Inf Syst Front (2018) 20:63–90
DOI 10.1007/s10796-017-9744-4

Evaluating Queries and Updates on Big XML Documents

Nicole Bidoit[1] · Dario Colazzo[2] · Noor Malla[3] · Carlo Sartiani[4]

Published online: 22 March 2017

Abstract In this paper we present Andromeda, a system for processing queries and updates on large XML documents. The system is based on the idea of statically and dynamically partitioning the input document, so as to distribute the computing load among the machines of a MapReduce cluster.

Keywords XML · Cloud computing · Map/Reduce

1 Introduction

In the last few years cloud computing has attracted much attention from the database community. Indeed, cloud computing architectures like Google Map/Reduce (Dean and Ghemawat 2004) and Amazon EC2 proved to be very scalable and elastic, while allowing the programmer to write her own data analytics applications without worrying about interprocess communication, recovery from machine failures, and load balancing. Therefore, it is not surprising that cloud platforms are used by large companies like Yahoo!, Facebook, and Google to process and analyze huge amounts of data on a daily basis.

The advent of this novel paradigm is posing new challenges to the database community. Indeed, cloud computing applications might also be built upon *parallel databases*, that were introduced nearly two decades ago to manage huge amounts of data in a very scalable way. These systems are very robust and very efficient, but for the following reasons their adoption is still very limited: (i) they are very expensive; (ii) their installation, set up, and maintenance are very complex; and, (iii) they require clusters of high-end servers, which are more expensive than cloud computing clusters.

In this paper we present the formal specifications, implementation details, and experimental results about Andromeda, a system for processing queries and updates on very large XML documents, usually generated and processed in contexts involving scientific data and logs (Choi et al. 2012). Andromeda supports a large fragment of XQuery (Boag et al. 2010) and XUF (XQuery Update Facility) (Robie et al. 2011), and exploits dynamic and static partitioning of input documents in order to distribute the processing load among the machines of a Map/Reduce cluster. The proposed technique applies to a class of queries and updates called *iterative*.

1.1 System Overview

The basic idea of our system is to dynamically and/or statically partition the input data to leverage on the parallelism

✉ Carlo Sartiani
sartiani@gmail.com

Nicole Bidoit
nicole.bidoit@lri.fr

Dario Colazzo
dario.colazzo@dauphine.fr

Noor Malla
noorwm@hotmail.com

[1] Laboratoire de Recherche en Informatique, Université Paris-Sud, CNRS UMR 8623, Université Paris-Saclay, Orsay, France

[2] Université Paris-Dauphine, PSL Research University, CNRS, LAMSADE 75016, Paris, France

[3] Saudi School of Paris, Paris, France

[4] DIMIE - Università della Basilicata, Potenza, Italy

of a Map/Reduce cluster and to increase the scalability. The architecture of our system is shown in Fig. 1, and described next.

Andromeda supports the execution of *iterative* XQuery queries and updates, i.e., queries and updates that i) use forward XPath axes, and ii) first select a sequence of subtrees of the input document, and then iterate, on each of the subtrees, the same operation.

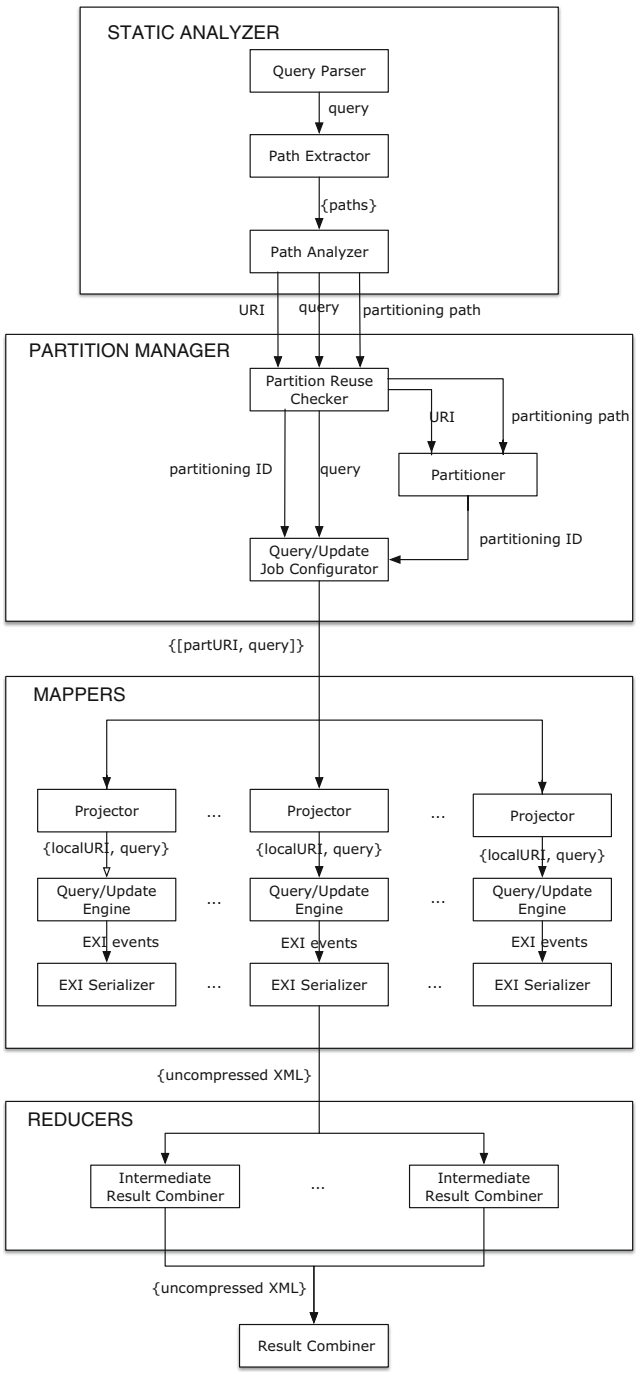


Fig. 1 Andromeda system architecture

As an example of iterative query, consider the following query on XMark documents (Schmidt et al. 2002) (assume $\$auction$ is bound to the document node $doc(\text{“}xmark.xml\text{”})$).

```
for $i in $auction/site//description
where contains(string(exactly-one($i)), “gold”)
return $i/node()
```

The query iterates the same operation on each subtree selected by $\$auction/site//description$ and, hence, is iterative.

This property is enjoyed by many real world queries: for instance, in the XMark benchmark 13 out of the 20 predefined queries are iterative.[1] Non iterative queries are typically those performing join operations on two independent sequences of nodes of the input documents, although iterative queries may perform join operations, as in:

```
for $i in $auction/site//description,
$x in $i//keyword,
$y in $i//listitem
where $x = $y
return x
```

Iterative updates are also of practical interest and characterized in the same spirit, and include the wide class of updates that modify a sequence of subtrees, and such that each `delete/ rename/insert/replace` operation does not need data outside the current subtree. As an example of iterative update, consider the following one:

```
for $x in $auction/site/regions//item/location
where $x/text() = “United States”
return (replace value of node $x with
“USA”)
```

This update iterates over *location* elements and replaces each occurrence of “United States” with “USA” and does not require information outside the subtrees rooted at *location* elements.

The following update, instead, is not iterative as it accesses all *description* elements (in the `if` clause), but deletes nodes in a distinct fragment of the input document.

```
if $auction//description//text() = “word”
then delete nodes
$auction/site/regions/australia//item
```

Andromeda handles query and update execution as follows. When a user submits a query or an update to the system, the STATIC ANALYZER parses the input query/update to extract the information required for checking the iterative property, and for partitioning the input document D.

[1] Queries from Q_1 to Q_5, and from Q_{13} to Q_{20} are iterative.

This information is essentially the set of paths used in the query/update, enriched with details about bound variables. It is passed to the PARTITION MANAGER, which uses it to create, from the input document, a set of subdocuments called *parts*. These parts are well-formed XML documents that are independently processed by multiple instances of the query processor to generate the final result.

To illustrate, consider the following iterative query:

$$\texttt{for } \$x \texttt{ in } /a, \$y \texttt{ in } \$x/b$$
$$\texttt{where } \$y/c/d$$
$$\texttt{return} < res > \$y/c/e < /res >$$

For this query the STATIC ANALYZER extracts the following set of paths:

$$\{\ /a\{for\ x\}, /a\{for\ x\}/b\{for\ y\},$$
$$/a\{for\ x\}/b\{for\ y\}/c/d, /a\{for\ x\}/b\{for\ y\}/c/e\}$$

The set of paths inferred by the STATIC ANALYZER comprises the paths contained in the `for`, `where`, and `return` clauses; however, for the sake of simplicity of the formal approach, only paths used in `for` iterations are taken into account as possible partitioning paths.

The STATIC ANALYZER processes theses paths and identifies $/a/b$ as the path on which the query iterates; this path, called *partitioning path*, is used during the partitioning process to appropriately build documents parts, and to ensure that information that should not be split among multiple parts is kept in the same part. Intuitively, if a node matches the partitioning path, then the subtree rooted at this node is required to be kept entirely in a single part. This *indivisibility* property is necessary to distribute the evaluation of the query over the document parts and recover the result of the query over the input document by a simple concatenation.

In the case of updates, the STATIC ANALYZER works in a similar way. The system distinguishes between *simple updates*, i.e., updates consisting of a single `delete`, `rename`, `insert`, `replace` operation without `for`-iterations, and updates containing iterations. As in the case of queries, partitioning paths are used to recognise subtrees that should not be split. Again, this indivisibility property is necessary in order to ensure semantics preservation once the update is distributed over the elements of the partition.

When a document is partitioned for the first time, the PARTITION MANAGER uses the partitioning paths to perform the actual partitioning. It also computes the input document DataGuide (Goldman and Widom 1997), which serves to check if an existing partition can be reused for a newly issued query or update. During the partitioning process, parts are encoded as EXI (Efficient XML Interchange) files[2] through the streaming encoder of Exificient (2015) to significantly reduce the storage space required and, most importantly, to cut network costs.

The MapReduce-based query/update evaluation proceeds as follows. Once the STATIC ANALYZER has extracted path information from the input query/update, and the PARTITION MANAGER has found an existing partition or created a new one for processing the query/update, parts are assigned to mappers for query/update processing.

When processing a query, each mapper not only receives the address of each assigned part, but also the paths extracted by the STATIC ANALYZER, to further optimize query evaluation by projection/pruning. The query is executed on each pruned part by a local instance of Qizx-open (2013), a main-memory query engine, which exports the results, encoded in XML format, to the distributed file system.

When processing an update, each document part must be kept whole and thus projection/pruning is irrelevant. The local instance of Qizx-open executes the update on each part, and stores the updated part in the distributed file system, encoded in the EXI format.

The query/update evaluation final step constructs the result from the partial ones. This step works a bit differently for queries and updates. Indeed, partial results of a query are simply concatenated, while partial results of an update must be merged.

Paper Outline The paper is organized as follows. Section 2 introduces preliminary definitions over which we develop, in Section 3, the static analysis for iterative queries and updates. Section 4 is dedicated to the presentation of the partitioning algorithms for iterative queries and updates as well as the final result construction. For sake of simplicity, this is done in a DOM-oriented fashion although the actual implementation of these algorithms complies with a SAX-based streaming approach presented in Section 5. Section 6, next, describes and analyzes the experiments that have been performed in order to validate these algorithms. The paper concludes with a discussion on related works in Section 7.

2 Preliminaries

Following (Benedikt and Cheney 2009), we represent an XML document as a store. Figure 2 describes a store σ associated to an XML tree[3] in order to illustrate the following notions. A store σ associates to each node location (identifier) l either an element node a[L] or a text node $text$[s].

[2]EXI is a binary format, proposed by the W3C, for compressing and storing XML documents.

[3]In the following, we will use *XML tree* to denote any rooted, well-formed XML document, while *XML forest* will denote a collection of rooted, well-formed XML documents.

When $\sigma(\mathrm{l})$=a[L] (also written $\mathrm{l} \leftarrow \mathrm{a}[\mathtt{L}]\in\sigma$), a is the node element tag and $\mathtt{L}=(\mathrm{l}_1,\ldots,\mathrm{l}_n)$ is the ordered sequence of the child locations for l. When $\sigma(\mathrm{l})=text[\mathrm{s}]$ (also written $\mathrm{l} \leftarrow text[\mathrm{s}]\in\sigma$), s is the textual content of the text node. For instance, the XML tree of Fig. 2 would be represented by a store containing a location l_1, describing the tree root, locations l_2, l_3, and l_4, describing root children, etc.

An XML tree t is a pair (σ, l_t), where l_t is a distinguished location in σ which is associated to the root element of the XML tree. We denote by $\mathrm{dom}(\sigma)$, resp. $\mathrm{dom}(t)$, the set of locations of the store σ, resp. of the tree t. Given a location $\mathrm{l}\in\mathrm{dom}(\sigma)$, $\sigma@\mathrm{l}$ denotes the subtree of σ rooted at l. For simplicity, for $t=(\sigma, \mathrm{l}_t)$, we abusively use t instead of σ and, write $\mathrm{l} \leftarrow \mathrm{a}[\mathtt{L}]\in t$ instead of $\mathrm{l} \leftarrow \mathrm{a}[\mathtt{L}]\in\sigma$.

In order to compare query and update results, we need to define the notion of equivalence among XML trees. Two trees t and t' are equivalent, denoted $t \equiv t'$, iff they are isomorphic, i.e., they possibly differ only in terms of location names. When σ and σ' are forests, i.e., collections of XML trees, and $\mathtt{L}=(\mathrm{l}_1,\ldots,\mathrm{l}_n)$ and $\mathtt{L}'=(\mathrm{l}'_1,\ldots,\mathrm{l}'_n)$ are sequences of locations, we write $(\sigma, \mathtt{L}) \equiv (\sigma', \mathtt{L}')$ to state that $\sigma@\mathrm{l}_i \equiv \sigma'@\mathrm{l}'_i$, for $i=1\ldots n$. Finally, when σ and σ' have no common location, the concatenation $(\sigma, \mathtt{L})\cdot(\sigma', \mathtt{L}')$ is the store defined by $(\sigma\cup\sigma', (\mathtt{L}, \mathtt{L}'))$, where L, L′ denotes the concatenation of L and L′.

Below, () denotes the empty sequence of locations, while {L} denotes the set of locations of the sequence L. We say that L′ is a projection of L, denoted $\mathtt{L}' \preceq \mathtt{L}$, when L′ is a subsequence of L. For instance, $\mathrm{l}_1,\mathrm{l}_3 \preceq \mathrm{l}_1,\mathrm{l}_2,\mathrm{l}_3$, while $\mathrm{l}_3,\mathrm{l}_1 \not\preceq \mathrm{l}_1,\mathrm{l}_2,\mathrm{l}_3$.

Definition 1 (XML Projection) A tree $t'=(\sigma', \mathrm{l}_{t'})$ is a *projection* of a tree $t=(\sigma, \mathrm{l}_t)$, noted as $t' \preceq t$, if $\mathrm{l}_{t'}=\mathrm{l}_t$, and for each location $\mathrm{l}\in dom(\sigma')$:

$$\mathrm{l} \leftarrow \mathrm{a}[\mathtt{L}']\in\sigma' \Rightarrow (\mathrm{l} \leftarrow \mathrm{a}[\mathtt{L}]\in\sigma \ \wedge \ \mathtt{L}' \preceq \mathtt{L})$$

Note that projection preserves tree root and that the projection of a tree is obtained by pruning some of its subtrees. Figure 2 shows a projection of a simple XML tree and its associated store. Projection is used to define XML partitions.

