THE NATIONAL UNIVERSITY
of SINGAPORE


School of Computing
Computing 1, Singapore 117590

**TRA9/07**

# *Twig'n Join: Progressive Query Processing of Multiple XML Streams*

***Wee Hyong TOK, Stephane BRESSAN and Mong-Li LEE***

*September 2007*

# Technical Report

## Foreword

*This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.*

OOI Beng Chin
Dean of School

# Twig'n Join: Progressive Query Processing of Multiple XML Streams

Wee Hyong Tok, Stéphane Bressan, and Mong-Li Lee

School of Computing
National University of Singapore
{tokwh,steph,leeml}@comp.nus.edu.sg

**Abstract.** We propose a practical approach to the progressive processing of (FWR) XQuery queries on multiple XML streams, called Twig'n Join (or TnJ). The query is decomposed into a query plan combining several twig queries on the individual streams, followed by a multi-way join and a final twig query. The processing is itself accordingly decomposed into three pipelined stages progressively producing streams of XML fragments. Twig'n Join combines the advantages of the recently proposed TwigM algorithm and our previous work on relational result-rate based progressive joins. In addition, we introduce a novel dynamic probing technique, called Result-Oriented Probing (ROP), which determines an optimal probing sequence for the multi-way join. This significantly reduces the amount of redundant probing for results. We comparatively evaluate the performance of Twig'n Join using both synthetic and real-life data from standard XML query processing benchmarks. We show that Twig'n Join is indeed effective and efficient for processing multiple XML streams.

**Key words:** XML, Progressive Join

## 1 Introduction

The ubiquity of network accessible XML data necessitates the design of XML query processors which can process complex queries over multiple XML data streams. For example, expressive RSS aggregators e.g. Yahoo Pipes [1], Danaides [2]) require support for effective and efficient processing of complex queries. Thus, we need to devise XML query processors for XML languages such as XPath or XQuery that supports the processing of structural and predicate constraints as well as join queries [3] over multiple XML data streams. In order to ensure a good user experience, the XML query processors must deliver initial results quickly, and maintain a consistent high result throughput. Main memory is limited and when it is full, data needs to be flushed to disk. As we need to produce results progressively with a high throughput, we need to effectively manage the XML data that is kept in memory and favor data that is most likely to contribute to the result. A key insight is to make use of statistics from either the input (i.e.

data) or output (i.e. result) distributions. The problem of effective management of in-memory data was first studied in [4, 5] for relational equijoins. However, to our knowledge, no work exists for progressive XML processing.

Consider the following motivating scenario. A user is interested to know the latest news based on his blog entries. This is achieved by comparing the tag of the blog entries and the keyword for the news entries. Both the news and the blog entries are made available as RSS feeds (i.e. XML streams). In order to combine the entries from the blog and the news entries, we can make use of a join between the new and blog XML streams. This can be expressed as the following XQuery query.

```
(for $s in doc("news.xml")//item
 for $q in doc("blogs.xml")//entry
 where $q/tag=$s//techNews/keyword
   and contains($s/title, "CNA")
 return
   <resultTuple>
     {($s/blurb), ($s/article), ($q/entryId) }
   </resultTuple>)
```

In this work, we propose a practical approach, called Twig'n Join (TnJ), to the progressive processing of XQuery queries on multiple XML streams. A (FWR) XQuery query is first decomposed into a query plan which consists of twig queries over the input streams, followed by a multi-way join and a final twig query.

TnJ processes the query plan on-the-fly and delivers the results progressively. The novelty of this approach compared, to conventional XQuery processing, lies in the decomposition of the XQuery queries into several independent components. This reduces the complexity for the design of XQuery query processing algorithms. We also introduce a dynamic probing technique, called Result-Oriented Probing (ROP), which is to effectively determine an optimal probing sequence for the multi-way join. This significantly reduces the amount of redundant probing for results.

Using both synthetic and real-life data from standard XML query processing benchmarks, we comparatively evaluate the performance of the Twig'n Join variants. We show that Twig'n Join, with a result-rate based approach is indeed effective and efficient.

The rest of the paper is organized as follows: In section 2, we discuss related work. In Section 3, we propose Twig'n Join, a progressive join algorithm for processing multiple XML data streams. We conduct an extensive evaluation in Section 4. We conclude in Section 5.

## 2 Related Work

Concrete XML query languages, such as XPath and XQuery, express both structural and predicate constraints on the XML document/stream. One good rep-

resentation of the structural constraints is twig queries. A twig query is a tree-pattern query that specifies the structural relationships (parent/child or ancestor/ descendant) between the nodes. Many existing XML query processing techniques focused on the efficient processing of twig queries. For predicate constraints, we focus on join predicates.

We classify existing XML query processing techniques by considering the following factors: (1) Non-streaming vs Streaming and (2) Handle single vs multiple XML documents/streams. We omit the discussion on non-streaming techniques which process multiple XML document/streams as we are unable to fit existing techniques into this category.

### 2.1 Non-streaming and Single XML document

The non-streaming techniques [6–8] processes disk-resident XML data. These techniques focused on the efficient processing of twig queries. The assumption made by these techniques is that a labeling scheme is available. The labeling scheme encodes the structural relationships within the XML documents. Common labeling schemes that have been used include Region [6] and Dewey-based [9, 10] encoding. The non-streaming algorithms rely on these encodings to efficiently answer the queries. In order to compute the results, the algorithms need to wait for all the intermediate results to be produced before the results of the twig queries can be computed. Due to the need for prior labeling of the XML data and the need to wait for all the intermediate results to be produced before results are available, these techniques are not suitable for processing XML data streams.

