XPEDIA: XML Processing for Data Integration

Manish Bhide, Manoj K Agarwal
IBM India Research Lab
India
{abmanish, manojkag}@in.ibm.com

Amir Bar-Or, Sriram Padmanabhan
IBM Software Group, USA
{baroram,srp}@us.ibm.com

Srinivas K. Mittapalli, Girish Venkatachaliah
IBM Software Group
India
{smittapa,girish}@in.ibm.com

ABSTRACT

Data Integration engines increasingly need to provide sophisticated processing options for XML data. In the past, it was adequate for these engines to support basic shredding and XML generation capabilities. However, with the steady growth of XML in applications and databases, integration platforms need to provide more direct operations on XML as well as improve the scalability and efficiency of these operations. In this paper, we describe a robust and comprehensive framework for performing Extract-Transform-Load (ETL) of XML. This includes (i) full computational model and engine capabilities to perform these operations in an ETL flow, (ii) an approach to pushing down XML operations into a database engine capable of supporting XML processing, and (iii) methods to apply partitioning techniques to provide scalable, parallel processing for large XML documents. We describe experimental results showing the effectiveness of these techniques.

1. INTRODUCTION

XML was introduced in the mid-1990's as a simple and extensible data mark-up language. It gained immediate foothold as a data interchange format. Over time, gaining from a wide variety of research, XML has become a valuable data format within and across enterprises for representing data in persistent and transient applications. There is an illustrious body of research in XML processing dealing with parsing, transformation, database processing, indexing, and search. Database and application vendors have made use of this research resulting in a support for XML as a first class data type in databases such as DB2 [9], Oracle [10], and SQL Server [11] as well as in many application languages such as Java, C++, and scripting languages. Hence, it is only natural that data integration engines should provide efficient and scalable techniques for XML processing.

Data Integration engines such as IBM's Information Server, Informatica's PowerCenter, etc., provide the capabilities to Extract-Transform-Load (ETL) from various data sources into various data targets. For XML, these ETL engines currently provide rudimentary capabilities to perform XPath based transformations into tuple formats or vice-versa. However, with the steady progress of XML adoption, there is a need for data integration engines to provide fast, scalable "next-generation" XML handling capabilities. While there is a wealth of literature in various areas of XML processing, surprisingly, there has been limited work on the topic of XML data integration. The initial work in this area has focused on algorithms to shred XML into a natural relational schema. The body of work on XML Stylesheet Transformation (XSLT) can be considered as peripherally related to our topic [1]. There is also quite a bit of work on XML query processing which we consider to provide some relevant foundational basis for our topic [8, 9]. However, there are still a lot of open issues that need to be addressed to enable "next generation" XML data integration. In this paper we present the XPEDIA (*XML ProcEssing for Data IntegrAtion*) system which addresses some of these issues.

VLDB '09, August 24-28, 2009, Lyon, France.

1.1 XPEDIA System

Data integration over relational data is a well studied topic. However, in an XML world, data integration is radically different due to the hierarchical nature of the data. Hence techniques which have been developed for relational world cannot be directly applied to XML data. The XPEDIA system is one of the first systems to incorporate techniques for efficiently supporting XML data integration. We now outline some of the key features and challenges addressed by XPEDIA.

Computational Model: The computational model used to represent ETL processes over relational data assumes data in the form of rows consisting of multiple columns. Such computational models represent each XML document as a single row consisting of a single column. Such a simplified representation of XML data is a major handicap in supporting complex XML operations. Hence there is a need for a technique to handle complex data transformation flows, while maintaining the easy of specification inherent in the relational computational model.

First Class Data Type: Most of the data integration engines available in the market today, treat XML as a CLOB (string of characters). However, the key for efficient handling of XML data is to treat it as a first class data object during transformation. Such a representation enables XPEDIA to support operations such as equi-hierarchical-join, xml-aggregate, etc., which are specialized operators for dealing with XML data (details in Section 3). These operators allow users to easily define intricate XML transformation flows, which hitherto were not possible.

ELT Support: If the source (or target in some cases) of an ETL flow is a database that supports XML processing, XPEDIA

applies rewriting techniques to transform parts of the ETL job flow into SQL/XML queries in order to push some significant processing into the database. This is called ELT (Extract, Load, Transform) and is a valuable technique to gain efficiency and performance by leveraging the database's capabilities.

Scalability: The size of a row/tuple in relational data is seldom larger than a few Kb's. However, we have observed from several customers that XML inputs for data integration tend to be large, aggregated inputs comprised of many smaller objects. Thus we need specialized techniques for handling large (> 2-3 GB) XML documents. XPEDIA uses a novel single-pass partitioning and parallel processing technique for XML objects. The hierarchical nature of the XML data presents some unique challenges for executing a single transformation job comprising of a series of operators in a parallel execution environment. In this paper we present various approaches supported by XPEDIA for partitioning input XML objects in order to perform the rest of the processing using parallel streams before combining the results. We also provide experimental results for all of these techniques to show the scalability and performance that can be obtained.

1.2 Contributions

The research contributions of our work can be summarized as follows:

- We propose a computational model for ETL applications on XML data. The new model is specifically tailored to handle hierarchical XML data and treats XML as a first class data type.
- XPEDIA is the first system that supports query rewriting techniques to convert an ETL flow into an ELT flow over XML data.
- We present novel parallel processing techniques for handling large XML documents. We believe that our single pass techniques are the first attempt at supporting efficient parallelism in ETL flows over XML data.
- We present experimental results that validate both our techniques and our results show that we achieve significant improvement in performance for a typical ETL flow.

Paper Organization: Section 2 introduces our computational model and also gives an overview of the various operators supported by the model. We present a sample ETL flow using the operators of our computation model in Section 3. The techniques for supporting ELT over XML data are summarized in Section 4. Section 5 presents the parallel processing techniques used by XPEDIA for handling large XML documents. The experimental validation of XPEDIA is outlined in Section 6. The related work is summarized in Section 7 and Section 8 concludes the paper.

2. COMPUTATION MODEL

A computational model is used to express ETL processes that move and transform data from sources to targets. Many ETL tools use the dataflow computational model to describe these processes. In the dataflow model, processes are expressed via a directed flow graph where the vertices of the graph are operators and the edges represent the flow of data. The operators in a dataflow model can perform one or more of the following operations: (i) read data from sources (ii) write data to targets and (iii) perform transformations on input data to produce new output data. The algorithm of an operator is defined by the operator type. For example, a join operator uses the join algorithm for joining the input data.

Existing dataflow based ETL engines assume relational data model for the data that flows between the operators, i.e., records that flow between two operators consist of "rows" having multiple columns. In order to handle XML data these engines consider each XML document to be a single row with one (XML) column. However, as is obvious, such an over-simplified representation is a major handicap in supporting complex operations over XML data. Hence, XPEDIA uses a new computational model which extends the relational dataflow model to support hierarchical data. Such an extension, as we explain next, requires a major shift in the representation methodology.

XPEDIA uses a dataflow model consisting of operators and edges. However, the key difference from the existing dataflow models is that the data that flows between two operators is an ordered list of XML documents that comply with a single XML schema element definition [2]. Notice that each document could itself be multi-dimensional or in XML terminology the document could have multiple repeated elements with *maxOccurs > 1* or *maxOccurs="unbounded"*. For example, consider an XML document with root node "*High_Value_Customers*" which in turn has 100 "*Customer*" child nodes. In this document if we map "*Customer*" node to a row in the relational world then the XML document will map to a table consisting of 100 rows. In such a setup the XML document would represent two dimensional data. In order to capture this multi-dimensional nature of the XML data, each XML document in our computational model consists of multiple "Vectors" – one for each repeating element type. In the "*High_Value_Customers*" example, the document will consist of one Vector (of size 100) as it has one repeating element of "*Customer*" type.

