generate keys using xmlpattern
    '/FpML/trade/tradeHeader/tradeDate' as sql double;
```

Figure 53: An index using a numeric type

```
select *
from trades
where xmlexists('$TRADEDOC/FpML/party[partyId="510026"]')
```

Figure 54: A query with a string predicate

Why? The literal 510026 in the query's predicate is contained in double quotes, so it is interpreted as a string. Since the index data type is DOUBLE, the index cannot be used to resolve the string predicate. You would either need to define the index using a VARCHAR type (if you want string comparisons) or keep the index type as DOUBLE and remove the double quotes around the literal value (numeric comparisons).

Similarly, the index in figure 55 can not be used for the query in figure 56. This is because "2002-02-14Z" is a string literal which would match a VARCHAR index but not a DATE index.

```
create index thtd on trades(tradedoc) generate keys using xmlpattern
    '/FpML/trade/tradeHeader/tradeDate' as sql date;
```

Figure 55: An index using a DATE type

```
xquery for $i in db2-fn:xmlcolumn("TRADES.TRADEDOC")/FpML
where $i/trade[tradeHeader/tradeDate >= "2002-02-14Z"]
return $i;
```

Figure 56: A query with a VARCHAR predicate

The simple solution is to cast the literal in the query to the XML type xs:date, as shown in figure 57. This allows the DATE index to be used.

```
xquery for $i in db2-fn:xmlcolumn("TRADES.TRADEDOC")/FpML
where $i/trade[tradeHeader/tradeDate >= xs:date("2002-02-14Z")]
return $i;
```

Figure 57: Use of xs:date to allow use of a DATE index

Index data types: VARCHAR(n) and VARCHAR HASHED

VARCHAR(n) indexes are appropriate in cases where the indexed character strings have a maximum length of "n", and the indexed elements might to be used in range predicates (<, >). If you attempt to create a VARCHAR(n) index on nodes that have values of length greater than "n", the index creation will fail. Likewise, an insert or update of such a document will fail if VARCHAR(n) index is defined. Thus, a VARCHAR(n) index enforces a length limitation of "n" for the values of indexed nodes.

A VARCHAR HASHED index, on the other hand, lets you index character strings of arbitrary length. Instead of storing index key values, it stores their hash values. This type of index allows fast evaluation of equality predicates, but cannot be used for range predicates or sorting. The table in figure 58 summarizes the properties of both types of indexes for XML character data:

Index type	Useful for	Length limit for index key?
VARCHAR(N)	All predicate types, sorting	N bytes; max(N) depends on index page size
VARCHAR HASHED	- Equality predicates only - Indexing long values	

Figure 58: Characteristics of VARCHAR(n) and VARCHAR HASHED indexes

In summary, a VARCHAR HASHED index is often a good choice for elements or attributes which will only be queried with equality predicates. Especially for long data values, a VARCHAR HASHED index is smaller than a VARCHAR(n) index. VARCHAR(n) is required if the index is to be useful for evaluating range predicates.

text() in index definitions

Let's briefly review the meaning of /text() in path expressions before considering its effect in index definitions. Consider the following portion of a term deposit trade document:

```
...
   <principal>
      <currency>EUR</currency>
      <amount>25000000.00</amount>
   </principal>
...
```

Figure 59: A snippet of a term deposit

The element nodes <principal>, <currency> and <amount> (and their respective closing tags) provide structure to the document. The actual data is contained in the text nodes EUR and 25000000.00, the leaf nodes of the tree. Furthermore, each element node has a value defined as the concatenation of all text nodes in the subtree underneath that element node. For example, the value of <principal> is EUR25000000.00. Trivially, the value of <currency> is "EUR".

More generally, the value of a leaf element is identical to its text node. Since XML predicates are almost always defined on leaf nodes, using /text() in predicates usually makes no difference for query results; for example, the where clauses of the two queries in figure 60 perform exactly the same filtering on our data. However, the /text() in the return clause does make a difference. The first query returns "<currency>EUR</currency>" while the second only returns "EUR".

```
xquery for $i in db2-fn:xmlcolumn("TRADES.TRADEDOC")/FpML/trade
where $i/termDeposit/principal/currency = "EUR"
return $i/termDeposit/principal/currency;

xquery for $i in db2-fn:xmlcolumn("TRADES.TRADEDOC")/FpML/trade
where $i/termDeposit/principal/currency/text() = "EUR"
return $i/termDeposit/principal/currency/text();
```

Figure 60: A query with and without /text()

A predicate with /text() can use an index only if the index is also defined with /text(). This is to ensure correct and consistent behavior in the rare cases that predicates or indexes are defined for non-leaf elements. For example, the value of <principal> is “EUR25000000.00”, but the value of <principal>/text() is empty.

The first query in figure 60 can use the first index but not the second index in figure 61. The second query in figure 60 can use the second index but not the first index in figure 61. The general recommendation is to not use /text(), either in predicates or in index definitions.

```
create index tradeIdIdx1 on trades(tradedoc) generate keys using xmlpattern
'/FpML/trade/termDeposit/principal/currency' as sql varchar hashed;

create index tradeIdIdx2 on trades(tradedoc) generate keys using xmlpattern
'/FpML/trade/termDeposit/principal/currency/text()' as sql varchar hashed;
```

Figure 61: Index definitions with and without /text()

XML Indexes on non-leaf nodes

Let’s look again at the document snippet in figure 59 and remember that the value of <principal> is EUR25000000.00. You can use this behavior to write queries against concatenations of text nodes:

```
xquery for $doc in db2-fn:xmlcolumn("TRADES.TRADEDOC")/FpML/trade
where $doc/termDeposit/principal = "EUR25000000.00"
return $doc/termDeposit/maturityDate;
```

Figure 62: Predicate on a non-leaf node

It is possible to define indexes over such concatenations of nodes by specifying a non-leaf node in the path expression of the index:

```
create index tdp on trades(tradedoc) generate keys using xmlpattern
   '/FpML/trade/termDeposit/principal' as sql varchar(30);
```

Figure 63: Index over non-leaf node

This index is not really like a composite index in the relational sense, since its data still comes from only one XML column. However, it can be useful if you are sure that the indexed node (in this case <principal>) does not contain other children besides <currency> and <amount>, since this would include unexpected data in the index keys.

With few exceptions, it's unusual to index non-leaf nodes. You should index non-leaf nodes only if it will clearly benefit expected query predicates and you know the document structure well.

While we are thinking about non-leaf nodes, we should mention one other possible pitfall that arises when defining indexes with wildcards. Suppose you define an index on

/FpML/trade/termDeposit/*

This index will have one entry for every child element of the termDeposit element. If any of these child elements is a non-leaf node, the corresponding key value might be a concatenation of multiple text nodes. This might also unexpectedly exceed the length limitation of a VARCHAR(n) index.

Using indexes in XML queries

Having carefully defined your XML indexes, you nevertheless might sometimes find that your indexes are not used as you expect. Perhaps a query's performance is disappointing, so you check its execution plan, and discover that it is doing a table scan rather than an index scan. We'll cover some common examples to show how to be sure that your indexes are being used correctly.

Queries with wildcards

One of the two conditions for index matching is that the path expression defining the index is at least as general as the path expression in the predicate of the query. In other words, an index can only be used for a query if it is certain that the query predicate qualifies a subset (or all) of the nodes that are included in the index, but never more nodes. Figure 64 illustrates that all nodes qualified by the query predicate are a subset of the nodes contained in the index.

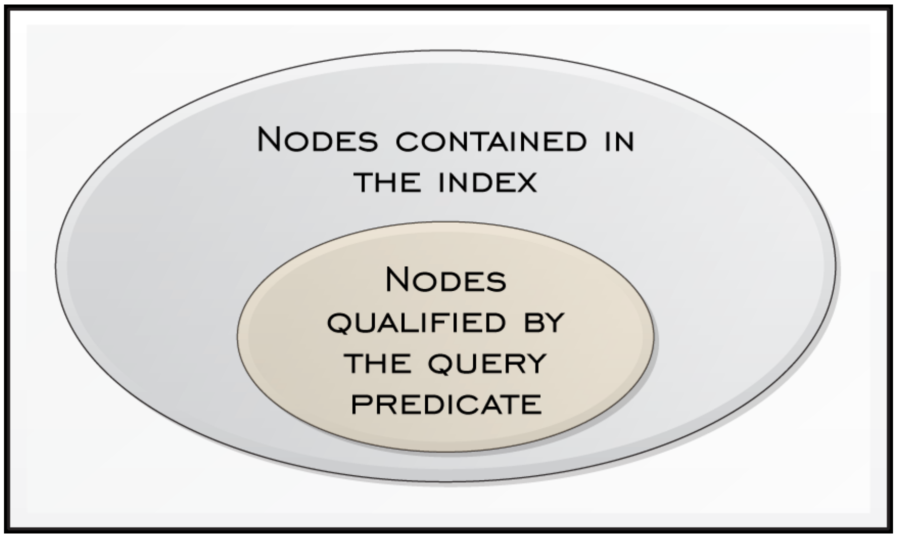
NODES CONTAINED IN THE INDEX
NODES QUALIFIED BY THE QUERY PREDICATE

Figure 64: Containment requirement for index use

A problem that can arise with queries having wildcards in their predicates is that they might qualify more nodes than are contained in a candidate index. In such a case, the containment requirement is not met and the index is not used. For example, consider the query in figure 65 which counts the number of trades of all types involving a particular currency, EUR:

```
select count(*) from trades t
where xmlexists('$TRADEDOC[FpML//currency="EUR"]');
```

Figure 65: Wildcard in query predicate

An index defined on the path /FpML/trade/fxSingleLeg//currency cannot be used to evaluate the query in figure 65. This index would supply a subset, but not all of the qualifying elements needed in the query and hence cannot be used.

While this case is obvious because our sample data is very limited, it is easy to be puzzled by the non-use of an index in cases where you are not completely familiar with the structure of your XML documents. You might not realize that a predicate with one or more wildcards qualifies more nodes than you had imagined, and in particular, more nodes than are contained in an index that you hoped would be used.

The best bet is to avoid wildcards in queries whenever possible and use fully qualified paths to the desired elements or attributes. If you do use wildcards (such as //), make sure that you do so for a well-defined reason, and not just to save typing.

Indexes in XML join predicates

In ”Using XML index data types correctly,” we saw how the type of a literal value in a query predicate defines the type of the comparison and, in turn, its eligibility for use of an index. Such a determination is usually not possible for join predicates, since there is no literal whose type can be discerned. In order to allow indexes to be used for XML join predicates, you’ll need to provide information on the type of the join keys. You can do this by using casting functions.

As an example, let’s join TRADES and PARTIES tables on the partyId element and create an appropriate index on each table:

```
create index tptyid on trades(tradedoc) generate keys using
xmlpattern '/FpML/party/partyId' as sql double;

create index pptyid on parties(partyinfo) generate keys using
xmlpattern '/Party/PtyID' as sql double;
```

Figure 66: Indexes on likely join keys

Then we perform a join to get a list of party names and their trading dates:

```
XQUERY
for $pdoc in db2-fn:xmlcolumn("PARTIES.PARTYINFO")/Party
   for $tdoc in db2-fn:xmlcolumn("TRADES.TRADEDOC")/FpML
where $tdoc/party/partyId=$pdoc/PtyID
return
<result>{$tdoc/trade/tradeHeader/tradeDate}{$pdoc/Name}</result>;
```

Figure 67: Join of TRADES table and PARTIES table

This query uses neither of the indexes defined in figure 66. The problem is that the join query contains no information about the possible data types of partyId and PtyID. We might know that party ID values are numeric values, but DB2 does not, so it must search for matching values of any type. The indexes defined are limited to key values that are numeric (sql double) and thus would not contain key values of other types, such as alphanumeric character strings. As a result, DB2 cannot rely on these indexes to perform the join, because they might contain only a subset of the matching values.

If you are interested only in matching Party IDs that have numeric values, then the indexes can be used. To search for numeric party ID values, use casting functions on the join keys. In this case, you can use the constructor xs:double() to indicate that they are numeric. This is shown in figure 68 in both XQuery and SQL/XML notation.