Non-streaming techniques [11–13] for processing XQuery have also been proposed. In [11], a transducer-based XML Query Processor translates XQuery to an intermediate form, known as XML Stream Machine (XSM). XSM is then translated into C code which is compiled and executed. [12] transforms XQuery into a Tree-Logical Class (TLC) algebra expression, which is then used as the basis for evaluating the XQuery query.

### 2.2 Streaming and Single XML document

Many streaming techniques [14, 11, 15–18] have been proposed for processing XPath and XQuery queries. In [11], the BEA/XQRL processor was proposed to support pipelined execution by using an iterator model over the data stream. [17] proposed transformation techniques to enable XQuery queries to be evaluated in one-pass. In addition, [17] proposed code generation techniques (from the XQuery queries) to handle user-defined aggregates and recursive functions. [18] proposed the *TwigM* machine, an efficient non-blocking method for evaluating twig queries over XML data streams. *TwigM* assumes an input sequence of SAX events (i.e. startElement, endElement), and uses a stack-based structure to compactly encode the solutions to the Twig join. The output consists of XML fragments. None of these techniques considered XML query processing over multiple XML data streams.

### 2.3 Streaming and Multiple XML documents/streams

[3] proposed a Massively Multi-Query Join Processing (MMQJP) technique for processing value joins over multiple XML data streams. Similar to our approach, MMQJP consists of two phases: XPath Evaluation and Join Processing phase. In the XPath evaluation phase, the XML data streams are matched and auxiliary information stored as relations in a commercial database management systems (DBMS) - Microsoft SQL Server. The auxiliary information are then used during the join processing phase for computing results. Thus, MMQJP can only deliver results when the entire XML documents have arrived. In addition, MMQJP have no control over the flushing policy due to its dependence on the commercial DBMS. In contrast to MMQJP, our proposed technique delivers results progressively as portions of the streamed XML documents arrived.

A physical algebra for XQuery was proposed in [19]. The algebra allows XML streaming operators to be intermixed with conventional XML and relational operators in a query plan. This allows pipelined plans to be easily defined. [19] do not consider memory management issues.

### 2.4 TwigM and RRPJ

In [18], the TwigM machine was proposed for processing XPath queries on a single XML stream. As compared to conventional holistic twig processing approaches which stores intermediate results, TwigM produces results on-the-fly. The authors did not demonstrate how the proposed technique can be generalize for join processing over multiple XML streams in a memory-constrained environment. In [5], the RRPJ method was proposed for maximizing the result throughout for the progressive relationa equi-join problem. Though [5] can be generalize for various data models, the authors do not show how it can be applied to handle XML data. The results presented in [5] considered the performance of the progressive relational equi-join for both uniformity and non-uniformity of data within the hash partitions, as well as varying data distribution. Our work differs from [5] in two aspects. Firstly, RRPJ uses numeric join attributes, whereas TnJ uses string attributes. Secondly, RRPJ deals directly with the source streams, whereas TnJ's inputs consists of intermediate XML fragments produced by the TwigM machine. In our work, we combined the advantages of using TwigM and RRPJ method for progressive query processing on multiple XML streams.

## 3 Twig'n Join

In this section, we discuss a practical approach to the progressive processing of (FWR) XQuery queries on multiple XML streams, called Twig'n Join (TnJ).

Given two XML data streams, R and S, where the XML data are delivered tag by tag from remote data sources. Twig pattern (extracted from the XQuery query) $T_r$ and $T_s$ are defined for R and S respectively. XML result fragments $F_r$ and $F_s$ are produced for portions of the XML documents that matches $T_r$

and $T_s$ respectively. The user define a set of join attributes A in which the XML fragments can be joined. A result $<F_r, F_s>$ is reported if $F_r$ and $F_s$ fulfill the join attribute condition defined by A. Our goal is to be able to progressively deliver the result.

A FWR XQuery query can be decomposed into three parts: (1) Structural filtering on the input streams (2) Predicate Processing and (3) Structural filtering on the results. We assume that a XQuery pre-processor will parse the (FWR) XQuery expression and generate a query plan. During predicate processing, we perform value-based filtering as well as process the joins between the input streams. We focus on join processing. Figure 1 shows a possible query plan for Scenario B. We note that further optimization of the query plan is possible. However, we consider query optimization as an orthogonal issue, and do not explore it in this paper.

The query plan consists of several twig queries on the individual XML streams, followed by predicate processing and a final twig query. XML data (news.xml and blogs.xml) are continuously streamed from remote sites. The data is then matched using the two twig matching operators ($TM_A$ and $TM_B$). The output from $TM_A$ and $TM_B$ (XML fragments) are then joined using a join operator (i.e. predicate processing).

