FMware: Middleware for Efficient Filtering and Matching of XML Messages with Local Data*

K. Selçuk Candan, Mehmet E. Dönderler, Yan Qi,
Jaikannan Ramamoorthy,Jong W. Kim

Computer Science and Engineering Department
Arizona State University, Tempe, AZ 85287-8809
{candan,mdonder,yan.qi,jaikannan,jong}@asu.edu

Abstract. XML message filtering systems are used for sifting through real-time messages to support business data mining and reporting. An XML message filtering system needs to (a) process registered filter predicates on multiple distributed real-time streams and (b) match and validate the filter results with local data to identify the relevant data that can be used for higher-level processing. Although efficient real-time filtering schemes exists, the *matching* phase of the operation where filter results have to be matched against local data to select those matches that are relevant to the particular task remains to be expensive as it requires expensive join operations. In this paper, we present an efficient middleware (*FMware*) for filtering and matching XML messages against locally available data. The proposed operator relies on a novel cluster-domain matching scheme to reduce the cost of the process. We analytically study the cost of the proposed middleware and experimentally show that it adaptively reduces the number of local data accesses and provides large savings in matching time with respect to cluster-unaware matching.

Keywords: XML messaging, matching with local data, cluster-domain hashing

1 Introduction

XML message brokers provide filtering, tracking, and routing services to enable processing and delivery of the message traffic within an enterprise. These tools (e.g. JMS [1] and IBM's MQSeries [2, 3]) listen to (possibly multiple) XML data streams within an enterprise (or across enterprises) and identify message data fitting the registered user profiles or filter queries. These messages are then passed to appropriate business intelligence modules for further processing. Thus, efficient middleware support for filtering and publish/subscribe services is critical for effective use of system resources, reducing the messaging delays, and simplifying the design of enterprise business intelligence systems.

In this paper, we first note that such *basic XML document processing tasks can be off-loaded* to a middleware. In fact, there is an increasing number of XML message process off-loading technologies. Yet, most of these technologies provide either low-level XML parsing acceleration support [4], (usually proxy-based) publish/subscribe solutions (e.g. SemCast [5], CoDD [6], NiagaraCQ [7]), message validation through XML-gateways and XML-firewalls (e.g. DataPower [8]

* This research is funded by NSF grants ITR-0326544 and IIS-0308268.

and Sarvega [9]), or purely network-level intelligent message routing solutions [10, 11] which do not go beyond interpreting the request and reply message headers.

Existing work in publish/subscribe middleware focuses on the problem of routing of data and the filter queries in a way to ensure that right filter results reach the correct users in the shortest amount of time with minimal resources. For instance, CoDD [6] uses subscription queries to create a hierarchical tree structure which disseminates subsets of a data stream to consumers through loosely-coupled peer nodes. On the other hand, in an enterprise business intelligence context, it is not common that there are thousand of widely distributed subscribers for filter results. Therefore, routing and dissemination are less critical in this domain then efficiency in filtering and matching: since large volumes of data arrives continuously, it is essential that the filtering rate matches the data arrival rate to prevent the loss of valuable information. Therefore, the collection of query patterns need to be indexed in-memory to enable real-time filtering of the data. The state of the art in XML filtering schemes include YFilter [12], AFilter [13], TurboXPath [14], and XSQ [15]. Although, thanks to these in-memory based filtering techniques (relying on state machines, push down automata, or transducers), the filtering step itself can generally be performed in real-time (on the order of 100K filter statements), a major remaining challenge in business context is the *impedance mismatch* between the *in-memory filtering schemes* and the *locally relevant data in secondary storage.*

1.1 Challenge: Impedance Mismatch Between In-Memory Filtering Schemes and Locally Relevant Data In Secondary Storage

Consider an enterprise with multiple sales offices and multiple suppliers. Let us assume that the `product shipment office` of this enterprise needs to identify for each sale, (a) the productid of the sold item, (b) number of units sold, and the (c) appropriate warehouse for product shipment. Let us also assume that this enterprise is relying on XML messaging for communicating between the various offices and branches. Without getting into the details of the corresponding schema, let us further assume that the XML message filtering system can listen to the sales messages (with a registered filter statement of the form "*//productid//unit_of_sales*") to extract $\langle productid, unit\_of\_sales\rangle$ information for shipment. However, let us further consider the case where the sales messages arriving from the local sales offices **do not** contain the *warehouseid* information for the products. This is expected in this case, as *warehouseid* is relevant only to the `product shipment office` and "*//productid//warehouseid*" will only be available locally (possibly at a secondary storage).

Therefore, although in-memory message filtering (such as YFilter [12]) can be used for extracting "*productid//unit_of_sales*" from incoming sales messages, an efficient middleware is needed for matching these against locally stored data to identify $\langle productid, unit\_of\_sales, warehouseid\rangle$ matches (Figure 1).

1.2 Contributions of this Paper

With the goal of supporting time critical filtering, tracking, and routing services for enterprises, in this paper we present a novel *FMware* middleware for efficient XML message stream filtering and matching against locally stored data

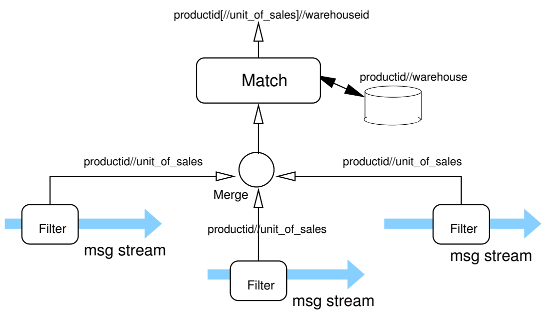


Fig. 1. Filtering and matching XML messages against locally available data

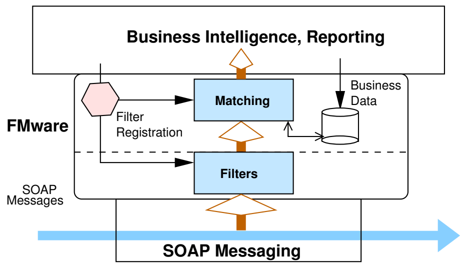


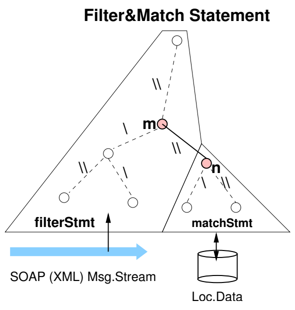
Fig. 2. FMware middleware for filtering and matching messages against local data

(Figure 2 and Section 2). In particular, we focus on the post-processing phase required for validation of the message filter results against XML data in secondary storage and we develop an index-driven $\mathcal{C}$Match (clustered-matching) operator for efficient implementation of the $FMware$ middleware.

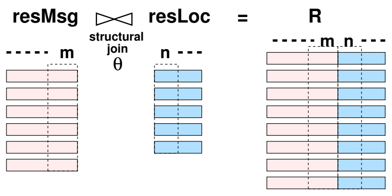
To reduce the cost of message filtering, in-memory schemes (such as YFilter and AFilter) rely on structural similarities of the filter statements. When the matching phase requires access to data in the secondary storage, however, exploiting structural similarities is not straight-forward. Existing index-structures (such as [16]), that are used in XML DBMS context, rely on *prefix* clustering through an ancestor-descendant interval labeling (Section 3). $\mathcal{C}$Match operator, on the other hand, relies on a multi-interval scheme to exploit other structural clustering opportunities to adaptively reduce accesses to the secondary storage (Sections 3 and 3.7). In Section 4, we experimentally show that cluster domain processing not only reduces the matching cost, but knowledge about *clustering power* of the data can be exploited by FMware to choose the appropriate available index for matching.

