Partitioning XML Documents for Iterative Queries

Nicole Bidoit  Dario Colazzo  Noor Malla  
BD&OAK Team - Université Paris Sud - INRIA 
{bidoit, colazzo, malla}@lri.fr

Carlo Sartiani  
DMI - Università della Basilicata 
sartiani@gmail.com

ABSTRACT

This paper presents an XML partitioning technique that allows main-memory query engines to process a class of XQuery queries, that we dub iterative queries, on arbitrarily large input documents. We provide a static analysis technique to recognize these queries. The static analysis is based on paths extracted from queries and does not need additional schema information. We then provide an algorithm using path information for partitioning the input documents of iterative queries. This algorithm admits a streaming implementation, whose effectiveness is experimentally validated.

Categories and Subject Descriptors
H.2.3 [Database Management]: Languages—query languages; 
H.2.4 [Database Management]: Systems—query processing

General Terms
Languages, Algorithms, Performance

Keywords
XML, XQuery

1. INTRODUCTION

In parallel with the high diffusion of XML as a data format for exchanging and processing semistructured data, recent years have seen the proliferation of a family of lightweight XML query engines, specially designed to process XML data stored in the file system or generated by a stream. These systems all support the W3C XML query language XQuery [7], and are often named main-memory engines, as they load all the data in main memory before querying. By quoting Cong and al. [9], these systems are the best choice in

... domains like Life sciences (e.g., Biology), Astronomy, and even for the management of typical XML documents corresponding to Microsoft Office files (since powerpoint presentations, Word files, and Excel spreadsheets are all currently stored as XML). In all these domains, the management of XML documents is file-system centric and no traditional XML data management systems is yet in place (since non-expert users often find these latter systems to be hard to use and maintain).

Especially in domains like Life science and Astronomy, XML documents are likely to be huge (several GBs), which can jeopardize the possibility of using a main-memory engine for query processing. In other words, main-memory systems, while very flexible and easy to set-up and use, cannot scale up with document size. A partial solution to this problem is offered by projection-based techniques [12, 4, 11] that allow one to prune out, at loading time, parts of the data that are not necessary for the query or the workload being processed. Projection-based approaches provide only a partial solution to the scalability issues of main-memory systems, as the projected input documents may still exceed the main-memory capacity. This may be the case when (i) the input file is huge, (ii) the query selectivity is low and it needs a large part of the input, or (iii) a workload (i.e., a set of queries) has to be evaluated on the document. In the last case, a single global projection meeting the query needs of the whole workload is likely to exceed the main-memory size, while running a query at a time, and projecting (and loading) data for each run would result in a quite inefficient and still failure-prone process.

To overcome the scalability issues of current main-memory query engines, we propose here an alternative technique based on data partitioning. This technique relies on the observation that, in many practical cases, XQuery queries first select a sequence of subtrees by means of a subquery (e.g, an XPath expression), and then iterate operations on this sequence of subtrees. For instance, 13 out of 20 queries of the XMark benchmark meet this property. When this property is satisfied by a query Q, the input document can be split into a collection of parts \{t_1, . . . , t_i\}, so that the evaluation Q(t_i) of the query Q over the document t turns out to be equal to the concatenation of the evaluations Q(t_i) of the query Q over the document parts t_i.

Our Contribution. The first contribution of this paper is a formal characterization of the class of queries that enjoy the splitting property: we dub these queries as iterative queries. By relying on this formal characterization, we develop a static analysis technique that first extracts paths and information about bound variables from the query, and then analyses them in order to statically detect how the document is navigated by the query.

The second contribution of this paper is a partitioning algorithm that exploits the paths extracted during the static analysis to identify the correct partitioning for the input document. Document partitioning is performed in a streaming fashion with a very limited
memory footprint, and can be easily combined with existing projection techniques, so to further improve the benefits of the proposed approach.

An extensive experimental evaluation corroborates that, when using the partitioning approach, main-memory engines can manage very large input documents, at the price of a modest overhead write projection techniques; our experiments also show that partitioning allows for a scalable management of workloads, as the input document is partitioned once for all.

**Paper Outline.** The paper is structured as follows. In Sections 2 and 3 we introduce a few preliminary notions; in Section 4, then, we formally describe iterative queries. In Section 5, next, we present our partitioning algorithm and show how it can be combined with existing projection techniques. In Section 6 we experimentally evaluate our approach. In Sections 7 and 8, finally, we discuss some related works and conclude.

### 2. PRELIMINARIES

Following [3], we represent an instance (document) of the XML data model as a store $\sigma$, which associates to each node location (or identifier) $l$ either an element node or a text node. When $l$ is an element node, $\sigma(l)=a[L]$ (also written $l \leftarrow a[L] \in \sigma$, where $a$ is the element tag and $L=(l_1, l_2, \ldots, l_n)$ is the ordered sequence of the child locations for $l$. Next, $()`$ denotes the empty sequence. When $l$ is a text node, $\sigma(l)=\text{text}[s]$ (also written $l \leftarrow \text{text}[s] \in \sigma$) gives the textual content (the string $s$) of $l$. A tree is a pair $(\sigma, L)$, where $L$ is the root location of the tree. We denote by $\text{dom}(\sigma)$ the set of locations of a store (analogously $\text{dom}(t)$ for a tree). Given a location $t \in \text{dom}(\sigma)$, $\sigma[t]$ denotes the subtree of $\sigma$ rooted at $t$. Sometimes, for simplicity, when $t=(\sigma, L)$, we abusively use $t$ instead of $\sigma$ and, for instance, we write $l \leftarrow \sigma[L] \in t$ instead of $l \leftarrow a[L] \in \sigma$.