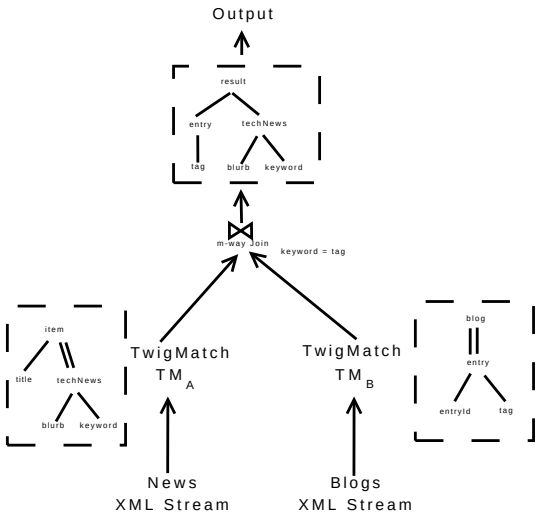



**Fig. 1.** Query Execution Plan

We use the TwigM machine [18] to efficiently perform the twig matches on the streaming XML data, and a hash-based join for joining the data. When intermediate XML fragments are continuously produced by the twig matching operators, the memory might become full. Whenever memory is full, we will need to flush some of these XML fragments to disk so that they can be joined

at a later stage, or whenever both data streams block. In this work, we focus on maximizing the results from the XML fragments that are retained in memory. The Result-Rate based approach [20, 5] is used to determine the XML fragments to be flushed to disk whenever memory is full.

### 3.1 Algorithms

**Algorithm 1:** Multi-way Twig'n Join Algortihm

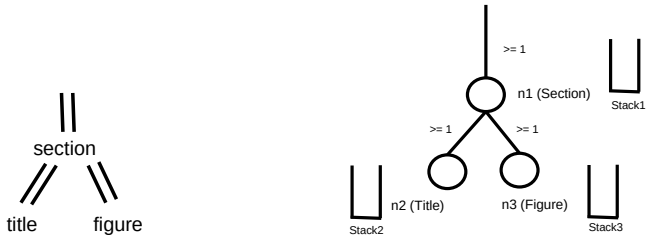
```
   Data   : n XML Data Streams - S_i,
            Twig Queries - T_i, where 0 ≤ i < n
   Result : R, Results
   begin
1  |  for (i=0; i < n; i++) do
2  |  |  TwigMachine TM_i = CreateTwigM_Machine(T_i,S_i) ;
3  |  |  HashPartition Ht_i = CreateHashPartitions() ;
   |  |
4  |  while ( XML fragments are available ) do
5  |  |  xmlfrag = Select(S) ;
   |  |
6  |  |  MultiWayJoin(xmlfrag) ;
7  |  |  Ht_src.insert(xmlfrag) ;
   end
```

Algorithm 1 shows the details. In Line 1 to 3, we create a TwigM machine and a hash partition for each of the XML data streams. . The TwigM machines exposes an iterator-style (i.e. *getNext()* ) interface in order for the TnJ algorithm to continuously get the next XML fragments that have been matched using $T_i$ ( $0 \leq i < n$, where n denotes the number of XML data streams). In Line 5, the *Select()* checks the availability of XML fragments from the various TwigM machines. In Line 6, the XML fragment is used in a multi-way join on the remaining *n - 1* hash partitions. Whenever memory is full, some of the XML fragments in the hash partitions will be flushed to disk. This is checked prior to the insertion of the XML fragment into the corresponding hash partitions (Line 7). Algorithm 3 describes how the insertion and flushing is done. Then, the XML fragment is inserted into its own partition. We make use of the Berkeley-DB [21] hash function for computing the hash value for each XML fragment.

Without loss of generality, we first describe how Twig'n Join processes two XML data streams. We then show how we can generalize this for handling multiple data streams using the multi-way join operator. Given the Twig query Q, a TwigM machine, M, is created (Figure 2(b)). M consists of machine nodes $n_i$. For each node $n_i$, there is an edge $e_i$ which connects it to its parent node ( $1 \leq i \leq |Q|$, where $|Q|$ refers to the number of tags specified in the query ). Depending on

whether it is a parent-child or ancestor-descendant relationship specified in the query, the edge is annotated with 1 (parent-child) or ≥1 (ancestor-descendant). For example, in Figure 2(b), we can see that the edges are all annotated with ≥1. This corresponds to the query Q. In addition, a stack is associated with each of the machine nodes. For a non-leaf machine node, an entry of the stack is a triple <N, C, B>, where N refers to a XML tag that is inserted, C is a candidate solution list, and B is a boolean array. For a leaf machine node, an entry of the stack only consists of just < N>. For a node with $b$ children, B consists of $b$ boolean variables. Initially, the $b$ boolean variables are all initialized to be false. M is then used to process the twig queries and deliver the results (in the form of XML fragments) whenever a match occurs.

Whenever portions of an XML documents satisfy the structural constraint expressed by the twig query, the TwigM machine outputs the results as XML fragments. The results are output whenever the structural constraints are met. Hence, the XML fragments can be delivered progressively for join processing.




(a) Twig Query, Q (b) TwigM Machine, M

**Fig. 2.** Twig Query and TwigM Machine Example

XML fragments from the twig matching on the multiple XML streams are fed to the hash-based progressive join. Without loss of generality, we first describe the processing of a binary join on the XML fragments produced. We make use of the generic Result Rate-based flushing (RRPJ) technique used in [5]. During join processing, RRPJ is used to determine the XML fragments to be flushed to disk whenever memory is full. An important characteristic of RRPJ is that by using statistics based on the result output statistics, it can be generalized gracefully for many data models (as shown in [20](spatial data) and [22](high-dimensional data ) ).

**Probing (Algorithm 2)** Whenever a new XML fragment, $f_d$, arrives, we first use it to probe the corresponding hash partition from the other data stream (Line 1). Based on the join predicates defined, we check each of the XML fragments found in the partition (Line 3-6). Results are output whenever the join predicates are satisfied. In addition, a counter, $numResults$, keeps track of the results produced by each of the partitions. The counter is updated when all the results have been produced (Line 7).