2 Overview of the FMware Middleware

Traditional XML filtering systems are concerned with finding instances of a given set of patterns in a continuous stream of data trees (or XML messages). More specifically, if $\{x_1, x_2, ...\}$ denotes a stream of XML messages, where x_i is i^{th} XML message in the stream, and $\{q_1, \ldots, q_m\}$ is a set of filter predicates (described in an XML query language, such as XPath [17] or XQuery [18]) then an XML filtering system identifies (in real-time) $\langle x_i, q_j, PT_{ij}\rangle$ triplets, such that



(a) Splitting a statement



(b) Corresponding structural join

Fig. 3. (a) A sample, tree-structured, filter statement which requires filter results from incoming messages matched against local data and (b) the corresponding structural join operation that needs to be performed efficiently

the message x_i satisfies the filter query q_j. The set PT_{ij} includes each instance of the query (referred to as path-tuples in [12]) in the message.

In order to enable (a) *filtering* of XML messages against registered queries and (b) *matching* filter results against locally stored data, $FMware$ middleware needs to interface available filtering engines with data available in secondary storage. For example, consider the XPath filter statement $A[//B]//C$. Let us assume that the XML messages contain enough information to match the $A//B$ pattern, however the $A//C$ should be verified using local data. Thus, the filter statement can be split into two parts:

$$filterStmt = A[//B] \;\; and \;\; matchStmt = resMsg.A//C,$$

where $resMsg$ denotes the results for the filter statement $filterStmt$. Thus, the stream of filtering results will need to be further matched against the locally stored data for evaluating the $resMsg.A//C$ relationship.

Definition 1 (Filtering and Matching with Local Data). *Let*

– *the $filter\&match$ statement can be split into two sub-filter statements: $filterStmt$ for filtering on XML message stream and matchStmt for local data,*
– *$resMsg$ denotes the stream of message filtering results, where each result, $rmsg_i$, is a tuple (as in [12]) of nodes satisfying conditions specified in $filterStmt$,*
– *resLoc denotes the set of tuples, where each result, $rloc_j \in resLoc$, satisfies conditions specified in matchStmt, and*
– *Θ is a structural condition between node $node_m$ in rmsg tuples and $node_n$ in rloc tuples.*

The filtered and matched result, R, consists of a stream of pairs, $\langle rmsg_i, rloc_j \rangle$, where $node_{i,m} \in rmsg_i$ and $node_{j,n} \in rloc_j$ satisfy the condition Θ (Figure 3).

Broadly speaking, there are two different ways to perform the *filtering and matching with local data* task:

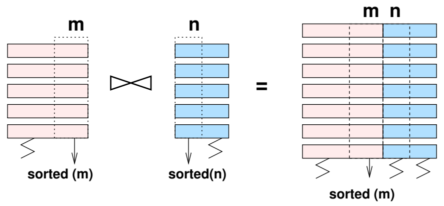


Fig. 4. Sort-merge-join based approaches assume that the inputs are structurally sorted in the joining nodes (m and n); naturally, the output can be sorted on only one of m or n (sorted on m in this example)

- **Alternative I (Periodic matches):** (a) $filterStmt$ is evaluated on the XML message stream to identify a batch $resMsg\_Batch$ of matches. (b) $matchStmt$ is evaluated on the local data to identify local candidates $resLoc\_Batch$. (c) $resMsg\_Batch$ and $resLoc\_Batch$ is joined.
- **Alternative II (Streaming matches):** Each filter result $rmsg_i$ is matched against the local data using an *efficient* index structure to locate the local matches.

The disadvantages of the first alternative is that (a) it is blocking and (b) it requires explicit materialization of all candidate matches in advance. The second alternative requires neither blocking nor explicit materialization; however, it is essential that the matching is performed efficiently.

2.1 Alternatives for Streaming Matching Implementations

Structural relationships within XML data constitute significant information that has to be used in querying, indexing, and retrieval. Various structural join algorithms are devised for speeding up the processing of queries which involve ancestor/descendant type of structural relationships.

Structural join algorithms can be classified into two: holistic and binary. Holistic join operators take the entire query and match it against the data as a whole. Since in a streaming environment data itself is distributed and available in pieces, such holistic approaches, which are shown to work well for static XML data, are not applicable. Many existing (binary or holistic) structural join operators, including TwigStack, PathStack [19], iTwigJoin [20], Stack-Tree-Desc/Anc [21], $\mathcal{EE}/\mathcal{EA}$-Join [22], and TSGeneric [23], are specially designed variants of the standard **sort-merge join** algorithm: they require that the ancestor and descendant lists be available in a structurally sorted order before the join operation can be performed (Figure 4). Consequently, these sort-merge based schemes face the problems common in traditional sort-merge-joins: **(a)** they risk being blocking (for sorting the inputs) or **(b)** they constrain query plans to only those that can provide appropriately sorted inputs. Unfortunately, when the filter results for validation arrive from multiple message queues (with potentially different message structures and arrival orders) it is not possible to assume that the data for the corresponding join operation will be structurally sorted in a desirable manner. Thus, once again, they are not applicable in a filtering and matching environment.

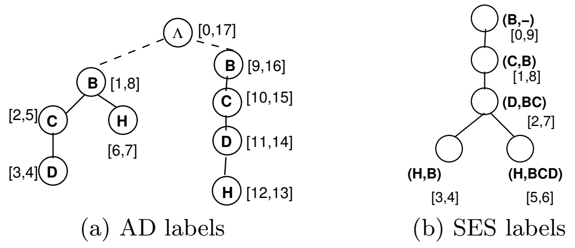


Fig. 5. (a) AD-labeling and (b) ViST-style SES-labeling of the same collection[26]

A natural alternative to sort-merge join based schemes, more suitable for filtering with unpredictable arrival patterns, is to rely on structural index joins, where the data nodes (not necessarily structurally sorted) are checked against a pre-existing index structure which can return required ancestor or descendant nodes in the data. Index-joins can be performed on unsorted streaming input data as long as appropriate index structures are available on the local data. There are a variety of existing index structures, such as B+-trees, XR-trees [16], XB-trees [19], and R-trees [24, 25] that can be used for indexing local data for efficient evaluation of structural conditions for identifying matching local data. The cost of this operation, per filtering result, $rMsg_i$ in the stream $resMsg$ is bound by the cost of the index access, which depends on the specific index structure used; but, generally, it is at least logarithmic in the local database size (depth of the index structure). When the arrival rate of the filtering results is high, on the other hand, the performance of existing index structures may not be sufficient (we experimentally evaluate this in Section 4). Therefore, for such operators to be useful in a real-time data filtering and matching middleware, they need to be implemented efficiently. In the next section, we propose a novel *clustered* matching approach addressing the needs of the $FMware$ middleware.

3 Cluster Support for Efficient Indexed Matching

To implement structural matching (or join) operations efficiently, most index structures, such as B+-trees, XR-trees [16], XB-trees [19], and R-trees [24, 25], assume an ancestor/descendant (AD) labeling scheme [22, 27] which assigns intervals to nodes such that descendant nodes have intervals that are contained within the intervals of the ancestor nodes. The AD interval of an XML node n_i *clusters* all descendants of n_i based on their common prefixes up to n_i (Figure 5(a)). Therefore, the ancestor/descendant relationship can be checked using *containment* or *enclosure* (= containment^{-1}) predicate on intervals. This renders interval-based AD labeling very common in implementation of structural joins.

