Building Large XML Stores in the Amazon Cloud

Jesús Camacho-Rodríguez, Dario Colazzo, Ioana Manolescu

LRI, Université Paris-Sud 11 and INRIA Saclay–Île-de-France

jesus.camacho_rodriguez@inria.fr, dario.colazzo@lri.fr, ioana.manolescu@inria.fr

***Abstract*—It has been by now widely accepted that an increasing part of the world's interesting data is either shared through the Web or directly produced through and for Web platforms using formats like XML (structured documents). We present a scalable store for managing a large corpora of XML documents built on top of off-the-shelf cloud infrastructure. We implement different indexing strategies to evaluate a query workload over the stored documents in the cloud. Moreover, each strategy presents different trade-offs between efficiency in query answering and cost for storing the index.**

I. Introduction

The recent development of commercial and scientific cloud computing environments has strongly impacted research and development in distributed software platforms. From a business perspective, cloud-based platforms release the application owner from the burden of administering the hardware, by providing resilience to failures as well as elastic scaling up and down of resources according to the demand. From a (data management) research perspective, the cloud provides a distributed, shared-nothing infrastructure for data storage and processing. This made cloud platforms the target of many recent works focusing e.g. on implementing basic database primitives on top of off-the-shelf cloud services [1], algebraic extensions of the Map-Reduce paradigm [2] for efficient parallelized processing of queries, based on simple relational-style data models such as in [3] or XML-style nested objects, as in [4], [5].

In this work, we investigate architectures for *large-scale distributed XML stores based on off-the-shelf cloud infrastructure*. One of our important goals was understanding the support that such cloud-based platform provides, and at what costs. Following other works [1], [6], we have used the Amazon cloud platform (Amazon Web Services [7], or AWS in short), among the most prominent commercial platforms today.

Our work was guided by the following objectives. First, our architecture should leverage AWS resources by scaling up to very large data volumes. Second, we aim at efficiency, in particular for the document storage and querying operations. We quantify this efficiency by the response time provided by the cloud-hosted application. Our third objective is to minimize cloud resource usage, or, in classical distributed databases terms, the total work required for our operations. This is all the more important since in a cloud, total work translates into monetary costs. In particular, a query asked against a repository holding a large number of documents should ideally be evaluated only on those documents on which it yields results. Fourth, to be generic, our architecture should be able to work based on any existing XML query processor. To attain these objectives, we have made the following choices:

- For scalability, we store XML documents within S3, Amazon Simple Storage Service.
- To minimize resource usage and for efficiency, we index the stored XML documents into SimpleDB, a simple database system supporting quite advanced SQL-style queries based on a key-value model. This allows to decide from the index only, which documents are concerned by a given query - at the cost of building and using the index.
- To re-use existing XML querying capabilities, we install an XML query processor into virtual machines within EC2, Amazon Elastic Compute Cloud. We use the index mostly to direct the queries to the XML documents which may hold results.

We have devised four XML indexing strategies within SimpleDB, and we present experiments on XML indexing and index-based query processing within AWS.

This document is organized as follows. Section II discusses related works. Section III describes the basic AWS features on which we build our work. Section IV details our basic XML indexing strategies within AWS, while Section V discusses how to scale up and distribute such indexes when the size of the XML database increases. Section VI briefly outlines the monetary costs in AWS. Section VII describes our experiments. Finally, Section VIII contains conclusive discussions.

II. Related works

To the best of our knowledge, distributed XML indexing and query processing directly based on a commercial cloud's services has not been attempted before. However, the greater problem of large-scale data management (and in particular XML data management) can be addressed from many other angles. We have mentioned before algebra-based approaches which aim at leveraging large-scale distributed infrastructures (a typical example of which are clouds) by intra-query parallelism [3], [4], [5]. In the case of XML, this means that within the processing of each query on each document, parallelism can be used. In contrast, in our work, we consider the evaluation of one query on one document as an atomic (inseparable) unit of processing, and focus on the horizontal scaling of (i) the index in SimpleDB and (ii) the overall indexing and query processing pipeline distributed over several AWS layers. Another line of works which may allow reaching the same global goal of managing XML in the cloud comprises commercial database products, such as Oracle Database, IBM DB2 and Microsoft SQL Server. These products have included

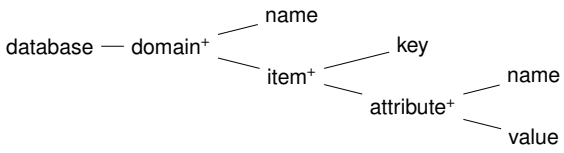


Fig. 1. Structure of a SimpleDB database.