**Algorithm 2:** Probing

```
   Data   : f_d - Newly Arrived XML Fragment
   Result : R, Results
   begin
1  |  p = findPartition(f_d) ;
2  |  numResults = 0 ;
3  |  for ( XMLFragment f in p) do
4  |  |  if ( f and f_d satisfies the join predicate ) then
5  |  |  |  R = R ∪ (f_d,f) ;
6  |  |  |  numResults++ ;
7  |  Update statistics for p ;
8  |  Return R;
   end
```

**Insertion and Flushing (Algorithm 3)** The hash value for an XML fragment is computed. The XML fragment is then inserted into its corresponding hash partition (Line 1). Whenever memory is full, the $FlushDataToDisk()$ routine flushes some of the in-memory XML fragments to disk. The number of XML fragments to be flushed is determined by a user-defined parameter, $NumFlush$.

**Algorithm 3:** Insertion and Flushing

```
   Data   : f_d - Newly Arrived XML Fragment
   begin
1  |  p = findPartition(f_d) ;
2  |  if ( memory is full() ) then
3  |  |  FlushDataToDisk();
4  |  Insert f_d into p ;
   end
```

In order to determine which partitions to be flushed, each of the $i$th hash partitions ( $1 \leq i \leq n$, where n is the total number of partitions), maintains a counter measuring its potential to produce results. This determines the partitions to be flushed.

We first present a naive extension of RPJ for determining the XML fragments to be flushed to disk whenever memory is full. In the naive extension to RPJ for XML (called Twig-RPJ), we keep track of the the $RPJ$ value - number of XML fragments in a partition divided by the total number of XML fragments that have arrived. The partner partition (that is the matching partition in the other streams) to the partition with the smallest values is flushed.

We also make use of the Result-rate based Join (RRPJ) flushing technique described in [5]. When making use of the Result Rate-based Flushing (RRPJ), we keep track of the $Th_i$ value. In RRPJ, the $Th_i$ value is an estimate of the

productivity of the i-th partition (with $1 \leq i \leq n$, where $n$ is the total number of partitions used to store the XML fragments).

$$Th_i = \frac{Ri}{N_i} \tag{1}$$

where $R_i$ and $N_i$ denotes the total number of results produced, and the total number of XML fragments for the i-th partition respectively. Twig'n Join flushes the partitions with the smallest $Th_i$ values, until a user-defined number of tuples to be flushed is reached.

### 3.2 Multi-way Join

In this section, we discuss how we can generalize Twig'n Join for processing XQuery queries on multiple XML streams. Each XML fragment, produced by the TwigM machine, consists of the XML data and a bitmap (i.e DoneBitmap) that is used to determine whether the XML fragment has been used to probe the other partitions. *DoneBitmap* consists of $n$ bits. Figure 3 shows the structure of the XML fragment. When the XML fragment first arrives, the bit corresponding to each own partition is set to 1. Whenever the XML fragment is used to probe the hash partitions for the other XML streams, the bit corresponding to the hash partition is set to 1 when it can be used to join with at least one other XML fragment in the partition. It is set to 0 otherwise. When all the bits of the *DoneBitmap* are set, the XML fragment is output as a result. In our work, we consider only the case where the join predicate is the same for all the XML streams.


**Fig. 3.** XML Fragment Structure

Existing multi-way join techniques for relational equi-join, such as MJoin [23], can be used as to handle the multi-way between the XML fragments that are produced. The performance of the multi-way join is dependent on the probing sequence. For example, MJoin sorts the hash partitions based on their respective join selectivity. The key intuition is that by probing partitions with a low join selectivity first, it filters away tuple that will not generate any result early. This helps to reduce the number of unnecessary probes to the remaining un-probed hash partitions. However, it is difficult to determine the join selectivities if the inputs to the multi-way join consists of intermediate results from a pipelined process. For example, the XML fragments are produced by the TwigM machines. Even if the join selectivity of the join attribute for the base XML streams can be accurately determined, it is not straightforward to determine the join selectivity of the intermediate XML fragments produced. In addition, determining the join

selectivity apriori might not be useful if the join selectivity changes during the lifetime of the multi-way join.

In order to deal with the problem of determining an effective probing sequence for the multi-way join, we propose a novel technique, called Result-Oriented Probing (RoP). RoP dynamically determines the order of the hash partitions to be probed in the multi-way join. RoP tracks the number of partial results that are produced by each hash partition. Whenever a XML fragment $f$ is used to probe a hash partition, a partial result is generated if the bits of the *DoneBitmap* for $f$ have not been completely set to 1. In contrast, a complete result is generated if all the bits of the *DoneBitmap* for $f$ are set to 1.

**Algorithm 4:** MultiWayJoin with RoP

```
   Data   : n - Number of XML streams
            f_d - Newly Arrived XML Fragment,
            Hash Partitions Ht_i, where 0 ≤ i < n
   Result : R, Results
   begin
1      ProbeSequence = SortHashPartitionsAsc() ;
2      for (i=0; i < n; i++) do
3          idx = ProbeSequence_i ;
4          numResults = Ht_idx.probe(f_d) ;
5          if ( numResults == 0 ) then
6              break;
   end
```

Algorithm 4 describes the multi-way join using RoP. In Line 1, the hash partitions are sorted (ascending order) based on the number of partial results produced. *ProbeSequence* stores the information on the probing sequence for the hash partitions. In Line 2-6, we then probe the hash partitions in the order specified by *ProbeSequence*. In Line 5-6, we terminate the probing when one of the partitions do not produce any results.