3.1 Clustering Power and Precision

Since there is a one-to-one correspondence between the data nodes and the intervals, given an XML document, the number of interval labels assigned to it by an AD-based schemes is the same as the number of nodes in the data (Figure 5(a)). In contrast, in order to reduce the number of intervals that need to be considered, structure-encoded-sequence (SES) [28, 29, 26] based approaches try to further *cluster* structurally related nodes. They achieve this by using labeling schemes (such as Prufer sequences [29]) that can capture more than the ancestor/descendant relationships[1]. For example, Figure 5(b) shows ViST style [26] SES-labeling: each node $node_i$ is assigned a sequence seq_j and an SES interval $node_i.ses = (s_j, e_j)$. Once again, resulting intervals are either disjoint or contained within each other. However, as shown in Figure 5(b), some nodes with the same label are clustered under the same SES-label. For instance, the SES label $[2, 7]$ in Figure 5(b) clusters two nodes in the original data (Figure 5(a)), both with tag D. Note that these two nodes tagged[2] D are also on similar paths on both trees. In other words, each structure-encoded interval *clusters* multiple data nodes. Based on this example, we can state that SES labels have higher cluster power than the AD labels.

Definition 2. Clustering Power of an SES-label ($cps(ses, l, d)$). *The clustering power, $cps(ses, l, d)$, of an SES-label, ses, in a given data source, d, for the labeling scheme l is the number of nodes with this SES label.* ◇

Since clustering applies to nodes with the same tags, given a data collection and a labeling scheme, we can also define the clustering power of a given tag:

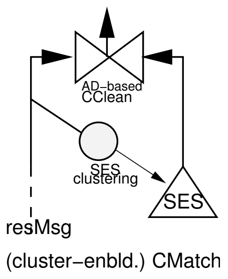
Definition 3. Clustering Power of a Tag ($cpt(\tau, l, d)$). *The clustering power, $cpt(\tau, l, d)$, of tag τ in a given data source, d, for the labeling scheme l is the average clustering power of the all SES labels corresponding to those nodes with tag τ.* ◇

Clearly using SES-labels of the nodes, as opposed to their AD-labels during matching can reduce the number of index checks that have to be performed. Furthermore, knowledge about the clustering powers of SES-labels and individual tags can enable the optimizer to decide whether a cluster-enabled scheme is likely to be effective (by reducing the number of inputs to consider) for a given matching condition. Thus we can benefit from the inherent clustering power of the SES-labels to reduce the number of times the existing index is accessed. However, this reduction in the number of index accesses do not come for free.

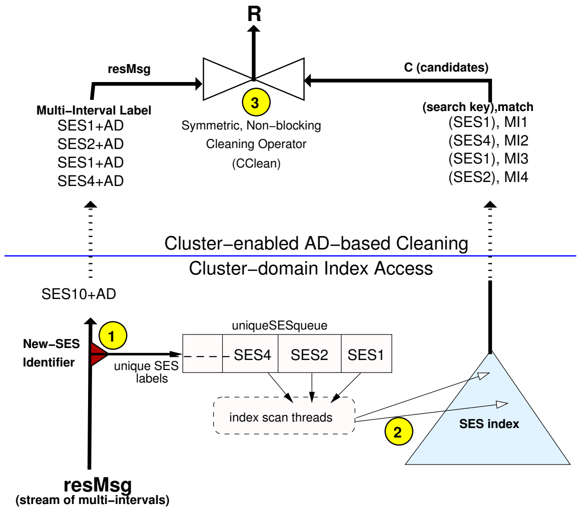
Unfortunately, SES-labels are not as precise as AD-labels in capturing structural relationships. In particular, unlike AD-labels, where $ancestor(node_i, node_j) \leftrightarrow contains(node_i.ad, node_j.ad)$, SES-labels satisfy only one direction of the implication: $ancestor(node_i, node_j) \rightarrow contains(node_i.ses, node_j.ses)$. Thus, a query of the form *"find all nodes with SES-labels contained within the SES label of a given node,"* might return more nodes than the descendants of the given node. Thus, although SES-labels can be used for clustering to reduce the number of

[1] There are a number of SES-labeling schemes. For instance, PRIX [29] uses Prüfer sequences, while [28] and other covering index based schemes consider path sequences. Details of SES-labeling processes have been omitted. Please refer to [28, 29, 26] for more details on SES-labeling schemes.

[2] A *tag* is the element name, attribute name, or the value associated with the XML node.



(a)



(b)

Fig. 6. (a) Clustered index matching in the SES domain is followed by an AD-domain cleaning phase: the faulty candidate nodes produced by the SES-based index scan are eliminated using AD-labels and (b) implementation of $\mathcal{C}$Match: nodes in $resMsg$ are first matched against indexed nodes using an SES-index, then candidates are cleaned using AD-labels; explained in Section 3.

disk accesses, to eliminate false retrievals that may result from the use of clustering, the filter-and-match operation would need a *cleaning phase*, based on the (unclustered) AD-labels (Figure 6(a)).

3.2 CMatch Operator for Multi-Labeled Matching and Cleaning

To enable both *efficient* and *correct* operation, $FMware$ middleware relies on a **multi-labeling** scheme which uses SES-labels for clustering and AD-labels to prevent false retrievals. A *Multi-Interval* (MI) label combines the AD and SES interval labels. Formally, the MI-label of $node_i$ with AD-label $node_i.ad$ and SES-label $node_i.ses$ is $node_i.mi = (node_i.ad, node_i.ses)$. FMware assigns MI-labels to each data node in the (global) XML data through exchange of structural information between distributed FMware entities.

Figure 6 provides an overview of the $\mathcal{C}$Match operator with which the $FMware$ middleware implements clustered index matching in the SES domain followed by an AD-domain cleaning phase: the faulty candidate nodes produced by the SES-based index scan are eliminated: *to reduce the number of accesses to the index, inputs are clustered based on their SES labels and the index is accessed only* **once** *for each SES label.* Since SES-based index access is not enough by itself to ensure correctness of the results, a symmetric, non-blocking $\mathcal{C}$Clean operator is used for cleaning the results from false hits.

Figure 6 shows the detailed implementation of the $\mathcal{C}$Match operator. The two complementary halves (cluster-domain index access and cluster-enabled cleaning) of the $\mathcal{C}$Match operator are described next.

3.3 Cluster-domain Index Access (Steps 1,2)

As shown in **Step 1** of Figure 6(b), the $\mathcal{C}$Match operator first identifies unique SES-labels observed in the input stream of filtering results. These unique labels are then used for accessing an index structure to fetch the matching nodes. (**Step 2** of Figure 6(b)). The details of these steps are as follows:

(Step 1. SES-clustering of Filter Results) The stream of inputs, $resMsg$, is passed through a $newSES$ label identifier, which identifies unique SES labels in the input nodes and pushes each **unique** SES label encountered in the stream into a queue, $uniqueSESqueue$. In our implementation, this process of unique SES identification is piggy-backed on the SES-hashing process used for cleaning, discussed later in Section 3.4.

(Step 2. SES-clustered Access to the Local Database) Using an existing SES index, the unique SES labels in the $uniqueSESqueue$ are compared against the matching condition Θ (more specifically Θ_{ses}) to identify candidate matches. The individual Θ_{ses} matching operations are performed by the SES index-access threads that are available in a thread pool. For each ses_{new} in the $uniqueSESqueue$, the SES index is accessed only once (i.e., the search key, ses_{key}, is equal to ses_{new}). The index returns a stream, $C(ses_{key})$, of candidates, where

$$C(ses_{key}) = \{\langle ses_{key}, rloc_j\rangle | \ (\Theta_{ses}(ses_{key}, node_{j,n}.ses) = true) \ \wedge \ (node_{j,n} \in rloc_j)\}.$$

Given a search key, ses_{key}, each candidate in this stream is a multi-interval label of the matching nodes. Each candidate is also marked with the search key, ses_{key}; this is used in the second phase of the algorithm for the AD-based cleaning operation which will clean false hits.

