View Selection for Stream Processing

Ashish Kumar Gupta Alon Y. Halevy* Dan Suciu †
University of Washington
{akgupta, alon, suciu}@cs.washington.edu

Abstract

Consider XML content-based document routing: a stream of XML documents are routed through a network, and routing decisions are taken based on the result of evaluating XPath predicates on these documents. Parsing XML documents and interpreting XPath expressions is the main bottleneck in such systems. We propose a novel solution to speedup the evaluation of XPath predicates based on precomputing views for the XML documents. There are both similarities and differences from the "view selection problem" in relational databases. We describe an architecture for using these views, discuss several design choices and make a brief theoretical analysis for one special case. Finally, we report some initial experiments, showing the potential for query speedup by using stream views.

1 Introduction

We consider a class of XML applications in which a continuous stream of XML documents is processed and routed in a network of servers. Examples of such applications include content-based XML routing [21], selective dissemination of information (SDI) [2, 6], and continuous queries [8]. XML documents are generated at certain nodes in the network. These documents then flow in the network and may get replicated through the network's servers. Servers only do some minimal processing on the documents: evaluate some boolean predicates, compute some aggregates and forward the documents to one or more servers in the network. Performance is critical in such applications, since servers usually need to keep up with the network's throughput. The main bottleneck in achieving a high throughput is the XML processing part: parsing and then evaluating a collection of XPath expressions.

*Halevy was partially supported by a Sloan Fellowship and gifts from Microsoft Research, NEC and NTT.

†Suciu was partially supported by the NSF CAREER Grant 0092955, a gift from Microsoft, and a Sloan Fellowship.

A particular example of such an application is the XML Routing system described in [21]. The project aims to achieve low latency in the presence of failures, by sending multiple copies of a document from a source to its destinations, and routing different copies through different paths. Each router forwards every XML document it receives to a subset of its output links (to other routers or clients), and makes these routing decisions based on the results of evaluating a large number of XPath predicates (corresponding to clients' subscription queries) on the XML document. No data processing is required beyond the evaluation of XPath predicates. But the performance reported in [21] with publicly available tools is very poor, since each document needs to be parsed and then XPath queries need to be evaluated at each server where it is forwarded.

This paper proposes a new method of using views that can significantly increase the throughput in such applications – by a factor of up to 100 in our initial experiments. In our approach, the XML data flow is complemented by a second flow of *views*, which is used by the servers to speed up the computation of their XPath expressions. The views for each XML document are computed only once, by the producer of that document, and can then be used by all the servers downstream. The key to our approach is that the answer to a view is encoded in the header of the XML document – the header contains the offsets of the answers to the views within the XML document. When a server receives an XML document, it tries to answer its set of XPath expressions using the views that are already precomputed on that document. In the case of a *hit*, i.e., the set of expressions can be evaluated using the views, both parsing the XML and XPath evaluation are avoided and replaced with simple lookups inside the XML document. The observed performance gains in this case are two orders of magnitude compared to normal processing. If a server cannot evaluate its expressions using the views (i.e., a *miss*), then it falls back on parsing the XML document and normal XPath evaluation.

Our goal in this paper is to describe the general architecture for using views in a stream processing

system, and then to define a specific representation of the views and evaluate their potential speedup in XML stream processing. In our particular representation, a view V is given by an XPath expression, and the result of evaluating V on an XML document consists of the byte offset of the node selected by V in the document, or NULL if the result does not consist of exactly one node. Given a set of views to be precomputed, called a *view configuration*, VC, the results of all the views in the VC on a given XML document is called a *header*. In principle, our approach raises problems on several dimensions:

View Selection: given a set of statistics on the XML streams, and the global query workload at all servers, choose a view configuration (VC) that maximizes the system's expected throughput. The VC is then made available to all servers in the network. Optionally, we may decide to choose different view configurations for different type of XML documents; for example choose a different VC for each DTD.

