HAAGA-HELIA
University of Applied Sciences

An Evaluation of DB2 Express-C pureXML Feature Pack using the TPoX Performance Benchmark

Li Wang

Bachelor's thesis
Degree Programme in Business Information Technology
2011

Abstract

08.04.2011

Degree programme

Authors Li Wang	**Group** 2007
The title of your thesis An Evaluation of DB2 Express-C pureXML Feature Pack using the TPoX Performance Benchmark	**Number of pages and appendices** 59 + 6
Supervisors Martti Laiho	

This thesis provides an introduction to XML indexes of DB2 pureXML Feature Pack, evaluating the effect of XML indexes to performance of application queries.

The thesis consists of two parts. The first part, chapters 1-4, present an overview of the "Transaction Processing over XML" (TPoX) benchmark and introduction to the implementation of XML indexes in DB2 9 LUW, based on literature research in handbooks, case studies and manuals. The smallest configured TPoX benchmark environment (XS scaling with 3 620 833 XML documents in 10GB disc size) is installed on Windows XP platform in a Microsoft Virtual PC 2007 computer. The second part, the empirical part in chapter 5, presents series of tests using selected XQuery queries of TPoX in the generated XS environment, with and without the XML indexes on XML columns. The use of indexes is studied from the extended EXPLAIN tables of hybrid DB2 giving performance estimates in TIMERON units of DB2. The performance effect is measured in the corresponding TPoX test runs of 25...100 concurrent virtual users.

According to the empirical tests, the XML indexes have a remarkable impact on the query performance and it proved to be effective to TPoX benchmark performance.

The study concludes that, the indexes on XML columns have a significant effect on the query performance in TPoX benchmark. Moreover, how to build up the indexes on XML column is also a highly considerable issue.

.

Key words
XML, database, benchmark, XQuery, SQL/XML, TPoX

Table of contents

Glossary of terms 1
1 Introduction 2
2 TPoX overview 3
2.1 TPoX data and XML schema 4
2.2 TPoX transactions and workload 5
2.2.1 Insert, update, delete 6
2.2.2 Queries 8
2.3 TPoX workload driver and documentation 9
3 Testing using TpoX 10
3.1 Installing TPoX 10
3.2 Preparing Testing 11
3.2.1 Download testing data 11
3.2.2 Performance test 12
4 Building of XML Index 15
4.1 XML index type 15
4.1.1 XML regions index 16
4.1.2 XML column path index 16
4.1.3 Index on an XML column 17
4.2 Creating index on an XML column 18
5 Experiment part 18
5.1 Test background and plan 18
5.2 Test on Q1 with and without the created index ORDER_ID 21
5.2.1 Test on Q1 with created indexes ORDER_ID 21
5.2.2 Test on Q1 without created index ORDER_ID 25
5.2.3 Test result comparison 28
5.3 Test on Q7 with and without created indexes on the joined table 29
5.3.1 Test on Q7 with created indexes 29

5.3.2 Test on Q7 by dropping the index 32
5.3.3 Test result comparison 36
5.4 Test on Q4 by reducing created indexes on table SECURITY 37
5.4.1 Test on Q4 with created indexes 37
5.4.2 Test on Q4 by dropping the indexes 40
5.4.3 Test result comparison 46
5.5 Query test on TPoX with & without created indexes on XML column ... 48
5.5.1 Query test on TPoX workload with created indexes 48
5.5.2 Query test on TPoX workload by dropping created indexes 52
5.5.3 Test result comparison 55
5.6 Summary of the tests 56
6 Conclusion 57
Bibliography 58
Appendices 60
Appendix 1. TPoX QUERIES 60

Glossary of terms

GRPBY	Group rows
IXAND	The ANDing of the results of multiple index scans
IXSCAN	Scans or probes an index on relational data
NLJOIN	Performs a merge-sort join
RETURN	Returns data from a query
RIDSCN	Scans a list of row identifiers (RIDs)
TBSCAN	Performs a table scan
TEMP	Stores data in a temporary table
TPoX	Transaction processing over XML
TIMERON	A unit of measurement used to give a rough relative estimate of the resources, or cost
XANDOR	Evaluates multiple predicates simultaneously with two or more XISCAN operators
XISCAN	Scans or probes an index on XML data
XSCAN	Navigates XML data to evaluate XPath expressions
XVIP	Physical index
XVIL	Logical index

1 Introduction

According to the work of Nicola's research group, XML database technology efficiently supports the use of XML. There is an increasing demand for XML database technology in commercial enterprises including finance and banking, industry, school, government, health care, and recording to such increasing demand, how to make XML database more functional becomes an important discussed topic. The performance is always a most considerable issue. (Nicola, Kogan, Schiefer 2007, 1)

Comparing variety of XML database, XML database on the needs of users are not the same. The performance has been chosen as the most important conditions by most users. So a variety of performance testing tools have emerged. Some tests only for the implementation of certain aspects of the database. Some are predominantly application-oriented, such as XMach-1(Böhme, Rahm 2001) and XBench (Yao, Özsu, Keenleyside 2002). Some are designed as abstract micro-benchmarks eg: XPathMark (Franceschet 2005), XMark (Schmidt, Waas, Kersten, Carey, Manolescu, Busse 2002). Further on, base on the situation, Nicola's research group developed an application-oriented and domain-specific benchmark called "Transaction Processing over XML" (TPoX). (Nicola, Kogan, Schiefer 2007a, 1) The goal of TPoX is to allow database designers, developers and users to evaluate the performance of XML database features, such as the XML query languages XQuery and SQL/XML, XML storage, XML indexing, XML Schema support, XML updates, transaction processing and logging, and concurrency control.(Nicola, Kogan, Schiefer 2007a, 1) Based on their analysis of real XML applications, they designed and implemented TPoX which simulates a financial multi-user workload with XML data conforming to the FIXML standard. The TPoX benchmark was originally developed and tested by IBM and Intel but became an open source at SourceForge in January 2007. (http://www.answers.com/topic/transaction-processing-over-xml)

Nicola research group indicate a comparing with other benchmark recording the XML benchmark requirements in their work. After analyse, they demonstrate the result of comparing. They try to find the solution to meet more XML benchmark requirements. The following is a short conclusion about their result and their goal to reach.

They believe that two separate XML benchmarks are required, one is data-centric scenario and other is document-centric. TPoX models a data-centric scenario. Many data centric XML applications deal with million to billions of relatively small XML document, but only TPoX defined multi-document tests scale from million to billions of XML documents. Rest of them they only touch the low end required scale. XML document often required to use flexibility, i.e change formats, business forms and other type's documents. In TPoX they address data variability by using a complex real-world XML Schema (FIXML, financial information exchange markup language). FIXML defines thousands of optional elements and attributes but only a very small subset appears in any given instance document. TPoX allows multi-user tests and make the isolated assessment of database performance much easier. Recording to the read/write workload, the TPoX defines a mixed workload of 30% writes and 70% reads which reach a higher level to stress all database system components. Except XPathMark, Only TPoX uses namespaces. This meets the real world applications' demands. The schema validation is required in XML applications and it efficiently affects the performance. TPoX requires the schema validation as a mandatory operation. Other benchmarks they might allow schema validation but none of them requires a mandatory operation. TPoX allow the multiple document types and joins for XML applications. Of the other benchmarks, only XBench includes such a join and only one. (Nicola, Kogan, Schiefer 2007a, 2)

Base on the work of the Nicola research group, in my thesis, I will use the TPoX benchmark as the test tool to analyse the XML indexes. I will use the queries which Nicola research group already defined. I will run the queries in different conditions to compare the work of XML indexes on columns. The purpose of the work is to observe how the XML indexes on columns efficiently affect the queries performance.

2 TPoX overview

Nicola research group state that TPoX is an application-level XML database benchmark based on a financial application scenario. It is mainly used to assess the performance of XML database system focusing on XQuery, SQL / XML, XML storage, XML indexing, XML Schema validation, XML update, logging, concurrency, etc. From various financial application models, they selected online brokerage& trading because it is an important application. It is easily understood by both benchmark participants and database users. (http://tpox.sourceforge.net/)

TPoX consists of the following parts:

- XML Schemas for all document types used in the benchmark. including a FIXML schema;
- A set of transactions to be run on the generated data.

– A toolset for XML data generation to efficiently generate millions of XML documents with well-defined value.

– A workload driver used to perform user-defined load, and collect and print test results; it can be customized through configuration files.

– Documentation description TPoX implementation details and how to use the document.

The TPoX benchmark was carefully designed and implemented. TPoX reaches a certain technical requirements such as portability, simplicity, and scalability. (http://tpox.sourceforge.net/) This chapter shows you an overview of TPoX. It will bring some understanding on how TPoX benchmark works.

2.1 TPoX data and XML schema

TPoX data model is based on the trading scene in the financial system and it uses FIXML to model some of its data. It consists of two business entities: customers and brokerage firms. (Nicola, Kogan, Schiefer 2007b, 2) Figure1 below gives an overview of TPoX application scenario.

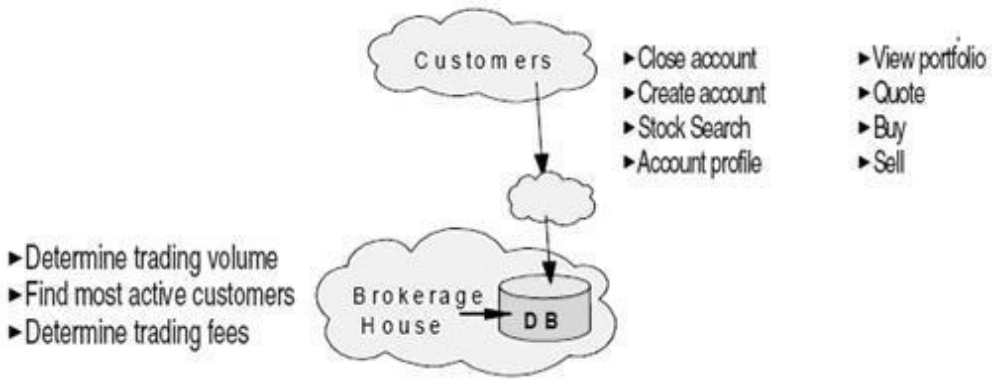


Figure 1. TPoX application scenario (An XML database benchmark)

The scenario presents a simplification of a real-world brokerage application. The customers buy and sell securities through the order. According to the customer requests, the brokers process transactions. The core of the system is a database to support XML features; its performance determines the performance of the application. (Figure1.)

The following figure shows the main logical data entities of TPoX relations and the corresponding schema. Each customer has 1 or more accounts. Each account could have 1 or more orders. Per order could buy or sell one warrants security each time for one account. Each warrant security can have 1 or more security holdings, which means that the warrant security can be purchased by multiple accounts; similarly, each account may include one or more of the Holdings. Each warrants security can exist in a number of orders or a number of holdings.

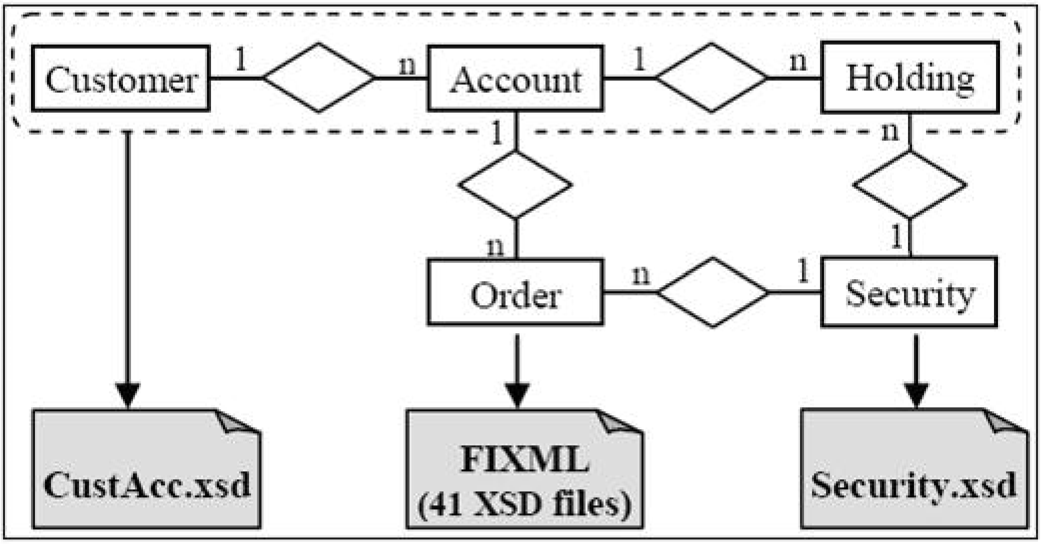


Figure 2. TPoX Entities and XML Schemas (A Transaction Processing Benchmark)

From the figure 2, we could see that TPoX's data entities are represented by three XML schemas. For each customer there is one XML document (CustAcc.xsd) that includes personal data and information about his accounts and holdings. The size of CustAcc is between 4KB and 20KB. Orders are represented using FIXML 4.4. FIXML is an industry standard XML schema for trade-related messages such as buy or sell orders. The size of the order document is between 1KB to 2 KB. Order document have many attributes and a high ratio of notes to data. Security documents represent the majority of US-traded stocks, bonds and funds using actual security symbols and names. Their size ranges between 3KB and 10KB. The three document collections are interrelated with each other. For example, Order documents contain security symbols and account numbers that exist in the security and the CustAcc documents. (Nicola, Kogan, Schiefer 2007a, 4). In TPoX, the database size can be ranged from extra small to extra-extra large depending on the number of Order and CustAcc documents. In my testing part, I will use extra small or small size database because of the limit of computer storage space.

2.2 TPoX transactions and workload

The TPoX benchmark execution has two stages: stage 1 performs concurrent inserts to populate the database and maintain all desired indexes at the same time. Stage 2 performs a multi-user read/write workload on the populated database, with 70% queries and 30% write operations including inserts, updates and deletes combined. Both stages are executed in the workload driver. (Nicola, Kogan, Schiefer 2007a, 4) The TPoX framework is very extensible and it can be use to define several different sets of transactions for the different purposes. A mixed workload consists of inserts, deletes, updates and queries. The queries are expressed in XQuery which can be embedded in SQL, e.g. through the use of SQL/XML functions. (http://tpox.sourceforge.net/WorkloadDriverUsage_v2.0.pdf) The way to process the performance testing for TPoX is to execute the transaction specified by user. TPoX provides basic transaction templates, which are stored in TPoX/WorkloadDriver/DB2/. The user can modify or add the necessary implementation transactions. Every transaction is defined in the file of its own. It could consist of one or a number of statements. Each statement is terminated by "%". The transaction templates can include parameters as shown below, including as "|1" The generation rules are provided by the load profile. The implementation of the test is generated by the parameter maker. In my testing part, I will execute the query testing on the workload and I will document the test process. Those queries will be referenced in the appendix part.

2.2.1 Insert, update, delete

In FIXML, the insert/update/delete transactions can be used in the following observations:

- Customer accounts are updated to reflect trades (execution of orders), but not necessarily immediately after every order.
- New orders arrive continuously, old order get pruned from the system eventually and at the same rate (many order inserts, many order deletes).
- Security prices are updated regularly during a business day (updates).
- The turnover of a customer is low (few CustAcc inserts and few CustAcc deletes).
- The number of securities remains fixed (no delete or insert of securities). (http://tpox.sourceforge.net/WorkloadDriverUsage_v2.0.pdf)

Table 1. TPoX **insert & update** transactions (A Transaction Processing Benchmark)

Transaction	Business Scenario	Result
Insert 1:	A customer places a new order to buy or sell a security	Insert a new Order document in the collection of order documents.

Insert 2:	A new customer signs up for online brokerage	Insert a new CustAcc document in the collection of CustAcc documents.
Delete 1:	An order is cancelled or archived	For a given order id, delete the corresponding Order document
Delete 2:	A customer closes all of his account and terminates business	For a given customer id, delete the corresponding CustAcc document
Update1:	A customer decides to close one of his/her accounts [delete subtree]	For a given account number, update the corresponding CustAcc document by removing the Account from the CustAcc document, unless it's the customer's last and only account.
Update2:	A customer opens (another) account [insert/append subtree]	For a given customer id, update the corresponding CustAcc document by appending a new "Account" subtree to the list of accounts in the CustAcc document, unless this would exceed the Maximum of number of accounts per customer (currently seven).
Update3:	The price of a security changes [simple value update]	For a given security symbol, replace the values of the following elements in the corresponding Security document: "LastTrade", "Ask", "Bid".
Update4:	Processing by the brokerage house updates an order [value update]	For a given order id, replace the value /FIXML/Order/@SolFlag with "Y" or "N" (choose randomly), and the value of "/FIXML/Order/Instrmt/@Src with a value randomly picked from this list of characters:

		"1","2",….,"9","A","B","C",….,"J".
Update5:	A previously placed buy order gets executed [value update, add/replace subtree]	For a given account number, security symbol, and quantity: if the CustAcc document already contains a holding of the given security in the given account, increase the value of the element "quantity".
Update 6:	A previously placed sell order gets executed [value update, delete/replace subtree]	For a given account number, [security symbol,] and quantity: if the given (sell-) quantity is equal or greater than the "quantity" in the corresponding "Position" in the CustAcc document, delete that "Position" subtree from the given account.

Table 1 gives an overview about the insert/update/delete transactions. Business scenario and result of each transaction can be found from the table. (Table 1)

2.2.2 Queries

In Nicola work, they defined seven core queries for a transaction processing workload. The Queries notation will be shown in the appendix. Below is an explanation for the Queries transaction. I will do the experiment test for the queries. I will build up XML indexes and run some queries performance testing on TPoX workload. I will compare the result and find out how XML index is useful for improving the benchmark performance in queries part.