## 4 Performance Evaluation

We implemented all the algorithms in C++, and conduct the experiments on a Pentium 4 2.4 Ghz PC (1GB RAM). Similar to [18], we make use of the SAX Parser - Expat [24]. Unless otherwise stated, the parameters presented in Table 1 are used for the experiments. We compare the performance of Twig-RPJ, Twig'n Join (TnJ). In addition, we also included a Random method as a baseline. Whenever memory is full, the Random method randomly selects a partition (containing XML Fragments) to be flushed to disk. We evaluated the performance of all the algorithms using both synthetic datasets as well as several real-life datasets.

| Parameter | Values |
|---|---|
| Disk Page Size | 40960 bytes |
| Memory Size, M | 10% of data size |
| Number of entries per page | 31 |
| Number of XML Fragments flushed to disk | 10% of M |
| Number of hash partitions | 1024 |
| Datasets | XMark, TPCH (XML version), DBLP vs SIGMOD Record, Swiss-Prot |

**Table 1.** Experiment Parameters and Values

### 4.1 X007

In this section, we evaluate the performance of Twig'n Join and Twig-RPJ using synthetic datasets generated using X007 [25]. We set the X007 parameters as given in Table 2. We varied the X007 parameter NumConnPerAtomic for values 3, 6 and 9. For each NumConnPerAtomic value, we generated two datasets to simulate the two XML streams. In this experiment, we join the type IDs reference for the Connections. The twig query given below is used for both datasets.

– //Connection[type][AtomicPart]

| X007 Parameters | Values |
|---|---|
| NumAtomicPerComp | 20 |
| NumConnPerAtomic | 3,6,9 |
| DocumentSize(bytes) | 500 |
| ManualSize(bytes) | 2000 |
| NumCompPerModule | 50 |
| NumAssmPerAssm | 3 |
| NumAssmLevels | 5 |
| NumConnPerAssm | 3 |
| NumModules | 1 |

**Table 2.** X007 Parameters

As the graph for values 3,6 and 9 exhibits similar trends, we present the results for NumConnPerAtomic = 9 in Figure 4. From the figure, we can observe that all the methods (TnJ, Twig-RRPJ and Random) perform similarly. This is the case because the values of the join attribute for the XML fragments are uniformly distributed. Thus, regardless of the XML fragments that are flushed to disk, there is no impact on the overall throughput. Similarly, the same observations are made in [5] for uniformly distributed relational data.

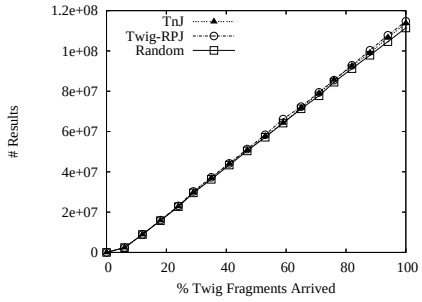



**Fig. 4.** X007

### 4.2 XMark

In this section, we evaluate the performance of Twig'n Join and Twig-RPJ using synthetic datasets generated using XMark [26]. Table 3 shows the size of the XMark datasets, and also the number of XML fragments extracted by the TwigM machine on-the-fly. The fragments are then used in the join of XML fragments.

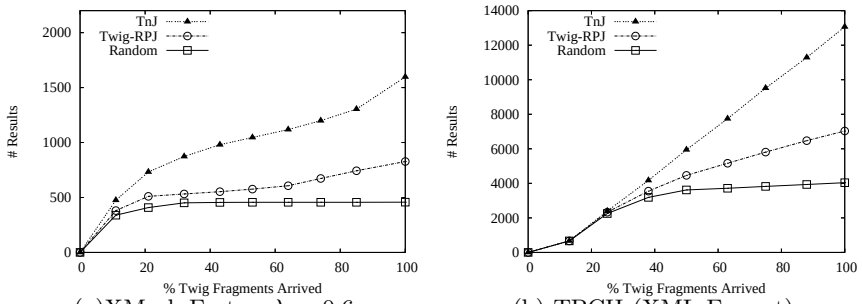
XMark generates a single XML document consisting of information on the annotation, person, category, closed auction, open auction and the items. For the purpose of the experiments, we extracted out the details of the items and closed auctions into 2 separate XML files. This is used to simulate two XML data streams. In this experiment, we join the item IDs reference of the closed auctions with the items. The join attribute is *id* (string). The following twig queries are defined on the Item and Closed Auctions streams respectively.

- **Items**: //item[id][name]
- **ClosedAuctions**: //closed_auction[itemref/id][price]

| XMark Factor, $\lambda$ | Items Fragments | ClosedAuctions Fragments | Total Fragments | Dataset Size(MB) |
|---|---|---|---|---|
| 0.2 | 4350 | 1950 | 6300 | 20 |
| 0.4 | 8700 | 3900 | 12600 | 38 |
| 0.6 | 13044 | 5845 | 18889 | 57 |
| 0.8 | 17400 | 7800 | 25200 | 76 |
| 1.0 | 21750 | 9750 | 31500 | 94 |
| 2.0 | 43500 | 19500 | 63000 | 187 |

**Table 3.** XMark Dataset Information

In these experiments, we varied the scaling factor of XMark, $\lambda$, between 0.2 and 2.0. As we observed similar trends for varying XMark factor, we present only the results for $\lambda = 0.6$ in Figure 5(a). In all cases, Twig'n Join outperforms Twig-RPJ and the random flushing strategy by a large margin.