Online v.s. Offline Configuration: the view selection can be done offline, in which case the VC is computed before the system starts operating. The assumption here is that a central server knows the query workload, network topology, as well as statistics on the XML documents, and can choose a VC that optimizes the global performance. In the online configuration the VC is chosen dynamically by the XML data providers, based on feedback from the network. No central server is needed, but, on the other hand, a global optimum is harder to achieve. This paper restricts the discussion to offline configuration.

Run-time Evaluation: given an XML document with a set of materialized views (i.e., its header), a server needs to choose a good plan to evaluate all queries in its workload in order to maximize the probability of a hit. In the case of conjunctive queries, servers may choose to evaluate conjuncts with low selectivity first and then short-circuit the evaluation whenever a condition evaluates to *false*.

Using materialized views in query processing is a widely applied technique in database query processing [23, 12, 7, 20, 25, 13, 5, 17]. The problem of view selection has also received significant attention as of late [1, 9, 14, 16, 22, 24, 4, 15, 18, 19, 10]. However, there are significant differences between materialized views in databases and materialized views in stream processing. First, space comes almost for free in database applications, while space is the primary limiting factor in stream processing, because the views and the XML data stream share the same network. For example, if the views are as large as the XML documents, then the network throughput for the combined stream is reduced to half of the throughput without views, more than offsetting any benefit gained from using those views. On the other hand, while the cost of updating views is a concern in databases, it is not in stream processing.

A second difference is that views in stream processing are dynamic, while in database applications they are static. Different documents may have different views and hence different headers. When the XML document is first generated, a specific header is selected that would best benefit all servers downstream in processing that particular document, and a tag is attached to specify which header the document carries. For example, in an XML document routing application where documents belong to different domains, there may be a different header for each domain. This idea can be pushed further and have multiple documents within the same domain: when a server has a miss for that document, and needs to parse the entire document, it may decide to compute another header to better help servers downstream.

This paper makes three contributions. First, it describes an architecture for using stream views, the first of its kind. Second, we provide experimental support for the potential speedup from using stream views. Finally, it explores a few directions in the design space, and provides a theoretical study for one particular choice.

2 Overview

We define here the problem formally and discuss a number of techniques that define the solution space.

2.1 Problem Setting

We define here an *XML Stream Processing Network*, to consists of a network of servers evaluating queries on an input XML stream. Queries are conjunctive queries, of the form:

```
Q  ::=  G ∧ ... ∧ G
G  ::=  Expr Oprel Const
```

Each `Expr` is an XPath expression, and each `Oprel` is any relational operator comparing the XML data value to a constant, including arithmetic comparisons, substring searchers, string regular expression matches, or datatype dependent operators such as date comparisons.

For example: /news//company/text() = "IBM" ∧ /news/agency/text() contains "Reuters".

A server, S, has a workload of conjunctive queries, denoted by $\mathcal{Q}_S$. The server accepts incoming XML documents, evaluates all queries in $\mathcal{Q}_S$ on each such document, and takes appropriate actions, such as forwarding the document to other servers, updating some aggregate values, updating a database, etc.

An XML stream processing network consists of a set of servers, $\mathcal{S}$, connected in a network topology, $\mathcal{N}$, that defines how documents are forwarded between servers. XML documents are generated at source nodes in the network, then sent along the network edges. Each server evaluates its queries on each XML document, and depending on the queries' results, forwards the document to some of its output links.

2.2 Basic Stream Processing with Views

A *view*, V, is one XPath expression. Given an XML document and a view, the value of the view consists of the byte offset representing the first byte of the XML fragment that represents that XPath expression, or NULL if the result does not consist of exactly one node. A *view configuration* consists of an ordered collection of views, $VC = (V_1, \ldots, V_k)$: the value of a VC on an XML document D is an ordered collection of offsets, $VC(D) = (d_1, \ldots, d_k)$, and is called a *header*, and k is called the *header size*.

