Optimizing Queries for Web Generated Sensor Data

Gerard Marks **Mark Roantree** **Dominick Smyth**

Interoperable Systems Group, School of Computer Science,
Dublin City University,
Glasnevin, Dublin 9, Ireland
Email: gmarks,mark,dsmyth @computing.dcu.ie

Abstract

The Web continues to offer new services and in doing so, generates large volumes of new data. The concept of the Sensor Web has led to web generated sensor data becoming available not only to the companies that own the sensors but also to end users and knowledge workers. Applications vary from climate, traffic and sports event monitoring. In one application that provides an availability service for bicycle rentals in major cities, large volumes of data are collected for data mining purposes. All transactions are recorded and stored in XML format leading to performance issues as volumes increase. In this paper, we apply a new optimization technique to boosting the performance of XML queries on Sensor Web data.

Keywords: Sensor Web, XML Queries, XML Optimization.

1 Introduction

The use of sensors in the physical-world is constantly increasing and could now be regarded as widespread. The number of applications built on top of this sensor data is also increasing. Examples are: urban traffic watch; weather monitoring; tracking of goods, etc.

Recently the city of Dublin, in keeping with many other European cities, deployed a bike sharing scheme where people can rent (and return) a bike from stations located throughout the city centre. Stations are equipped with sensors that monitor bike availability and publish such data to the DublinBikes website (www.dublinbikes.ie). Consumers can connect to the website (either through PC or mobile application) to check where stations are, how many bikes are available for rent, how many spaces are available to return bikes, and what type of payment methods are available. This data is of great interest to both consumers and providers of the service. Consumers can check where to rent or return a bike while providers can understand at which station it is better to pick up or return bikes for maintenance in order to minimize service disruption. In effect, the web service offers an efficient mechanism for determining the current status of bike or space availability.

Funded by Enterprise Ireland Grants No. CFTD/07/201 and IR/2009/001

Motivation. There are many requirements where it is necessary to access historical data or look for trends and patterns over time. For example, city planners or the companies offering the bicycle rental service must determine the optimum location for new sites; determine those sites that require expansion, or reduce or close sites that are unpopular. Furthermore, this analysis must take place over large periods of time to avoid any bias that could result from poor weather patterns or holiday season. The problem with large volumes of XML data is that queries often perform badly. What is needed is an efficient mechanism for harvesting the sensed data; updating the data warehouse, and optimizing the user queries.

Contribution. The contribution of this paper is twofold. Firstly, we provide a framework for harvesting sensor web data and automatically preparing the data for insertion into the data warehouse. To be fully automated, data must be harvested online, enriched or transformed automatically and presented for mining operations using standard query languages such as XPath and XQuery. Secondly, we must provide a superior XML processor to those currently available to address the performance issues of current approaches. Our BranchIndex was first introduced in (Marks and Roantree 2010) where we used standard benchmarking datasets to compare it against other approaches. Here, we use a real world dataset for the first time and introduce new optimizations to the index structure. We provide details of our systems performance through a series of experiments where the repository exceeds 2GB in size, with continuous harvesting from a number of bike rental sites every 60 seconds.

Paper Structure. The paper is structured as follows: in §2, we provide a more detailed description of the problem involved in extracting information from sensor repositories; in §3, we present a discussion of similar research in this area; in §4, a description of data management processors for Web Sensor data is provided; in §5, we describe how XML queries are optimized; in §6, we use an end-user query set to provide details as to how our system performs against other XML storage solutions; and finally in §7, we provide some conclusions.

2 Problem Description

The bicycle rental application records information on bike availability in cities and towns across the world. The data is collected from each location at regular intervals (see Table 1) and the dataset at the time of our experiments was 2.06 GB in size. Later in the description of our architecture, we show a typical XML document for a single station in the city of Lyon) in Fig. 2.

The following queries were supplied to us to carry out an evaluation of the *bicycle rental dataset* dataset.

City	Country	Stations	Data Size
Aix-en-Provence	France	16	8 KB
Amiens	France	25	8 KB
Besancon	France	30	8 KB
Lyon	France	340	80 KB
Mulhouse	France	35	12 KB
Nancy	France	25	8 KB
Nantes	France	89	24 KB
Plaine-Commune	France	44	12 KB
Rouen	France	18	8 KB
Dublin	Ireland	40	12 KB
Toyama	Japan	16	8 KB
Luxembourg	Luxembourg	46	12 KB
Santander	Spain	13	4 KB

Table 1: Bicycle Rental Data Collection

- Q1. Return all information recorded for each station in the city of Nantes.
  `/bikes/city/Nantes/stations/station`
- Q2. Return the number of bikes that are free across all stations in Dublin.
  `/bikes/city/Dublin/stations/station/free`
- Q3. Return all stations, from all cities, that have bicycle availability.
  `//city//stations[./station/available]`
- Q4. Return all stations, from all cities, that have information on wind direction and speed; and the time it was recorded.
  `//city//stations[.//weather/time]`
  `[.//weather//wind/direction]`
  `[.//weather//wind/speed]`
- Q5. Return all information regarding stations in Luxembourg in which there were no free bikes.
  `//Luxembourg/stations[./station/available = ‘0’]`
- Q6. Return the *id* (identification) of each station that had no bicycle availability.
  `//stations/station[./available = ‘0’]/id`
- Q7. Return those cities with a wind speed greater than 6 miles per hour.
  `//stations[.//wind/speed > ‘6’]/parent::*`
- Q8. Return the stations that had a wind direction of 40.
  `//direction[.= ‘40’]/ancestor::stations /station`
- Q9. Return all entries for Lyon for a specific date: *01/06/2010*.
  `//Lyon[./@day = ‘01’][./@month = ‘06’][./@year = ‘2010’]`
- Q10. Return the wind chill in Lyon for a specific date: *01/06/2010*.
  `//Lyon[./@day = ‘01’][./@month = ‘06’][./@year = ‘2010’]//chill`

There is a considerable gap between the user requirements outlined by the query set and the information generated by service providers. Unlike most sensor networks, data is provided in XML format which provides high degree of interoperability and some level of semantics. However, none of the current query set can be processed without a number of processing layers. The best solution should see the provision of generic services with domain-specific or application-specific processors reduced to a minimum.