Below, we use $[L]$ to denote the set of locations in the sequence $L$. Also, we say that $L'$ is a projection of $L$, denoted by $L' \preceq L$, if $L'$ is obtained from $L$, by erasing some of its locations. For instance, $l_1, l_2 \preceq l_1, l_2, l_3$, while $l_1, l_2 \not\preceq l_1, l_2, l_3$ (ordering is not preserved).

In order to define XML partitions, we need the following notion of XML projection.

**Definition 2.1 (XML Projection)** A tree $t=(\sigma', L')$ is a projection of a tree $t=(\sigma, L)$, denoted by $t' \preceq t$, if $t_1=\sigma$, and for each location $l \in \text{dom}(\sigma')$:

$$l \leftarrow a[L] \in \sigma' \implies (l \leftarrow a[L] \in \sigma \land L' \preceq L)$$

Note that projection preserves tree roots. Figure 1 shows a simple XML tree, its associated store, and a possible projection.

**Definition 2.2 (XML Partition)** A collection of trees $\{t_1, \ldots, t_n\}$ is a partition of a tree $t$ if, for each $i=1, \ldots, n$, $t_i \preceq t$, and for each location $l \in \text{dom}(t)$, we have:

1. $l \leftarrow \text{text}[s] \in t_i \implies \exists j \in [1, \ldots, n], l \leftarrow \text{text}[s] \in t_j$.
2. $l \leftarrow a[L] \in t \implies L = \bigcup_{l \leftarrow a[L] \in t_i} \{L_i\}.$

A tree $t_i$ of the partition is called a part. The two above properties say that each text node has to belong to at least one part, and that element nodes are partitioned in such a way that no child is left out.

Figure 2 contains two possible partitions of the document in Figure 1. As a document can be partitioned in multiple ways, it is crucial to carefully design the partitioning strategy, so that the query result equals the concatenation of query results on each part of the partition. We will see next how to choose the right partition in terms of a path analysis on the query.

The fragment of XQuery considered in this paper is described by the following grammar. This fragment comprises $/$, $\text{let}$, and return clauses as well as $\text{if-then-else}$ statements, and allows the user to specify self, child, and descendant-or-self XPath axes [5] (for simplicity we will write $\text{dos}$ instead of descendant-or-self). The grammar uses a for tag symbols.

$$Q ::= () | Q_1, Q_2 | a\{Q\} | \text{Exp}$$

$$\text{Exp} ::= x | x/\text{Step} | \text{Step}$$

$$\text{Step} ::= \text{Axis} :: NT$$

$$\text{Axis} ::= \text{self} | \text{child} | \text{dos} | \text{@attr}$$

$$\text{NT} ::= a | \text{node}() | \text{text}()$$

In the following, we say that a query is well-formed if and only if i) it does not contain free variables (i.e., variables with no corresponding for/let bindings), ii) no variable name is used twice in for/let bindings, and iii) it starts navigating the document by means of non-self step. Condition i) ensures that well-formed queries start navigating documents from their root. For instance, the query for $y$ in $x$/Step return $Q$ is not well-formed, while the query for $y$ in $/$Step return $Q$ is. The restriction ii) simplifies the analysis, and can be always obtained by $\alpha$-renaming. Condition iii) excludes queries like for $y$ in $/$self :: NT return $Q$, which is still assumed to simplify formalizations, and is non restrictive as in most practical cases queries start the navigation by means of a child or dos axis.

In this work, we focus on queries issued on a single document. Indeed, multiple document queries are likely to be not iterative, and their treatment goes far beyond the scope of this paper. Also, we focus on for/let expressions using element construction only on the right-hand side expression $Q_2$, as happens in most practical cases (e.g., all XMark queries are of this form, provided that in some queries let bindings are inferred).

The evaluation of a query $Q$ on an input tree $t=(\sigma, L)$, denoted by $Q(t)$, yields a pair $(\sigma_Q, L_Q)$, where the store $\sigma_Q$ is a forest extending the initial store $\sigma$ with the new elements built by $Q$, while $L_Q$ is the sequence of node locations that are returned by the query and whose contents are described in $\sigma_Q$. Due to a lack of space, we do not report here formal semantics of this XQuery fragment (a concise and elegant formalization can be found in [3]).

In order to define equivalence among query results, we also need the following notions. Equivalence among two trees, denoted by $t \equiv t'$, holds iff they are isomorphic (they possibly differ only in terms of name of locations). When $\sigma$ and $\sigma'$ are forests and $L=(l_1, \ldots, l_n)$ and $L'=(l'_1, \ldots, l'_n)$ are sequences of locations, we write $\sigma \equiv \sigma'$ to state that, for $i=1,\ldots, n$, we have $\sigma[l_i] \equiv \sigma'[l'_i]$. Finally, when $\sigma$ and $\sigma'$ have disjoint domains (no common location), we define the concatenation $(\sigma, L) \cdot (\sigma', L')$ as the pair $(\sigma \cup \sigma', (L, L'))$, where $L', L'$ denotes the concatenation of $L$ and $L'$.

### 3. PATH EXTRACTION

In our approach, paths are used for characterizing iterative queries, and for partitioning an input document. Paths are extracted from a

\footnote{For instance, infilling is needed for $Q_{10}$.}
query by using the path extraction function of Figure 3; this function resembles that proposed in [12, 4]. However, paths extracted according to \( E() \) carry a richer information, as they also describe the relation with \( \Gamma \)-variables. Extracted paths obey the following grammar:

\[
P := \epsilon \mid /S \mid P/S \quad S := \text{Step} \mid \text{Step}\{\text{for } x\}
\]

where \( \epsilon \) denotes the empty path.

