Binary XML Storage and Query Processing in Oracle 11g

Ning Zhang Nipun Agarwal Sivasankaran Chandrasekar Sam Idicula
Vijay Medi Sabina Petride Balasubramanyam Sthanikam

Oracle Corporation
{ning.x.zhang, nipun.agarwal, sivasankaran.chandrasekar, sam.idicula,
vijay.medi, sabina.petride, balasubramanyam.sthanikam}@oracle.com

ABSTRACT

Oracle RDBMS has supported XML data management for more than six years since version 9i. Prior to 11g, text-centric XML documents can be stored as-is in a CLOB column and schema-based data-centric documents can be shredded and stored in object-relational (OR) tables mapped from their XML Schema. However, both storage formats have intrinsic limitations—XML/CLOB has unacceptable query and update performance, and XML/OR requires XML schema. To tackle this problem, Oracle 11g introduces a native Binary XML storage format and a complete stack of data management operations. Binary XML was designed to address a wide range of real application problems encountered in XML data management—schema flexibility, amenability to XML indexes, update performance, schema evolution, just to name a few.

In this paper, we introduce the Binary XML storage format based on Oracle SecureFiles System[21]. We propose a lightweight navigational index on top of the storage and an NFA-based navigational algorithm to provide efficient streaming processing. We further optimize query processing by exploiting XML structural and schema information that are collected in database dictionary. We conducted extensive experiments to demonstrate high performance of the native Binary XML in query processing, update, and space consumption.

1. INTRODUCTION

As XML evolved as a data model for semi-structured data and the *de facto* format for data exchange, relational database systems have been extended to offer supports to store, update, and query XML data. Oracle has provided XML data management capabilities for many years by means of an abstract data type XMLType. This data type provides a unified and consistent interface that encapsulates multiple storage and indexing options. Since different storage models have their own pros and cons, the user can choose the one that fits their query and data manipulation requirements best without changing the application code.

There are basically three approaches to storing XML documents in RDBMS: (1) storing the original XML documents as-is in a LOB column (e.g., [17, 23]), (2) shredding XML documents into object-relational (OR) tables and columns (e.g., [28, 4]), and (3) designing a *native* storage format for XML data model from scratch (e.g., [11, 29, 22]). Each approach has their own advantages and disadvantages. For example, the LOB storage approach is the simplest one to implement and support. It provides byte-level fidelity (e.g., it preserves extra white spaces which may be ignored by the OR or the native formats) that could be needed for some digital signature schemes. CLOB is also efficient for inserting or extracting the whole documents to or from the database. However it crawls in query processing due to unavoidable XML parsing at query processing time.

On the other hand, the OR storage format, if designed and mapped correctly, could fly in query processing, thanks to many years of research and development in object-relational database systems. However, insertion, fragment extraction, structural update, and document reconstruction require considerable processing. In addition, for schema-based OR storage applications need to have a well-structured, rigid XML schema whose relational mapping is tuned by a DBA in order to take advantage of this storage model. Loosely structured schemas could lead to unmanageable number of tables and joins. Also, applications requiring flexibility and schema evolution are limited by those offered by relational tables and columns. The result was that applications encountered a large gap—if they could not map well to a relational way of life due to tradeoffs mentioned above, they suffered a big drop in performance or capabilities.

To start off, Oracle took advantage of existing storage technologies (CLOB and OR storage) as much as possible. However, based on working with many customer use cases, it became clear that these storage options have many tradeoffs that make it difficult to satisfy a large class of XML applications. Oracle 11g introduces a native storage format that addresses the above difficulties and the following additional challenges that arise from real-world business applications.

Schema chaos: In this scenario, customers want the flexibility to manage XML data that may or may not have schema, or may have "any" schema. For instance, a telecommunication customer wants to manage XML data generated from different towers, which generate documents with slightly different schemas from each other. They want to store them in one table and perform efficient query on the shared common pieces.

	CLOB	OR	Binary XML
Query	poor	excellent	good/excellent
DML	poor	good/excellent	excellent
document retrieval	excellent	good/excellent	excellent
schema flexibility	good	poor	excellent
document fidelity	excellent	poor	good/excellent
mid-tier integration	poor	poor	excellent

Table 1: Comparisons of different storage models

Schema evolution: This is a scenario for schema flexibility in the time dimension. That is XML schema could be changed over time due to new application requirements. XML documents conforming to one schema should be evolved to conform to a new schema. The underlying storage should be able to support the migration efficiently.

Mid-tier integration: Most Oracle customers deploy their applications in a multi-tier architecture requiring that XML processing needs to work in concert with mid-tier Java business logic. Therefore, the storage format should support efficient data transfer from database layer to the mid-tier.

Performance: Most of our customers expect high performance in updating and querying of XML data. Although space consumption is usually not a major concern, it is a highly desirable feature particularly for customers that require XML data transferred over the wire. In addition, many operational features such as partitioning are also requested by many customers that manage large amount of data.

Table 1 compares different storage models with respect to different application requirements. To summarize, the Binary XML storage is intended to satisfy many application needs for which XML itself was originally envisaged. It is designed to bridge the gap between relational performance for highly structured use cases and the needs of applications that require schema flexibility and mid-tier integration.

In this paper, we show our experience of building a more native XML storage that provides high performance with schema flexibility. The main contributions of this paper are as follows.

- We introduce the Binary XML format that provides a good balance of compact size, decoding and encoding speeds, as well as the basis of storing and indexing XML in a database (Section 3).
- We show that the Binary XML storage based on the above format can efficiently handle common database operational needs (e.g., supporting XML indexes, partitioning, updates, and schema evolution) (Section 3.4).
- We propose a suite of novel query processing and optimization techniques that fully leverage the format to provide good performance for XML processing (Section 4). The core of the query processing techniques is a *search-based decoder* that allows us to push down XPath pattern matching to the storage layer, where physical optimizations can be achieved by exploiting the storage format. To further speedup tree navigation, the decoder can also utilize a per-document *summary*, a lightweight navigational index, which leads to sub-linear I/O complexity. For complex XQuery expressions, we propose novel structural/schema-aware rewrite and caching techniques to simplify the XQuery to SQL translation as well as to save CPU and I/O cost. The combination of all these techniques may significantly improve the query performance without sacrificing DML performance.
- We conducted extensive experiments that compare the performance of query processing, DML and space consumptions (Section 5).

2. RELATED WORK

There are generally two widely adopted approaches to support XML storages in DBMS—the extended relational approach and the native approach. In the extended relational approach, XML documents are converted to object-relational (OR) tables and are stored in relational databases or in object repositories (e.g., Shore [6]). In the native approach, XML documents are stored using a specially designed native format. Each of the two approaches has advantages and disadvantages, although both are feasible for supporting XML queries. Due to the lack of space, we briefly survey the native storage and query processing approach and compare them with Oracle Binary XML format.

2.1 Native Storage

Native XML storage has been studied for many years. Kanne and Moerkotte propose the Natix storage system [15, 11] that partitions a large XML tree into subtrees, each of which is stored in a *record*. The partition is designed such that the record is small enough to fit into a disk page. Updating is relatively easy since insertions and deletions are usually local operations to a record. In general, this approach is more flexible than the interval or preorder-postorder encodings in handling frequent updates. IBM System RX [3, 13] employs a similar technique, which partitions large XML trees into small subtrees and uses an auxiliary structure *Regions Index* to connect them. Each tree node keeps pointers to its parent and children to support fast navigation. Updating to the XML tree is similar to what Natix does. Oracle Binary XML storage uses a different approach by serializing XML document in a logically sequential format. This format is similar to the token stream defined in the BEA streaming XQuery processor [12] (now the Oracle service bus token stream). In addition to providing serialized access to the input XML for streaming processing, Oracle Binary XML format is also optimized for persistent storage. With the help of auxiliary data structure, Binary XML supports fast sublinear navigational XPath evaluation (see Section 4.3 for detail). Binary XML storage also supports efficient piecewise update on top of Oracle 11g SecureFiles storage, which extends the BLOB structure by supporting compression, encryption, deduplication, and piecewise update [21].