Stream processing with views proceeds in two phases. The first is the *view selection problem* whose goal is to compute the set VC, and the *plan generation problem* whose goal is to produce a plan at each server S. This phase is done offline: view selection is done on a central server, while plan generation on all the servers. The inputs to view selection problem are the set of servers $\mathcal{S}$, the connection network $\mathcal{N}$, the workload $\mathcal{Q}_S$ at each server, as well as statistics such as distribution of the size of the XML documents, selectivities of different predicates, probability distributions on the XML stream, probability distributions for each link in the network. The result of the view selection problem consists a number k, called the header size, and a set $VC = (V_1, \ldots, V_k)$. Both k and VC are now distributed to all servers in the network, and each server S generates a plan for its workload $\mathcal{Q}_S$. The plan essentially chooses an evaluation order for the conjuncts in each query $Q \in \mathcal{Q}_S$, with *short-circuit*: whenever a conjunct evaluates to *false*, the rest of the conjuncts are no longer evaluated.

In the second phase, the network processes XML documents, as follows. For each XML document D, its producer computes a header $H = VC(D)$, and "attaches" it to the document. Packets are then routed through the network. Whenever a server S receives an (H, D) pair, it evaluates its query plan. The plan consists of repeated evaluation of XPath expressions, and condition tests. As long as the XPath expressions requested can be satisfied by looking up the header, the server does exactly that. If some XPath expression is encountered for which there is a miss, then the server parses D, and enters a traditional evaluation mode.

2.3 Advanced Stream Processing with Views

We discuss here a number of extensions to the basic stream processing method described in Sec. 2.2.

Dynamic Headers: in this approach we generate multiple view configurations, $VC_1, \ldots, VC_m$, and dynamically choose a header type for each document. An additional tag, $t \in \{1, 2, \ldots, m\}$, is attached to each header H to indicate its type. Header types may differ both in their size, and in what XPath expressions they choose to precompute. The following two examples illustrate the usefulness of dynamic headers. (1) The DTDs for the XML documents are known ahead, and there are m different DTDs. Then it makes sense to choose a different view configuration for each DTD. (2) Different servers in the network focus on different parts of the data. Some servers test primarily the fields /news/content/address/city and /news/content/address/country, while other servers focus on /news/header/agency and /news/header/date. As documents travel through the network it makes sense to dynamically change their header to improve the hit rate downstream.

Nested XML Elements: assume that the following XPath expressions occur frequently in the workload : /news/content/address/country, /news/content/address/city, /news/content/address/phone. There are two ways to support them with views. The first is to define three views in VC: this however uses up three entries in the header. Alternatively, we could have a single entry in VC, corresponding to the view /news/content/address. Now servers can read directly the byte offset of the <address> element, and parse the document from that offset in order to retrieve <country>, or <city>, or <phone>. By moving up or down the XML hierarchy we can trade off header size with query speedup.

Multiple XML Elements: so far we have assumed that if some XPath expression in VC evaluates to two or more nodes on a specific document, then the corresponding entry in the header is NULL: this is required, in order to implement XPath's existential semantics. For example, the XPath predicate /`news/content/address/country` = "`France`" is true on a document with two `address` elements, one in France and one in Belgium, and storing only one offset in the header may mislead the server in believing that there is only one value. It is possible to extend views to cope with multiple occurrences of elements. For example, we could have two distinct views in VC: `/news/content/address[1]/country` and `/news/content/address[2]/country`. If there are at most two addresses in the document, then both can be represented in the header; if there are three or more addresses, then they cannot be represented and we issue a *miss*.

3 The View Selection Problem

We now discuss the view selection problem, which is, in essence, an optimization problem. We show that it is hard, even in the simplest settings. We then propose a simple greedy algorithm for it, which we use in our experiments to compute view configurations.

