FLUX: Content and Structure Matching of XPath Queries with Range Predicates*

Hua-Gang Li S. Alireza Aghili Divyakant Agrawal Amr El Abbadi
University of California at Santa Barbara
{huagang, aghili, agrawal, amr}@cs.ucsb.edu

Abstract. Range queries seek the objects residing in a constrained region of the data space. An XML range query may impose predicates on the numerical or textual contents of the elements and/or their respective path structures. In order to handle content and structure range queries efficiently, an XML query processing engine needs to incorporate effective indexing and summarization techniques to efficiently partition the XML document and locate the results. In this paper, we describe a dynamic summarization and indexing method, FLUX, based on Bloom filters and B^+-trees to tackle these problems. We present the results of extensive experimental evaluations which indicate the efficiency of the proposed technique.

1 Introduction

XML has gained wide acceptance as an emerging standard and is being employed as a key technology for data exchange, integration and storage of semi-structured data. The XML data model, due to its rich presentation (content and semi-structuredness), poses unique challenges to effectively support complex queries. Powerful and flexible query capabilities have been developed [1, 2, 6, 8, 13, 17–19, 21, 23] to extract structural patterns from XML documents. These techniques are mainly based on the structural join by using some encodings on XML document elements. Queries on such ordered XML trees often impose predicates on the content of `ELEMENT` *labels* (keyword search) and/or their corresponding *structural relationships* (structural pattern search). These queries require the presence of some keywords in the document tree along with the conformation of the keyword instances with some structural patterns, which might be a specific linear path structure or a subtree/twig structure in the underlying data. For instance, `Q = /dblp//article/[2004` $\leq$ `year` $\leq$ `2005]` represents such a query with a linear path structure, which matches all the journal articles published between year `2004` and `2005` from the `dblp` [15] bibliography database. In addition, approximate top-k matching of XML queries were studied in [20].

XML query languages [7, 11] provide support for *content-and-structure* (CAS) class of queries. Additionally, full-text keyword search techniques [5] have been added to XML query languages to support more sophisticated full-text content retrieval. Furthermore, the XQuery and XPath query languages provide support for queries with *range predicates* which are also one of the fundamental functionalities supported by general database query processing engines. In this paper, the class of content-and-structure (CAS) single path queries are extended to include (*i*) *range* predicates over content, as well as (*ii*) *structure* predicates and,

* This research was supported by the NSF under IIS-0223022 and CNF-0423336.

furthermore, efficient techniques are proposed for processing them. We refer to them as *XPath range* queries.

The efficient evaluation of such XPath range queries is determined by the choice of an efficient execution and data access plan which is one of the critical responsibilities of the database optimizer. For instance, consider a possible query plan for `Q` where the query engine has to perform the XPath range query `Q′ = /dblp//article/[year = 2004 OR year = 2005]` to find all the journal articles published in the year `2004` or `2005`. The `dblp` dataset contains 111,609 instances of the `[/article/year]` path structure and only 259 instances of `[/year/2005]`. That is the `[year = 2005]` predicate will return 259 instances while the structure predicate `[/dblp//article /year]` results in 111,609 instances. Hence, it is essential to utilize the *selectivity*[1] of the structural elements for efficient evaluation of the XPath range queries. An efficient query execution plan should apply the evaluation starting at the more selective segments of the query. However, one of the main challenges involved in such execution plans for XPath range queries is that range predicates, which happen to be more selective in this case, typically involve the leaf level of the XML document tree. Moreover, pushing the evaluation down to the leaves of the tree should be accompanied with the appropriate leaf indexing techniques to avoid inspecting a large number of leaf nodes. It is clear that plans such as `Q′` do not utilize the common optimization technique of pushing down the `selection` operation down to the leaves of the query plan tree. Ignoring the selectivity of the path elements results in the exponential growth of the intermediate result set which must be retrieved from the database. We argue that it is essential to utilize effective summarization and indexing techniques to reduce the search space based on the content and most *selective* elements of the XML document collections.

In this paper, we develop an XML query processing system for XPath range queries named FLUX. FLUX employs an efficient B^+-tree based index structure to locate the leaf matches to the range predicate of a query in its initial stage. Each leaf match, n_i, of the document tree stores a *compact path signature* of the root-to-leaf path structure ending at n_i, using the notion of Bloom filter [4]. In the next step, the path signatures of each matched leaf instance n_i is compared with the query's path signature to eliminate those instances whose path signatures are very different from that of the query. To the best of our knowledge, this is the first attempt to specifically address the matching of XPath queries with range predicates in XML document collections. The main features of FLUX are summarized as follows:

- An efficient B^+-tree based indexing scheme is constructed on the *indexable* (e.g., textual, numerical, date, etc.) elements/attributes of the XML document for effective retrieval and matching of the query's range predicate.
- FLUX incorporates a novel bit-wise hashing scheme based on the notion of Bloom filter on `ELEMENT` and `ATTRIBUTE` contents of XML document trees. A family of hash functions are applied on the path components where each path is summarized to a compact bit vector signature. As a result, the path

[1] The fraction of the structural elements that satisfy the predicate.

matching can be performed very efficiently through the comparison of path signature bit vectors.
- Extensive experimental evaluations demonstrate the effectiveness of FLUX for XPath range queries on real and synthetic XML datasets.

The rest of the paper is organized as follows: Section 2 presents the problem definition. Sections 3 and 4 provide the descriptions of range and path matching procedures, respectively. Section 5 finalizes the FLUX algorithm followed by Section 6 which provides the experimental results and analysis. Section 7 concludes the work.