Microsoft SQL Server stores XML typed data as a byte sequence of large binary object (BLOB) to preserve the XML data model faithfully [24, 25, 20]. SQL Server 2008 has a limitation that XML data has to be stored in a separate table if an XML index is needed (unless an Index Organized Table is used). Oracle Binary XML storage does not have this limitation and XML data can be stored in the same table as relational data. Also, for evaluating XQuery in the absence of XML Index, SQL Server 2008 manifests a node table containing one row per node at run time. This seems to be expensive in terms of performance. Oracle Binary XML storage provides a streaming evaluation of XPaths.

Another approach of native storing trees is to store tree nodes in some serialized format. Koch proposes the *Arb* storage model [16] to store a tree in its binary tree representation on disk. It is well known that any ordered tree T can be translated into a binary tree B by translating the first child of a node in T to the left child of the corresponding node in B and the following sibling in T to the right child in B. Arb stores tree nodes in document order. Each node uses two bits to represent whether it has a left child and/or a right child. Zhang et al. [29] propose a serialized storage format for XML tree data model. The idea is based on the fact that an ordered tree has an equivalent representation of a balanced parenthesesed string, where the open/close tag of an XML element corresponds to an open/close parenthesis. Navigating on the tree is then translated to finding corresponding open/close parentheses in the string. Updating to the tree is also easily translated to inserting/deleting a substring in the string representation. Wong et al. [27] extended [29] by introducing hierarchies in the balanced parenthesis strings. Oracle Binary XML storage format falls into this category of serialized storage format. In addition to representing XML tree data model as balanced parentheses, Oracle Binary XML encodes into the stream the schema information and other structural properties, based on which more optimization can be performed during query processing. Oracle Binary XML also embed data into the encoding stream while others do not.

Recently, AgileDelta proposed a binary XML format called Efficient XML (EFX) [2] to the W3C working group. The focus of EFX is on superior compression ratio based on XML schema or structural information. The motivation of EFX is to be able to transfer XML data efficiently over the network. While this motivation is one of design goals of Oracle Binary XML as well, the latter is also designed to accomplish much wider application requirements, including efficient update and indexing capabilities.

2.2 Query Processing

Path queries can be evaluated based on these native storage systems by navigational approach. Basically there are two types of navigational approach: query-driven and data-driven. In the query-driven navigational operators (e.g., Natix [5]), each location step in the path expression is translated into a transition from one set of XML tree nodes to another set. In the data-driven operators (e.g., Yfilter [9] and XNav [14]), the query is translated into an automaton and the data tree is traversed according to the current state of the automaton. The query-driven approach is easier to implement, but it may need multiple scans of the input. On the other hand, the data-driven approach only needs one scan of the data, but its implementation is much complex.

The streaming evaluation techniques proposed in this paper is similar to YFilter, but with a number of novel extensions. First, we propose a search-based decoder that is lightweight and much faster than an pure NFA pattern matching. One of the reasons is that the decoder have the complete information about storage metadata which can be exploited to speedup navigation. One of the novel storage metadata is called *summary*, which can be treated as a lightweight navigational index. Secondly we propose a number of novel optimization techniques based on the schema and structural information that rewrite the XML queries in a much simpler form and execute the query in a faster fashion.

3. ORACLE BINARY XML STORAGE

In this section, we introduce the design consideration of Oracle Binary XML, the storage format, and how it handles the challenges mentioned in Section 1.

3.1 Storage Format

The Oracle Binary XML format is designed to be compact and efficient for a wide variety of operations including streaming XPath evaluation, fragment extraction, node-level updates, conversion to/from XML 1.0/1.1 (text) as well as building a DOM. It works well in all tiers including the database tier, on XML schema-based as well as non-schema-based documents. In the Oracle Database, the underlying storage for a binary XML document is a BLOB column, hence allowing binary XML documents to leverage Oracle SecureFiles LOB features such as encryption, deduplication, compression and piecewise update. Due to the space limitation, only the key features of the format are explained below. A complete definition and explanation can be found at the Oracle Binary XML RFC [7].

Binary XML lays the foundation of the native XML database by storing the complete XML InfoSet [8] as well as the XQuery data model [10], and supporting scalable performance for common XML processing tasks such as query processing, XML indexing, XML Schema validation and evolution. From a high level viewpoint, the Binary XML format can be viewed as a serialized set of SAX [19] opcodes (operation code) that are persist in a logically sequential form. In addition, the format has the following properties.

- The format allows mixing of data and metadata to provide complete schema flexibility. For applications that have more structured needs, the format also allows metadata to be stored separately.
- The format itself is transferable as a document or file over standard protocols by fitting into IETF MIME type conventions. This enables Oracle mid-tiers and other parties to handle the format directly without converting back-and-forth to XML in text form.
- The format attempts to provide relational performance and functionality by approaching a relational row format for very structured schemas. The Binary XML storage option takes advantage of XML schemas, if available, to store a highly compacted XML using Oracle data types such as numbers and dates. This has the added advantage of faster processing by saving on data conversion costs.
- The format provides a mechanism of locators that serve as node identifiers and can be used to start processing from an arbitrary point. This forms the basis of XML indexing and streaming evaluation for XPaths.

3.1.1 *Tokenization*

One of the major compression techniques is the use of a token table (i.e., symbol table) to minimize space needed for repeated items in XML. The format supports tokens of the following types: QNames (for elements/attributes), Namespace URLs, and Namespace prefix. These are mapped to an ID (integer) as follows:

NamespaceURL	$\longleftrightarrow$	NamespaceTokenID
NamespaceTokenID, LocalName	$\longleftrightarrow$	TokenID
NamespaceTokenID, Prefix	$\longleftrightarrow$	PrefixTokenID

Token definitions can be specified in the encoded document itself or by reference to a *token repository*. An inlined token definition may appear at any point in the encoding, assigning integer token IDs to each of these items. A token repository allows a set of related or similarly structured documents to share a set of token definitions. Tokens from a repository can be used by specifying the globally unique identifier (GUID) of the processor or repository in the document header. This allows for several optimizations including compression, compilation of XPaths using token IDs, as well as the ability to decode the InfoSet of any subtree without starting at the beginning of the document.

3.1.2 Format Overview

The format consists of a set of document *sections*. A section is a self-contained unit of data transfer, and represents either an entire document or a subtree. In case of sections corresponding to subtrees within a document, a section reference is used to provide a link from the parent section to the child section. The ability to create sections at the level of elements has many benefits including scalability and more efficient storage layout.

Each section has a header, followed by the actual XML data represented as a set of instructions. Each instruction consists of an opcode followed by its operands. These instructions correspond to a document-ordered serialization of the XML InfoSet. The section header contains meta information such as processor GUID (to which token definitions and schema references are scoped), DocID (identifier of the document to which this section belongs), PathID (identifier for the simple path from the root node of the section), Order key (node unique identifier containing the ordering information corresponding to the root node of the section), and flags (indicating which of the above fields are present, whether there are inline token definitions in the section etc).