3 Related Research

As we combine the topics of harvesting sensor data and optimization of XML repositories, we separate our discussion on state of the art into two categories.

3.1 Extraction of Web Based Data

There has been considerable research into information extraction from the web. Examples include Stalker (Knoblock et al. 2003) and TextRunner (Banko et al. 2007). A comparative survey of a number of methods was conducted by Chang (Chang et al. 2006). All of these efforts focus on wrapper induction and generation for extracting information from semi-structured data. However, they do not address enrichment of data from heterogeneous sources, a necessary process for querying and data mining.

Wang (Wang et al. 2006) explores the issue of data transformation in the context of RFID data streams. While they limit their attention to semantic enrichment of data by exploiting contextual information and on complex ECA rules, we are also interested in storing historic data while harvesting data from web sources. Other researchers such as (Liu et al. 2006) and (Erwig 2003), consider the general transformation of XML data in the interest of interoperability. While this research provides generic constructs and operations to transform XML data, they no not focus on either data enrichment and mining capabilities. In approaches to storing web sensor data, Balazinska (Balazinska et al. 2007) provides extensive information on storing sensor data from web sources, previous trend analysis and emerging trend recognition.

Efforts focusing on mining for *association rules* in XML documents can be found in ((Liu, Zeleznikow et al. 2006), (Wan and Dobbie 2004) and (Rusu et al. 2007)). Nayak et al. (Nayak et al. 2002) survey the data mining techniques that can be applied to the structure or content of XML documents. While these approaches all focus on discovering relationships between data in XML format, they do not deal with the task of collecting, enriching and structuring the data in preparation for high level queries.

In summary, all of the above provide differing solutions to harvesting of web data. However, none provide any mechanism for optimization. In the remainder of this section, we focus on those research efforts that offer XML optimization in a manner similar to us.

3.2 Optimization Using Partition Indexes

XML repositories differ from their relational counterparts in that they have a tree type structure and employ a set of axes to navigate across the tree structure to compute query results. Current query optimization efforts can be classified into two broad categories: *index based* approaches that build indexes on XML documents to provide efficient access to nodes, as found in XPath Accelerator (Grust 2002) and Xeek (Luoma 2007b); and *algorithmic based* approaches that are focused on designing new *join* algorithms, e.g. TJFast (Lu et al. 2005), StaircaseJoin (Grust et al. 2003). The first approach can use standard relational databases to deploy the index structure and thus, benefit from mature relational technology. The second depends on a modification to the underlying RDBMS kernel (Boncz et al. 2006), or a *native* XML database that is newly created. The *XPath Accelerator* (Grust 2002) demonstrated that an optimized XPath index stored inside a relational database can be used to evaluate all 13 XPath axes. However, the XPath Accelerator, and similar approaches (Grust et al. 2007), suffer from scalability issues, as this type of node evaluation (even across relatively small XML documents) is inefficient (Luoma 2007b).

A more recent solution is to partition nodes in an XML tree into *disjoint* subsets, which can be identified more efficiently as there will always be less partitions than there are nodes. After the relevant

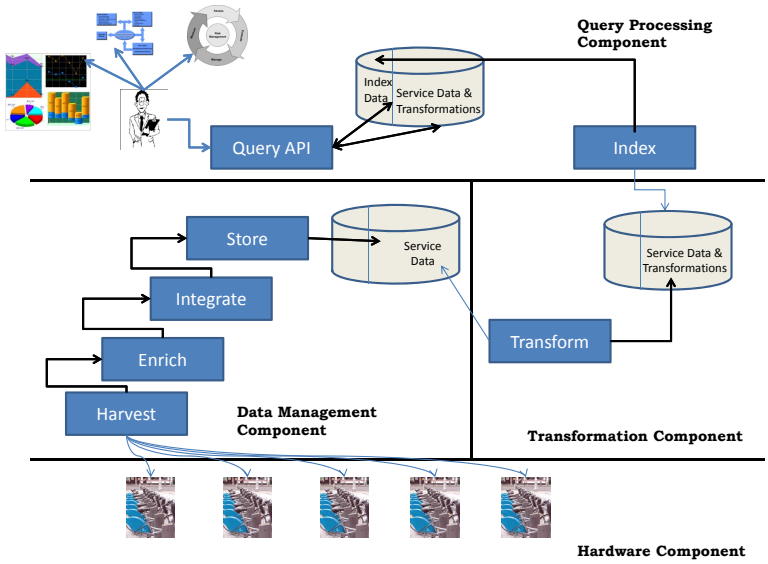

Figure 1: System Components and Data Management

partitions are identified, only the nodes that comprise these partitions are evaluated using the inefficient node comparison step. Based on *pre/post* encoding, (Luoma 2007a) is an index based approach that requires a user defined *partitioning factor* to divide the *pre/post* plane into disjoint sub-partitions. At present, this approach lacks flexibility in determining the size of partitions without a time-consuming preprocessing stage to identify suitable partitioning factors.

Optimization of XML repositories using node partitioning will segment documents into disjoint subsets. In effect, one creates an index of partitions rather than an index of nodes. As there are fewer partitions than nodes in an XML dataset, highly efficient join operations can be performed between partitions, which reduces the workload for the more inefficient task of node comparisons.

In (Luoma 2007a), the *pre/post* plane is partitioned based on a user defined *partitioning factor* of 4. For each node, the *pre/post* identifier of its *part* is the *lower bound* of its x and y values respectively. The ancestors of node x can only exist in the *parts* that have a lower bound x value ≤ 4 and a lower bound $y \geq 4$. They can also compute similar queries for the other major XPath axes, i.e. descendant, following and preceding. The problem with this approach is that an ideal partitioning factor is not known in advance and requires rigorous experimentation to identify. For example, in reported experiments each XML document was evaluated for the partitioning factors 1, 2, 4,.., 256 (Luoma 2007a). In general, this type of experimentation is not feasible even for relatively small XML documents. Additionally, as XML data is irregular by nature, a single partitioning factor per dataset is less than ideal. Finally, although they suggest that the partitioning approach may be tailored to other encoding schemes such as: *order/size*, it relies heavily on the *lower bound* of each x and y value in the partitioned *pre/post* (or order/size) plane. Therefore, this approach does not lend itself naturally to *prefix* based encoding schemes such as ORDPATH (O'Neil et al. 2004), which have become very popular in recent years for reasons of updatability.