The concept of vector also makes a difference in the way data is handled by an operator. In our data flow model, operators (except the source operator which simply reads data from the source) iterate through the list of objects (XML documents) in their input data. As each object (document) can consist of multiple sub-vectors, the operators can also iterate through a sub-vector of the input data. The iterated vector is defined as the "scope" vector of the operator. For each scope instance, the operator processes the input data that is contained in the scope instance and produces a result that is also contained in the scope instance. For example, in the "*High_Value_Customers*" example the scope of an operator will be "*/High_Value_Customers/Customer*". An operator will treat each "*Customer*" sub-tree below the root independently of the other "*Customer*" sub-trees. Hence operators are not allowed to maintain state between instances of the scope vector. Thus each scope instance (i.e., each "*Customer*" sub-tree) can be thought of as being similar to a row in the relational world. The concept of vector and scope help us to support a rich set of operators over XML data and also help us to support parallelization of the ETL flow (details in Section 5).

Another key difference from relational data flow systems is that our computational model only supports linear sub-graphs. Relational dataflow systems, on the other hand, typically allow parts of the data to be separated and processed in different sub-graphs (of the ETL flow). In an XML world, this would map to

splitting the contents of a vector (of a single XML document) to different sub-graphs and then merging them into one or more vectors. As this can get quite complicated, our computational model only allows linear graphs thereby simplifying the job design process and increasing its usability. In order to compensate for the restriction of the model into a linear graph, each operator in our model produces a result that also contains its initial input. We then employ runtime optimizations, graph rewrites and dead field elimination to optimize the XPEDIA ETL flows by removing false dependencies and eliminating the transfer of data that is no longer needed downstream.

Thus the computational model of XPEDIA incorporates a radical shift in the representation methodology by including support for hierarchical XML data due to the use of features such as Scope, Vector and linear sub-graphs. Using these concepts, Figure 1 describes an operator's algorithm in our computational model in terms of an XQuery update statement.

```
Let $input be the set of input documents the
operator is processing
Let $scope_path be the operator scope path
for $currentDoc in collection($input)
return
 for $scope in $currentDoc//$scope_path
  return (
   insert node<Result>{op_alg($scope)}</Result>
                  as last into $scope)
```

Figure 1: Operator Algorithm

op_alg in the above figure stands for the operator algorithm that accesses the data contained in *$scope* and produces a result only using this data. We explain next a sample set of specialized operators supported by our model for handling XML data.

2.1 XML Operators

Filter Operator: The filter operator can filter one of the vectors contained within the scope instance and produce a new vector which will contain only instances that passes the filter predicate. The following SQL/XML clause illustrates the Filter operator algorithm:

```
Let $scope be the scope vector path
Let $child_vector be the path to the aggregated
vector
Select  * From ( $scope//$ child_vector))
Where  $scope//$child_vector/$key1 > 5
```

Project Operator: The project operator iterates over a single vector in its input and produces a new vector that is based on a set of select expressions. The set of select expressions allows the user to modify the input by removing an element or a sub tree, renaming an element, or enriching the document with a new attribute by computing a scalar expression. The following SQL/XML clause illustrates the Project operator algorithm:

```
Let $scope be the scope vector path
Let $child_vector be the path to the aggregated
vector
Select  a, b,  (a || b) as ab
From ( $scope//$child_vector))
```

Aggregate Operator: The aggregate operator can produce statistics by aggregating one of the vectors contained in the scope instance. Similar to a database aggregation, the aggregate operator takes a set of aggregation functions and a group by clause which defines the key aggregation columns. The result of the aggregate operator is a new vector with a summary record for each unique key that is found in the input vector. Notice that the aggregation restarts for each scope item. The following SQL/XML clause illustrates the Aggregate operator algorithm:

```
Let $Scope be the scope vector path
Let $aggr_vector be the path to the aggregated
vector
Let $Key1, $Key2 be the paths to the key
aggregation columns
Let $cost be a path to an attribute of
aggregated vector

Select $Key1, $Key2, Avg($cost) as avg_cost,
count(*) cnt_rcd, …
From ( $scope//$aggr_vector))
Group by Key1, Key2
```

Equi-Hierarchical-Join Operator: The equi-hierarchical-join operator performs an equality based join between two vectors that are contained within the scope instance. Similar to a relational equi-join, the join operator takes a set of equality predicates between attributes from one vector and attributes of the second vector. The result of the join consists of two nested vectors, where each instance of the input left vector contains all the matching instances of the right input vector. The following SQL/XML clause illustrates the Equi-Join operator algorithm:

```
Let $Scope be the scope vector path
Let $left_vector, $right_vector be the paths to
the joined vectors
Let $left_key1, $left_key2 be the paths to the
left vector join columns
Let $right_key1, $right_key2 be the paths to
the right vector join columns

Select ($scope//$rightVector.*) insert into
$scope//$leftVector.*
From ( $scope//$aggr_vector))
Where  $scope//$left_vector/$left_key1 =
$scope//$right_vector/$right_key1 AND
       $scope//$left_vector/$left_key2 =
$scope//$right_vector/$right_key2
```

Read Table Operator: The Read Table operator reads all the rows of a single table and outputs either relational tuples or an XML document. This operator takes as a parameter a SQL or SQL/XML query which executes on the input table to generate the required output.

Write Table Operator: The Write Table operator is used for writing relational or XML data to a table. For each instance of the scope vector present in the input, the operator creates a new record in the output table.

OutputStage Operator: The OutputStage operator transforms a relational input into an XML Document. This operator has been included due to legacy support issues. The operator takes as input a mapping from each relational attribute present in the input to an XPath in the output XML document (which is to be created). One of the XPaths is designated as the repetition path. This is used to decide the structure of the XML document. An example of the function of OutputStage is given in Figure 2. In this example, the mapping from relational attribute to XPath is as follows: (1)

Department → */Company/Country/Dept* (2) Project → */Company/Country/Employee/PName* and (3) Emp ID → */Company/Country/Employee/EInfo/EmpID*. The *EmpID* node is designated as the repetition output path. The OutputStage operator makes use of the repetition path to decide the structure of the output XML document. The repetition path works by comparing values between input rows. The following rules apply:

Rule 1: A change in an input column value triggers the closing of at least one element and the opening of at least one element.

Rule 2: When a single input column value changes, the repetition path applies as follows: Every opened element is closed up to and including the first element that is part of the repetition path. E.g., Let the repetition path be: */w/x/y/z* and let the XPath of the affected column be: */w/x/y/a/b*. Then the y element is closed and new elements are opened, down to the last element of the XPath expression of the column for which the value has changed.

Rule 3: When more than one column changes values, elements are closed and opened, starting with closest to the root element.

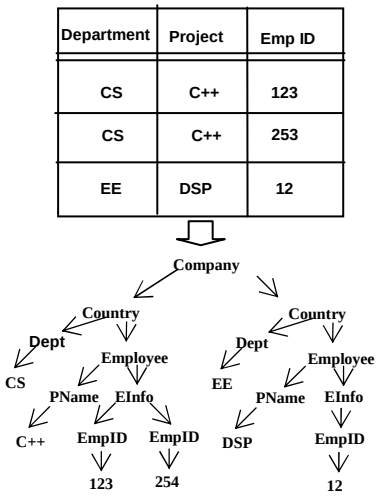

Figure 2: Functioning of OutputStage

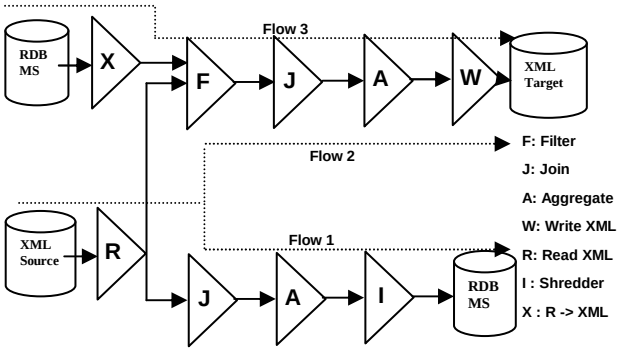

Figure 3: Typical ETL Flows

3. TYPICAL ETL SCENARIO WITH XML

Figure 3 shows three sample ETL flows. In these flows, there are two sources and targets, one of which is relational and the other is XML. Each flow consists of a series of operators which transform the input data. Flow 1 converts XML data to relational format, Flow 2 converts XML data to XML and flow 3 converts relational data to XML format.