Definition 2 (XML Partition) A collection $\{t_1,\ldots,t_\kappa\}$ of trees is a *partition* of a tree t if we have $t_i \preceq t$, for each tree t_i and for each location $\mathrm{l}\in dom(t)$:

$$\mathrm{l} \leftarrow text[\mathrm{s}]\in t \Rightarrow \exists\, t_i.\ \mathrm{l} \leftarrow text[\mathrm{s}]\in t_i$$
$$\mathrm{l} \leftarrow \mathrm{a}[\mathtt{L}]\in t \Rightarrow \{\mathtt{L}\}= \bigcup_{\mathrm{l}\leftarrow \mathrm{a}[\mathtt{L}_i]\in t_i} \{\mathtt{L}_\mathrm{i}\}$$

An element of the partition, for instance the tree t_i, is called a *part*, and is a projection of t. Properties above say that each text node has to belong to at least one part, and that element nodes are partitioned such that no child is left out. Figure 3 shows two partitions of the document in Fig. 2.

Since we have defined the formal representation of XML documents, and introduced the notions of XML projection and XML partition, which form the theoretical foundations of our approach, we are now ready to describe the query and update languages supported by our system.

2.1 Queries

We use the fragment of XQuery described by the grammar below. It comprises `for`, `let`, and `return` clauses as well as `if-then-else` statements, and allows one to specify *self*, *child*, and $descendant-or-self$ XPath axes (Berglund et al. 2010). Next, $descendant-or-self$ is abbreviated by *dos*.

Queries	Q	::= () | Q, Q | <a>Q | Exp | |`if` (Q) `then` Q `else` Q |`for` x `in` Q `return` Q |`let` x := Q `return` Q
XPath expressions	Exp	::= x | $x/Step$ | $/Step$
Step expressions	$Step$	::= $Axis :: NT$
Axis	$Axis$	::= *self* | *child* | *dos*
Node tests	NT	::= a | *node*() | *text*()

In this grammar, we only consider queries in *canonical form*; this means that any clause of the form

`for` x `in` $/a/b/c$

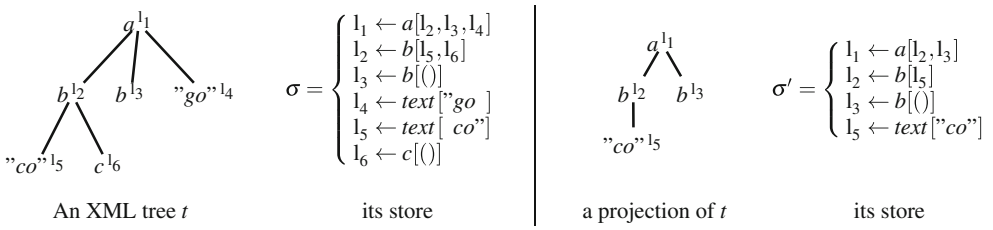


Fig. 2 Representation of XML trees as stores and projection

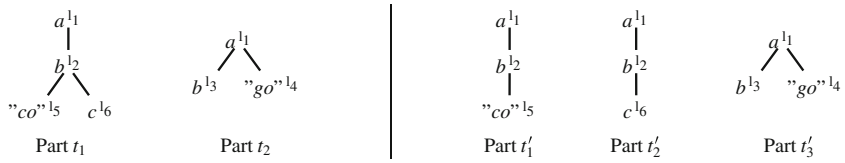


Fig. 3 Two possible partitions of the XML tree t in Fig. 2

is represented as

```
for x in /a
return for y in x/b
        return for z in y/c
```

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 `let`/`for` binders), and **ii)** no variable name is used twice in `for-let` bindings. Property **i)** ensures that $well-formed$ queries start navigating documents from their root: for instance, `for` y `in` $x/Step$ `return` Q is not $well-formed$, while `for` y `in` $/Step$ `return` Q is. Property **ii)** simplifies the formal treatment, and can be always obtained by α-renaming.

In this work, we focus on queries working 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 `return` clause of a `for/let` expression, as happens in most practical cases (e.g., all XMark queries are of this form, provided that in some queries `let` bindings are inlined).[4]

The evaluation of a query Q on an input tree $t{=}(\sigma, \mathrm{l}_t)$, denoted by $Q(t)$, yields a pair $(\sigma_Q, \mathtt{L}_Q)$, where the store σ_Q is a forest which extends the initial store σ with the new nodes built by Q, while $\mathtt{L}_Q$ is the sequence of location nodes returned by the query and defined in σ_Q. Due to lack of space, we do not report here formal semantics of this XQuery fragment (a concise and elegant formalization can be found in Benedikt and Cheney (2009)).

2.2 Updates

Concerning updates, we address XQuery update expressions obeying the following grammar, where Q refers to the query grammar of the previous section, and ϵ denotes the empty path. The resulting language is a significant fragment of XUF and comprises `for`, `let`, and `return` clauses, as well as the `if-then-else` conditional statement. The language also supports all *elementary* XUF update expressions (*delete*, *insert*, *rename*, and *replace*).

Path	P	$::= \epsilon \mid Step/P$
Target Path	P_{tg}	$::= /P \mid x/P$
Simple Query	Q_s	$::= ()\mid b\mid /P\mid x/P\mid$ <a> Q_s $\mid Q_s, Q_s$
Target Position	Pos	$::=$ `as first into` $\mid$ `as last into` $\mid$ `before` $\mid$ `after`
Node Case	N	$::=$ `node` $\mid$ `nodes`
Updates	U	$::= deleteN\ P_{tg}$
		$\mid renameN\ P_{tg} as$ a
		$\mid replaceN\ P_{tg} with Q_s$
		$\mid insertN\ Q_s Pos P_{tg}$
		$\mid U, U$
		$\mid$ `if` Q `then` U `else` U
		$\mid$ `for` x `in` Q `return` U
		$\mid$ `let` $x := Q$ `return` U

The main restrictions on updates are the following:

- simple query expressions Q_s, used as source expressions in `replace`/`insert`, can only use element and sequence construction, plus path navigation to select nodes in the input document;
- query expressions Q used in `for`/`let` and conditional updates are defined by the query grammar.

As already said, these restrictions have the purpose of ensuring a smooth formal characterization of iterative updates. At the same time, our update language is expressive enough to cover most of the needs of practical scenarios. For instance, several update expressions used in W3C XQuery Update Facility 1.0 (Robie et al. 2011) strictly respect the syntax of our update language, while other updates use function calls, conditions, and arithmetic operations that are not supported by our grammar. However, as we will illustrate, our approach can be easily extended to deal with these mechanisms by means of simple query rewritings. As other examples, all update expressions used in Baazizi et al. (2011) and Bidoit et al. (2013) meet our restrictions. Examples of basic and complex (e.g., nested updates inside query iterations) expressions are shown below, where $/a$ abbreviates $/child :: a$.

$U_1 =$ `delete nodes` $\$doc/a/f$

$U_2 =$ `insert node` <n/> `as first into` $\$doc/a/b$

$U_3 =$ `rename node` $\$doc/a/f$ `as` "new"

[4] For instance, inlining is needed for Q_{10}.

$U_4 =$ `for` $\$x$ `in` $\$doc/a/b$
`return insert node` $<m>$"$toto$"$</m>$
`after` $\$x$

Due to lack of space, we do not report the formal semantics of update expressions. An elegant formalization can be found in Benedikt and Cheney (2009).

3 Iterative queries and updates

The purpose of this section is to formally characterize iterative XQuery queries and updates, and prepare the reader to the presentation of the partitioning algorithms of Section 4. For both queries and updates, the iterative property is defined based on extracting relevant paths from the input query or update. Intuitively, these paths capture the traversals that are necessary for evaluating the query/update. These paths are central to our approach as they are analysed (i) for checking the iterative property, and also used (ii) for partitioning and, possibly, projecting the input document.

Extracted paths obey the following grammar:

$$P ::= \epsilon \mid /S \mid P/S \qquad S ::= Step \mid Step\{for\ x\}$$

where ϵ denotes the empty path.

Intuitively, for a query Q, if a path $P'\{for\ \ x\}/P''$ is extracted from Q, then it indicates that a subquery of Q has the shape `for` x `in` Q_1 `return` Q_2, where P' is extracted from Q_1, and P'' is extracted from Q_2, as a suffix of P'.

Variable information ($\{for\ \ x\}$) is important to identify iterative queries/updates and partitioning paths, while ignored for the purposes of partitioning and projection. Thus, in the following $ErV(P)$ denotes the path obtained from P by removing any variable information.

Definition 3 $ErV(P)$ Given an extracted path P, $ErV(P)$ is defined as follows:

$$ErV(P) = \begin{cases} P & \text{if } P = \epsilon \\ /strip(S) & \text{if } P = /S \\ ErV(P')/strip(S) & \text{if } P = P'/strip(S) \end{cases}$$

$$strip(S) = \begin{cases} S & \text{if } S \text{ does not end with } \{for\ x\} \\ S' & \text{if } S = S'\{for\ x\} \end{cases}$$

3.1 Iterative queries

Path extraction for queries captures query navigation and is used to determine, for a given query Q if it is possible to split any input document t into a collection $\{t_1, \ldots, t_\kappa\}$, so that $Q(t) \equiv Q(t_1) \cdot \ldots \cdot Q(t_\kappa)$.

Our path extraction approach, defined by means of the function $E_q(Q, \Gamma, m)$ of Fig. 4, resembles that of Marian and Siméon (2003) and Benzaken et al. (2006), although paths extracted according to $E_q(Q, \Gamma, m)$ carry a richer information, i.e., variable information. This function takes as input a query Q, a variable environment Γ, and a flag m, and returns a set of path extracted from Q and enriched with variable information.

For queries of the form `for` x `in` Q_1 `return` Q_2 (rule 11), the function first extracts paths from Q_1; they are then enriched with variable bindings and added to the environment Γ used for the recursive extraction of paths from Q_2.[5] In particular, Γ is used to associate the paths to each free occurrence of the variable x in Q_2 (rules 4 and 5). Rules for `let` expressions are similar, except that they do not keep track of variable information.

In these rules, we use a bivalued 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). This distinction is introduced because extracted paths are also used for projection, so it is important to know in which case a subtree selected by a path must be kept in the projection. When m=1, the terminal rules 5, 7, and 9 extend extracted paths with a $dos :: node()$ step, so as to capture the fact that descendants of nodes selected by the extended paths must be projected.

Please, observe that paths without variable information, as those returned by $ErV(P)$, could be directly computed by removing, from rules 3, 4, 5, 8, 9, 10, and 11 of $E_q(Q, \Gamma, m)$, any occurrence of $\{for\ x\}$.

Example 1 Consider the following query Q:

`for` x `in` $/a$ `return for` y `in` x/b `return`$(y/d, y/e)$

Paths are extracted from Q by evaluating $E_q(Q, \emptyset, 1)$, and are reported below:

$$P_1 = /a\{for\ x\} \qquad P_2 = /a\{for\ x\}/b\{for\ y\}$$
$$P_3 = /a\{for\ x\}/b\{for\ y\}/d/dos :: node()$$
$$P_4 = /a\{for\ x\}/b\{for\ y\}/e/dos :: node()$$

Intuitively, the issue here is to discover, from the extracted paths, if the query could be evaluated by first i) selecting a sequence `L` of nodes in the input XML tree, and then ii) iterating, in isolation, the same subquery over the subtrees rooted at these nodes. This sequence `L` should be selected by a path extracted from the query. For an iterative query, there may be more than one such path.[6] These paths are called candidate partitioning paths because they are also used to partition the XML tree. Below, we say that the path

[5] In Rule 11, $E_q(Q_1, \Gamma, 0)\{for\ x\}$ is a shorthand for $\{P\{for\ x\} \mid P \in E_q(Q_1, \Gamma, 0)\}$.

[6] The iterative property definition is based on the query syntax and does not deal with query rewriting issues which are beyond the scope of this article.

$$
\begin{array}{ll}
\textbf{1}.\ E_q((),\Gamma,m) = () & \textbf{2}.\ E_q(Q_1,Q_2,\Gamma,m) = E_q(Q_1,\Gamma,m) \cup E_q(Q_2,\Gamma,m) \\
\textbf{3}.\ E_q(l[Q],\Gamma,m) = \{/P\{for\,x\} \mid P\{for\,x\} \in \Gamma\} \cup E_q(Q,\Gamma,1) & \\
\textbf{4}.\ E_q(x,\Gamma,0) = \{P\{for\,x\} \mid P\{for\,x\} \in \Gamma\} & \textbf{5}.\ E_q(x,\Gamma,1) = \{P\{for\,x\}/dos::node() \mid P\{for\,x\} \in \Gamma\} \\
\textbf{6}.\ E_q(/P,\Gamma,0) = \{/P\} & \textbf{7}.\ E_q(/P,\Gamma,1) = \{/P/dos::node()\} \\
\textbf{8}.\ E_q(x/P,\Gamma,0 \quad \{P'\{for\,x\}/P \mid P'\{for\,x\} \in \Gamma\} & \textbf{9}.\ E_q(x/P,\Gamma,1) = \{P'\{for\,x\}/P/dos::node() \mid P'\{for\,x\} \in \Gamma\} \\
\multicolumn{2}{l}{\textbf{10}.\ E_q(\texttt{if}\ (Q)\ \texttt{then}\ Q_1\ \texttt{else}\ Q_2,\Gamma,m) = E_q(Q,\Gamma,0) \cup E_q(Q_1,\Gamma,1) \cup E_q(Q_2,\Gamma,1)} \\
\multicolumn{2}{l}{\textbf{11}.\ E_q(\texttt{for}\ x\ \texttt{in}\ Q_1\ \texttt{return}\ Q_2,\Gamma,m) = E_q(Q_1,\Gamma,0)\{for\,x\} \cup E_q(Q_2,\Gamma \cup \Gamma',m) \quad where\ \Gamma' = \{P\{for\,x\} \mid P \in E_q(Q_1,\Gamma,0)\})} \\
\multicolumn{2}{l}{\textbf{12}.\ E_q(\texttt{let}\ x := Q_1\ \texttt{return}\ Q_2,\Gamma,m) = E_q(Q_1,\Gamma,0) \cup E_q(Q_2,\Gamma \cup \Gamma',m) \quad where\ \Gamma' = \{P \mid P \in E_q(Q_1,\Gamma,0)\}}
\end{array}
$$

Fig. 4 Path extraction for queries

$P \in E_q(Q)$ is *maximal* if no other path in $E_q(Q)$ contains P as a prefix. In the following, unless otherwise noted, we will use $E_q(Q)$ as a shorthand for $E_q(Q, \emptyset, 1)$.

Definition 4 (**Candidate Partitioning Paths**) The set of candidate partitioning paths for a $well-formed$ query Q, denoted $Cand(Q)$, is defined as the set of paths $ErV\,P_{cand}$ with $P_{cand} \in E_q Q$ satisfying:

(i) P_{cand} is of the form $P_0\{for\ x\}$;
(ii) P_{cand} does not contain *text*() node-test conditions;
(iii) for each maximal path $P' \in E_q(Q)$, $\exists P'' \mid P' = P_{cand}/P''$.

Condition (i) states that each candidate path is used for iterating a sub-query. Condition (ii) rules out candidate paths that would iterate on text nodes (as in the query `for` x `in` $/dos :: text()$ `return` Q').[7] Condition (iii) is the most important one: the restriction on maximal paths is needed since, otherwise, the minimal common prefix of $E_q(Q)$ paths would be the only candidate.

Going back to Example 1, $ErV(P_1)$ and $ErV(P_2)$ are candidate paths, while $ErV(P_3)$ is not, as it violates conditions (i) and (iii) wrt P_4. Note that, if we alter the query by considering a new return clause `return` $(x/child :: d, y/child :: e)$, then the only candidate path is $ErV(P_1)$.

Clearly, in order for a query Q to be iterative, there must be at least one extracted path of the query identifying the sequences of nodes over which the same sub-query is iterated. This is reflected in the following definition.

Definition 5 (**Iterative Queries**) A $well-formed$ query Q is iterative iff $Cand(Q) \neq \emptyset$.

[7]The technical reason is that projection of text nodes which are sibling produces a single text node (the concatenation of the text nodes) rather than a sequence of text nodes, and this would introduce several complications that we preferred to avoid. Queries with these forbidden paths are quite infrequent.

The query of Example 1 is iterative. Iterative queries are quite common in practice and, as concrete examples, 13 out of the 20 XMark queries are iterative: namely, queries from Q_1 to Q_5, and from Q_{13} to Q_{20}.