XML storage and query processing capabilities in their relational databases over the last ten years, and then have ported their servers to cloud-based architectures [8]. Clearly, such systems have many functionalities beyond being XML stores, in exchange for complex architectures possibly influenced by the important existing code base. For example, as stated in [8]: "The design of Cloud SQL Server was heavily influenced by that of SQL Server, on which it is built". In this work, starting directly from the AWS platform allows us to get a detailed picture of attainable performance and associated monetary costs.

III. Preliminaries

We first describe AWS, then outline our architecture.

A. AWS overview

As of today, Amazon offers more than 20 different services in the cloud. In the following we describe those relevant for our system architecture. The first two are storage services, which we will use for storing the index and data respectively, the third is a computing service used for creating the index and evaluating queries, while the fourth is a queue service used for coordinating processes across machines.

1) Amazon SimpleDB: SimpleDB is a simple database that supports storing and querying simple, *structured* data. SimpleDB data is organized in domains. Each domain is a collection of items identified by their name. The length of the item name should be at most 1024 bytes. In turn, each item contains one or more attributes, each one consisting of a name-value pair. Furthermore, several values can be associated to the same attribute name under the same item. An attribute name or value can have up to 1024 bytes. Figure 1 shows the structure of a SimpleDB database. Each index entry (key, attribute name and attribute value) incurs a 45 byte storage charge for billing purposes only. Note that in contrast to relational databases, SimpleDB does not have a schema. For instance, different items within a domain may have different attribute names.

We can query a specific domain using a simple get operator: *get(D,k)* permits to retrieve all items in the domain D having a key value k. Moreover, one can execute a SQL-like $Select$ query over a domain. For example, the query *select * from mydomain where Year > '1955'* returns all items from domain *mydomain* where *Year* is greater than *1955*. On the other hand, to attach attributes to an item having a given key, the *put* operator can be used: *put(D,k,(a,v)+)* inserts the attributes *(a,v)+* into an item with key k in domain D (the item is created if does not exist); a *batch-put* variant inserts 25 items at a time. Finally, a *conditional put* operator can be used: *put(D,k,(a,v)+,c)* inserts or updates the item with key k in domain D if condition c is met.

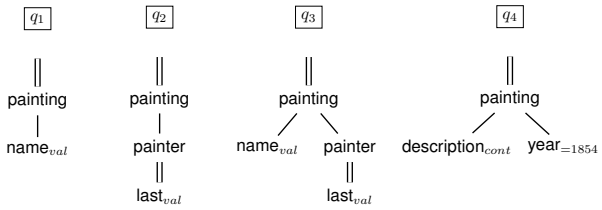


Fig. 2. Sample queries.

Multiple domains can not be queried by a single query. The combination of query results on different domains has to be done in the application layer. However, AWS ensures that operations over different domains run in parallel. Hence, splitting data into multiple domains could improve performances.

a) SimpleDB limits: AWS establishes some global limits in the data model used for SimpleDB. We briefly discuss the ones having an impact on the design of our platform. The maximal size a domain can have is 10GB, while the maximal number of attribute name-value pairs is 10^9. Concerning items, these can contain at most 256 attribute name-value pairs. The same limit is posed for the number of name-value pairs that can be associated to an item.

Querying using $Select$ is limited as follows. A query result may contain at most 2500 items, and can take at most 1 MB. If the query does not meet any of these conditions, the remaining results can be retrieved by iterating query execution.

2) Amazon Simple Storage Service: Amazon Simple Storage Service (or S3, in short) is a storage web service for raw data, hence ideal for large objects or files. S3 stores the data in buckets identified by their name. Each bucket contains objects, which have an associated unique name (within that bucket), metadata (both system-defined and user-defined), an access control policy for AWS users and a version ID. The number of objects that can be stored within a bucket is unlimited. Amazon states that there is not a performance difference between storing objects in many buckets or just a few.

3) Amazon Elastic Compute Cloud: Amazon Elastic Compute Cloud (or EC2, in short) is a web service that provides resizable computing capacity in the cloud. Using EC2, one can launch as many virtual computer instances as desired with a variety of operating systems and execute applications on them. Amazon provides some pre-configured images ready to be used immediately.