```
XQUERY
for $pdoc in db2-fn:xmlcolumn("PARTIES.PARTYINFO")/Party
for $tdoc in db2-fn:xmlcolumn("TRADES.TRADEDOC")/FpML
where $tdoc/party/partyId/xs:double(.) = $pdoc/PtyID/xs:double(.)
return <result>{$tdoc/trade/tradeHeader/tradeDate}{$pdoc/Name}</result>;

-- SQL/XML
select xmlquery('<result>{$TRADEDOC/FpML/trade/tradeHeader/tradeDate}
                 {$PARTYINFO/Party/Name}</result>')
from trades, parties
where xmlexists('$TRADEDOC/FpML/party[partyId/xs:double(.) =
                                      $PARTYINFO/Party/PtyID/xs:double(.)]');
```

Figure 68: Join with casting function on join keys

Note that in DB2 9.5 and earlier, the SQL/XML version of this query pre-determines the join order, forcing the TRADES table to be the inner table of the join, such that our index tptyid on the TRADES table can be used. This restriction has been removed in DB2 9.7; i.e. the join order and index use chosen by the optimizer is independent of the formulation of the SQL/XML query. For the XQuery version of the query, the join order chosen by the query optimizer determines whether the index is used in all releases. For further details, see [7].

In summary, to allow index use for XML join queries, always cast join predicates to the type of the XML index that should be used. Otherwise, the query semantics do not allow index usage. The following table shows the casting functions to use in the predicate for different index data types:

Index SQL type	Cast join predicate using	Comment
Double	xs: double	For any numeric comparison
varchar(n), varchar hashed	fn:string	For any string comparison
Date	xs: date	For date comparison
Timestamp	xs:dateTime	For timestamp predicates

Figure 69: Type casting functions for use in joins

Dealing with XML namespaces

XML namespaces are a W3C® XML standard for providing uniquely named elements and attributes in an XML document. XML documents might contain elements and attributes from different vocabularies but have the same name. By giving a namespace to each vocabulary, the ambiguity is resolved between identical element or attribute names. All pureXML features in DB2 support XML namespaces, such as SQL/XML, XQuery, XML indexes, and XML schema handling. We review namespace declarations, and then show how to deal with namespaces in queries and index definitions.

Declaring XML namespaces

In XML documents, XML namespaces are declared with the reserved attribute xmlns, whose value must contain a Universal Resource Identifier (URI). URIs are used as identifiers; they typically look like a URL but they don't have to point to an existing web page. A namespace declaration can also contain a prefix, used to identify elements and attributes that belong to the namespace.

Figure 70 is an INSERT of an FpML document with namespace declarations. The first xmlns attribute does not define a prefix and is therefore a default namespace.

All elements in its scope that do not have a namespace prefix automatically inherit this default namespace, attributes do not. The 2nd and the 3rd namespace declarations in the document assign their URIs to prefixes "fpml" and "xsi:" respectively. The prefix "fpml" is not used in this particular document. The prefix "xsi" is used to indicate that the attributes "schemaLocation" and "type" belong to the XML Schema namespace and hence have a special meaning. For example, the attribute xsi:schemaLocation defines which XML Schema this document belongs to and where this schema can be found. Please refer to [1], [14] and [8] for further background on namespaces.

```
insert into trades values (120,
'<FpML xmlns="http://www.fpml.org/2007/FpML-4-3"
      xmlns:fpml="http://www.fpml.org/2007/FpML-4-3"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      version="4-3"
      xsi:schemaLocation="http://www.fpml.org/2007/FpML-4-3 ../fpml-main-4-3.xsd
                   http://www.w3.org/2000/09/xmldsig# ../xmldsig-core-schema.xsd"
      xsi:type="DataDocument">
   <trade>
     <tradeHeader>
       <partyTradeIdentifier>
          <partyReference href="party1"/>
          <tradeId tradeIdScheme="http://www.MyGlobalIntl.com
                                  /trade-id">MyGlobal123</tradeId>
       </partyTradeIdentifier>
       <partyTradeIdentifier>
          <partyReference href="party2"/>
          <tradeId tradeIdScheme="http://www.NationalV.com/trade-id">123</tradeId>
       </partyTradeIdentifier>
       <tradeDate>2001-04-29Z</tradeDate>
     </tradeHeader>
     <bulletPayment>
….
</FpML>');
```

Figure 70: Insert of document with namespace declarations (partial listing)

Namespaces in queries

Our query examples so far have assumed that element and attribute names were not part of any namespace. It's important to realize that without modification, the query examples will not return any data from XML documents that have namespaces.

The query examples will not return any data is because an element without a namespace is distinct from an element by the same name in a particular namespace. The namespace is a critical part of an element's name. Consider the following query:

```
select tradeid, t.*
from trades, xmltable('$TRADEDOC/FpML'
columns
tradeDate    date       path   'trade/tradeHeader/tradeDate',
partyId1     integer    path   'party[@id="party1"]/partyId',
partyId2     integer    path   'party[@id="party2"]/partyId'
)  as t;
```

Figure 71: Query assuming no namespaces

This query will return rows for all trade documents except the newly inserted document with TRADEID 120 from figure 70, since its path expressions and predicates don't allow for elements and attributes belonging to a namespace. A simple way to make sure that all trades that match the predicate are returned, regardless of namespaces, is to use wildcards in the namespace prefixes. The wildcard "*" matches any namespace as well as no namespace. This query returns rows for all trades, whether the documents have namespaces or not:

```
select tradeid, t.*
from trades, xmltable('$TRADEDOC/*:FpML'
columns
tradeDate    date       path  '*:trade/*:tradeHeader/*:tradeDate',
partyId1     integer    path  '*:party[@id="party1"]/*:partyId',
partyId2     integer    path  '*:party[@id="party2"]/*:partyId'
)  as t;
```

Figure 72: Query with wildcard namespace prefixes

If you know that all elements you want to query belong to a particular namespace, you can indicate the namespaces using the XMLNAMESPACES function. For XMLTABLE, the default namespace is applied to both the row-generating and all column-generating expressions.