(a)XMark Factor, $\lambda = 0.6$ (b) TPCH (XML Format)

**Fig. 5.** XMark and TPCH Datasets

### 4.3 TPCH

In this section, we evaluate the performance of Twig'n Join and Twig-RPJ using XML datasets which were converted from datasets generated by TPC-H [27]. We join the data between Orders and Customer. We specify CUST_KEY as the join attribute. The characteristics of the dataset is tabulated in Table 4. The following twig queries are defined on the Orders and Customer XML data streams.

- **Orders**: //T[CUSTKEY][O_ORDERSTATUS]
- **Customer**: //T[CUSTKEY][C_NAME]

From Figure 5(b), we can observe that Twig'n Join outperforms Twig-RPJ significantly. This further shows that Twig'n Join is able to keep XML fragments that have a higher probability to produce results in-memory. Thus, this enables it to be able to produce more results compared with Twig-RPJ.

| Dataset | Number of Elements | DataSet Size |
|---|---|---|
| orders.xml | 150001 | 5MB |
| customer.xml | 13501 | 503KB |

**Table 4.** TPC-H Benchmark (XML version)

### 4.4 DBLP vs SIGMOD Record

In this section, we evaluate the performance of Twig'n Join and Twig-RPJ using two real-life datasets. We used the DBLP dataset (scaled down to 29MB), and SIGMOD Record (467K) [27]. In the experiments, we join on the author attribute (i.e. we want to find authors who have published in SIGMOD Record and have at least one publication listed in DBLP). The following twig queries are defined on the SigmodRecord and DBLP data streams.

– **SigmodRecord**: //issue[volume][//article[title][//author]]
– **DBLP**: //inproceedings[author][title]

From Figure 6, we can see that Twig'n Join outperforms the Twig-RPJ method when approximately 24% of the XML fragments have arrived. We can also observe that the number of result fragments produced increases quickly between 16% - 24% of the XML fragments have arrived. This is because the TwigM machine has not produced sufficient XML fragments which can be joined between the two XML data streams. Beyond 24%, there are sufficient XML fragments available from the DBLP XML data streams in-memory to be joined. Thus, the number of results produced grows linearly beyond that.

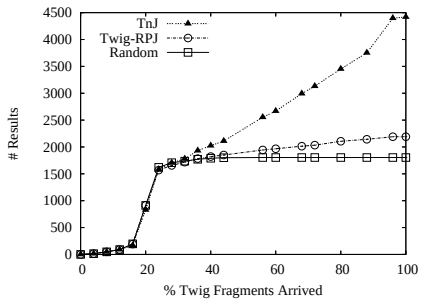



**Fig. 6.** DBLP vs SIGMOD Record

### 4.5 Swiss-Prot

In this section, we evaluate the performance of Twig'n Join and Twig-RPJ using a real-life dataset (Swiss-Prot, available at [27]) and a synthetic dataset (BioExpts). Using the Swiss-Prot dataset, we generate the BioExpts dataset to simulate the details of biological experiments conducted using the protein sequence found in Swiss-Prot. The BioExpts XML file consists of the following information: (1) Experiment ID (ID), (2) Researcher Userid (Researcher), (3) Accession Number (AC) and (4) Observation. The researcher userid and observation consists of randomly generated strings of length 10 and 100 respectively. Figure 7 shows a snippet of the XML generated.

When generating the synthetic dataset, the parameter $\mu$, controls the probability in which an Accession Number from the Swiss-Prot dataset is used in an experiment. When $\mu = 0.0$, then none of the Accession Number are used in the experiments (i.e. no results produced during the join of the Swiss-Prot and the BioExpts dataset). When $\mu = 1.0$, all Accession Numbers are used in the experiments. In other words, $\mu$ controls the join selectivity. We vary $\mu$ from 0.2 to 0.8. In the experiments, we join on the Accession (AC) Number attribute. The

```
<BioExpts>
<Expt>
  <Info>
     <ID>1</ID>
     <Researcher>gXKhK4hkXP</Researcher>
  </Info>
  <AC>P14914</AC>
  <Observation>ABzAW71t ...</Observation>
</Expt>
...
</BioExpts>
```

**Fig. 7.** Synthetic Dataset based on Swiss-Prot

size of Swiss-Prot is 110MB, and the sizes of the synthetic datasets for varying $\mu$ are presented in Table 5.

| $\mu$ | Number of fragments | Size(MB) |
|---|---|---|
| 0.2 | 11420 | 2.7 |
| 0.4 | 22612 | 5.3 |
| 0.6 | 34180 | 8.0 |
| 0.8 | 45377 | 11 |

**Table 5.** Sizes of BioExpts

The following twig queries are defined on the SwissProt and synthetic dataset.

- **SwissProt**: //Entry[AC][Species]
- **BioExpts**: //Expt[/Info/ID][AC]

From Figure 8, we can observe that Twig'n Join consistently outperforms Twig-RPJ for varying $\mu$. Another interesting observation is that when $\mu = 0.8$, the baseline Random method performs better than Twig-RPJ. This shows that the naive Twig-RPJ is not as effective in determining the XML fragments to be flushed to disk whenever memory is full.

### 4.6 Multi-way XML Join

In this section, we compare the performance of the multi-way join using various probing techniques. These includes: (1) RoP (2) Sequential and Apriori. RoP uses the dynamic probing sequence described in Section 3.2. In the Sequential probing strategy, we probe the hash partitions in the order in which the XML streams arrive. In the Apriori strategy, we assume that we know the join selectivity of each of the XML streams. We then probe the hash partitions in order of increasing join selectivity. Thus, hash partitions with lower join selectivity are probed first.