Images can be run in as many EC2 virtual computer instances as desired. AWS provides different types of instances, each of them with different hardware specifications etc.

4) Amazon Simple Queue Service: Amazon Simple Queue Service (or SQS, in short) provides reliable and scalable queues that enable asynchronous message-based communication between distributed components of an application.

B. Query language

In this paper we assume that queries are formulated by means of an expressive fragment of XQuery, amounting to value joins over tree patterns.

The tree patterns feature the ancestor and descendant axis, value-equality selections, and allow to return from one tree

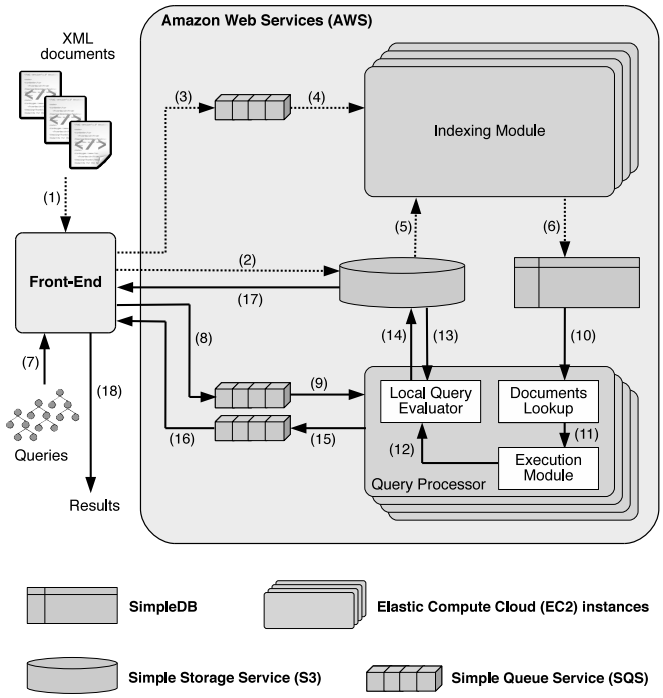


Fig. 3. Architecture overview.

pattern node its value ("string value" in the XPath/XQuery semantics) and/or its full serialized XML content. Figure 2 exemplifies some queries. We use $_{cont}$ to denote that the full XML subtree from the respective node should be returned, and $_{val}$ to return just the string value.

C. Proposed architecture

Figure 3 depicts our global architecture. We store the XML documents as uninterpreted BLOB objects in S3, because it is optimized for storing large objects inexpensively. On the other hand, we store the document index as small structured data items into SimpleDB, as it is optimized for data access speed. EC2 is used for creating the index and evaluating queries, while SQS is used for the communication between the different modules of the application.

A user interaction with the system, from storing data to querying it, involves the following steps.

When a document arrives (1), the front-end of our application stores it in S3 (2). Then, the front-end module creates a message containing the reference to the document and inserts it into the *loader request* queue (3). Such messages are retrieved by any of the EC2 instances running our indexing module (4). When a message is retrieved, the application loads the document referenced by the message from S3 (5) and creates the index data that is finally inserted into SimpleDB (6).

When a query arrives (7), a message containing the query is created and inserted into the *query request* queue (8). Such messages are retrieved by any of the EC2 instances running our query processor module (9). Then the index data is extracted from SimpleDB by API requests (10). Any other processing step needed for retrieving the required values after extracting data from the index is performed by a standard tuple-based XML querying engine (11), providing value- and structural joins, selections, projections etc.

After the document references (and possibly some other data) have been extracted from the index, the local query evaluator receives this information (12) and the XML documents cited are retrieved from S3 (13). Our framework includes "standard" XML query evaluation, i.e. the capability of evaluating a given query q over a given document d. This generalizes easily into an XQuery evaluation operator that computes the results of q over a set of documents $\{d_1, d_2, \ldots, d_k\}$. If needed, other operations can be performed on the results obtained by this operator.

Finally, we write the results obtained for the input query in S3 (14) and we create a message with the reference to those results and insert it into the *query response* queue (15). When the message is read by the front-end (16), the results are retrieved from S3 (17) and returned (18).