```
select tradeid, t.*
from trades, xmltable(XMLNAMESPACES
     (DEFAULT 'http://www.fpml.org/2007/FpML-4-3'), '$TRADEDOC/FpML'
columns
tradeDate   date      path   'trade/tradeHeader/tradeDate',
partyId1    integer   path   'party[@id="party1"]/partyId',
partyId2    integer   path   'party[@id="party2"]/partyId'
)  as t;
```

Figure 73: Declaring a default element namespace

Note that the query in figure 73 returns values only from the one trade document with the matching namespace declaration, since the other documents in our sample data have no namespaces.

While a default namespace is a common solution when only one namespace is present in your documents, you need a different approach if you want to select elements and attributes from multiple specific namespaces. In that case, using namespace prefixes in your query is the best option. See [14] and [8] for more detailed examples.

Note that other SQL/XML functions require similar attention when dealing with namespaces. In figure 74 the “fpml” prefix is defined in both the XMLQUERY and XMLEXISTS functions of this query. There no construct that allows you to define a namespace for all SQL/XML functions in a query, or even for an entire session.

```
select tradeid, xmlquery(
   'declare namespace fpml="http://www.fpml.org/2007/FpML-4-3";
    $TRADEDOC/fpml:FpML/fpml:trade/fpml:tradeHeader/fpml:partyTradeIdentifier')
from trades
where
xmlexists('declare namespace fpml="http://www.fpml.org/2007/FpML-4-3";
          $TRADEDOC/fpml:FpML/fpml:trade/fpml:tradeHeader[
                                 fpml:tradeDate=xs:date("2001-04-29Z")]')%
```

Figure 74: Namespace declarations in XMLQUERY and XMLEXISTS functions

In summary, be careful when writing queries against documents with one or more namespaces. Any XPath expressions need to include suitable default namespace or namespace prefix definitions otherwise your queries will not return the results you expect.

Namespaces in index definitions

The existence of namespaces in documents also affects their eligibility for inclusion in indexes.

Index definitions for documents with namespaces need to include the same namespaces in their XML pattern expression. None of the indexes defined in ”Defining indexes” will contain entries for the document added to the TRADES table in figure 70, since the index definitions specify elements without a namespace.

Figure 75 shows two examples of how to rewrite the index definition in figure 55 so that it includes the new document. The first defines a namespace prefix and uses it in the element names of the defining path. The second simply defines a default element namespace, which applies to all elements in the path. Although the notations are different, both definitions are equivalent and you can use either one, but not both.

```
create index thtdns1 on trades(tradedoc) generate keys using xmlpattern
  'declare namespace fpml="http://www.fpml.org/2007/FpML-4-3";
   /fpml:FpML/fpml:trade/fpml:tradeHeader/fpml:tradeDate' as sql date %

create index thtdns2 on trades(tradedoc) generate keys using xmlpattern
  'declare default element namespace  "http://www.fpml.org/2007/FpML-4-3";
  /FpML/trade/tradeHeader/tradeDate' as sql date %
```

Figure 75: Index definitions with namespace declarations

Either index could be used in the following query. Although the query specifies an explicit namespace prefix, the second index, thtdns2, could be used as well because it is logically equivalent to the first.

```
select count(*)
from trades where
xmlexists('declare namespace fpml="http://www.fpml.org/2007/FpML-4-3";
        $TRADEDOC/fpml:FpML/fpml:trade/fpml:tradeHeader[fpml:tradeDate
                                    =xs:date("2001-04-29Z")]')%
```

Figure 76: A query with namespaces which could use either index

However, while both index definitions in figure 75 will include the trade document with the namespace definitions, neither of them will include any of the remaining trade documents, which have no namespace definitions. If you wanted to include both types of documents in the index, you could use wildcards in the namespace prefix:

```
create index thtdns3 on trades(tradedoc) generate keys using
xmlpattern '/*:FpML/*:trade/*:tradeHeader/*:tradeDate' as sql date %
```

Figure 77: Index definition with wildcards in the namespace prefix

This index would also be eligible to be used in the query shown in figure 76, since it includes trade documents with and without namespace declarations. This brings us back to the “containment” requirement for index eligibility discussed in ”Queries with wildcards.” A path expression with namespace wildcards potentially contains a larger set of XML elements (any namespace) than the same XPath expression with one specific namespace. Hence, the rules for index matching with namespaces are a natural extension of the rules that we know already. For further details, see [6].

Effectively updating XML data

DB2 Version 9.5 and later support the standardized XQuery Update Facility which allows you to make changes inside an XML document at the DB2 storage level without having to read and parse the document in your application. This can help to significantly improve application performance and reduce application complexity. You can change the value of specific XML elements or attributes (collectively referred to as "nodes"), you can replace nodes with new nodes, delete or rename nodes, or insert new nodes at specific locations in the document.

Guidelines for simple XML updates

Consider the interest rate derivative document from figure 4. Suppose you want to update the tradeId from party1 in this trade. Since there are two parties to every trade, it is important that the update expression contains a predicate to pick the right node to update. In other words, the path to the target node of the update has to specify exactly one node, not zero and not more than one. Otherwise the update fails. The update in figure 78 would fail with error SQL16085N (indicating that the node being updated was not valid) if it did not contain the predicate [partyReference/@href="party1"]. Without the predicate, two tradeId elements qualify as nodes to be updated, not one. The update also fails if you misspell any of the elements in the path, in which case zero nodes are qualified for update.