Our approach overcomes these issues by automatically partitioning nodes based on their individual layout and structural properties within each XML dataset. We do not rely on user defined partitioning factors, therefore avoiding the time-consuming processes that are required to identify suitable partitioning factors. Also, our approach is not dependent on the specific properties of XML node labels, and thus can be used in conjunction with any XML encoding scheme.

Finally, an algorithm based partitioning approach (Wong et al. 2008) uses partitioning in the *pre/post* plane to optimize the performance of native XML structural join operations. This approach is similar to (Luoma 2007a) as it depends on the properties of *pre/post* labels to avoid unnecessary node comparisons. Additionally, as (Wong et al. 2008) adopts an algorithm based approach, the partitions are dynamically calculated i.e. they are not indexed. Thus, in contrast to our approach, the calculation of these partitions leads to a performance overhead during the query process.

4 Data Acquisition and Management

In the next two sections, we present our architecture used to acquire and manage data. In this section, we cover: station sensors; the harvesting, enrichment, transformation processors; and the query and mining component. We then dedicate the following section to a discussion on how we optimize XML queries.

4.1 Hardware Component: Sensors

Irrespective of the city or service provider, each station consists of a number of parking stands for bicycles, the actual bicycles, and a sensor based system to determine the status of the parking stands (empty or occupied). For each station this information is made available through the service provider's web site which provides:

- station ID
- total number of bike stands
- number of bikes available
- number of free bike stands available
- number of tickets

Timestamp information is not available through the online service so we generate our own timestamp when harvesting data.

4.2 Data Management Component

The Data Management component collects and organizes data. It is composed of four processors: Data Harvester, Enrichment and Integration and Storage.

The role of the **Harvest Processor** is to pull data from specified online services and as such, is a website wrapper that retrieves data every 60 seconds. Input to this component is the list of URLs where service data is located, while the output is a number of XML sensor files, one for each station. At present, the process takes between 12 and 22 seconds per site, with 13 websites in the system. A failures to retrieve data, for example when the site in unavailable, is managed using a timeout policy. The harvesting process was repeated at one minute intervals throughout the day.

The **Enrichment** processor is used to provide context for current station sensor data. There has been different approaches to context provision for sensor data in the past (Whitehouse et al. 2006, Devlic et al. 2008, Marks and Roantree 2009) and here, we adopt a simple system of enrichment of harvested data using available information such as station location, time and date, and weather (see XMLschema in figure 3). The time at which the information is retrieved is added as a timestamp to the station data. The time required to retrieve data from each station is also added. Finally, station data is integrated with the most recent weather conditions using various weather sites. Within the database, a schema was defined to group files into MonetDB collections based on the frequency of change of common attributes, i.e. date. The schema is displayed in Fig. 3. For a single city (in this case Dublin), this schema results in a parent collection, "Dublin", at level one; a new collection each year at level 2; a new collection each month at level 3 and finally a new collection each day at level 4.

The **Integration** processor has the task of combining inputs from distributed sources. At present, stations are synchronized by time and by city.

The **Storage** processor places the final XML document into its appropriate place in the schema. As existing XML technology is not robust enough to manage continuous updates, we cache all data for and execute a single transaction to the MonetDB database once every 30 minutes. Part of ongoing research is to optimize our updates and reduce the time between updates. However, as new sites are added to the system, the size of each transaction is growing and the time necessary to commit updates increases accordingly.

4.3 Transformation Component

Processors in the Data Management component are generic and applicable across domains. The data transformations required by end users or specific query expressions are based upon the needs of end users and thus, the architecture uses a separate component to perform transformations. There are currently two transformations required. The first reorganizes data from the minute-by-minute view output of the collection to a station-by-station view; the second provides an aggregation of all station data on a minute-by-minute basis. The transformation process takes place once a day in order to eliminate latency within the system.

Every 24 hours, the system harvests 1,440 sensor outputs per city, each representing the state of all

```
<bikes>
  <city>
    <Lyon day='01' month='06' year='2010'>
      <stations>
        <time>
          <hour>19</hour>
          <minute>59</minute>
          <second>50</second>
        </time>
        <timeOfDay>19:59:50 01-06-2010</timeOfDay>
        <timeUnit>milliseconds</timeUnit>
        <timeStart>1275418790000</timeStart>
        <weather>
          <time>Tue, 01 Jun 2010 8:30 pm CEST</time>
          <wind>
            <chill>63</chill>
            <direction>40</direction>
            <speed unit="mph">7</speed>
          </wind>
          <humidity>59</humidity>
          <pressure unit="inches">29.97</pressure>
          <temp unit="degrees farenheit">63</temp>
          <condition>Partly Cloudy</condition>
          <weatherTimeTaken>75</weatherTimeTaken>
        </weather>
        <station>
          <id>9052</id>
          <timeTaken>2853</timeTaken>
          <available>2</available>
          <free>20</free>
          <total>22</total>
          <ticket>1</ticket>
          <error>0</error>
        </station>
      </stations>
    </lyon>
  </city>
</bikes>
```

Figure 2: Single Station Sample for Lyon

stations concerning the number of bicycles or parking slots at the given time stamp. For live queries, this information is sufficient to determine bike or free slot availability. However, not all of the data mining queries from our motivation section can be easily facilitated with the data in this format.

4.3.1 Transform 1 - Station Files

For each city, the system harvests from each station and viewing each dataset in chronological order, the XQuery code listed in Algorithm 1 was used to produce the *station files*. In Table 1, the `Stations` column shows the number of files generated for each city. Each station file contains a minute by minute status of one station for an entire day. While harvested data provides detailed information on the number of bikes and free spaces available at any given moment, station files only report on the status of a station, which is limited to one of 5 options.

Station Status:

- Full - all bike stands occupied by a bike.
- Empty - all bike stands empty.
- Error - an error occurred during data collection from the web site.
- Bad Sum - the sum of free spaces and available bikes did not equal the total stands.
- Normal - data collected and processed as expected (default value).

In addition, the average scrape time for collection of data from the given station over the course of the day was also recorded. An example of this transformation has been represented in Algorithm 1. The result of these transformations can be seen in figure 3 and are used in a number of the queries during the evaluation phase.