3.1 A Hardness Result

We consider here a simplified form of the view selection problem. Let $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ be the set of all XPath expressions occurring in the system and let $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ be the set of servers in the network. Each server S_i has a set of z_i queries $Q_i = \{q_{i1}, q_{i2}, \ldots, q_{iz_i}\}$ associated with it. The first simplifying assumption we make concerns the topology of the network: we assume the servers are arranged in a linear chain. Each server evaluates its set of queries on every incoming document and forwards the document to the next server in the chain. Second, we assume that the number of conjuncts in each query q_{ij} is one, i.e., $q_{ij} = p_{\alpha_{ij}}$ *Oprel Const* for some $p_{\alpha_{ij}} \in \mathcal{P}$.

If the maximum possible size of the view configuration VC, is k (i.e., the header can store the offsets for no more than k XPath expressions), then the View Selection Problem is to select $VC \subseteq \mathcal{P}$, of size k, such that the total number of hits in the network is maximized. For a hit to occur at a server S_i, VC must contain all the XPath expressions that need to be evaluated at that server. If we denote $\{p_{\alpha_{ij}} \mid j = 1, 2, \ldots, z_i\}$ by s_i, then this essentially means $s_i \subseteq VC$. Therefore, to maximize the number of hits, we need to select such a VC of size k that it covers as many s_i as possible.

It turns out that there is another way of looking at the same problem. Let $k' = |\mathcal{P}| - k$ and $VC' = \mathcal{P} - VC$. Note that retaining VC of size k that maximizes hits is the same as discarding VC' of size k' that minimizes misses. (For a document, the number of hits plus the number of misses is equal to the number of servers in the network. Therefore maximizing hits is equivalent to minimizing misses.) Let us first associate with each $p_j \in \mathcal{P}$, a set $P_j = \{S_i \mid p_j \in s_i\}$. P_j is the set of servers at which the XPath expression p_j occurs in a query. If for some j, $p_j \in VC'$, then a miss occurs at every server in P_j. Therefore, to minimize the number of misses, we need to select such a VC' that $|\bigcup_{p_j \in VC'} P_j|$ is minimized. The restricted view selection problem can now be stated as follows.

Definition 3.1 Restricted View Selection *(a) Optimization Problem: Given the sets $\mathcal{P}$, $P_1, P_2, \ldots, P_n$ as defined above and a number k', compute a $VC' \subseteq \mathcal{P}$ of size at least k' such that $|\bigcup_{p_j \in VC'} P_j|$ is minimized. (b) Decision Problem: Given the sets $\mathcal{P}$, $P_1, P_2, \ldots, P_n$ as defined above and numbers k' and x, does there exist a $VC' \subseteq \mathcal{P}$ of size at least k' such that $|\bigcup_{p_j \in VC'} P_j| \leq x$.*

Theorem 3.2 *The Restricted View Selection Problem is NP-complete.*

Proof: Proof by reduction from the clique problem [11]. Given a graph $G = (V, E)$ and a number k, we can construct an instance of the restricted view selection problem. The set $\mathcal{P}$ is the set of edges of the graph G. For each edge $e_j \in E$, define $P_j = \{v_i \mid e_j \textit{ is incident on the vertex } v_i\}$. The number k' is $k(k-1)/2$, and x is k. It is easy to see that a solution to the restricted view selection problem exists if and only if the graph $G = (V, E)$ has a clique of size k.

□

Previous theoretical analyses of the view selection problem (e.g., [10, 16]) have focused on SPJ queries and on traditional database cost models. Here our queries are simpler (conjunctions of path selections), and our cost is modeled by hits and misses. Hence, Theorem 3.2 does not follow from previous results.

3.2 A Greedy Algorithm

We propose here a simple greedy algorithm for the view selection problem, that essentially uses heuristics to guide its search. It can be extended to more

complex versions of the problem. The greedy algorithm works by discarding, at each stage, the XPath expression p_x that is required by minimum number of remaining servers to answer their queries.