```
update trades
set tradedoc = xmlquery('
     copy $new := $TRADEDOC
     modify do replace value of
$new/FpML/trade/tradeHeader/partyTradeIdentifier[
                                partyReference/@href="party1"]/tradeId
     with "MyGlobal999"
    return $new')
where tradeid = 456;
```

Figure 78: Updating the tradeId

If you need to update multiple elements in a document, combine the updates in a single update statement. Assume we need to change the currency and the amount of the principal of the FpML trade shown in figure 5. Figure 79 shows how the modify clause can contain a list of update expressions to update several nodes in a document. This is more efficient than issuing multiple update statements within a single transaction.

```
update trades
set tradedoc = xmlquery('
       copy $new := $TRADEDOC
       modify (
         do replace value of $new/FpML/trade/termDeposit/principal/currency
            with "EUR",
         do replace value of $new/FpML/trade/termDeposit/principal/amount
            with 40000000  )
       return $new')
where tradeid = 789;
```

Figure 79: Updating two nodes at the same time

Further examples of XML updates can be found in [10].

Merging data from multiple XML documents

When trade documents are retrieved from the database for consumption by a particular application, they might need to be enriched with additional information, such as further detail on the trading parties. As an example, we read trade documents and for each party we insert the name and rating information which we retrieve from the XML documents in the PARTIES table. The result is a single document that can be sent as a message (for example, via a web service), to another application.

The query in figure 80 does exactly that. It reads the trade with tradeId 123 and uses the XMLQUERY function in the SELECT clause to apply an XQuery update expression to the trade document. The COPY clause assigns the XML document from the XML column TRADEDOC to the variable $new. The MODIFY clause contains an XQuery iteration (“for $i in $new/…”) to iterate over the party elements in the document. For each party element, an insert operation is performed. The XML elements Name and Rating, which are to be inserted, are retrieved from the PARTYINFO XML column of the PARTIES table via a join on the partyId value.

```
select xmlquery ('
      copy $new := $TRADEDOC
      modify
         for $i in $new/FpML/party
           return do insert
  db2-fn:xmlcolumn("PARTIES.PARTYINFO")/Party[PtyID=$i/partyId]/(Name,Rating)
                   into $i
      return $new' )
from trades
where tradeId=123;
```

Figure 80: Inserting XML data into a trade document

Figure 81 shows the tail of the trade document retrieved by the query above. The insertion of the name and rating information happens on-the-fly during query processing. It is not a permanent update of the document in the database.

```
..(...).
  </trade>
..<party id="party1">
    <partyId>510026</partyId>
    <Name>MyGlobal International Bank</Name>
    <Rating>
      <RatingDate>2006-05-16</RatingDate>
      <RatingValue>Baa1</RatingValue>
    </Rating>
  </party>
  <party id="party2">
    <partyId>67781</partyId>
    <Name>National Village Bank</Name>
    <Rating>
      <RatingDate>2006-06-01</RatingDate>
      <RatingValue>Aa</RatingValue>
    </Rating>
  </party>
</FpML>
```

Figure 81: Additional information added to the party elements of a trade document

Enriching XML documents with relational data

What if the parties table was purely relational and looked like in figure 82?

```
create table parties2 (
       ptyId         BIGINT,
       shortname     VARCHAR(10),
       name          VARCHAR(30),
       status        VARCHAR(30),
       ratingDate    DATE,
       ratingValue   VARCHAR(10) ) ;
```

Figure 82: Relational table containing party rating information

We can still read the party names and ratings and insert them as XML elements into a selected trade document. This is done in figure 83 which differs in two significant ways from the previous query in figure 80. First, the new XML elements Name and Rating are constructed using direct element constructors. Their values are taken from relational columns which are referenced as variables, such as {$NAME}. Second, the join is established by passing the value “$i/partyId” as a parameter into the embedded SQL statement. The function db2-fn:sqlquery() has two parameters here, the SQL statement and the parameter. Inside the SQL statement the value of “$i/partyId” is referenced via the function “parameter(1)”. This kind of parameter passing is supported in DB2 Version 9.5 and later.

```
select xmlquery ('
   copy $new := $TRADEDOC
   modify for $i in $new/FpML/party
      return do insert
          db2-fn:sqlquery("select xmlquery(''
                          <Name>{$NAME}</Name>,
                          <Rating>
                             <RatingDate>{$RATINGDATE}</RatingDate>
                             <RatingValue>{$RATINGVALUE}</RatingValue>
                          </Rating>'')
                          from parties2
                          where ptyID=parameter(1)", $i/partyId)
           into $i
     return $new' )
from trades
where tradeId=123;
```

Figure 83: Inserting relational data as XML elements into a trade document

Maintaining and monitoring an XML database

If you're a DBA, the good news about maintaining and monitoring a database with XML data is that the presence of XML doesn't introduce any fundamentally new tasks. While the DB2 commands for the most common maintenance and monitoring tasks remain largely unchanged with pureXML, there are a few things you should know about their use when XML data is present.

Collecting statistics on XML data

The **runstats** command has been extended to collect statistics on XML data and XML indexes. The DB2 cost-based optimizer uses these statistics to generate efficient execution plans for XQuery and SQL/XML queries. Thus, you can continue to use the **runstats** command as you do for relational data.

If your table contains relational and XML data and you want to refresh the relational statistics only, you can execute the **runstats** command with the EXCLUDING XML COLUMNS clause. Without this clause, the default and preferred behavior is to always collect statistics for relational and XML data. Distribution statistics are currently collected only for relational columns in tables.

For relational data, as well as XML data, you can enable sampling to reduce the time for executing the **runstats** command. On a large data set, the statistics from 10% of the data are often still sufficiently representative of the total population. Whatever sampling percentage you choose, the **runstats** command allows you to sample rows (Bernoulli sampling) or pages (system sampling).

Figure 84 shows some examples. The first **runstats** command collects the most comprehensive and detailed statistics for the TRADES table and all of its indexes without sampling. This is ideal if execution time allows. The second **runstats** command collects the same statistics but only for 10% of the pages. In many cases, this will provide the optimizer with nearly as accurate statistics as the first command, but takes less time. The third command samples 15% of all rows, does not collect distribution statistics, and also applies sampling to indexes which the first and second commands didn't.