For instance, a path \( P'\{\text{for } x\}/P'' \) is extracted from \( Q \) when

(i) a subquery of \( Q \) has the shape for \( x \) in \( Q \), return \( Q_s \), (ii) \( P' \) is extracted from \( Q_s \) and selects possible bindings for \( x \) while (iii) \( P'' \) is extracted from \( Q_d \) in the context of the previous bindings (in other words, \( x/P'' \) is extracted from \( Q_d \)).

**Example 3.1** Consider the following query \( Q \):

\[
Q = \text{for } x \text{ in } /\text{child} : a \text{ return } \\
\text{for } y \text{ in } x/\text{child} : b \\
\text{return } (y/\text{child} : d, y/\text{child} : e)
\]

The set of extracted paths of this query is \( \{P_1, P_2, P_3, P_4\} \), with

\[
P_1 = /\text{child} : a\{\text{for } x\}
\]

\[
P_2 = /\text{child} : a\{\text{for } x\}/\text{child} : b\{\text{for } y\}
\]

\[
P_3 = /\text{child} : a\{\text{for } x\}/\text{child} : b\{\text{for } y\}/\text{child} : d/\text{dos} : \text{node}()
\]

\[
P_4 = /\text{child} : a\{\text{for } x\}/\text{child} : b\{\text{for } y\}/\text{child} : e/\text{dos} : \text{node}()
\]

Variable information in paths is important to characterize iterative queries and to identify partitioning paths (see Section 4), while it will be ignored for the purpose of partitioning.

Our path extraction function \( E() \) is defined in Figure 3 by structural induction. For queries of the form for \( x \) in \( Q \), return \( Q_s \) (rules 11), the function first extracts paths from \( Q_s \); these paths are then enriched with information about variable bindings and added to an environment \( \Gamma \) which is used for the recursive extraction of paths from \( Q_s \). In particular, \( \Gamma \) is used to associate the right path to each free occurrence of the variable \( x \) in \( Q_s \) (rules 4 and 5). Rules for let expressions are similar, except that they do not keep track of information about let-variables.

In these rules, we use a boolean flag \( m \) to distinguish between subqueries that generate fragments of the result for the outer query \((m=1)\) and subqueries that are only used for binding variables or filtering results \((m=0)\). When \( m=1 \), the terminal rules 5, 7, and 8 extend extracted paths with a dos : node() step, so to capture and include in the projection all the nodes required by the query to build its result. Hereafter, for simplicity, we will often abbreviate \( E(Q, \emptyset, 1) \) with \( E(Q) \).

Our extraction function ignores attributes (rule 13); indeed, when an element is copied into a part, it carries its attributes, regardless of their use in the input query.

In the following, \( E(V(P)) \) denotes the path obtained from \( P \) by removing \{for \} occurrences.

### 4. Iterative Queries and Partitioning Paths

Our proposal is based on the idea of partitioning an input document \( t \) into a collection of documents \( \{t_1, \ldots, t_n\} \) and possibly projecting each \( t_i \) according to \( Q \), so that:

\[
Q(t) \equiv (Q(t_1), \ldots, Q(t_n))
\]

where \( t_i \) is the projection of \( t_i \). The input document is partitioned according to a partitioning path \( P \), which is opportunely chosen among the paths extracted from \( Q \).

In order to guarantee the correctness of query evaluation, this approach can be applied only when \( Q \) first selects a sequence of nodes \( S \), and then iterates over the nodes in \( S \) by exploring their corresponding subtrees. Queries satisfying this requirement are called iterative and are quite common in practice. The query of Example 3.1 is iterative. As a concrete example, 13 out of the 20 XMark queries are iterative: namely, queries from \( Q_1 \) to \( Q_{14} \), and \( Q_{14} \) to \( Q_{20} \) are in this class.

For an iterative \( Q \) over a document \( t \), there may be more than one path that could be used for partitioning \( t \). We first characterize this set of candidate partitioning paths and then show how to pick the best one. In the definition below, we say that \( P \in E(Q) \) is maximal if no other path in \( E(Q) \) contains \( P \) as a prefix.

**Definition 4.1 (Candidate Partitioning Paths)** The set of candidate partitioning paths for a well-formed query \( Q \) is denoted by \( \text{Candidates}(Q) \), and is defined as the set of paths \( E(V(P)) \) with \( P \in E(Q) \) such that:

(i) \( P \) is of the form \( P \{\text{for } x\} \);
Figure 3: Path extraction function.

(ii) P does not use text node test;
(iii) for each maximal path \( P' \in E(Q) \), \( P'=P/P'' \).

Condition (i) states that each candidate path is used for iterating inside the query \( Q \). Condition (ii) rules out candidate paths that would iterate on text nodes (like in the query for \( x \) in \( /\text{doctype} :: \text{text}() \) return \( Q' \)) because we want to ensure that partitioning is performed on a sequence of element nodes rather than a sequence of text nodes.\(^5\) Condition (iii) is the most important one: the restriction on maximal paths is needed since, otherwise, only the minimal common prefix of \( E(Q) \) paths would be a candidate.

For instance, for the query and extracted paths in Example 3.1, \( E(P_1) \) and \( E(P_2) \) are candidate paths, while \( E(P_3) \) is not, as the prefix context does not hold wrt the path \( P_3 \). Note that, if we alter the query by considering the new return clause return \( (x/\text{child} :: d, y/\text{child} :: e) \), then only the candidate path is \( E(P_1) \).

**Definition 4.2 (Iterative Queries)** A well-formed query \( Q \) is iterative iff \( \text{Candidates}(Q) \neq \emptyset \).

If \( Q \) is iterative, then the sequence of nodes selected by a candidate path in a document \( t \) can be partitioned in order to split the query evaluation.