Note that multiple SES-index scan threads pipe their results into a single stream, C. Therefore, this stream contains results for different SES labels (potentially interleaved due to simultaneously outputting index scan threads).

3.4 AD-based Cleaning (Step3)

The AD-based cleaning operator (**Step 3** of Figure 6(b)) matches the candidates (C) returned by SES index lookup with the original results in the message filter stream ($resMsg$) based on their SES labels used for index accesses and performs AD-based cleaning on Θ (more specifically Θ_{ad}) to remove faulty candidates. Thus, the result in the output stream, R, consists of pairs, $\langle rmsg_i, rloc_j\rangle$, where $node_{i,m} \in rmsg_i$ and $node_{j,n} \in rloc_j$ satisfy the condition Θ_{ad}:

$$\begin{aligned} R = \{\langle rmsg_i, rloc_j\rangle \ | \ & (rmsg_i \in resMsg) \ \wedge \ (\langle ses_{key}, rloc_j\rangle \in C) \ \wedge \\ & (n_{i,m}.ses = ses_{key}) \ \wedge \qquad \text{/*match for SES-clustering*/} \\ & (\Theta_{ad}(node_{i,m}.ad, node_{j,n}.ad) = true)\} \quad \text{/*AD-based cleaning*/} \end{aligned}$$

Therefore, simultaneously with the SES-based index access by the $\mathcal{C}$Match operator, the original stream of filter results ($resMsg$) are passed to a symmetric, non-blocking operator for AD-based cleaning (**Step 3** of Figure 6(b)).

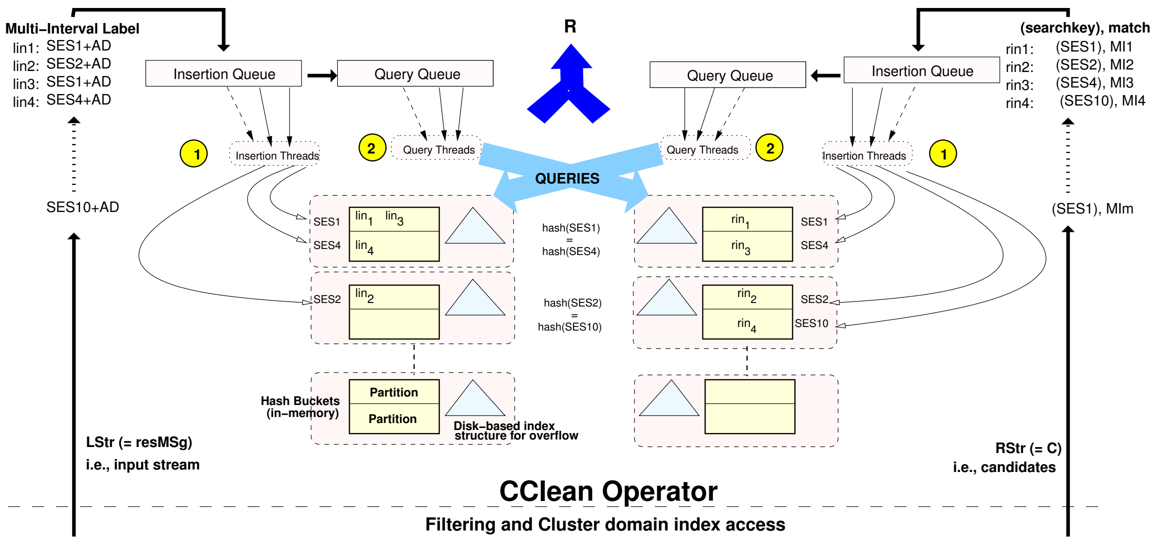


Fig. 7. Symmetric, non-blocking cleaning operator ($\mathcal{C}$Clean), explained in Section 3.4.

Before the AD-conditions can be checked for cleaning, however, $resMsg$ and C have to be **matched** based on the SES search key, ses_{key}, used for accessing SES clustered index. Unfortunately, neither of the streams is sorted or clustered on these SES values. In $resMsg$, results with the same SES values may be far apart from each other depending on how $resMsg$ has been computed prior to being passed to this operator. Similarly, since multiple SES-index access threads (each performing a separate index access with a different search key, ses_{key}) are writing onto the same stream C, the candidates are not likely to be clustered on the ses_{key} values to be used for cluster-based matching.

Overview of the $\mathcal{C}$Clean Operator The key for efficient cleaning, therefore, is symmetric non-blocking matching of tuples $n_{i,m} \in resMsg$ and $\langle ses_{key}, rloc_j\rangle \in C$, based on the condition, $(n_{i,m}.ses = ses_{key})$. Figure 7 shows the overview of the cluster-enabled cleaning operator, $\mathcal{C}$Clean. The two left and right input streams ($LStr = resMsg$ and $RStr = C$) to the operator are streams of multi-interval labels. Each candidate element in $RStr$ is also annotated with the SES-based search key, ses_{key}, used for fetching this candidate from the index.

Since correct candidates must satisfy both $\Theta_{ad}(node_{i,m}.ad, node_{j,n}.ad) = true$) and $n_{i,m}.ses = ses_{key}$, inputs to the cleaning operator are clustered, using hash tables, based on their SES-labels: (a) for the nodes in $LStr$, the corresponding SES labels and (b) for the candidates in $RStr$, the search key (ses_{key}) used for fetching the candidate from the index are used for hashing. For those node pairs which satisfy the SES equality, the AD condition will be checked for cleaning. In a sense, SES labels are used for clustering the input nodes (and their AD labels), thereby reducing the number of nodes that need to be compared based on their AD-values.

Most importantly, to prevent the clustering phase itself from becoming a bottleneck, the $\mathcal{C}$Clean operator achieves both SES-based clustering and the AD-based joins *simultaneously.* To facilitate this, the operator keeps an **in-memory hash structure** for each of its input streams. Inputs are inserted into these hash structures based on their SES-labels:

– Each hash bucket contains inputs that have the same $hash(ses)$ value. Note that since multiple SES labels may have the same hash value, a bucket may contain input nodes with different SES labels.
– Partitions in buckets cluster nodes with the same SES.
– To achieve the non-blocking behavior, each stream queries the other one without waiting for all the data with the same SES label being available.

To prevent duplicate results, timestamps are associated with the inputs being inserted into hash tables.

Since the in-memory space allocated to each $\mathcal{C}$Match operator is limited, in addition to the in-memory hash tables, the operator also maintains **disk-based AD index structures** to manage overflow buckets. Note that these index structures are used for identifying descendants or ancestors, depending on which input stream they are indexing. Candidate index structures include XR-trees [16] and R-trees [24, 25]. When a new SES label is hashed into a bucket with no empty partition, a victim partition is selected based on the *fullest least-recently-used* principle: the *fullest* partition that has not been used for the *largest* duration of time is selected to be the victim to be pushed onto the disk. In addition, if a partition becomes full and there is a new AD label to be inserted to that partition, it is pushed to disk (the *physical* partitions have a fixed size).