2 Problem Formulation

XML documents are rooted ordered tree structures where each node in the document tree corresponds to the document's `ELEMENT`, `ATTRIBUTE`, or `TEXT` nodes. The `TEXT` nodes represent the values of their parent `ELEMENT` nodes, and `ATTRIBUTE` nodes introduce branches of their parent `ELEMENT` nodes. In this paper, we focus on simple XPath Range expressions which are defined as follows:

Definition 1. (*XPath Range Expression*). *A simple XPath expression* $p = e_1t_1e_2t_2 \ldots e_k\mathtt{R}$ *is called an* XPath range expression *(XPR), where* e_i *denotes an Ancestor-Descendant (AD, //) or Parent-Child (PC, /) edge, and* t_i *denotes the tag of an* `ELEMENT` *or* `ATTRIBUTE`*, and* `R` *represents a range predicate (sentinel) over an* indexable *element/attribute (e.g., numerical, textual, date, etc.), respectively.*

Example 1. q_1 = `/dblp//article/[2004` $\leq$ `year` $\leq$ `2005]`, and q_2 = `/management//employee/[90K` $\leq$ `salary` $\leq$ `100K]` represent XPR expressions on `dblp` and an employee database, respectively. For instance, in q_1: e_1 `= /,`t_1 `= dblp,`e_2 `= //,` t_2 `= article,` e_3 `= /`, and `R = [2004` $\leq$ `year` $\leq$ `2005]`.

Definition 2. (*Path Signature*). *Assume that* $p = e_1t_1e_2t_2 \ldots e_kt_k$ *is an XPath expression, where* t_k *is an indexable element/attribute, and* HF *is a family of hash functions, which map each tag of* p *onto a set of hash values. The hash values of the tags* $(t_1, t_2, \ldots, t_k)$ *are collectively combined*[2] *to construct a single bit-vector signature for the path structure* p*.*

Given an XML dataset and an XPath range expression, we need to locate and retrieve all the qualifying matching instances. Matching the query against an XPR instance of the dataset involves comparing their corresponding path structures and evaluating the range predicate over the instance. The range predicate match of the sentinel R of the query expression seeks all the corresponding instances in the database having sentinel r such that $r \in R$. For instance, considering the query q_1 (from Example 1), this phase corresponds to locating all the instances of the *year* attribute being in the range of [2004, 2005], which are referred to as *range-matched* instances. Furthermore, the path structure signatures of all the range-matched instances are compared against the query's path

[2] Details are discussed in Section 3. Intuitively speaking, it is to use these hash values to set the bits in a bit-vector.

structure signature. If the path structure signature is close enough (similarity measurement as defined in Definition 3) to the query's path structure signature, the path structure will be further checked against the query's structure to determine whether it is an exact match to be finally reported as answer to the query.

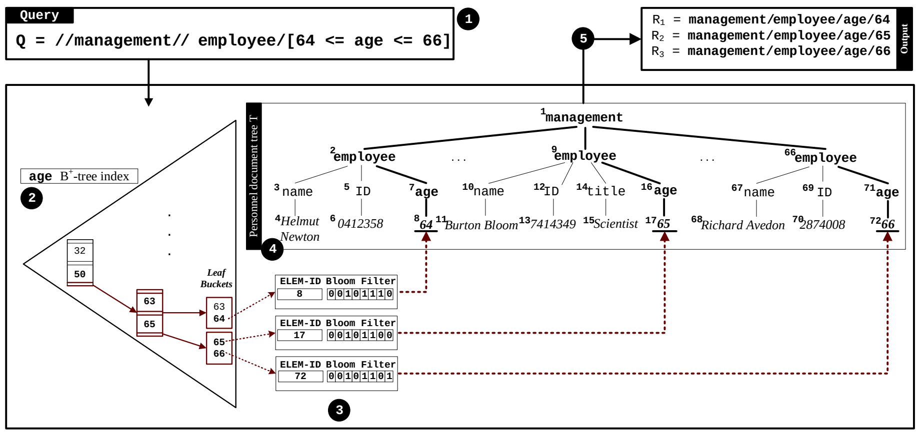

Fig. 1. FLUX Search model.

Definition 3. **(*Candidate Matching Instance*).** *Given the XPR expression $q = e_1t_1e_2t_2 \ldots e_tR$, and any XPR range-matched instance $p = e'_1t'_1e'_2t'_2 \ldots e'_{t'}r$ of the database, let HF denote a family of hash functions which map a path structure onto a bit-vector. Moreover, let $f{:}u \rightarrow 2^{\aleph}$ denote a function on bit-vectors which returns the set of all indices of the "set" bits of any bit-vector u. Then, the XPR range-matched instance p is called a candidate matching instance to q, if*

$$f(\,HF(e_1t_1 \ldots e_t)\,) \subseteq f(\,HF(e'_1t'_1 \ldots e'_{t'})\,), and\ r \in R.$$

For instance, given two path structures q and p, where $\texttt{HF}(q) = 100001$ and $\texttt{HF}(p) = 101101$, then p is called a candidate matching instance of q because $f(\texttt{HF}(q)) = \{1,\ 6\} \subseteq f(\texttt{HF}(p)) = \{1,\ 3,\ 4,\ 6\}$.

Definition 4. **(*Path & Range Components*).** *An XPath range expression $p = e_1t_1e_2t_2 \ldots e_kR$ consists of two main components, a* ***path expression component*** *denoted by $Q^\rho = e_1t_1e_2t_2 \ldots e_k$ and a* ***range predicate (sentinel) component*** *$Q^\eta = R$.*

Example 2. The XPath range expression q_2 = `/management//employee/[90K` $\leq$ `salary` $\leq$ `100K]` consists of two components: the *path expression component* Q^ρ = `/management//employee/salary` and the *range predicate component* Q^η = `90K` $\leq$ `salary` $\leq$ `100K`.