**Definition 4.3 (Partitioning Path)** Let the query \( Q \) be iterative. The path \( P \) is the partitioning path for \( Q \) if and only if \( P \) is the candidate partitioning path of \( Q \) having maximum length.

In the following, a partitioning path will be denoted by \( PP \). Going back to the query of Example 3.1, we have \( PP = /\text{child} :: a/\text{child} :: b \).

Picking up the longest candidate as partitioning path minimizes the size of trees belonging to the sequence selected by the path, hence maximizing the likelihood that each part yielded by partitioning fits in the available main-memory.

5. **PARTITIONING**

This section presents our partitioning algorithm. First, we give an overview of the approach and a few preliminary definitions. Then, we present an abstract and DOM-based specification of the algorithm, which assumes the store-based representation of XML trees. Next, we describe how the algorithm gracefully adapts to the case of a workload. Finally, we show how the DOM-like algorithm can be implemented in a streaming fashion.

\(^5\) Although this restriction can be relaxed, we give priority to presenting the core of the partitioning method here.

5.1 Overview and Preliminary Definitions

Before illustrating the partitioning process, we need a few preliminary definitions and notions. Hereafter a match for a path is called a terminal match, while an ancestor of a match is called a non-terminal match.

For instance, for the input tree in Figure 5, and the path \( P = /\text{doctype} :: \text{c} \), terminal matches are nodes \( l_1, l_2, l_3 \) and \( l_4 \), while non-terminal matches are ancestors of these nodes, i.e., \( l_1, l_2, l_3, l_4 \) and \( l_5 \).

Given a tree \( t = (e, 1) \) and a path \( P \), we find terminal and non-terminal matches of \( P \) by means of an iterative procedure that visits the tree \( t \) in a top-down manner, and matches each node to a set of paths obtained from \( P \) by means of two rewriting operations.

A first rewriting aims at aligning paths each time a deeper level is visited. For instance, in the previous example the root node is compared to the set of paths \( \{ /\text{self} :: \text{c}, /\text{self} :: \text{node}() /\text{doctype} :: \text{c} \} \), obtained by the alignment of \( P = /\text{doctype} :: \text{c} \). The path \( /\text{self} :: \text{node}() /\text{doctype} :: \text{c} \) is an unfolding of the original one, and is needed to match \( c \) nodes at deeper levels in subsequent steps of the process.

Alignment is performed by the function \( \text{Down}(\tau) \), which aligns all paths in a set \( \tau \), according to the following definition.

**Definition 5.1 (Path Alignment)** The alignment \( \text{Down}(\tau) \) of a path set \( \tau \) is defined as \( \bigcup_{P \in \tau} \text{Down}(P) \) where:

\[
\text{Down}(/\text{child} :: \text{NT} / P) = \{ /\text{self} :: \text{NT} / P \}
\]

\[
\text{Down}(/\text{doctype} :: \text{NT} / P) = \{ /\text{self} :: \text{NT} / P \}
\]

\[
\text{Down}(/\text{doctype} :: \text{node}() /\text{doctype} :: \text{NT} / P) = \{ /\text{self} :: \text{node}() /\text{doctype} :: \text{NT} / P \}
\]

\[
\text{Down}(e) = \{ e \}
\]

Paths obtained by alignment all start with a self step, which a terminal or non-terminal node has to necessarily match. After alignment, resulting paths may contain consecutive steps using the self axis (in particular, if the path already contained a self step before alignment).\(^4\) If consecutive self steps like in \( /\text{self} :: b /\text{self} :: c \) occur in an aligned path, then the path is discarded from the process as it has empty semantics. Non-contradictory consecutive self steps (like \( /\text{self} :: b /\text{self} :: \text{node}() \)) are collapsed in a single self step (like \( /\text{self} :: b \)) by means of a simple rewriting. These simple rewritings are routinely made after each alignment operation.

\(^4\) We assume that a path extracted from a query does not contain the self axis in the first step (according to W3C semantics, this path has empty semantics).
obtained paths are then considered for matching with the current node, as discussed shortly.

We discuss now the second rewriting operation. In the search of matches for a path \( P \) in a tree \( t \), given a node whose tag or text value is \( c \in \{a, \text{text}[s]\} \), and the corresponding set of aligned paths is \( \tau \) (obtained from \( P \)), the residuation function \( \text{Res}(\alpha; \tau) \) returns a path set \( \tau' \) and a value \( \text{Match}(\text{ok}_t, \text{ok}_n, \text{fail}) \). The path set \( \tau' \) will be recursively matched against children of the node after an alignment operation, while \( \text{Match} \) specifies whether the current node is a terminal match, a possible non-terminal match, or neither a terminal nor a non-terminal node. A possible non-terminal match is finally confirmed as such when one of its descendants happens to be deemed as a terminal match in subsequent steps.

Deriving the value \( \text{Match} \) produced by residuation relies on the following binary (commutative and associative) function \( \uplus \), shown in Table 1, where the symbol \( \uplus \) indicates any value.

<table>
<thead>
<tr>
<th>( \text{Match}_1 )</th>
<th>( \text{Match}_2 )</th>
<th>( \text{Match}_1 \uplus \text{Match}_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{ok}_t )</td>
<td>( \text{ok}_t )</td>
<td>( \text{ok}_t )</td>
</tr>
<tr>
<td>( \text{ok}_n )</td>
<td>( \text{ok}_n )</td>
<td>( \text{ok}_n )</td>
</tr>
<tr>
<td>( \text{fail} )</td>
<td>( \text{fail} )</td>
<td>( \text{fail} )</td>
</tr>
</tbody>
</table>

Table 1: \( \uplus \) function.