Algorithm 1 Station Transform

```
1:  for i in 1 to numStation do
2:    for $s in /station/station[id=i] do
3:      if $s/error)=1 then
4:        $result = 'error'
5:      else if $s/available + $s/free ≠ $s/total
        then
6:        $result = 'bad sum'
7:      else if $s/free = 0 then
8:        $result = 'full'
9:      else if $s/available = 0 then
10:       $result = 'empty'
11:     else
12:       $result = 'normal'
13:     end if
14:     order by time
15:     return $result
16:   end for
17: end for
```

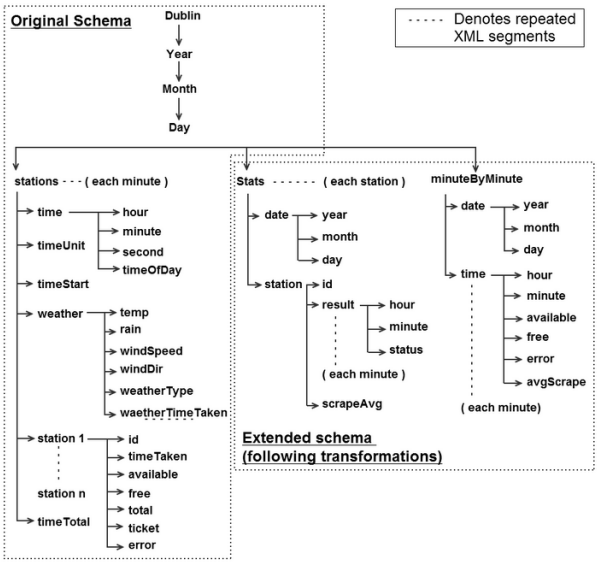

Figure 3: Extended Schema

4.3.2 Transform 2 - System File

Our second transformation process again takes the 1,440 sensor outputs for each city, for the 24-hour period. Operating independently of transform 1, this transform generates one output file per city that provides, on a minute by minute basis: the total number of bikes available, the number of available stands and a count of stations that reported an error across the entire system.

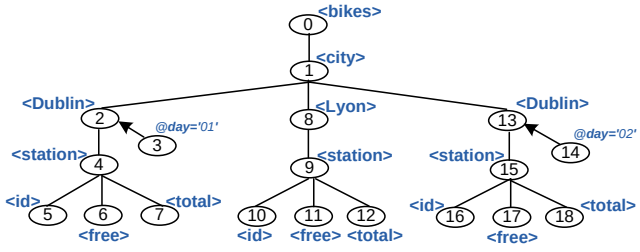

Figure 4: Small Segment from the Bicycle Rental Repository

5 Query Processing Component

In this section, we describe how to optimize queries through indexing XML documents. To do this, we present the key constructs used in our optimization model and then provide a step-by-step description of how we create a partition index that facilitates fast generation of result sets.

Definition 1 *A branch is a* set *of connected node identifiers within an XML document.*

A branch is the abstract data type used to describe a partition of nodes. In our work, we will deal with the local-branch and path-branch sub-types of a branch.

Definition 2 *A local-branch is a branch, such that its members represent a single branching node and the nodes in its subtree. A local-branch cannot contain a member that represents a descendant of another branching node.*

The local-branch uses the branching node to form each partition. Our process uses the rule that each local-branch must not contain nodes that are descendants of another branching node to create primary partitions.

Definition 3 *A path-branch is a branch with a single path.*

The path-branch is an abstract type with no branching node. Each member is a child member of the preceding node. Its three sub-types (*orphan-path*, *branchlink-path* and *leaf-path*) are used to partition the document.

Definition 4 *An orphan-path is a path-branch such that its members cannot belong to a local-branch.*

The orphan-path definition implies that members of the orphan-path cannot have an ancestor that is a branching node. The motivation is to ensure that each node in the XML document is now a member of some partition.

Definition 5 *A branchlink-path is a path-branch that contains a link to a single descendant partition of its local-branch.*

In any local-branch, there is always a single branching node and a set of non-branching nodes. With the non-branching nodes, we must identify those that share descendant relationships with other partitions. These are referred to as branchlink-path partitions and each member occupies the path linking two branching nodes (i.e. two partitions).

Definition 6 *A leaf-path is a path-branch that contains a leaf node inside its local-branch.*

A leaf-path differs from a branchlink-path in that it does not contain a link to descendants partitions. In other words, it contains a single leaf node and its ancestors.

Definition 7 *A branch class describes the structure of a branch, from the document node to its leaf node, and includes both elements and attributes.*

label	name	type	level	class	value
(0,18)	bikes	3	0	n/a	-
(1,17)	city	1	1	11	-
(2,5)	Dublin	1	2	5	-
(3,0)	day	2	3	5	01
(4,4)	station	1	3	4	-
(5,1)	id	1	4	1	-
(6,2)	free	1	4	2	-
(7,3)	total	1	4	3	-
(8,10)	Lyon	1	2	10	-
(9,9)	station	1	3	9	-
(10,6)	id	1	4	6	-
(11,7)	free	1	4	7	-
(12,8)	total	1	4	8	-
(13,16)	Dublin	1	2	5	-
(14,11)	day	2	3	5	02
(15,15)	station	1	3	4	-
(16,12)	id	1	4	1	-
(17,13)	free	1	4	2	-
(18,14)	total	1	4	3	-

Table 2: NODE Index

name	class	level	type
id	1	4	1
free	2	4	1
total	3	4	1
station	4	3	1
Dublin	5	2	1
day	5	3	2
id	6	4	1
free	7	4	1
total	8	4	1
station	9	3	1
Lyon	10	2	1
city	11	1	1

Table 3: NCLT Index

ac	dc
1	1
2	2
3	3
4	1
4	2
4	3
4	4
5	1
5	2
5	3
5	4
5	5
6	6
7	7
8	8
9	6
9	7
9	8
9	9
10	6
10	7
10	8
10	9
10	10
11	1
11	2
11	3
11	4
11	5
11	6
11	7
11	8
11	9
11	10
11	11

Table 4: CLASS Index

5.1 Indexing the Node Set

The algorithms for encoding an XML document using a *pre/post* encoding scheme were provided in (Grust 2002). In brief, each time a *starting tag* is encountered a new element object is created, which is assigned the attributes: *name, type, level,* and *preorder*. Subsequently, the new element is pushed onto an *element stack*. Each time an *end tag* is encountered an element is popped from the element stack and is assigned a postorder identifier. Once an element has been popped from the stack, we call it the *current node*, and the *waiting list* is a *set* in which elements reside temporarily prior to being indexed.