Given an XPR query Q, FLUX proceeds in two different phases, (i) finding the regions in the database satisfying the range predicate component Q^η of the query (**range matching**), and (*ii*) matching the query path component Q^ρ against the *range-matched* instances of the database (**path matching**). Range matching is the initial step and the results of this stage are passed to the path matching phase for structure matching and refinement of the answers. The following sections provide the details of the range and path matching procedures.

3 Range Matching

Any range query may benefit from efficient indexing mechanisms to quickly locate and retrieve the intersecting portions of the database satisfying the range predicate. Popular indexing techniques such as B^+-trees and R-trees have been extensively applied to alleviate such problems in the general context of range predicate queries. The *range matching phase* of FLUX employs an indexing technique based on B^+-trees on the *range predicate component* Q^η of the query for the effective reduction of the search space.

An offline procedure constructs a B^+-tree index on the *indexable elements/ attributes* (e.g, numerical, textual, date, ...) of the XML document. Part 2 in Figure 1 depicts a portion of one such index tree, constructed on the `age` element of a typical XML employee database. For instance, the last leaf bucket stores the `age` content information for two existing `age` values 65 and 66 in the database. Each instance (e.g. `age`$=$ 66) also holds the bit-vector signature of the actual path component leading to this node (details provided in the next section), and its corresponding `ELEM-ID` information. The `ELEM-ID` is the *preorder traversal rank* of the corresponding node in the actual XML document. For instance, the node instance with `age` $=$ 66 has preorder rank of 72, which is shown in the document tree of Figure 1, named as the node $^{72}66$. Note that, each individual occurrence of an internal or leaf node has a unique preorder value.

It is important to note that our proposed encodings (as explained above) is different from the encoding schemes used in [1, 6, 11, 17]. Those encoding schemes associate interval/regional encoding with every node, based on the document order. For instance, each label may consist of (*start*, *end*, *level*) values for each node, acquired from the *preorder* traversal of the document, which is used to (*i*) help identify PC or AD relationships, and (*ii*) impose a logical document order among the nodes. We argue that, it is enough to use the preorder ranks of the nodes to impose the document order. Moreover, each node is associated with a parent pointer in order to locate its parent node. Given a leaf instance node n_i, the parent pointer $parent(n_i)$ is used to construct the complete leaf-to-root path originating from n_i. This complete path structure is constructed in the last stage of the path matching phase as the final round of path comparison.

4 Path Matching

Given an XPR query Q and the range-matched instances p_i of the database, the *path matching phase* performs the necessary steps to identify those path structure instances p_i whose path component p_i^ρ matches the path expression component Q^ρ. In the offline phase, each path expression of the database is hash-mapped and summarized by a compact bit-vector signature by collectively applying a family of hash functions on the element tags of each path based on the notion of Bloom filter [4]. In the following, we will introduce the Bloom filter and the motivations behind incorporating it.

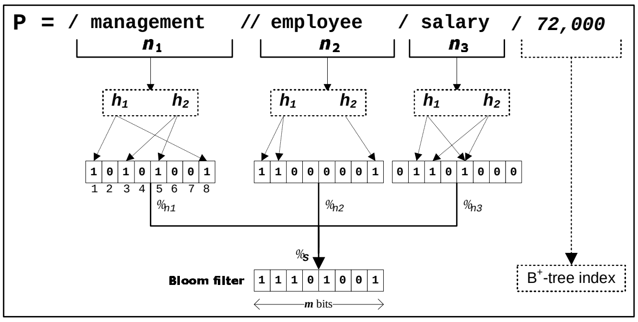

Fig. 2. A Bloom filter example.

Bloom filter is a space-efficient data structure to probabilistically represent a set and its elements to support highly accurate *set membership* queries [4]. The Bloom filter B consists of a bit vector of length m, and a family of k independent hash functions. Given a set $S = \{n_1, n_2, \ldots, n_{|S|}\}$, a family of hash functions are used to construct a bit-vector signature for S. Figure 2 depicts the construction of a Bloom filter bit-vector signature using $k = 2$ independent $m = 8$-bit hash functions h_1 and h_2, on the path set $S = \{$`management, employee, salary`$\}$, where $n = |S| = 3$, from an employee database.

In general, given each element $n_i \in S$, the family of hash functions h_j $(1 \leq j \leq k)$ are used to map n_i into a bit-vector. All the entries of the bit vector are initially set to zero. In order to construct the desired bloom bit-vector B_{n_i}, all the k hash functions h_j are applied to n_i. The application of each h_j on n_i results in "*setting*" some entries of B_{n_i} to 1. For instance in Figure 2, the application of hash function h_1 on n_1, $h_1(n_1 =$ `management`$)$ *sets* the 1^{st} and 8^{th} bits of the corresponding bit-vector B_{n_1}. Similarly, $h_2($`management`$)$ sets the 3^{rd} and 5^{th} bits of B_{n_1}. To construct the bloom bit-vector for the whole set $S = \{n_1, n_2, \ldots, n_{|S|}\}$, the resulting bit-vectors B_{n_i} are combined to form the bloom bit-vector B_S. The combination of the bit vectors B_{n_i} may be performed through a simple logical OR operation. That is, the bit vectors resulting from the application of h_1 and h_2 on the path element `management, employee,` and `salary` of Figure 2, are combined using a logical OR function to construct the bloom bit-vector signature B_{P^ρ} $(=B_S)$ for the path component $P^\rho =$ `/management//employee/salary`. The i^{th} entry of B_{P^ρ} is *set* to 1 if and only if the i^{th} bit vector entry of at least one of the path components B_{n_1}, B_{n_2} or B_{n_3} has been set to 1. For instance, in Figure 2 the 8^{th}-bit of the final Bloom filter B_S is *set* because the 8^{th}-bit of B_{n_1} (or similarly B_{n_2}) is set. Note that, such application of Bloom filter relaxes the edge requirement as imposed by the query. This feature helps to additionally identify and report those instances whose path structure components are very similar to the query, yet having different edge structure.