```
runstats on table myschema.trades
with distribution and detailed indexes all;

runstats on table myschema.trades
with distribution and detailed indexes all tablesample system (10);

runstats on table myschema.trades
and sample detailed indexes all tablesample bernoulli (15);
```

Figure 84: Use the **runstats** command to collect statistics

While relational statistics are visible in catalog tables, statistics for XML columns are stored internally in the table's packed descriptor and are not readily visible.

However, the **db2cat** utility can be used to dump XML statistics for an XML column into a text file, if that should be necessary for support purposes.

Statistics for XML indexes are represented in the catalog table SYSSTAT.INDEXES much like for relational indexes. However, note that each XML index is represented by a logical and a physical index. The logical index contains the index definition and has the name that you provide in the create index statement. The corresponding physical index contains the actual B-tree structure and has a system-generated name. XML index statistics are associated with the physical index, not the logical index. The query in figure 85 shows the association of logical and physical indexes, as well as other useful information, in SYSCAT.INDEXXMLPATTERNS:

```
select indname, pindname, pattern, datatype from syscat.indexxmlpatterns;
```

Figure 85: Showing association between logical and physical indexes

Monitoring XML workloads

Whether you are investigating the benefit of different page sizes or other aspects of XML performance, chances are you want to use the DB2 snapshot monitor as you would for relational data. And you can do just that. For example, XQuery and SQL/XML statements appear in dynamic SQL snapshots just as regular SQL statements do.

DB2 also offers buffer pool and table space snapshot monitor elements for XML data that match the existing counters for relational data and indexes. Since relational data and indexes are stored in separate storage objects within a table space, they have separate read and write counters. Similarly, DB2 Version 9.5 and later use a distinct storage object for XML data, called XDA (XML Data Area), and it too has its own buffer pool counters for these pages.

The example in figure 86 is a snippet from snapshot monitor output. You see the various snapshot monitor elements for the three different storage objects: data, index, and XDA. This allows you to monitor and analyze buffering and I/O activity for XML separately from relational data. Any activity pertaining to XML indexes is included in the existing index counters. The interpretation of the XDA counters is the same as of their corresponding relational counters. For example, a low ratio of XDA physical reads to XDA logical reads indicates a high buffer pool hit ratio for XML data, which is desirable. For more details on the buffer pool snapshot monitor elements, see the DB2 documentation.

Note that XML inlining (see ”Inlining and compression of XML data”) causes XML data to be stored in the data object instead of the XDA object. Hence, any activity on inlined XML documents is included in the data object counter, not the XDA counters.

```
Buffer pool data logical reads              = 221759
Buffer pool data physical reads             = 48580
Buffer pool temporary data logical reads    = 10730
Buffer pool temporary data physical reads   = 0
Buffer pool data writes                     = 6
Asynchronous pool data page reads           = 0
Asynchronous pool data page writes          = 6

Buffer pool index logical reads             = 8340915
Buffer pool index physical reads            = 54517
Buffer pool temporary index logical reads   = 0
Buffer pool temporary index physical reads = 0
Buffer pool index writes                    = 0
Asynchronous pool index page reads          = 0
Asynchronous pool index page writes         = 0

Buffer pool xda logical reads               = 2533633
Buffer pool xda physical reads              = 189056
Buffer pool temporary xda logical reads     = 374243
Buffer pool temporary xda physical reads    = 0
Buffer pool xda writes                      = 0
Asynchronous pool xda page reads            = 97728
Asynchronous pool xda page writes           = 0
Asynchronous data read requests             = 0
Asynchronous index read requests            = 0
Asynchronous xda read requests              = 83528
```

Figure 86: Snapshot Monitor output for data, index, and XDA storage objects

Using REORG and backup and restore utilities with XML data

The guidelines for reorganizing, backing up, and restoring XML data are no different than for relational data. The DB2 backup and restore utilities automatically include XML data. The REORG command has no impact on the tree structure in which XML documents are stored. The main effect of the REORG command for XML data is that the space left behind by deleted documents is reclaimed if the LONGLOBDATA option of the REORG command is used. There is a separate DB2 best practices document titled "Minimizing Planned Outages" that provides guidelines for the DB2 LOAD command, REORG command, and backup and restore utilities.

Developing pureXML applications

The most fundamental value of the DB2 pureXML feature for the application developer is that most manipulations of XML documents will no longer require tedious and inefficient DOM programming in the application layer. Since XML documents are stored in a parsed format in DB2 pureXML, document fragments or individual values can be extracted or updated without having to parse the XML. The application sends appropriate XML query or update statements to DB2 instead of fetching and parsing full documents. In situations where applications still require access to XML data through DOM or SAX APIs, they can use the new JDBC 4.0 features. These and other guidelines for application developers are provided in the following sections.

Use parameter markers for short XML queries

Very short database queries often execute so fast that the time to compile and optimize them is a substantial portion of their total response time. Thus, it's useful to compile ("prepare") them only once and pass predicate literal values for each execution of the query. While you cannot use SQL-style parameter markers in XQuery, the SQL/XML functions XMLQUERY, XMLTABLE and XMLEXISTS allow you to pass SQL parameter markers as variables into the embedded XQuery expression. This is recommended for applications with short and repetitive queries.