In this section, we describe the Flow 1 in detail. The ETL process shown in Flow 1 reads a set of XML documents that are stored in a database, transforms the documents and eventually writes them to the target relational database. The source table contains a single XML column that holds the source document content. All source documents conform to the "*Company*" element which is described in Figure 4. We now enunciate each step of the flow in detail.

Step 1: The Read_XML_Table operator simply reads the XML documents from the database and outputs it to the next operator.

Output:

```
<element name="Company" type="Company' maxOccurs="unbounded" />
  <element name="Company" type="Company' />
   <complexType name="Company">
     <sequence>
       <element name="Country" type="tns:Country" maxOccurs="unbounded"/>
     </sequence>
  </complexType>
  <complexType name="Country">
     <sequence>
       <element name="CountryName"/>
       <element name="CountryCode"/>
       <element  name="Departments" type="tns:Department" maxOccurs="unbounded"/>
       <element  name="Employee" type="tns:Employee" maxOccurs="unbounded" />
     </sequence>
  </complexType>
  <complexType name="Department">
     <sequence>
       <element  name="DeptName" type="string" />
       <element  name="DeptCode" type="string" />
       <element name="ManagerID" type="string"/>
     </sequence>
  </complexType>
  <complexType name="Employee">
     <sequence>
       <element  name="DeptCode" type="string" />
       <element name="EmpName" type="string"/>
       <element name="EmpId" type="string"/>
       <element name="EmpLocation" type="string"/>
       <element name="Salary" type="double"/>
     </sequence>
  </complexType>
```

Figure 4: Output of Read_XML_Table operator

Output:

```
<element name="Company" type="Company' maxOccurs="unbounded" >
 <complexType >    <sequence>
    <element name="Country" maxOccurs="unbounded">
     <complexType>  <sequence>
       <element name="CountryName"/>
       <element name="CountryCode"/>
       <element  name="Departments" type="tns:Department" maxOccurs="unbounded"/>
       <element  name="Employee" type="tns:Employee" maxOccurs="unbounded" />
       <!--  THIS IS THE JOIN PRODUCT BEGINNING--!>
        <element  name="Dept2" maxOccurs="unbounded">
         <complexType>  <sequence>
           <element  name="DeptName" type="string" />
           <element  name="DeptCode" type="string" />
          <element name="ManagerID" type="string"/>
          <!--  EMPLOYEE IS NESTED WITHIN THE DEPTARTMENT--!>
             <element  name="Employee" type="tns:Employee" maxOccurs="unbounded" />
          <!--  JOIN PRODUCT ENDS HERE--!>
          … (definition closed)
```

Figure 5: Equi-Hierarchical Join operator

Step 2: The Equi-Hierarchical Join operator (Figure 5) has the following parameters: {scope: Company/Country, left vector : Department, right vector: Employee, join key : Department/DeptCode = Employee/DeptCode, result= Dept2//Emp2 }. For each "Country" sub-tree in the input XML document, the operator finds the set of employees working in each department (in that country) and creates a new element named

"Dept2" (for each department) which has the list of all employees working in that department.

Step 3: The Aggregation operator has the following parameters: {scope: Dept2, aggregated vector= Dept2/Employee, aggregate key: Dept2/DeptCode, aggregate function =sum(Dept2/Employee/salary) result= Dept2/totalSalary }. The operator finds the total salary of all the employees in a department and adds it to the XML document as "*totalSalary*".

Output:

```
<element name="Company" type="Company' maxOccurs="unbounded" >
 <complexType >    <sequence>
    <element name="Country" maxOccurs="unbounded">
     <complexType>  <sequence>
        <element name="CountryName"/>
        <element name="CountryCode"/>
        <element  name="Departments" type="tns:Department" maxOccurs="unbounded"/>
        <element  name="Employee" type="tns:Employee" maxOccurs="unbounded" />
         <!--  THIS IS THE JOIN PRODUCT BEGINNING--!>
         <element  name="Dept2" maxOccurs="unbounded">
          <complexType>  <sequence>
            <element  name="DeptName" type="string" />
             <element  name="DeptCode" type="string" />
            <element name="ManagerID" type="string"/>
            <!--  EMPLOYEE IS NESTED WITHIN THE DEPTARTMENT--!>
               <element  name="Employee" type="tns:Employee" maxOccurs="unbounded" />
            <!--  JOIN PRODUCT ENDS HERE--!>
            <!--  AGGREGATION RESULT INSERTED HERE--!>
            <element name="totalSalary" type="double"/>
  …  (definition closed)
```

Step 4: The final operator Shredder writes the totalSalary in the modified XML document to the relational database. The attributes of the operator are: {scope: Dept2vector, totalSalary=/Company/Country/Dept2/totalSalary}.

Notice that in each step of the above ETL flow, for each scope instance (i.e., each "*country*" node) the output includes the entire input data (in that "*country*" node) along with the result of the transformation applied to the data within the scope (i.e., the "*country*" sub-tree). XPEDIA also supports other optimizations such as false-dependency elimination via linear graph to acyclic graph rewrite, dead field elimination via usage analysis and streaming large document with a bracket model. Details are omitted due to lack of space. We present next, the ELT feature of XPEDIA.

4. ELT OPTIMIZATION IN XPEDIA

Recent times have seen a proliferation of databases with native XML support. Databases such as DB2 9, Oracle 11g and SQL Server 2005 store XML documents in native format and have inbuilt XQuery and SQL/XML query engines. Another important feature of these databases is that they allow users to define indexes on XML documents. If part of the processing involved in the ETL job is executed inside the database engine, then it can make use of the XML indexes to significantly reduce job execution time. Thus the goal of ELT is to delegate some part of the ETL job to the database engine. This is accomplished by generating a SQL/XML query which does the same job as one or more steps in the ETL flow. For example, the flow 1 from XML data source to RDBMS in Figure 3 can be represented by a single SQL/XML query. Hence the modified job would have a single Read Table operator which will directly output the relational tuples which are then fed to the Write Table operator.

Another advantage of ELT is the reduction of the size of the data that needs to be moved between the source and the target. This is especially true when the job involves reading XML data and transforming it to a relational format (E.g. flow 1 in Figure 3). Notice that the size of the source XML document (which consists of a plethora of XML tags) will be significantly larger than the relational tuples generated in output. While converting an ETL job (which reads XML data and shreds it to relational format) to an ELT flow, XPEDIA tries to push maximum possible processing inside the database engine. This helps to significantly reduce the I/O costs which contribute a significant percentage of the overall execution time.

Thus, converting an ETL job into an ELT flow can provide significant advantage and requires us to do the following tasks:

1. Rewrite the ETL flow in terms of simpler operators.
2. Convert each operator into a SQL/XML query.
3. Merge the SQL/XML queries of adjacent operators into a single SQL/XML query.
4. Convert the merged SQL/XML queries to an ELT job definition which can be executed on XPEDIA.

We explain each of the above steps in the following sections.

4.1 Rewriting ETL Flow Using Simpler Operators

Most of the operators in XPEDIA can be directly converted to a SQL/XML query. However, some of the operators like OutputStage are quite complex. Hence it is difficult to generate the SQL/XML queries directly for such operators. We overcome this problem by rewriting the complex operators using a simpler set of new operators. In this section, we describe the algorithm used for converting the OutputStage operator into a sequence of simpler operators, namely (i) XMLIZE and (ii) Sibling Group-By.