Table 2. TPoX OLTP queries (A Transaction Processing Benchmark)

Q	Query Name	CustAcc	Security	Order	Characteristic
1	get_order			x	Return full order document without the FIXML root element
2	get_security		x		Return a full security document

3	customer_profile	x			Extract 7 customer elements to construct a new profile document
4	search_securities		x		Extract elements from some securities, based on 4 predicates
5	account_summary	x			Construction of an account statement
6	get_security_price		x		Extract the price of a security
7	customer_max_order	x		x	Join CustAcc & Order to find the largest order from a certain customer

Table 2 lists the seven queries of the TPoX benchmark, the database tables accessed, and the characteristics of the queries. The actual TPoX queries are listed in Appendix 1.

2.3 TPoX workload driver and documentation

Workload driver is a lightweight Java application that spawns 1 to n concurrent threads. Each thread simulates a user that connects via JDBC to the database and submits a stream of transactions without thinking times. All transactions and their weight are described in workload description file which is input to the workload driver. Load description file used to control the load-driven implementation; it tells the driver to carry the load configuration and how to achieve one of the affairs of the parameters. Some examples of the load description files are located in the TPoX / WorkloadDriver / properties. Load description file to specify the directory that contains the template or explicitly pointed out that the list of templates to be executed. (Nicola, Kogan, Schiefer 2007a, 4) The below figure 3 is an example of workload description file.

```
NumOfTransactions = 4

 t1 = myqueries/listSecurities.xqr
 w1 = 50
 p1|1 = file|input/security_types.txt

 t2 = myqueries/getCustomerProfile.xqr
 w2 = 20
 p2|1 = uniform|2000-4000
 p2|2 = uniform|5000-20000

 t3 = myqueries/listOrders.xqr
 w3 = 15

 t4 = myqueries/customized.xqr
 w4 = 15
```

Figure 3. Sample of workload description file

NumOfTransactions specify the number of transactions contained in the template. t1 is the name of the template. w1 is the weight of the transaction. If a weight is specified, then all the transactions should be assigned the right value, and the total transaction weight value should be 100, otherwise there must be some errors; the role of the weight is that, if the user specifies the test time for the 100s, then under this load, t1 will be taken 50% in the whole implementation process which is 50s.

T1 * p1 is the generation rule for the parameter in transaction t1. Parameter maker will generate the value according to the rule generated in the test execution process. p1 | 1 indicates that the first argument in the first transaction. p2 | 1 indicates first argument in the second transaction. p2 | 2 indicates second parameter in the second transaction. p1 | 1 shows that the first parameter randomly selected from a file. After"|", it shows the address of the file; p2 | 1 shows that in the transaction 2, the first parameter 1 Integer parameter random integer uniformly distributed from the distribution 2000-4000. If the second transaction in the statement that "... where $ doc / num = | 1", then "| 1" will be presented by a random number from 2000-4000.

3 Testing using TpoX

3.1 Installing TPoX

This research is done with my laptop. I download the virtual machine. In the virtual machine I have TPoX package there. The following structure is the TPoX package. The newest TPoX version package could be found and downloaded from the website http://sourceforge.net/projects/tpox/files/. After extracting the file, we can see the folder structure under the TPoX.

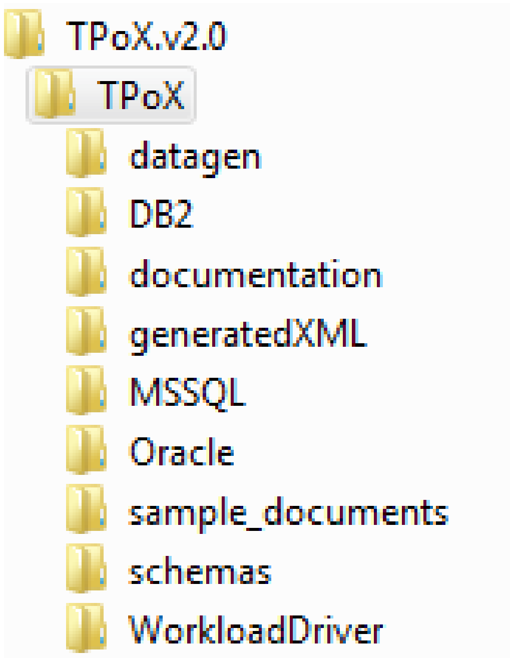


Figure 4. TPoX folder structure

Following is a short description for each folder's function.

- Datagen : test data generation tools;
- DB2, MSSQL, Oracle: used to test a specific database-related documents;
- GeneratedXML: used to store the generated XML files;
- Schemas: test used schema file;
- WorkloadDriver: is load driver folder, TPoX main program is located here.

3.2 Preparing Testing

3.2.1 Download testing data

In my testing, I installed the Window XP visual machine on my laptop and installed the DB2 and TPoX package in the visual machine. The testing data are already installed in the whole package. In case if you don't have available test data, usually you could just go to http://tpox.sourceforge.net/ to download the testing data. The data is generated by datagen by XXS standards. If you need more test data, you could run separately datagen to generate test data which match your testing criteria.

You could just unzip the file after downloading the data, then copy the data and put them under generatedXML.

TPoX/generatedXML/XXS/custacc/batch-[1-7]

TPoX/generatedXML/XXS/order/batch-[1-7]

TPoX/generatedXML/XXS/security

TPoX/generatedXML/XXS/account/batch-1

3.2.2 Performance test

Now I will demonstrate couple query performance test examples after setting up some of the load TPoX description file. First of all ensure the Classpath contains the following Class or Jar Packages:

db2jcc.jar

db2jcc_license_cisuz.jar (or any other db2_jcc_license *. jar file)

TPoX/WorkloadDriver/plugins/commons-cli-1.0.jar

TPoX /WorkloadDriver / classes

TPoX/DB2/classes

I do the performance test under the folder of TPoX/WorkloadDriver. In order to run a query test we have to set the correct java classpath on the command prompt window. The following figure will show how I set up the path.

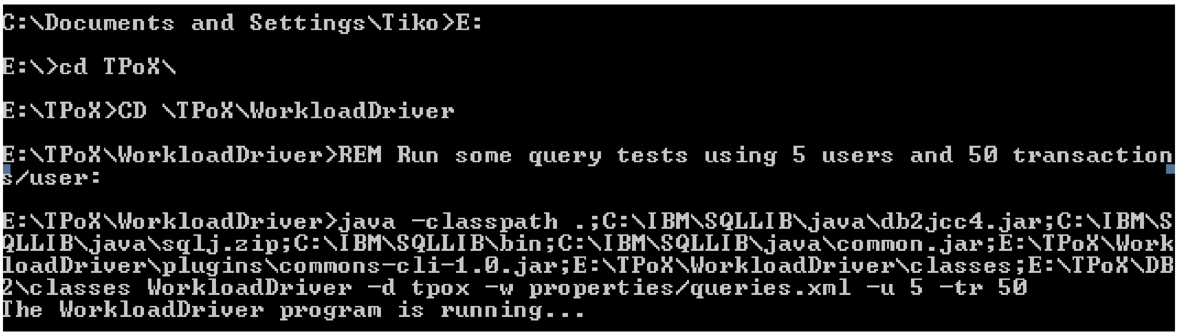
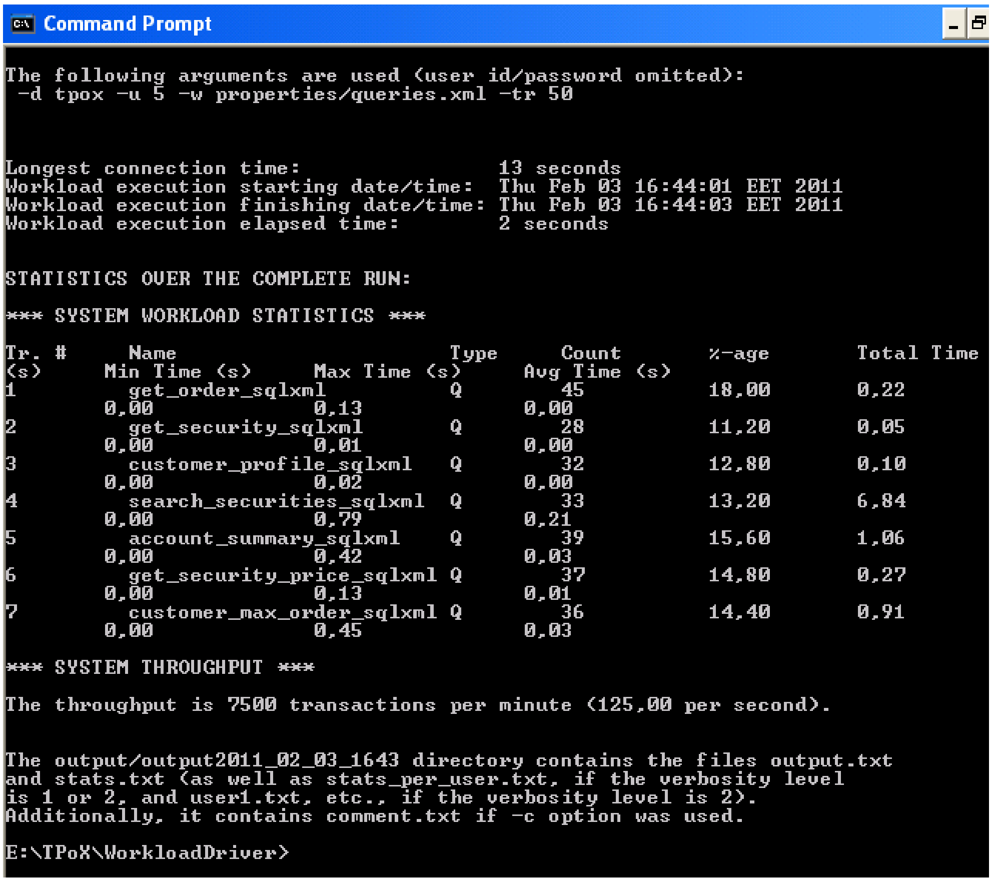


Figure 5. Path setting on command prompt window for query testing

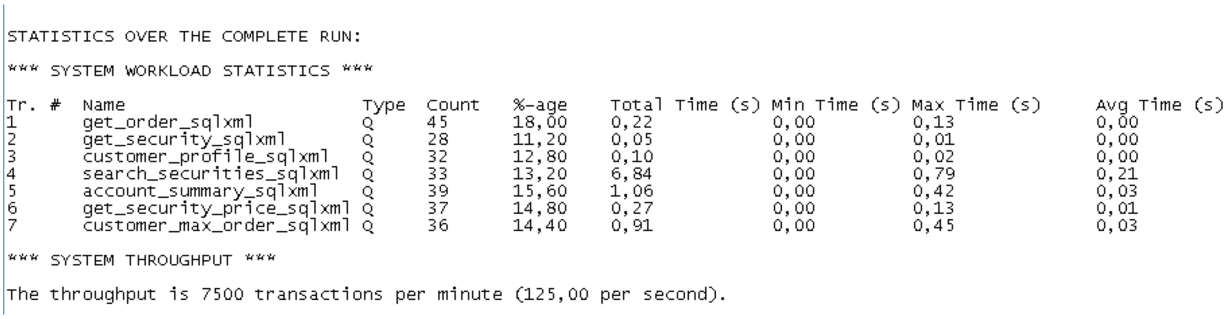
Figure 5 indicates the path setting when I do the query test on the workloadDriver. I installed the test template queries.xml in the folder properties under the workloadDriver. The queries.xml file includes seven queries. I present them as a reference in the appendix.

– I perform the first test on the command prompt window. The test is referring to 5 concurrent users, 50 transactions for each user. The following figures are the snapshot of the working status and testing result is showed in the snapshot of the statistics.



```
Command Prompt
The following arguments are used (user id/password omitted):
 -d tpox -u 5 -w properties/queries.xml -tr 50


Longest connection time:               13 seconds
Workload execution starting date/time: Thu Feb 03 16:44:01 EET 2011
Workload execution finishing date/time: Thu Feb 03 16:44:03 EET 2011
Workload execution elapsed time:        2 seconds

STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                     Type     Count        %-age       Total Time
(s)     Min Time (s)     Max Time (s)     Avg Time (s)
1         get_order_sqlxml         Q          45         18,00       0,22
        0,00             0,13             0,00
2         get_security_sqlxml      Q          28         11,20       0,05
        0,00             0,01             0,00
3         customer_profile_sqlxml  Q          32         12,80       0,10
        0,00             0,02             0,00
4         search_securities_sqlxml Q          33         13,20       6,84
        0,00             0,79             0,21
5         account_summary_sqlxml   Q          39         15,60       1,06
        0,00             0,42             0,03
6         get_security_price_sqlxml Q         37         14,80       0,27
        0,00             0,13             0,01
7         customer_max_order_sqlxml Q         36         14,40       0,91
        0,00             0,45             0,03

*** SYSTEM THROUGHPUT ***

The throughput is 7500 transactions per minute (125,00 per second).

The output/output2011_02_03_1643 directory contains the files output.txt
and stats.txt (as well as stats_per_user.txt, if the verbosity level
is 1 or 2, and user1.txt, etc., if the verbosity level is 2).
Additionally, it contains comment.txt if -c option was used.

E:\TPoX\WorkloadDriver>
```

Figure 6. Query test for 5 concurrent users and 50 transactions per users



```
STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #  Name                       Type  Count  %-age  Total Time (s) Min Time (s) Max Time (s)  Avg Time (s)
1      get_order_sqlxml           Q     45     18,00  0,22           0,00         0,13          0,00
2      get_security_sqlxml        Q     28     11,20  0,05           0,00         0,01          0,00
3      customer_profile_sqlxml    Q     32     12,80  0,10           0,00         0,02          0,00
4      search_securities_sqlxml   Q     33     13,20  6,84           0,00         0,79          0,21
5      account_summary_sqlxml     Q     39     15,60  1,06           0,00         0,42          0,03
6      get_security_price_sqlxml  Q     37     14,80  0,27           0,00         0,13          0,01
7      customer_max_order_sqlxml  Q     36     14,40  0,91           0,00         0,45          0,03

*** SYSTEM THROUGHPUT ***

The throughput is 7500 transactions per minute (125,00 per second).
```

Figure 7. Statistics of the query test result for 5 concurrent users 50 transactions per user

Figure 6 is the transaction result on the workloadDriver. Figure 7 is a statistics result which is created by TPoX under the folder WorkloadDriver/output/output2011_02_03_1643 after running the query test.

– The second test is referring to 50 users and 50 transactions for each user. The testing structure and result are showed in the following figures.

```
E:\TPoX\WorkloadDriver>java -classpath .;C:\IBM\SQLLIB\java\db2jcc4.jar;C:\IBM\S
QLLIB\java\sqlj.zip;C:\IBM\SQLLIB\bin;C:\IBM\SQLLIB\java\common.jar;E:\TPoX\Work
loadDriver\plugins\commons-cli-1.0.jar;E:\TPoX\WorkloadDriver\classes;E:\TPoX\DB
2\classes WorkloadDriver -d tpox -w properties/queries.xml -u 50 -tr 50
The WorkloadDriver program is running...

The following arguments are used (user id/password omitted):
 -d tpox -u 50 -w properties/queries.xml -tr 50


Longest connection time:               5 seconds
Workload execution starting date/time:  Thu Feb 03 17:14:29 EET 2011
Workload execution finishing date/time: Thu Feb 03 17:14:35 EET 2011
Workload execution elapsed time:        6 seconds


STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                      Type      Count         %-age        Total Time
(s)     Min Time (s)     Max Time (s)     Avg Time (s)
1         get_order_sqlxml          Q          387          15,48        17,12
        0,00             0,26             0,04
2         get_security_sqlxml       Q          333          13,32        15,16
        0,00             0,25             0,05
3         customer_profile_sqlxml   Q          358          14,32        14,26
        0,00             0,24             0,04
4         search_securities_sqlxml  Q          359          14,36        20,67
        0,00             0,26             0,06
5         account_summary_sqlxml    Q          368          14,72        15,76
        0,00             0,26             0,04
6         get_security_price_sqlxml Q          357          14,28        17,68
        0,00             0,25             0,05
7         customer_max_order_sqlxml Q          338          13,52        10,67
        0,00             0,25             0,03

*** SYSTEM THROUGHPUT ***

The throughput is 25000 transactions per minute (416,67 per second).


The output/output2011_02_03_1714 directory contains the files output.txt
and stats.txt (as well as stats_per_user.txt, if the verbosity level
is 1 or 2, and user1.txt, etc., if the verbosity level is 2).
Additionally, it contains comment.txt if -c option was used.

E:\TPoX\WorkloadDriver>_
```

Figure 8. Query test for 50 concurrent users and 50 transactions per users

```
STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #Name                       Type  Count  %-age   Total Time (s) Min Time (s) Max Time (s)     Avg Time (s)
1    get_order_sqlxml           Q     387    15,48   17,12          0,00         0,26             0,04
2    get_security_sqlxml        Q     333    13,32   15,16          0,00         0,25             0,05
3    customer_profile_sqlxml    Q     358    14,32   14,26          0,00         0,24             0,04
4    search_securities_sqlxml   Q     359    14,36   20,67          0,00         0,26             0,06
5    account_summary_sqlxml     Q     368    14,72   15,76          0,00         0,26             0,04
6    get_security_price_sqlxml  Q     357    14,28   17,68          0,00         0,25             0,05
7    customer_max_order_sqlxml  Q     338    13,52   10,67          0,00         0,25             0,03

*** SYSTEM THROUGHPUT ***

The throughput is 25000 transactions per minute (416,67 per second).
```

Figure 9. Statistics of the query test result for 50 concurrent users 50 transactions per user

Figure 8 is the transaction result on the workload. Figure 9 is a statistics result which is created under the folder WorkloaDriver/output/output2011_02_03_17:13 after running the query test. These two tests just show how I run the query test on the WorkloadDriver. I will run more tests in chapter 5.