The section data can be encoded in one of two modes: *basic* or *chunked*. The basic mode consists of a single chunk possibly containing inlined token definitions, in which case the Binary XML decoder needs to process the section instruction by instruction, starting from the beginning. In the chunked mode, a token definition is present at the beginning of the section followed by a set of chunks free of token definitions. This guarantee can speed up processing by allowing the receiver to process Binary XML data in that chunk without scanning it instruction by instruction. For example, if we insert a section into a database, we can bulk append the node data in the section chunk into a target BLOB without interpreting every opcode in the Binary XML stream.

In the general case, there is one Binary XML opcode corresponding to one SAX event (e.g., `START_ELEMENT`, `END_ELEMENT` and so on). If an element contains multiple text node children and/or has interspersed comments, we need to use these opcodes to mark the beginning and end of an element. However, we can optimize it in many cases where the encoding target is an attribute or the element is a simple element that it does not have any child. In these cases, we use a single instruction to represent the whole element, without the need for a separate `END_ELEMENT` opcode.

Opcodes are of one-byte fixed length. Instructions may only have a single operand of variable length, and that operand must be the last one. In this case, the length of the variable length operand itself must be the first operand. The node data values can be of 1/2/4/8-byte length. The first byte of the length indicates the encoding type. For example, the higher two bits 0x00 indicates the encoding type is string and 0x01 binary. The following are some types of opcodes:

- Token Definition opcodes allow definition of different types of token IDs.
- Element & Attribute opcodes specify the token ID or kid number (the ordering of children based on XML schema) of the attribute or element, and optionally a prefix ID and type ID.
- Typed & untyped data opcodes specify the length and value of the data operand of the current node. The typed opcodes also specify the type of the data item.
- Schema-related opcodes indicate the start and end of a schema scope.
- DTD-related opcodes represent DTD constructs like element & attribute definitions.
- Opcodes for Text, CDATA, namespace declaration, processing instructions and comments. Character data is encoded in the UTF-8 character set.

3.1.3 Schema-based Encoding

If the document conforms to a particular XML schema, the schema can be used for more optimal and type-aware encoding. For example, occurrence constraints and type definitions can be used to derive a more compact and processing-efficient representation of the XML InfoSet. Text nodes are more compactly encoded using the data types (integer, float etc.) specified in the schema. In highly structured cases, this can result in the binary representation containing just the lengths and values of element/attribute data (i.e., text nodes represented in typed format).

When an XML schema is registered, each distinct element or attribute (distinct by name & namespace) in the schema is assigned a "property ID". This ID is unique within a set of related schemas (related by include/import). The property ID is added as a special attribute of the corresponding element or attribute declaration in the registered schema.

The registration process also computes the list of possible child elements and attributes for each complex type in the schema. Each element or attribute in this list is assigned a sequential "kid number" (kidNum), to be used as a more compact identifier than the property ID. Since the kidNum is a local identifier for the child property within the context of the parent element, it is typically smaller than the property ID. An annotation is added to the complex type mapping the property IDs to the corresponding kid numbers. For example, consider the following snippet of XML schema.

```
<xsd:annotation>
  <xsd:appInfo>
    <csx:kidList sequential=‘‘true">
      <csx:kid propertyID=‘‘3456" kidNum=‘‘1"/>
      <csx:kid propertyID=‘‘3457" kidNum=‘‘2"/>
    </csx:kidList>
  </xsd:appInfo>
</xsd:annotation>
```

In particular, the "sequential" attribute in the above annotation refers to the usage of the schema-sequential mode.

3.1.4 Schema-sequential Mode

Binary XML storage can detect and annotate two special modes, *schema-sequential* mode and *array* mode, during encoding and they are used for efficient query processing.

The schema-sequential mode can be used whenever the Binary XML encoder determines that for a particular element, all of the immediate element children must appear in a fixed order. Typically, this is the order in which they appear in the schema model. This case exists when the content model for a complexType element consists only of any number of `<sequence>` and `<choice>` elements with maxOccurs=1. The schema-sequential mode cannot be used if there is any `<sequence>` or `<choice>` group with maxOccurs > 1, or when there is an `<all>` group.

In schema-sequential mode, the Binary XML stream consists only of the data values and does not contain any property IDs or kidNums. Hence, the Binary XML decoder must keep track of the current kidNum to infer the node information for each piece of data. The current kidNum is increased after each child operand. Note that even in the schema-sequential mode, the child operands could carry explicit token ID known as the *partial schema-sequential* mode. This allows us to handle Substitution Groups where the declared element has been substituted by another element.

3.1.5 Array Mode

The array mode is used whenever the binary XML processor detects multiple adjacent elements with the same QName. A special opcode, `ARRBEG`, is used to indicate the beginning of the array mode, and `ARREND` to indicate the end. In the array mode, the property or token ID (either explicitly in the stream or derived from the kidnum) for the previous element is implicitly reused for each subsequent data item encountered until `ARREND` is encountered. There are two flavors of array mode—scalar mode and complex mode. In complex array mode, the contents of the array may be of complexType, and an end-of-element opcode is used to delimit each array element. In scalar array mode (used for simple typed elements), each array element contains a single text node, and is encoded using a simple DAT opcode containing only the actual data.

3.2 Locators

One of the major issues in the design of the format is the amount of state that must be built up while processing a binary XML stream. To handle XPath processing for use in queries, indexes, printing, and XInclude, it is desirable for indexes and other external reference mechanisms to be able to use byte offsets to point into the stream without requiring processing of the entire stream from the beginning. This requires that any state necessary to start processing at an arbitrary instruction be serializable into a small, fixed set of bytes. That state, along with the byte offset and stream reference, forms a *locator*, which can be used by indexes and other external references to start Binary XML processing.

Binary XML uses the start of a complex element definition as the point at which stateless processing may commence. It allows jumping directly to the point. The state needed to start processing at any arbitrary node is the state built up from the start of the parent element. An implementation of a locator might contain the current schema ID, a byte offset into the underlying Binary XML stream, and bits indicating whether sequential mode and array mode are in effect. If the locator is in some special mode, more information is included, such as a kid number and parent property ID, or the last token ID processed.

3.3 Schema Evolution

XML Schemas in Oracle serve two purposes: (1) they describe the structure of the XML documents as defined in the W3C XML Schema standard, and (2) they can be annotated to dictate the mapping between the documents and physical layout on disk in some cases.

The XML Schema significantly influences the Binary XML format of a XML document. In fact, several pieces of information contained within the XML Schema are exploited to optimize the Binary XML format. However, this implies that we need efficiently maintain the conformance of the document storage when the schema evolves. The evolution of XML Schemas are constrained such that: (1) the Binary XML instances encoded using the old XML Schema can still be decoded correctly using the new XML Schema, (2) the new XML schema should validate a superset of the XML instances that were valid against the old XML Schema, and (3) schema annotations that impact the Binary XML format cannot be modified. Some of the schema evolutions supported by Binary XML are adding an optional attribute to a complexType or AttributeGroup, increasing the value of maxOccurs, adding values to enumeration SimpleType at the end of the enumeration, increasing the value of maxLength facet, and adding mixed content.

When a schema is evolved by adding new (optional) elements in the middle of the sequence, the newly added children are not encoded using schema-sequential mode. The kidNums of the earlier children are not changed, hence existing Binary XML data will continue to be valid per the new version of the schema. The new element is assigned a new kidNum, and the maxSeqKidNum value is set such that the earlier children are encoded using schema-sequential mode. The new element has to be encoded with an explicit kidNum or token ID value, i.e., partial schema-sequential mode.