XMLIZE Operator: The XMLIZE operator converts relational tuples into a flat XML document. The schema of a "flat" XML document is very similar to the relational schema and is given in Figure 6. As the output of this operator is a flat XML document, we need to perform various operations on the output to get the XML document in the desired hierarchical format. This requires the use of a special operator which is explained next.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="root">
 <xs:complexType>
  <xs:element name="row">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="empID" type="xs:string" maxOccurs="1"/>
     <xs:element name="deptID" type="xs:string" maxOccurs="1"/>
     <xs:element name="name" type="xs:string" maxOccurs="1"/>
     <xs:element name="salary" type="xs:string" maxOccurs="1"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
 </xs:complexType>
</xs:element>
</xs:schema>
```

Figure 6: "Flat" XML Document schema

Sibling Group-By Operator: The Sibling Group-By operator is a special operator which is required to convert the flat XML document into a hierarchical format. The Sibling Group-By operator, as the name suggests, does a group-by only amongst immediate siblings present within the scope instance, which allows us to nest the input data. Given a set-valued element '*s*' and a set of group-by attributes within *s* (which have to be atomic-

valued elements) it groups all "contiguous" tuples in *s* by the values of the group-by attributes. Thus *s* is replaced in the output by a set that contains: a) the group-by attributes and b) for each contiguous value for them a nested set with all the contiguous tuples in *s* that have the same values on their group-by attributes.

Figure 7 shows the effect of applying Sibling Group-By on a sample XML document. In this example we have applied the Sibling Group-By on the */Company/Country/Dept* node. This node has two distinct values CS and EE. Hence the Sibling Group-By operator creates two sub-trees – one for each distinct value amongst contiguous nodes.

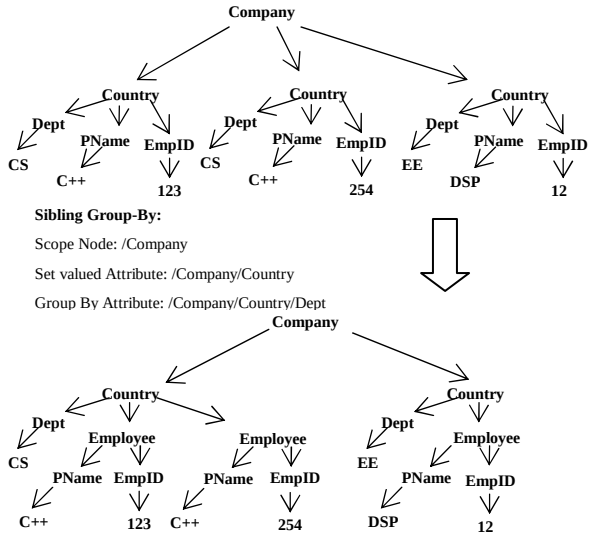

Figure 7: Sibling Group-By Operation

Consider the example transformation given in Figure 2 which creates an XML document from the given relational data. In this example the repetition path is *"/Company/Country/Employee/EInfo/EmpID*". In order to generate the XML document corresponding to this example, the first step is to convert the relational data into XML format using the XMLIZE operator. After that, we repeatedly apply the Sibling Group-By operator as follows: (1) Apply the Sibling Group-By on the */Company/Country* node with the group by node set to */Company/Country/Dept* and the scope set to */Company*. The output of this operation is shown in Figure 7. (2) Apply the Sibling Group-By operator on the output of the first step. The set valued attribute is */Company/Country/Employee* and the group by attribute is set to */Company/Country/Employee/PName*. The output of this step gives us the required XML document which is shown in Figure 2.

Thus the OutputStage operator is represented by an XMLIZE operator followed by a sequence of one or more Sibling Group-By operators. The algorithm to convert the OutputStage operator to the set of simpler operators is given below.

Step 1: Apply XMLize operator on the relational data to obtain flat XML document

Step 2: For all output nodes except the repetition output node:

- The level of an output node is the level at which its XPath intersects with the repetition output node's XPath
- Starting from the top to bottom (based on the level), apply Sibling Group-By for all nodes which meet at the same position on the repetition output node's XPath.

Step 3: Use Project Operator to add and drop nodes, so as to bring the height of all output node at correct position.

Step 4: Use Project Operator to change names of nodes

4.2 Query Generation and Merging:

The XPEDIA ELT optimizer has a set of algorithms for generating the SQL/XML query for each operator. As the query corresponding to each of the operators is fixed, the optimizer uses pre-built techniques/rules for merging the SQL/XML queries of adjacent operators. As the set of operators is exhaustive, due to lack of space, in this paper we only outline the technique to generate the SQL/XML queries corresponding to the Sibling Group-By operator and also present the technique for merging the queries corresponding to a sequence of these operators.

```
select xmlelement(name "Country", xmlagg(TB3.O)) from
    (select xmlelement(name "Department", xmlelement(name "Name", TB2.D),
        xmlagg(TB1.SS)) from session.TBTemp TB2,
            (select TB4.id, TB4.D, TB4.J1, xmlelement(name "ProjEmp",
                xmlelement(name "PName", TB4.A), xmlelement(name "EmpID",
                TB4.z)) from session.TBTemp as TB4) as TB1(ID, D, J1, SS)
            where TB1.j1 = TB2.j1 and TB1.D = TB2.D and TB1.ID = TB2.ID group by
            TB1.J1, TB2.J1, TB1.D, TB2.D order by TB1.J1
    ) as TB3(O)
```

Figure 8: SQL/XML for Sibling Group-By

As seen in the previous section, a typical use of the Sibling Group-By operator is to nest the XML document which is generated by the XMLIZE operator. Notice that the unique nature of this operator is that it only does a group by on adjacent siblings. Hence, we cannot use the "group by" clause present in SQL/XML as it will group together even those values which may not be 'adjacent' in the input XML document. In order to circumvent this problem we generate an extra attribute in the input data whose value can be used by the regular "group by" clause of the SQL/XML query. This means that, for the data shown in Figure 2, the value of the new attribute will be say 1 for the first two tuples (<CS,C++,123> and <CS,C++,253>) and 2 for the third tuple. Generating this attribute using SQL is a tricky operation. We use the OLAP function in SQL to generate these attributes in a temporary table which also contains the rest of the attributes present in the input table.

Once the temporary table is generated, we use the group by clause on the new attribute '*J1*' to generate the desired output. The sample SQL/XML query which uses a temporary table to generate the output corresponding to the Sibling Group-By mentioned in Figure 7 is shown in Figure 8.

Notice that the query operates on a temporary table. A major advantage of using the temporary table is that we can define indexes on the attributes of the table. This speeds up the processing of any joins present in the SQL/XML query.

Once the SQL/XML queries are generated for each of the Sibling Group-By operators, they can be merged with each other as well as the XMLIZE operator. The XPEDIA ELT optimizer uses a set of rules for merging these SQL/XML queries. In order to give a flavor of the merging process, we provide an example in Figure 9, of the merged SQL/XML query for the ETL job given in Figure 2. The merged SQL/XML query uses one group-by clause for each Sibling Group-By operator.