When it is known that the query Q is iterative, one of the candidate paths is going to be selected and used to opportunely partition the input document and distribute the evaluation of Q over the document parts. The partitioning path, denoted PP, is defined as follows.

Definition 6 (**Partitioning Path**) Given an iterative query Q, the partitioning path PP for Q is the maximum length candidate path in $Cand(Q)$.

For Example 1, we have $PP = /a/b$. Picking up the longest candidate as the partitioning path is motivated by minimizing the size of the trees selected by the path, and thus maximizing the likelihood that each part yielded by partitioning fits in the available main memory.

3.2 Iterative Updates

As for queries, path extraction for updates captures update navigation and is meant to check whether, for a given update U, it is possible to split any input document t into a collection of parts $\{t_1, t_2, \cdots, t_\kappa\}$, such that

$$U(t) \equiv \oplus(U(t_1), \ldots, U(t_\kappa)) \tag{1}$$

where $\oplus$ is a fusion operator, slightly more complex than concatenation for queries used to obtain their final results. More details about $\oplus$ will be introduced next.

Before the formal presentation of update path extraction and the notion of iterative updates, we illustrate by examples several issues raised by updates wrt their possible partitioned evaluation.

We start the discussion with the case where partitioning is impossible. Typical updates of this class are such that the evaluation of the update source expression needs the whole input tree. This entails that, if the document is partitioned, then the source expression cannot be safely

evaluated in each part, as this misses pieces of the whole tree. To illustrate consider the elementary update below:

$U_1 =$ `replace node` $/a/b/c$ `with` $/a/f/g$

The source expression $/a/f/g$ makes the update not iterative. Similarly, the update U_2 below is not iterative as it performs an insert operation in terms of a sequence of subtrees determined by visiting the whole input.

$U_2 =$ `for` $\$x$ `in` $/a/b$
`return insert node` $/a/f/g$ `as last into` $\$x$

On the other hand, a class of updates that can be safely distributed over a partitioned tree is that of elementary updates whose source expression does not perform any kind of navigation, and whose target expression consists of a simple path expression. This kind of updates is frequently used in practice. Consider the following update, which resembles U_1 except that the source expression is now a constant value.

$U_3 =$ `replace nodes / a/ b/ c with < n/ >`

For such updates, their target path ($/a/b/c$ for U_3) is used as a partitioning path, thus ensuring that the content of subtrees selected by the target path is not spread over multiple parts.

Among iterative updates, we can also consider updates of the form `for` $\$x$ `in` Q `return` U, while ensuring, roughly speaking, that the embedded query Q and update U is iterative and moreover that their partitioning scheme is compatible (i.e., in some sense that paths in the U component have the partitioning path of Q as a prefix). The same holds for updates of the form `let` $\$x$ `:=` Q `return`. These cases will be made more formal shortly, by reusing the characterisation of iterative queries (Definition 6). To illustrate consider the following update.

$U_4 =$ `for` $\$x$ `in` $/a/b$
`return insert nodes` $\$x/f/g$ `as last into` $\$x$

For this update the following paths are extracted: $P_1 = /a/b\{for\ x\}$ and $P_2 = /a/b\{for\ x\}/f/g$. So, if we use the same reasoning as for iterative queries, the partitioning path is P_1 and, as already seen, this ensures that subtrees selected by this path are not split, thus gathering all the needed data to correctly evaluate each insert update. It is interesting to note that, according to this approach, update U_1 is not iterative as it has no candidate partitioning path: extracted paths are $P_1 = /a/b/c$ and $P_2 = /a/b/d$ and these have no common prefix including a $\{for\ x\}$ element (indicating a common top level iteration).

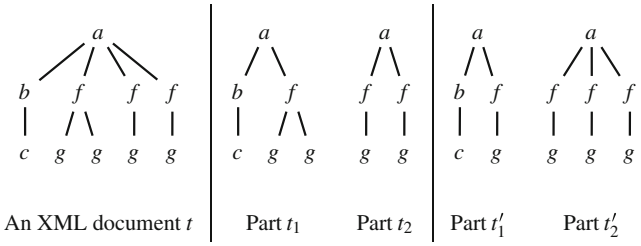


Fig. 5 An XML document t and two different kinds of partition

In order to further motivate the iterative update definition, we now introduce an example showing that, while partitioning the input tree, subtrees selected by the target expression of an update should not be split. Otherwise the update sub-operation could be performed mistakenly more times than needed.

$U_5 =$ `for` $\$x$ `in` $/a/f$
`return insert node <n/> as last into` $\$x$

This update inserts a new empty node `<n/>` as the last child of the target nodes selected by the path $P_1 = /a/f$. So, in order to safely distribute this update, the *full* sub-tree rooted at a target node is required, because of the `as last into` clause.[8] As illustrated by Figs. 5 and 6, if the subtree rooted at a P_1 target node is split, say, in two parts like in $\{t'_1, t'_2\}$ of Fig. 5, the `<n/>` insertion occurs twice: U_5 inserts a new `n`-node as last of each subtree rooted at `f`-node in the parts t'_1 and t'_2 and thus two `n`-nodes appear in $\oplus(t'_1, t'_2)$ which is of course incorrect wrt $U_5(t)$.

Instead, the partition $\{t_1, t_2\}$ of Fig. 5 ensures safe distribution, since $U_5(t) \equiv \oplus(U_5(t_1), U_5(t_2))$ now holds (see Fig. 7).

To summarise, we can conclude that, in order to enforce Eq. 1 for a given update U, our partitioning update scenario can be applied only when U performs many times the same operation on subtrees selected by a path expression, and, moreover, when each of these subtrees contains all the information for evaluating the update. Our examples also illustrate that these subtrees should absolutely not be split by partitioning. Updates satisfying this requirement are called *iterative updates*.

Path extraction for updates is specified by the function $E_u(U\ , \Gamma, m)$ in Fig. 8, that takes as input an update U, a variable environment Γ, and a flag m, and returns the set of paths extracted from U. As it can be seen, it strictly resembles path extraction for queries, especially in the use of parameters Γ (variable environment) and m. Also, note the use of query path extraction for extracting paths from path

[8] The same phenomenon would take place in case of a `as first into` update target position.

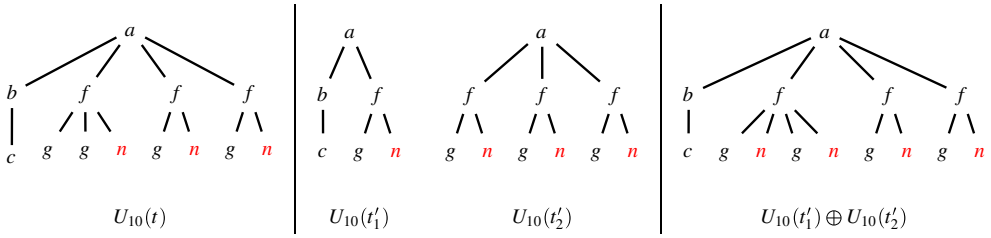


Fig. 6 Non-equivalence between $U_5(t)$ and $U_5(t'_1) \oplus U_5(t'_2)$

updates including queries as sub-expressions. In the following, for the sake of simplicity we use $E_u(U)$ as a shortcut of $E_u(U, \emptyset, 1)$

Example 2 Consider the update below:

$$U_{5'} = \texttt{for } \$x \texttt{ in } /a/f \\ \texttt{return rename node } \$x/g \texttt{ as } \text{“}n\text{”}$$

For this update, path extraction through $E_u(U_{5'})$ leads to $\{P_1, P_2\}$ below.

$$P_1 = /a/f\{for\ x\}$$
$$P_2 = /a/f\{for\ x\}/g/dos :: node()$$

We are now ready to provide a formal characterization of iterative updates, based on the previous introductory discussion and on $E_u(U)$.

Definition 7 (**Iterative Updates**) Iterative updates are defined according to the following case analysis:

- if U is an elementary update, then it is iterative if and only if one of the following holds:
  1. $U = delete\ N\ P_{tg}$;
  2. $U = rename\ N\ P_{tg}\ as$ a;
  3. $U = replace\ N\ P_{tg}\ with\ Q_s$;
  4. $U = insert\ N\ Q_s\ Pos\ P_{tg}$;

  where Q_s does not use any XPath expressions;
- if U is either a let-update or a for-update expression, then it is iterative iff $Cand(U) \neq \emptyset$ where this set of candidate paths is defined by Definition 4 based on the set of extracted paths $E_u(U)$.
- if $U{=}U_1, U_2, \cdots, U_n$, then it is iterative if each U_i is.

In the above definition, the first case has been already motivated by our examples. In particular, the condition on Q_s ensures that no global visit is needed to evaluate the source expression. The second case relies on, and strictly resembles, the characterisation of iterative queries (Definition 5), as discussed before, while the third case captures a sequence of updates, and relies on the two preceding cases for each update U_i in the sequence.

Before continuing with the formalization, it is worth noticing that `let-updates` are iterative only if the `let` binding does not use paths. For instance, the following update is not iterative.

$$U_6 = \texttt{let } \$x := /a/b \texttt{ return} \\ \texttt{if } \$x/c \texttt{ then delete node } \$x$$

This is because the `let` binding performs a global visit of the document before evaluating the inner update which, for reasons explained earlier, prevents any possible partitioning-based evaluation. Instead, the following update is iterative:

$$U_7 = \texttt{let } \$x :=\texttt{<c/>} \texttt{ return} \\ \texttt{for } \$y \texttt{ in } /a/b \texttt{ return insert } \$x \texttt{ after } \$y$$

It is worth to observe that, in the second item of Definition 7, if-expressions are not considered. Indeed, these expressions may occur as inner sub-expressions of iterative updates, like in the following example, a variant of U_7.

$$U_8 = \texttt{let } \$x :=\texttt{<c/>} \texttt{ return} \\ \texttt{for } \$y \texttt{ in } /a/b \texttt{ return} \\ \texttt{if } \$y/d \texttt{ then insert } \$x \texttt{ after } \$y$$

Once again, the reason why if-expressions are excluded as top-level expressions is that, in general, the if-condition

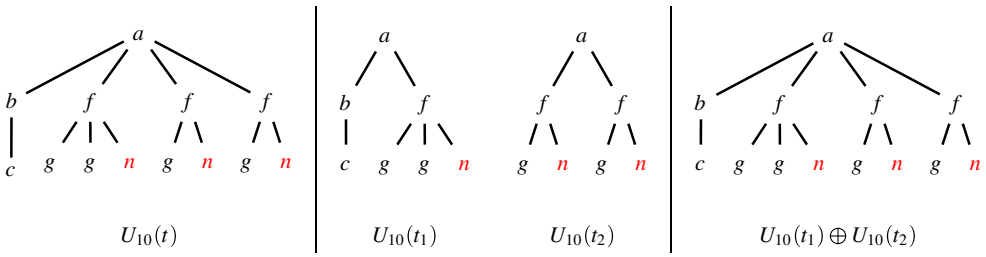


Fig. 7 Equivalence between $U_5(t)$ and $U_5(t_1) \oplus U_5(t_2)$

1. $E_u(\textit{delete } N \ P_{\text{tg}}, \Gamma, 1) = E_q(P_{\text{tg}}, \Gamma, 1)$
2. $E_u(\textit{rename } N \ P_{\text{tg}} \textit{ as } \text{a}, \Gamma, 1) = E_q(P_{\text{tg}}, \Gamma, 1)$
3. $E_u(\textit{replace } N \ P_{\text{tg}} \textit{ with } Q_{\text{s}}, \Gamma, 1) = E_q(P_{\text{tg}}, \Gamma, 1) \cup E_q(Q_{\text{s}}, \Gamma, 1)$
4. $E_u(\textit{insert } N \ Q_{\text{s}} \textit{ Pos } P_{\text{tg}}, \Gamma, 1) = E_q(Q_{\text{s}}, \Gamma, 1) \cup E_q(P_{\text{tg}}, \Gamma, 1)$
5. $E_u((U_1, U_2), \Gamma, m) = E_u(U_1, \Gamma, m) \cup E_u(U_2, \Gamma, m)$
6. $E_u(\texttt{if } Q \texttt{ then } U_1 \texttt{ else } U_2, \Gamma, m) = E_q(Q, \Gamma, 0) \cup E_u(U_1, \Gamma, 1) \cup E_u(U_2, \Gamma, 1)$
7. $E_u(\texttt{for } x \texttt{ in } Q \texttt{ return } U, \Gamma, m) = \Gamma' \cup E_u(U, \Gamma \cup \Gamma', m)$ *where* $\Gamma' = \{P\{for\ x\} \mid P \in E_q(Q, \Gamma, 0)\}$
8. $E_u(\texttt{let } x := Q \texttt{ return } U, \Gamma, m) = \Gamma' \cup E_u(U, \Gamma \cup \Gamma', m)$ *where* $\Gamma' = E_q(Q, \Gamma, 0)$

Fig. 8 Path extraction for updates

query may require a global visit of the document, which prevents iterativeness.

We now deal with identifying partitioning paths for updates. In order to specify the partitioning algorithm, as already explained, we need to identify the subtrees of the document over which some update operation operates and, therefore, that should be kept in single parts. To this end, we proceed in a way similar to queries: if the update U is iterative and elementary, we use the target paths P_{tg} of U, otherwise we use the partitioning path (Definition 6).

As an update can be a sequence of updates, the partitioning process has to consider a set of partitioning paths, as illustrated by the following example.

Example 3 Consider the update U_9.

$U_9 =$ (`for` $x `in` /a/b `return delete node` $x),
(`for` $x `in` /a/f `return rename node` $x `as` "n")

The set of partitioning paths for U_9, denoted $PP(U_9)$, is $\{P_1, P_2\}$ with $P_1 = /a/b$ and $P_2 = /a/f$.

Definition 8 (**Partitioning Paths for Updates**) The set of partitioning paths $PP(U)$ for an iterative update U is defined as follows:

- if one of the following holds:
  1. $U = delete\ N\ P_{tg}$;
  2. $U = rename\ N\ P_{tg}\ as$ a;
  3. $U = replace\ N\ P_{tg}\ with Q_s$;
  4. $U = insert\ N\ Q_s\ Pos\ P_{tg}$;

  then $PP(U) = \{P_{tg}\}$;
- if U is either a let-update or a for-update expression, then $PP(U) = \{PP\}$, where PP is the partitioning path of U according to Definition 6, where $Cand(U)$ is determined as already specified in Definition 7;
- if $U = U_1, \ldots, U_n$, then $PP(U) = \bigcup_{i=1}^{n} PP(U_i)$.

Note that the above two definitions directly provide the conditions to deal with a workload of n iterative updates $U_1, \ldots, U_n$. In this case the entire workload is iterative, and partitioning paths can be extracted just as indicated above for the sequence case.

4 Partitioning Algorithms

This section first provides a general algorithm that, given a document t and a set of partitioning paths, builds a partition of t. Roughly, the algorithm is strongly guided by the partitioning paths extracted from the input workload, as these paths return the subtrees which should not be sliced. This algorithm can be used for dealing with a workload including multiple queries and updates.

Then we investigate, for iterative queries and updates separately, additional processes that may be introduced during or after the partitioning phase, in order to further improve the efficiency of query and update evaluation.

4.1 Path alignment and residuation

This section presents two basic operations that will be used in our algorithms for data partitioning. A common aspect of these algorithms is that they rely on parsing and matching XML documents against a set of paths. To this end, these algorithms check whether a path in the parsed document matches at least one path in the input partitioning path set. This is done by means of a top-down traversing of document paths, where query/update partitioning paths need to be opportunely rewritten in order to get rid of the steps that have already been matched in the traversal, and prepare subsequent matching steps. To this end, the following two functions are introduced: the level alignment function and the residuation function.