We are now ready to describe the functioning of the cluster-enabled hash-based cleaning operator, $\mathcal{C}$Clean, depicted in Figure 7 in detail.

$\mathcal{C}$Clean Algorithm Figure 7 demonstrates the implementation and the operation of the symmetric, non-blocking $\mathcal{C}$Clean operator. The operator is non-blocking in that input streams are queued and processed as they are received. It is also symmetric in that the operator consumes and processes both its input streams, $LStr$ and $RStr$, simultaneously. The inputs to the operator are two streams, $LStr$ and $RStr$, of multi-interval labeled nodes ($LStr = resMsg$ and $RStr = C$).
(Step 1.) Data from both input streams are queued for insertion into the corresponding hash tables based on the corresponding SES labels as described above. A time-stamp that specifies the insertion time, called *insertion time-stamp*, is associated with each element in the hash table. These insertion time-stamps are used to prevent duplicate results. The insertion process is carried out by threads that are available in the corresponding insertion-thread pools.

(Case 1) If there is already a partition in the corresponding hash table with the SES label and the partition is not full, then the input is time-stamped and inserted into the partition. If the partition is already full, the partition is pushed to the disk, along with the new (time-stamped) input. The preempted partition is made available as a free partition.

(Case 2) If there is no partition in the corresponding hash bucket with the SES label of the input, then an empty partition is allocated. If there is also no empty partition, then a victim partition is pushed to the disk (into the disk-based index structures) and the partition is allocated for this SES. The input is time-stamped and inserted into the partition.
(Step 2.) Hashed or indexed inputs on both sides are pushed into the respective query queues to initiate AD-join queries on the other stream. These queries are executed by the query threads that are available in the query-thread pools. Each query thread initiates a query on the other stream:

1. If the corresponding SES partition is found on the in memory hash table of the other stream, the thread first performs an in-memory AD-join based on the AD condition, Θ_{ad}.
2. The thread, then, consults an SES bitmap which specifies whether the corresponding SES partition is in the disk or not:
   (a) if the SES partition is found on the disk, the appropriate AD range query is performed on the *index structure* corresponding to the given SES label.
   (b) if the partition is not on the disk either, the AD-join is not performed as there are no matches.

In order to prevent duplicate results, only those pairs of inputs, lin and rin, whose insertion time-stamps satisfy the following condition are included in the output stream:

1. if the input, for which the query is initiated, is from the $LStr$ stream, then $ts_{lin} > ts_{rin}$, where ts denotes the insertion time-stamp.
2. if the input, for which the query is initiated, is from the $RStr$ stream, then $ts_{rin} \geq ts_{lin}$.

For each pair of matching lin and rin, the concatenated multi-interval list $\langle lin, rin \rangle$ is inserted into the output. $\diamond$

Dynamic Hash Bucket Allocation In $\mathcal{C}$Clean, nodes are hashed into the buckets based on SES-labels. All nodes having the same SES label are mapped into the same partition of the same bucket. If the variation in the clustering power of individual SES-labels in the data is extremely high (Section 3.1), it is possible that some buckets will be extremely full whereas others are relatively empty. Thus, instead of allocating fixed size buckets, $\mathcal{C}$Clean allocates memory to the hash buckets dynamically and goes to the disk **only** after all memory allocated for the cleaning task has been consumed. In our implementation, we are allocating memory dynamically to each bucket on per-need basis. Note that random hashing of the SES-labels ensures that the utilization of the in-memory pages is not low due to SES-labels with very low clustering powers.

Complexity of the $\mathcal{C}$Clean Operator If during $\mathcal{C}$Clean, data is found in the in-memory hash tables, the cost of search is negligible. Otherwise, a search has to be done in the corresponding overflow structure. In the following discussion of the complexity of the $\mathcal{C}$Clean operator, for simplicity, we will consider the *worst case* where all insertions and searches go to disk.

Let $\mathcal{I} = |resMsg|$ be the number of input elements, $Suniq$ be the set of unique SES labels in $resMsg$, and $\mathcal{C} = |C|$ be the number of candidates returned by index accesses. Let also $r \in resMsg$ and $c \in C$ be two input nodes. Ignoring in-memory hash tables, nodes r and c will both require disk access and initiate searches in the opposite structure.

Insertion cost: The insertion cost of input $r \in resMsg$ is determined by the number of nodes in $resMsg$, with the SES label $s_r = r.ses$, that are already received and indexed. In particular, if we denote the number of nodes in $resMsg$ with SES label s_r, $\mathcal{I}(s_r)$, the worst case insertion cost of r is $O(log(\mathcal{I}(s_r)))$. Thus, the total insertion cost for elements in $resMsg$ can be computed as

$$\mathbf{O}\left(\sum_{\mathbf{s}\in\mathbf{Suniq}}\mathcal{I}(\mathbf{s}) \times \mathbf{log}(\mathcal{I}(\mathbf{s}))\right).$$

The insertion cost of the candidate $c \in C$ with the corresponding SES search key k_c is, on the other hand, determined by the number of candidate nodes with the same SES key, k_c, already received in C. If the number of nodes in C with SES-based search key k_c is $\mathcal{C}(k_c)$, then the worst case insertion cost of c is $O(log(\mathcal{C}(k_c)))$. Since search keys are unique SES labels in $resMsg$, the total insertion cost for elements in C can be computed as $O\left(\sum_{s\in Suniq}\mathcal{C}(s) \times log(\mathcal{C}(s))\right)$. Thus, in the worst case, the insertion costs

$$\mathbf{O}\left(\sum_{\mathbf{s}\in\mathbf{Suniq}}\mathcal{I}(\mathbf{s}) \times \mathbf{log}(\mathcal{I}(\mathbf{s})) + \mathcal{C}(\mathbf{s}) \times \mathbf{log}(\mathcal{C}(\mathbf{s}))\right).$$

Search cost: Ignoring the in-memory hash tables, the overall worst case search cost (in terms of disk accesses) is

$$\mathbf{O}\left(\sum_{\mathbf{s}\in\mathbf{Suniq}}\mathcal{I}(\mathbf{s}) \times \mathbf{log}(\mathcal{C}(\mathbf{s})) + \mathcal{C}(\mathbf{s}) \times \mathbf{log}(\mathcal{I}(\mathbf{s}))\right).$$

Total cost: Based on these, we can compute the total $\mathcal{C}$Clean cost as

$$\mathbf{O}\left(\mathbf{2} \times \sum_{\mathbf{s}\in\mathbf{Suniq}}(\mathcal{I}(\mathbf{s}) + \mathcal{C}(\mathbf{s})) \times \mathbf{max}\{\mathbf{log}(\mathcal{C}(\mathbf{s})), \mathbf{log}(\mathcal{I}(\mathbf{s}))\}\right).$$

If an SES label in $resMsg$ clusters a large number of nodes or returns a large number of candidates, this label is likely to impose high cleaning cost. Nevertheless, index structures that maintain these intermediary nodes are likely to be smaller (and more efficient) than a large AD-index.

3.5 Complexity of the $\mathcal{C}$Match Operator

Since the clustering effect of the SES-labels reduces the number of requests that are sent to the SES index structure, higher clustering rates of SES-labels in the input stream would help the performance of the $\mathcal{C}$Match.