4 Building of XML Index

In DB2 9, the pureXML provides intelligent and rich features for storing and working with XML documents. One of them is the indexing feature that can index over XML columns and return result sets from XQuery and SQL/XML. (Nicola, Kumar-Chatterjee 2010, 174) Index is the way to speed up finding and accessing data. They are used normally to improve query performance. In this chapter, I will introduce the pureXML index features and how to use the XML indexes to improve the performance and how to create indexes on XML column.

4.1 XML index type

In DB2 9 pureXML guide, there is a detailed introduction of three XML indexes: XML regions index, XML column path index, Index on an XML column. The following figure shows how XML indexes work in DB2.

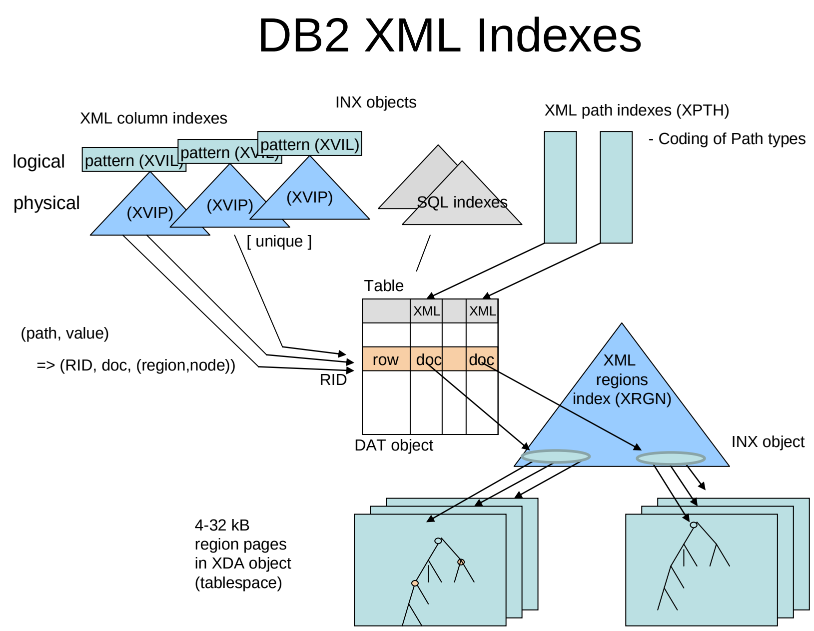


Figure 10. XML indexes structure in DB2 (From teacher's handout)

The figure 10 illustrated how the XML indexes work. The table with two XML columns is maintained in a DAT object. The XML column in this table doesn't contain the actual XML documents but only the logical pointers to them, because the XML documents can be too big to fit into a relational row on a single page. There are three types of indexes in figure 10: XML regions indexes, XML path indexes, XML column indexes. I will explain these three types' indexes separately in the following section.

4.1.1 XML regions index

XML regions index stores the locations of each XML document that is stored in XML storage in DB2.9. XML regions index is created automatically by DB2 9 when the first XML column is created or added to a table. Even the table has multiple XML columns only one XML regions index is created. (http://www.redbooks.ibm.com/redbooks/pdfs/sg247315.pdf 2006, 174-175) Every regions index is identified by the value XRGN in the column INDEXTYPE and it is recorded in SYSCAT.INDEXES. (Nicola, Kumar-Chatterjee 2010, 34) The XML regions index captures how an XML document is divided up internally into regions, which are sets of nodes within a page. (publib.boulder.ibm.com) By default, XML documents are stored in the XDA object. If a table has multiple XML columns, all of them share the same XDA object. When a document tree does not fit on a single page, DB2 automatically and transparently breaks the tree into multiple subtrees, which are called regions. Each region is then stored on a separate XDA page so a single document can span many pages. On the other hand, if the documents are much smaller than the page size, multiple regions (documents) can be stored on a single page so that no space is wasted. The key aspect of physical database design is the page size of a table space. The lower the number of regions per XML document the better the performance. The number of regions per documents depends on the page size (4KB, 8KB, 16KB, or 32KB). The large the page size of the table space the lower the number of regions per document. (Nicola, Kumar-Chatterjee 2010, 34) The accessing to XML documents stored in XML storage always goes through XML regions index. The XML regions index provides a logical mapping of those regions so that the document data can be retrieved from the XML data pages. The document ID and version ID in the XML data descriptor are used to do an index look-up in the regions index. (http://www.redbooks.ibm.com/redbooks/pdfs/sg247315.pdf, 175.) In the TPoX benchmark, there are three tables: ORDER, SECURITY, and CUSTACC. Each table has one XML regions index and those regions index can not be dropped.

4.1.2 XML column path index

The XML column path index is system-generated for each XML column created or added to the table. It is recorded in SYSCAT.INDEXES. The XML path index is shown as XPTH in SYSCAT.INDEXES.INDEXTYPE. If a table with two XML columns is created, there is one XML regions index, but two XML column path indexes generated by DB2.9. XML column path index maps paths to path IDs for each XML column. The XML column path index is used to improve index access performance during queries. (http://www.redbooks.ibm.com/redbooks/pdfs/sg247315.pdf, 175-176.) In the TPoX benchmark, the path index is created by system itself for each table.

4.1.3 Index on an XML column

XML regions index and XML column path indexes are internal indexes which are associated with XML column. These indexes are not recognized by any application programming interface that returns index metadata. (publib.boulder.ibm.com)

Comparing with these two types of indexes, Index on an XML column is distinct from them. Index on an XML column is an index created over an XML column. It is used for users to enhance performance of XQuery and SQL/XML. XML index is created as B-tree index and stored in the same place as relational indexes are stored. We can define multiple XML indexes for one XML column. At same time we must be careful for creating indexes on XML column, because it may cost to decrease the performance for INSERT, UPDATE, and DELETE, as indexes also take spaces. We should only create indexes that are really needed. (http://www.redbooks.ibm.com/redbooks/pdfs/sg247315.pdf, 176.) When we create an index on an XML column, two indexes are actually created, a logical index and a physical index. The logical index contains the XML pattern information specified in the CREATE INDEX statement. The physical index has DB2 generated key columns to support the logical index and contains the actual index value. The user works with an index on an XML column at the logical level for the CREATE INDEX and DROP INDEX statements. Processing of the underlying physical index by DB2 is transparent to the user. The logical index has the index name specified in the CREATE INDEX statement and has the index type XVIL. The physical index has a system generated name and has the index type XVIP. (http://www.redbooks.ibm.com/redbooks/pdfs/sg247315.pdf, 181-182.) In my experiment part, I will create indexes on XML column and also I will present the index result table which will show all indexes and types including logical index and physical index.

4.2 Creating index on an XML column

When creating an XML index, the certain fields are required:

Index name: specify the name of XML index.

Table and column names: specify which XML column is indexed

XMLPATTERN: specify the node we want to index.

Data type: Specify SQL data type for XML index.

The following shows the CREATE INDEX statement structure for an XML index.

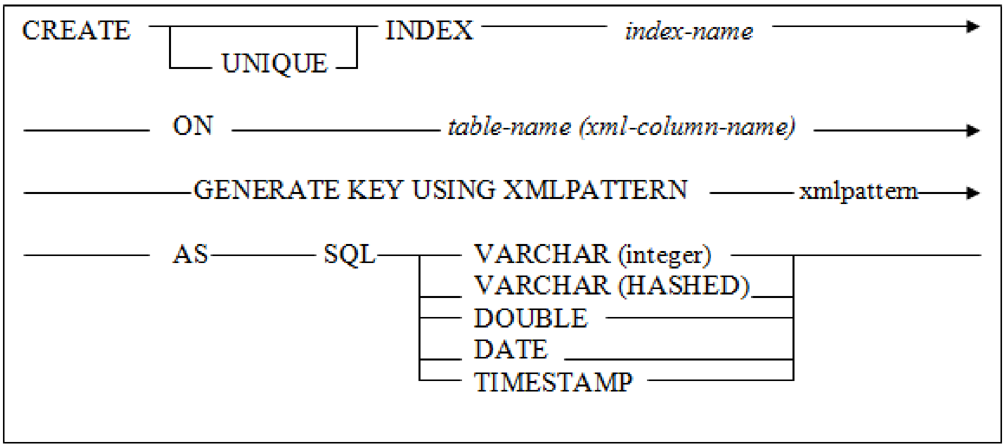


Figure 11. Structure of CREATE INDEX on XML column (DB2 9 pureXML Guide)

Figure 11 shows the most relevant part of the CREATE INDEX statement syntax for XML indexes. The UNIQUE keyword in the statement is to enforce uniqueness across and within all XML documents stored in a single XML column (Nicola, Kumar-Chatterjee 2010, 364). In my experiment part, I will use the CREATE INDEX sentence to create few indexes on XML column for tables ORDER, CUSTACC, and SECURITY. The screen script will be demonstrated.

5 Experiment part

5.1 Test background and plan

In my experiment part, I will try to find out how the XML index influences the TPoX benchmark performance. In Nicola's work, they already built some basic indexes on the tables. I will try to find out how those indexes influence the query performance. All testing will be run on a virtual machine. General information of testing background is as following figures.

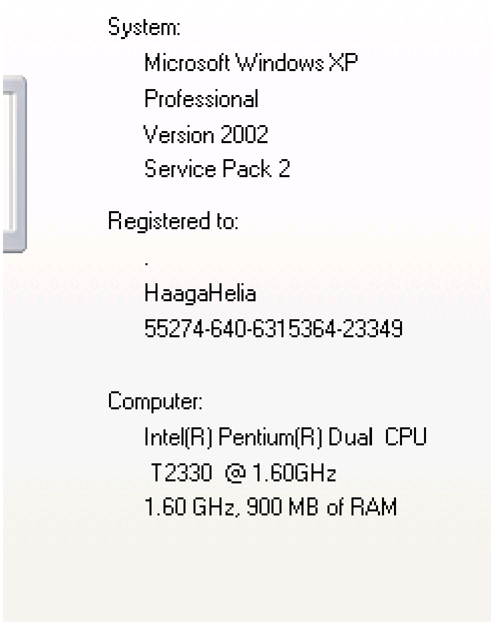
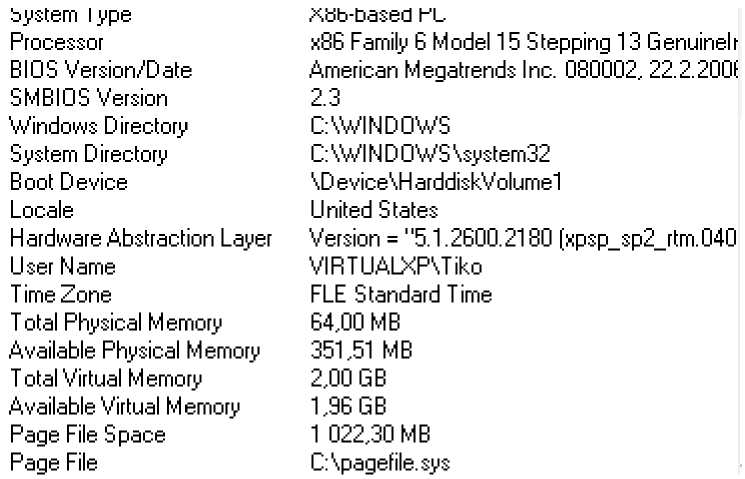


Figure 12. Virtual xp system information



System Type	X86-based PC
Processor	x86 Family 6 Model 15 Stepping 13 GenuineI
BIOS Version/Date	American Megatrends Inc. 080002, 22.2.200
SMBIOS Version	2.3
Windows Directory	C:\WINDOWS
System Directory	C:\WINDOWS\system32
Boot Device	\Device\HarddiskVolume1
Locale	United States
Hardware Abstraction Layer	Version = "5.1.2600.2180 (xpsp_sp2_rtm.040
User Name	VIRTUALXP\Tiko
Time Zone	FLE Standard Time
Total Physical Memory	64,00 MB
Available Physical Memory	351,51 MB
Total Virtual Memory	2,00 GB
Available Virtual Memory	1,96 GB
Page File Space	1 022,30 MB
Page File	C:\pagefile.sys

Figure 13. Processor information of virtual machine

Figure 14. Host window vista system information

The following table is a test plan. The table illustrates how the tests are planned and what purposes they have.

Table 3. Test Plan

	Test contents	Method	Purpose
5.2	Test the single query Q1 performance without any created indexes on XML columns and with some created indexes on XML columns on table ORDER	First test Q1 with created indexes on XML column (ORDER_ID) on table ORDER Second test Q1 without created indexes on XML column on table ORDER Third compare the test result	To observe how a single index on XML column (ORDER_ID) influences the single query(Q1) on **single table** (ORDER)
5.3	Test the single query performance Query7 with some created indexes and without created indexes on XML columns on table ORDER&CUSTACC	First test Q7 with created indexes on XML column (ORDER_ACCOUNTID, CUSTACC_ID) Second test Q7 by dropping the index ORDER_ACCOUNTID &CUSTACC_ID separately Third compare the test result	To observe how single index (ORDER_ACCOUNTID&CUSTACC_ID) on XML column affects the single query (Q7) performance executed with two **joined tables** ORDER and CUSTACC.
5.4	Test the single query Q4 performance with the created indexes on XML columns and performance after separately dropping single index SEC_SECTOR, SEC_PE, SEC_YIELD on table SECURITY	First test Q4 with created indexes on XML column(SEC_SECTOR, SEC_PE, SEC_YIELD) Second test Q4 by separately dropping single index (SEC_SECTOR, SEC_PE, SEC_YIELD) Third compare the test result	To observe how the single index on XML column influences the single query performance including **multiple created indexes** on XML column in single table SECURITY
5.5	Query performance	First test Q1-Q7 perfor-	To observe how the created

	test on TPoX workloadDriver with created indexes on XML column and by dropping certain indexes on XML column	mance on workload with created indexes on XML column Second test Q1-Q7 performance on workload after dropping certain indexes on XML column Third compare the test result	indexes on XML column influence the whole throughout and CPU utilization in the query performance.

The test plan shows how I will process the test systematically. I will demonstrate the test in the following sections following the plan.

5.2 Test on Q1 with and without the created index ORDER_ID

Before the test, I need to set up the certain environment. Performance of a query can be evaluated using DB2 explain tools which will give cost estimate of the query in special DB2 timeron units and report of the access plan of the query providing information on optimizer selected indexes for steps of the access plan. In order to use the visual explain tool, I have to create first the explain tables manually by using the script EXPLAIN.DDL. So I go to the directory sqllib\misc and write the command “db2 –tf EXPLAIN.DDL”. The explain tables are created with a schema of the current DB2 user name. This allows me to control who can use and share the tables. Now I can start my testing generating the access plans into the explain tables, and reporting the plans in textual format by the command-line tool “db2exfmt”.

5.2.1 Test on Q1 with created indexes ORDER_ID

In this test, I built up one index on XML column of table ORDER which Nicola suggested in the TPoX benchmark. I use command monitor to issue the following command:

– create unique index order_id on order(ODOC) generate key using xmlpattern 'declare default element namespace "http://www.fixprotocol.org/FIXML-4-4";/FIXML/Order/@ID' as sql varchar(15) COLLECT STATISTICS %

```
create unique index order_id on order(ODOC) generate key using xmlpattern
'declare default element namespace "http://www.fixprotocol.org/FIXML-4-4";/FIXML/Order/@ID'
 as sql varchar(15) COLLECT STATISTICS %
```

Figure 15. Create index ORDER_ID command

– Run runstats on table tiko.order to update statistics. (this command updates statistics about the physical characteristics of a table and the associated indexes) Figure 16 shows how to run a RUNSTATS command on the table ORDER.

```
RUNSTATS ON TABLE TIKO.ORDER%
------------------------------------------------------

RUNSTATS ON TABLE TIKO.ORDER
DB20000I  The RUNSTATS command completed successfully.
```

Figure 16. RUNSTATS command

After running the command, I check the index result of table ORDER by issuing command - SELECT indname, TABNAME, INDEXTYPE FROM SYSCAT.INDEXES WHERE TABNAME='ORDER'.

INDNAME	TABNAME	INDEXTYPE
SQL10021412195...	ORDER	XRGN
SQL10021412195...	ORDER	XPTH
ORDER_ID	ORDER	XVIL
SQL11031618513...	ORDER	XVIP

Figure 17. Indexes on table ORDER with created index on XML column (ORDER_ID)

From the figure 17, we can see there are 4 different types and totally 3 indexes here XRGN (XML Regions index), XPTH (XML column paths index), XVIL (Logical index on an XML column), XVIP (Physical index on an XML column). From the theory explanation part, we know XRGN and XPTH are created by system automatically. XVIL and XVIP are those indexes created by issuing the CREATE command. From chapter 4.2, we know that indexes on an XML column are only the indexes which we created over an XML column. So from the above table, we can see the index which I created is ORDER_ID.