The level alignment function $Down(SPath)$ serves in our algorithms to modify paths in $SPath$ in order to prepare the next parsing "down" moves:

$Down(SPath) = \bigcup_{P \in SPath} Down(P)$ where:

$$
\begin{aligned}
Down(self :: NT/P) &= \{/self :: NT/P\} \text{ (path already aligned)} \\
Down(child :: NT/P) &= \{/self :: NT/P\} \\
Down(dos :: NT/P) &= \{/self :: NT/P\} \cup \\
&\quad \{/self :: node()/dos :: NT/P\} \\
Down(\epsilon) &= \{\epsilon\}
\end{aligned}
$$

The residuation function $Res(\alpha, P)$ returns a path P' and a value M$\in$\{ok_t, ok_nt, fail\} respectively capturing that **i)** $\alpha \in \{\text{a}, text[\text{s}]\}$ matches the final step of the path P -terminal case-, **ii)** α matches the top step of the path with at least two

steps -non terminal case-, **iii)** none of the previous two cases holds.[9] The residual of a path P is defined by distinguishing the following cases:

$$
\begin{array}{lll}
Res(\mathrm{a},\ /self :: NT) & = <\epsilon\ ;\ \mathrm{ok\_t}> & \text{if } NT \in \{\mathrm{a}, node()\} \\
Res(\mathrm{a},\ /self :: NT/P) & = </P\ ;\ \mathrm{ok\_nt}> & \text{if } P \neq \epsilon \wedge NT \in \{\mathrm{a}, node()\} \\
Res(text[\mathrm{s}],\ /self :: NT) & = <\epsilon\ ;\ \mathrm{ok\_t}> & \text{if } NT \in \{text(), node()\} \\
Res(\alpha,\ /P) & = <\epsilon\ ;\ \mathrm{fail}> & \text{otherwise}
\end{array}
$$

The residual of a path set $SPath=\{P_1, \cdots, P_n\}$ is then defined as follows:

$$Res(\alpha,\ \tau) = <\bigcup_{i=1}^{n}\{P'_i\};\ \biguplus_{i=1}^{n}\mathrm{M}_i> \text{ with } Res(\alpha,\ P_i) = <P'_i;\ \mathrm{M}_i>$$

where the (commutative and associative) function $\uplus$ is defined by:

$\mathrm{ok\_t} \uplus - = \mathrm{ok\_t} \quad \mathrm{ok\_nt} \uplus \mathrm{fail} = \mathrm{ok\_nt} \quad \mathrm{fail} \uplus \mathrm{fail} = \mathrm{fail}$

$\mathrm{ok\_nt} \uplus \mathrm{ok\_nt} = \mathrm{ok\_nt}$, where $-$ denotes any value.

When for a given node of an XML tree, path residudation returns ok_t, we say that the node is a terminal node for the path.

To illustrate these functions, the aligned and residuated paths for 3 nodes of the tree in Fig. 9 are provided, according to the document order.

- For $P=/dos :: b$, node=l_1, $\alpha = a$, we have:
  $Down(\{P\})=\{/self :: b,\ /self :: node()/dos :: b\}$,
  $Res(a,\ Down(\{P\}))=<P;\ \mathrm{ok\_nt}>$
- For $P=/dos :: b$, node=l_2, $\alpha = b$, we have:
  $Down(\{P\})=\{/self :: b,\ /self :: node()/dos :: b\}$,
  $Res(b,\ P)=<P;\ \mathrm{ok\_t}>$
- For $P=child :: b$ node=l_7, $\alpha = f$, we have:
  $Down(\{P\})=\{/self :: b\}$, $Res(f,\ P)=<P;\ fail>$

4.2 Generic Partitioning

Algorithm 1 is described in a DOM-oriented fashion and provides a formal presentation of the general partitioning process. This algorithm is recursive and takes as input a tuple <l; *SPath*; *cSize*; *pId*> representing the current state of the recursive process. Namely, this input tuple indicates that (1) the current node to be matched against the current paths in *SPath* is l; (2) the current size of the part being built is *cSize*, denoting the number of bytes in the textual representation of the part; (3) the current number of built parts is *pId*.

[9] We assume that, if a path contains contradictory consecutive *self* steps like in $/self :: b/self :: c$., then the path is discarded from *SPath*. Concerning other non-contradictory consecutive *self* steps (like $/self :: b/self :: node()$), these are simply rewritten in a single *self* step (like $/self :: b$) by means of a simple rewriting. These operations are routinely made before computing the residual path. We also assume that paths may have a step $Axis :: text()$ only as the last step.

The function $Store\_size(\sigma)$ is used to return the size of a store σ in bytes. Its definition is obvious and avoided here.

The algorithm is initially invoked with $cSize$=0, pId=1, the location l is the root of the input XML document (σ, l), and *SPath* is the set of partitioning paths extracted from the query or update workload. We also assume that the part size threshold is known and given, in bytes, by *pSize*.

Algorithm 1 Generic Partition

Input:
  $\mathrm{l} \in dom(\sigma)$, /* location */
  SPath, /* partitioning path set */
  cSize, /* current part size */
  pId /* current part number */
Output:
  σ^{P}, /* a store gathering built parts */
  $cSize'$, /* part size */
  pId' /* new current part number */

begin
  let $\sigma(\mathrm{l}) = \mathrm{a}[\mathrm{L}]$ *and* $\mathrm{L} = (\mathrm{l}_1, \mathrm{l}_2, \cdots, \mathrm{l}_n)$
  /* *Case 1.* location l is a *SPath* target node */
1 **if** $Res(\mathrm{a},\ SPath) = <-; \mathrm{ok\_t}>$ **then**
2 $\sigma^{\mathrm{P}} := RenameStore(\sigma@\mathrm{l},\ pId)$;
3 $cSize' := cSize + Store\_size(\sigma^{\mathrm{P}})$
  /* *Case 2.* location l is a partial matching node wrt *SPath* or fails to answer *SPath* */
4 **else**
5 $pId_{\mathrm{first}} := pId$; $\sigma^{\mathrm{P}} := \emptyset$;
6 **for** $i = 1 \ldots n$ **do**
7 $(\sigma^{\mathrm{P}}_{\mathrm{i}},\ cSize,\ pId) :=$
8 $Partition(\mathrm{l}_i,\ Down(SPath),\ cSize,\ pId)$;
9 $\sigma^{\mathrm{P}} := \sigma^{\mathrm{P}} \cup \sigma^{\mathrm{P}}_{\mathrm{i}}$
10 $pId_{\mathrm{last}} := Max\text{-}Pid(\sigma^{\mathrm{P}})$; $D := dom(\sigma^{\mathrm{P}})$
  /* *Max-Pid()* returns the greatest part number used in the store */
11 **for** $i = pId_{\mathrm{first}} \ldots pId_{\mathrm{last}}$ **do**
12 $\sigma^{\mathrm{P}} := \sigma^{\mathrm{P}} \cup \{(\mathrm{l}^i \leftarrow a[rename\_extr(\mathrm{L}, i, D)])\}$
13 $cSize' := cSize + 2.length(\mathrm{a})$
14 **if** $cSize' \leq pSize$ **then**
15 $pId' := pId$ /* there is still space in the current part */
16 **else**
17 $cSize' := 0$; $pId' := pId + 1$ /* a new part must be created */
18 **return** $(\sigma^{\mathrm{P}},\ cSize',\ pId')$

In Algorithm 1, the function $RenameStore(\sigma,\ pId)$ produces a new store from σ by renaming each location l to l^{pId}. This renaming is used (i) to avoid collisions of locations (any two distinct parts of the partition σ^{P} are disjoint in terms of locations) that could arise when the sub-tree rooted at an element node bound to l is split during partitioning, and at the same time (ii) to keep track of the original location (l^{pId} is the copy of the input document node l) which is needed for the fusion step.

Thus, the algorithm *Partition* builds a store which contains a partition of the store σ and whose parts have a size determined by the threshold *pSize*.

The algorithm distinguishes two main cases:

- In the first case (lines 1-3), the current node l is the target of some path p in *SPath*, that is, l is an answer of the path query p. In this case, the subtree rooted at node l, which is denoted σ@l, should be entirely kept in the same part and indeed in the current part being built. The subtree σ@l is simply copied in σ^{P} while renaming node labels with the current part number (line 2). Then (line 3), the new size *cSize*$'$ of the current part is calculated in the obvious manner.
- In the second case (lines 4-13), the current node l either is a partial match wrt *SPath* meaning that there exists a path p in *SPath* and a prefix pp of p such that the node l is a target of pp, or l fails to answer any (prefix of) path in *SPath*. In both cases, the computation iterates over the children of the node l; for each child l_i, the algorithm first aligns the paths in *SPath* to the new parsed tree level, and, then, recursively partitions l_i's subtrees (line 7-8). As a result, multiple parts may be built in σ^P.

  Then (lines 11-12), the partition σ^P is enriched by linking the node l as the root of each new parts generated by the previous phase. This process requires to copy the node l with the adequate use of location indexes. To this end, the function *rename_extr*($\mathtt{L}$, i, D) takes as input the children location sequence $\mathtt{L}$, a part identifier i, and the domain $D = dom(\sigma^P)$ of the recently created sub-partition. The function *rename_extr*($\mathtt{L}$, i, D) extracts the sub-sequence of $\mathtt{L}$ used to create part i in σ^P, and adorns each location of this sub-sequence with p. More formally, we have:

$$
\begin{array}{ll}
\textit{rename\_extr}(\mathtt{L}, p, D) = & \\
() & \text{if } \mathtt{L} = () \\
\mathrm{l}_i^p, \textit{rename\_extr}(\mathtt{L}', p, D) & \text{if } \mathtt{L} = \mathrm{l}_i, \mathtt{L}' \text{ and } \mathrm{l}_i^p \in D \\
\textit{rename\_extr}(\mathtt{L}', p, D) & \text{if } \mathtt{L} = \mathrm{l}_i, \mathtt{L}' \text{ and } \mathrm{l}_i^p \notin D
\end{array}
$$

  After the recursive calls on child location l_i have been completed, and the partitioning store is built adequately, the new size *cSize*$'$ of the current part is calculated (line 13).
- In both of the above cases, the partitioning call ends (lines 14-17) by preparing the next step wrt part completion and thus checking whether the size of the current part exceeds the threshold *cSize*: if this is the case, a new empty part is prepared by resetting the current size *cSize* to 0 and by incrementing the current part number *pId*.

Example 4 Assume that the input partitioning path *PP* is *dos* :: *b*. Let us assume that the part size threshold *pSize* is 11 bytes, and that each single character takes a byte (recall that for each element tag we need to take intro account both start and end tags). Figure 9 displays a document t and the output of applying Algorithm 1 on t. Note that the whole document t could be easily rebuilt from the three parts of the partition.

During the partition process, we first add the subtree rooted at l_2 to the first part because this node is a partition path terminal node. This makes the *cSize* exceed the threshold *pSize* and thus a new part is initialized. Notice that l_7 is not matching the partition path, thus its subtrees may be spread over several parts, which is the case here. After scanning node l_{11}, the second part size exceeds the threshold *pSize* and thus a new part is started. The third part is first filled with the third subtree of l_7 and then, as the node l_{14} is a partition path terminal node, the whole subtree rooted at l_{14} is included in the third part. Note that, while building this subtree, the threshold has been largely exceeded. However, since this subtree is selected by a partitioning path, it cannot be split. The process ends up with three parts, as indicated in Fig. 9.

Of course, the part size threshold *pSize* plays a key role in the whole process. It depends on many factors, such as the input document, the query or update 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

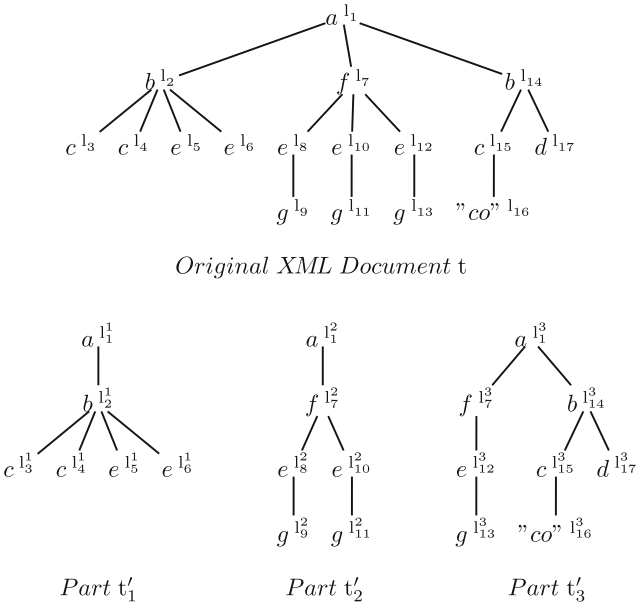


Fig. 9 Generic partition

system running on the hardware. *pSize*, therefore, can be determined only through a *trial-and-error* process depending on the overall configuration, and cannot be formally predicted.

Note that if *pSize* is too large, it can happen that one or more parts are too large to be loaded in main memory, hence undermining the whole approach. A good trade-off is a value that is close (but not too close) to the maximal document size supported by the query engine being used. In our experimental analysis (Section 6) we made this choice for *pSize*.

4.3 Partitioning wrt iterative queries

If we know that partitioning is used for supporting the evaluation of iterative queries only, then we can improve the generic partitioning algorithm (Algorithm 1) in a very simple manner by splitting case 2 into two different cases:

- case 2a corresponds to the current node l being a partial match wrt *SPath* and remains unchanged. This subcase arises when $Res(\text{a}SPath) = <SPath'; \text{ok\_nt}>$.
- case 2b corresponds to the current node failing to answer any path in *SPath*, and then, for iterative queries, we know for sure that the subtree rooted at location l is not relevant for query evaluation and can be ignored. In that case, an empty part set σ^P of size $cSize' = 0$ is then returned. This sub case arises when $Res(\text{a}SPath) = <SPath'; \text{fail}>$.

Algorithm 2 outlines the partitioning algorithm for iterative queries and is displayed by a straightforward reuse of Algorithm 1.

Algorithm 2 Partition for Iterative Queries

```
Input:  /* same as Alg. 1                                   */
Output: /* same as Alg. 1                                   */

begin
    lines 1-3 of Alg. Partition
    else
        if Res(a, SPath) = <SPath'; ok_nt> then
            /* Case 2a.  location l is a partial
               matching node wrt SPath                     */
            lines 5 - 13 of Alg. 1
        else
            /* Case 2b.  location l fails to answer
               SPath                                       */
            σ^P := ∅ ; cSize' := cSize
    lines 14 to 18 of Alg. 1
```

Example 5 Let us continue with the previous example and now assume that the partitioning path $PP = /dos :: b$ has been extracted from the query Q below:

$Q = \texttt{for}\ x\ \texttt{in}\ dos :: b\texttt{return}(x/child :: c,\ x/child :: d).$

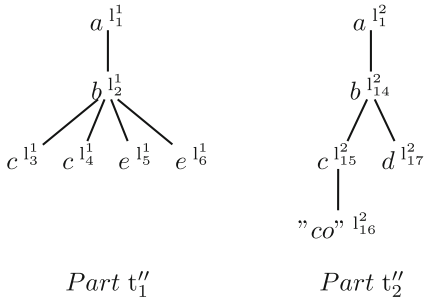


Fig. 10 Query partition

Figure 10 displays the result of applying the query partitioning algorithm on the document t. Note that this time the initial document t in Fig. 9 is not entirely retrieved in the generated parts of the new partition.

The partition process starts as in the generic case. The main difference arises when, starting the second part, the node l_7 is processed. Recall that this node is not matching the partitioning path. Thus, its subtree is not included in the partition at all because it is known in advance that it contains no relevant data for the query Q.

4.4 Partition and Projection for Iterative Queries

Projection is a well known technique allowing one to increase XML query evaluation efficiency both in terms of memory consumption and time evaluation (Marian and Siméon 2003; Baazizi et al. 2011; Benzaken et al. 2006). In our setting, projection can be performed either at partitioning time (projection is then merged in the partition algorithm) or after partitioning and then applied over each part of the partition. In Bidoit et al. (2012), the first approach has been adopted because developed in a centralized setting (no distribution). Here, we choose to perform projection after partitioning. This ensures a faster (centralized) partitioning process plus the possibility to project parts in parallel during the map phase.