When creating the first set of partitions, the goal is to ensure that all nodes are included in local-branch or path-branch partitions. The first step in the process is to determine if the current node is a branching node by checking if it has more than one child node. The next steps are as follows:

1. If the current node is non-branching and does not reside at *level 1*, it is placed on the *waiting list*.
2. If the current node is branching, it is assigned to the next local-branch in sequence. Therefore, current nodes in the waiting list become its descendants and will occupy the same local-branch as the current node.
3. If the current node is non-branching, but a node at *level 1* is encountered, the current node does not have a branching node ancestor. Therefore, the current node is assigned to an *orphan-path* (Definition 4). For the same reason, any node currently on the waiting list is assigned to the same orphan-path.

At the end of this process, only the *document node* is unassigned. Fig 5 illustrates the set of local-branches `LB-1` to `LB-8` and orphan-paths `OP-9` and `OP-10`.

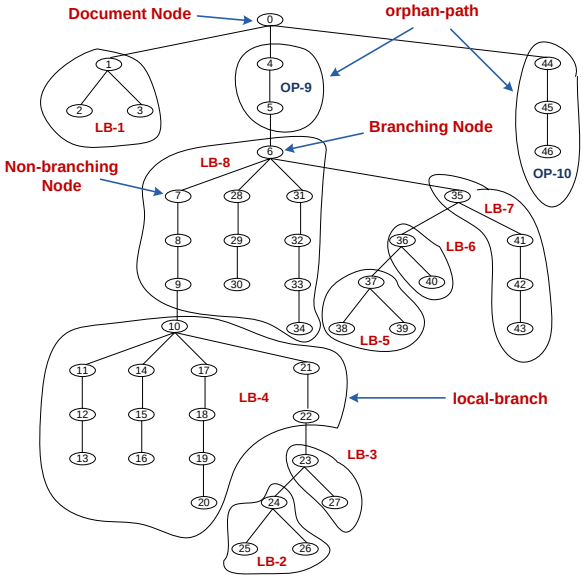

Figure 5: Assigning Nodes to Partitions

5.2 Reducing Large Partitions

Although local-branches are rooted subtrees, they may contain nodes that do not have an ancestor or descendant relationship. A separation of nodes that do not have a hierarchical association leads to an optimized pruning effort.

Each local-branch instance has a single branching node *root* which may have many (non-branching node) descendants. It is the non-branching descendants of the root that are examined to determine if they share a hierarchical association. For this reason, we partition the non-branching nodes (in each local-branch) into disjoint path-branches (Definition 3). As orphan-paths and local-branches are *disjoint*, each of these path-branch instances will be a *branchlink-path*

(Definition 5) or a *leaf-path* (Definition 6).

The initial partitioning method was introduced to describe how partitions are formed. In reality, we employ a refined version of this algorithm, `RefinePartitions` (algorithm 2), for creating BranchIndex partitions. The new branch partitions are created by processing *two* local-branches simultaneously. Now, *current nodes* up to and including the first branching node, are placed in the first *waiting list* (*wList1*) where they *wait* to be indexed. Subsequently, the next set of current nodes, up to and including the second branching node, are placed on the second waiting list (*wList2*). At this point, *wList1* and *wList2* contain the nodes that comprise the first and second local-branches respectively.

Algorithm 2 RefinePartitions

```
1:  if node at level 1 encountered then
2:      move nodes that comprise wList2 to orphan-path;
3:  end if
4:  move non-branching nodes from wList1 to leaf-path;
5:  for each node n in wList2 do
6:      if n = ancestor of wList1.ROOT ∧ n ≠ branching node then
7:          move n to branchlink-path;
8:      else if n ≠ ancestor wList1.ROOT then
9:          move n to leaf-path;
10:     end if
11: end for
12: move local-branch from wList1 to local-branch;
13: move local-branch from wList2 to wList1;
```

If a node at *level 1* is encountered, the nodes that comprise *wList2* are an orphan-path (*line 2*). If a *branchlink-path* (Definition 5) exists, `RefinePartitions` identifies it as the non-branching nodes in *wList2* that are *ancestors* of the root node in *wList1* (*lines 6-7*). If one or more *leaf-paths* (Definition 6) exist, they will be the nodes in *wList2* that are not ancestors of root node in *wList1* (*lines 8-9*). The remaining nodes that comprise the first local-branch (*wList1*) are then moved to the index (*line 12*). This will be the *single* branching node root of the first local-branch only.

At this point, the only node that remains in *wList2* is the root node of the second local-branch. This local-branch is then *moved* to *wList1* (*line 13*) and *wList2* is now empty. The next local-branch is placed in *wList2* and the process is repeated until no branches remain. Using this algorithm, a lot more partitions are created and this should lead to increased pruning during query processing. The refinement is illustrated in Fig 6.

The process will also track the *ancestor-descendant* relationships between branch partitions. This is achieved by maintaining the *parent-child* mappings between branches. Given two branches `B1` and `B2`: `B2` is a child of `B1` *if and only if* the *parent* node of a node that comprises `B2` belongs to `B1`. When the `RefinePartitions` process is complete, the *ancestor-descendant* relationships between branches are determined using a recursive function across these parent-child relationships.

5.3 Partition Classes

The indexing process results in a large number of branch partitions. This benefits the optimization process as it facilitates a highly aggressive pruning process and thus, reduces the inefficient stage of node comparisons. However, this type of pruning requires smaller partitions with the effect of a larger index.