```
select xmlelement(name "Country", xmlagg(TB4.all)) from
    (select xmlelement(name "Department", xmlelement(name "Name", TB3.D),
       xmlagg(TB3.all)) from
       (select D, j2, xmlelement(name "ProjEmp", xmlelement(name "PName", TB2.A),
        xmlagg(TB2.all)) from
          (select D, j3,j2, A, xmlelement(name "Employee", TB1.zz) from
                (select D,j3, j2, A, xmlagg(xmlelement(name "EmpID", Z))
                 from session.TBTemp group by j3,j2,j1, A,D) as TB1(D,j3,j2, A,zz)
           ) as TB2(D,j3,j2,A,all) group by TB2.j3, TB2.j2,TB2.A,  TB2.D
       ) as TB3(D,j2, all) group by TB3.j2, TB3.D
    ) as TB4(all)
```

Figure 9: Merged SQL/XML query for OutputStage

4.3 Generating the ELT Job Definition

Once the SQL/XML queries have been generated, the next task is to map them back to the XPEDIA job definition. This is a straightforward process wherein the generated SQL/XML query is mapped to a Read TABLE operator and the rest of the stages remain as is.

Using the above procedure, XPEDIA is able to take advantage of the native XML processing capabilities of the database engine. However notice that we cannot generate an ELT flow when the data is present in a database which does not have native XML support or is present in a flat file. Even in such cases, XPEDIA improves the scalability by using a novel parallel processing technique which is explained next.

5. PARALLEL PROCESSING OF XML DATA

The size of a row/tuple in relational data is seldom larger than a few Kb's. However, we observed from several customers that XML inputs for data integration tend to be large aggregated inputs comprised of several smaller objects. Hence XML documents of the order of 2-3 GBs are fairly common in practice. In this section, we present a technique for parallel processing of such large XML documents.

Parallelism can be achieved in two different ways namely, pipeline parallelism and partitioned parallelism. Pipeline parallelism (also know an assembly line parallelism) occurs when different operators work on different XML documents simultaneously. Such a parallelism is supported by XPEDIA and occurs whenever multiple operators operate on a stream of XML documents in serial manner. However, this kind of parallelism can provide limited benefit for large XML documents as each operator would have to process all the documents thereby requiring large memory and processing power. Unfortunately, the ETL processing engines available today only support pipeline parallelism for XML data leading to an inferior performance. The key to solving the scalability problem (in the presence of large XML data) is the use of partitioned parallelism – a technique supported by XPEDIA. Partition parallelism is achieved when multiple instances of the ETL job (consisting of a sequence of operators) are executed in parallel on different machines/processors with each instance working on different parts of the same XML document. This reduces the size of the XML document (partition) that needs to be processed on each processor thereby improving scalability. The key to achieving partitioned parallelism, as the name suggests, is the partitioning of the large XML document. However, the hierarchical nature of XML makes the task of partitioning an inherently complex task. We outline the various challenges for achieving partitioned parallelism and propose a technique for identifying the right type of partition in Section 5.1. Once the correct partition has been identified, the next task is to actually generate the partitions efficiently in a single pass of the XML document and that too without doing a full parse of the document. We present such a technique for generating these partitions in Section 5.2. One of the most expensive operations (in terms of time) over XML documents is their schema validation. We show in Section 5.3 that the schema validation task can also be done in parallel on multiple machines thereby reducing it execution time significantly.

5.1 Identifying the Optimal Partition

As mentioned earlier, partitioned parallelism involves the following tasks: (i) partitioning a large XML document into multiple parts, (ii) running multiple instances of an ETL job on multiple machines/processors, (iii) each job instance operating on a different partition of the XML document and (iv) finally, merging the output of the job instances. In order to achieve an efficient parallelism, each partition should be self sufficient, i.e., the data required by each job instance should be contained within the partition available to it and no data should be shared across partitions. If this is not ensured then it would necessitate communication between the processors or alternatively copies of the same data would have to be made available to multiple processors, which in turn would reduce the effectiveness of the parallel algorithm. In order to highlight this further, we present an XML partitioning example.

Example: Consider a job consisting of the first two steps of the ETL job outlined in Section 3. The input to the job is an XML document whose schema is outlined in Figure 4. The join operator in the ETL job joins the data present in the *Employee* sub-tree with the data present in the *Department* sub-tree. For this job if we partition the XML document into two parts such that all the *Department* nodes go to the first partition and all *Employee* nodes go to the second partition then the join operator will require access to both the partitions. Hence no parallelism would be achieved as there will be no reduction in the size of the data processed by each operator. Thus the partitioning technique needs to ensure that the data required by an operator is contained within the partition available to it.

Consider another partitioning technique, where we partition the data at the "*/Company/Country*" level. In other words, if there are 10 *Country* sub-trees below the root (*Company*) node, then the first partition will consist of an XML document rooted at "*Company*" with only the first 5 *Country* sub-trees where as the second partition will consist of the next 5 *Country* sub-trees. Recall that the scope of the join operator is "*/Company/Country*". This means that the join operator treats each "*Country*" sub-tree independently from the rest. In other words, it joins the *Department* and *Employee* data present within each sub-tree and not across sub-trees. Further the join result of each sub-tree is also contained within that sub-tree. Hence a partitioning made at the level of "*/Company/Country*" (which is called as the partition node) would ensure that all the data required by each instance of the join operator is contained within the partition available to it.

The above example highlights the difficulties in partitioning an XML document for a job consisting of a single operator. As can be imagined, the problem gets further exacerbated for a job consisting of multiple operators. We present next some key insights which are used by the partitioning algorithm of XPEDIA

to find the partition node of a large XML document for a given ETL job.

1. **Insight 1:** Operators do not preserve any state between instances of their scope vector. Therefore, each scope instance can potentially belong to a different partition.
2. **Insight 2**: Some of the operators (such as Filter) perform stateless transformations, i.e., transformations that do not maintain state between vector instances. E.g., the filter operator computes the filter predicate per vector instance and does not maintain any state across instances. Hence if an ETL job consists entirely of such operators, then (irrespective of the scope value) each vector instance in the input can potentially belong to a different partition.
3. **Insight 3**: Some of the operators (such as Aggregate) maintain state information between different instances of their input vectors (within a particular scope instance). E.g., the Aggregate operator in the ETL job given in Section 3 finds the total salary for each *Department* within a scope instance, i.e., for each distinct *Department* in each "*Country*" sub-tree. Notice that it does not store any state information across different "*Department*" nodes within each "*Country*" sub-tree. Hence parallelism can be achieved by ensuring that all the *Employee* and *Department* sub-trees that share the same *DeptCode* are contained within a single partition. Operators which are amenable to such partitioning are called as *State Key Correlated* operators. In the above example, the Aggregate operator is state key correlated as the *Employee* sub-trees are correlated with respect to the key *DeptCode*. The *Employee* node is called as the correlated node for the operator.

We present next an algorithm which uses these insights to find the best partition node for a given ETL job.

Algorithm: Find Partition Node

1. For each operator in the ETL job