Then I run the Q1 on the command window. I use the command "db2 connect to tpox" to connect to database. I put the query in "c:\temp\Q1.txt". Then I write the command "db2 –td% -f c:\temp\Q1.txt" to run the Q1. The query content is described in the figure 18.

```
SELECT XMLQUERY
(
'declare namespace o="http://www.fixprotocol.org/FIXML-4-4";

for $ord in $odoc/o:FIXML
return $ord/o:Order
'
PASSING odoc AS "odoc"
)
FROM order
WHERE XMLEXISTS
(
'declare namespace o="http://www.fixprotocol.org/FIXML-4-4";
$odoc/o:FIXML/o:Order[@ID=$id]
'PASSING odoc AS "odoc", cast (? as varchar(10)) as "id"
)

%
```

Figure 18. Q1 on XML file

After the command is run successfully, I use command "de2exfmt –d tpox -1" to report the execution plan for the query. The following is the access plan from the execution plan.

```
Access Plan:
-----------
        Total Cost:             22,7652
        Query Degree:           1

                          Rows
                         RETURN
                         (   1)
                          Cost
                           I/O
                           |
                        0,931034
                         NLJOIN
                         (   2)
                         22,7652
                            3
                     /----+----\
               0,931034          1
                ^NLJOIN        GRPBY
                (   3)         (   9)
                15,1936        7,57106
                   2              1
              /--+--\             |
            1       0,931034    0,972
         FETCH       XSCAN      XSCAN
         (   4)      (   8)     (  10)
         7,62392     7,56966    7,57073
            1           1          1
      /----+----\
     1          250
  RIDSCN    TABLE: TIKO
  (   5)       ORDER
0,0541239       Q6
     0
     |
     1
   SORT
  (   6)
0,0532729
     0
     |
     1
  XISCAN
  (   7)
0,0512718
     0
     |
    250
XMLIN: TIKO
   ORDER_ID
     Q6
```

Figure 19. Access plan of Q1 with created indexes (ORDER_ID) on table ORDER

In order to interpreter the nodes of the access plan in figure 19, there are five lines and three numbers in each node which we need to understand. The number above each operator name (return, nljoin, grpby, tbscan, xscan) is the estimated number of rows produced by the operator. Next two numbers are the estimated cost of the operation in timerons and the estimated number of I/Os the operator will perform. As an example, Figure 20 provides explanation for a step node in the access plan:

1	Estimated number of rows returned by the operator
XISCAN	Operator name
(7)	Unique identifier of the operator
0, 0512718	Estimated cost of the operator
0	Estimated I/O cost of the operator

Figure 20. Explanation of access plan in DB2 (DB2 pureXML cookbook, page405)

Now Let us see how the access plan above in figure 19 described the query Q1 execution. The elements of the access plan are read from the bottom up, and from left to right. In step 7, the index scan XISCAN probes the index with the path-value pair (/Order/@ID) and returns the row identifiers (RIDs) for the documents to the sort operation in step 6. The RID scan will build a list of the pages calling the prefetchers to retrieve the pages into the buffer pool and passes the row IDs to the fetch operator. The fetch operation in step 4 can then fetch and process the pages because they should already be in the buffer pool. For each row fetched, the NLJOIN passes a document pointer to the XSCAN operator, which processes the corresponding XML document. It evaluates the predicate on ORDER. Then it is passed to the nested loop join (NLJOIN) in step 3. The nested loop join (NLJOIN) then accesses the inner table. Then each element is passed up through the NLJOIN operator to the RETURN operator. The RETURN operator returns the result set to the calling application. A return result of execution plan could be seen from the following figure:

```
Plan Details:
-------------

        1) RETURN: (Return Result)
                Cumulative Total Cost:          22,7652
                Cumulative CPU Cost:            128890
                Cumulative I/O Cost:            3
                Cumulative Re-Total Cost:       15,1918
                Cumulative Re-CPU Cost:         108524
                Cumulative Re-I/O Cost:         2
                Cumulative First Row Cost:      22,7648
                Estimated Bufferpool Buffers:   4
```

Figure 21. Execution return result of Q1 with created index ORDER_ID on table ORDER

From the above figure 21, we can see the plan details of the returned performance results. It explains all details including total cost, CPU cost and I/O cost after running the Q1 under the index ORDER_ID which I created.

Now I run the query test on TPoX workloadDriver to observe the test result from workloadDriver for 100 concurrent users and 50 transactions per user. The following figure shows the result.

```
The following arguments are used (user id/password omitted):
 -d tpox -u 100 -w properties/queries.xml -tr 50


Longest connection time:                7 seconds
Workload execution starting date/time:  Thu Feb 24 19:25:51 EET 2011
Workload execution finishing date/time: Thu Feb 24 19:26:19 EET 2011
Workload execution elapsed time:        27 seconds

STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                      Type     Count        %-age       Total Time
(s)     Min Time (s)      Max Time (s)     Avg Time (s)
1         get_order_sqlxml          Q        765          15,30       152,57
        0,00              0,87             0,20
```

Figure 22. Test result of Q1 on workloadDriver with created index ORDER_ID

```
STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #Name              Type   Count  %-age    Total Time (s)   Min Time (s)   Max Time (s)   Avg Time (s)
1    get_order_sqlxml  Q      765    15,30    152,57           0,00           0,87           0,20
```

Figure 23. Statistics of the test result with created index ORDER_ID

5.2.2 Test on Q1 without created index ORDER_ID

In this test, first part is that I want to see the query performance for Q1 when there are no any created indexes on XML column on the table ORDER.

- I dropped the indexes which Nicola already built up by issuing command "Drop index ORDER_ID" as presented in figure 24.

```
DROP INDEX ORDER_ID%
--------------------------------------------------------
DROP INDEX ORDER_ID
DB20000I  The SQL command completed successfully.
```

Figure 24. Snapshot of dropping ORDER_ID

– After dropping the index, we can check the current indexes from SYSCAT.INDEXES as the figure 25; there are only region index and path index on the table ORDER.

INDNAME	TABNAME	INDEXTYPE
SQL10021412195...	ORDER	XRGN
SQL10021412195...	ORDER	XPTH

Figure 25. Indexes on table ORDER without created indexes on XML column

Then I start to run the Q1 again on the command window by issuing the following command.

– C:\IBM\SQLLIB\BIN>db2 -td% -f c:\temp\Q1.txt

– C:\IBM\SQLLIB\BIN>db2exfmt -d tpox -1

After running the command, system populates a new execution plan for Q1 without any created index on XML column as figure 26.

```
Access Plan:
-----------
        Total Cost:             1915,44
        Query Degree:           1

                     Rows
                    RETURN
                    (   1)
                     Cost
                      I/O
                      |
                   0,931034
                    NLJOIN
                    (   2)
                   1915,44
                     253
              /-----+-----\
          0,931034          1
           ^NLJOIN         GRPBY
           (   3)          (   6)
           1907,86         7,57106
             252              1
         /--+---\             |
       250     0,00372414   0,972
     TBSCAN      XSCAN      XSCAN
     (   4)      (   5)     (   7)
     15,4479    7,56966     7,57073
        2          1          1
       |
       250
 TABLE: TIKO
      ORDER
       Q6
```

Figure 26. Access plan of Q1 without created indexes on XML column on table ORDER

Starting at the bottom of the access plan, we see that the base table accessed for this query is the TIKO ORDER, and it has a cardinality of 250 rows. When no suitable indexes are defined on the ORDER table, the ORDER table is accessed by the TBSCAN operator. The TBSCAN reads all rows from the table. The NLJOIN operator connects the TBSCAN with an XSCAN.

For each row, the NLJOIN operator passes a pointer to the corresponding XML document to the XSCAN operator. This tells the XSCAN which XML documents to operate on. Then each name element is passed up through the NLJOIN operator to the RETURN operator. The RETURN operator returns the result set to the calling application. A return result of execution plan could be seen from the following figure:

```
Plan Details:
-------------


        1) RETURN: (Return Result)
                Cumulative Total Cost:          1915,44
                Cumulative CPU Cost:            4,16674e+006
                Cumulative I/O Cost:            253
                Cumulative Re-Total Cost:       1915,43
                Cumulative Re-CPU Cost:         4,16352e+006
                Cumulative Re-I/O Cost:         253
                Cumulative First Row Cost:      1915,43
                Estimated Bufferpool Buffers:   62503
```

Figure 27. Execution return result of Q1 without created indexes on table ORDER

The figure 27 shows a return result. It is a part of the execution plan. From the result, we can see how much total cost, CPU cost and I/O cost it takes to execute the Q1 without any created indexes on XML column on table ORDER.

Then I run the query test on TPoX workloadDriver to observe the result for 100 concurrent users and 50 transactions per user.

```
The following arguments are used (user id/password omitted):
 -d tpox -u 100 -w properties/queries.xml -tr 50


Longest connection time:                6 seconds
Workload execution starting date/time:  Thu Feb 24 19:40:13 EET 2011
Workload execution finishing date/time: Thu Feb 24 19:40:55 EET 2011
Workload execution elapsed time:        41 seconds

STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                        Type      Count        %-age        Total Time
(s)     Min Time (s)     Max Time (s)     Avg Time (s)
1         get_order_sqlxml             Q         765          15,30        323,77
```

Figure 28. Test result of Q1 on workloadDriver without created XML index on ORDER

```
STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #Name              Type    Count   %-age    Total Time (s)    Min Time (s)    Max Time (s)    Avg Time (s)
1    get_order_sqlxml   Q      765     15,30    323,77            0,01            1,60            0,42
```

Figure 29. Statistics of the test result without created XML index on ORDER

From the figure 29, we can see the statistics result after running the query testing on the TPoX workloadDriver. I will compare this result with the first test in the next section.

5.2.3 Test result comparison

Comparing the results of these two tests, one is with created indexes on XML column and another one is without created indexes on XML column. We can see how the created indexes on XML column work dramatically for the Q1. From the following table 4, we see after creating the index, the query works much faster comparing to the statement without created indexes. AS the index ORDER_ID significantly reduces the number of rows fetched from the table. It efficiently saves CPU cost and I/O cost during the query execution process.

Table 4. Difference of return result between two tests (figure21 &figure 27)

Return Result	**Without created index**	**With created indexes Order_id**
cumulative total cost	1915,44	22,7652
cumulative CPU cost	4,16674e+006	128890
cumulative I/O cost	253	3

The following table shows the difference of Q1 performance test result on TPoX workload driver between with created indexes and without created indexes on XML column on table ORDER.

Table 5. Difference of Q1 performance test result on TPoX with and without created indexes on XML (figure 22 & figure 28)

Get_order_sqlxml	**Total Time**	**Avg Time**	**Max Time**	**Min Time**
With index	152,57	0,20	0,87	0,01
Without index	323,77	0,42	1,60	0,00

From table 5, we see the difference with the two tests. After setting up the index (ORDER_ID) on XML column on table ORDER, the Q1 performance is faster almost 1 time than without the index on XML column according to the figure.

5.3 Test on Q7 with and without created indexes on the joined table

In this test, I am going to test Q7 of Nicola. The purpose of this test is to observe how the created indexes on XML column affect the query performance which is executed with the joined tables (ORDER & CUSTACC). The following figure shows query content which you could find also from Appendix page.

```
SELECT DECIMAL(CAST(MAX(price) AS INTEGER), 15, 2) AS maxprice
FROM
(SELECT XMLCAST(XMLQUERY(
'
declare default element namespace "http://www.fixprotocol.org/FIXML-4-4";
let $orderprice := $odoc/FIXML/Order/OrdQty/@Cash
return $orderprice
'
PASSING odoc AS "odoc") AS DOUBLE) AS price
FROM custacc, order
WHERE XMLEXISTS
(
'
declare namespace c="http://tpox-benchmark.com/custacc";
$cadoc/c:Customer[@id=$id]'
PASSING cadoc AS "cadoc", cast (? as double) as "id"
)
AND XMLEXISTS
(
'
declare default element namespace "http://www.fixprotocol.org/FIXML-4-4";
declare namespace c="http://tpox-benchmark.com/custacc";
$odoc/FIXML/Order[@Acct=$cadoc/c:Customer/c:Accounts/c:Account/@id/fn:string(.)]
'
PASSING cadoc AS "cadoc", odoc AS "odoc")
) AS T
%
```

Figure 30. Q7 on XML file

5.3.1 Test on Q7 with created indexes

First I run Q7 with indexes created by Nocola's group on XML column in tables ORDER and CUSTACC, presented in the following figure 31:

```
create index order_accountid on order(ODOC) generate key using xmlpattern
'declare default element namespace "http://www.fixprotocol.org/FIXML-4-4";/FIXML/Order/@Acct'
 as sql varchar(15) COLLECT STATISTICS %

create unique index order_id on order(ODOC) generate key using xmlpattern
'declare default element namespace "http://www.fixprotocol.org/FIXML-4-4";/FIXML/Order/@ID'
 as sql varchar(15) COLLECT STATISTICS %

create unique index custacc_id on custacc(CADOC) generate key using xmlpattern
'declare default element namespace "http://tpox-benchmark.com/custacc";/Customer/@id'
  as sql double %

create unique index custacc_accountid on custacc(CADOC) generate key using  xmlpattern
'declare namespace c="http://tpox-benchmark.com/custacc";/c:Customer/c:Accounts/c:Account/@id'
 as sql varchar(15) %
```

Figure 31. Create indexes on table ORDER & CUSTACC

The following figures show the index results of two tables after the command executed.

INDNAME	TABNAME	INDEXTYPE
SQL100214121952050	ORDER	XRGN
SQL100214121952330	ORDER	XPTH
ORDER_ACCOUNTID	ORDER	XVIL
SQL100214122112900	ORDER	XVIP
ORDER_ID	ORDER	XVIL
SQL110222183122210	ORDER	XVIP

Figure 32. Indexes result on table ORDER with the created indexes on XML column

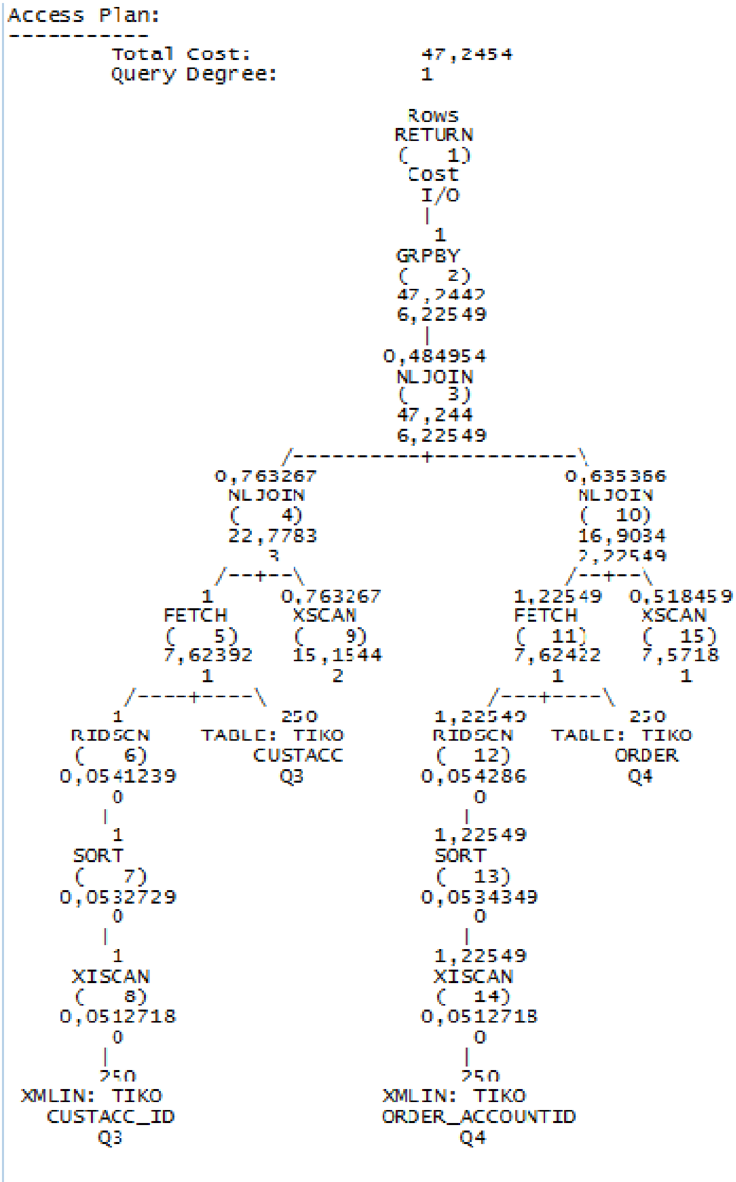
INDNAME	TABNAME	INDEXTYPE
SQL10021412195...	CUSTACC	XRGN
SQL10021412195...	CUSTACC	XPTH
CUSTACC_ID	CUSTACC	XVIL
SQL10021412211...	CUSTACC	XVIP
CUSTACC_ACCO...	CUSTACC	XVIL
SQL11030121423...	CUSTACC	XVIP

Figure 33. Indexes result on table CUSTACC with the created indexes on XML column

Now I run the query test:

- I run Q7 on command window with the command “db2 –td% -f c:\temp\Q7.txt”
- I populate the execution plan with the command “ db2exfmt –d tpox -1”

An access plan where the created indexes on XML column on both tables (ORDER & CUSTACC) are used is shown in the following figure.



```
Access Plan:
-----------
        Total Cost:             47,2454
        Query Degree:           1

                                Rows
                               RETURN
                               (   1)
                                Cost
                                 I/O
                                  |
                                  1
                                GRPBY
                                (   2)
                                47,2442
                                6,22549
                                  |
                               0,484954
                                NLJOIN
                                (   3)
                                47,244
                                6,22549
                    /-----------+-----------\
              0,763267                    0,635366
               NLJOIN                      NLJOIN
               (   4)                      (  10)
               22,7783                     16,9034
                  3                        2,22549
             /--+--\                      /--+--\
            1      0,763267          1,22549   0,518459
         FETCH      XSCAN             FETCH      XSCAN
         (   5)     (   9)            (  11)     (  15)
         7,62392    15,1544           7,62422    7,5718
            1          2                 1          1
      /----+----\                   /---+----\
     1          250             1,22549       250
  RIDSCN     TABLE: TIKO        RIDSCN     TABLE: TIKO
  (   6)       CUSTACC          (  12)        ORDER
 0,0541239       Q3            0,054286        Q4
     0                             0
     |                             |
     1                          1,22549
   SORT                          SORT
  (   7)                        (  13)
 0,0532729                     0,0534349
     0                             0
     |                             |
     1                          1,22549
  XISCAN                        XISCAN
  (   8)                        (  14)
 0,0512718                     0,0512718
     0                             0
     |                             |
    250                           250
XMLIN: TIKO                   XMLIN: TIKO
  CUSTACC_ID                  ORDER_ACCOUNTID
    Q3                            Q4
```

Figure 34. Access plan for Q7 with created indexes on table ORDER & CUSTACC

Figure 34 shows the access plan that is obtained after creating indexes on two tables: ORDER and CUSTACC. Again, we read the execution plan from the lower-left corner. The XISCAN operator probes the index with the path-value pair (/Customer/@id /) on table CUSTACC. At same time, another XISCAN operator probes the index with the path-value pair (/Order/@Acct) on table ORDER. These two XISCAN operators work together and one for each table. They find the row IDs of the documents that match their own predicates. After fetching on both tables, for each row fetched, the NLJOIN passes a document pointer to the XSCAN operator, which processes the corresponding XML document. After each table get its own result set, then they join together and another NLJOIN operator to process the corresponding document then return a final result.

```
Plan Details:
-------------

        1) RETURN: (Return Result)
                Cumulative Total Cost:          47,2454
                Cumulative CPU Cost:            273224
                Cumulative I/O Cost:            6,22549
                Cumulative Re-Total Cost:       15,2546
                Cumulative Re-CPU Cost:         203551
                Cumulative Re-I/O Cost:         2
                Cumulative First Row Cost:      47,2447
                Estimated Bufferpool Buffers:   6,22549
```

Figure 35. Return result for Q7 with created indexes on table ORDER&CUSTACC

Figure 35 shows a return result of the execution which is under two created indexes ORDER_ACCOUNTID and CUSTACC_ID.