3.4 Operational Deployment

Oracle Binary XML is designed to support the entire lifecycle of database applications. This means it supports DDL, DML, query (SQL/X and XQuery), partitioning, XML indexing, importing/exporting, expedite data loading, as well as integration with mid-tier APIs (e.g., Java and C). We highlight some features here due to space limitations:

Piecewise Update Binary XML format allows for piecewise updates by applying only the changed portions to disk. This support builds on delta-update features provided by the Oracle SecureFiles [21] whereby the entire data does not need to be over-written but simply the changed portions. This reduces the I/O bandwidth significantly. In many deployments, maintenance of the XML index could easily be the dominant factor during the execution of updates. However, in Oracle, the index is also made aware of the exact changed portions whereby only those rows in the XML index that are related to the affected portions are updated. Together, these two optimizations result in order of magnitude better update processing compared to typical CLOB and OR uses where the entire document might needed to be loaded/updated/written to disk.

Partitioning Binary XML storage supports partitioning based on Virtual Columns. This leverages the benefits of partitioning provided by Oracle Database. The benefits include partition pruning and partition-wise joins which result in much better performance and higher scalability. Partitioning also eases administration of large data. A Virtual Column (VC) is a user accessible table column that does not exist on disk but is evaluated based on values of other columns. For using partitioning with Binary XML the user creates a VC using either XMLQUERY or EXTRACTVALUE operators. The operator for VC is evaluated to determine the partition key.

Mid-tier processing: The Binary XML format can be understood by the Oracle mid-tier products allowing many real-world applications to obtain improved performance. The application logic in the mid-tier can directly process the binary XML shipped from the database. This avoids the overhead of serialization in the database and the cost of repeated parsing in the mid-tier. In a similar vein, XML serialized data can be encoded directly to the Binary XML format in the mid-tier thus freeing more expensive database CPU cycles.

4. QUERY REWRITE AND PROCESSING

One of the major design considerations of Binary XML storage format is to support efficient query processing. The Binary XML storage would typically be used in conjunction with an XML index for speeding up query access. The index is designed to handle unstructured XML content (by means of a path-value storage) as well as structured content (by means of a property table). Query processing in the presence of these indexes is covered in [18]. In this section, we focus on query processing that leverages the binary XML format assuming that desired documents or fragments might have already been identified by the index. We start by briefly introducing the techniques that rewrites XQuery queries into Binary XML specific operators and row sources (constructs that produce rows). A more comprehensive introduction to Oracle's approach to rewrite and normalization can also be found in [18]. Then we present our NFA-based streaming query processing and optimization techniques.

4.1 XPathTable Rewrite

The streaming XPath evaluation mechanism available with Binary XML is exposed to other layers in the RDBMS in two ways: (1) functional streaming evaluation, and (2) XPathTable row source. Functional streaming evaluation is achieved by using the NFA-based XPath evaluation from XML-related SQL operators. XPathTable row source implements the row source abstraction. That is it produces rows containing one or more columns and can be joined with other row sources. The columns can be either of XMLType or SQL primitive types such as `NUMBER` and `VARCHAR2`. By implementing the row source interface, we leverage a lot of the existing functionality in the RDBMS such as the relational optimizer and SQL query processing engine.

XPathTable row source is created automatically during query optimizations (rewrite) at compilation time. Just like the XMLTable construct in SQL/XML, XPathTable is a construct that bridges the XML data model to the relational data model—it takes an XMLType data as input and produces relational tables, which can also contain XMLType columns. The XPathTable row source is create for two purposes: (1) to evaluate multiple XPaths in one pass in streaming fashion, and (2) to handle XPath predicates, rewrite XMLTable and XMLQuery constructs. In the second case, XPathTable row source serves roughly the same purpose as a TABLE(XMLSEQUENCE) construct on Object Relational Storage which produces a virtual table row source. Though TABLE(XMLSEQUENCE) can be used against Binary XML storage, XPathTable has much better functionality and performance.

If a set of XPaths are evaluated on the same base table column, we detect this at query rewrite time and transform the query by introducing the XPathTable row source and replacing all the original references with columns of the new XPathTable. The advantage of this is that all these XPaths will be evaluated in one pass over the document using the NFA based streaming evaluation (see Section 4.2).

Another case of using XPathTable is for projecting rows from XML fragments. This case arises in XPath Predicates, XMLTable and XML Query FLWR constructs. For example, consider the following XQuery:

```
for $b in $doc("XMARK")/site/people/
           person[@id = "person0"]
return $b/name/text()
```

This query can be rewritten to XMLTable and Oracle XML operators as follows.

```
SELECT
  (SELECT SYS_IXMLAGG(P.C2) COLUMN_VALUE
   FROM XMLTABLE(
     '/site/people/person' PASSING OBJECT_VALUE
     COLUMNS C0 XMLTYPE PATH '.',
             C1 VARCHAR2 PATH '/person/@id',
             C2 XMLTYPE PATH '/person/name/text()'
     ) P
   WHERE P.C1 = 'person0'
) RET
FROM XMARK XM
```

During query rewrite we transform the XPath predicate on `id` attribute into a SQL predicate. To be able to do that, we need to have a mechanism to generate one row per `person` node in the document. For this (and for evaluating many other kind of path expressions where we need one row per node in the doc) we use the XMLTable construct. Here, XMLTable `P` produces one row per `person` node with three columns `C0`, `C1` and `C2`. The XPath `/site/people/person` is referred to as the *driving XPath* for the XMLTable. The input to XMLTable in this case is the XMLType column `OBJECT_VALUE` in the table `XMARK`. This input operand is called the *driving operand* for XMLTable. The `SYS_IXMLAGG` is Oracle XML aggregation operator and gathers the results from all the rows from the XMLTable `P` into one fragment. All the XMLTable columns are produced by one pass over the input XML documents.

The above rewritten query can be further rewritten using XPathTable by simply replacing XMLTALBE with XPathTable. One of the reasons for the mapping is that XMLTable is a

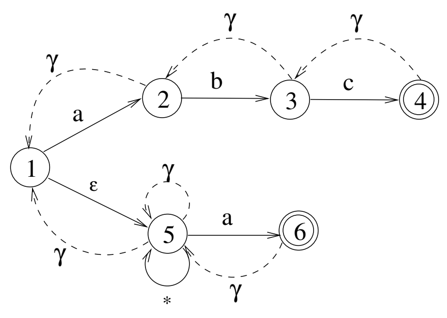

Figure 1: NFA constructed from //a and /a/b/c

logical function and it can be rewritten to different evaluation operators, and XPathTable is a physical operator on Binary XML input and using streaming evaluation introduced in Section 4.2.

4.2 Baseline Streaming Evaluation

XPath path expressions allow users to traverse XML trees in different directions (termed "axes"), to filter tree nodes by specifying predicates on tag names and values, and to return qualified tree nodes. By analogy, path expressions on trees are similar to regular expressions on strings: they both allow the users to specify a pattern of interest in the data to match, except that regular expressions specify patterns in a character sequence (e.g., the regular expression "Elapsed.*seconds" find the string that contains substrings "Elapsed" and "seconds" in that order regardless of any characters in between), while path expressions on trees can specify node tags, value constraints and structural relationships between nodes. For example, a path expression "//purchaseOrder[@orderDate < '12-04-2008']/item" returns all items of any purchase orders whose order date is less than December 04, 2008. In this path expression, // and / are axes that represent the ancestor-descendant and parent-child relationship, respectively; purchaseOrder represents elements whose tag name is "purchaseOrder" and @orderDate represents attributes whose names are "orderDate". The parts in the square brackets [ ] are called predicates, which could be a path expression itself or a relational expression (such as the one in the above example) between a path expression and a value or between two path expressions. In summary, a path expression evaluation operator needs to find all tree nodes that satisfy three types of constraints: tag name constraints, value constraints, and structural constraints.