Algorithm 1 Greedy Algorithm for Selecting the View Configuration

```
1:  for p_j ∈ P do
2:      P_j = {S_i | p_j ∈ s_i}
3:  end for
4:  k' = |P| − k
5:  for i = 1 → k' do
6:      Select a p_x ∈ P that minimizes |P_x|
7:      P ← P − {p_x}
8:      for p_j ∈ P do
9:          P_j ← P_j − P_x
10:     end for
11: end for
```

The algorithm works as follows. The set $\mathcal{P}$ contains, at each stage, the set of XPath expressions that are candidates for being a part of the view configuration. Line 6 is the greedy decision-making step. An XPath expression p_x is chosen that is required by minimum number of servers. After p_x is discarded, we take out the servers in P_x from the list of servers for the remaining XPath expressions. This is because once p_x is discarded, a miss is going to result at all the servers in P_x and for the later iterations, we want to consider only those servers which could still have a hit. When the algorithm terminates, the set $\mathcal{P}$ is the required view configuration.

4 Experiments

Our experiments demonstrate the following: (a) when the view configuration contains all the XPath expressions used in the system, then very high speedup can be achieved. (b) significant speedup can be achieved even if a good fraction of the XPath expressions are missing from the view configuration.

Our execution environment consists of a dual 450MHz Pentium II with 1536MB memory, running Red Hat Linux 7.1. Our compiler is gcc version 2.96.2, without any optimization options. We use the `Xerxes` SAX parser (available from the Apache foundation [3]) to parse XML. We run each experiment five times and we report the average. To simulate a stream of documents, we take a document and replicate it multiple times in the same file. There are 10000 servers in this experiment. The number of queries at servers vary from 1 to 5. Each query contains a single conjunct. The XPath expressions in the conjuncts for the queries come from a Zipf distribution. Instead of distributing the queries on 10000 different machines, the queries for all the servers reside on the same machine. And during the experiment, we evaluate the set of queries associated with each server on every replica of the document in the input file, and record the total execution time. We use the greedy algorithm from Sec. 3.2 to compute the view configurations for varying header sizes. Only the queries associated with a given server are processed in parallel. For every document-server pair, we reset the parser to the beginning of the document. We ran this experiment on documents of varying sizes. We observed similar trends. So, we report numbers for just one document, for lack of space.

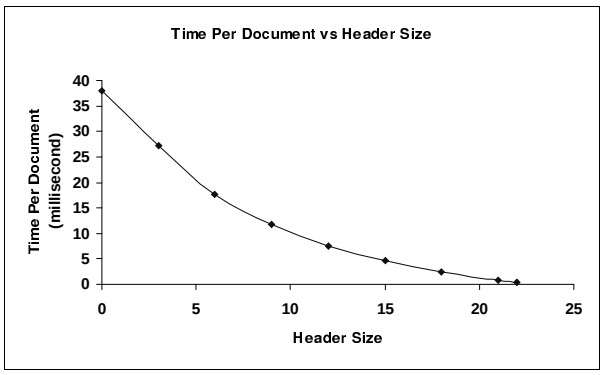

Figure 1: Average time to process a document at a server as a function of header size

Figure 1 shows the average time to process a document at a server as a function of header size for a document of size 52KB. Without the header, this time is 37.99 ms, and with the header containing offsets for all the required XPath expressions, this time is 0.372 ms, giving a speedup of more than 100. Even when the header contains offsets for just two-thirds (15 out of 22) of the XPath expressions present in the system, we still get an average processing time of 4.61 ms, a speedup of 8.3. Thus, even when a fairly good fraction of XPath expressions is not present in the view configuration, a healthy speedup can be obtained by using these views.

One might argue that, using a faster parser will negate most of the speedup achieved by using the views. However, this is not the case. If we use a parser that is 10 times faster than the `Xerces` parser, the maximum speedup achieved does go down by a factor of 10. However, very rarely will we have the case that the header will contain the offsets for all the XPath expressions in the system. The portion of the graph in Figure 1 that we want to concentrate on is when some of the XPath expressions are missing from the view configuration. When the view configuration

is missing 7 of the 22 XPath expressions, we still get a speedup of 5.2 (down from 8.3), if we use a parser that is 10 times faster.