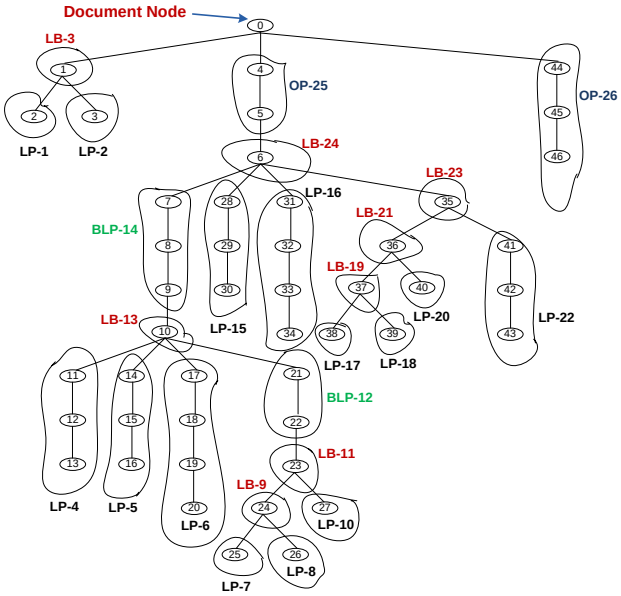

Figure 6: After Partition Refinement

The final phase in constructing the partition index is to reduce its size while maintaining the same degree of pruning. To achieve this, we use a classification process for all branches based on root to leaf structure of the partition.

Every branch instance can belong to a *single* branch class. A process of classifying each branch will use the structure of the branch instance and its relationship to other branch instances as the matching criteria. Additionally, in order to belong to the same class, each branch instance must have an identical *set* of descendant branches. The latter is required to ensure that there is no overlap between branch classes.

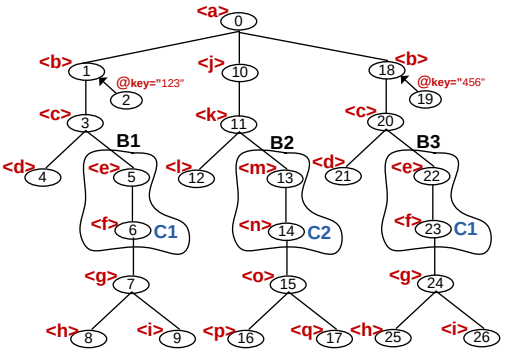

Figure 7: XML Tree Showing Branch Classifications

Order	Class C1	Class C2
1	a	a
2	a/b{@key}	a/j
3	a/b/c	a/j/k
4	a/b/c/e	a/j/k/m
5	a/b/c/e/f	a/j/k/m/n

Table 5: Branch Classes

To explain this concept, we use figure 7 which contains three branch instances, `B1`-`B3`. Table 5 shows the *extended* DataGuides associated with branch classes: `C1` and `C2`. Note that the *order* of the Class type associated with each branch class is important. After classification, as `B1` and `B3` have an identical set of descendant branch instances, they will be instances of the `C1` class, while branch `B2` is an instance of the `C2` class.

Finally, the process that maintains parent-child relationships between branch instances (discussed earlier), must be replaced with one that maintains parent-child relationships between branch classes. The ancestor-descendant relationships are then generated for branch classes with the same approach as that used for branch instances.

5.4 Dataset Statistics

To add context to our experimental evaluation, we provide the statistics for our index after processing the 2.06GB bicycle rental repository.

- Size = 2.06GB
- Class Count = 1,067
- Node Count = 85,965,102
- NCLT Entries = 1,224
- Class Count = 5,032

The number of tuples in the `CLASS` relation is greater then the number of branch classes identified for that dataset: the *bicycle rental* dataset has 1,067 branch classes and the number of tuples in the class index is 5,032. The deliberate decision to duplicate selected branch classes allows the ancestor or descendant branch classes associated with a set of context nodes to be identified using *equijoins*, which have been shown to be more efficient than *non-equijoins* (Luoma 2007b).

The `NODE` relation shows the number of nodes in the dataset to be in excess of 85 million which provides a challenge for any XML query processor. The `NCLT` relation contains a tuple for each distinct *name*, *class*, *level* and *type* identified in the `NODE` relation. As the `NCLT` relation is used instead of the `NODE` relation (where possible), the number of tuples in the `NCLT` relation is significantly smaller than the number of tuples in the `NODE` relation which boosts query performance.

6 Experiments and Evaluation

In previous work (Marks and Roantree 2010), we demonstrated that the approach most similar to ours (Luoma 2007a) and a traditional node based approach to XPath (Grust et al. 2007) were not capable of scaling to the volumes of data required in this target application. Thus, in this current paper, we compare our *BranchIndex* to the approach described by *Georgiadis et al* in (Georgiadis and Vassalos 2006) (a recent DataGuide approach) - we will refer to this approach as the *PathIndex*. In addition, we evaluate the performance of two vendor systems: *MonetDB/XQuery* (Boncz et al. 2006) and *SQL Server 2008*.

MonetDB/XQuery is a leading open-source XML database; we evaluate this approach because it was evaluated by *Georgiadis et al* in (Georgiadis and Vassalos 2006) (i.e. the *PathIndex*). *SQL Server* was chosen as it uses the optimization techniques described in (Pal et al. 2004), which were subsequently discussed by *Grust* in (Grust et al. 2007).

All experiments were run on identical servers with a 2.66GHz Intel(R) Core(TM)2 Duo CPU and 4GB of RAM. The *BranchIndex* and *PathIndex* were deployed in an *Oracle 11g* relational database. *Oracle 11g* and *MonetDB/XQuery version 4.34.4* were both deployed on *Fedora 12 Linux* (64bit) platforms; SQL Server was deployed on a *Windows 7* (64bit) platform. A small sample of one of the XML documents is shown in figure 2.

Across the vendor systems, we call the (XQuery) `count()` function to ensure that any overhead associated with *document reconstruction* (Chebotko et al. 2007) is not included in the query response times - this approach was also used in (Georgiadis and Vassalos 2007) for evaluating the comparative query response times of vendor systems. Similarly - to provide a balanced evaluation - for the *BranchIndex* and the *PathIndex*, we called the (SQL) `count()` function on the `PRE` column of the `NODE` relation to count the result nodes.

Finally, each XPath query was executed eleven times. In each case, the first query execution was ignored (to ensure *hot cache* response times) and the remaining ten queries were averaged to provide the final result in milliseconds. A *timeout* of ten minutes was placed on each query to allow us to perform the evaluation in a reasonable amount of time - each query that took longer than ten minutes is marked as: *>10mins*.

6.1 Query Classification

Experimental analysis was based on the end user queries listed earlier in §2. We classified queries into five different types to help evaluate our approach to optimization.