Projection is made wrt paths extracted from the iterative queries.

Definition 9 (**Query projector**) Given an XML query Q, the *projector* τ of Q is the set $\tau = \{ErV(P) \mid P \in E_q(Q)\}$.

In the general case, several paths are used during projection: each path is handled as indicated in the example, through the path rewriting functions $Down(\tau)$ and $Res(\alpha, \tau)$ defined in Section 2.

Differently from Marian and Siméon (2003), we provide here a formal specification of the projection algorithm.

Algorithm 3 presents the code of the *Projection* algorithm in a DOM-oriented fashion. It takes as input a store σ, a current location l, and a projector τ. It outputs the projection σ' of the tree $\sigma@l$ rooted at l wrt the projector τ.

Note that $\mathtt{L}_{|\pi}$ denotes the location sequence obtained from L by keeping only locations in π while preserving the π ordering (we have $\mathtt{L}_{|\pi} \preceq \mathtt{L}$).

The algorithm is organized wrt two main cases.

- When the current node location l contains a text node (lines 1-4), if residuation does not fail, then for at least one path in the projector the last step matches the node l (recall that only the final step in a path can use the text node condition).
- When the current location, instead, contains an element node (lines 5-18), a more complex analysis is necessary. If residuation fails (lines 6-7), then the output is the empty store. If the current node is an intermediate match for the current projector, and the node has no child (lines 8-9) then the node is added to the projection; this is necessary because this node can be later on matched as a terminal node after residuation of the projector, during the recursive process.[10] Otherwise, (lines 12-13) projection is recursively propagated on children nodes. Then (lines 14-16), if the current element node is a terminal match for the projector, this node is added to the projection together with its projected subtrees; if the current element matches an intermediate step of a path in the projector, then the node will be added to the projection if at least one of its descendant will match a final step in the projector. If none of the above conditions holds (lines 17-18), the empty projection is output.

Projection is designed in such a way that, given a query Q, its projector τ and an XML tree $t{=}(\sigma, \mathrm{l}_t)$, assuming that $\mathit{Projection}(\sigma,\ \mathrm{l}_t,\ \mathit{Down}(\tau)){=}(\sigma',\ \mathrm{l}_t){=}t'$, we have:

$$Q(t) \equiv Q(t').$$

In Section 5 we will provide some detail about our SAX-based streaming implementation, which has a negligible memory footprint.

The following example illustrates how projection works and, although very simple, it gives a flavor of the memory optimization.

Example 6 Let us go back to our running example and project the parts, displayed in Fig. 10, produced by the partitioning algorithm for the iterative query Q. The projector for this query is the set of paths $\tau = \{/dos :: b,\ /dos :: b/c/dos :: node(),\ /dos :: b/d/dos :: node()\}$. Let us consider the part t_1'' of Fig. 10. Its projection wrt τ is preceded by a level alignment step, leading to consider the set of paths:

$\tau_1 = \{/self :: b,\ /self :: b/c/dos :: node(),\ /self :: b/d/dos :: node()\} \cup$ [1]

$\{/self :: node()/dos :: b,\ /self :: node()/dos :: b/c/dos :: node(),$ [2]

$/self :: node()/dos :: b/d/dos :: node()\}$.[3]

Algorithm 3 Projection

```
Input:
    σ,          /* a store */
    l∈dom(σ),   /* a location */
    τ           /* a projector */
Output:
    σ'          /* a projected store */

begin
    /* Case 1.  σ(l) = text[s]  ..............  */
1   if Res(text[s], τ) = <−;fail> then
2       σ' := ∅;
3   else
4       σ' := {l ← text[s]};
    /* Case 2.  σ(l) = a[L]  ..................  */
5   <τ';M> := Res(a, τ);
6   if M = fail then
7       σ' := ∅;
8   else if M = ok_nt and L = () then
9       σ' := {l ← a[()]};
10  else
11      let L = (l1, l2, ··· , ln)
12      for i = 1...n do
13          σi := Projection(σ, li, Down(τ'))
14      π := {li∈L | σi ≠ ∅}
15      if (M = ok_t) or (M = ok_nt and π ≠ ∅) then
16          σ' := {l ← a[L|π]} ∪ ⋃_{i=1}^{n} σi;
17      else
18          σ' := ∅;
19  return (σ')
```

We can then check that the node l_1^1 matches the first step of any simple path of lines [2] and [3]. As a side effect, these paths are rewritten into the *residual paths* (which produces τ again) in order to prepare the next tree level examination. Before analyzing the node l_2^1, a new alignment operation is performed which leads to produce τ_1 again. The node l_2^1 matches the first step of any simple path of lines [1], [2], and [3]. The residual paths obtained from τ_1 are:

$$Res_{\tau_1} = \{\epsilon,\ \ /c/dos :: node(),\ \ /d/dos :: node()\} \cup \tau$$

[10] For instance, consider a projector including $/a/b/self :: node()$ and a tree where the root a has an empty b element as child.

Once again, the examination of the children of the node l_2^1, that is nodes l_3^1, l_4^1, l_5^1, and l_6^1, is prepared by the level alignment of Res_{τ_1} which contains all the path in τ_1 plus the following paths: ϵ, $self :: c/dos :: node()$, $self :: d/dos :: node()$.

It is then obvious to see that node l_3^1 matches the first step of the path $self :: c/dos :: node()$. Residuation and alignment leads to the simple paths ϵ, $self :: node()/dos :: node()$. Because node l_3^1 has no children, the empty path is matched, hence leading to the projection of the concrete path $l_1^1 l_2^1 l_3^1$.

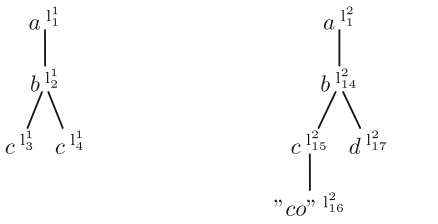
The same arises for node l_4^1 although path matching will fail for the siblings l_5^1 and l_6^1. The projection of t_1'' is given in Fig. 11 as well as the projection of t_2''.

As it can be seen in Fig. 10, projection entails size reduction of parts, which, in turn, entails optimisation in terms of time, as the amount of data to be processed by the query processor is decreased. As we will see next, in the case of updates we do not adopt projection as this would make the fusion process very complex and inefficient, due to complex and time consuming operations that would be needed to recover/identify pruned subtrees, and to put them in the final result.

4.5 Partitioning wrt iterative update

The generic partitioning algorithm (Algorithm 1) can also be customized for anticipating the evaluation of iterative updates. Clearly, for iterative updates, it is forbidden to discard or project some of the parts. However, the partitioning process may provide information about which parts may be potentially modified by the update and which part will remain unchanged. Thus the idea is to mark the parts of the partition in the scope of the update (Case 1 of Algorithm 1). This will be used at update evaluation time to skip unmarked parts.

Algorithm 4 outlines the partitioning algorithm for iterative updates and is displayed by a straightforward reuse of Algorithm 1. Notice that the input and output of the algorithm are enriched by a list of part identifiers collecting the marked parts. The initial call to Algorithm 4 will be done with an empty $List_{pId}$ list.



Projection of Part t_1'' Projection of Part t_2''

Fig. 11 Projection after partitioning for iterative queries

Algorithm 4 Partition for Iterative Updates

```
Input: /* same as Alg. 1 + List_pId a list of part
   identifiers                                          */
Output:  /* same as Alg. 1 + List_pId' * a new list
   of part identifiers                                  */
begin
   /* Case 1. of Alg. 1: location l is a SPath
      target node                                       */
   if Res(a, SPath) = <−;ok_t> then
      lines 2-3 of Alg. 1
      if pId ∉ List_pId then
         List_pId' := List_pId, pId /* Current part is
            marked                                      */
   lines 4 - 18 of Alg. 1
```

Example 7 Consider the following update: U =`for $x in /dos :: b return insert node <n/> as last into $x`, whose partitioning path is still PP=$/dos :: b$.

Then, the partitioning Algorithm 4 proceeds as the generic one and produces the parts displayed in Fig. 9 and the list $List_{pId}'$=[1, 3] as the second part contains a portion of the document t on which the update has no impact at all: $U(t_2') = t_2'$. At execution time, the update U will not be performed over t_2'. This part is only used for building the final updated document from updated parts.

The next section provides the details of how the final query or update result is composed from the evaluation of the query or update over the parts of the partition.

4.6 Result Combination

Result combination for iterative queries is straightforward. After document partitioning, part projection and query evaluation over each part providing partial results, the final result of the query for the input document is built by concatenation of the partial results. This applies immediately to iterative query workload as follows.

Let $pSize$ be the part size threshold value, let $Q_1, \ldots, Q_m$ be iterative queries with their resp. partitioning path PP_j and projector τ_j. Assume that $SPath$=$\cup_1^m\{PP_j\}$ and τ=$\cup_1^m\tau_j$. Let us consider an XML document t=(σ, l_t). Then:

$$Q_j(t) \equiv Q_j(\tau(t_1)) \cdot \ldots \cdot Q_j(\tau(t_{pId}))$$

where $(t_1, \ldots, t_{pId})$ is the output of the partitioning algorithm for iterative queries (Algorithm 2) run with, as input, the location l_t, the partitioning path set $Down(SPath)$, the current part size 0 and current part number 1.

The last step of our partitioning update scenario, as illustrated in the Section 2.1, relies on a fusion operation. This operation takes as input the set of updated parts $U(t_i)$ and

returns $U(t)$. A particular issue in the fusion process concerns the copies of a same location in distinct parts. For our running example, copies of location are:

- l_1^1, l_1^2 and l_1^3 are three copies of the root, and
- l_7^2, l_7^3 are also copies of the same node l_7.

Fusion, denoted by $\oplus$, has to be carefully specified in order to ensure that copied locations collapse to a unique location, as illustrated in Fig. 12, where the final update result $U(t_1') \oplus t_2' \oplus U(t_3')$ contains no copy of the root l_1 nor of the node l_7.

Indeed, in Fig. 12 and next, locations $\mathrm{l}_\perp$ are intended to capture the new nodes created by the update.

Fusion uses the two following functions:

- Given an indexed location l_i^j, the function $ErInd(\mathrm{l}_i^j)$ removes the index j: $ErInd(\mathrm{l}_i^j) = \mathrm{l}_i$ and $ErInd(\mathrm{l}_\perp) = \mathrm{l}$.
- Assuming that $C = \{\mathrm{t}_1, \cdots, \mathrm{t}_\kappa\}$ is a collection of trees, the function $F(\mathrm{l}_i, C)$ takes all copies of the subtrees in C whose root l_i^j is a copy of l_i and pull them together again by removing their part numbers. Thus:

  $F(\mathrm{l}_i, C) = \{\mathrm{l}_i \leftarrow \mathrm{a}[\mathtt{L}]\}$ where $\mathtt{L} = ErInd(\mathtt{L}_n) \cdot ErInd(\mathtt{L}_{n+1}) \cdot \ldots \cdot ErInd(\mathtt{L}_m)$ and $\mathrm{l}_i^j \leftarrow \mathrm{a}[\mathtt{L}_\mathrm{j}] \in \mathrm{t}_\mathrm{j}$ for $j = n \ldots m$, and for some n and m with $1 \leq n \leq m \leq \kappa$.

For instance, if C is the collection of the three updated parts of Fig. 12, and if we consider the node l_7, then $F(\mathrm{l}_7, C)$ simply builds the single subtree rooted at node l_7 in the final result given in Fig. 12.

Fusion is presented below in a very general setting although it should be understood that it is intended to be applied on the collection C of updated parts, where the parts are generated by the partition algorithm.

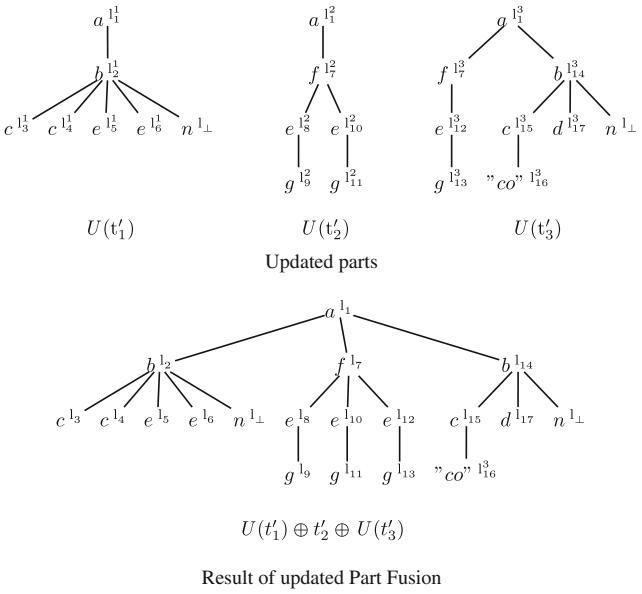


Fig. 12 Fusion scenario on distinct (updated and non-updated) parts

Definition 10 (Fusion $\oplus$) Let $C = \{\mathrm{t}_1, \mathrm{t}_2, \cdots, \mathrm{t}_\kappa\}$ be a collection of trees with $\mathrm{t}_\mathrm{j} = (\sigma^\mathrm{j}, \mathrm{l}_0^j)$, where the root node l_0^j is a copy of some node l_0, and, moreover, assume that any node in σ^j is indexed by j. Let $D = dom(C) - New(C)$ where $New(C)$ collects locations of the form $\mathrm{l}_\perp$ in C. The fusion $\oplus(C)$ is the tree (σ, l_0) such that:

$$\sigma = \bigcup_{\mathrm{l}_i \in D} F(\mathrm{l}_i, C) \cup \{\mathrm{l} \leftarrow \mathrm{a}[ErInd(\mathtt{L})] \mid \mathrm{l}_\perp \in New(C)\}$$

We are now ready to put together the 3 steps of our partitioning scenario for iterative updates and we do so for the general case of an update workload. Without loss of generality, we consider a workload with two updates only.

Let *pSize* be the part size threshold value, let U_1, U_2 be iterative updates with their resp. partitioning path sets $SPath_j$. Let $SPath{=}\cup_1^2\{SPath_j\}$. Let us consider the XML tree $t{=}(\sigma, \mathrm{l}_t)$. Then we have:

$$U_2(U_1(t)) \equiv \oplus(\cup_{i \in List'} U_2(U_1(t_i)) \quad \cup_{i \notin List'} t_i).$$

where $(t_1, \ldots, t_{pId})$ and the list of part indexes $List'$ are output by the partitioning algorithm for iterative updates (Algorithm 4) run with, as input, the location l_t, the partitioning path set $Down(SPath)$, the current part size 0, the current part number 1 and the empty list $List_{pId}$ of part identifiers.

Recall that $List'$ captures the document parts potential targets of the updates U_1 or U_2 and that, above, at combination time, the untouched parts are captured by $i \notin List'$.

5 Implementation Issues

While Andromeda is not a full-fledged database management system, it is still a complex system and its implementation poses several challenges. In the following we will focus on the most prominent issues we faced during the development and implementation of Andromeda.

5.1 Partitioning

In Section 4 we described a general-purpose partitioning algorithm as well as two optimized versions specifically designed for query-only workloads and for updates. These algorithms were specified by assuming a DOM-like model based on locations, and, given an input document, they return a set of parts, each one being a well-formed document; if Algorithms 1 and 4 are used, these parts can be later on merged to obtain a new XML document.

The use of this model, however, while easing the formal development, is not realistic when dealing with large XML documents, as it would require the system to load the entire XML document in main memory before partitioning. Therefore, in our system we implemented our partitioning algorithm (in particular, Algorithm 1) in a streaming fashion by relying on the Java SAX parser.