**Definition 5.2 (Path Residuation)** The residual of a path \( P \) is defined by distinguishing the following cases:

\[
\text{Res}(a; /\text{self} : NT) = \begin{cases} \epsilon ; \text{ok}_t & \text{if NT} \in \{a, \text{node}(\alpha)\} \\ \text{fail} & \text{otherwise} \end{cases}
\]

\[
\text{Res}(a; /\text{self} : NT/P) = \begin{cases} \epsilon ; \text{ok}_n & \text{if P} \in \{a, \text{node}(\alpha)\} \\ \text{fail} & \text{otherwise} \end{cases}
\]

\[
\text{Res}(\text{text}[s]; /\text{self} : NT) = \begin{cases} \epsilon ; \text{ok}_t & \text{if NT} \in \{\text{text}(\alpha), \text{node}(\alpha)\} \\ \text{fail} & \text{otherwise} \end{cases}
\]

\[
\text{Res}(\alpha; P) = \begin{cases} \epsilon ; \text{fail} & \text{otherwise} \end{cases}
\]

The residual of a path set \( \tau = \{P_1, P_2, \cdots, P_n\} \) is then defined as follows:

\[
\text{Res}(\alpha; \tau) = \bigcup_{i=1}^{n} \text{Res}(\alpha; P_i) \uplus \text{Match}_i
\]

As illustrated shortly, residuation is always applied after an alignment operation, and produces paths that are immediately aligned when descending to a new level of the tree. That said, going back to our observation concerning the handling of consecutive self steps, note that, since each path extracted from a query never starts with a self step, and since multiple and consecutive self steps are eventually collapsed (otherwise the path is discarded) after alignment, residuation always takes as input paths starting with a self step, followed by a non-self step, and eventually produces new paths by simply discarding the initial (matched) self step. This explains why the definition of alignment (Definition 5.1) does not include a case for a first self-step.

To illustrate how the just presented rewriting functions are used to select terminal and non-terminal matches of a path, consider again the input tree in Figure 5, and the path \( P = /\text{dos} : c \). Terminal and non-terminal matches of this path are determined as shown in Figure 4, where for each node we indicate the corresponding aligned and residuated paths. We focus on the first 8 nodes, according to document order.

According to the residuation above indicated, \( l_1 \) and \( l_2 \) are deemed as non-terminal matches since both nodes have a descendant node (\( l_3 \)) being a terminal match. Observe that a terminal match is selected when a single-step path in the current set of aligned paths is matched by the current node: this means that the last step of the initial path is successfully matched. Concerning nodes \( l_4 \) and \( l_7 \), they have no descendant that residuation deems as a terminal match, hence these nodes are not deemed as non-terminal matches.

Residuation and path alignment are at the core of our partitioning strategy. To illustrate consider the following example.

**Example 5.3** Consider the query \( Q \) below and the XML document of Figure 5.

\[
Q = \text{for } x \text{ in } /\text{dos} : c \text{ return } (x/\text{child} : d, x/\text{child} : e)
\]

According to previous definitions, this query is iterative with partitioning path \( PP = /\text{dos} : c \). Also, the set of extracted paths is (for-variables are erased):

\[
\tau = \langle/\text{dos} : c, /\text{dos} : c/\text{child} : d/\text{dos} : \text{node}(\alpha), /\text{dos} : c/\text{child} : e/\text{dos} : \text{node}(\alpha)\rangle
\]

Partitioning of the input tree in Figure 5 is made according to the partitioning path \( PP = /\text{dos} : c \). The tree is traversed top-down according to document order and the first part is determined as follows. During the visit of the tree, non-terminal and terminal matches of the partitioning path are added to the part. Whenever a terminal match of \( PP \) is met, its subtree is projected according to a given projection algorithm (if any), in order to limit as much as possible the number of created parts. In the full version [6] we provide a projection algorithm which uses extracted paths, along the lines of [12], and which is able to project an arbitrary subtree of the input tree. It is worth stressing that our partitioning algorithm can use any other available projection algorithm [4, 11] for subtree projection. Also, note that partitioning could also work with no projection; in this case an higher number of parts would be created, but each one having a size next to a given threshold (see below) ensuring that available main memory is able to allocate each single part.

Just after a projected sub-tree of a \( PP \) terminal match has been added to the part, a check is made in order to verify whether the current size of the part has exceeded a given threshold \( pSize \). In the current example we consider \( pSize = 12 \), which is exceeded when the subtree rooted at the second \( PP \) terminal match is added to the part.\footnote{Note that each time an element is added to a part, the current size is incremented by twice the length of the element tag (both starting and ending tags have to be taken into account), while each time a text node is added to the part the current size is incremented by the length of the text content of the node.} This causes the creation of a second part. With \( pSize = 12 \) we finally have the two parts indicated in Figure 5. Note that nodes that are neither non-terminal nor terminal matches of the partitioning path are pruned out during partitioning. These nodes can be safely pruned out because they are useless to the evaluation of the query \( Q \). This is because \( PP \) is a prefix of each path in \( \tau \) (extracted from the query, Definition 4.1), and that a node is needed by \( Q \) if it is (an ancestor of) a match of a path in \( \tau \) (for the same reasons, in Figure 5, note that since subtrees rooted at terminal matches of \( PP \) are projected according to \( \tau \); for instance, the node \( l_{14} \) is not in the second part).

Besides path alignment and residuation, the threshold value \( pSize \) plays a key role in the whole partitioning process. The choice of \( pSize \) depends on many factors, such as the input document, the
query being processed, the specific query processor being used, the
hardware configuration and the available main memory, the
programming language used for implementing the query processor, the
memory management technique adopted, and the operating system
running on the hardware. pSize, therefore, can be determined only
through a trial-and-error process depending on the overall config-
uration, and cannot be formally predicted.