1.1 Initialize a vector containing all the scope instances for this operator
2. For each operator in the ETL job
If the operator is stateless
2.1 Do nothing
Else if the operator is state key correlated then
2.2 Generate a partition for the vector using the key for the operator
2.3. if a vector contains sub-trees rooted at the correlation node for this operator then merge the vector with the generated partition vectors
Else
2.4 Union all the vectors that contain sub-trees rooted below the scope of the operator

Result: *Each vector has a root that is a candidate for partitioning. Output the highest root amongst all the vectors as the partition node(s). If there are multiple nodes at same level, output multiple partition nodes.*

It is possible that the partition node found by the above algorithm is the root of the document, i.e., no partition of the document is possible. In such a case we could still partition a subset of the operators in the ETL flow. Extending XPEDIA to handle such cases is part of our future work.

Once the partition node has been identified, XPEDIA uses the partition node to generate the partitions as follows:

- *Round-robin partitioning technique:* In this technique, each partition gets the sub-trees rooted at the partition node in a round robin manner. In other words, the first instance of the partition node goes to the first partition; the second partition node instance goes to second partition and so on and so forth.
- *Chunking scheme:* This partitioning technique generates the partitions based on the size of the XML document. Given an XML document (of size say 4 GB) for which we have to generate (say 4) partitions, then this partitioning technique tries to generate each partition of size 1 GB. Notice that we cannot generate say the 2nd partition by directly seeking to the 1 GB location in the XML document and then finding the first occurrence of the partition node. This is because the encountered partition node could occur at multiple locations in the XML document or could be inside a CDATA section. Hence this technique generates the partition by iterating over the sub-trees rooted at the partition node (starting from the beginning of the file) till it reaches the 1 GB location in the XML file. Thereafter the second partition starts and the sub-trees rooted at the partition node constitute the second partition till we reach the beginning of the 3rd GB location in the XML file. The technique also ensures that each partition is a well formed XML document by adding the nodes from the document root to the partition node at the beginning of each partition.

5.2 Generating the Optimal Partition

Once the set of partition nodes and the partition strategy has been identified, the next task is to make the right partition available to each ETL job instance. Notice that each job instance will have the entire XML document available to it. The process of generating the partition (which is run for each job instance) involves the parsing of the entire XML document and producing the right partition for that job instance. If we do a full parse of the XML document then it will not provide us any performance gain. Hence, as we explain next, XPEDIA uses a "*shallow parsing*" technique to generate these partitions.

5.2.1 Shallow Parsing

The basic idea behind shallow parsing is that it parses only those nodes which occur between the root node and the partition node, i.e., the nodes that appear in the partition node XPath. The shallow parsing process ignores all the other nodes thereby avoiding the high cost associated with a full parse of the XML document. E.g., consider a scenario where we have a 4 GB XML document conforming to the schema given in Figure 4. Let the partition node be *\Company\Country* and we have to generate the second partition (out of the 4 partitions) using the chunking scheme. In this case XPEDIA does a shallow parse of the XML document till it reaches the 2nd GB of the XML file. In this shallow parse, XPEDIA only looks for the beginning and end of the "*Company*" and "*Country*" nodes. All the other nodes are ignored. Once it reaches the 2nd GB it outputs the sub-trees rooted at the "*Country*" node till it reaches the beginning of the 3rd GB. At this point it adds a closing "*Company*" tag and the 2nd

partition is thus generated as output. This strategy helps XPEDIA to generate the partitions in a single pass over the input XML document.

However, the process of shallow parsing can get complicated when there are multiple partition nodes. Such cases are handled by XPEDIA by looking for the nodes that appear in the XPaths of all the partition nodes. We explain the details of the shallow parsing algorithm used by XPEDIA with the help of the example given in Figure 10. The XML document in the figure has three partition nodes */A/B/C/D#*, */A/B/C/E#*, */A/B/F#*. XPEDIA maintains a set called as *ValidExtension* for each node that appears in the XPath of one or more partition nodes. The *ValidExtension* set for a node *B* would contain the set of all nodes that appear after *B* in any of the partition node XPaths. Thus the *ValidExtension* for *B* will be the set {*C, F#*}. XPEDIA also maintains a stack called *PathStack*. Whenever we encounter a node that is part of at least one partition node's XPath, that node is pushed on to the *PathStack*. If the node *B* is at the top of PathStack then the shallow parsing is done by looking for (i) the closure of the node *B* or (ii) looking for the nodes that are in the *ValidExtension* set of *B*. XPEDIA ignores all the other nodes which helps it to avoid the high cost associated with a full parse. If we see the closure of the node which is at the top of the stack, we pop it from the *PathStack*. To illustrate the algorithm further, consider a scenario where we are at the node B in Figure 9. In this case, we either push a *C* node, or a *F#* node on the stack. All other nodes are ignored since they are not part of any XPath leading to the partition nodes.

Whenever a partitioning node is found, if we have not reached the required partition in the XML document, we ignore the XML sub-tree rooted at the partition node. If on the other hand, we have reached the required partition (i.e., the 2nd GB in our earlier example) we output the sub-tree rooted at the partition node.

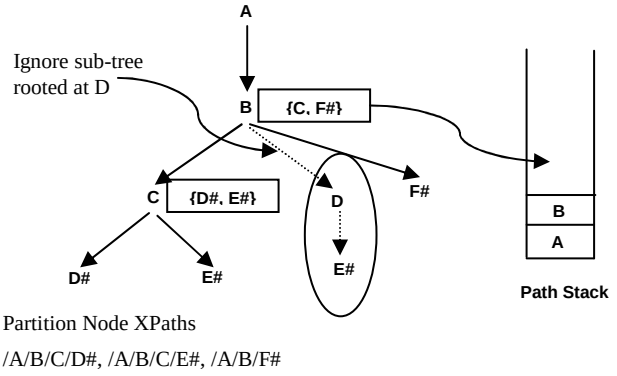

Figure 10: Shallow Parsing for multiple partition nodes.

5.2.2 Generating Balanced Partitions

The chunking scheme mentioned earlier generates equal size partitions which are processed in parallel on multiple processors. These equal sized partitions are generated by performing a shallow parse of the XML document in parallel on multiple processors where each shallow parsing instance outputs different parts of the same XML document. If we have N parallel processors (numbered 1 to N), notice that the Nth processor which will generate the last partition will have to do a shallow parse of a very large portion of the XML document as compared to the rest. Although shallow parsing is a very light weight parsing technique, nevertheless it still incurs some overhead. Hence the time required for finishing the ETL job on each processor will be different as the time required for generating the partition will gradually increase with an increase in the processor number (due to an increase in the shallow parsing time). In this section we outline a technique to balance the partitions such that the time required to generate the partitions and complete the ETL job is almost the same for all the processors. The gist of the balancing technique used by XPEDIA is that it changes the size of each partition such that partition size reduces with an increase in the processor number. Notice that the balancing technique is only required for the chunking scheme as the shallow parsing costs in the round robin partitioning technique is the same for all the partitions.

Let the size of the input XML document be S and let the number of parallel processors be N. The total time required to finish the ETL job on a single processor (without partitioning) be t_t and let the time needed to shallow parse the entire document be t_s. By definition $t_t > t_s$. Let the size of the ith partition available to the ith processor be $S_i; 1 \le i \le N$. Then,

$$\sum_{1 \le i \le N} S_i \ge S \quad (1)$$

The summation will be marginally greater than S since each partition will include the nodes from the root to the partition node. Our goal is to ensure that each processor finishes processing its partition at the same time. This means that the time required to generate each partition and to run the ETL job on that partition should be the same for each processor. This condition can be stated by the following set of equations:

$$0 + t_t S_1 = t_s S_1 + t_t S_2 = t_s (S_1 + S_2) + t_t S_3 = .. = t_s \left(\sum_{1 \le i \le N-1} S_i \right) + t_t S_N \quad (2)$$