Preliminaries and data structures Our partitioning algorithms and, in particular, Algorithm 1, exploit a representation of the input document as a store of locations, and partition the document recursively. In our actual implementation, based on SAX, our code receives as input a stream of events and *must* process them in a non-recursive fashion. To this end, as usual, our actual implementation of the partitioning algorithm is based on the use of a *double-ended* queue[11] `statusStack`, which records the status of the algorithm while processing each element of the document. `statusStack` contains `status` objects, each one describing the tag of the element being processed (a string), its attributes (a SAX *AttList* type), as well as the match value (a `MatchValue` object) and the set of partitioning paths after residuation (an array-list); these object fields correspond to the parameters of the recursive call to *Partition* in line 8 of Algorithm 1 (*cSize* is not needed to be stored as only one part can be open at a time, unlike what happens in Algorithm 1). The implementation, in the form of a SAX handler, also exploits several global variables, such as `currentMatch`, which describes the current match value, and `partSize`, which indicates the size of the part currently open.

These data structures allow the system to correctly split the input document in multiple parts, but cannot help the system in understanding when an element has been repeated in multiple partitions: indeed, as shown in the previous section, there are many elements that are shared among multiple consecutive parts. To keep track of these elements and ease the fusion of parts, as a consequence of an update operation, the system marks these elements, during the partitioning phase, with unique identifier, stored in a special-purpose XML attribute `__artID`, and called *artificial ID*; repeated elements, hence, are deemed as *artificial*.

Artificial IDs are used as keys for accessing a hash map `artSet` that associates to each artificial element the last part in which the element has been closed.

SAX handler Our partitioning SAX handler takes as input an XML document and a list of partitioning paths `pathList`. The SAX parser creates a stream of SAX *events*, generated by performing a DFS of the XML tree. In particular, five major kinds of events are generated: `startDocument`, when the header of the XML document has been read; `endDocument`, when the parser reaches the end of the document; `startElement` and `endElement`, when a tag is open or closed, respectively; and `characters`, when the textual content of an element is read.

Before reading the SAX stream, the handler creates and initializes all data structures: in particular, `statusStack` is created empty and `currentMatch` is set to ok_nt.

When processing a `startElement` event, the partitioner first inspects `statusStack`: if `statusStack` is empty, hence meaning that the element being processed has no father in the XML tree (i.e., it is just the root), then the partitioner sets `currentMatch` to ok_nt; otherwise, it peeks the top of `statusStack`, hence retrieving the `status` object generated by its father element, hence restoring the status of the algorithm just after the current event. In particular, it sets `pathList` to `status.pathList` and `currentMatch` to `status.currentMatch`. In both cases, the partitioner residuates `pathList`.

If `currentMatch` = ok_t, then the father element was part of an *indivisible* tree, i.e., a tree that cannot be split, and, hence, the current element must belong to the same tree; therefore, the `startElement` event is passed to the output serializer, `partSize` is increased by the number of bytes needed to store, in their textual form, the opening and closing tags of the current element, and a new `status` object is pushed on top of `statusStack`.

If `currentMatch` = fail, then the father element fails in matching the partitioning paths, which implies that the current element too cannot match `pathList`. Therefore, the system forwards the `startElement` event to the output serializer, updates `partSize`, and pushes a new `status` object on top of `statusStack`.

If `currentMatch` = ok_nt, then we have two subcases, that are related to the match value `value` generated as result of `pathList` residuation. If `value` = ok_t, then the partitioner has reached a *switch node*, i.e., a node on which `currentMatch` changes its value, and proceeds as for `currentMatch` = ok_t. If `value` = ok_nt or `value` = fail, then the handler marks the current element as (possibly) artificial by associating to it a unique artificial identifier, sends the event to the output serializer, updates `partSize`, and, finally, pushes a new `status` object on the top of `statusStack`.

When processing an `endElement` event, the system first pops the `status` object on top of `statusStack`, storing it in a variable `currentStatus`, and, then,

[11] A flexible Java equivalent of a stack.

peeks the object on top of statusStack, storing it in a variable fatherStatus; currentStatus and fatherStatus describe the status of the algorithm after the startElement events for the element currently being closed and its father element have been processed, respectively.

Afterwards, the system forwards the endElement event to the output serializer and inspects both currentStatus and fatherStatus. If fatherStatus.currentMatch = ok_nt and currentStatus.currentMatch = ok_t or fail, then the system checks whether partSize > *pSize* (the maximal part size); if the check is not successful, then the system does nothing. Otherwise, it first looks if the element currently being closed is artificial, and, in the positive case, the handler maps its artificial ID to the current part number in artSet; then, the system closes the current part and creates a new one, by iterating on statusStack and closing/opening repeated elements as needed.

If both fatherStatus.currentMatch and currentStatus. currentMatch are equal to fail, then the system works as in the previous case. No other case is possible.

Example 8 Consider again the XML document of Fig. 9 and, as in Example 4, assume that pathList = {*dos* :: *b*} and *pSize* = 11. The textual representation of this document is shown below.

```
< a >
    < b >
        < c > < /c >
        < c > < /c >
        < e > < /e >
        < e > < /e >
    < /b >
    < f >
        < e >
            < g > < /g >
        < /e >
        < e >
            < g > < /g >
        < /e >
        < e >
            < g > < /g >
        < /e >
    < /f >
    < b >
        < c > Co < /c >
        < d > < /d >
    < /b >
< /a >
```

Each open tag generates a corresponding startElement event, while a closed tag raises an endElement event. When the startElement event corresponding to <a> is generated, statusStack is empty, pathList = {*dos* :: *b*}, and currentMatch is undefined. As statusStack is empty, the handler sets currentMatch to ok_nt and residuates pathList. As shown in Section 4.1, the residuation of {*dos* :: *b*} returns <{*dos* :: *b*}; ok_nt>; hence, the handler marks the current element as possibly artificial, by adding an attribute __artID = '1' to <a>, sets partSize to 7, creates a status object <a; ok_nt; {*dos* :: *b*}>, and pushes status on top of statusStack. The event, enriched with the newly created attribute for <a>, is finally passed to the output serializer.

After this event, the handler processes the startElement event for . Since statusStack is not empty, it peeks the top of statusStack, retrieving the object <a; ok_nt; {*dos* :: *b*}>, sets pathList to {*dos* :: *b*} and currentMatch to ok_nt. Again, as currentMatch = ok_nt, the handler residuates {*dos* :: *b*} and inspects the results. In this case, as previously shown in Section 4.1, the residuation returns <{*dos* :: *b*}; ok_t>; hence, the event is passed to the output serializer, partSize is set to 7 + 7 = 14, and the object <b; ok_t; {*dos* :: *b*}> is pushed on top of statusStack.

The handler, then, processes the startElement event corresponding to <c>. By peeking the top of statusStack, currentMatch is set to ok_t and pathList to {*dos* :: *b*}; this implies that the handler has to update partSize, to forward the event to the serializer, and to push the object <c; ok_t; {*dos* :: *b*}> on top of statusStack without residuating pathList. Immediately after this event, the handler receives an endElement event for </c>, that is processed by processed by popping the previously pushed object <c; ok_t; {*dos* :: *b*}>, by peeking the object <b; ok_t; {*dos* :: *b*}>, and, since the the match values for these objects are both equal to ok_t being closed is not artificial, by sending the event to the output serializer.

The events generated by the remaining elements nested inside and are processed in the same way. When, finally, the endElement event for is issued, the handler pops the object status = <b; ok_t; {*dos* :: *b*}> and peeks the object fatherStatus = <a; ok_nt; {*dos* :: *b*}>. Since fatherStatus.currentMatch = ok_nt and status. currentMatch = ok_t, the handler looks at partSize; as partSize = 42 > 11, the handler closes the current part and creates a new one, having <a __artID = '1'> as the root opening tag.

At the end of the partitioning process, the handler creates the following three parts.

PART 1
```
< a __artID = '1' >
  < b >
    < c > < /c >
    < c > < /c >
    < e > < /e >
    < e > < /e >
  < /b >
< /a >
```
PART 2
```
< a __artID = '1' >
  < f __artID = '2' >
    < e __artID = '3' >
      < g __artID = '4' >< /g >
    < /e >
    < e __artID = '5' >
      < g __artID = '6' >< /g >
    < /e >
  < /f >
< /a >
```
PART 3
```
< a __artID = '1' >
  < f __artID = '2' >
    < e __artID = '7' >
      < g __artID = '8' >< /g >
    < /e >
  < /f >
  < b >
    < c > Co < /c >
    < d > < /d >
  < /b >
< /a >
```

The SAX handler also returns the `artSet` hash map described in Table 1.

5.2 Partition catalog

Andromeda processes queries and updates by distributing the computing load among the machines of a MapReduce cluster. Given the hierarchical nature of textual XML documents, document parsing and partitioning is an inherently sequential activity that cannot be easily distributed among multiple machines (see Sonar and Ali (2015) for a brief analysis of the problem). Therefore, it is important to maintain a catalog of existing partitions and to have some form of *partition reuse*.

Table 1 `artSet` hash map

ID	Part Number
1	3
2	2
3	2
4	2
5	2
6	2
7	3
8	3

In Andromeda the partition catalog is a persistent hash map, implemented by relying on the persistence services offered by MapDB (2015) and stored in the local file system of the master machine, that associates to each document a collection of `Partition` objects as well as a DataGuide (Goldman and Widom 1997) summarizing its structure. Each `Partition` object contains a list of part URIs, that denote the location of parts inside the distributed file system, as well as a list of *useful* parts, i.e., parts that contain indivisible trees. The `Partition` object also stores the `artSet` hash map.

While a distinct `Partition` object is created for each new partition during the partitioning phase, the DataGuide associated to a document is created once for all. Indeed, during the very first partitioning of a document $\mathcal{D}$, the system infers a DataGuide $\mathcal{D}_\mathcal{G}$; after parts have been created by applying the partitioning algorithm of Section 5.1, the system re-evaluates the partitioning paths on $\mathcal{D}_\mathcal{G}$ and marks the nodes matching these paths, describing the switch nodes of the previous section, with a partition unique identifier. When a new query or update is submitted, the system evaluates its partitioning paths `pathList` on $\mathcal{D}_\mathcal{G}$; if there exists a partition identifier j such that all matching nodes of `pathList` are descendant of or equal to nodes marked with j, then the partition identified by j can be reused for processing the input query/update. If no compatible partition can be found, then the system re-partitions the document according to `pathList`, creates a `Partition` object and a partition identifier j' for the new partition, and marks the matching nodes of `pathList` in $\mathcal{D}_\mathcal{G}$ with j'.

5.3 Compression

Document partitioning requires the system to read an input document from the distributed file system and to write parts in the DFS; furthermore, as described in the Introduction, the system processes queries and updates by reading parts deployed on HDFS and storing intermediate results or updates parts in the DFS.

All these I/O activities generate a huge traffic on the network, which may significantly slow down the system. To limit the performance penalty induced by network flooding, Andromeda stores parts as compressed EXI files (Schneider et al. 2014) rather than plain XML files; EXI, which stands for *Efficient XML Interchange*, is a binary compressed

storage format for XML documents proposed by the World Wide Web Consortium, that can nearly double the I/O bandwidth wrt textual XML (Snyder 2010). In particular, we rely on Exificient (2015), a Siemens open-source implementation of EXI, which allows the system to directly generate compressed files by using a SAX API.

To avoid unnecessary decompression activities and to leverage on EXIficient ability to create SAX streams from compressed files, we also extended Qizx-open (2013) so that it can directly take as input compressed XML files stored in HDFS.

5.4 Result fusion

In Section 4.6 we described the algorithm used by the RESULT COMBINER (see Fig. 1) to merge updated parts and create a new document as output of a XUF update. Similarly to what happens for partitioning, to implement this algorithm we faced two major challenges: first of all, in Section 4.6 the fusion process was illustrated by implicitly assuming a DOM-like representation of XML trees, which, unfortunately, cannot be used when processing large documents. Second, our partitioning technique is based on the use of artificial elements to make each part a well-formed XML document; artificial element tags, while transparent to query result combination, must be properly managed to guarantee the soundness of update evaluation.

To deal with these issues, the result fusion algorithm of Section 4.6 has been implemented as a SAX handler that processes compressed document parts one at a time. This handler exploits two main data structures: an hash set `artificials`, that is used to record artificial tags, i.e., the tags of artificial elements, that have already been opened; and a stack `artificialStack` that keeps track of the structure of the document.

When processing a `startElement` event, the system first checks if the tag `lname` being opened is artificial by inspecting its attributes. If `lname` is not artificial, then the event is forwarded to the output serializer and the tag, together with its attributes, is pushed on `artificialStack`. Instead, if `lname` is artificial, the system extracts its artificial ID `artID` and looks for (`lname`, `artID`) in `artificials`: if the check is successful, then the artificial tag has already been opened, and the system just pushes it on `artificialStack`; otherwise, the system removes the `__artID` attribute from the `lname` element, forwards the `startElement` event to the output serializer, adds (`lname`, `artID`) to `artificials`, and, finally, pushes the element `artificialStack`.

When processing an `endElement` event, the system removes the tag `lname` being closed from the top of `artificialStack` and checks if it is artificial. If `lname` is not artificial, then the event is forwarded to the ouput serializer. If `lname` is artificial, instead, the system uses its artificial ID to retrieve from `artSet` the identifier of the part in which the tag must be closed: if this identifier is equal to that of the current part, then the event is pushed to the output serializer.

5.5 Workload processing

As shown in Sections 4.2 and 5.1, our partitioner takes as input an XML document as well as a list of partitioning paths. This is motivated by the fact that path residuation can generate multiple paths from a single path, hence our partitioner must be able to cope with this situation.

This requirement for the partitioner made easier the support for multiple concurrent queries evaluated on the same document. Indeed, to evaluate a workload W of multiple queries, our system just collects the partitioning paths of each query in W, and passes the path list to the partitioner. During query evaluation, each mapper evaluates all queries on its assigned parts, and stores query results in a different HDFS directory for each query.

6 Experimental evaluation

In this section we evaluate the performance and the scalability of Andromeda. Our experiments aim at i) proving the efficiency of the system in processing queries and updates on large documents, and ii) showing how the system scales with the document size and the number of nodes in the cluster.

6.1 Experimental setup

We performed our experiments on a *multitenant* cluster. For these experiments we used a single master machine and 100 slave machines. The master has two Intel Westmere (Hex-core) CPUs (24 cores total), 96 GB of RAM, and 6x136 GB drives (RAID5); slave machines have two Intel Westmere (Hex-core) CPUs (24 cores total), 48 GB of RAM, and 12x2 TB drives (7200 rpm). Each cluster node runs 64-bit RHEL 6.4, Java 1.7, and Hadoop 2.2.0. Cluster nodes are connected through an Infiniband network. We assigned 3 GB (precisely, 3200MB) of main memory to each mapper and 1.5 GB (precisely, 1524 MB) of memory to each reducer; we also set the maximum heap size for mappers to 2.5 GB (precisely, 2500 MB). Due to the complex memory management policy of Hadoop 2.2, we had no real control on the memory allocated by Hadoop to the *application master* container; by inspecting at run-time the allocated memory, we discovered that Hadoop 2.2.0 assigned to the application master a bit less than 1 GB of main memory.

Table 2 Query dataset

Name	Factor	Actual Size (GB)
10GB	100	10.95
15GB	150	16.43
20GB	200	21.92
25GB	250	27.41
30GB	300	32.89

To reduce issues related to independent system activities and other jobs in the cluster, we ran each experiment five times, discarded both the highest (worst) and the lowest (best) processing times, and reported the average processing time of the remaining runs.

6.2 Datasets

We performed our experiments on two distinct datasets. The first dataset is dedicated to query experiments, and comprises five XMark [(Schmidt et al. 2002)] XML documents obtained by running the XMark data generator with factors 100, 150, 200, 250, and 300, respectively; the resulting documents have approximate sizes ranging from 10GB to 32GB. These documents contain data concerning an auction site, and only have textual content coming from Shakespeare's plays, without any binary object. To give an idea of the nature of these documents, the 20GB document contains about 650 million nodes, 74 distinct element and attribute names, and has 13 nested levels.

The second dataset is used for update tests and contains ten XMark documents whose size ranges approximately from 1GB to 10GB. In Tables 2 and 3 we summarise the characteristics of these datasets.