Note that if pSize is too large, it can happen that one or more
dparts are too large to be loaded in main memory, hence under-
ing the whole approach. Surprisingly enough, as we will see in Sec-
tion 6, our experimental evaluation showed that the actual value of
pSize does not influence either partitioning time or the total query-
ing time on the partition.

Also, note that our partitioning approach could fail in the case
that a single subtree selected by the partitioning path has a size
exceeding main-memory capacity. In our experiments based on
XMark queries, this never happened, and we believe that the ma-
ority of iterative queries used in practice iterates on sequences of
subtrees where each tree has a size that fits standard memory ca-
acity of main memory systems.

5.2 The Partitioning Algorithm

Algorithm 1 provides a formal presentation of our partitioning
scheme. It is a recursive algorithm and takes as input a 4-tuple
\(<l; \tau; cSize; pld>\) representing the current state of the recursive
process: namely, this tuple indicates that the current node to be
matched against the current aligned partitioning path-set \(\tau\) is \(l\),
that the current size of the part under construction is \(cSize\), and that
the current number of created parts is \(pld\). Of course, the algorithm is
initially invoked with \(cSize=0\) and \(pld=1\), while the location \(l\) is
the root of the input XML tree \(t=(\sigma, l, \text{and } \tau=\text{Down}(\{PP_Q\}))\),
the alignment of the initial partitioning path for the iterative query
\(Q\) to execute. The store \(\sigma\) and the query \(Q\) are both assumed global
parameters.

In the algorithm, the function \(\text{RenameStore}(\sigma; \ldots)\) produces a
new store obtained from \(\sigma\) by renaming each location \(l\) to \(l^{\text{temp}}\). We
will use \(\text{PartLabel}\text{-}^{-1}(\sigma')\) to undo the renaming in the store \(\sigma'\).

The algorithm distinguishes among three main cases. In the first
case (lines 3-10) the current node is an element node being a ter-
nimal match for the initial partitioning path \(PP_Q\). In this case a
projection algorithm, if any, is called to compute the projection of
the subtree rooted at this node together with its size.\(^5\) Then (lines 7-
10) the algorithm adds the resulting subtree to the current part, and
checks whether the size of the projected subtree plus the current
size does not exceed the maximal size: if the check is positive, then
the current size is incremented with the projection size, otherwise
the current size is reset to 0 and a new (empty) part is created (this
empty part will be filled in subsequent steps of the processing).

In the second case (lines 11-29) the current \(l\) node is a possible
non-terminal match for the partitioning path \(PP_Q\). A temporary
current size variable \(cSize_{\text{temp}}\) registers the current size plus twice
the length of the current tag (both start and ending tags have to be
taken into account). By considering \(\text{PartLabel}\text{-}^{-1}(\sigma')\), the
computation recursively goes on for each child \(l\) of the \(l\) node
(lines 13-19). For each \(l\), partitioning is made according to paths
obtained by alignment of paths resulted by residuation (line 12).
Also, partitioning for each child \(l\) is made according to the
current size and partition number produced by the partition process
for the child \(l\).

Once partitioning for all children is done, the resulting store
\(\sigma'_{\text{temp}}\) is checked for emptiness (line 20). If the store is empty,
then the current node \(l\) is not deemed as a non-terminal match as it
has no descendant being a terminal match. Hence, the node does
not contribute to the current part (it is pruned out), and the output
current size is set to the input current size; note that in this case
the input part identifier \(pld\) is unchanged. Otherwise (lines 23-28),
the current partition and size have to be updated. The output cur-
cent size is set to \(cSize_{\text{temp}}\) (line 23), registering the current size of
the current part or, eventually, the last part created while processing
\(^5\)If no projection algorithm is available, \(\text{Projection}(\sigma; l, Q)\) just
returns the input subtree and its size.
children \( l_i \)'s. After this (lines 24-28), the current partition \( \sigma' \) is enriched with \( \sigma_{temp} \) and (lines 27-28) with elements for the current location \( i \) indexed by all new part numbers \( pId_j \)'s produced while processing \( l_i \)'s sub-trees (recall that for a child \( l_i \) more than one part could be created). In this case \( i \) has to be indexed accordingly.

To this end the algorithm uses a function \( ren-extr(L, p, D) \) which takes as input the sequence \( L \) of children locations, a part identifier \( p \), and the domain \( D=dom(\sigma') \) of the created sub-partition. The role of the renaming function \( ren-extr(L, p, D) \) is to extract the sub-sequence of \( L \) used to create the part \( p \) in \( \sigma' \), and to adorn with \( p \) each location in this sub-sequence. Formally, we have:

\[
ren-extr(L, p, D) = \begin{cases} 
\{ \} & \text{if } L = \{ \} \\
\{ \{ l_i \mid ren(L', p, D) \} \} & \text{if } L = \{ l_i \} \text{ and } l_i \in D \\
\{ \{ l_i \} \} & \text{if } L = \{ l_i, l_j \} \text{ and } l_i, l_j \notin D 
\end{cases}
\]

For instance, assume that the current node of the case is \( l \to a[1,2,1,3] \), and that for subtrees rooted at \( l_1, l_2 \) data are put in part 3, while for the subtree rooted at \( l_3 \) data are put/split in two parts 4, 5. Then the renaming extraction produces \( \{ l_1, l_2 \} \) and \( \{ l_3 \} \).

Finally, the third case applies when the current node does not match the partitioning path, hence the algorithm produces an empty part set, and preserves the current part size and number.

