

Part 6. XML Indexes and Access Plans

Hello DB2 friends! My name is Guogen Zhang. I'm a Senior Technical Staff Member at IBM Silicon Valley Lab, responsible for the delivery of pureXML in DB2 for z/OS server.

This is part 6 of the pureXML podcast. The title is “XML indexes and access plans.”

There are three kinds of indexes related to XML in DB2 9 for z/OS. They are DOCID indexes, NODEID indexes, and XML value indexes. When we call XML indexes, we are talking about XML value indexes. A DOCID index is a regular index on the hidden DOCID column in the base table. Since the DOCID column is BIGINT type, there is nothing special in a DOCID index (except that it is always a NPI).

A NODEID index, on the other hand, is on the internal XML table. It contains DOCID and NODEID as the key, where NODEIDs are generated from the third column of an XML table, which is XMLDATA column. A NODEID index is required to access XML data; it provides mapping from a logical nodeID to physical Record ID. It is a sparse index in terms of XML nodes as it's unnecessary to have one entry for each node. On average, it contains two entries per XML table row (record). And DOCID indexes and NODEID indexes are created implicitly by DB2.

Now let's focus on the XML indexes. An XML index is user-created on an XML column. In general, there are other kinds of XML indexes, such as tag index and path index etc. But value indexes are the most important, and that's what DB2 9 supports.

The syntax of CREATE XML index is some minor extension to that of a regular index, for example:

```
CREATE INDEX idxname ON purchaseorders(XMLPO) GENERATE KEYS USING XMLPATTERN '/purchaseorder/items/item/desc' AS SQL VARCHAR(100);
```

This is to create index on the description of items contained in purchaseorder.

XML indexes provide quick search from a value key to the document locations. Logically the location is DOCID and NODEID, and physically it's the RID. The index entries contain the key value, DOCID, NODEID, and RID of the XML table rows (records). There are three important components in CREATE XML index. First is the XML column, the same as any other type of indexes. The second is the XPath pattern. The XPath expresses the interested nodes to index in a document. The XPath that can be used for XML indexes are a subset of XPath, basically without predicates in it. Third, CREATE INDEX also specifies the data type to use in the index.

In DB2 9 we support two data types. One is for numeric values, which uses DECFLOAT. And the other is for strings, which uses VARCHAR, with a length limit. The longest key value is limited to 1000 bytes for a VARCHAR index in UTF-8 encoding.

If you are familiar with DB2 LUW's XML index, there are more data types supported there, including VARCHAR HASHED, and also date and timestamp. DB2 9 for z/OS does not support these. And for numeric indexes, LUW uses DOUBLE, z/OS uses DECFLOAT. The reason is that DECFLOAT provides much better precision than DOUBLE and LUW didn't have the DECFLOAT support.

For XML date and time data, if they do not use timezone, or use the same timezone always, then string comparison provides the same semantics as long as the format follows the ISO standard, so index support can be provided through VARCHAR indexes. This also requires the XPath to use string comparison for indexes to be applicable.

XML indexes possess some special properties. First, it can have zero, one, or more entries from one XML document, depending on the document and index XML Pattern definition. This is different from regular relational indexes, where there is one-to-one correspondence between the keys and the rows. Second, for a numeric index, if some matched node contains a value that cannot be cast to a numeric number, it will be ignored. However, for a string index, if a value is longer than the limit specified in the CREATE INDEX definition, then the CREATE INDEX will fail if data already exists, or insert will fail. Ignoring mistyped data in XML is part of the flexibility of XML data processing.

Indexed nodes are typically the leaf nodes. But you can index non-leaf nodes. In such an index, the key value is generated from concatenation of text nodes under the node that is identified by the XPath. By the way, there is a limit in V9 that an index key has to come from a single XML table row (record).

In general, the more steps involved in XML index pattern, the more expensive it is in generating XML index key entries. Also it's more expensive to support // than XML patterns without //. Typically each XML index adds 10-20% of CPU time on top of the basic insert. Because of the index overhead, follow the same principle in creating indexes: only create indexes that are to be used.

Let's talk about access plans for SQL/XML queries involving XML data.

When there are no indexes that can be used to answer an XML query, document scan (or DocScan) will apply. There is no new access type for this. R-scan will show up in PLAN_TABLE output for the access type column for the EXPLAIN result.

XML indexes can be used for XMLEXISTS and XMLTABLE functions. There are two criteria to apply an XML index to XPath predicates:

The first criterion is that the XML Pattern in the index definition matches the query. The match can be exact or containment relationship, that is, the XML index contains the data required by the query. In general, XML Pattern in an index has to be more general than the XPath in the query. The second criterion is the data type also has to match, either numeric or string. If the two criteria hold, then the index can be applied to the query.

For example, the above index on product description in purchaseorders can be used to answer a query with a predicate on the description. For example, XMLEXISTS('/purchaseorder/items/item[desc = "Baby Monitor"]' passing XMLPO). Usually the XPath query will have the last steps within the square brackets, but in XMLPATTERN of the index, they don't have square brackets.

DB2 can use one or more XML indexes for a query. When one index is used, DB2 will get the DOCID list from the index, and sort it to remove duplicates. Then it goes to the DOCID index on the base table to convert the DOCID list to a base table RID list. That's the major usage of the DOCID index. Then use the RID list to fetch the base table rows and get the XML data aslo. The XML predicates are re-evaluated by DocScan, just like in the regular RID list access. Two reasons for this: one is that during access of XML indexes, XML data is not locked so changes are possible. Two is that the index matching may not be exact. For example, an index with XML Pattern containing // can be used for XPath query without //. In certain sense, it's similar to MQT matching, which is not exact also. Re-evaluation ensures that the returned data satisfies the predicates exactly. The single index access type is "DX" for DOCID list access.

To exploit multiple XML indexes, multi-index access plan is generated. The access type will be "M", with additional "DX", "DI", or "DU" (similar to MI and MU). If two or more indexes are used for AND predicates, DI is used. I.e. two or more DOCID lists are produced from indexes, and then sorted with duplicates removed, and intersected. Then follows the same process as single index access plan (DX).

For index ORing, each of the primitive predicates under OR must have an index matched. For example, [color = "red" or size > 10], you must have two indexes matching color and size for index ORing access to apply. The DOCID lists from all the indexes are unioned to produce a unique DOCID list, then follow the single index access plan.

During the DOCID list processing, including ANDing and ORing, agent storage pool is used to get the storage. The RID list pool is used for RID list processing.

To recap, XML indexes are critical to query performance, and XML indexes use XPath to specify what data to index. Their usage and administration are similar to regular indexes.

Again, if you have any questions, please contact us by sending emails to db2zxml@us.ibm.com. Enjoy pureXML and have wonderful days. Until next time, bye-bye.