In analogy to the fact that regular expressions are usually evaluated using finite state machine (FSM) or finite automata (FA), path expressions can also be efficiently evaluated using finite automata. Oracle 11g implements a non-deterministic finite automaton (NFA) to evaluate a set of simple path expressions in one scan of the input encoding. For example, we can combine the two simple path expressions //a and /a/b/c into one NFA as shown in Figure 1[1]. The benefits of this normalization are that we can potentially combine different path expressions in an XQuery FLWOR expression or SQL/XML query into one NFA and evaluated them in one pass of the XML data, resulting in a much efficient I/O cost.

[1]A γ-transitions are taken if no successful matches are found till the end of the context element. Since a γ-transition is implicit for every regular transition, they can be ignored from the diagram.

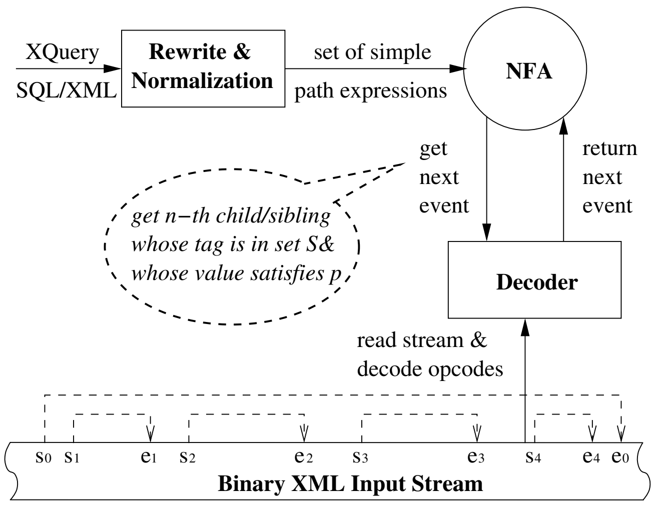

Figure 2: Binary XML Streaming Evaluation Architecture

Figure 2 shows the data flow of the NFA-based evaluation system. In this system, the NFA module obtains a set of simple path expressions and constructs a state machine (states and transitions between states) accordingly. During evaluation, the NFA asks the underlying Decoder module to get the next event in sequence from the input stream. Similar to a standard XML SAX event, which contains information such as whether the input stream encountered the beginning or end of a document, element, attribute, or value, a Binary XML event also contains information related to Binary XML streaming evaluation, e.g., properties of the element in case of schema-based storage. The job of the decoder is to simply decode the encodings in the input stream to an event that can be understood by the NFA. We call this decoder as the *event-based decoder*. The event-based decoder is used if we need to decode every opcode in the input, e.g., when printing an XML document or fragment.

However, event-based decoder combined with NFA is not efficient in query processing. With the event-based decoder, it is up to the NFA to decide whether the input event is a match to its current states. If so the NFA triggers a state transition or output of results, otherwise it just discards the event. Since there could be a large portion of events to be discarded by the NFA, the event-based decoder wastes a lot of computing resources on generating irrelevant events. In a nutshell, there are two major drawbacks of using event-based decoder for query processing:

1. The event-based decoder needs to scan the whole input stream and generate events for all encodings, even though they are not useful to the NFA. This causes significant overheads in I/O and CPU time in scanning and constructing irrelevant events.
2. For schema-based documents, the schema information is embedded in the input stream. The event-based decoder is unable to exploit them to do efficient skipping since the NFA is asking for all events.

4.3 Search-based Decoder

We propose a *search-based decoder* to tackle the two problems mentioned above by pushing down the constraint checking to the decoder module. The gist of the search-based decoder is similar to the predicate-pushdown technique that is widely adopted in the relational query processing. With the search-based decoder, the Binary XML streaming processing architecture remains the same but the API from the NFA to the decoder is changed to the italic text inside the dashed call-out in Figure 2. Rather than pulling the next event in sequence, the NFA first prepares a set of *search terms* based on its current states, and then it calls the decoder for the first event that matches one of these search terms. The search-based decoder will return an event that either matches one of the search terms or an `END_ELEM` or `END_DOCUMENT` event that indicates a mismatch.

Algorithm 1 NFA combined with Search-based Decoder

```
NEXT-MATCH
1  while TRUE
2      do terms ← PREPARE-TERMS(currentStates);
3          evt ← SEARCH-EVENT(terms);
4          if evt = START_DOCUMENT
5              then PROCESS-START-DOC(evt);
6                   update current states;
7          elseif evt = START_ELEM | START_ATTRIB
8              then PROCESS-START-ELEMENT(evt);
9                   update current states;
10                  return TRUE;
11         elseif evt = END_ELEM
12             then PROCESS-END-ELEMENT(evt);
13                  update current states;
14         elseif evt = END_DOCUMENT
15             then return FALSE;
16         elseif evt = CHARACTER | PCDATA | CDATA
17             then PROCESS-CHARS(evt);
```

The pseudo-code of the matching process is illustrated in Algorithm 1, which is very straightforward since most of the heavy-lifting work has been pushed-down to the search-based decoder SEARCH-EVENT(*terms*). One caveat is that the function PREPARE-TERMS(*currentStates*), if not designed carefully, could be very expensive since it is called every time before we call the search-based decoder. We will introduce the optimization techniques for PREPARE-TERMS after introducing the data structure of search terms.

A search term consists of a keyword (QNameID, kidNum, or wildcard *) indicating the element or attribute name, a level the keyword should appear relative to the current context node, and an optional callback function that is used to evaluate the predicate on the values. In the current implementation, the level could be 0, 1 or * (any), nodes at which represent the following siblings, children or descendants of the current context node, respectively. Theoretically, levels could be negative, which indicates the following ancestors of the context node, and a bit could indicate that the search is in reverse document order for backward axes. In this paper, we only consider forward axes for simplicity.

The search terms can be constructed at NFA compilation time. Given a state in the NFA, there will be a corresponding keyword for each out-edge, whose label is mapped to a QNameID or kidNum. If the state has no ϵ-edge, all keywords are at level 1, otherwise all keywords are at level *. Level 0 is reserved for preceding-/following-sibling axes, which is common in DOM API.

Given a set of current states N, a naïve implementation of the function PREPARE-TERMS would just enumerate every state $n \in N$ and merge the search terms associated with n in a set data structure S. The set S should be organized in a way to support efficient searching by the function SEARCH-EVENT. The search-based decoder traverse the tree structure by keeping the relative level and decode the opcode. It then needs to search inside S based on its level and keyword after decoding any opcode, it is desirable to sort the terms by their $(level, keyword)$, so that we could use more efficient binary search. To avoid expensive sorting for each call of PREPARE-TERMS, we cache the sorted terms in a hash table keyed by the set of current set of states. Since the number of states are usually very small, we can use a small bit vector to represent the set of the states as the key to the hash function. For exceptionally large NFA, we fall back to the naïve algorithm (sorting) for merging search terms.