In this set of equations, the first term signifies the time spent by each node in shallow parsing, where as the second term represents the actual time spent in running the ETL job on its partition. We have removed the denominator S, as it is common for all equations. By substituting, we get the following solution for this set of equations

$$S_i = \left(\frac{t_t}{t_t - t_s} \right)^{N-i} \cdot \left(\frac{t_t S_1 - t_s S}{t_t - t_s} \right); 1 < i \le N \quad (3)$$

The above equation gives us the size of each partition which is to be processed on each processor. These partition sizes ensure that the ETL job on all the parallel processors finishes at the same time thereby providing us maximum parallelization. In order to use Equation 3, we need to know the ratio t_s/t_t. Notice that, even if we underestimate this ratio, it will still result in some performance improvement, although may not be up to the extent of the case when we know this ratio accurately. On the other hand if we overestimate this ratio, then the performance deteriorates as one processor would process a larger partition as compared to its rightful size, thereby reducing the parallelism advantage. In practice XPEDIA learns this ratio as follows: Typically an ETL job processes hundreds of XML documents. Hence XPEDIA finds the value of the ratio while processing the first XML document and then uses it for the rest of the documents.

Thus this novel partition balancing technique helps XPEDIA to extract the maximum parallelism from the set of available parallel processors. In the next section, we outline the parallel schema validation feature of XPEDIA.

5.3 Parallel Schema Validation

Schema validation of XML documents is a computationally intensive task that takes a large amount of time to execute. XPEDIA is the first system to support XML schema validation in parallel on multiple machines/processors which helps it to significantly reduce the time required for the process. At a high level, XPEDIA achieves parallel schema validation by partitioning the XML document as mentioned in the earlier sections and providing a modified XML schema file to each parallel processor. Each parallel processor then validates its partition using the provided XML schema file. If each processor confirms that its partition is compliant with the input XML schema file, then XPEDIA guarantees that the un-partitioned XML document conforms to the original XML schema. In order to provide this guarantee XPEDIA classifies the input XML schema as either being *partition safe* or p*artition unsafe*.

An XML schema is said to be partition unsafe if it uses any of the following XML schema indicators on the partition nodes: *MinOccurs*, *MaxOccurs*, *All*, *Sequence* and *Choice*. The problem with these indicators is that it is not possible for a single partition to check these indicators. Consider the example schema given in Figure 4. Let there be a *maxOccurs* constraint on the */Company/Country* node and let it be the partition node. If we use the chunking based partitioning scheme then each partition will get some sub-set of the sub-trees rooted at */Company/Country*. Hence we cannot check the *maxOccurs* constraint using the data available within a single partition. The same holds true even in the case when we use a round robin partitioning scheme. If on the other hand there was no such schema indicator on the partition node, then notice that each partition will conform to the input XML schema. Hence such schema which do not have schema indicators on the partition node are said to be partition safe and we provide the original XML schema file to each of the processors.

In case the schema is partition unsafe then XPEDIA uses a special technique for schema validation. As mentioned earlier, the problem with unsafe schemas was that it was not possible to check the schema using the data available within a single partition. However, notice that the N^{th} (i.e., the last) processor generates its partition by shallow parsing the entire XML document and producing the last portion of the XML document as its partition. Recall that the shallow parsing involves the iteration over the sub-trees rooted at the partition node. Hence the schema validation for the schema indicators defined on the partition node can be easily done while doing the shallow parsing. For the example mentioned earlier, if there was a "*maxOccurs=10000"* constraint on the */Company/Country* node, then the shallow parser on the last processor can easily keep track of the number of *Country* nodes that it has encountered and signal an error if the number crosses 10000. Even in the case of round robin based partitioning scheme, the last processor does a shallow parsing of the entire XML file and hence can easily check the schema indicator constraints. Thus the schema indicator is checked during the shallow parsing on the N^{th} processor and hence these schema indicators are removed from the schema file that is provided as input to each of the processors. Thus each processor checks the constraints present in the schema file except the schema indicators defined on the partition node which are checked by the shallow parser. Thus XPEDIA handles both partition safe and partition unsafe schemas which helps it to significantly reduce the time required for schema validation.

6. PERFORMANCE EVALUATION

In this section we present the experimental evaluation of XPEDIA. The aim of the experimental evaluation was to showcase the performance gain that can be achieved by using the two techniques presented in this paper, namely: (1) Rewriting an ETL job to an ELT flow when the source has native XML support and (2) running the ETL job in parallel on multiple processors. We first describe the experimental setup in the next section and then present the results for ELT and parallel processing of XML data in Section 6.2 and Section 6.3 respectively.

6.1 Experimental setup

We conducted two different sets of experiments one for each of the two scenarios mentioned above (i.e., ELT and parallel XML processing). As discussed earlier, in the first scenario, we rewrote the ETL job definition to generate an ELT flow. We used the ETL job described in Section 3. In the original job the XML document was retrieved from the database, transformed by the different operators of the ETL job and finally shredded to relational format and output to the target. In the modified ELT job, the transformations and shredding tasks were pushed inside the database engine by executing a single SQL/XML query. The output of the SQL/XML query consisted of relational tuples which were then output to the target. Both the original and the rewritten jobs were executed on IBM Information Server V 8.1 which is an ETL engine. These experiments were conducted on an Intel Xeon machine with 3.16 GHz processor and 3.75 GB memory. The operating system was Windows 2003 server edition. The data source used to store the original XML data was DB2 v9.5 [13] which provides native XML support.

In the second scenario, we conducted experiments to validate the performance advantage provided by our parallel processing techniques. For this experiment we again used the ETL job described in Section 3. We ran the job using Information Server v8.1 [12] running on a single machine. We then modified the job by adding a shallow parsing step at the beginning of the job. This modified job was also run using Information Server v8.1 which provides supports for executing an ETL job in parallel on multiple machines/processors. The ETL job was run on a 4 CPU Intel Xeon Quad-core machine with each CPU having a processor speed of 3.16 GHz. Thus the value of N in this case was 4 and the ETL job instance on each processor processed a different partition of the input XML document. The overall memory in the machine was 4 GB and its OS was Windows 2003 server edition. The XML documents used in these experiments were generated synthetically using the schema defined in Figure 4.

6.2 Impact of ELT optimization

In this section, we present the results for the experiments which showcase the benefit of using the ELT optimization. We compared the execution time for the ETL and ELT job for different sizes of XML documents. The results for this experiment are shown in Figure 11. The figure shows that the time taken by the ELT job for an XML document of size 700 MB

is 70% less than the time taken by the ETL job. Thus this experiment shows the the ELT approach is able to significantly reduce the job execution time by making use of the indexes available within the database engine.

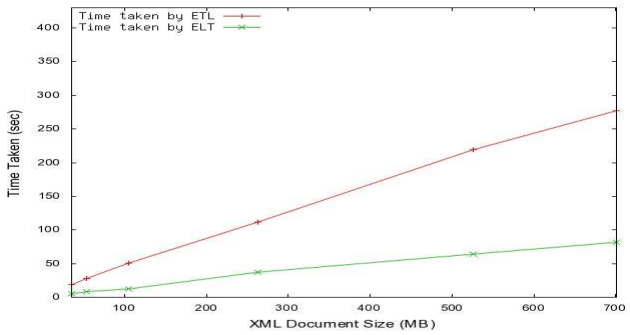

Figure 11: Comparison of ETL and ELT job execution time

6.3 Effect of Parallelization