Subsequently, to test whether the query's path component Q^ρ is similar to an instance path component B_{P^ρ} of the database, the same set of hash functions are applied to B_{Q^ρ} and all the corresponding bit-vector entries are *set* to 1. If all the "*set*" entries of B_{Q^ρ} match with their counterpart in B_{P^ρ} (that is $h(B_{Q^\rho}) \subseteq h(B_{P^\rho})$), it implies that the database path component B_{P^ρ} is identical to B_{Q^ρ} with some probability. The set of all such path structure instances is a superset of the actual (exactly-matched) answer set.

However, there is a chance of B_{P^ρ} and B_{Q^ρ} being identical while the actual path components P^ρ and Q^ρ are different (e.g. by-chance collisions/similarity of the "*set*" entries of Q^ρ and P^ρ). In such a case, a *filter error* (*false positive*) is said to have occurred. The performance of the hash functions of the Bloom filter depends on the *filter error ratio*, which is proven by B. H. Bloom [4] to be as follows. Let n be the number of nodes (or elements) in the set S (or path component P^ρ), m the size of the bit vector and k the total number of hash functions. The filter error ratio is defined as $\left(1 - e^{-\frac{kn}{m}}\right)^k$. For instance, the formula suggests a filter error of only 2.8% for $n = 3$, $m = 8$ and $k = 2$. Moreover, one of the most interesting features of the Bloom filter is that it guarantees not to incur any *false negatives* while being highly accurate and very space-efficient. Note that one of the shortcomings of this approach is the lack of support for updates. However, a variation of Bloom filter, *Counting Bloom Filter* [10], can be employed to resolve such a shortcoming.

Based on the above discussion, we can observe that the Bloom filter representation of each path structure provides an efficient mechanism to compare each path component of the document tree against their counterpart in the query. We next introduce the overall procedure of the FLUX algorithm which combines the features of range and path matching schemes.

5 FLUX Algorithm

Given the document tree T, the offline phase starts by performing a *preorder traversal* on T and assigns preorder ranks (ELEM-ID) to each node of T (the number on the top-left of each node in Figure 1). These preorder ranks create a virtual document order. FLUX consists of five individual phases as described in the following. Due to the space limit, we omit the algorithmic details of the FLUX procedure here.

1) Offline Index Creation. The FLUX offline manager constructs a B$^+$-tree index structure on the *indexable attributes* of the XML document collection (e.g. `age`, `salary`, `year`, and `date`). The leaves of each such B$^+$-tree store the attribute content (e.g., `age` value), ELEM-ID, and the bloom bit-vector signature of the root-to-leaf path structures of the corresponding nodes. For instance, the node corresponding to `age = 64` at the leaf bucket level of B$^+$-tree of part 2 in Figure 1 stores the *preorder rank* ELEM-ID (e.g. `8` in this instance) of the actual node of the document tree whose `age` attribute has the value `64`. Moreover, it stores the bloom bit-vector signature of the root-to-leaf path structure ending at that particular node. For instance, for the node `age` $= 64$ located at the B$^+$-tree leaf bucket of part 2 in Figure 1, the bit-vector `00101110` represents the bloom signature of the root-to-leaf path structure `/management/employee/age` of the node $^{8}64$ of the document tree in part 4 of Figure 1, where the numbers `1`,`2` and `7` denote the ELEM-IDs of the element tag instances of `management`, `employee` and `age` element nodes, respectively.

2) Query Segmentation. This phase segments the query expression Q = `//management//employee/[64≤age≤66]` into the path component Q^{ρ} = `//management//employee/age` and the numerical predicate component Q^{η} = `[64` $\leq$ `age` $\leq$ `66]`.

3) Range Lookup. The search part of this phase corresponds to find the range-matched instances of the query range predicate (Lines 2-5 of the algorithm in the appendix shows such a procedure). For instance, the corresponding B^{+}-tree of the `age` range attribute is searched for potential candidate bucket nodes matching the predicate in Q^{ρ} (e.g. nodes `64, 65` and `66` in the part 2 of Figure 1 for `[64` $\leq$ `age` $\leq$ `66]`).

4) Path Matching and Filtration. Let $B_{p_1}, \ldots, B_{p_k}$ denote the bloom signatures of each of the k matches of the database (e.g., the Bloom filter of the path component `/`^{1}`management/`^{2}`employee/`^{7}`age` which ends at node ^{8}`64`), whose contents have already been matched with the query's range predicate Q^{η}. This stage is responsible for matching the path component of the query $B_{Q^{\rho}}$ against the path components of the range-matched instances $B_{p_1}, \ldots, B_{p_k}$. It ranks each matching instance B_{p_i} based on its similarity to $B_{Q^{\rho}}$. The path matching procedure corresponds to the invocation of the `BloomFiltration()` function at lines 14-15 of the algorithm described in the appendix where its definition is provided at lines 30-36. After filtering out the false positives, the `candidPath` holds the results of candidate matching instances to Q in the database. Finally, the actual path structures of the non-filtered matches are constructed (using the node pointers from leaf-to-root), compared against the query and reported to the user.

6 Experimental Evaluations

We implemented the FLUX system using *Java 1.4.2* and ran our experiments on a *Pentium M-2GHz* processor with *2GB* of main memory, using a page size of 1KB (determine the number of indexed data items which a leaf node can have and the number of key/pointers which an internal node can have for the B^{+}-tree.), cache size of 100KB, and LRU cache replacement policy. We compared our proposed technique with PathStack [6] which is the best in the literature for simple XPath queries. The PathStack technique is also implemented using *Java 1.4.2*. Two variations of PathStack were implemented in terms of the way of retrieving the XML document elements residing in the range specified in the query for the structural join: one variation uses B^{+}-tree index and the other variation does not. This is mainly because that we wanted to make sure that the advantage of using FLUX is not necessarily overshadowed by the indexing solution alone.