5.3.2 Test on Q7 by dropping the index

Now I try to drop one index ORDER_ACCOUNTID from table ORDER to observe the query execution and see how the index ORDER_ACCOUNTID affects the execution.

I drop the index ORDER_ACCOUNTID but keep the indexes in the table CUSTACC. The following figure shows the drop command.

```
------------------------------ Commands Entered -----------
DROP INDEX ORDER_ACCOUNTID%
-----------------------------------------------------------
DROP INDEX ORDER_ACCOUNTID
DB20000I  The SQL command completed successfully.
```

Figure 36. Snapshot of dropping ORDER_ACCOUNTID

After dropping the index, the index result is as the following figure:

INDNAME	TABNAME	INDEXTYPE
SQL10021412195...	ORDER	XRGN
SQL10021412195...	ORDER	XPTH
ORDER_ID	ORDER	XVIL
SQL11030710114...	ORDER	XVIP

Figure 37. Indexes result after dropping ORDER_ACCOUNTID

INDNAME	TABNAME	INDEXTYPE
SQL10021412195...	CUSTACC	XRGN
SQL10021412195...	CUSTACC	XPTH
CUSTACC_ID	CUSTACC	XVIL
SQL10021412211...	CUSTACC	XVIP
CUSTACC_ACCO...	CUSTACC	XVIL
SQL11030121423...	CUSTACC	XVIP

Figure 38. Indexes result table on CUSTACC

From figure 38, we can see I didn't change indexes on table CUSTACC. Now I try to observe the query testing after dropping ORDER_ACCOUNTID in order to find out how an index influence transaction through two tables. Now I run the query again with same command on command window.

– I run Q7 on command window with the command "db2 –td% -f c:\temp\Q7.txt"
– I populate the execution plan with the command " db2exfmt –d tpox -1"

After running the command I got the new access plan for Q7 as the following figure 39.

```
Access Plan:
-----------
        Total Cost:             1939,45
        Query Degree:           1

                 Rows
                RETURN
                (   1)
                 Cost
                  I/O
                  |
                  1
                GRPBY
                (   2)
                1939,45
                  256
                  |
               0,484954
                NLJOIN
                (   3)
                1939,45
                  256
             /----+-----\
      233,28           0,00207885
      NLJOIN            TBSCAN
      (   4)            (   7)
      1908,4            22,8088
        252               3
     /-+--\               |
   250     0,93312     0,763267
 TBSCAN    XSCAN        TEMP
 (   5)    (   6)      (   8)
 15,4479   7,5718      22,7865
    2        1            3
    |                     |
   250                 0,763267
TABLE: TIKO             NLJOIN
  ORDER                 (   9)
   Q4                   22,7783
                          3
                      /--+--\
                     1      0,763267
                  FETCH      XSCAN
                  (  10)     (  14)
                  7,62392    15,1544
                     1         2
              /----+----\
             1            250
          RIDSCN      TABLE: TIKO
          (  11)         CUSTACC
         0,0541239          Q3
             0
             |
             1
           SORT
          (  12)
         0,0532729
             0
             |
             1
          XISCAN
          (  13)
         0,0512718
             0
             |
            250
       XMLIN: TIKO
        CUSTACC_ID
            Q3
```

Figure 39. Access plan for Q7 after dropping index ORDER_ACCOUNTID

From the access plan, we can see there is only one index CUSTACC_ID in the whole execution process. We can see the difference comparing with the first test. Because I dropped the index ORDER_ACCOUNTID on XML column from table ORDER, so the table scanner (TBSCAN) on table ORDER has to scan all 250 rows. It takes much more time to scan all tables. The total costs is 1906, 4 timerons to read through ORDER table. Comparing with the previous test, when using the index ORDER_ACCOUNTID, it only takes 16, 9034 total time to read the table ORDER. We could see the total cost is increased about 100 times after dropping ORDER_ACCOUNTID. On the other hand we could say after using ORDER_ACCOUNTID index, the query execution is faster than before about 100 times according to the timeron unit. Since there is an efficient saving on CPU cost and I/O cost. The following figure 40 shows a return result. Which also indicates the CPU cost and I/O cost.

```
Plan Details:
-------------


        1) RETURN: (Return Result)
                Cumulative Total Cost:          1939,45
                Cumulative CPU Cost:            6,19141e+006
                Cumulative I/O Cost:            256
                Cumulative Re-Total Cost:       1909,09
                Cumulative Re-CPU Cost:         5,99763e+006
                Cumulative Re-I/O Cost:         252
                Cumulative First Row Cost:      1939,45
                Estimated Bufferpool Buffers:   62503
```

Figure 40. Return result for Q7 after dropping index ORDER_ACCOUNTID

Now I will run Q7 by dropping CUSTACC_ID to observe how the index CUSTACC_ID affects the query performance.

```
------------------------------ Commands Entered --------
DROP INDEX CUSTACC_ID%
--------------------------------------------------------
DROP INDEX CUSTACC_ID
DB20000I  The SQL command completed successfully.
```

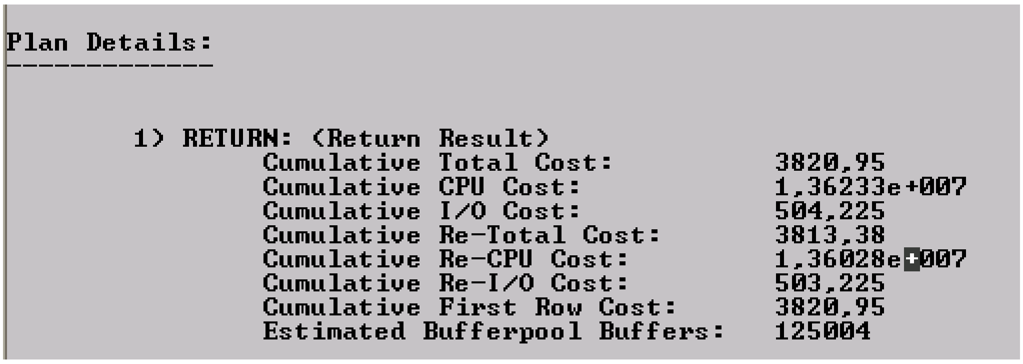
Figure 41. Snapshot of dropping CUSTACC_ID

The following figure 42 shows an access plan for Q7 where the index CUSTACC_ID on table CUSTACC was dropped.

```
Access Plan:
-----------
        Total Cost:             3820,95
        Query Degree:           1

                           Rows
                          RETURN
                          (   1)
                           Cost
                            I/O
                            |
                             1
                          GRPBY
                          (   2)
                          3820,95
                          504,225
                            |
                         0,484954
                          NLJOIN
                          (   3)
                          3820,95
                          504,225
                     /-------+-------\
              0,763267              0,635366
               NLJOIN                NLJOIN
               (   4)                (   7)
               3804,05               16,9034
                 502                 2,22549
              /--+---\              /--+--\
          250     0,00305307   1,22549  0,518459
        TBSCAN      XSCAN       FETCH    XSCAN
        (   5)      (   6)      (   8)   (  12)
        15,4479     15,1544     7,62422  7,5718
           2           2           1        1
           |                 /---+----\
          250           1,22549        250
      TABLE: TIKO        RIDSCN    TABLE: TIKO
        CUSTACC          (   9)       ORDER
          Q3            0,054286       Q4
                           0
                           |
                        1,22549
                          SORT
                         (  10)
                       0,0534349
                           0
                           |
                        1,22549
                         XISCAN
                         (  11)
                       0,0512718
                           0
                           |
                          250
                     XMLIN: TIKO
                    ORDER_ACCOUNTID
                          Q4
```

Figure 42. Access plan for Q7 after dropping index CUSTACC_ID

From the result, we can see the table scanner has to read through whole CUSTACC table after dropping CUSTACC_ID. TBSCAN takes about 15, 4479 timerons to read all 250 rows from the table CUSTACC. For each row the NLJOIN operator passes a pointer to the corresponding XML document to the XSCAN operator. For each document, the XSCAN operator traverses the document tree, evaluates the predicates, and extracts the element if the predicate are satisfied. Each element is passed up through the NLJOIN operator to the RETURN operator. Here the XSCAN takes about 15, 1544 timerons to traverse the document tree on CUSTACC table. The NLJOIN operator takes about 3804, 05 timerons to finish its work. Comparing the test with CUSTACC_ID, The NLJOIN only takes about 22, 7783 timerons to get the job done, which is how the index CUSTACC_ID works for the query performance. Following figure is a return result. I will compare the result with the previous tests in the next section.



```
Plan Details:
-------------

        1) RETURN: (Return Result)
                Cumulative Total Cost:          3820,95
                Cumulative CPU Cost:            1,36233e+007
                Cumulative I/O Cost:            504,225
                Cumulative Re-Total Cost:       3813,38
                Cumulative Re-CPU Cost:         1,36028e+007
                Cumulative Re-I/O Cost:         503,225
                Cumulative First Row Cost:      3820,95
                Estimated Bufferpool Buffers:   125004
```

Figure 43. Return result for Q7 after dropping index CUSTACC_ID

5.3.3 Test result comparison

Now let us compare the return result of these two tests to see how the index ORDER_ACCOUNTID and CUSTACC_ID had affection on the execution of Q7.

Table 6. Difference of return result between three tests (figure43, 40, and 35)

Return Result	**Without ORDER_ACCOUNTID index**	**Without CUSTACC_ID index**	**With both created indexes**
cumulative total cost	1939,45	3820,95	47,2454
cumulative CPU cost	6,19141e+006	1,36233e+007	273224
cumulative I/O cost	256	504,225	6,22549

From the table 6, we can see the big difference between three results. When the query is executed with both created indexes (ORDER_ACCOUNTID&CUSTACC_ID), the total cost is only 47, 2454, which is much less than the cost after dropping index ORDER_ACCOUNTID (1939, 45) and CUSTACC_ID (3820, 95). On the other hand, comparing the total cost after dropping ORDER_ACCOUNTID and CUSTACC_ID, we see after dropping CUSTACC_ID, the total cost is more than after dropping ORDER_ACCOUNTID, Which means the CUSTACC_ID index has a more weight on affecting the query performance. Next question is why the CUSTACC_ID had more affection on the query performance. From my study, the document size has an influence on the work. According to the Nicola's research, the CUSTACC documents are between 4KB and 20KB in size and the Orders are between 1KB to 2KB. Therefore, it will cost more timeron to execute the table CUSTACC than ORDER. So the indexes on XML column in table CUSTACC have a more weight on affecting the query performance.

5.4 Test on Q4 by reducing created indexes on table SECURITY

In this test, I am going to test Q4 on SECURITY table. The single query is executed with 3 created indexes on XML column on single table. First I test the Q4 with all created indexes which Nicola suggested. Then I test by dropping one index each time to observe the query execution plan. I will record the test step by step. I try to analyse and find out how each index influences the query performance. The following figure 44 is the query content. Also you will find it in appendix page.

```
SELECT XMLQUERY
(
'declare default element namespace "http://tpox-benchmark.com/security";
for $sec in $sdoc/Security
return
    <Security>
      {$sec/Symbol}
      {$sec/Name}
      {$sec/SecurityType}
      {$sec/SecurityInformation//Sector}
      {$sec/PE}
      {$sec/Yield}
    </Security>
'
PASSING sdoc AS "sdoc"
)
FROM security
WHERE XMLEXISTS
(
'declare default element namespace "http://tpox-benchmark.com/security";
$sdoc/Security[SecurityInformation/*/Sector=$sector and
PE[. >=$pe1 and . <$pe2] and Yield>$yield]'
PASSING sdoc AS "sdoc", cast (? as varchar(25)) as "sector",
cast (? as double) as "pe1", cast (? as double) as "pe2", cast (? as double) as "yield"
)
```

Figure 44. Q4 on XML file

5.4.1 Test on Q4 with created indexes

Here are the indexes which Nicola group already built up on table SECURITY. I run the following command to build up the indexes on SECURITY table according to Nicola's suggestion.

```
create index sec_sector on security(SDOC) generate key using xmlpattern
'declare default element namespace "http://tpox-benchmark.com/security";/Security/SecurityInformation//Sector'
 as sql varchar(25) %

create index sec_PE on security(SDOC) generate key using xmlpattern
'declare default element namespace "http://tpox-benchmark.com/security";/Security/PE'
 as sql double %

create index sec_Yield on security(SDOC) generate key using xmlpattern
'declare default element namespace "http://tpox-benchmark.com/security";/Security/Yield'
 as sql double %
```

Figure 45. Create indexes on table SECURITY

The following figure 46 shows the indexes result after I run the command on command editor.

– SELECT indname, TABNAME,INDEXTYPE FROM SYSCAT.INDEXES
  WHERE TABNAME='SECURITY'

SQL10021412195...	SECURITY	XRGN
SQL10021412195...	SECURITY	XPTH
SEC_YIELD	SECURITY	XVIL
SQL11031709374...	SECURITY	XVIP
SEC_PE	SECURITY	XVIL
SQL11031923584...	SECURITY	XVIP
SEC_SECTOR	SECURITY	XVIL
SQL11031620500...	SECURITY	XVIP

Figure 46. Indexes result table of SECURITY after creating indexes on XML column

Then I run the Q4 on command window by issuing “db2 –td% -f c:\temp\Q4. txt” and “db2exfmt –d tpox -1”. After I run these commands then I get the execution plan for Q4. So the following figures are the access plan and return result for Q4. From the access plan, we can see the query was executed under 3 created indexes: SEC_SECTOR, SEC_PE and SEC_YIELD. The return result shows the details of the execution including CPU, I/O cost.

```
Access Plan:
-----------
        Total Cost:             22,88
        Query Degree:           1

                                         Rows
                                        RETURN
                                        (   1)
                                         Cost
                                          I/O
                                           |
                                     1,56149e-006
                                        NLJOIN
                                        (   2)
                                         22,88
                                           3
                                 /-----+------\
                         1,56149e-006          1
                            ^NLJOIN          GRPBY
                            (   3)           (  12)
                            15,3075          7,5719
                               2               1
                         /---+---\             |
                  0,418333  3,73263e-006   0,93696
                    FETCH       XSCAN      FILTER
                    (   4)     (  11)      (  13)
                   7,73787     7,56965     7,57157
                      1           1           1
                /---+----\                    |
          0,418333        250               0,976
           RIDSCN   TABLE: TIKO             XSCAN
           (   5)      SECURITY             (  14)
          0,168422        Q7                7,57071
              0                                1
              |
          0,418333
            SORT
           (   6)
          0,167571
              0
              |
          0,418333
            IXAND
           (   7)
          0,165696
              0
  +-----------------+-----------------+
20,8333            25,1               50
XISCAN            XISCAN            XISCAN
(   8)            (   9)            (  10)
0,0512718         0,0512718         0,0512718
  0                 0                 0
  |                 |                 |
 250               250               250
XMLIN: TIKO      XMLIN: TIKO      XMLIN: TIKO
 SEC_SECTOR        SEC_PE          SEC_YIELD
   Q7                Q7               Q7
```

Figure 47. Access plan for Q4 after creating indexes on SECURITY

The access plan in figure 47 contains three XISCAN (XML index scans) operators, one for each XML predicate. The IXAND operator uses these XISCAN to alternately probe into the three indexes to efficiently find the row IDs of the documents that match the predicates. The FETCH operator then only retrieves these rows. These row IDs are sorted to remove duplicates (if any) and to optimize the subsequent I/Os to the table. For each row fetched, the NLJOIN passes a document pointer the XSCAN operator, which processes the corresponding XML document. For each document, the XSCAN operator traverses the document tree, evaluates the predicates, and extracts the element if the predicates are satisfied. The each element is passed up through the NLJOIN operator to the RETURN operator. The RETURN operator returns the result set to the calling application. The following figure 48 shows a return result including CPU cost and I/O cost.

```
1) RETURN: (Return Result)
        Cumulative Total Cost:          22,88
        Cumulative CPU Cost:            302455
        Cumulative I/O Cost:            3
        Cumulative Re-Total Cost:       15,3066
        Cumulative Re-CPU Cost:         282187
        Cumulative Re-I/O Cost:         2
        Cumulative First Row Cost:      22,8793
        Estimated Bufferpool Buffers:   4
```

Figure 48. Return result for Q4 with created indexes on table SECURITY

The following figure shows the query testing on the TPoX workload. We can see the Q4 performance.

```
The following arguments are used (user id/password omitted):
 -d tpox -u 100 -w properties/queries.xml -tr 50



Longest connection time:                6 seconds
Workload execution starting date/time:  Mon Feb 28 15:26:03 EET 2011
Workload execution finishing date/time: Mon Feb 28 15:26:18 EET 2011
Workload execution elapsed time:        14 seconds


STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                        Type      Count         %-age       Total Time
(s)     Min Time (s)     Max Time (s)     Avg Time (s)
1         get_order_sqlxml            Q         765           15,30       87,18
        0,00             0,46             0,11
2         get_security_sqlxml         Q         696           13,92       83,45
        0,00             0,43             0,12
3         customer_profile_sqlxml     Q         719           14,38       84,13
        0,00             0,47             0,12
4         search_securities_sqlxml    Q         722           14,44       115,73
        0,00             0,74             0,16
5         account_summary_sqlxml      Q         716           14,32       85,38
        0,00             0,41             0,12
6         get_security_price_sqlxml   Q         702           14,04       88,20
        0,00             0,43             0,13
7         customer_max_order_sqlxml   Q         680           13,60       62,01
        0,00             0,35             0,09

*** SYSTEM THROUGHPUT ***

The throughput is 21428 transactions per minute (357,14 per second).
```

Figure 49. Query test on TPoX for 100 concurrent users and 50 transactions per user

5.4.2 Test on Q4 by dropping the indexes

Now I reduce the indexes to run the test again.