5 Conclusions

We have described an architecture for XML stream processing that uses views in a novel way. The key idea is to encode the results of a view in the header of the document and hence to avoid parsing the XML and XPath expression evaluation. We outlined a general architecture in which this idea can be applied, and studied one instance of this architecture. For this instance, we proved that the view selection problem is hard and proposed a greedy algorithm. Our preliminary experiments show that our approach can yield significant speedups, even in cases where the headers do not contain all the relevant views. Our approach raises a rich set of problems that we are currently pursuing, including the development of methods for view selection, run-time evaluation, methods for synchronization of data and views, and consideration of different network topologies and application characteristics.

References

[1] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes in Microsoft SQL Server. In *Proc. of VLDB*, pages 496–505, Cairo, Egypt, 2000.

[2] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination. In *Proceedings of VLDB*, pages 53–64, Cairo, Egipt, September 2000.

[3] Apache. Xerces C++ parser, 2001. http://xml.apache.org.

[4] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *Proc. of VLDB*, pages 156–165, 1997.

[5] R. Bello, K. Dias, A. Downing, J. Feenan, J. Finnerty, W. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in Oracle. In *Proc. of VLDB*, pages 659–664, 1998.

[6] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressiosn. In *Proceedings of the International Conference on Data Engineering*, 2002.

[7] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proc. of ICDE*, pages 190–200, Taipei, Taiwan, 1995.

[8] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for internet databases. In *Proceedings of the ACM/SIGMOD Conference on Management of Data*, pages 379–390, 2000.

[9] R. Chirkova and M. Genesereth. Linearly bounded reformulations of conjunctive databases. In *Proc. of DOOD*, pages 987–1001, 2000.

[10] R. Chirkova, A. Halevy, and D. Suciu. A formal perspective on the view selection problem. In *Proc. of VLDB*, 2001.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[12] D. Florescu, L. Raschid, and P. Valduriez. Answering queries using OQL view expressions. In *Workshop on Materialized Views, in cooperation with ACM SIGMOD*, Montreal, Canada, 1996.

[13] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: a practical, scalable solution. In *Proc. of SIGMOD*, pages 331–342, 2001.

[14] H. Gupta. Selection of views to materialize in a data warehouse. In *Proc. of ICDT*, pages 98–112, 1997.

[15] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *Proc. of ICDE*, pages 208–219, 1997.

[16] H. Gupta and I. S. Mumick. Selection of views to materialize under a maintenance cost constraint. In *Proc. of ICDT*, pages 453–470, 1999.

[17] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal*, 10(4), 2001.

[18] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. of SIGMOD*, pages 205–216, 1996.

[19] H. J. Karloff and M. Mihail. On the complexity of the view-selection problem. In *Proc. of PODS*, pages 167–173, Philadelphia, Pennsylvania, 1999.

[20] R. Pottinger and A. Halevy. Minicon: A scalable algorithm for answering queries using views. *VLDB Journal*, 2001.

[21] A. Snoeren, K. Conley, and D. Gifford. Mesh-based content routing using XML. In *Proceedings of the 18th Symposium on Operating Systems Principles*, 2001.

[22] D. Theodoratos and T. Sellis. Data warehouse configuration. In *Proc. of VLDB*, pages 126–135, Athens, Greece, 1997.

[23] O. G. Tsatalos, M. H. Solomon, and Y. E. Ioannidis. The GMAP: A versatile tool for physical data independence. *VLDB Journal*, 5(2):101–118, 1996.

[24] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proc. of VLDB*, pages 136–145, Athens, Greece, 1997.

[25] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *Proc. of SIGMOD*, pages 105–116, 2000.