- **QC_1:** *Non Text Node Queries (without predicates)*. The query does not evaluate a text node nor does it contain predicate filters.
- **QC_2:** *Non Text Node Queries (with predicates)*. This query does not evaluate a text node but contains one or more predicate.
- **QC_3:** *Low Cardinality Text Node Queries*. The query evaluates text node(s) that have *low selectivity*.
- **QC_4:** *High Cardinality Text Node Queries*. The query evaluates text node(s) that have *high selectivity*.
- **QC_5:** *Single Step Path Fragment Queries* - this type of query does not contain a primary path fragment (PPF), as defined by *Georgiadis* in (Georgiadis and Vassalos 2006), that spans more than one step in the XPath expression.

If a column in a relation has *low cardinality* (e.g. gender, genre), the number *values* that have identical character content will be high. Thus, queries predicated on these columns usually return a large number of tuples, i.e. they have *low selectivity* (Marks and Roantree 2009). In contrast, if a column in a relation has *high cardinality* (e.g. name, title), the number of *values* that have identical character content will be low; thus, queries predicated on these columns usually return a small number of tuples, i.e. they have *high selectivity* (Marks and Roantree 2009). Columns that have *high cardinality* provide the best performance for B-tree indexes.

Text nodes, in the *BranchIndex* and the *PathIndex* are stored as the attribute: *value* in the `NODE` relation (Table 2). It will become clear throughout this section, that the *selectivity* of these text nodes is important when describing the best and worse case queries for our *BranchIndex* - as text nodes were not considered during the branch classification process to keep the size of the *BranchIndex* small, and therefore optimized.

Query	Type	BranchIndex	PathIndex	MonetDB	SQLS	Result Nodes
Q01	QC_1	1,395ms	3,110ms	5,502ms	258ms	1,411,451
Q02	QC_1	1,193ms	1,070ms	3,010ms	166ms	634,320
Q03	QC_2	140ms	>10mins	18,840ms	>10mins	191,680
Q04	QC_2	282ms	>10mins	16,766ms	>10mins	191,680
Q05	QC_3	87,385ms	157,090ms	11,890ms	>10mins	13,046
Q06	QC_3	>10mins	>10mins	108,192ms	>10mins	429,585
Q07	QC_3	96,143ms	>10mins	107,629ms	>10mins	145
Q09	QC_4	302ms	140ms	>10mins	1ms	1
Q10	QC_4	164ms	68,690ms	>10mins	166ms	642
Q08	QC_5	3,379ms	>10mins	105,129ms	error	6,529,626

Table 6: Results for the Bicycle Rental Dataset

6.2 Results and Evaluation

The results of these queries are shown in Table 6. `BRANCHI`, *`PATHI`*, `MDB`, and `SQLS` show, for each query, the time taken (in milliseconds) by the *BranchIndex*, *PathIndex*, *MonetDB/XQuery*, and *SQL Server 2008* respectively.

- Query Type $\mathbf{QC}_1$. For queries **Q01** and **Q02**, the *BranchIndex* and *PathIndex* both perform well. For query **Q01**, the *PathIndex's* system will access the `PATHS` relation using a regular path expression to identify all *path identifiers* associated with the path: `/bikes/city/Nantes/stations/station`. The `NODE` relation is subsequently accessed to determine the nodes associated with these paths and thus, form the result set. Our BranchIndex processor accesses the `NCLT` relation for the first step (`/bikes`); the pair (`NCLT`, `CLASS`) for steps two, three and four. Finally, the `NODE` index is accessed for step five to generate the result set.

  Thus, for $\mathbf{QC}_1$ queries, neither *PathIndex* nor *BranchIndex* systems require inefficient node comparisons as they access the `NODE` relation only to locate the result nodes (the final step). However, the *BranchIndex* performs better than the *PathIndex* for queries in this category as the number of tuples in the `NCLT` and `CLASS` indexes is small (see *Dataset Statistics* in the previous section). MonetDB and SQL Server also perform well for queries in this category. We believe this is due to MonetDB/XQuery's Staircase Join algorithm and SQL Servers's (*secondary*) `PATH` index respectively.

- Query Type $\mathbf{QC}_2$. For these queries, the *PathIndex* system must access the `NODE` index to perform a join between each *primary path fragment* (PPF) (Georgiadis and Vassalos 2007). Query **Q03** has two primary path fragments: *//city//stations* and */station/available*, whereas query **Q04** has four PPF's. Thus, there is one structural join (based on individual node comparisons) required in **Q03** and four in **Q04** and this leads to a significant performance overhead.

  In contrast, the *BranchIndex* still accesses the `NODE` relation just once to retrieve the result set - again inefficient node comparisons are not required in this approach. Therefore, the *BranchIndex* outperforms *PathIndex* (and *MonetDB/XQuery*) by *orders of magnitude* for queries in category $\mathbf{QC}_2$, i.e see queries **Q03** and **Q04** (Table 6).

- Query Type $\mathbf{QC}_3$. These queries highlight the sole weakness in our approach. The reasons are: the request for text nodes, e.g. *'0'*, requires that our system accesses the `NODE` index; and the *low selectivity* of these text nodes requires a large number of nodes to be evaluated using inefficient *pre/post* node comparisons. In fact, **Q06** took the *BranchIndex* longer than ten minutes to return the result. The *PathIndex* also performs poorly for this category of queries as it also requires a large number of node comparisons between PPFs. *MonetDB/XQuery* performs best overall for category $\mathbf{QC}_3$ queries, which is most likely due to the *Staircase Join* (Grust et al. 2003) algorithm. However, it is not particularly efficient, e.g. **Q07** and **Q08** took more than a minute each, and **Q09** took more than ten minutes. SQL Server did not answer any $\mathbf{QC}_3$ query inside the threshold ten minutes.

- Query Type $\mathbf{QC}_4$. Here, the high selectivity of the text nodes ensures that even though it is necessary to access the `NODE` index (because of the text nodes), the text nodes have high selectivity leading to fewer *pre/post* node comparisons for the *BranchIndex*. For the same reason, *PathIndex* performs well in this category. We do not know why *MonetDB/XQuery* performs poorly for these queries (i.e. **Q09** and **Q10**). We believe this could be bug-related as we anticipated a reasonable performance from MonetDB.