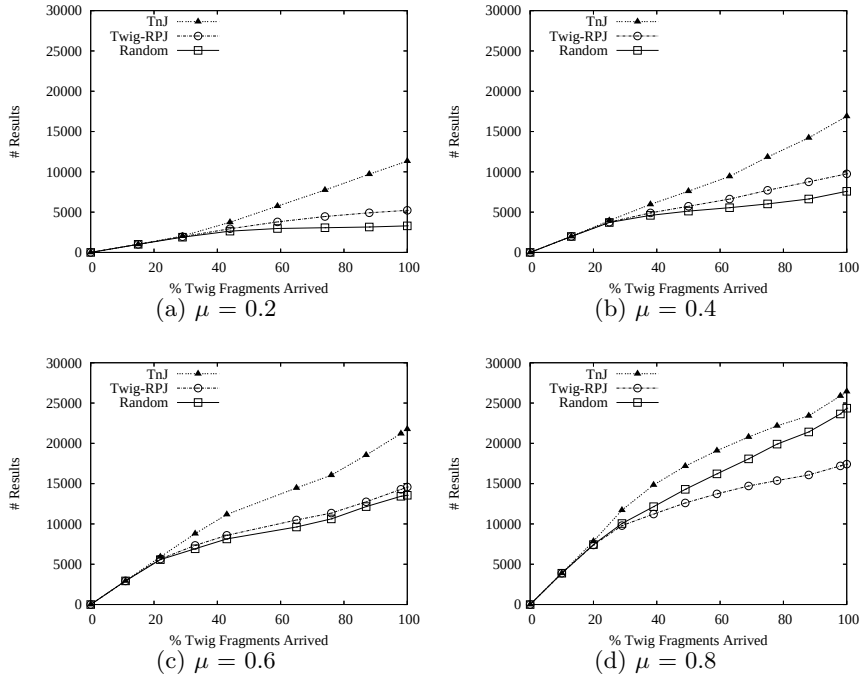
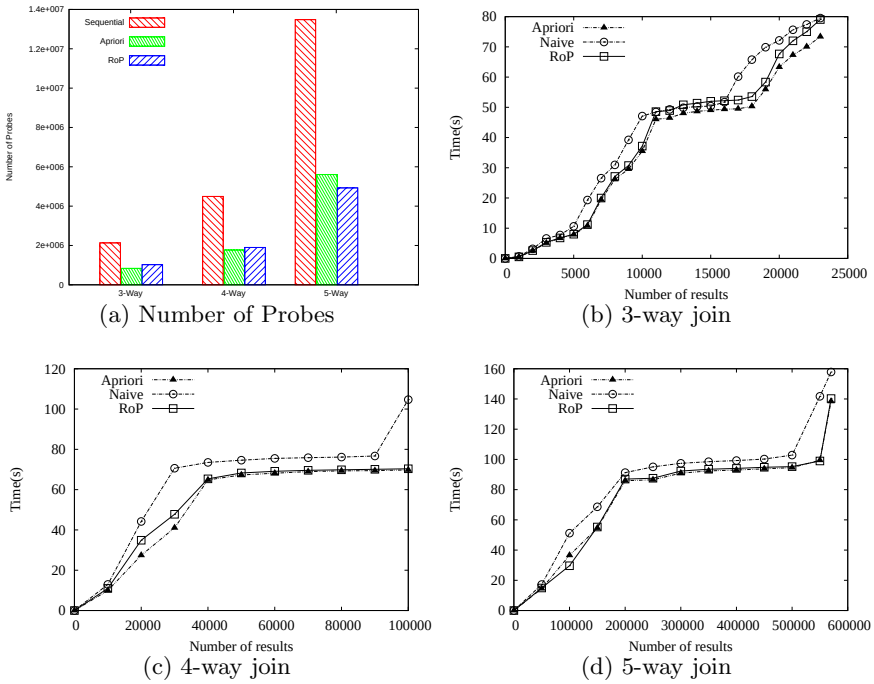



**Fig. 8.** Swiss-Prot vs BioExpts : Varying $\mu$

We evaluate the performance based on two metrics. Firstly, we count the total number of probes on the hash partitions. Secondly, we measured the time taken to produce results.

The XML streams used in this experiment is generated as follows. We first extracted all the name of authors from SIGMOD Record. Using the names of authors, we generated a reference XML stream in which consists of blog entries written by the authors. Next, we generated the other XML streams to be used in the multi-way join by controlling the selectivity, $\mu$. $\mu$ determines the probability that a author from the reference XML stream is included in the stream to be generated. We vary $\mu$ between 0.0 to 1.0. When $\mu = 0.0$, none of the authors from the reference XML stream are included. When $\mu = 1.0$, all the authors from the reference XML streams are included. Various m-way joins are evaluated (m varies between 3 to 5).

From Figure 9(a), we can observe that dynamic result-oriented probing (RoP) outperforms the Sequential probing technique. In addition, RoP performs almost as well the Apriori strategy. This shows that the dynamic RoP technique is effective even without prior information on the join selectivities. From Figure 9(b)-(d), we can observe that RoP outperforms the Sequential probing technique. This commensurates with the findings from Figure 9(a). As a result of




(a) Number of Probes (b) 3-way join

(c) 4-way join (d) 5-way join

**Fig. 9.** Multi-Way Join (with different probing sequence)

the significant reduction on the number of unnecessary probes, RoP takes less time to produce the same number of results.