Table 3 Update dataset

Name	Factor	Actual Size (GB)
1GB	10	1.09
2GB	20	2.18
3GB	30	3.28
4GB	40	4.375
5GB	50	5.47
6GB	60	6.57
7GB	70	7.66
8GB	80	8.75
9GB	90	9.85
10GB	100	10.95

6.3 Evaluating queries

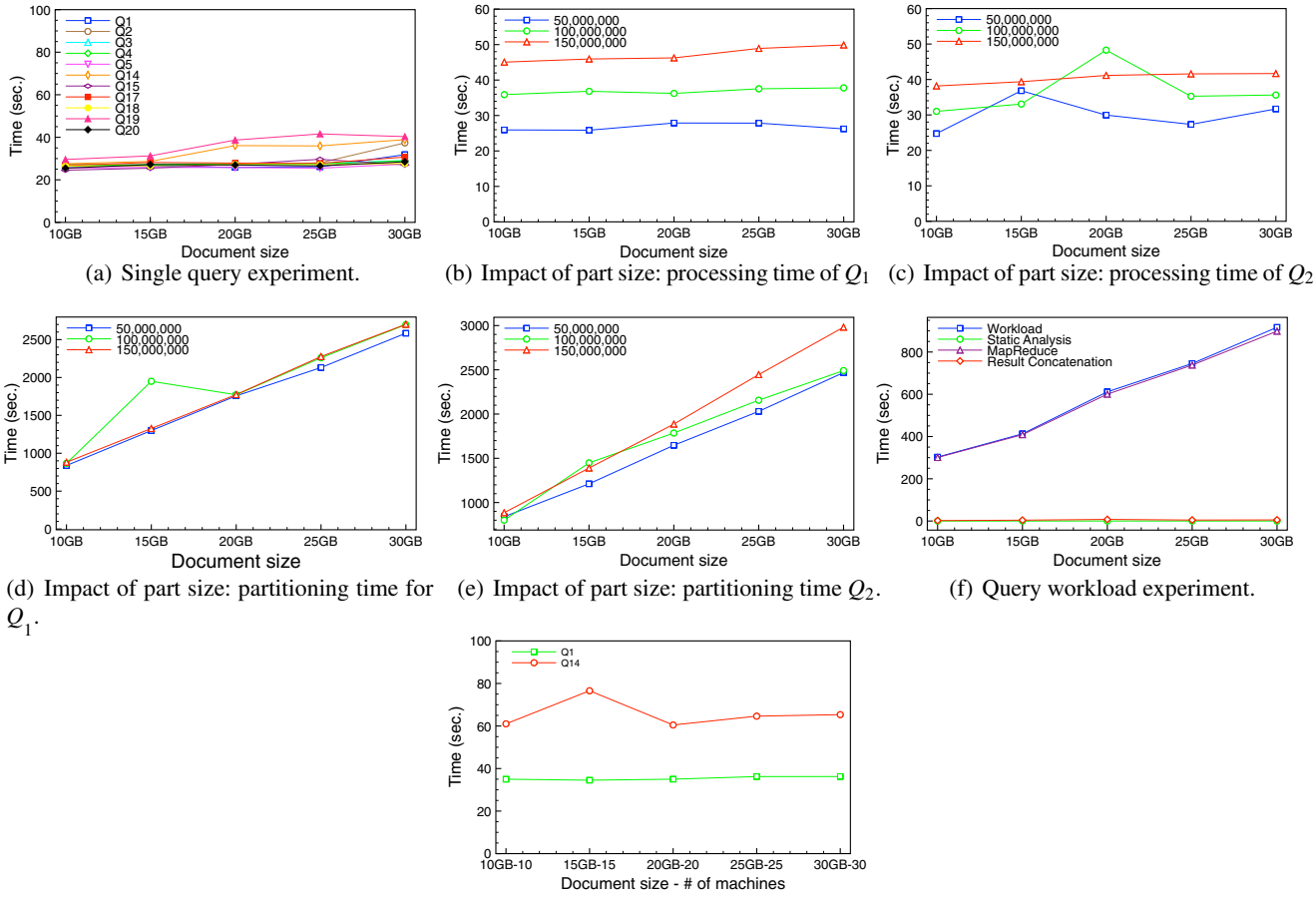
In our first battery of experiments we tested the performance and the scalability of our system when processing queries. In the first test we selected a subset of the iterative fragment of the XMark benchmark query set, and, in particular, queries Q_1, Q_2, Q_3, Q_4, Q_5, Q_{14}, Q_{15}, Q_{17}, Q_{18}, Q_{19}, and Q_{20}, and processed each query individually on the documents of the first data set; in this experiment we used parts of 100 million bytes size, as this dimension is a good trade-off between efficiency, Hadoop latency, and space overhead. The results we obtained are shown in Fig. 13a. This graph indicates that the evaluation time is only partially affected by the size of the input document; this is motivated by the fact that Andromeda filters out parts that do not structurally match the input query, and processes the query only on those parts that may give a contribution to the result; hence, even for large documents, the number of machines actually used by the system is below the cluster size.

Partitioning time for exemplifying queries Q_1, Q_2, Q_5, and Q_{14} is reported in Table 4, together with the number of *generated* and *used* parts. As we mentioned before, unused parts are discarded. We chose these queries as Q_1 is simple and very selective, Q_2 selects textual content, Q_5 contains a nested query and an aggregation function, and Q_{14}, finally, has low selectivity and uses a full-text predicate.

As it can be easily observed, the partitioning time grows linearly with the size of the input document and the number of used parts is only a small fraction of the total number of parts, with the only notable exception of query Q_{14}, which is not very selective. This explains why the processing time of queries Q_{14} and Q_{19}, that uses exactly the same partitioning scheme of query Q_{14}, is bigger than that of the remaining queries.

From this table we can also observe that partitioning induces a modest space overhead wrt the size of the original document. This overhead is related to artificial tags and elements, that must be repeated in multiple parts; the use of EXI compression helps us in tackling this problem, allowing the system to cut the space requirements in half.

Impact of part size on query processing In our second experiment we analyzed how the part size impact query processing and partitioning time. Our intuition was that smaller parts should improve the parallelism of the system by increasing the number of map input records to be processed, at the price of an higher space overhead. On the contrary, bigger parts should lower the overhead and thus the *global* number of writings to disk, hence improving the I/O efficiency. To understand if our intuition was correct, we evaluated queries Q_1 and Q_2 on the first dataset by using three different part sizes: 50, 100, and 150 million bytes. The results we collected are shown in Fig. 13b, c, d, and e.



(a) Single query experiment. (b) Impact of part size: processing time of Q_1 (c) Impact of part size: processing time of Q_2

(d) Impact of part size: partitioning time for Q_1. (e) Impact of part size: partitioning time Q_2. (f) Query workload experiment.

(g) Query horizontal scalability.

Fig. 13 Query experiments

Figure 13b shows the impact of part size on the processing time of query Q_1. This graphs confirms our intuition: indeed, smaller parts result in lower evaluation times.

Figure 13c, instead, illustrates the behaviour of the system when processing query Q_2. As it can be easily observed, the behaviour of system in this case is a bit irregular and the gap among processing times seems to narrow when document size increases. Actually, when performing this experiment, we observed a heavy load on the cluster, due to simultaneous activities by other research groups, which explains the peaks corresponding to the 15GB and the 20GB documents.

Cluster load, however, does not explain why the difference in processing times is smaller than in the case of query Q_1. Indeed, in this case query selectivity comes into play. Query Q_2, actually, is less selective than query Q_1, as it can be observed from Table 4; this implies that the number of parts to be processed is greater than that for query Q_1. In particular, this number can easily exceed the cluster size for parts of 50 million bytes, which implies that each

Table 4 Partitioning time (sec.), generated parts, and used parts

Size	Q_1			Q_2			Q_5			Q_{14}		
	Time	Gen.	Used (%)	Time	Gen.	Used (%)	Time	Gen.	Used (%)	Time	Gen.	Used (%)
10GB	851.686	142	13 (9.1 %)	706.45	138	31 (22.4 %)	813.448	144	17 (11.8 %)	810.014	138	59 (42.7 %)
15GB	1148	214	19 (8.8 %)	1060	207	47 (22.7 %)	1243	217	26 (11.9 %)	1250	208	89 (42.7 %)
20GB	1564	285	26 (9.1 %)	1461	277	62 (22.3 %)	1666	290	35 (12 %)	1700	277	118 (42.5 %)
25GB	2007	357	32 (8.9 %)	1808	347	77 (22.1 %)	2215	363	43 (11.8 %)	2299	347	148 (42.6 %)
30GB	2391	429	38 (8.8 %)	2147	417	93 (22.3 %)	2526	436	52 (11.9 %)	2534	417	177 (42.4 %)

mapper must process more than one part; this may introduce an additional overhead induced by Hadoop, which explains the gap narrowing.

Figure 13d and e describe the impact of part size on partitioning time for queries Q_1 and Q_2, respectively. In the case of Q_1, there is no big difference among partitioning times; this difference, however, widens in the case of Q_2, in particular for what concerns the 25GB and 30GB documents. This phenomenon is probably related to higher garbage collection times. In both cases, partitioning time grows linearly with the document size.

Processing workloads In our third experiment we evaluated the performance of our system when processing a workload comprising all the queries of our query set. Queries were processed on a single partition compatible with each query, by exploiting the ability of Andromeda to partition a document according to multiple paths. The results we collected are shown in Fig. 13f and Table 5.

In Fig.13f we reported the total workload processing time. It is worthy to note that workload processing time grows linearly with the size of the input document. This is implied by the fact that, even on smaller documents, the parallel execution of the queries in the workload involves the use of all the machines in the cluster, as confirmed by Table 5, which reports the partitioning time, the number of generated parts, and the number of map input records (parts to process) for each input document: as shown in this table, even on the 10GB document the cluster is fully exploited.

Horizontal scalability: changing cluster size In our last experiment on queries we evaluated the horizontal scalability of the system when processing queries Q_1 and Q_{14}: we chose these queries as they are representative of high selectivity (Q_1) and low selectivity (Q_{14}) queries; Q_{14} also contains a *full-text* predicate that is quite stressful for XQuery engines. In particular, we increased the cluster size as the size of the input document increases, as reported in Table 6. The results of this experiment are reported in Fig.13g. As expected, the system scales beautifully on query Q_1, as this exploits only a modest number of machines. Surprisingly enough, we got a similar result for query Q_{14} too.

Table 5 Workload: partitioning time (sec.), generated parts, and map input records

Size	Time	Gen.	Map input records
10GB	694.342	120	1309
15GB	1070106	181	1980
20GB	1424.379	241	2618
25GB	1876.539	302	3289
30GB	2138.638	362	3938

Table 6 Horizontal scalability: cluster configuration

Size	Machines
10GB	10
15GB	15
20GB	20
25GB	25
30GB	30

This shows that, even when fully loaded, the system scales well and can efficiently process complex iterative queries.

6.4 Evaluating updates

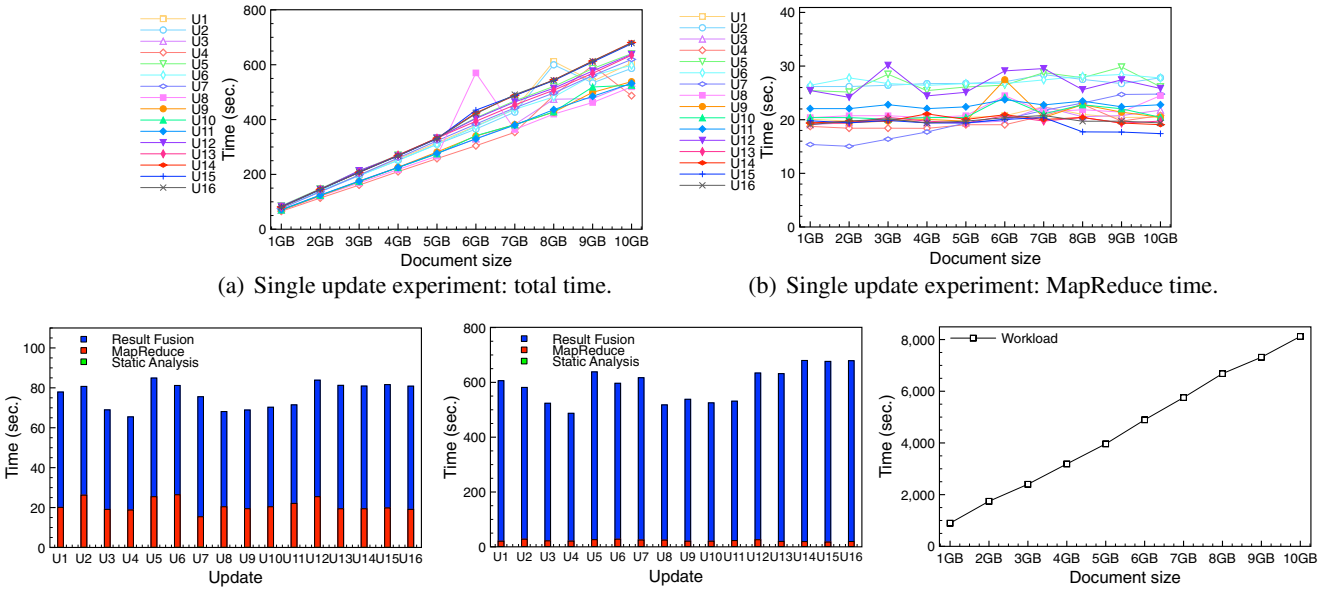
In our second battery of experiments we evaluated the performance of Andromeda when processing updates in different scenarios. We evaluated each update in a set of iterative updates (see Appendix A for more details) against the documents in the second dataset of Section 6.2; in all tests we used parts of 100 million bytes (about 95 MB).

Scalability of update processing In our first test we analysed the behaviour of Andromeda when individually executing 16 iterative updates. All these updates return a new document. Figure 14a illustrates the total execution time for each update without partitioning time.

Unlike what happens for queries, update processing is deeply influenced by the input document size, as execution time grows linearly with it. This is motivated by the fact that the system must produce an updated document by combining the updated parts with the parts of the original document that were not touched by the update: this requires the system to traverse all the document parts. To validate this claim we reported in Fig. 14b the update processing time without part concatenation; as it can be observed, in this case update processing exposes a behavior close to that shown on queries (see Fig. 13a).

To further validate the previous claim, we decomposed the total processing time of updates for the 1GB and the 10GB documents, as shown in Fig. 14c and d; as it can be noted, the impact of result concatenation becomes more and more relevant as document size increases. In particular, for the biggest document the execution time is largely dominated by result concatenation, which counts for over a 90% of the total time.

Processing mixed workloads In our second test we created a random query/update workload and analyzed the behaviour of the system when processing the workload on



(a) Single update experiment: total time.

(b) Single update experiment: MapReduce time.

(c) Analysis of update processing time: 1GB.

(d) Analysis of update processing time: 10GB.

(e) Query/update workload.

Fig. 14 Update experiments

documents of increasing size. The workload comprises 20 expressions randomly chosen by an initialization script, that also chooses the execution order: queries and updates are executed according to the *reader/writer* semantics, hence queries can be evaluated simultaneously, while updates have to be processed individually. Queries and updates are selected by respecting a 80:20 ratio, hence the workload contains 16 queries and 4 updates. The composition of the workload we considered is reported below:

$$W = (U_2, U_{12}, [Q_{18}, Q_{17}, Q_3, Q_1, Q_{18}], U_4, U_{14}, \\ [Q_{15}, Q_5, Q_2, Q_{17}, Q_{15}, Q_{15}, Q_{20}, Q_{10}, Q_1, Q_5, Q_{18}])$$

Figure 14e describes the behaviour of the system when processing the workload. As it can be observed, the workload execution time grows linearly with the input size, despite the fact that 16 tasks out of 20 are queries. This is caused by the presence of updates, which not only require result concatenation, but also force the system to partition the updated document for processing the next task, hence making partition reuse much less effective.