First part, I drop the index SEC_SECTOR.

```
DROP INDEX sec_sector%
------------------------------------------------
DROP INDEX sec_sector
DB20000I  The SQL command completed successfully.
```

Figure 50. Snapshot of dropping SEC_SECTOR

Now we can see the index SEC_SECTOR was dropped from the result table from the following figure 51.

INDNAME	TABNAME	INDEXTYPE
SQL10021412195...	SECURITY	XRGN
SQL10021412195...	SECURITY	XPTH
SEC_PE	SECURITY	XVIL
SQL11041209222...	SECURITY	XVIP
SEC_YIELD	SECURITY	XVIL
SQL11041209243...	SECURITY	XVIP

Figure 51. Index result table of SECURITY after dropping index SEC_SECTOR

I run the test again on the command window to get execution plan for Q4.

```
Access Plan:
-----------
        Total Cost:             53,2615
        Query Degree:           1

                              Rows
                             RETURN
                             (   1)
                              Cost
                               I/O
                               |
                          1,56149e-006
                             NLJOIN
                             (   2)
                             53,2615
                              7,02
                        /-----+------\
                1,56149e-006            1
                   ^NLJOIN            GRPBY
                   (   3)            (  11)
                   45,689            7,5719
                    6,02                1
               /---+---\                |
            5,02      3,11053e-007  0,93696
           FETCH         XSCAN       FILTER
           (   4)       (  10)       (  12)
           7,68933      7,56965      7,57157
              1            1            1
        /---+----\                      |
      5,02        250                 0,976
     RIDSCN    TABLE: TIKO            XSCAN
     (   5)       SECURITY           (  13)
    0,117114        Q7               7,57071
        0                               1
        |
      5,02
      SORT
     (   6)
    0,116263
        0
        |
      5,02
      IXAND
     (   7)
    0,112194
        0
  /-----+------\
 25,1            50
XISCAN         XISCAN
(   8)         (   9)
0,0512718      0,0512718
   0              0
   |              |
  250            250
XMLIN: TIKO    XMLIN: TIKO
  SEC_PE        SEC_YIELD
   Q7             Q7
```

Figure 52. Access plan for Q4 after dropping indexes SEC_SECTOR on SECURITY

Figure 52 shows the access plan is obtained after dropping index SEC_SECTOR. The only difference is access plan contains two XISCAN operators. The IXAND operator uses these two XISCANs to alternately probe into the two indexes to efficiently find the row IDs of the documents that match both predicates. The rest of the query execution works as in the previous plan in figure 47. Following figure 53 is a return result after the RETURN operator returns the result set to the calling application.

```
Plan Details:
-------------


        1) RETURN: (Return Result)
                Cumulative Total Cost:          53,2615
                Cumulative CPU Cost:            287727
                Cumulative I/O Cost:            7,02
                Cumulative Re-Total Cost:       45,6866
                Cumulative Re-CPU Cost:         265292
                Cumulative Re-I/O Cost:         6,02
                Cumulative First Row Cost:      53,2607
                Estimated Bufferpool Buffers:   28,2004
```

Figure 53. Return result for Q4 after dropping indexes SEC_SECTOR on table SECURITY

Second part, now I only drop index SEC_YIELD.

```
DROP INDEX SEC_YIELD%
------------------------------------------------------
DROP INDEX SEC_YIELD
DB20000I  The SQL command completed successfully.
```

Figure 54. Snapshot of dropping SEC_YIELD

Now we can see the index SEC_YIELD was dropped from the result table from the following figure 55.

INDNAME	TABNAME	INDEXTYPE
SQL10021412195...	SECURITY	XRGN
SQL10021412195...	SECURITY	XPTH
SEC_SECTOR	SECURITY	XVIL
SQL11041209184...	SECURITY	XVIP
SEC_PE	SECURITY	XVIL
SQL11041209222...	SECURITY	XVIP

Figure 55. Index result table of SECURITY after dropping index SEC_YIELD

I run the test again on the command window to get execution plan for Q4.

```
Access Plan:
-----------
        Total Cost:             31,0884
        Query Degree:           1

                              Rows
                             RETURN
                             (   1)
                              Cost
                               I/O
                               |
                           1,56149e-006
                             NLJOIN
                             (   2)
                             31,0884
                             4,09167
                         /-----+------\
                 1,56149e-006             1
                    ^NLJOIN             GRPBY
                    (   3)              (  11)
                    23,5159             7,5719
                    3,09167                1
                  /---+---\                |
            2,09167  7,46527e-007      0,93696
             FETCH      XSCAN          FILTER
             (   4)     (  10)         (  12)
             7,68275    7,56965        7,57157
                1          1              1
           /---+----\                     |
     2,09167          250                0,976
     RIDSCN     TABLE: TIKO              XSCAN
     (   5)        SECURITY              (  13)
    0,112297          Q7                 7,57071
        0                                   1
        |
     2,09167
      SORT
     (   6)
    0,111446
        0
        |
     2,09167
      IXAND
     (   7)
    0,108903
        0
   /-----+------\
20,8333            25,1
XISCAN            XISCAN
(   8)            (   9)
0,0512718        0,0512718
    0                0
    |                |
   250              250
XMLIN: TIKO     XMLIN: TIKO
  SEC_SECTOR       SEC_PE
     Q7              Q7
```

Figure 56. Access plan for Q4 after dropping indexes SEC_YIELD on SECURITY

Figure 56 shows the access plan is obtained after dropping index SEC_YIELD. The access plan contains two XISCAN operators. The IXAND operator uses these two XISCANs to alternately probe into the two indexes to efficiently find the row IDs of the documents that match both predicates. The rest of the query execution works as in the previous plan in figure 47. Following figure 57 is a return result after the RETURN operator returns the result set to the calling application.

```
Plan Details:
-------------


        1) RETURN: (Return Result)
                Cumulative Total Cost:          31,0884
                Cumulative CPU Cost:            235038
                Cumulative I/O Cost:            4,09167
                Cumulative Re-Total Cost:       23,5146
                Cumulative Re-CPU Cost:         214179
                Cumulative Re-I/O Cost:         3,09167
                Cumulative First Row Cost:      31,0877
                Estimated Bufferpool Buffers:   7,37507
```

Figure 57. Return result for Q4 after dropping indexes SEC_YIELD on table SECURITY

Third part, now I drop the SEC_PE

```
DROP INDEX SEC_PE%
---------------------------------------------------
DROP INDEX SEC_PE
DB20000I  The SQL command completed successfully.
```

Figure 58. Snapshot of dropping SEC_PE

INDNAME	TABNAME	INDEXTYPE
SQL10021412195...	SECURITY	XRGN
SQL10021412195...	SECURITY	XPTH
SEC_SECTOR	SECURITY	XVIL
SQL11041209184...	SECURITY	XVIP
SEC_YIELD	SECURITY	XVIL
SQL11041209184...	SECURITY	XVIP

Figure 59. Index result table of SECURITY after dropping index SEC_PE

I run same command to get the access plan for Q4 as the figure 60:

```
Access Plan:
-----------
        Total Cost:             46,8005
        Query Degree:           1

                                 Rows
                                RETURN
                                (   1)
                                 Cost
                                  I/O
                                  |
                              1,56149e-006
                                NLJOIN
                                (   2)
                                46,8005
                                6,16667
                            /-----+------\
                   1,56149e-006            1
                      ^NLJOIN            GRPBY
                      (   3)             (  11)
                      39,228             7,5719
                      5,16667              1
                   /---+---\               |
             4,16667  3,74756e-007      0,93696
             FETCH        XSCAN         FILTER
             (   4)       (  10)        (  12)
             7,68777      7,56965       7,57157
               1            1             1
         /---+----\                       |
    4,16667       250                   0,976
    RIDSCN     TABLE: TIKO              XSCAN
    (   5)      SECURITY                (  13)
   0,116067        Q7                   7,57071
      0                                   1
      |
    4,16667
    SORT
    (   6)
   0,115216
      0
      |
    4,16667
    IXAND
    (   7)
   0,111637
      0
   /-----+------\
20,8333          50
XISCAN         XISCAN
(   8)         (   9)
0,0512718      0,0512718
   0              0
   |              |
  250            250
XMLIN: TIKO    XMLIN: TIKO
 SEC_SECTOR     SEC_YIELD
    Q7             Q7
```

Figure 60. Access plan for Q4 after dropping indexes SEC_PE on SECURITY

Figure 60 shows the access plan is obtained after dropping index SEC_PE. The access plan contains two XISCAN operators. The rest of the query execution works as in the previous plan in figure 47. Following figure 61 is a return result after the RETURN operator returns the result set to the calling application.

```
Plan Details:
-------------

        1) RETURN: (Return Result)
                Cumulative Total Cost:          46,8005
                Cumulative CPU Cost:            272913
                Cumulative I/O Cost:            6,16667
                Cumulative Re-Total Cost:       39,226
                Cumulative Re-CPU Cost:         251006
                Cumulative Re-I/O Cost:         5,16667
                Cumulative First Row Cost:      46,7997
                Estimated Bufferpool Buffers:   20,3611
```

Figure 61. Return result for Q4 after dropping indexes SEC_PE on table SECURITY

5.4.3 Test result comparison

Now I will compare the test results with a table 7. Comparing the test results, we can see the difference after dropping SEC_SECTOR, SEC_PE and SEC_YIELD. From the table, we also can recognize that the weight of affecting the performance for each index is different. The SEC_SECTOR has a heaviest effect on performance. The SEC_PE has a less effect on performance. The SEC_YIELD has the least effect on performance.

Table7. Difference of return result between four tests (figure 53, 57, 61 and 48)

Return result	**After dropping SEC_SECTOR index**	**After drop-ping SEC_YIELD index**	**After drop-ping SEC_PE index**	**With three cre-ated indexes**
cumulative total cost	53,2615	31,0884	46,8005	22,88
cumulative CPU cost	287727	235038	272913	302455
cumulative I/O cost	7,02	4,09167	6,16667	3

Now I compare the result with two query tests on the TPoX workloadDriver. We can see the Q4 running statement before and after dropping the index SEC_SECTOR. The performance is better when using the index SEC_SECTOR. After dropping the index SEC_SECTOR, the Q4 performance turned to be slower.

```
The following arguments are used (user id/password omitted):
 -d tpox -u 100 -w properties/queries.xml -tr 50


Longest connection time:                6 seconds
Workload execution starting date/time:  Mon Feb 28 15:26:03 EET 2011
Workload execution finishing date/time: Mon Feb 28 15:26:18 EET 2011
Workload execution elapsed time:        14 seconds


STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                      Type      Count        %-age       Total Time
(s)      Min Time (s)     Max Time (s)     Avg Time (s)
1          get_order_sqlxml         Q          765         15,30       87,18
        0,00             0,46             0,11
2          get_security_sqlxml      Q          696         13,92       83,45
        0,00             0,43             0,12
3          customer_profile_sqlxml  Q          719         14,38       84,13
        0,00             0,47             0,12
4          search_securities_sqlxml Q          722         14,44       115,73
        0,00             0,74             0,16
5          account_summary_sqlxml   Q          716         14,32       85,38
        0,00             0,41             0,12
6          get_security_price_sqlxml Q         702         14,04       88,20
        0,00             0,43             0,13
7          customer_max_order_sqlxml Q         680         13,60       62,01
        0,00             0,35             0,09

*** SYSTEM THROUGHPUT ***

The throughput is 21428 transactions per minute (357,14 per second).
```

Figure 62. Query test on TPoX for 100 concurrent users and 50 transactions per user before dropping index

```
Longest connection time:                6 seconds
Workload execution starting date/time:  Mon Feb 28 15:34:37 EET 2011
Workload execution finishing date/time: Mon Feb 28 15:34:54 EET 2011
Workload execution elapsed time:        16 seconds


STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                      Type      Count        %-age       Total Time
(s)      Min Time (s)     Max Time (s)     Avg Time (s)
1          get_order_sqlxml         Q          765         15,30       100,23
        0,00             0,60             0,13
2          get_security_sqlxml      Q          696         13,92       114,83
        0,00             0,93             0,16
3          customer_profile_sqlxml  Q          719         14,38       95,61
        0,00             0,57             0,13
4          search_securities_sqlxml Q          722         14,44       168,27
        0,00             1,29             0,23
5          account_summary_sqlxml   Q          716         14,32       90,87
        0,00             0,57             0,13
6          get_security_price_sqlxml Q         702         14,04       110,97
        0,00             1,11             0,16
7          customer_max_order_sqlxml Q         680         13,60       70,00
        0,00             0,50             0,10

*** SYSTEM THROUGHPUT ***

The throughput is 18750 transactions per minute (312,50 per second).
```

Figure 63. Q4 test on TPoX for 100 concurrent users and 50 transactions per user after dropping index SEC_SECTOR

Table 8. Difference of the performance result between before and after dropping the index SEC_SECTOR

Search_securities_sqlxml	**Total Time**	**Avg Time**	**Max Time**	**Min Time**
With three indexes	115,73	0,16	0,74	0,00
dropping SEC_SECTOR	168,27	0,23	1,29	0,00

5.5 Query test on TPoX with & without created indexes on XML column

In this testing, I will test the query from Q1-Q7 under the condition with created indexes on XML column and without created indexes on XML column in three tables for different concurrent users and 50 transactions. The test is to observe how the created indexes on XML column influence the whole throughout and CPU utilization in the query performance.

5.5.1 Query test on TPoX workload with created indexes

First I observe the test with indexes which Nicola built up. The indexes for different table are listed as the following:

INDNAME	TABNAME	INDEXTYPE
SQL10021412195...	SECURITY	XRGN
SQL10021412195...	SECURITY	XPTH
SEC_SECTOR	SECURITY	XVIL
SQL11041211220...	SECURITY	XVIP
SEC_PE	SECURITY	XVIL
SQL11041209222...	SECURITY	XVIP
SEC_YIELD	SECURITY	XVIL
SQL11041209243...	SECURITY	XVIP
SECSYMBOL	SECURITY	XVIL
SQL11041211291...	SECURITY	XVIP

Figure 64. Indexes on table SECURITY with created indexes on XML column

INDNAME	TABNAME	INDEXTYPE
SQL10021412195...	ORDER	XRGN
SQL10021412195...	ORDER	XPTH
ORDER_ACCOUN...	ORDER	XVIL
SQL11022814434...	ORDER	XVIP
ORDER_ID	ORDER	XVIL
SQL11022814434...	ORDER	XVIP

Figure 65. Indexes on table ORDER with created indexes on XML column

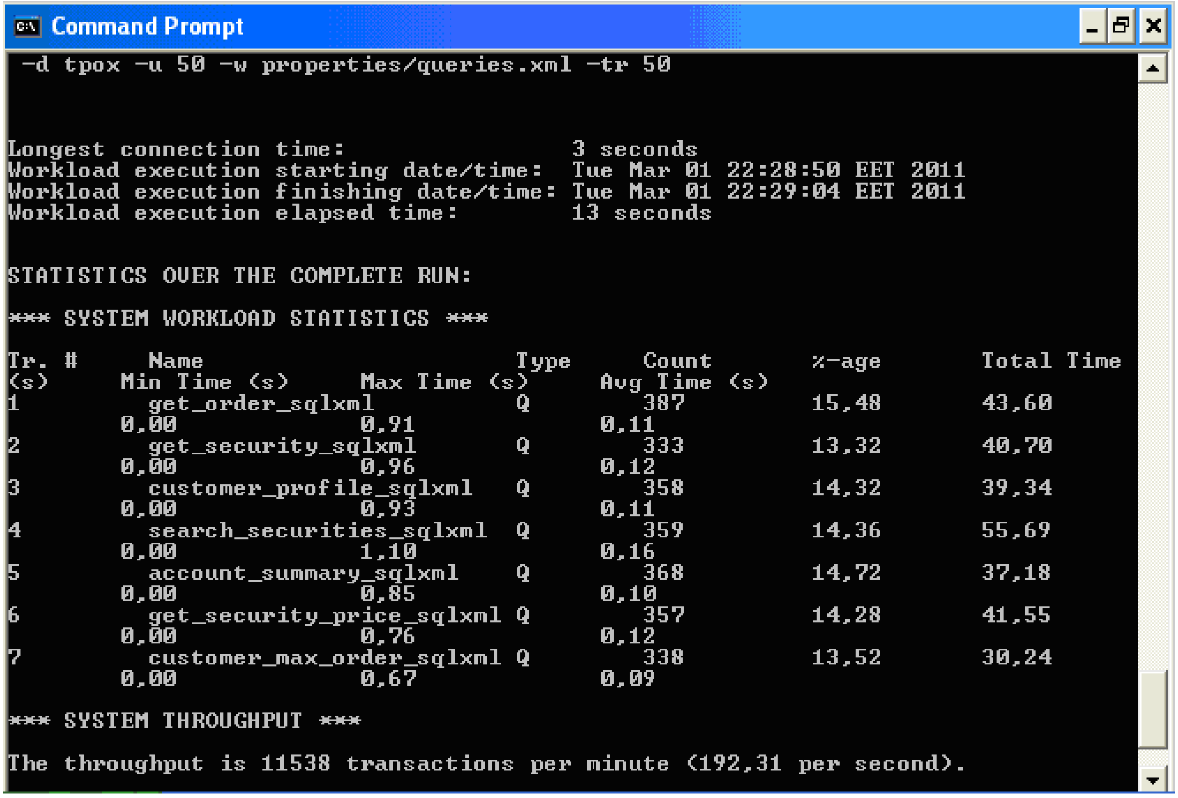
INDNAME	TABNAME	INDEXTYPE
SQL10021412195...	CUSTACC	XRGN
SQL10021412195...	CUSTACC	XPTH
CUSTACC_ID	CUSTACC	XVIL
SQL10021412211...	CUSTACC	XVIP
CUSTACC_ACCO...	CUSTACC	XVIL
SQL11022314370...	CUSTACC	XVIP

Figure 66. Indexes on table CUSTACC with created indexes on XML column

Now I run the query test on the command window.