With the knowledge of what it is looking for, the search-based decoder may exploit the knowledge of specific storage metadata to quickly skipping data that do not match with the search terms. For example, consider an NFA that is constructed from two simple path expressions /A/B and /A/C[.=‘C1’] that may be normalized from /A[C=‘C1’]/B. At the beginning, the NFA is expecting an element node whose tag name is A. Any element whose name does not match the tag name constraint is not of interest to the NFA. Therefore, the NFA can call the Decoder to return the next event which is a child element of the root and whose tag name is ‘A’. The Decoder, who maintains the schema and structural information, can quickly and safely skip the child elements of the root and whose tag name is not A. For example, if the Decoder knows that, based on the schema, element A is the tenth element of the root node, then it can directly jump to the right position, saving I/O for reading all subtrees of the first nine elements and saving CPU for constructing the events. Once the NFA got the event containing element A, it is expecting either an element B or C. Therefore, the NFA calls the Decoder to return an event whose a child element of A and whose tag name is in the set B, C, and the value of C is ‘C1’. If an event returned by Decoder satisfies either condition (say C), the NFA enters another state that requests the next sibling satisfying the rest of the conditions (i.e., elements whose tag name is B).

Based on the above description, one of the requirements of the search-based decoder is to be able to skipping a specific subtree efficiently. Since the Binary XML serializes a subtree by its document order, a naïve way is to decode every opcode while maintaining its level, which is incremented on START_ELEM and decremented on END_ELEM. Since we do not need to know the actual data in order to get the type of the opcode and its length, we can simply decode the metadata of each opcode and jump to specific offset based on the opcode’s length. Although this naïve algorithm is simple, it is effective since we do not need to read the actual node data, which usually consumes the majority of I/O bandwidth. In fact, this is the algorithm we use if the input data are transferred over the network. However, if the Binary XML is stored in database column, skipping subtrees could be much efficient if we pay a small overhead.

We propose a **per-document summary** to expedite navigation in the tree. It can be treated as a lightweight navigational index that is optional to build on top of the Binary XML storage. The idea is very straightforward: we maintain an array of start and end offsets (s_i, e_i) for "large" subtrees sorted by s_i. A visualization of a navigational summary is depicted in Figure 2, where the dashed lines on the Binary XML Input Stream indicates the start s_i and end e_i offsets of elements el_i. Note that el_0 is the ancestor of el_1–el_4 and not all descendants of el_0 have an entry in the summary. Likewise, the summary also maintains the start and end offsets of arrays of elements. Based on the array mode definition, if the first element does not match the search term, we can safely skip all elements till the end of the array. This makes it efficient to skip large number of small elements. To make the summary small and efficient for lookup, we only include subtrees or arrays whose sizes are larger than a certain threshold. This is effective in practice since smaller subtrees are usually completely contained in one page and are already read into cache by I/O prefetching. Therefore the threshold should be greater than the database page size.

4.4 Structural/Schema-aware Optimizations

As introduced earlier, an XPath consists of several location steps. We will refer a node in the XPath location step as the XPath-node. The presence of a schema can speed up the XPath searches in some cases.

The first schema-based optimization is called *instantiation*, where we replace a more generic path expression with a specialized one. One example is that XPaths with wildcard ('*') and descendant axes ('//') can be expanded to simple location steps consists of /-axes only. While these rewrites are particular amenable to lookups in XML index whose key are simple path IDs, the latter is also desirable for streaming evaluation using search-based decoder. The reason is that the search-based decoder can skip unmatched subtrees with simple paths with /-axes only while it has to search the whole subtree with //-axes. Schemas provide vital information that is necessary to expand such XPaths.

The second schema-based optimization is called *early-out*, where the search-based decoder can terminate searching in the child list before exhaustively exploring it. The idea is that for each search term (set of keywords) we can also pass a *term-list* (termination list). Once any keyword in the term-list is found, the search-based decoder could safely skip any children after that till the end of the context node. The term-list can be generated by examining the *compositor* of a complexType from XML Schema [1] during query compilation. For example, if the XML schema has a sequence, the node that occurs after the corresponding XPath node is a candidate for the term-list. We will add all such nodes into the term-list until we encounter a node with minOccurs $>=$ 1. For instance, consider the following sequence definition under path `/a/b`.

```
<sequence>
  <element name="c" minOccurs="2" maxOccurs="2"/>
  <element name="d" minOccurs="3" maxOccurs="3"/>
  <sequence>
    <element name="e" minOccurs="2" maxOccurs="5"/>
    <element name="f" minOccurs="1" maxOccurs="2"/>
  </sequence>
</sequence>
```

To process the XPath `/a/b/c`, we will add `d` to the term-list for keyword `c`. If the minOccurs of `d` is 0, we will add both `d` and `e` to the term-list.

The third optimization is called *singleton identification*, where we annotate XPaths that will return a single node as result. This annotation is particular useful when XQuery is rewritten to XPathTable and singleton paths can be added as a column in an existing XPathTable rather than generating a new XPathTable and a subquery. Examples of singletons XPaths are those ended with an attribute, or elements whose minOccurs and maxOccurs equals to 0 or 1.

4.5 Caching in XPathTable Row Source

Some use cases need two or more nested XQuery FLWRs both of which are driven by XPaths on the base table's column. For example consider the following XQuery snippet. Here $auction refers to the base table column.

```
for $p in $auction/site/people/person let $a :=
  for $t in $auction/site/closed_auctions/
             closed_auction
  where $t/buyer/@person = $p/@id return $t
return <item person="{$p/name/text()}">{count($a)}
       </item>
```

This XQuery will need two XPathTables one for the outer FLWOR and the other for the inner FLWOR. The inputs to both of these are the document stored in the database (referred by `$auction`). For each row produced by the outer XPathTable, the inner XPathTable makes a pass over the document and generates rows. For a given document in database, this is repeated as many times as the number of rows produced by the outer XPathTable. The rows produced by the inner XPathTable prior to predicate evaluation will be the same for all the repetitions since they only depend on the document. This is analogous to the nested-loop joins between two relational tables.

As an optimization we cache the column values of the inner XPathTable after the first execution for a given document. For subsequent executions we apply the filters and return the results from the cache without evaluating the inner XPaths again. The cache will be flushed when we move on to the next document. In essence, this evaluation strategy is similar to the blocked nested-loop join of two tables.

The cache uses relatively small amount of memory since it caches only Binary XML locators or SQL primitive types. The memory usage of cache can be controlled by user using database tuning parameters. When the memory consumption reaches the threshold, part of the cache will be automatically written to temporary tables in the database. Therefore, this scheme will not exhaust memory but provide a significant performance improvement for such queries containing nested FLWORs.

5. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of Binary XML storage and query processing. In particular, we look at the compression ratio, query performance, and update performance in both schema-based and non-schema-based settings. All experiments are conducted on a PC running Linux kernel 2.6.9 with 3 GHz Intel Xeon CPU and 6 GB of main memory. We configured the Oracle DBMS server with 512 MB of DB cache size and 8 KB of block size for all cases.

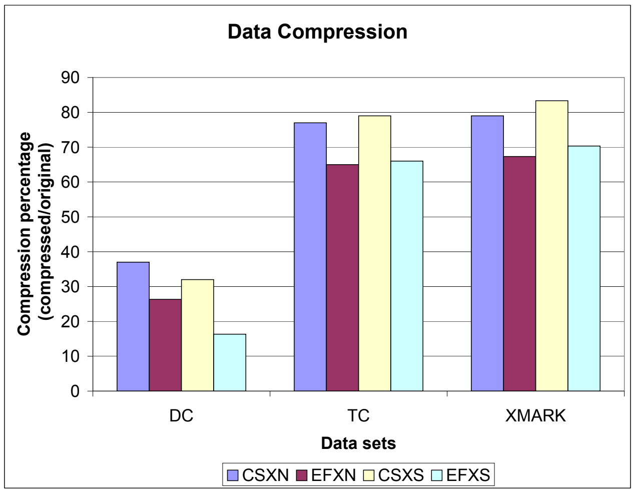

Figure 3: Compression percentages (smaller is better)

5.1 Compressions

In this experiment, we measure the compression ratios that schema-based and non-schema-based Binary XML storage achieved over the original data. We tested different data sets various from 1 KB to 100 MB with different schema characteristics: the DC is highly structured data-centric schema; XMark is also a data-centric schema but with more complex structures; and TC is a text-centric schema. Figure 3 shows the compression ratios (c/x where c is the compressed size and x is the original document size). As we seen from the figure, we achieve good compression ratios in both schema-based (CSXS) and non-schemabased (CSXN) but lags behind EFX (EFXN and EFXS) in all cases, respectively. The reasons are EFX are mainly concentrate on compression ratio but sacrificing other capabilities. For example, EFX compresses a document by scanning the document from beginning and utilizes the information that it has seen to compress later elements. Based on this nature, it is not clear whether it can provide a locator mechanism to allow the query processor jumping to the middle of the input and start decoding from there. This capability is critical in supporting indexes.