The experimental evaluations were performed on a set of both synthetic (XMark [24] containing information about an auction site) and real (dblp[3]) XML datasets. The dblp dataset (sized of 127MB) consists of 3,332,130 element nodes with an average and maximum depth of 2.9 and 6, respectively. We generated a set of synthetic XMark datasets with scaling factor ranging from 0.1 to 1.2 for the experimental evaluation. The average depth for the XMark datasets is 5. The number of hash functions used for constructing the Bloom filter is 4. For each element along the path which leads to an instance of the range attribute, its MD5 digest (a 128-bit cryptographic message) is computed [16]. This 128-bit message is evenly divided into 4 groups. Each 32-bit group is further transformed into an integer ranging from 0 to the Bloom filter size − 1. Unless otherwise stated, the Bloom filter size was chosen to be 14 bits for dblp dataset and 16 bits for XMark datasets which will be explained later in this section.

The results presented in this section were generated by averaging the results from running a workload of 100 random queries on dblp and XMark datasets. The dblp query template was chosen as $\texttt{Q}_D$ `= /dblp/article/[`$\$LB \leq$ `year` $\leq \$UB$`]`, for different random values of $\$LB$ and $\$UB$. Similarly, the XMark query template was selected as $\texttt{Q}_X$ `= /regions//item//mail/[`$\$LB \leq$ `date` $\leq \$UB$`]`. The range values [$\$LB$, $\$UB$] were chosen randomly from the `<year>` and `<date>` domain space in the year range 1945 to 2003 and date range 01/01/1998 to 12/28/2001. The dblp dataset includes 328,831 path instances leading to the `year` element, which is the reason behind using Q_D as the query template for dblp dataset since it provides a large candidate set. The richness of the path structure which leads to the `<date>` element is the reason behind choosing Q_X as the query template for XMark dataset (more structural variations on Q_X can be applied for the structural effect study). Moreover, different amount of random noise was imposed on the dblp and XMark datasets to create path structure variation at the element names. For instance, if $x\%$ noise is imposed on the dblp dataset and assume that there are N root-to-leaf paths leading to `year` element, then $N\times$ $x\%$ of them will be modified by randomly changing one or more element tags to create the path structure variation. Following are some notations used in the upcoming figures:

• **Total Candidates**: Number of all the possible year instances (dblp) and date instances (XMark) in the database for the inspected range resulting from the range query search on the B^+-tree index structure lookup phase.
• **Remaining Tuples**: The number of candidates left for further inspection after pruning the intermediate results by comparing their Bloom filter signature against the Bloom filter signatures of the query.
• **Actual Answers**: The number of actual answers in the database to the query.
• **False Positive Rate (FPR)**: The FPR is calculated as ($RemainingTuples$ − $ActualAnswers$)/$RemainingTuples$, which indicates how close the filtration gets to the actual answer set.

[3] Acquired from the University of Washington's XML Data Repository accessible through *http://www.cs.washington.edu/research/xmldatasets/*

Figures 3-6 analyze the effect of *range length*, *Bloom filter size*, the imposed *noise*, and the *scalability analysis* on the ***Filtration***, ***False Positive Rate (FPR)*** and ***Response Time*** effectiveness of FLUX, on the dblp and XMark datasets, respectively.

6.1 Effect of Range Length

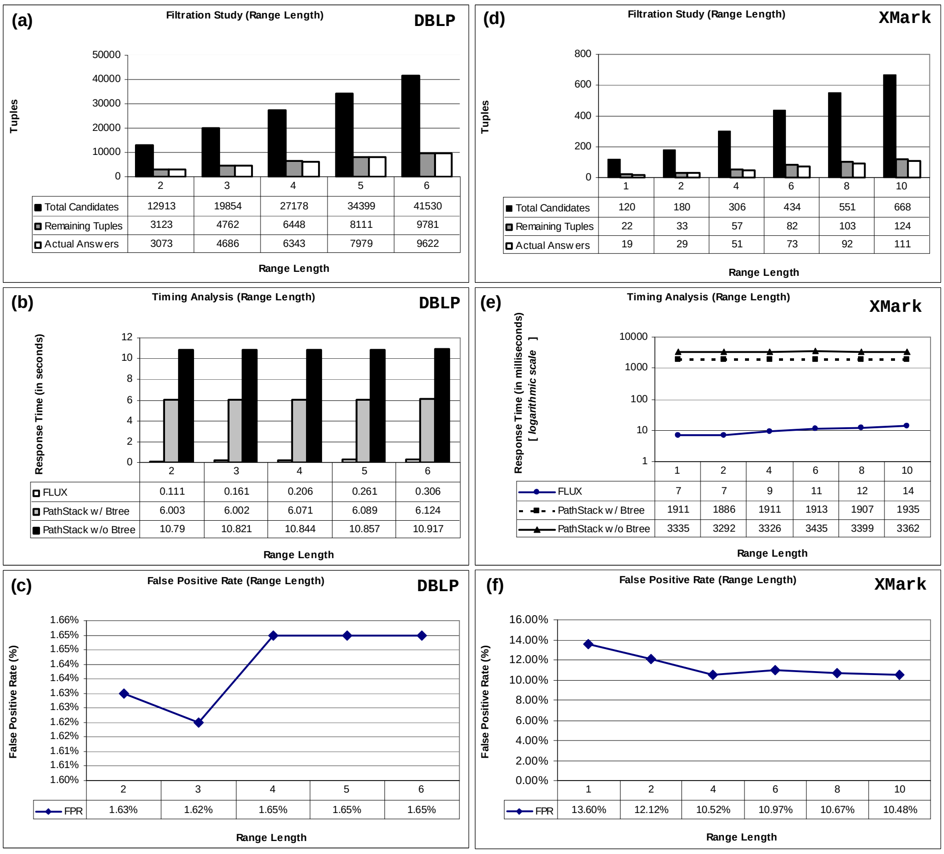

Fig. 3. Effect of range length variation on the filtration, FPR and response time.