Cluster-domain index scan cost: Since the SES-domain index structure is accessed only once for each *unique* SES-label in the input, using the same notation as before, the access cost to the index structure could be written as

$$\mathbf{O}\left(\sum_{\mathbf{s}\in\mathbf{Suniq}}\mathbf{SES\_index\_access\_cost}\right).$$

AD-based cleaning cost: In Section 3.4, we computed the $\mathcal{C}$Clean operator used for AD-cleaning process as

$$\mathbf{O}\left(\mathbf{2} \times \sum_{\mathbf{s} \in \mathbf{Suniq}} (\mathcal{I}(\mathbf{s}) + \mathcal{C}(\mathbf{s}))\, \mathbf{max}\{\mathbf{log}(\mathcal{C}(\mathbf{s})), \mathbf{log}(\mathcal{I}(\mathbf{s}))\}\right).$$

Here, $\mathcal{I}(s) = \mathcal{I} \times rps(s, l, d)$, where rps denotes the relative clustering power of SES labels (in the data collection d and using labeling scheme l), as defined in Section 3.1. $\mathcal{C}(s) = match(\kappa, s)$ is the number of matches contained in the SES-interval s.

Total cost: We can compute the worst case overall cost (in terms of disk accesses) of the $\mathcal{C}$Match as the sum of the cluster-domain scan and AD-based cleaning costs given above. Since the two streams to $\mathcal{C}$Clean are processed in parallel, allocating independent resources to them would reduce the overall cleaning time. Similarly, since $\mathcal{C}$Clean is pipelined and non-blocking, cluster-domain index scan and AD-based cleaning phases can be performed *in parallel.* Thus, mostly, the observed execution time is only the maximum of the two phases, not their sum.

3.6 $\mathcal{C}$Match versus AD-only Matching

If the index scan was performed in the AD-domain rather than in the SES-domain, the total access cost to the existing AD index structure would be

$$\mathbf{O}\left(\sum_{\mathbf{s} \in \mathbf{Suniq}} \mathcal{I}(\mathbf{s}) \times \mathbf{AD\_index\_access\_cost}\right).$$

Since, for AD-only match, there is no need for cleaning, this is also the total cost of the AD-only match operation. One major advantage of $\mathcal{C}$Match (versus AD-only match) is that for a given SES label, s, the index structure is accessed only once for $\mathcal{C}$Match, whereas the index is accessed $\mathcal{I}(s)$ times for AD-only matches (see equations above) Furthermore, since SES indexes are more compact than AD indexes, it is likely that searches on the existing SES index structures will be faster than searches on the AD index structures.

However, the $\mathcal{C}$Match operator has an AD-based cleaning overhead that has to be accounted for. Computation of the size of in-memory space needed to hold incoming inputs and candidates is trivial using the statistics described above. However, when the in-memory space is not large enough, $\mathcal{C}$Clean operator needs to use disk-based structures. Comparing the **worst case** $\mathcal{C}$Clean cost and the AD-only match cost, we see that cluster-domain scan followed by cleaning is worthwhile as long as the accesses to clustered intermediary structures are cheaper than scans on the large AD index structure.

3.7 Per-Query and Per-SES Adaptation in FMware

Given a matching statement with two query tags, based on the cost models and statistics presented above, we can estimate whether $\mathcal{C}$Match or AD-only match will cost less. One can also choose between different SES-labeling schemes based on the clustering rates they provide. We refer to this as *per-query adaptation.* Note that, it is also possible to consider each node in the input stream individually based on $\mathcal{C}$Match or AD-only match on a per-input basis (Figure 8). Furthermore, if the expected number of candidates is large, cluster-domain processing provides further opportunities. In the next section, we show that the $FMware$ middleware benefits from both alternatives, based on available statistics.

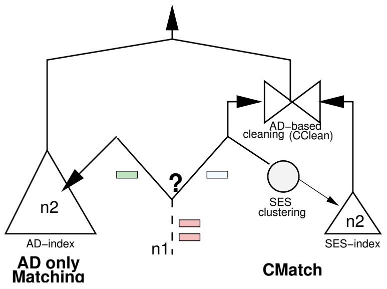


Fig. 8. Per-input choice between $\mathcal{C}$Match and AD-only index match in FMware

4 Experimental Evaluation

In this section, we experimentally evaluate the effectiveness of cluster-domain matching, by comparing performances of $\mathcal{C}$Match and AD-only match. In particular, we show that the $\mathcal{C}$Match exploits available (per matching) task memory significantly better than an AD-only match, especially when the clustering powers of the nodes are high. We also show that the relative performances of $\mathcal{C}$Match and AD-only index join follow the cost patterns discussed earlier in the paper, thus it is possible to choose between $\mathcal{C}$Match and AD-only match, case-by-case, based on easy to collect statistics.

4.1 Setup

The operators presented here have been implemented in Java and ran on Redhat 7.2 Linux workstations, with 1.8 GHz Pentium IV processor. For the AD- and SES- index structures, we used B-tree implementation of BerkeleyDB [30]. The overflow data in $\mathcal{C}$Clean is indexed on disk.

We compared the AD-based index match and the $\mathcal{C}$Match operators under varying conditions. Both AD-only match and $\mathcal{C}$Match operator implementations are non-blocking and pipelined for fair comparison. Note that AD-only match does not need a cleaning phase. Table 1 provides a diverse set of matching conditions, selected for inclusion here as they illustrate the behavior of the $\mathcal{C}$Match under various matching characteristics. The table reports the following parameters:

- `AD:SES`, denotes the number of unique SES and AD labels in the inputs,
- `Cl_Pow`, denotes the clustering power of the SES labels in the input stream,
- `Cand`, is the number of candidates generated by cluster-domain index scan,
- `Buffer`, denotes the buffer allocated for each operator (more specifically, BT is buffers for B-tree, HT is hash table size, and OT is buffers for overflow-index trees),
- `#BScan`, denotes the number of accesses to existing (AD or SES) indexes,
- `#Omiss` is the number of misses from the overflow-index tree buffers
- `Exper.Tot.` is the execution time for the entire experiment
- `Per Msg. Avg.` is the average time (per message) required for matching. This is what we would like to have as small as possible.

	AD:SES(**Cl_Pow**) InStream	AD:SES LocalDB	Matching Operator	Buffer(MB) Total	(BT+HT+OT)	Performance Indicators #BScan	Cand	#Omiss	**Time(ms.)** Exper.Tot.	**Per Msg. Avg.**
• No Clustering: AD-only matching is expected to perform better										
MQ1	1:1	5821:5280	ADMatch	1.11MB	(1.11+0+0)	1	n/a	n/a	498	498ms
	(**1**)		$\mathcal{C}$Match		(1+0.1+0.01)	1	5821	~22K	~87K	~87K ms = 87s
			ADMatch	2.1MB	(2.1+0+0)	1	n/a	n/a	455	**455ms**
			$\mathcal{C}$Match		(1+ 1+ 0.1)	1	5821	8	941	941ms
• Very Low Clustering Power: AD-only or $\mathcal{C}$Match										
MQ2	5821:5280	1:1	ADMatch	1.11MB	(1.11+0+0)	5821	n/a	n/a	558	0.096ms
	(**1.1**)		$\mathcal{C}$Match		(1+0.1+0.01)	5280	1	~22K	~86K	14.77ms
			ADMatch	2.1MB	(2.1+0+0)	5821	n/a	n/a	554	**0.095ms**
			$\mathcal{C}$Match		(1+ 1+ 0.1)	5280	1	8	490	**0.085ms**
• High Clustering Power: $\mathcal{C}$Match is expected to perform better										
MQ3	222:1	2:1	ADMatch	1.011MB	(1.011+0+0)	222	n/a	n/a	182	0.82ms
	(**222**)		$\mathcal{C}$Match		(1+0.01+0.001)	1	2	411	443	1.99ms
			ADMatch	1.11MB	(1.11+0+0)	222	n/a	n/a	220	0.99ms
			$\mathcal{C}$Match		(1+0.1+0.01)	1	2	8	65	**0.29ms**
MQ4	12897:1	4:2	ADMatch	2.1MB	(2.1+0+0)	12897	n/a	n/a	~ 6K	0.465ms
	(**12897**)		$\mathcal{C}$Match		(1+1+0.1)	1	4	~17K	~172K	13.33ms
			ADMatch	3.2MB	(3.2+0+0)	12897	n/a	n/a	~ 6K	0.465ms
			$\mathcal{C}$Match		(1+2+0.2)	1	4	8	722	**0.056ms**
• High Clustering Power: $\mathcal{C}$Match is expected to perform better										
MQ5	463:224	5821:5280	ADMatch	1.011MB	(1.011+0+0)	463	n/a	n/a	~30K	64.8ms
	(**2.07**)		$\mathcal{C}$Match		(1+0.01+0.001)	224	2	~1K	~1K	2.16ms
			ADMatch	1.11MB	(1.11+0+0)	463	n/a	n/a	~ 30K	64.8ms
			$\mathcal{C}$Match		(1+0.1+0.01)	224	2	8	115	**0.25ms**
MQ6	12897:1	5821:5280	ADMatch	1.11MB	(1.11+0+0)	12897	n/a	n/a	~900K	69.78ms
	(**12897**)		$\mathcal{C}$Match		(1+0.1+0.01)	1	5790	~82K	~506K	39.23ms
			ADMatch	2.1MB	(2.1+0+0)	12897	n/a	n/a	~900K	69.78ms
			$\mathcal{C}$Match		(1+ 1+ 0.1)	1	5790	~17K	~190K	**14.73ms**