Another tradeoff by EFX is the schema evolution capability. By encoding elements closely coupled with schema, it is hard to implement efficient schema evolution technology. Oracle Binary XML is designed as a general purpose format to database servers to handle all these requirements.

5.2 Update Performance

Binary XML storage's DML performance is much better than CLOB and OR since the fact that the former does not need to render the document as a DOM before update or shred the document into pieces before insert. So we only evaluate the more interesting piece-wise update performance here. To evaluate the performance of piece-wise update, we tested different operations on the SecureFiles (piece-wise update) and BasicFile (basic update) BLOB with different data sizes, ranging from 1 KB to 2 MB. The tested operations are shown as the x-axis in Figure 4. The y-axis is the average

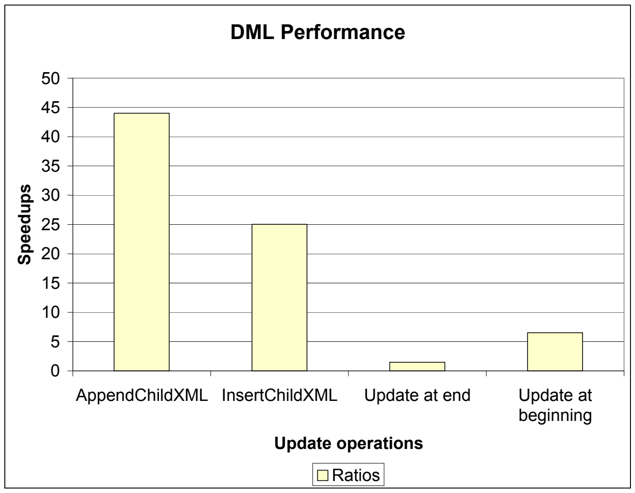

Figure 4: Piece-wise update speedups

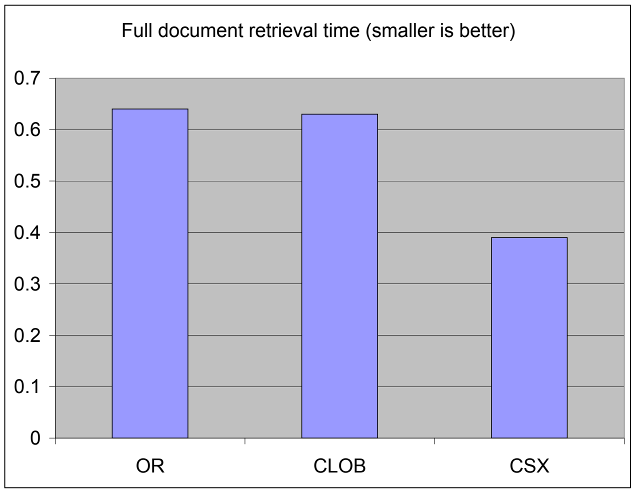

Figure 5: Full Document Retrieval Timings

speedups using piece-wise update.

From the figure, we can see that insertions to the beginning of the LOB significantly improves from the basic update, as a result of the chaining technique [21].

5.3 Full Document Retrieval

It is a common operations to retrieve the full XML document from an XML repository. In Oracle 11g, XML repository consists of an XML typed table and metadata. Figure 5 shows the performance of the full document retrieval through the repository interface with different storage formats. The y-axis is the timing (in seconds) to retrieve 10 MB XMark document from a repository. The CSX storage incurs approximately 40% less time than CLOB and OR.

5.4 Query Performance

To experimentally study the query performance proposed in Section 4, we compare the performance of 19[2] XMark

[2]Q14 is excluded in all tests since it exceeds an Oracle lim-

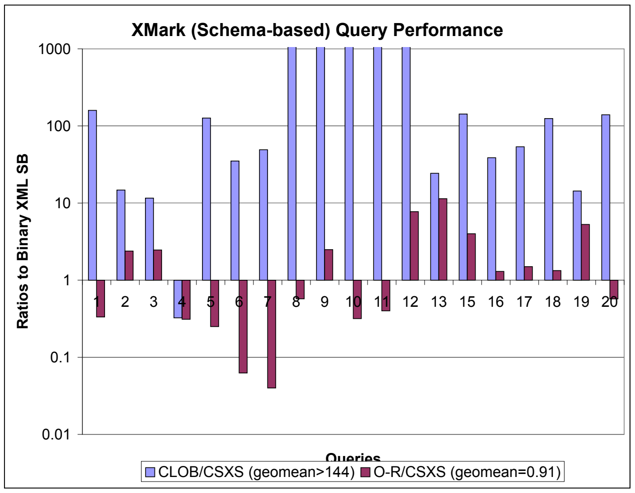

Figure 6: XMark Query Performance for Schema-based Storages

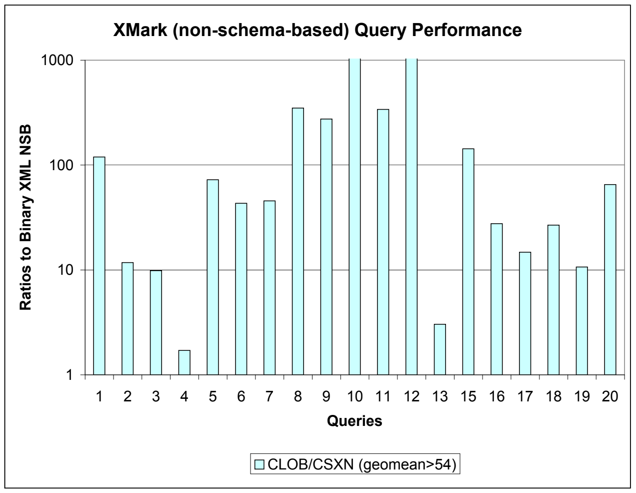

Figure 7: XMark Query Performance for Non-schema-based Storages

queries [26] using different storage options. XMark data sets are data-centric and its schema is reasonably complex. Note that although all storage options support XML index to speedup their performance, the experiments were conducted without any indexes.

Figure 6 shows the performance numbers of 19 XMark queries for schema-based (SB) storages—CLOB, OR, and schema-based Binary XML (CSXS). The x-axis is the queries, and the y-axis is the ratios, in logarithmic scale, of the numbers of CLOB, OR to the numbers of CSXS. A ratio greater than 1 indicates the speedups the Binary XML schema-based storage over the other storage formats.

As we can see from the figure, Binary XML schema-based storage performs very good comparing to other storage options in all queries except Q4. The reason Q4 is not performing well is that there is an XQuery order operator (<<) which is not currently supported in streaming evaluation. CLOB storage almost consistently gives the worst performance as expected. Q8–Q12 even did not finish after 3 hours. The reasons are twofold: (1) the whole document need to be parsed before querying, and (2) the query processing engine is based on DOM, which is very expensive.