Figure 3 depicts the effect of the range length $r = |\$UB - \$LB|$ on the performance of FLUX on dblp and XMark (scaling factor = 1, size ≈ 113MB and noise = 30%) datasets. The query's range length/extent is varied from 2 (narrow) to 6 (moderately wide), and 1 (narrow) to 10 (wide) on the dblp[4] and XMark datasets, respectively. FLUX succeeds in pruning a substantial fraction of the candidate result set in the Bloom filter comparison phase as we can observe from Figure 3(a) and 3(d). For instance, in Figure 3(a), the column pertaining to $r = 3$ indicates that the application of bloom filtration reduces the number

[4] e.g. $1999 \leq$ /year ≤ 2003 has the range extent of $r = |2003 - 1999| = 4$

of total candidates from 19854 tuples to 4762 tuples, or in other words, to 24% of the total candidate result set. Figures 3(b) and 3(e) depict the total response time of performing the designated operations, as a function of range length compared with PathStack [6] (with and without using B^+-tree index structure). The running time of FLUX consistently outperforms PathStack on both dblp and XMark datasets. For instance, in Figure 3(e) FLUX performs 100-times faster on average when compared to PathStack (with B^+-tree index structure). Figures 3(c) and 3(f) depict the stability of False Positive Rate (FPR), which stays within 2% of the remaining tuples as the range length varies for dblp dataset and 14% for the XMark dataset.

6.2 Effect of Bloom Filter Size

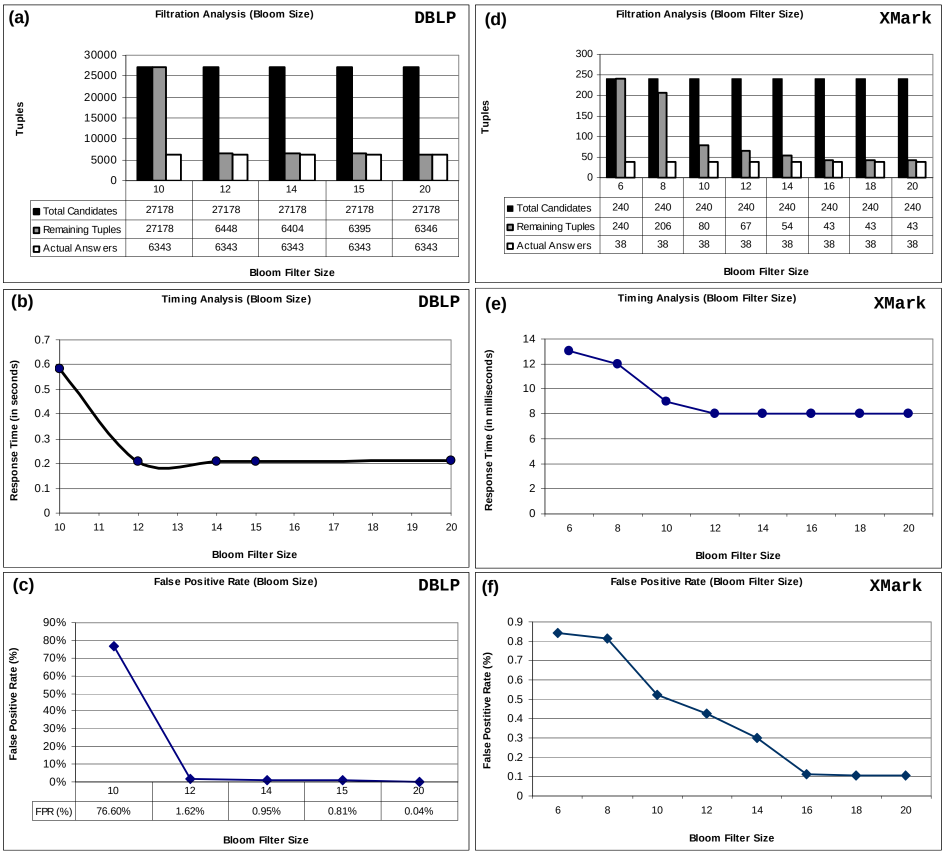

Fig. 4. Effect of the size of Bloom filter signature (in *bits*) on filtration, FPR, and response time.

Figure 4 analyzes the effect of Bloom filter size (in *bits*) as it varies from 10 to 20 bits and 6 to 20 bits on dblp and XMark datasets. The XMark dataset of this section was generated with a scaling factor of 1, with about 113MB in size and 30% imposed noise at the path element names. Figures 4(a) and 4(d) validate the intuitive expectation that the larger choice of the bloom signature length should result in more effective filtration. Figures 4(b) and 4(e) depict the response time analysis of FLUX when varying the bloom bit-vector size in answering the same set of 100 random queries on each respective dataset. The filtration (Figures 4(a) and 4(d)) and response time (Figures 4(b) and 4(e)) performance of FLUX improves consistently as the size of the bloom bit-vector increases from 10 to 14 bits for the dblp dataset and 10 to 16 bits for the XMark dataset. This is due to the fact that, the chance of bloom signature collision[5] reduces as the size of the bloom signatures increases. When the bloom bit-vector increases from 14 to 20 for the dblp dataset and 16 to 20 for the XMark dataset, the filtration effectiveness still increases while the query response time does not due to the fact that larger size of Bloom filter will incur more time to retrieve the corresponding data. Hence, we choose 14 bits for the dblp dataset and 16 bits for the XMark datasets for constructing Bloom filters in a timely manner. Moreover, Figures 4(c) and 4(f) demonstrate the filtration effectiveness of FLUX which is shown in the reduction of FPR when increasing the size of Bloom filter.