Dealing with a workload. A nice property of our partition-projection system is that it can deal with a workload formed by queries \( Q_1, \ldots, Q_n \) in a natural way. To this end it suffices to consider a global partitioning path set \( \tau_PP=\bigcup_{i} \{ PP_i \} \), where \( PP_i \) is the partitioning path of \( Q_i \).

This follows from the fact that our system is already able to deal with a set \( PP \) (recall that \( Down() \) produces set of paths in the presence of \( DOS \) axis). So, with \( \tau_PP=\bigcup_{i} \{ PP_i \} \) partitioning is performed in terms of nodes matching at least one of the paths \( PP_i \)'s, and, if a projection algorithm is available, the corresponding sub-trees are projected by keeping into account the data needs of the whole workload.

Soundness of partition algorithm is stated by the following theorem, using the notation \( Part_i(\sigma') \) to indicate the \( i \)-th part tree \( \sigma'_i \) in the partition \( \sigma' \). Formally \( \sigma_i=\{ l_i \to a[L] \mid i' \to a[L]\in\sigma' \} \).

Theorem 5.4 (Soundness of Partition) Let \( pSize \) be a size threshold value, let \( Q_j \) be iterative with partitioning paths \( PP_j \), with \( j = 1 \ldots m \), and let \( \tau_PP=\bigcup_{i} \{ PP_i \} \). Let \( t=(l_i) \) be an XML tree. If \( Part(t) \cup Down(\tau_PP) = \{ \} \), then:

\[
Q_j(t) \equiv Q_j(t_1) \cdots Q_j(t_{pId})
\]

where \( t_i = \text{PartLabel}^{-1}(Part_i(\sigma')) \).

5.3 Streaming Implementation

We implemented our algorithms in a streaming fashion on top of a SAX parser. The main data structure we used is a stack where we record the current status of the algorithm, with the following information:

- the tag of the element being processed;
- the attributes of the element being processed;
- the result of the resudson of \( PP \);
- a boolean flag open;
- the size of the XML tree fragment above the sequence of nodes matching \( PP \) (size).

Algorithm 1: Part

```
Input: A location \( \xi dom(\sigma) \), a partitioning path-set \( \tau \), a part size \( cSize \), a part number \( pId \).
Output: A store \( \sigma', \) a part size \( cSize' \), part number \( pId' \).

1 begin
2 let \( \sigma(l) = a[l] \)
3 % Case 1: \( l \) is a PP target node
4 if \( Res(a; \tau) = \langle \_ ; \_ ; \_ \rangle \) then
5 let \( \sigma'(\xi), \text{Size} = \text{Projection}(\sigma; \tau; \xi) \)
6 % projection always keeps the node \( l \) in \( \sigma' \)
7 \( \sigma' := \text{RenameStore}(\sigma'; \text{pId}) \)
8 if \( \text{cSize} + \text{Size} \leq \text{pSize} \) then
9 \( \text{cSize} := \text{cSize} + \text{Size}; \text{pId}' := \text{pId} \)
10 else
11 \( \text{cSize}' := 0; \text{pId}' := \text{pId} + 1 \)
12 % Case 2. \( l \) is not a PP target node
13 if \( Res(a; \tau) = \langle \_ ; \_ ; \text{ok}, \_ \rangle \) then
14 \( \text{pId}' := \text{pId}; \text{temp} := \emptyset \)
15 \( \text{cSize}' := \text{cSize} + 2 \cdot \text{length}(a) \)
16 \( \text{cSize}' := \text{cSize} + 2 \cdot \text{length}(a) \)
17 for \( i = 1 \ldots \text{m} \) do
18 \( \text{cSize}' := \text{cSize} \)
19 \( \text{cSize}' := \text{cSize} \)
20 \( \text{pId}' := \text{pId} \)
21 % Case 3. \( l \) does not match PP
22 else if \( Res(a; \tau) = \langle \langle ; \_ ; \_ \rangle \rangle \) then
23 \( \sigma' := \emptyset; \text{cSize}' := \text{cSize}; \text{pId}' := \text{pId} \)
24 return \( \langle \sigma', \text{cSize}', \text{pId}' \rangle \)
```

The implementation also tracks the size of the current part (\( \text{currSize} \)). By using this status information, we can split our algorithm in two distinct parts, which are executed when \( \text{startElement} \) and \( \text{endElement} \) are invoked, respectively.

In \( \text{startElement} \) we put the updates of size and \( \text{currSize} \), as well as the resudson and alignment of the current partitioning path, but we defer partitioning decisions to \( \text{endElement} \) calls. If a projection algorithm is available, it is invoked here.

In \( \text{endElement} \) we compare the size of the current part with the maximal part size, and create a new part if necessary. The creation of a new part requires one to iterate on the stack, close all the open tags, and reopen them in reversal order in the new part; the algorithm also resets \( \text{currSize} \) to the value of \( \text{size} \) that we find on top of the stack.

6. EXPERIMENTAL EVALUATION

In the previous sections, we described a novel XML data partitioning scheme that, given a query \( Q \) and an input document \( t \),
partitions $t$ in a set of fragments $\{t_1, \ldots, t_n\}$ so that $Q(t)$ is equivalent to the concatenation of $Q(t_1), \ldots, Q(t_n)$. When this partitioning scheme is applicable, it can improve the scalability of existing main-memory engines, as it allows the system to process one part per time.

In this section we present an experimental evaluation of the proposed approach. We will first show that the proposed algorithm significantly improves the scalability of a popular main-memory query engine. Then, we will show that partitioning, when combined with a projection algorithm, introduces little overhead with the projection algorithm. Finally, we will experimentally analyze the relation between the overall performance of the system and the actual value of $pSize$ (the maximum part size).