In the next set of experiments, we evaluated the speedup achieved by the partitioning technique of XPEDIA. Recall that XPEDIA supports two different partitioning schemes: Round robin based partitioning and chunking based partitioning. We showcase the performance improvement achieved by both these techniques in this section.

Round-robin scheme: In this partitioning scheme, the shallow parsing is done by one processor which then sends the sub-trees rooted at the partition node to the rest of the processors in a round robin manner. We ran this experiment on 4 processors where the first processor did the shallow parsing of the XML document and it provided the sub-trees below the partition nodes to the remaining 3 processors. The result of this experiment for different XML document sizes is shown in Figure 12. The results show that the round robin partitioning technique provides a speedup of 2.7 times over the non-partitioned approach. The results also show that the speedup is scalable and is not affected by the size of the XML document. We also measured the time required for shallow parsing as a percentage of the total time required to execute the ETL job. The results, shown in Figure 13, validate the fact that the shallow parsing overhead is constant across different XML document sizes.

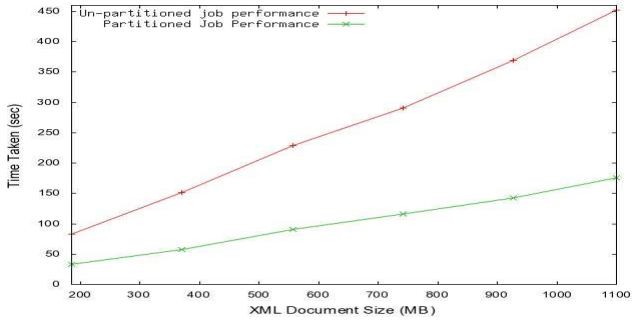

Figure 12: Performance of round robin partitioning scheme

Chunking scheme: In the chunking scheme, each processor does a shallow parsing of the XML file starting from the beginning of the file till it reaches the start of its partition. We compared the execution time of the ETL job without partitioning with that of an ETL job running on multiple processors using the chunking based partitioning scheme.

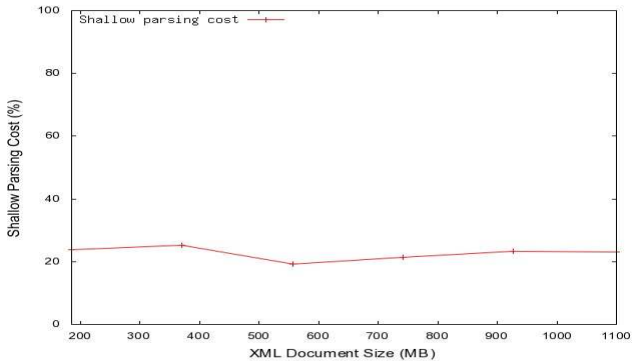

Figure 13: Shallow parsing overhead

In the first set of experiments, we generated the partitions without using our partition balancer. Thus, each partition in this case is of the same size. As a result of this, the first processor has to do the least amount of shallow parsing and hence finishes it work in the minimum amount of time whereas the last processor has to do to maximum amount of shallow parsing and hence requires the maximum amount of time to finish the ETL job on its partition. Figure 14 shows the time required by the un-partitioned job as well as the minimum and maximum time taken by the partitioned job across all the processors for different XML document sizes. The final speed up of the job is calculated based on the finish time of slowest job instance. The results show that the chunking based partitioning scheme without partition balancer provides a speedup of 2.25 times over the un-partitioned approach.

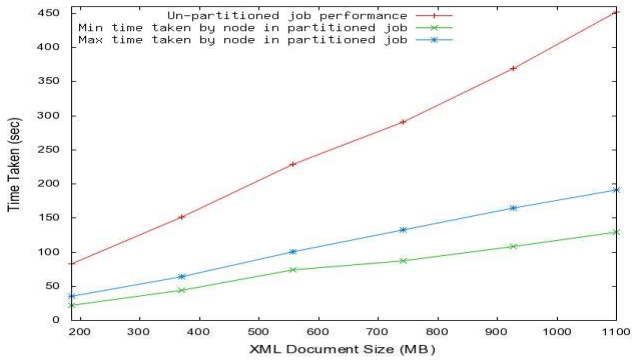

Figure 14: Performance of chunking scheme without partition balancer

In our second set of experiments, we used the partition balancer in the chunking based partitioning scheme. In this technique each partition is of different size such that the overall ETL job execution time is almost the same across all processors. The result of this experiment is shown in Figure 15. For this experiment, we report only the time taken by slowest job instance since the difference between slowest and fastest ETL job instance is very small. We assumed the ratio of shallow parsing to total ETL job execution time as 0.2 for this experiment. The results show that technique provides a speedup of 2.9 times over the un-partitioned approach. This validates the effectiveness of the partition balancer which helps it to provide better performance than the round-robin scheme across various XML document sizes.

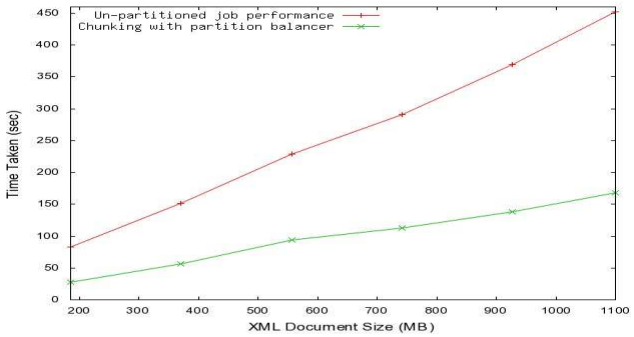

Figure 15: Performance of chunking scheme with partition balancer

In summary, our experimental results show that:

- We can get a performance gain of up to 70% by pushing the processing inside the database engine.
- Our strategy of partitioning the ETL job on multiple nodes is scalable and can improve the processing speed of the ETL job by up to 2.9 times for a 4 processor configuration.

7. RELATED WORK

There has been significant amount of work on performing efficient ETL processing over relational data. However, as mentioned earlier, ETL processing over XML data is not a well studied topic. [3] presents a system that allows users to specify declarative mapping specifications and generate ETL jobs. The proposed system is not specific to XML but can perform XML mappings. This work is complimentary to the XPEDIA system which incorporates significant XML specific improvements such as providing a XML specific computational model, providing support for ELT and handling large XML documents. [4] presents an approach to perform XML data integration, but its primary focus is on data federation. In contrast, our paper is focused on ETL techniques for XML data.

Query rewriting techniques based on schema mapping constraints have been studied in [5]. These techniques are similar to our ideas for ELT processing but more general in scope. There has been some work on efficient parallel processing of large XML documents [6, 7]. The technique presented in [6] partitions data based on the XML Infoset model but involves a sequential pre-processing step. Similarly [7] describe a technique for parallelizing XML parsing using a pre-parsing approach. The fundamental problem with all these works is that they require a sequential pre-parsing step which can be quite costly. Speed of execution is a key requirement for ETL processing flows and hence techniques which need multiple passes over the XML data do not work in practice. XPEDIA uses an innovative single pass algorithm that avoids these drawbacks resulting in improved scalability. In summary, XPEDIA is one of the first systems to support a XML specific computation model, ELT support and specialized techniques for parallel processing of large XML documents.

8. CONCLUSION

In the past, data integration (or ETL) engines were only required to shred or generate XML documents of modest sizes. However, with the growing adoption of XML for data integration, we see significant new demands for performing complex transformation and processing operations on large XML documents and document sets. In this paper, we studied several new requirements for processing XML data inside data integration engines. We presented the XPEDIA system which has a XML specific computational model for performing a variety of operations in an ETL engine. We also showed how a data flow of operations can be composed as a pipeline and executed in the ETL engine. We described the methodology and steps for converting an ETL flow for XML operations into an equivalent SQL/XML query that can be executed in databases capable of XML processing. This ELT technique is effective in pushing down the operations when the source or target of XML ETL flows is a database capable of processing SQL/XML efficiently. We then described two techniques (supported by XPEDIA) for partitioning large XML documents in order to process the XML in a parallel execution environment. We also proposed a technique for XML schema validation in parallel on multiple machines. We did an experimental evaluation of XPEDIA which showed that the ELT and partitioning techniques are very effective in improving the performance of XML based ETL integration tasks. We are currently working towards incorporating these techniques into IBM's Information Server data integration engine. In the future, we expect to provide a unified set of processing options combining ETL, ELT and automatic parallelism for XML data flows. We also intend to study the parallelism topic for XML data flows in further detail.

9. REFERENCES

[1] W3C XSLT Specification, *http://www.w3c.org/TR/xslt*

[2] W3C XML Schema, *www.w3.org/XML/Schema*.

[3] Dessloch, S., et. al., "Orchid: Integrating Schema Mapping and ETL", in *Proceedings of ICDE*, 2008.

[4] Draper, D., Halevy, A., Weld, D.S., "The Nimble Data Integration System", in *Proceedings of ICDE*, 2001.

[5] Yu, C., and Popa, L., "Constraint-based XML query rewriting for data integration", in *SIGMOD*, 2004.

[6] Kurita, H., et. al. "Efficient Query Processing for Large XML Data in Distributed Environments", in *21st Intl. Conf. on Advanced Networking and Applications*, 2007.

[7] Lu, W., Chiu, K., Pan, Y., "A Parallel Approach to XML Parsing", in *Proceedings 7th IEEE/ACM Intl. Conf. on Grid Computing*, 2006, pp. 223-230.

[8] Chamberlin D., "XQuery: A Query Language for XML", in *Proceeding of ACM SIGMOD*, 2003: 682.

[9] Nicola M., Linden B., "Native XML Support in DB2 Universal Databases", in *VLDB*, 2005, pp. 1164-1174.

[10] Liu Z. H., Krishnaprasad M., Arora V., "Native XQuery Processing in Oracle XMLDB", in *SIGMOD*, 2005.

[11] Rys M., "XML and relational database management systems: inside Microsoft SQL Server", in *SIGMOD*, 2005.

[12] IBM Infosphere Information Server, *http://www-01.ibm.com/software/data/integration/info-server*.

[13] IBM DB2 Database, *http://public.boulder.ibm.com/infocenter/db2luw/v9r5*.