6.3 Effect of Noise in Data

For this set of experiments, we introduced random noise at the element names, varying from 1% to 5% on dblp dataset and 1% to 8% on XMark dataset, respectively. Figure 5 depicts the effect of the imposed noise ratio on the overall performance of FLUX. As expected, the introduction of more noise results in larger FPR as shown in Figures 5(c) and 5(f). However, despite the introduction of noise, FLUX performs very efficiently in filtration ratio and response time as observed in Figures 5(a) and 5(d), and Figures 5(b) and 5(e), respectively. FLUX substantially outperforms PathStack regardless of the amount of noise imposed on the data as shown in Figures 5(b) and 5(e). Relative to PathStack, FLUX performs even better when more noise is inherent in the dataset, which is a very desirable feature when the query is posed on datasets with variations in their representation or not necessarily conforming to a unified schema or Document Type Definition (DTD).

6.4 Query Structure Variation

Table 1 depicts the response time analysis when varying the query structure in FLUX and PathStack (with and without B^+-tree index). From type Q_1 to Q_3, more element tags are imposed on top of the range attribute to create more complex path structures. The results were acquired by averaging the running time of 100 random range queries of type Q_i (of Table 1). The range domain was selected in the 01/01/1998 to 12/28/2001 date range and each random range query has length 4. The Bloom filter size was selected to be 16 bits. The incorporated XMark dataset was generated using a scaling factor of 1 with 30%

[5] The probability bloom hash functions assign an identical bloom signature to two different path structures.

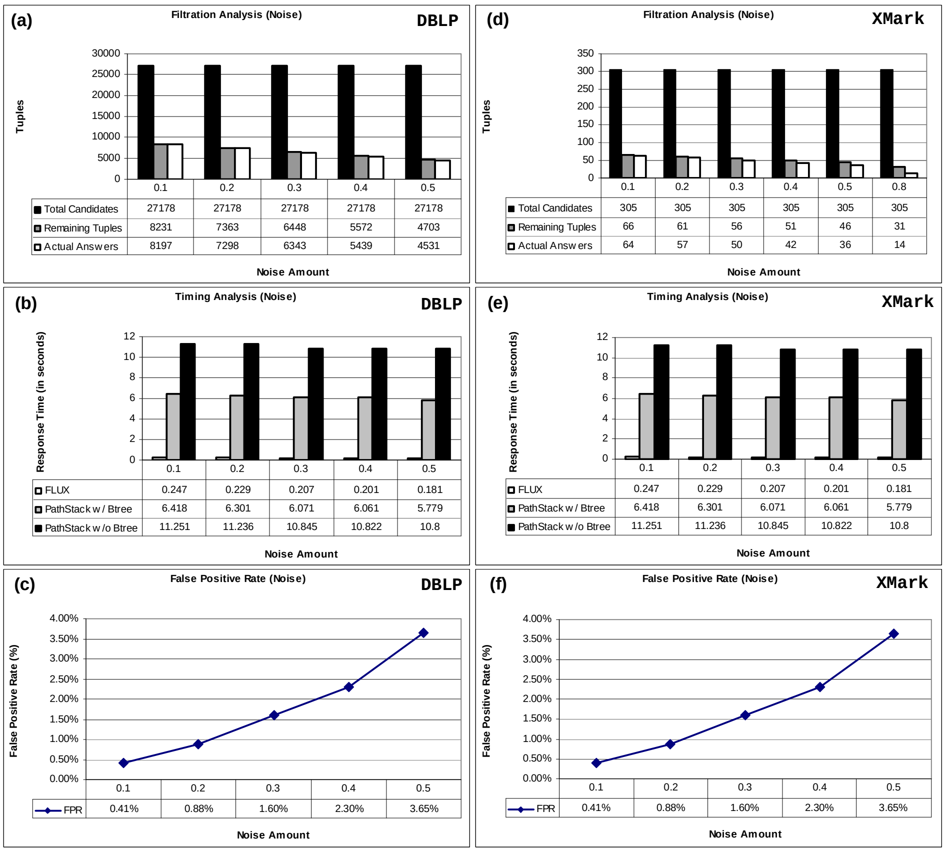

Fig. 5. The results of applying random noise with various intensity on the element names.

noise. In all the observed cases, FLUX consistently outperformed PathStack. The performance of FLUX is slightly affected when the path structure of the query tends to get more complicated due to the bottom-up computation approach. The set of the remaining tuples for each type of query is the same after using the bloom filtration. Thus the cost of retrieving the corresponding paths for the remaining tuples for further inspection against the query is approximately the same. However, for PathStack, more structures with the query will incur more document elements retrieved from the disk for the structural join to produce the matching instances of the query. Hence, the performance of PathStack will decrease when more path structures are imposed on the same range attribute.

6.5 Scalability Analysis

In this set of experiments, we generated a set of XMark datasets with scaling factors ranging from 0.1 to 1.2 to study the effects of document size on the effectiveness of FLUX. Figure 6 depicts the filtration efficiency and response time analysis of FLUX versus PathStack resulted from running a set of the

Query	FLUX	PSB	PS
$Q_1 = regions//mail/date$	7.9	1521	2937
$Q_2 = regions//item//mail/date$	8	1901	3323
$Q_3 = regions//item/mailbox/mail/date$	8.1	2307	3708

Table 1. Response time (in *milliseconds*) comparison of FLUX v.s. PathStack on XMark dataset varying the query structure. PSB = "*PathStack with Btree*" and PS = "*PathStack without Btree*".

same 100 random range queries selected in the 01/01/1998 to 12/28/2001 date range. The performance of both FLUX and PathStack suffers as the size of the dataset increases, however, FLUX experiences from 98 times to 215 times less performance degradation rate compared with PathStack with B$^+$-tree index structure. The comparison with PathStack without B$^+$-tree index structure, is even more dramatic.