### 6.1 Experimental Setup

We implemented our partitioning algorithm, as well as a standard path-based projection algorithm, in Java 6 and tested their behavior on the XMark benchmark [13]. In particular, we evaluated our system on XMark documents whose size ranges from 1GB to 9GB, and restricted our analysis to queries $Q_1, Q_2, Q_{10}, Q_{14}, Q_{20}$ (the iterative core of XMark). Unless otherwise specified, we set $pSize$ to 10000000 bytes ($\sim$ 95.36 MB), which is a very conservative value.

In our experiments we use Qizx-open [1] as query engine (similar results can be obtained for BaseX [2]). All experiments were performed on a 2.53 Ghz Intel Core 2 Duo machine (4 GB main memory) running Mac OS X 10.6.8. We allocated 1024 MBs of main memory to the Java Virtual Machine; all XML documents were loaded on an external USB2 7200 rpm 1 TB disk unit.

To avoid the perturbations introduced by system activity, we ran each experiment ten times, discarded the best and the worst performance, and computed the average of the remaining times.

### 6.2 Experiments

In our first experiment we evaluate the scalability of the system composed by the partitioning algorithm, the projection algorithm (plugged in the partitioning algorithm), and a main-memory query engine. To this end, we execute all the queries in the iterative core of XMark on the whole collection of input documents.

The results we obtained are shown in Figure 6a, where we report the overall time required by the system to partition and project the input document, to evaluate the input query on each fragment, and to concatenate the final results. As it can be observed, the system scales beautifully and successfully evaluates all the queries, even on very large XML documents. It can also be noted that queries $Q_{10}$ and $Q_{14}$ are more expensive to process, as they traverse larger fractions of the input documents; however, our system does not expose scalability issues even on these queries.

In our second experiment we evaluate the scalability of our approach on a workload comprising the queries of the iterative core of XMark. To perform this experiment, whose results are shown in Figure 6b, we create a global partitioning path set and a global projection path set by collecting all the partitioning and projection paths of the single queries; these sets are used to partition the input document (and to project each single part), so to meet the data needs of all the queries being executed.

The results of this experiment indicate that our system is able to successfully process rather complex workloads on large documents and that it exhibits a linear behavior. For the sake of completeness, we repeated this experiment by using projection only; however, we got no experimental results, as, even in the case of the 1GB document, the projected documents were too big to be handled by the query engine.

In our final experiment we investigate the relation between $pSize$ and the processing time, i.e., the impact of different values of $pSize$.
on the overall performance of our approach. To this end, we evaluate all the queries in the iterative core on the 4GB document and vary \( pSize \) from 100000000 bytes to 700000000 bytes (\( \sim 668 \) MB).

The results we obtained are shown in Figure 8. Surprisingly enough, we can observe that the value of \( pSize \) has no significant impact on the overall performance. This is quite counter-intuitive, as bigger values of \( pSize \) should decrease the total number of bytes written to disk.

6.3 Summing Up

To summarize, our experiments prove that the partitioning approach scales beautifully and is only slightly slower than the projection approach, which was unable to complete the assigned tasks.

We also discovered that the actual value of \( pSize \) has no significant impact on the overall performance; this suggests that \( pSize \) can be tuned by looking only at scalability issues.

7. RELATED WORKS

Compared to the path-based approach for XML projection [12], as shown by our tests, our approach provides a significant improvement in the case of queries requiring a large part of the input in order to be processed, and, quite importantly, in the case of a workload. The same considerations hold for the type-based projection techniques proposed in [4, 11]. As [12], these two techniques keep in a single document all the data needed by a query or a workload. Hence, this document is likely to be too big to fit in main-memory, in the cases mentioned above.

Techniques for partitioning XML documents have already been explored. Our technique resembles that of [8] where an horizontal partitioning technique has been proposed in order to ensure parallel execution of single XPath queries. The partitioning technique proposed in this work can be performed on the main-memory representation of the XML document. As a consequence, very large XML documents cannot be managed. In [10] a vertical partitioning technique has been proposed still with the aim of parallel and distributed execution of XPath queries. The technique can handle very large documents, but requires the use of schema information on the input document. Both techniques [8, 10] require strong interventions inside a query engine. A recent work [9] proposes new efficient algorithms for the distributed evaluation of XPath queries. This work uses horizontal-vertical partitioning, and assumes data have been statically partitioned according to existing techniques.

Differently from these works, we address a wide class of XQuery queries, we do not require schema information, and are able to deal with a workload executed on very large documents. Also, in contrast to existing related works, our technique does not require to modify the internal components of a query engine: it can be simply plugged on top of an existing one.

8. CONCLUSIONS

In this paper we presented a novel projection-partitioning technique for XML document. This technique generalizes existing path-based approaches, and applies to a large class of queries.

The proposed approach analyzes an input query and, if the query is iterative, extracts all the relevant paths and uses them to partition the input document. As shown in our experimental evaluation, by executing the input query on each part and combining the partial results, existing main-memory query engines can process an iterative query on very large input documents.

As each part can be queried independently by a distinct instance of the query engine, we are currently investigating potential applications of the proposed approach to cloud computing environments.
Finally, we plan to investigate on partitioning techniques for executing XQuery updates. Some preliminary results have already shown that also for updates, in many cases partitioning can be applied in order to handle very large documents.

9. ACKNOWLEDGEMENTS

We would like to thank Amine Baazizi, Marina Sahakyan and Federico Ulliana for helpful discussions and comments on this work. We would also like to thank the anonymous referees for their useful feedback and suggestions. This work has been partially funded by the Codex project, Agence Nationale de la Recherche, decision ANR-08-DEFIS-004.

10. REFERENCES