6.5 Comparison with other systems

In this section we analyze the performance of other systems supporting XQuery queries and updates, and compare it with that of Andromeda.

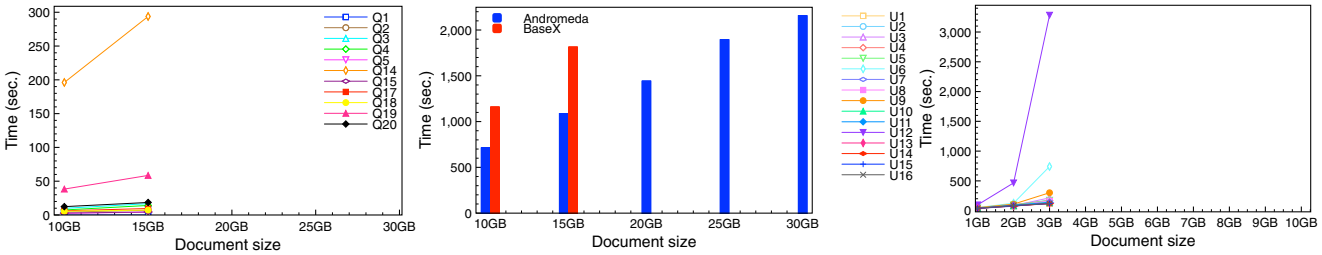
To the best of our knowledge, Andromeda is the only system based on MapReduce able to process both queries and updates. There are a few other systems, like HadoopXML (Choi et al. 2012), that only support XPath, but have no updating capabilities.

There are a few centralized systems, like Sedna (2011), Monetdb (2013), and Basex (2015), that process both queries and updates. However, Sedna and MonetDB are written in C/C++ and, for security reasons, cannot be run on our cluster, which accepts only pure Java executables. Hence, we evaluate here the performance of BaseX only.

We warn the reader that any comparison between a centralized system and one based on Hadoop is unfair. The centralized system, indeed, can exploit the resources of a single machine only, while the other one is executed on top of a cluster; furthermore, Hadoop introduces significant performance penalties related to its latency and to the use of a distributed file system, while a centralised system does suffer these issues and can leverage on the local file system, which is usually much faster than HDFS. Therefore, the comparison we are presenting here has the only purpose of highlighting the circumstances under which a given solution is more suitable than the other one.

To make the comparison a bit less unfair, we assigned to BaseX the same memory allocated to Andromeda.

The first BaseX experiment, whose results are shown in Fig. 15a, replicates the first experiment of Section 6.3.



(a) Single query experiment.

(b) Andromeda partitioning time vs BaseX database creation time.

(c) Single update experiment.

Fig. 15 BaseX experiments

We took advantage here of the ability of BaseX to create a database from an XML document; this database contains several indices that are used during query evaluation. As for Andromeda partitioning time, database creation and data indexing times are not reported in Fig. 15a.

We can observe two main things. First of all, BaseX was not able to complete the test and failed in indexing the 20 GB document due to memory errors. Second, as expected, BaseX is much faster than Andromeda in evaluating selective queries, but its performance is close to that of Andromeda on query Q_{19}, and even much worse on query Q_{14}. These queries are not selective, as they access a significant fragment of the input document; query Q_{14}, furthermore, contains a full-text predicate, which is not efficiently evaluated by BaseX. In Fig. 15b we compare the partitioning time of Andromeda with the database creation and indexing time of BaseX. In this task Andromeda is much faster than BaseX; this is not surprising, as BaseX creates multiple complex indices.

In our last BaseX experiment, whose results are shown in Fig. 15c, we executed each update of Section 6.4 on the update dataset. This test reproduces the first one of Section 6.4. As BaseX does not synchronize the internal representation of the document and the corresponding indices, we created a fresh database before each update evaluation; as for queries, database creation and indexing times were not reported in Fig. 15c.

As it can be observed from Fig. 15c, BaseX failed in completing the experiment, as it exhausted memory resources while processing the 4GB document. On smaller documents BaseX proved to be faster than Andromeda, with the only notable exception of update U_{12}, whose navigational clauses are not selective at all and require BaseX to access a large fragment of the input document.

To summarise, a system like Andromeda is best-suited for non selective queries and updates on large documents, while BaseX represents an effective and efficient alternative for small documents and selective queries.

7 Related works

Query processing systems There exist only a few systems able to process queries on XML data in distributed and cloud environments, e.g., ChuQL (Khatchadourian et al. 2011), MRQL (Fegaras et al. 2011), HadoopXML (Choi et al. 2012), PAXQuery (Camacho-Rodríguez et al. 2014), and VXQuery (Jr et al. 2015). Among them, HadoopXML is the system that most closely resembles Andromeda as it can transparently process XPath queries on an Hadoop cluster. HadoopXML requires a preliminary document indexing phase, close to Andromeda partitioning phase. Despite these similarities, HadoopXML only supports XPath queries, and, unlike Andromeda, cannot process XQuery queries or XUF updates.

PAXQuery and VXQuery are systems for processing XML queries on collections of (relatively) small XML documents scattered across a cloud computing cluster. While very efficient even on small clusters, they were not designed to evaluate queries on big documents. MRQL is a query processing system that supports an SQL-like query language that can be used to query XML and JSON data; MRQL directly translates queries into Java code that can be executed on top of Hadoop or Spark. While more powerful than PigLatin, MRQL cannot process complex XQuery queries and does not support updates. ChuQL, finally, is a language embedding XQuery that allows the programmer to distribute XQuery queries over MapReduce clusters. The programmer has the duty to manage low-level details about query parallelization, while Andromeda completely hides the underlying processing environment.

To the best of our knowledge, there is no system supporting XUF updates on big XML documents.

Partitioning techniques The partitioning technique employed by our system resembles that of Bordawekar et al. (2009), where an horizontal partitioning technique has

been proposed in order to ensure parallel execution of single XPath queries. The partitioning technique proposed in that work can be performed only on the main-memory representation of the input document, and, as a consequence, is not suitable for very large XML documents.

In Kling et al. (2010) 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, unlike our system, requires the use of schema information on the input document. Both techniques (Bordawekar et al. 2009; Kling et al. 2010) require strong interventions inside a query engine, while our system required only a minor extension of Qizx-open to support EXI compression.

A recent work (Cong et al. 2012) 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 some pre-existing technique. Another recent work (Choi et al. 2012) proposes an Hadoop-based architecture for processing multiple twig-patterns on a very large XML document. This system is able to deal with a subset of XPath 1.0 queries, and adopts static partitioning: the input document is statically partitioned into several blocks and some path information is added to blocks to avoid loss of structural information. Differently, our system supports both dynamic and static partitioning, and, importantly, supports mixed workloads containing both XQuery queries and updates.

8 Conclusions

In this paper we described the architecture, the basic principles, and the algorithms used in Andromeda, and analysed its performance and scalability. In particular, we described a partitioning model that can be exploited to process, in a distributed way, both queries and updates, and we also showed how this model can be improved for evaluating query-only workloads; furthermore, we illustrated the main issues we faced during the implementation of the system.

The experimental analysis confirms that Andromeda scales with the document size and the number of nodes in the cluster, and that it can efficiently process queries and updates on very large XML documents, in particular in the case of non selective queries and updates, unlike what happens for other systems.

Acknowledgements Authors would like to thank all the people that contributed to the design and development of Andromeda: Maurizio Nolé, Alessandro Solimando, and Federico Ulliana.

Appendix

A Updates

U_1: `for` $x `in` $auction/site/closed_auctions/closed_auction
`where` not ($x/annotation)
`return insert node`
< annotation > Empty Annotation < /annotation >
`as last into` $x

U_2: `for` $x `in` $auction/site/people/person/address
`where` $x/country/text()= "United States"
`return` (`replace node` $x `with`
< address >
< street > $x/street/text() < /street >
< city > "NewYork" < /city >
< country > "USA" < /country >
< province > $x/province/text() < /province >
< zipcode > $x/zipcode/text() < /zipcode >
< address >

U_3: `for` $x `in` $auction/site/regions//item/location
`where` $x/text()= "United States"
`return` (`replace value of node` $x `with` "USA")

U_4: `delete nodes` $auction/site/regions//item/mailbox/mail

U_5: `for` $x `in` $auction/site//text/bold
`return rename node` $x `as` "emph"

U_6: `for` $x `in` $auction/site/people/person
`where` not($x/homepage)
`return insert node` < homepage >
www.{$x/name/text()}
Page.com < /homepage > `after` $x/emailaddress

U_7: `delete nodes` $auction/site/regions/australia

U_8: `for` $x `in` $auction/site/open_auctions/open_auction
`where` ($x/privacy= "Yes")
`return delete node` $x

U_9: `for` $x `in` $auction/site/open_auctions/open_auction
`where` $x/bidder/increase <20
`return insert node`
< bidder >
< date > 08/17/2000 < /date >
< time > 15:15:15 < /time >
< personref / >
< increase > 1.50 < /increase >
< /bidder >
`after` $x/initial

U_{10}: `for` $x `in` $auction/site/regions//item
`where` ($x/mailbox/mail/date/text()= "07/04/1998")
`return insert node` < incategory / > `before`
$x/mailbox

U_{11}: `for` $x `in`
$auction/site/open_auctions/open_auction/annotation/
description/text
`where` ($x/keyword/emph/text()= “unique”) and
($x/bold)
`return insert node` <emph> newTexT</emph>
`before` $x/bold

U_{12}: `for` $x `in` $auction/site//text/emph
`return delete node` $x

U_{13}: `for` $x `in`
$auction/site/categories/category/description/parlist
`where` ($x/listitem/parlist)
`return replace node` $x `with` $x/listitem/parlist[1]

U_{14}: `for` $x `in` $auction/site/categories/category/
description/parlist/listitem
`where` ($x/parlist)
`return replace node` $x/parlist `with` <text>
newText</text>

U_{15}: `for` $x `in` $auction/site/categories/category/
description/parlist/listitem
`return replace node` $x `with` $x/parlist/listitem[1]

U_{16}: `for` $x `in` $auction/site/categories/category/
description/parlist/listitem
`return replace node` $x `with` $x/parlist/listitem

References

Baazizi, M.A., Bidoit, N., Colazzo, D., Malla, N., & Sahakyan, M. (2011). Projection for XML Update Optimization. In *Proceedings of the 14th International Conference on Extending Database Technology* (pp. 307–318).

Benedikt, M., & Cheney, J. (2009). Semantics, types and effects for xml updates. In Gardner, P., & Geerts, F. (Eds.) *DBPL, Springer, Lecture Notes in Computer Science*, (Vol. 5708 pp. 1–17).

Benzaken, V., Castagna, G., Colazzo, D., & Nguyen, K. (2006). Type-based xml projection. In *VLDB*.

Berglund, A., Boag, S., Chamberlin, D., Fernández, M.F., Kay, M., Robie, J., & Siméon, J. (2010). Xml path language (xpath) 2.0 (2nd edition). Tech. rep., World Wide Web Consortium, w3C Recommendation.

Basex (2015). http://www.basex.org.

Bidoit, N., Colazzo, D., Malla, N., & Sartiani, C. (2012). Partitioning XML documents for iterative queries. In *IDEAS, 2012, ACM, pp 51–60*.

Bidoit, N., Colazzo, D., Malla, N., Ulliana, F., Nolé, M., & Sartiani, C. (2013). Processing XML queries and updates on map/reduce clusters. In Guerrini, G., & Paton, N.W. (Eds.) *EDBT, ACM* (pp. 745–748).

Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., & Siméon, J. (2010). XQuery 1.0: An XML Query Language (2nd Edition). Tech. rep., World Wide Web Consortium, w3C Recommendation.

Bordawekar, R., Lim, L., & Shmueli, O. (2009). Parallelization of XPath queries using multi-core processors: challenges and experiences. In *EDBT*.

Camacho-Rodríguez, J., Colazzo, D., & Manolescu, I. (2014). Paxquery: A massively parallel xquery processor. In *Proceedings of the Third Workshop on Data analytics in the Cloud, 2014* (pp. 6:1–6:4).

Choi, H., Lee, K., Kim, S., Lee, Y., & Moon, B. (2012). Hadoopxml: a suite for parallel processing of massive XML data with multiple twig pattern queries. In *CIKM*.

Cong, G., Fan, W., Kementsietsidis, A., Li, J., & Liu, X. (2012). Partial evaluation for distributed XPath query processing and beyond. *ACM TODS*, *37*(4), 43.

Dean, J., & Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *OSDI, USENIX Association, pp 137–150*.

Exificient (2015). http://exificient.sourceforge.net.

Fegaras, L., Li, C., Gupta, U., & Philip, J. (2011). XML Query Optimization in Map-Reduce. In *WebDB*.

Goldman, R., & Widom, J. (1997). DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB'97*.

Jr, E.P.C., Westmann, T., Borkar, V.R., Carey, M.J., & Tsotras, V.J. (2015). Apache vxquery: A scalable xquery implementation. CoRR arXiv:1504.00331.

Khatchadourian, S., Consens, M.P., & Siméon, J. (2011). Having a chuql at XML on the cloud. In *Proceedings of the 5th Alberto Mendelzon International Workshop on Foundations of Data Management*.

Kling, P., Özsu, M. T., & Daudjee, K. (2010). Generating efficient execution plans for vertically partitioned xml databases. *PVLDB*, *4*(1), 1–11.

MapDB (2015). http://www.mapdb.org.

Marian, A., & Siméon, J. (2003). Projecting xml documents. In *VLDB, pp 213–224*.

Monetdb (2013). https://www.monetdb.org/Home.

Qizx-open (2013). http://www.xmlmind.com/qizxopen/.

Robie, J., Chamberlin, D., Dyck, M., Florescu, D., Melton, J., & Siméon, J. (2011). XQuery Update Facility 1.0. Tech. rep., World Wide Web Consortium, w3C Recommendation.

Sedna (2011). http://www.sedna.org.

Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., & Busse, R. (2002). XMark: A Benchmark for XML Data Management. In *VLDB, Morgan Kaufmann, pp 974–985*.

Schneider, J., Kamiya, T., Peintner, D., & Kyusakov, R. (2014). Efficient XML Interchange (EXI) Format 1.0 (2nd Edition). Tech. rep., World Wide Web Consortium, w3C Recommendation.

Snyder, S.L. (2010). *Efficient XML Interchange (EXI) compression and performance benefits: Development, implementation and evaluation. Master of science in modeling, virtual environments and simulation (moves)*. USA: Naval Postgraduate School.

Sonar, R.P., & Ali, M.S. (2015). Xml parsing: A review. *International Journal of Emerging Science and Engineering*, *3*(7), 40–43.

Nicole Bidoit was appointed as a professor at the University of Paris-Sud in 2001. Before joining the UPSud and the LRI (UMR 8623 CNRS), she was a researcher at CNRS, professor at the University of Paris 13 and University of Bordeaux 1. Her research interests are in the area of database models and languages. She participated to the Verso project at INRIA and contributed to the development of the Verso algebra. She has been one of the researchers in France to work on deductive databases (Datalog with negation). More recently, her work has been oriented towards temporal issues in database (dynamic constraints, transactions, temporal query languages) as well as logic foundations for query languages for XML data, optimization of XQuery queries an updates, and also provenance (Why not provenance).

Dario Colazzo graduated from University of Pisa and received his PhD from the same university, in 2004. After his PhD, Dario has been research visitor at Ecole Normale Supérieure in Paris, and a post-doc, first at University of Venezia and then at Universit'e Paris-Sud, where he became associate professor in 2005. Since 2013 he is full professor at Université Paris- Dauphine. His research activities focus on safe and efficient management of massive semi-structured data.

Noor Malla obtained his PhD degree from Université Paris-Sud in 2012, with a thesis focusing on partitioning techniques for XML processing. She is currently a teacher at Saudi school of Paris.

Carlo Sartiani graduated in Computer Science at University of Pisa and got his PhD in Computer Science from the Department of Computer Science of University of Pisa. He is Assistant Professor at University of Basilicata. His research interests range from databases to programming languages and type theory.