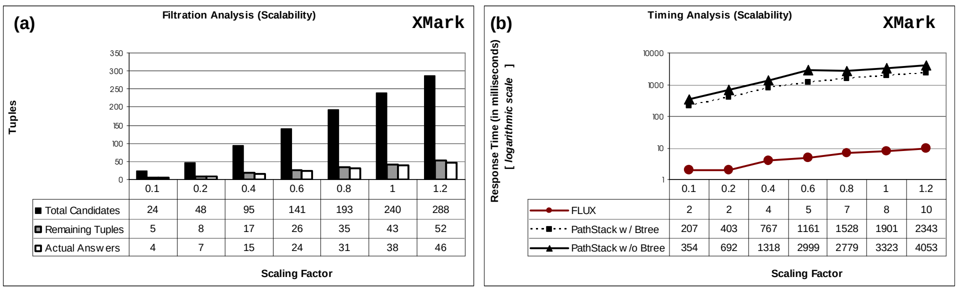

Fig. 6. The scalability analysis on XMark datasets

7 Conclusion

This paper proposed an efficient technique, named FLUX, for answering complex range queries in a database of XML documents. FLUX incorporated a B$^+$-tree based index structure on the contents of range attributes. It uses the notion of Bloom filters to associate a structure signature to each range attribute instance. The filtration performed by the bloom signatures of FLUX reduced the search space to a minor fraction of the intermediate result set. Experimental results demonstrate that the filtration, response time, false positive rate, speedup and scalability of FLUX consistently outperforms PathStack [6] on both real and synthetic datasets. The FLUX procedure proceeds with range matching followed by path matching. Nevertheless, depending on the selectivities of both the range and the path structure, it might be preferable to apply the structure matching first and then the range matching, or vice versa. Part of our future research work will include adapting FLUX or designing new index structures to handle the cases with selective structures.

References

1. S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas and D. Srivastava, Structural Joins: A Primitive for Efficient XML Query Pattern Matching. ICDE, 141–152 (2002).
2. S. Al-Khalifa et al., Querying Structured Text in an XML Database. SIGMOD, 4–15 (2003).
3. S. Amer-Yahia, L.V.S. Lakshmanan and S. Pandit, FleXPath: Flexible Structure and Full-Text Querying for XML. SIGMOD, 83–94 (2004).
4. B.H. Bloom, Space/Time Trade-offs in Hash Coding with Allowable Errors. Communications of the ACM **13(7)**, 422–426 (1970).
5. C. Botev, J. Shanmugasundaram and S. Amer-Yahia, A TeXQuery-Based XML Full-Text Search Engine. SIGMOD, 943–944 (2004).
6. N. Bruno, N. Koudas and D. Srivastava, Holistic twig joins: optimal XML pattern matching. SIGMOD, 310–321 (2002).
7. D. Chamberlin, Daniela Florescu, Jonathan Robie, Jérôme Siméon and Mugur Stefanescu, XQuery: A Query Language for XML. W3C Working Draft, http://www.w3.org/TR/xquery (2001).
8. S. Chien, Z. Vagena, D. Zhang, V.J. Tsotras and C. Zaniolo, Efficient Structural Joins on Indexed XML Documents. VLDB, 263–274 (2002).
9. CiteSeer Scientific Literature Digital Library and Search Engine, *http://citeseer.ist.psu.edu*
10. L. Fan, P. Cao, J.M. Almeida and A.Z. Broder, Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. SIGCOMM, 254–265 (1998).
11. T. Grust, Accelerating XPath location steps. SIGMOD, 109–120 (2002).
12. L. Guo, J. Shanmugasundaram, K.S. Beyer and E.J. Shekita, Efficient Inverted Lists and Query Algorithms for Structured Value Ranking in Update-Intensive Relational Databases. ICDE, (2005).
13. H. Jiang, W. Wang, H. Lu and J. Xu Yu, Holistic Twig Joins on Indexed XML Documents. VLDB, 273–284 (2003).
14. R. Kaushik, P. Shenoy, P. Bohannon and E. Gudes, Exploiting Local Similarity for Indexing Paths in Graph-Structured Data. ICDE, 129–140 (2002).
15. DBLP Bibliography Server, *http://dblp.uni-trier.de/*
16. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1997 (.)
17. Q. Li and B. Moon, Indexing and Querying XML Data for Regular Path Expressions. VLDB, 361–370 (2001).
18. J. Lu, T. Chen and T.W. Ling, Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach. CIKM, 533–542 (2004).
19. J. Lu, T.W. Ling, C.Y. Chan and T. Chen, From Region Encoding to Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. VLDB, 193–204 (2005).
20. A. Marian, S. Amer-Yahia, N. Koudas and D. Srivastava, Adaptive Processing of Top-K Queries in XML. ICDE, (2005).
21. P. Rao, B. Moon, PRIX: Indexing And Querying XML Using Prüfer Sequences. ICDE, 288–300 (2004).
22. F. Weigel, H. Meuss, K.U. Schulz and F. Bry, Content and Structure in Indexing and Ranking XML. WebDB, 67–72 (2004).
23. H. Wang, S. Park, W. Fan and P.S. Yu, ViST: A Dynamic Index Method for Querying XML Data by Tree Structures. SIGMOD, 110–121 (2003).
24. A. R. Schmidt et al., The XML Benchmark Project. Technical Report INS-R0103, CWI (2001).