Table 1. MQ1-MQ6 are the various matching queries used for comparing $\mathcal{C}$Match and AD-only match. $\mathcal{C}$Match can exploit available memory significantly better than AD-only index match, especially when the clustering powers are high.

In these experiments, we report results based on ViST-style SES-labels [26] and traditional Dietz style AD-labels [27]. As the local as well as streaming data, we used fragments of the DBLP XML data from [31].

Buffers are allocated (and varied) in such a way that AD-based matching and $\mathcal{C}$Match operators get to use the same amount of memory:

$$\underbrace{buf\_AD\_Btree}_{AD\ only\ index\ match} = \underbrace{\overbrace{buf\_SES\_Btree}^{1MB} + mem\_Hash\_Table + \overbrace{buf\_overflow\_index}^{0.1\times mem\_Hash\_Tables}}_{\mathcal{C}Match}$$

Since the index structures for SES-labels tend to be smaller than the index-structures for AD-labels, to evaluate the worst case behavior for the $\mathcal{C}$Match operator, we significantly constrain the available buffer (1-3MB per $\mathcal{C}$Match operator). This also reflects the situation observed in practice, where there are multiple filter-and-match operations to be processed in $FMware$ middleware and where the available buffer has to be shared among $\mathcal{C}$Match operations.

Finally, in the setup we used, the cost of each page miss was around $10ms$ and the access and processing cost for hits in the buffers was around $0.01ms$.

4.2 Experiment Results

Table 1 compares $\mathcal{C}$Match against AD-only matching for queries with different characteristics, including degrees of clustering of the involved tags and degree of expected candidates that need to be cleaned. Table 1 also presents results under constrained and non-constrained buffer availabilities for each query.

(MQ1) In this case, the input nodes in the filtered message stream have no clustering power. Since SES-based clustering **is not** applicable, as expected, AD-only match is relatively faster (though both alternatives are costly and realtime filtering and matching may not be applicable).

(MQ2,MQ3,MQ4) In these cases, the input stream of filtered nodes have some clustering power. On the other hand, for all three cases, the number of matching nodes in the local databases (and thus the candidates returned by the index accesses) are low.

In all three cases, the costs of the $\mathcal{C}$Match operator depends on whether the hash table is large enough for the required cleaning operation: If the hash table used during the cleaning phase is large enough to balance the expected number of hash misses with the savings from the access to the large B-tree index structures, then even a very low 1.1 clustering power can lead to savings. Note that in all three cases, the amount of hash-space allocated was less than the amount of buffer allocated for the AD-based index structures; in other words, when the clustering power is non-negligible $\mathcal{C}$Match uses the available memory more effectively than AD-only matching. Furthermore, the degree of saving increases predictably with the clustering power of the filtered nodes.

(MQ5) In this case, the clustering power of the nodes in the input stream is non-negligible ($\sim$2) and the number of relevant nodes in the local index structure is relatively high. However, the number of candidates returned by SES scan for cleaning is relatively low.

The clustering power ($\sim$2) of the input stream ensures that the number of index scans for $\mathcal{C}$Match is only half of those of AD-only match. *Thus leads to significant savings even with a relatively small hash table.*

(MQ6) In this case, the clustering power of the nodes in the input stream, the relevant number of nodes in the local database, as well as the number of matching candidates that are returned by the SES-scan are all high.

Due to its clustering power, $\mathcal{C}$Match provides significant savings, even when it has to rely on disk-resident trees in the cleaning phase. Note that since the index structures (used for efficient access to the overflow buckets) are significantly smaller than the B-trees used for AD-only matches, $\mathcal{C}$Match is able to provide better results even when the number of overflow-index access is significantly larger than the number of AD-index accesses.

Summary and Discussions: The experiment results show that

– cluster-domain processing helps the performance of $FMware$ *by significantly reducing the total number of disk accesses to the local index structure*; and

– $\mathcal{C}$Match exploits available memory very effectively. In the experiments, increasing the buffer available for the AD index did not help reduce the cost of AD-only index match, yet when the same amount of increase is provided to the $\mathcal{C}$Match, we observed significant reductions in cost.

5 Related Work

In addition to the discussions in the Introduction, here we provide an overview of the work in adaptive query processing and index supported XML processing.

5.1 Adaptive Query Processing with Relational Data

In the relational domain, continuous query processing with unpredictable data arrival characteristics has been studied from various angles. Telegraph [32], for instance, is a dataflow engine which recognizes that cost of the operators, their selectivities, and the rates at which tuples arrive from the input vary during the processing of queries. Thus it routes data through operators adaptively, based on arrival characteristics. Aurora [33, 34] focuses on QoS- and memory-aware operator scheduling and load shedding for coping with transient spikes in data.

Other works, which focus on adaptive query processing for continuous queries include [35–37]. Especially in the distributed relational query processing context, it has been long recognized that variations in the data arrival rates necessitate special join operator implementations. In particular, XJOIN [38] and HM-Join [39] are two non-blocking join operators suitable for deployment in systems where data with, high variable arrival rate, from remote sources have to be joined. The algorithms rely on symmetric non-blocking hash-joins.

5.2 Index- and Multi-Index Support for XML Processing

Structural join schemes sometimes exploit on-the-fly-created index structures (such as B^+ trees or its augmented variations [19, 40, 16], R trees [24, 25]) to skip unpromising ancestor (descendant) elements. DataGuides [28], IndexFabric [41], T-Index [42], BLAS [43], FB-Index [44], XJoinIndex [45], APEX [46], and other covering indices [47], on the other hand, use pre-computed indexes. A DataGuide [28] is a structural summary of the database, and provides an efficient mechanism to enumerate matching nodes when a tag path starting from the root is given as input. T-Index is also tailored to identify nodes matching a given path template, but paths are not limited to those starting from the root. IndexFabric indexes trees in a hierarchy of Patricia tries, reducing the number of disk accesses needed to find paths satisfying a path expression[41]. APEX[46] is similar to DataGuides and T-Indexes, but it only maintains frequent paths.