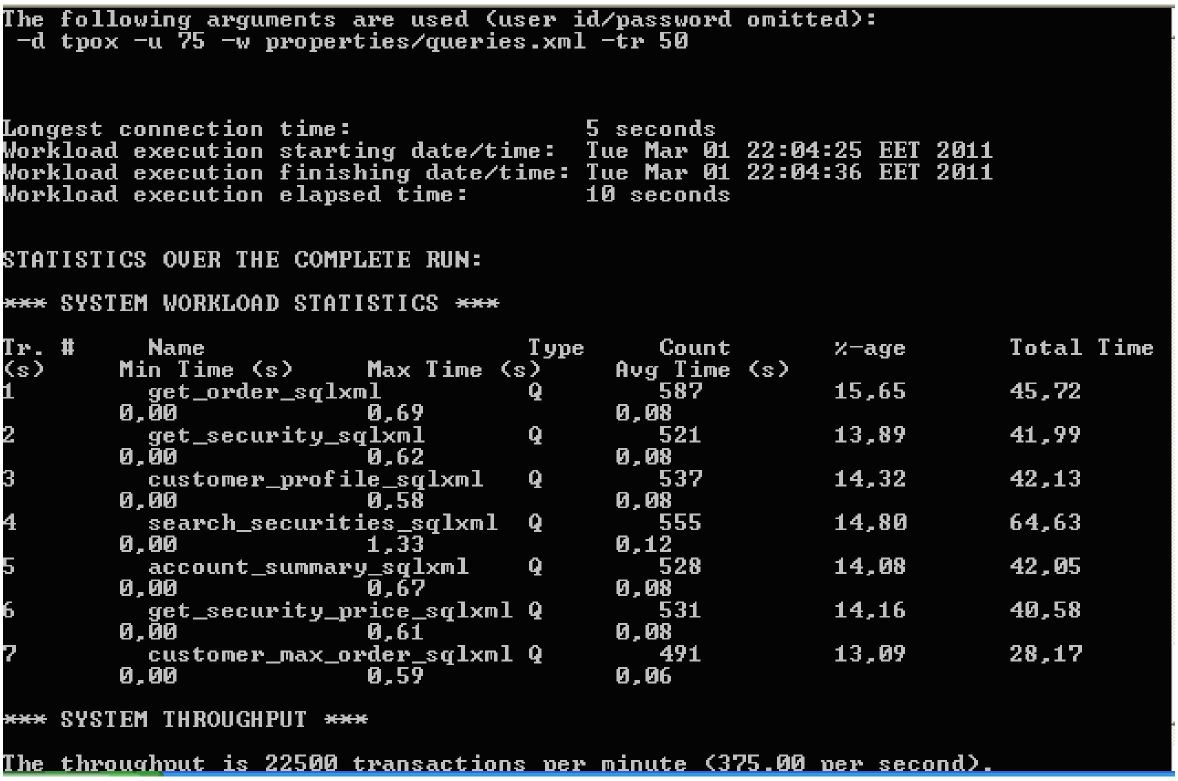
```
 -d tpox -u 25 -w properties/queries.xml -tr 50


Longest connection time:                2 seconds
Workload execution starting date/time:  Tue Mar 01 22:10:23 EET 2011
Workload execution finishing date/time: Tue Mar 01 22:10:30 EET 2011
Workload execution elapsed time:        7 seconds


STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                       Type     Count        %-age       Total Time
(s)     Min Time (s)     Max Time (s)     Avg Time (s)
1         get_order_sqlxml           Q        199          15,92       11,05
        0,00             0,32             0,06
2         get_security_sqlxml        Q        162          12,96       8,70
        0,00             0,23             0,05
3         customer_profile_sqlxml    Q        191          15,28       10,75
        0,00             0,35             0,06
4         search_securities_sqlxml   Q        174          13,92       10,74
        0,00             0,38             0,06
5         account_summary_sqlxml     Q        179          14,32       8,12
        0,00             0,31             0,05
6         get_security_price_sqlxml Q         188          15,04       9,79
        0,00             0,36             0,05
7         customer_max_order_sqlxml Q         157          12,56       6,41
        0,00             0,26             0,04

*** SYSTEM THROUGHPUT ***

The throughput is 10714 transactions per minute (178,57 per second).
```

Figure 67. Query test on workload for 25 users 50 transactions with created indexes



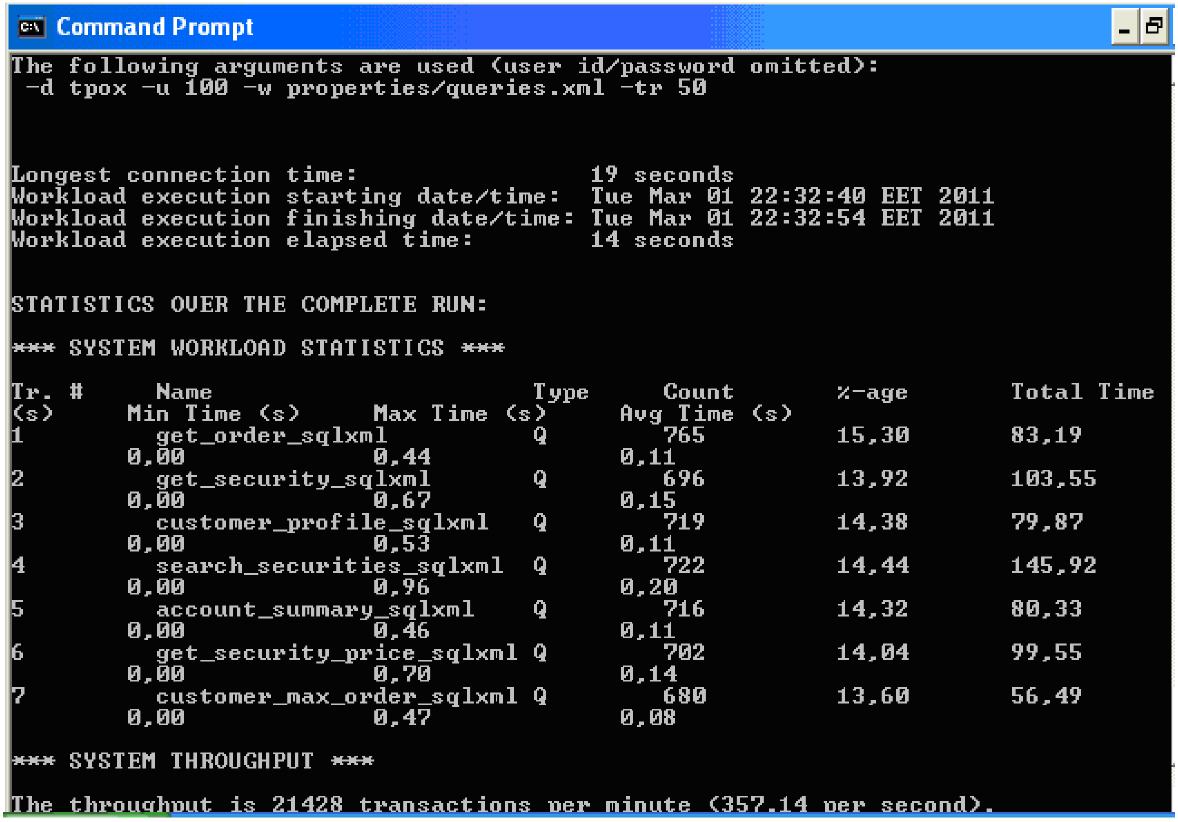
```
Command Prompt
 -d tpox -u 50 -w properties/queries.xml -tr 50


Longest connection time:                3 seconds
Workload execution starting date/time:  Tue Mar 01 22:28:50 EET 2011
Workload execution finishing date/time: Tue Mar 01 22:29:04 EET 2011
Workload execution elapsed time:        13 seconds

STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                         Type      Count         %-age       Total Time
(s)     Min Time (s)      Max Time (s)      Avg Time (s)
1         get_order_sqlxml             Q          387          15,48       43,60
        0,00              0,91              0,11
2         get_security_sqlxml          Q          333          13,32       40,70
        0,00              0,96              0,12
3         customer_profile_sqlxml      Q          358          14,32       39,34
        0,00              0,93              0,11
4         search_securities_sqlxml     Q          359          14,36       55,69
        0,00              1,10              0,16
5         account_summary_sqlxml       Q          368          14,72       37,18
        0,00              0,85              0,10
6         get_security_price_sqlxml Q             357          14,28       41,55
        0,00              0,76              0,12
7         customer_max_order_sqlxml Q             338          13,52       30,24
        0,00              0,67              0,09

*** SYSTEM THROUGHPUT ***

The throughput is 11538 transactions per minute (192,31 per second).
```

Figure 68. Query test on workload for 50 users 50 transactions with created indexes



```
The following arguments are used (user id/password omitted):
 -d tpox -u 75 -w properties/queries.xml -tr 50


Longest connection time:                5 seconds
Workload execution starting date/time:  Tue Mar 01 22:04:25 EET 2011
Workload execution finishing date/time: Tue Mar 01 22:04:36 EET 2011
Workload execution elapsed time:        10 seconds

STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                         Type      Count         %-age       Total Time
(s)     Min Time (s)      Max Time (s)      Avg Time (s)
1         get_order_sqlxml             Q          587          15,65       45,72
        0,00              0,69              0,08
2         get_security_sqlxml          Q          521          13,89       41,99
        0,00              0,62              0,08
3         customer_profile_sqlxml      Q          537          14,32       42,13
        0,00              0,58              0,08
4         search_securities_sqlxml     Q          555          14,80       64,63
        0,00              1,33              0,12
5         account_summary_sqlxml       Q          528          14,08       42,05
        0,00              0,67              0,08
6         get_security_price_sqlxml Q             531          14,16       40,58
        0,00              0,61              0,08
7         customer_max_order_sqlxml Q             491          13,09       28,17
        0,00              0,59              0,06

*** SYSTEM THROUGHPUT ***

The throughput is 22500 transactions per minute (375,00 per second).
```

Figure 69. Query test on workload for 75 users 50 transactions with created indexes



```
Command Prompt
The following arguments are used (user id/password omitted):
 -d tpox -u 100 -w properties/queries.xml -tr 50


Longest connection time:              19 seconds
Workload execution starting date/time: Tue Mar 01 22:32:40 EET 2011
Workload execution finishing date/time: Tue Mar 01 22:32:54 EET 2011
Workload execution elapsed time:       14 seconds

STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                          Type     Count        %-age        Total Time
(s)     Min Time (s)      Max Time (s)      Avg Time (s)
1         get_order_sqlxml              Q        765          15,30        83,19
        0,00              0,44              0,11
2         get_security_sqlxml           Q        696          13,92        103,55
        0,00              0,67              0,15
3         customer_profile_sqlxml       Q        719          14,38        79,87
        0,00              0,53              0,11
4         search_securities_sqlxml      Q        722          14,44        145,92
        0,00              0,96              0,20
5         account_summary_sqlxml        Q        716          14,32        80,33
        0,00              0,46              0,11
6         get_security_price_sqlxml     Q        702          14,04        99,55
        0,00              0,70              0,14
7         customer_max_order_sqlxml     Q        680          13,60        56,49
        0,00              0,47              0,08

*** SYSTEM THROUGHPUT ***

The throughput is 21428 transactions per minute (357.14 per second).
```

Figure 70. Query test on workload for 100 users 50 transactions with created indexes

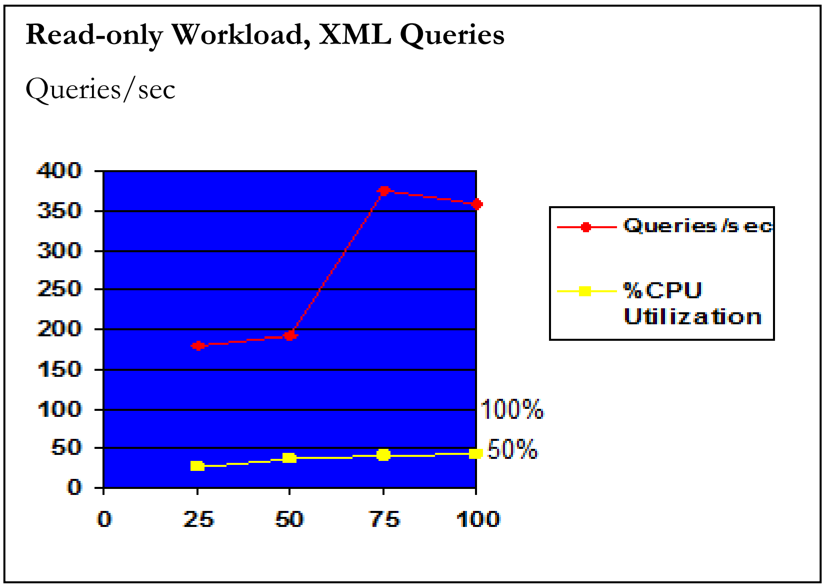


Figure 71. Read-only workload XML queries throughput with created indexes

Figure 71 illustrates the query throughput (left y-axis) as well as the CPU utilization (right y-axis) when the concurrent users are 25, 50, 75 and 100 (x-axis). The query throughput increased with the number of the concurrent users as the CPUs were better utilized. On the other hand, the throughput will show a decrease level when the CPU capacity exhausted. The result is coinciding with Nicola's result. Only difference is the queries throughput amount per second. Since the work environment and system storage in my computer is much less. You could find the value details of the above figure in the following table 9.