- Query Type $\mathbf{QC}_5$. The *PathIndex's* primary path fragments cannot optimize queries in category $\mathbf{QC}_5$. For example, **Q08** has three primary path fragments, each of which spans just one XPath step. The `PATHS` relation, used by *PathIndex* to process multiple XPath steps simultaneously, is redundant if each PPF spans just one step each; thus, no optimization is achieved. Only our BranchIndex and MonetDB managed to return results for this category.

7 Conclusions

In this paper, we presented an architecture for harvesting web sensor data. This data was enriched and then indexed to facilitate high level XPath queries used to determine station usage patterns. This architecture was used to bridge the considerable gap between end-user requirements and the data available from web service providers. Even with enrichment of sensor data, the performance using existing XML solutions was not adequate or even possible for query processing. We also presented our system of document partition indexing which is shown to outperform other approaches for four of the five query classifications.

Our current focus is on fine-tuning our index to improve results on the poorly performing $\mathbf{QC}_3$ queries. We are also building extensions to our XPath query interface to incorporate data mining primitives for more complex query types.

References

M. Balazinska, A. Deshpande, M.J. Franklin, P.B. Gibbons, J. Gray, M. Hansen, M. Liebhold, S. Nath, A. Szalay, and V. Tao. Data management in the worldwide sensor web. In *IEEE Pervasive Computing*, pages 30–40, 2007.

M. Banko, M.J. Cafarella, S. Soderland, Broadhead, and O. M., Etzioni. Open information extraction from the web. In *Proceedings of IJCAI*, pages 2670–2676, 2007.

Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 479–490, New York, NY, USA, 2006. ACM.

C-H. Chang, M.R. Girgis, M. Kayed, and K.F. Shaalan. A survey of web information extraction systems. In *IEEE Transactions on Knowledge and Data Engineering*, pages 1411–1428, 2006.

Artem Chebotko, Mustafa Atay, Shiyong Lu, and Farshad Fotouhi. XML subtree reconstruction from relational storage of XML documents. *Data Knowl. Eng.*, 62(2), pages199–218, 2007.

Devlic A., Koziuk M., and Horsman W. Synthesizing Context for a Sports Domain on a Mobile Device. *3rd European Conference on Smart Sensing and Context (EuroSSC)*, LNCS vol. 5279, pages 206–219, 2008.

M. Erwig. Toward the automatic derivation of xml transformations. 2003.

Haris Georgiadis and Vasilis Vassalos. Improving the Efficiency of XPath Execution on Relational Systems. In *EDBT*, pages 570–587, 2006.

Haris Georgiadis and Vasilis Vassalos. XPath on Steroids: Exploiting Relational Engines for XPath Performance. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 317–328, New York, NY, USA, 2007. ACM.

Torsten Grust. Accelerating XPath Location Steps. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120, New York, NY, USA, 2002. ACM.

Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch Its (axis) Steps. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 524–535. VLDB Endowment, 2003.

Torsten Grust, Jan Rittinger, and Jens Teubner. Why off-the-shelf RDBMSs are better at XPath than you might expect. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 949–958, New York, NY, USA, 2007. ACM.

C.A. Knoblock, K. Lerman, S. Minton, and I. Muslea. Accurately and reliably extracting data from the web: A machine learning approach. In *IEEE Data Engineering Bulletin*, pages 275–287, 2003.

H.C. Liu, J. Zeleznikow, and H. Jamil. Logic-based association rule mining in xml documents. In *Advanced Web and Network Technologies, and Applications*, pages 97–106, 2006.

J. Liu, H.H. Park, M. Vincent, and C. Liu. A Formalism of XML Restructuring Operations. IN *The Semantic WebASWC*, pages 126–132, 2006.

Jiaheng Lu, Ting Chen, and Tok Wang Ling. TJFast: Effective Processing of XML Twig Pattern Matching. In *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 1118–1119, New York, NY, USA, 2005. ACM.

Olli Luoma. Efficient Queries on XML Data through Partitioning. In *WEBIST (Selected Papers)*, pages 98–108, 2007.

Olli Luoma. Xeek: An Efficient Method for Supporting XPath Evaluation with Relational Databases. In *ADBIS Research Communications*, 2006.

McCann D. and Roantree M. A Query Service for Raw Sensor Data. In *Proceedings of 4th European Conference on Smart Sensing and Context* (EuroSSC), LNCS vol. 5741, Springer, pp. 38-50, 2009.

G. Marks and M. Roantree. Metamodel-based Optimization of XPath Queries. In *26th British National Conference on Databases*, pages 146–157, 2009.

Gerard Marks and Mark Roantree. Classification of Index Partitions to boost XML Query Performance. In *29th International Conference on Conceptual Modeling Vancouver, BC, Canada*, 2010. To Appear.

R. Nayak, R. Witt, and A. Tonev. Data mining and xml documents. In *International Conference on Internet Computing*, pages 660–666, 2002.

Patrick O'Neil, Elizabeth O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 903–908, New York, NY, USA, 2004. ACM.

Shankar Pal, Istvan Cseri, Oliver Seeliger, Gideon Schaller, Leo Giakoumakis, and Vasili Zolotov. Indexing XML data stored in a relational database. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1146–1157. VLDB Endowment, 2004.

L. Rusu, W. Rahayu, and D. Taniar. Mining association rules from xml documents. In *Web data management practices: emerging techniques and technologies*, pages 79–102, 2007.

J.W.W. Wan and G. Dobbie. Mining association rules from xml data using xquery. In *Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, pages 169–174, 2004.

F. Wang, S. Liu, P. Liu, and Y. Bai. Bridging physical and virtual worlds: Complex event processing for rfid data streams. In *Advances in Database Technology-EDBT*, pages 588–607, 2006.

Whitehouse K., Zhao F., and Liu J. Semantic Streams: a Framework for Composable Semantic Interpretation of Sensor Data. In *3rd European Workshop on Wireless Sensor Networks (EWSN)*, LNCS 3868, pages 5–20, 2006.

J.X. Wong K.-F. Li J. Tang, N. Yu. Fast XML Structural Join Algorithms by Partitioning. *Journal of Research and Practice in Information Technology*, 40:33–54, 2008.