[20] notes that a combination of XML indexing methods can be useful for improving *stream*-based processing of structural queries, since different schemes are better for different classes of XML twig patterns. Similarly, in (XDG) [28], node labels are indexed by a term index *T-Index*, which gives the sequence of all nodes with the same label in the XDG. A second index, called *P-Index*, which is a path index, is used to determine the instances of a certain rooted tag path and also to identify the addresses of the physical data locations in an efficient way. ViST [26] also uses two index structures, namely *S-Index* (for SES-based labels) and *D-Index* (for ancestor-descendent labels). BLAS [43] uses a similar observation to develop a bi-labeling system for reducing the number of joins.

6 Conclusion

XML message filtering systems may need to match results with local data to identify those relevant for higher-level processing. We presented a $FMware$ middleware for performing filtering in the presence of locally stored data which need to be matched against filter results. The $\mathcal{C}$Match operator, underlying $FMware$, obtains its efficiency from the clustering effect of the structure-encoded labels, which significantly reduces the number of secondary storage accesses required for accessing the locally stored data. The operator also has a highly efficient, non-blocking cleaning phase to remove any spurious results that may have been created due to the imprecise clustering of structure-encoded labels. We experimentally showed that this approach provides significant savings in filter result validation time by reducing the total number of disk accesses to the local data.

References

1. JMS: Sun microsystem inc. http://java.sun.com/products/jms (2006)
2. Mohan, C., Dievendorff, D.: Recent work on distributed commit protocols, and recoverable messaging and queuing. IEEE Data Eng. Bull. **17**(1) (1994) 22–28
3. IBM-MQSeries. www.ibm.com (2005)
4. Letz, S., Zedler, M., Thierer, T., Schutz, M., Roth, J., Seiffert, R.: Xml offload and acceleration with cell broadband engine. In: XTech: Building Web 2.0. (2006)
5. Papaemmanouil, O., Çetintemel, U.: Semcast: Semantic multicast for content-based data dissemination. In: ICDE. (2005) 242–253
6. Anand, A., Chawathe, S.S.: Cooperative data dissemination in a serverless environment. In: CS-TR-4562, University of Maryland, College Park. (2004)
7. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: Niagaracq: a scalable continuous query system for internet databases. In: SIGMOD. (2000)
8. DataPower: Xs40 XML firewall. http://www.datapower.com/products/ (2006)
9. Sarvega: Xml security gateway. http://www.sarvega.com/xml-guardian-gateway.html (2006)
10. DataPower: Xs40 XML router. http://www.datapower.com/products/ (2006)
11. Sarvega: Xml context router. http://www.sarvega.com/xml-context.html (2006)
12. Diao, Y., Franklin, M.: Query processing for high-volume xml message brokering. In: VLDB. (2003)
13. Candan, K., Hsiung, W.P., Chen, S., Tatemura, J., Agrawal, D.: Afilter: Adaptable xml filtering with prefix-caching and suffix-clustering. In: VLDB. (2006)
14. Josifovski, V., Fontoura, M., Barta, A.: Querying xml streams. The VLDB Journal **14**(2) (2005) 197–210
15. Peng, F., Chawathe, S.S.: Xsq: A streaming xpath engine. In: CS-TR-4493, University of Maryland, College Park. (2003)
16. Jiang, H., Lu, H., Wang, W., Ooi, B.C.: XR-Tree: Indexing XML data for efficient structural joins. In: ICDE. (2003)
17. Xpath. http://www.w3.org/TR/xpath (1999)
18. Xquery. http://www.w3.org/TR/xquery (2006)
19. Bruno, N., Srivastava, D., Koudas, N.: Holistic twig joins: Optimal XML pattern matching. In: SIGMOD. (2002)
20. Chen, T., Lu, J., Ling, T.: On boosting holism in xml twig pattern matching using structural indexing techniques. In: SIGMOD. (2005)

21. Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M.: Structural joins: A primitive for efficient XML query pattern matching. In: ICDE. (2002)
22. Li, Q., Moon, B.: Indexing and querying XML data for regular path expressions. In: VLDB. (2001)
23. Jiang, H., Wang, W., Lu, H.: Holistic twig joins on indexed XML documents. In: VLDB. (2003)
24. Chien, S.Y., Tsotras, V.J., Zaniolo, C., Zhang, D.: Efficient complex query support for multiversion XML documents. In: EDBT. (2002)
25. Grust, T.: Accelerating XPath location steps. In: SIGMOD. (2002)
26. Wang, H., Park, S., Fan, W., Yu, P.: ViST: A dynamic index method for querying XML data by tree structures. In: SIGMOD. (2003)
27. Zhang, C., Naughton, J.F., DeWitt, D.J., Luo, Q., Lohman, G.M.: On supporting containment queries in relational database management systems. In: SIGMOD. (2001)
28. Bremer, J., Gertz, M.: An efficient XML node identification and indexing scheme. In: VLDB. (2003)
29. Rao, P., Moon, B.: PRIX: Indexing and querying xml using Prüfer sequences. In: ICDE. (2004)
30. BerkeleyDB. http://www.sleepycat.com/ (2006)
31. UW XML Repos. http://www.cs.washington.edu/research/xmldatasets/ (2006)
32. M.A.Shah and S.Chandrasekaran: Fault-Tolerant, Load-Balancing Queries in Telegraph. SIGMOD Record **30**(2) (2001)
33. Carney, D., Cetintemel, U., Cherniack, M., Lee, C.C.S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Monitoring Streams-A New Class of Data Managment Applications. In: VLDB. (2003)
34. Tatbul, N., Cetintemel, U., Zdonik, S.B., Cherniack, M., Stonebraker, M.: Load Shedding in a Data Stream Manager. In: VLDB. (2003)
35. S.Babu, *et al.*: Adaptive ordering of pipelined stream filters. In: SIGMOD. (2004)
36. Tian, F., DeWitt, D.: Tuple Routing Strategies for Distributed Eddies. In: VLDB. (2003)
37. Carey, M.J., Lu, H.: Load Balancing in a Locally Distributed DB System. SIGMOD Record **15**(2) (1986) 108–119
38. Urhan, T., Franklin, M.J.: XJoin: Getting fast answers from slow and bursty networks. Technical Report CS-TR-3994 (1999)
39. Mokbel, M., Lu, M., Aref, W.: Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In: ICDE. (2004)
40. Chien, S.Y., Vagena, Z., Zhang, D., Tsotras, V., Zaniolo, C.: Efficient structural joins on indexed XML documents. In: VLDB. (2002)
41. Cooper, G.R.H.B., Franklin, M.J., Shadmon, M.: A fast index for semistructured data. In: VLDB. (2001)
42. Milo, T., Sicuo, D.: Index structures for path expressions. In: ICDT. (1999)
43. Yi Chen and Susan Davidson and Yifeng Zheng: BLAS: An Efficient XPath Processing System. In: SIGMOD. (2004)
44. Kaushik, R., Bohannon, P., Naughton, J., Korth, H.: Covering indexes for branching path queries. In: SIGMOD. (2002)
45. Bertino, E., Catania, B., Wang, W.Q.: XJoin Index: Indexing XML data for efficient handling of branching path expressions. In: ICDE. (2004)
46. Chung, J.M.C., Shim, K.: Apex: An adaptive path index for xml data. In: ACM SIGMOD. (2002)
47. Ramanan, P.: Covering indexes for xml queries: Bisimulation - simulation = negation. In: VLDB. (2003)