```
xquery for $t in db2-fn:xmlcolumn(‘TRADES.TRADEDOC')/FpML
where $t/party/partyId = 12345
return $t;

select tradedoc
from trades
where xmlexists('$TRADEDOC/FpML/party[partyId=12345]' );
```

Figure 87: Two XML queries with hard-coded literal values in the predicate

```
select tradedoc
from trades
where xmlexists('$TRADEDOC/FpML/party[partyId=$x]'
                               passing cast(? as integer) as "x");
```

Figure 88: SQL/XML query with parameter marker

Avoid code page conversion between DB2 and the application

XML is different from other types of data in DB2 because it can be internally and externally encoded. Internally encoded means that the encoding of the XML data can be derived from the data itself. Externally encoded means that the encoding is derived from the application code page. The data type of the application variable which you use to exchange XML data with DB2 determines how the encoding is derived. If your application uses character type variables for XML, then it is externally encoded (that is, in the application code page). If you use binary application data types, then the XML data is considered internally encoded. In that case the encoding is determined by either a

Unicode Byte-Order mark (BOM) or an encoding declaration in the XML document itself, such as

<?xml version="1.0" encoding="UTF-8" ?>

From a performance point of view, the goal is to avoid code page conversions as much as possible since they consume extra CPU cycles. Internally encoded XML data is preferred over externally encoded data because it can prevent unnecessary code page conversion. This means that in your application you should prefer binary data types over character types. For example, in CLI, when you use SQLBindParameter() to bind parameter markers to input data buffers, you should use SQL_C_BINARY data buffers rather than SQL_C_CHAR, SQL_C_DBCHAR, or SQL_C_WCHAR. When inserting XML data from Java applications, reading in the XML data as a binary stream (setBinaryStream) is better than as a string (setString). Similarly, if your Java application receives XML from DB2 and writes it to a file, code page conversion might occur if the XML is written as non-binary data.

When you retrieve XML data from DB2 into your application, it is serialized. Serialization is the inverse operation of XML parsing. It is the process of converting DB2's internal XML format (a parsed, tree-like representation) into the textual XML format that your application can understand. In most cases it is best to let DB2 perform implicit serialization. This means your SQL/XML statements simply select XML-type values, and DB2 performs the serialization into your application variables as efficiently as possible. You do not need to use the XMLSERIALIZE function explicitly.

Access XML data in DB2 via a DOM or SAX API

Although the DB2 pureXML feature lets you avoid a lot of XML parsing in the application layer, access to XML documents through the DOM API or the SAX API can still be useful, depending on the design and requirements of your application. JDBC 4.0 introduces a new data type called SQL/XML, together with a variety of methods which facilitate DOM and SAX access to XML documents retrieved from DB2. This is supported in DB2 Version 9.5 (and later) and illustrated in figure 89. To use these capabilities, you need an SDK for Java Version 6 or later.

```
// get the result XML as a binary stream
SQLXML sqlxml = resultSet.getSQLXML(tradedoc);
InputStream binaryStream = sqlxml.getBinaryStream();

// get the result XML as a DOMSource
SQLXML sqlxml = resultSet.getSQLXML(tradedoc);
DOMSource domSource = sqlxml.getSource(DOMSource.class);
Document document = (Document) domSource.getNode();
Node myNode = …

// create a SQLXML object with the input XML document in it
DOMResult domResult = sqlxml.setResult(DOMResult.class);
domResult.setNode(myNode);