Table 9. Value of read-only workload XML queries throughput with created indexes

	25	50	75	100
Queries/sec	178,57	192,31	375	357,16
%CPU Utilization	24,87	35,28	39,74	41,26

5.5.2 Query test on TPoX workload by dropping created indexes

Now I drop the indexes on three tables. I only leave one unique index on every table. The result lists are as the below tables:

INDNAME	TABNAME	INDEXTYPE
SQL10021412195...	SECURITY	XRGN
SQL10021412195...	SECURITY	XPTH
SECSYMBOL	SECURITY	XVIL
SQL10021412211...	SECURITY	XVIP

Figure 72. Indexes on table SECURITY without created indexes on XML column

INDNAME	TABNAME	INDEXTYPE
SQL10021412195...	ORDER	XRGN
SQL10021412195...	ORDER	XPTH
ORDER_ID	ORDER	XVIL
SQL11022814434...	ORDER	XVIP

Figure 73. Indexes on table ORDER without created indexes on XML column

INDNAME	TABNAME	INDEXTYPE
SQL10021412195...	CUSTACC	XRGN
SQL10021412195...	CUSTACC	XPTH
CUSTACC_ID	CUSTACC	XVIL
SQL10021412211...	CUSTACC	XVIP

Figure 74. Indexes on table CUSTACC without created indexes on XML column

Now I run the query test on the command window again.

```
 -d tpox -u 25 -w properties/queries.xml -tr 50


Longest connection time:                  4 seconds
Workload execution starting date/time:   Tue Mar 01 15:34:25 EET 2011
Workload execution finishing date/time:  Tue Mar 01 15:34:42 EET 2011
Workload execution elapsed time:         17 seconds


STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                        Type      Count          %-age        Total Time
(s)     Min Time (s)      Max Time (s)      Avg Time (s)
1         get_order_sqlxml            Q          199           15,92        18,67
        0,00              0,60             0,09
2         get_security_sqlxml         Q          162           12,96        13,94
        0,00              1,01             0,09
3         customer_profile_sqlxml     Q          191           15,28        16,47
        0,00              0,87             0,09
4         search_securities_sqlxml    Q          174           13,92        107,15
        0,03              4,08             0,62
5         account_summary_sqlxml      Q          179           14,32        16,32
        0,00              0,66             0,09
6         get_security_price_sqlxml Q            188           15,04        18,05
        0,00              1,03             0,10
7         customer_max_order_sqlxml Q            157           12,56        35,01
        0,00              2,29             0,22

*** SYSTEM THROUGHPUT ***

The throughput is 4411 transactions per minute (73,53 per second).
```

Figure 75. Query test on workload for 25 users 50 transactions without created indexes

```
 -d tpox -u 50 -w properties/queries.xml -tr 50


Longest connection time:                  8 seconds
Workload execution starting date/time:   Tue Mar 01 15:43:21 EET 2011
Workload execution finishing date/time:  Tue Mar 01 15:43:50 EET 2011
Workload execution elapsed time:         29 seconds


STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                        Type      Count          %-age        Total Time
(s)     Min Time (s)      Max Time (s)      Avg Time (s)
1         get_order_sqlxml            Q          387           15,48        61,25
        0,00              1,41             0,16
2         get_security_sqlxml         Q          333           13,32        52,71
        0,00              1,37             0,16
3         customer_profile_sqlxml     Q          358           14,32        110,87
        0,00              2,69             0,31
4         search_securities_sqlxml    Q          359           14,36        217,27
        0,01              3,23             0,61
5         account_summary_sqlxml      Q          368           14,72        125,33
        0,00              2,94             0,34
6         get_security_price_sqlxml Q            357           14,28        49,34
        0,00              1,56             0,14
7         customer_max_order_sqlxml Q            338           13,52        161,99
        0,01              2,94             0,48

*** SYSTEM THROUGHPUT ***

The throughput is 5172 transactions per minute (86,21 per second).
```

Figure 76. Query test on workload for 50 users 50 transactions without created indexes

```
The following arguments are used (user id/password omitted):
 -d tpox -u 75 -w properties/queries.xml -tr 50


Longest connection time:                6 seconds
Workload execution starting date/time:  Tue Mar 01 15:55:41 EET 2011
Workload execution finishing date/time: Tue Mar 01 15:56:24 EET 2011
Workload execution elapsed time:        43 seconds


STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                        Type     Count          %-age        Total Time
(s)      Min Time (s)     Max Time (s)       Avg Time (s)
1          get_order_sqlxml           Q         587           15,65        154,63
         0,00             1,99               0,26
2          get_security_sqlxml        Q         521           13,89        140,21
         0,00             1,98               0,27
3          customer_profile_sqlxml    Q         537           14,32        132,79
         0,00             1,90               0,25
4          search_securities_sqlxml   Q         555           14,80        651,72
         0,03             7,23               1,17
5          account_summary_sqlxml     Q         528           14,08        131,77
         0,00             1,98               0,25
6          get_security_price_sqlxml  Q         531           14,16        125,67
         0,00             1,98               0,24
7          customer_max_order_sqlxml  Q         491           13,09        192,30
         0,01             5,13               0,39

*** SYSTEM THROUGHPUT ***

The throughput is 5232 transactions per minute (87,21 per second).
```

Figure 77. Query test on workload for 75 users 50 transactions without created indexes

```
The following arguments are used (user id/password omitted):
 -d tpox -u 100 -w properties/queries.xml -tr 50


Longest connection time:                22 seconds
Workload execution starting date/time:  Tue Mar 01 15:51:55 EET 2011
Workload execution finishing date/time: Tue Mar 01 15:52:48 EET 2011
Workload execution elapsed time:        52 seconds


STATISTICS OVER THE COMPLETE RUN:

*** SYSTEM WORKLOAD STATISTICS ***

Tr. #     Name                        Type     Count          %-age        Total Time
(s)      Min Time (s)     Max Time (s)       Avg Time (s)
1          get_order_sqlxml           Q         765           15,30        289,90
         0,00             2,18               0,38
2          get_security_sqlxml        Q         696           13,92        216,45
         0,00             1,48               0,31
3          customer_profile_sqlxml    Q         719           14,38        391,86
         0,00             3,25               0,55
4          search_securities_sqlxml   Q         722           14,44        814,75
         0,03             3,47               1,13
5          account_summary_sqlxml     Q         716           14,32        340,27
         0,00             3,83               0,48
6          get_security_price_sqlxml  Q         702           14,04        223,52
         0,00             1,52               0,32
7          customer_max_order_sqlxml  Q         680           13,60        378,96
         0,01             4,02               0,56

*** SYSTEM THROUGHPUT ***

The throughput is 5769 transactions per minute (96,15 per second).
```

Figure 78. Query test on workload for 100 users 50 transactions without created indexes

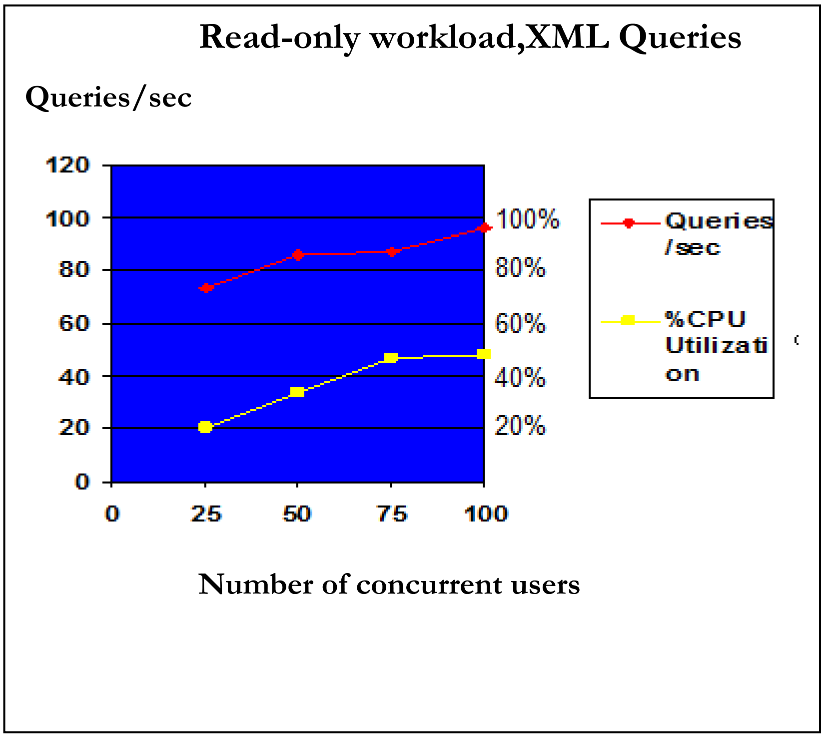


Figure 79. Read-only workload XML queries throughput by reducing the created indexes

Figure 79 illustrates the queries throughput for the different concurrent users by reducing certain indexes. The measurement method is same as the figure 71. Comparing with figure 71, we can see the throughput is much less and CPU utilization is a bit higher. Similarly, in both figures, the query throughputs increased with the number of the users as the CPU were better utilized. More, you could find the value details of the above figure in table 10.

Table 10. Value of read-only workload XML queries throughput without created indexes

	25	50	75	100
Queries/sec	73,53	86,21	87,21	96,15
%CPU Utilization	20,48	33,75	46,58	47,91

5.5.3 Test result comparison

From the above figures, we could see the difference between two tests. After building up the indexes, the throughputs for different concurrent users per second were increased about 3-4 times. The entire figures give a proof how the indexes on XML columns affect and improve the query performance on TPoX benchmark workload.

5.6 Summary of the tests

The aim of first test was to examine the single query Q1 performance without any created indexes on XML columns and with some of created indexes on XML columns in the single table ORDER. To observe the query performance on TPoX workload the test was performed under the condition with and without the index ORDER_ID. As shown in table 4, the result clearly indicates, that the ORDER_ID dramatically affect the performance. Thus, the test simply gives a proof that a created index on XML column can improve query performance much fast.

The goal of second test was to monitor the single query performance Q7 with some of created indexes and without those created indexes on XML columns in the joined table ORDER&CUSTACC. The test was processed by populating three access plans in order to observe how the created indexes on XML column affect the query performance within a joined table. One of the plans is with created indexes ORDER_ACCOUNTID & CUSTACC_ID. Another one is after dropping one index ORDER_ACCOUNTID. The third one is after dropping index CUSTACC_ID. The comparing result table displays how single index affect the query performance in a joined table. A short analysis was also executed, showing that the CUSTACC_ID has a more weight on affection the query performance. This suggests that the document size might have an effect.

Third test was to test the single query Q4 performance with three created indexes on XML columns (SEC_SECTOR, SEC_PE, and SEC_YIELD) also after separately dropping those indexes on table SECURITY. The target of test is to observe how the multiple created indexes on XML columns affect the query performance in a single table. A comparison of the results obtained from the runs were made and demonstrated in the table 7. The data exhibit that the multiple indexes created on XML column in the single table have different effect on query performance with the most affection seen by the SEC_SECTOR. In addition, the SEC _PE has more affection on the query performance comparing to SEC_YIELD. Furthermore, a short analysis was also carried out to explore why these three created indexes have different affection on the query performance.

Forth test was to examine multi-user query performance on the TPoX workloadDriver under the condition with created indexes on XML column and by reducing certain created indexes. A series of multi-user query tests were performed using the seven queries. The workload for

25, 50, 75, 100 concurrent users were executed. After each run, the performance was demonstrated by the figure that indicates how the throughputs were increased with the number of users as the CPUs were better utilized. The results from all runs were further compared. From two performance structures, the throughputs were increased about three times with more created indexes on XML columns.

6 Conclusion

The series of tests show how the indexes on XML columns affect the query performance. More specified, the thesis presented a set of tests and examines to show how XML indexes are used to avoid table scans and provide high query performance based on the TPoX benchmark. In the theory part, I gave an explanation about three types of XML indexes. The indexes on XML columns are illustrated by the structures and moreover how the indexes on XML columns affect the query performance was demonstrated in my experiment part. The results from experiment part are shown in the summary.

In conclusion, from my tests and case studies, I realized that the indexes on XML columns indeed have huge affections on the XML database performance. Especially in modern market, there are a lot of demands on XML database applications. For instance, finance, banking and stock marketing… The topic how to improve the application performance is always to be considered as an important issue. Moreover, how to build up XML indexes becomes a key point to improve the XML database performance. The XML indexes are essential for high query performance, but their usage for query evaluation depends on how the query predicates are formulated. In DB2, the new query operators allow DB2 to generate execution plans for SQL/XML and XQueries. The optimizer can decide to not use an index even if it could be used. According to my project plan and time schedule, I didn't put much research on how to create the more effective indexes on XML columns. This might be a work for future study. I do hope my thesis and study could give a brief report on XML indexes in the DB2 pureXML and it may bring some basic understanding on the topic.

Acknowledgement

I want to thank Dr. Nicola from IBM on his explanations to my questions on TPoX and XML index implementation in DB2 pureXML.

Bibliography

Answers.com 2010.Transaction Processing over XML.URL:
http://www.answers.com/topic/transaction-processing-over-xml.Quoted:15.12.2010.

Böhme.T, Rahm.E. 2001. XMach-1: A Benchmark for XML Data Management. Proceedings of German database conference BTW2001. pp. 264-273.

Franceschet.M. 2005. XPathMark – An XPath benchmark for XMark generated data. International XML database Symposium (XSYM). pp. 129-143.

IBM 2007. DB2 9 pureXML Guide. URL:
http://www.redbooks.ibm.com/redbooks/pdfs/sg247315.pdf. Quoted: 18.02.2011.

IBM 2009. Other database objects associated with XML columns. URL:
http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp?topic=/com.ibm.db2.luw.xml.doc/doc/c0024071.html. Quoted: 08.02.2011.

Kogan. I, Nicola.M. 2009. Transaction Processing over XML (TPOX) Benchmark: Workload Driver Overview and Usage. URL:
http://tpox.sourceforge.net/WorkloadDriverUsage_v2.0.pdf. Quoted: 26.02.2011.

Nicola. M, Kogan. I and Schiefer. B. 2007a. An XML Transaction Processing Benchmark. Proceedings of the 2007 ACM SIGMOD. pp. 1-12.

Nicola. M, Kogan. I, Schiefer. B. 2007b. An XML Database Benchmark: Transaction Processing over XML (TPoX). URL: http://tpox.sourceforge.net/. Quoted: 13.12.2010.

Nicola. M, Kumar-Chatterjee. P. 2010. DB2 9 pureXML Cookbook. IBM Press. United States.

Schmidt. A, Waas. F, Kersten. M. L, Carey. M. J, Manolescu. I and Busse. R 2002. XMark: A Benchmark for XML Data Management. International Conference on Very Large Data Bases (VLDB). pp. 974-985.

Yao. B, Özsu. M. T, and Keenleyside. J. EEXTT 2002 and DiWeb 2002. XBench – A Family of Benchmarks for XML DBMSs. pp. 162-164.

Appendices

Appendix 1. TPoX QUERIES

This appendix presents the code of the queries of the TPoX Benchmark. The percentage characters (%) at the end of queries need to be configured as the statement terminator.

Q1: get_order

```
SELECT XMLQUERY
(
'declare namespace o="http://www.fixprotocol.org/FIXML-4-4";
for $ord in $odoc/o:FIXML
return $ord/o:Order
'
PASSING odoc AS "odoc"
)
FROM order
WHERE XMLEXISTS
(
'declare namespace o="http://www.fixprotocol.org/FIXML-4-4";
$odoc/o:FIXML/o:Order[@ID=$id]
'PASSING odoc AS "odoc", cast (? as varchar(10)) as "id"
)
%
```

Q2: get_security

```
SELECT XMLQUERY
(
'declare default element namespace "http://tpox-benchmark.com/security";
for $sec in $sdoc/Security
return $sec
'
PASSING sdoc AS "sdoc"
)
FROM security
WHERE XMLEXISTS
```

```
('
declare default element namespace "http://tpox-benchmark.com/security";
$sdoc/Security[Symbol=$sym]
'
PASSING sdoc AS "sdoc", cast(? as varchar(10)) as "sym"
)
%
```

Q3: customer_profile

```
SELECT XMLQUERY
(
'declare default element namespace "http://tpox-benchmark.com/custacc";
for $cust in $cadoc/Customer
return
	<Customer_Profile CUSTOMERID="{$cust/@id}">
		{$cust/Name}
		{$cust/DateOfBirth}
		{$cust/Gender}
		{$cust/CountryOfResidence}
		{$cust/Languages}
		{$cust/Addresses}
		{$cust/EmailAddresses}
	</Customer_Profile>'
PASSING cadoc AS "cadoc"
)
FROM custacc
WHERE XMLEXISTS
(
'declare default element namespace "http://tpox-benchmark.com/custacc";
$cadoc/Customer[@id=$id]'
PASSING cadoc AS "cadoc", cast (? as double) as "id"
)
%
```

Q4: search_securities

```
SELECT XMLQUERY
(
'declare default element namespace "http://tpox-benchmark.com/security";
for $sec in $sdoc/Security
return
		<Security>
				{$sec/Symbol}
				{$sec/Name}
				{$sec/SecurityType}
				{$sec/SecurityInformation//Sector}
				{$sec/PE}
				{$sec/Yield}
		</Security>
'
PASSING sdoc AS "sdoc"
)
FROM security
WHERE XMLEXISTS
(
'declare default element namespace "http://tpox-benchmark.com/security";
$sdoc/Security[SecurityInformation/*/Sector=$sector and PE[. >=$pe1 and . <$pe2] and
Yield>$yield]'
PASSING sdoc AS "sdoc", cast (? as varchar(25)) as "sector", cast (? as double) as "pe1", cast
(? as double) as "pe2", cast (? as double) as "yield"
)
%
```

Q5: account_summary

```
SELECT XMLQUERY
(
'declare default element namespace "http://tpox-benchmark.com/custacc";
for $cust in $cadoc/Customer
return
<Customer>{$cust/@id}
```

```
  {$cust/Name}
  <Customer_Securities>
   {
   for $account in $cust/Accounts/Account
   return
     <Account BALANCE="{$account/Balance/OnlineActualBal}"
     ACCOUNT_ID="{$account/@id}">
                  <Securities>
                   {$account/Holdings/Position/Name}
       </Securities>
     </Account>
   }
  </Customer_Securities>
</Customer>
'
PASSING cadoc AS "cadoc"
)
FROM custacc
WHERE XMLEXISTS
(
'declare default element namespace "http://tpox-benchmark.com/custacc";
$cadoc/Customer[@id=$id]'
PASSING cadoc AS "cadoc", cast (? as integer) as "id"
)
%
```

Q6: get_security_price

```
SELECT  XMLQUERY
(
'declare namespace s="http://tpox-benchmark.com/security";
for $sec in $sdoc/s:Security
return
<print>The open price of the security "{$sec/s:Name/text()}" is
{$sec/s:Price/s:PriceToday/s:Open/text()} dollars
</print>
```

```
'
PASSING sdoc AS "sdoc"
)
FROM security
WHERE XMLEXISTS
(
'declare namespace s="http://tpox-benchmark.com/security";
$sdoc/s:Security[s:Symbol=$sym]
'
PASSING sdoc AS "sdoc", cast (? as varchar(10)) as "sym"
)
%
```

Q7: customer_max_order

```
SELECT DECIMAL(CAST(MAX(price) AS INTEGER), 15, 2) AS maxprice
FROM
(SELECT XMLCAST(XMLQUERY(
'
declare default element namespace "http://www.fixprotocol.org/FIXML-4-4";
let $orderprice := $odoc/FIXML/Order/OrdQty/@Cash
return $orderprice
'
PASSING odoc AS "odoc") AS DOUBLE) AS price
FROM custacc, order
WHERE XMLEXISTS
(
'
declare namespace c="http://tpox-benchmark.com/custacc";
$cadoc/c:Customer[@id=$id]'
PASSING cadoc AS "cadoc", cast (? as double) as "id"
)
AND XMLEXISTS
(
'
declare default element namespace "http://www.fixprotocol.org/FIXML-4-4";
```

```
declare namespace c="http://tpox-benchmark.com/custacc";
$odoc/FIXML/Order[@Acct=$cadoc/c:Customer/c:Accounts/c:Account/@id/fn:string(.)]
'
PASSING cadoc AS "cadoc", odoc AS "odoc")
) AS T
%
```