The OR storage performs better than Binary XML schema-bases storage in 9 out of 19 queries. These queries usually have simple path expressions that can be mapped nicely to a column in a table and few XML elements are constructed as results. This means most cost resides in the OR query processing rather than fragment extraction or document reconstruction. On the other hand, if the queries cannot be rewritten to a columns or many results are returned, streaming evaluation using Binary XML storage beat OR.

In summary if the documents have schemas, the geometric means of all the query performance ratios (CLOB and OR vs. Binary XML schema-based) are 144.3 (34.6 if not including Q8–Q12) and 0.91, respectively. Here OR gives the best performance. Binary XML schema-bases storage lags by only 9% in query performance but gives users the schema flexibility and DML performance.

itation of 4 KB of text node.

If the documents have no schema, CLOB and Binary XML non-schema-based storages are the two options. Figure 7 shows Binary XML outperforms CLOB in all queries with factor of 54 (or 23 if not including Q8–Q12).

5.4.1 *Comparing with Partitioning-based Storage*

We also conducted performance comparisons to an open source state-of-the-art tree-partitioning-based storage structure. Since that implementation does not support the full XQuery and only support simple path expressions without predicates. We tested path expressions with //-axes, wildcards and simple paths that consists of /-axes only. All queries are wrapped by a `count()` function so that the results construction cost is eliminated. We obtain about 2x speedup consistently in all paths even it is a simple short path. The reasons seems to be twofold: (1) with //-axes and long simple paths with /-axes only, the whole or most part of the XML tree need to be traversed. In a tree-partitioning based storage, this needs to follow the pointers to access another subtree stored in another disk page, which could result in random I/O, whereas we benefit from sequential I/O and caching from the SecureFiles storage; and (2) with the simple short path, the tree-partitioning approach can skip subtrees that does not match a certain step and jump between siblings following sibling pointers. With the help of navigational summary, Binary XML can also jump to the end of a large subtree. Furthermore, for small subtrees in array mode, Binary XML streaming evaluation can jump to the end of an *array of elements*, if they are not matches. With the schema-aware optimizations, we can terminate the search early even if we have not exhausted a child list.

6. FUTURE WORK AND CONCLUSION

We would like to extend streaming evaluation to broader scope including the ordering operation and backward axes. We also would like to develop cost models and optimization techniques for these streaming operations. We will continue working on improving the query and DML performance for the Binary XML storage format.

In this paper, we present the Oracle Binary XML format that consists of Oracle's native XML storage and processing. Binary XML is designed to satisfy a wide range of requirements that are common for real-world business applications. In summary, Binary XML provides good balances in schema flexibility, mid-tier integration, query processing, update, and space consumptions.

7. REFERENCES

[1] XML Schema. Available at http://www.w3.org/XML/Schema, May 2001.

[2] AgileDelta. Theory, Benefits and Requirements for Efficient Encoding of XML Documents. `http://www.agiledelta.com/w3c_binary_xml_proposal.html`.

[3] K. S. Beyer, R. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. M. Lohman, B. Lyle, F. Ozcan, H. Pirahesh, N. Seemann, T. C. Truong, B. V. der Linden, B. Vickery, and C. Zhang. System RX: One Part Relational, One Part XML. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 347–358, 2005.

[4] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 479–490, 2006.

[5] M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Full-fledged Algebraic XPath Processing in Natix. In *Proc. 21st Int. Conf. on Data Engineering*, pages 705–716, 2005.

[6] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up Persistent Applications. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 383–394, 1994.

[7] S. Chandrasekar, S. Idicula, T. Yu, and N. Agarwal. Oracle Binary XML Format. Available at `http://www.oracle.com/technology/tech/xml/xmldb/Current/oracle_binaryxml_rfc.pdf`.

[8] J. Cowan and R. Tobin. XML Information Set. Available at http://www.w3.org/TR/xml-infoset/.

[9] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Trans. Database Sys.*, 28(4), December 2003.

[10] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 Formal Semantics. Available at `http://www.w3.org/TR/query-semantics/`.

[11] T. Fiebig, S. Helmer, C.-C. Kanne, J. Mildenberger, G. Moerkotte, R. Schiele, and T. Westmann. Anatomy of a Native XML Base Management System. *The VLDB Journal*, 11(4):292–314, 2002.

[12] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, J. Carey, and A. Sundararajan. The BEA streaming XQuery processor. *The VLDB Journal*, 13(3):294–315, 2004.

[13] J. E. Funderburk, G. Kiernan, J. Shanmugasundaram, E. Shekita, and C. Wei. XTABLES: Bridging relational technology and XML. *IBM Systems Journal*, 41(4):616–641, 2002.

[14] V. Josifovski, M. Fontoura, and A. Barta. Querying XML Streams. *The VLDB Journal*, 14(2):197–210, 2005.

[15] C.-C. Kanne and G. Moerkotte. Efficient Storage of XML Data. In *Proc. 16th Int. Conf. on Data Engineering*, pages 198–209, 2000.

[16] C. Koch. Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata -based Approach. In *Proc. 29th Int. Conf. on Very Large Data Bases*, pages 249–260, 2003.

[17] M. Krishnaprasad, Z. H. Liu, A. Manikutty, J. W. Warner, and V. Arora. Towards an industrial strength SQL/XML Infrastructure. In *Proc. 21st Int. Conf. on Data Engineering*, pages 991–1000, 2005.

[18] Z. H. Liu, S. Chandrasekar, T. Baby, and H. J. Chang. Towards a Physical XML independent XQuery/SQL/XML Engine. In *Proc. 34th Int. Conf. on Very Large Data Bases*, pages 1356–1367, 2008.

[19] D. Megginson. The Simple API for XML. Available at http://www.saxproject.org/.

[20] Microsoft. White Paper: What's New for XML in SQL Server 2008. White Paper.

[21] N. Mukherjee, B. Aleti, A. Ganesh, K. Kunchithapadam, S. Lynn, S. Muthulingam, K. Shergill, S. Wang, and W. Zhang. Oracle SecureFiles System. In *Proc. 34th Int. Conf. on Very Large Data Bases*, pages 1301–1312, 2008.

[22] M. Nicola and B. V. der Linden. Native XML Support in DB2 Universal Database. In *Proc. 31st Int. Conf. on Very Large Data Bases*, pages 1164–1174, 2005.

[23] S. Pal, I. Cseri, O. Seeliger, M. Rys, G. Schaller, W. Yu, D. Tomic, A. Baras, B. Berg, D. Churin, and E. Kogan. XQuery Implementation in a Relational Database System. In *Proc. 31st Int. Conf. on Very Large Data Bases*, pages 1175–1186, 2005.

[24] S. Pal, I. Cseri, O. Seeliger, G. Schaller, L. Giakoumakis, and V. Zolotov. Indexing XML Data Stored in a Relational Database. In *Proc. 30th Int. Conf. on Very Large Data Bases*, pages 1134–1145, 2004.

[25] M. Rys. XML and relational database management systems: inside Microsoft SQL Server 2005. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 958–962, 2005.

[26] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, 2001.

[27] R. K. Wong, F. Lam, and W. M. Shui. Querying and Maintaining a Compact XML Storage. In *Proc. 16th Int. World Wide Web Conference*, pages 1073–1082, 2007.

[28] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 425–436, 2001.

[29] N. Zhang, V. Kacholia, and M. T. Özsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In *Proc. 20th Int. Conf. on Data Engineering*, pages 54 – 65, 2004.