## 5 Conclusion

We propose a practical approach for progressive processing of (FWR) XQuery queries on multiple XML streams, called Twig'n Join. We decompose a (FWR) XQuery query into a query plan consisting of twig queries and join processing. The twig queries are used for processing the structural constraints. The hash-based join operator is used to process the join predicate constraints. The novelty of this approach compared, to conventional XQuery processing, lies in the decomposition of the XQuery queries into several independent components. This reduces the complexity for the design of XQuery query processing algorithms. Though we show this for XQuery queries involving joins, the technique can be applied to the the various type of (FWR) XQuery queries as well.

Due to the large amount of streaming XML data, it is infeasible to keep all the XML data in-memory during join processing. We make use of the RRPJ method [5] to flush the XML data whenever memory is full. In addition, we introduce a novel dynamic probing technique, called Result-Oriented Probing (RoP), which determines an optimal probing sequence for the multi-way join. This significantly

reduces the amount of redundant probing for results. Experiment results show that Twig'n Join is indeed effective and efficient for the processing of both synthetic and real-life datasets. As future work, we are investigating the use of RoP for multi-way joins with different join predicates.

## References


1. Yahoo Pipes: http://pipes.yahoo.com/pipes/ (2007)
2. Tok, W.H., Bressan, S., Lee, M.L.: Danaides: Continuous and progressive complex queries on rss feeds. In: DASFAA. (2007) 1115–1118
3. Hong, M., Demers, A., Gehrke, J., Koch, C., Riedewald, M., White, W.: Massively multi-query join processing in publish/subscribe systems. In: SIGMOD, Beijing, NY, China, ACM Press (2007)
4. Tao, Y., Yiu, M.L., Papadias, D., Hadjieleftheriou, M., Mamoulis, N.: RPJ: Producing fast join results on streams through rate-based optimization. In: SIGMOD. (2005) 371–382
5. Tok, W.H., Bressan, S., Lee, M.L.: RRPJ : Result-rate based progressive relational join. In: DASFAA. (2007) 43–54
6. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal xml pattern matching. In: SIGMOD. (2002) 310–321
7. Lu, J., Chen, T., Ling, T.W.: Efficient processing of xml twig patterns with parent child edges: a look-ahead approach. In: CIKM. (2004) 533–542
8. Chen, S., Li, H.G., Tatemura, J., Hsiung, W.P., Agrawal, D., Candan, K.S.: Twig$^2$stack: Bottom-up processing of generalized-tree-pattern queries over xml documents. In: VLDB. (2006) 283–294
9. Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and querying ordered xml using a relational database system. In: SIGMOD. (2002) 204–215
10. Lu, J., Ling, T.W., Chan, C.Y., Chen, T.: From region encoding to extended dewey: On efficient processing of xml twig pattern matching. In: VLDB. (2005) 193–204
11. Florescu, D., Hillery, C., Kossmann, D., Lucas, P., Riccardi, F., Westmann, T., Carey, M.J., Sundararajan, A., Agrawal, G.: The bea/xqrl streaming xquery processor. In: VLDB. (2003) 997–1008
12. Paparizos, S., Wu, Y., Lakshmanan, L.V.S., Jagadish, H.V.: Tree logical classes for efficient evaluation of xquery. In: SIGMOD. (2004) 71–82
13. Re, C., Siméon, J., Fernández, M.F.: A complete and efficient algebraic compiler for xquery. In: ICDE. (2006) 14
14. Ludäscher, B., Mukhopadhyay, P., Papakonstantinou, Y.: A transducer-based xml query processor. In: VLDB. (2002) 227–238
15. Peng, F., Chawathe, S.S.: Xpath queries on streaming data. In: SIGMOD. (2003) 431–442
16. Olteanu, D., Kiesling, T., Bry, F.: An evaluation of regular path expressions with qualifiers against xml streams. In: ICDE. (2003) 702–704
17. Li, X., Agrawal, G.: Efficient evaluation of xquery over streaming data. In: VLDB. (2005) 265–276
18. Chen, Y., Davidson, S.B., Zheng, Y.: An efficient xpath query processor for xml streams. In: ICDE. (2006) 79

19. Stark, M., Fernández, M., Michiels, P., Siméon, J.: XQuery streaming á la carte. In: ICDE. (2007)
20. Tok, W.H., Bressan, S., Lee, M.L.: Progressive spatial joins. In: Proc. of International Conference on Scientific and Statistical Database Management. (2006) 353–358
21. Olson, M., Bostic, K., Seltzer, M.: Berkeley db. In: Summer Usenix Technical Conference, Monterey. (1999)
22. Tok, W.H., Bressan, S., Lee, M.L.: Progressive high-dimensional similarity joins. In: DEXA. (2007) (To be published)
23. Viglas, S., Naughton, J.F., Burger, J.: Maximizing the output rate of multi-way join queries over streaming information sources. In: VLDB. (2003) 285–296
24. Clark, J.: The expat xml parser, http://expat.sourceforge.net (2003)
25. Bressan, S., Dobbie, G., Lacroix, Z., Lee, M.L., Li, Y.G., Nambiar, U., Wadhwa, B.: X007: Applying 007 benchmark to xml query processing tool. In: CIKM. (2001) 167–174
26. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: Xmark: A benchmark for xml data management. In: VLDB. (2002) 974–985
27. XML Data Repository: http://www.cs.washington.edu/research/xmldatasets/ (2002)