// set that xml document as the input to parameter marker 1
Mystmt.setSQLXML(1, sqlxml);
```

Figure 89: Sample code with the SQLXML types in JDBC 4.0

For further information on JDBC 4.0 in DB2 Version 9.5, see the following topics in the DB2 information center:

- JDBC 4.0 support has been added
- Driver support for JDBC APIs

Best Practices

Storage options for XML data to improve performance and storage efficiency

- Use DB2 DMS table spaces with automatic storage.
- Use a large page size for XML data, such as 16 KB or 32 KB.
- Choose a different table space page size for XML data if a performance analysis indicates the need.
- Use base table inlined storage for XML documents if many of them are small enough to be stored on the table's data pages together with non-XML data. Otherwise documents are stored separately from the table, similar to LOBs, and are accessed through the regions index.
- Use compression to reduce the size of XML documents

Techniques for adding XML data into a DB2 database:

- To improve performance when you use insert, import or load to add data:
    - Use DMS table spaces with a large page size, such as 16 KB or 32 KB.
    - Provide sufficient buffer pool space to support index reads for XML regions and path indexes.
    - If you have multiple user-defined XML indexes, in general it is better to define them before adding the XML data.
- If needed, extract selected XML element values into relational columns in the same row as the XML document. Having the data in relational columns offers the following benefits:

- Provides easy SQL-only access to important or frequently accessed data items
- Provides the ability to define primary key, foreign key, or other constraints
- Provides the ability to define multi-column (composite key) relational indexes
- Enables better-performing relational joins
- Provides a source of potential distribution keys in DPF systems, partitioning keys for range-partitioned tables, and clustering keys for MDC tables

- Split large XML documents into smaller pieces, if the split results in a better fit for the predominant granularity of data access.

- Define triggers to automate XML data validation upon insert or update.

Techniques for querying and updating XML documents efficiently:

- Use the SQL/XML functions XMLTABLE or XMLQUERY to extract data values from XML documents. Use SQL constructs to perform grouping and aggregation on the extracted data if needed.

- Use the XMLEXISTS predicate in the SQL WHERE clause to specify predicates over XML data to help improve query performance by examining fewer rows.

- Use a fully specified XML path rather than the wildcard characters * or // to navigate to the desired XML elements. This can help provide better performance since DB2 can navigate directly to the desired XML elements, skipping over non-relevant parts of the XML document.

- Join XML and relational data in your queries as needed to increase business insight and maximize the value of a single hybrid database server.

- Implement XML joins as relational joins of extracted (from XML) columns in DPF systems to improve performance.

- When updating multiple elements of an XML document, combine the updates into a single transform expression to help obtain significantly increased update performance.
- Declare namespaces in queries, updates and XML indexes to match your XML business data. This enables you to handle XML documents from multiple or complex domains.

Techniques for using indexes over XML data with queries effectively:

- Ensure an index over XML data define the appropriate elements or attributes. An index can only be used by a query if the query predicate qualifies a subset, or all, of the nodes that are included in the index.
- Define indexes using fully specified XML paths rather than the wildcard characters * or // to improve performance and minimize index size.
- Define index data types appropriately as it affects the potential use of the index in query predicates. For example, if an index data type is DOUBLE, it cannot be used in a query with a string predicate.
- Use a VARCHAR(n) index where the indexed character strings have a maximum length of n, and the indexed elements are used in range predicates (<, >).
- Use a VARCHAR HASHED index with elements or attributes which will only be queried with equality predicates.
- Avoid defining indexes on non-leaf nodes. Index non-leaf nodes only if it will clearly benefit expected query predicates and you know the document structure well.
- Avoid using /text() in queries. A predicate with /text() can use an index only if the index is also defined with /text().
- To allow index use with XML join queries, cast join predicates to

the type of the XML index.

Techniques for effectively maintaining and monitoring an XML database:

- Use the **runstats** command with the EXCLUDING XML COLUMNS clause to refresh only the relational statistics, if a table contains relational and XML data.
- Use sampling with the **runstats** command to reduce the time for executing the **runstats** command.
- Use the DB2 snapshot monitor to investigate the benefit of different page sizes or other aspects of XML performance.

Techniques for developing efficient pureXML applications:

- Compile (“prepare”) very short database queries and pass predicate literal values for each execution of the query to improve total response time.
- Avoid code page conversion of XML data between DB2 and the application to improve performance.
- Avoid parsing XML documents in the application layer to improve performance. To access XML documents, use APIs such as the DOM or SAX API, and the Java data type SQL/XML.

Summary

The best practices presented in this paper are the result of lessons that have already been learned through the use of the DB2 pureXML feature in real-world applications where performance is a critical factor. These practices will help you avoid common mistakes and to fine-tune your database to obtain the desired goals for both your business and IT environment when using pureXML.

The principles and guidelines in this paper are organized roughly according to the life cycle of a database project, starting with a look at the data and ending with some recommendations for maintaining and monitoring the database for best performance. This paper also describes some best practices for application developers working with DB2 pureXML. This paper describes the best practices in the following areas:

- Discussion of the storage options for XML data:
  - Using DB2 DMS table spaces with automatic storage
  - Selecting the correct page size for the XML data.
  - When and how to configure different page sizes for XML and relational data
- Techniques for adding XML data into a DB2 database:
  - The differences between the methods of adding XML data to a database: insert, import and load
  - How add a large number of small XML documents efficiently
  - How to use compression and base-table inlined storage for XML documents
  - Methods of validating XML data
- Techniques for querying XML documents
  - When and how to use SQL and XQuery to search for XML and relational data
  - Using paths to search for data in the XML document hierarchy
  - Guidelines for using indexes over XML data
  - Guidelines for handling XML namespaces
  - Performing updates to data in XML documents

- Maintaining and monitoring the database
    - Collecting statistics on XML data and XML indexes to improve query performance
    - Monitoring database XML workload
- Developing DB2 applications
    - How to improve performance of short queries
    - The affect of code page conversions when working with XML data and DB2
    - Using the Java DOM or SAX APIs.

As XML becomes increasingly critical to the operations of an enterprise, it becomes an asset that needs to be shared, persisted, searched, secured, and maintained. Depending on its use, XML data might also need to be updated, audited, and integrated with other data. Storing XML data in its native hierarchical format in a DB2 database has advantages, including:

- DB2 is aware of the internal structure of the XML data.
- DB2 can maintain the hierarchical and flexible nature of XML data.
- Integrated access to XML documents together with relational in a single database

Further Reading

- DB2 Best Practices http://www.ibm.com/developerworks/db2/bestpractices/

[1] "DB2 pureXML Cookbook", IBM Press, 2009
http://tinyurl.com/pureXML

[2] "Enhance business insight and scalability of XML data with new DB2 9.7 pureXML features",
http://www.ibm.com/developerworks/data/library/techarticle/dm-0904db297purexml/index.html

[3] "Customizing XML storage in DB2",
http://www.ibm.com/developerworks/data/library/dmmag/DMMag_2009_Issue3/Tips/index.html

[4] “pureXML™ in DB2 9: Which way to query your XML Data?”
http://www.ibm.com/developerworks/db2/library/techarticle/dm-0606nicola/

[5] “15 best practices for pureXML performance in DB2 9”
http://www.ibm.com/developerworks/db2/library/techarticle/dm-0610nicola/

[6] "Indexing XML Documents in DB2"
http://www.ibm.com/developerworks/wikis/download/attachments/1824/indexingXMLdocuments.pdf

[7] “Exploit XML indexes for XML query performance in DB2”
http://www.ibm.com/developerworks/db2/library/techarticle/dm-0611nicola/

[8] “XMLTABLE by Example – Part 1”
http://www.ibm.com/developerworks/db2/library/techarticle/dm-0708nicola/

[9] “XMLTABLE by Example – Part 2”
http://www.ibm.com/developerworks/db2/library/techarticle/dm-0709nicola/

[10] “Update XML in DB2 9.5”
http://www.ibm.com/developerworks/db2/library/techarticle/dm-0710nicola/

[11] “Improving the derivatives trading process with DB2 pureXML”
http://www.ibm.com/developerworks/wikis/download/attachments/1824/Derivatives+Trading+DB2+XML.pdf

[12] “Automating the confirmation of derivative trades”
http://www.ibm.com/developerworks/wikis/download/attachments/1824/Derivatives+Confirm+Engine+Final.pdf

[13] “Evolving your XML schemas using DB2 pureXML”
http://www.ibm.com/developerworks/db2/library/techarticle/dm-0803faraaz/

[14] "Query XML data that contains namespaces"
http://www.ibm.com/developerworks/db2/library/techarticle/dm-0611saracco/

[15] Technical Papers and Articles on DB2 pureXML
http://www.ibm.com/developerworks/wikis/display/db2xml/Technical+Papers+and+Articles

[16] DB2 pureXML Success Stories
http://www.ibm.com/developerworks/wikis/display/db2xml/DB2+pureXML+Case+Studies

Contributors

We would like to thank the following people for their reviews and help and with this paper:

Bob Harbus
DB2 Technical Evangelist

Tim Vincent
Chief Architect DB2 LUW

Notices

Trademarks