Note that the number of Amazon EC2 instances running our loader and query processor modules will scale dynamically depending on the workload.

IV. Indexing strategies

Several XML indexing schemes within AWS can be envisioned, with different trade-offs between query efficiency and indexing costs. We start with a few useful notations.

We denote by $URI(d)$ the Uniform Resource Identifier (or URI, in short) of d. For a given node $n \in d$, we denote by $inPath(n)$ the label path going from the root of d to the node n, and by $id(n)$ the node identifier (or ID, in short). In this work, we used simple (start, end, depth) identifiers used e.g. in [9] and many follow-up works. Such IDs allow identifying if node n_1 is an ancestor of node n_2 by testing whether n_1.*start*<n_2.*start* and n_1.*end*<n_2.*end*. If this is the case, moreover, n_1 is the parent of n_2 iff n_1.*depth*+1=n_2.*depth*.

For a given node $n \in d$, we use a function $key(n)$ computing a (string) value of an indexing key which we extract from n. Let $\underline{e}$, $\underline{a}$, $\underline{v}$ and $\underline{w}$ be four constant string tokens, and $\|$ denote string concatenation. We define $key(n)$ as:

$$key(n) = \begin{array}{l} \underline{e}\|n.label \text{ if } n \text{ is an XML element} \\ \underline{a}\|n.name \text{ if } n \text{ is an XML attribute} \\ \underline{v}\|n.val \text{ if } n \text{ is an attribute value} \\ \underline{w}\|n.val \text{ if } n \text{ is a word} \end{array}$$

In the sequel, to simplify reading, we will omit the $\|$ whenever this does not lead to confusion.

Given a document d, an *indexing strategy* I is a function that returns a set of tuples of the form $(k, (a, v)^+)^+$. In other words, $I(d)$ represents the set of items (or keys) k that must be added to the index store to reflect the new document d as well as the attribute name-value pairs (a, v) that are stored on the respective key(s).

A. Strategy LU (Label-URI)

Index For each node $n \in d$, strategy LU associates to the key $key(n)$ the pair $(URI(d), \epsilon)$ where ϵ denotes the null string. For example, applied to the documents depicted in Figure 4, LU produces among others the tuples shown in Figure 5.

Limits Due to the cardinality limitations of SimpleDB, if a single domain is used for the index, a given element or

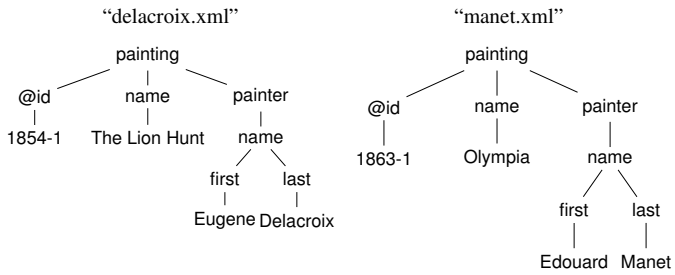


Fig. 4. Sample XML documents.

key	attribute name	attribute value
$\underline{e}$name	"delacroix.xml"	ϵ
$\underline{e}$name	"manet.xml"	ϵ
$\underline{a}$id	"manet.xml"	ϵ
$\underline{v}$1863-1	"manet.xml"	ϵ
$\underline{w}$Olympia'	'manet.xml"	ϵ

Fig. 5. Sample LU index tuples.

attribute name, attribute value or word cannot appear in more than 256 documents. Moreover, the index would not be able to hold more than 10^9 documents.

Look-up Index look-up based on the LU strategy is quite simple: all node names, attribute and element string values are extracted from the query and the respective look-ups are performed. Then URI sets thus obtained are intersected. The query is sent for evaluation on those documents whose URIs are part of the intersection.

B. Strategy LUP (Label-URI-Path)

Index For each node $n \in d$, LUP associates to $key(n)$ the pair $(URI(d), inPath(n))$. On the "delacroix.xml" and "manet.xml" documents, extracted tuples include:

key	attribute name	attribute value
$\underline{e}$name	"delacroix.xml"	/$\underline{e}$painting/$\underline{e}$name
$\underline{e}$name	"delacroix.xml"	/$\underline{e}$painting/$\underline{e}$painter/$\underline{e}$name
$\underline{e}$name	"manet.xml"	/$\underline{e}$painting/$\underline{e}$name
$\underline{e}$name	"manet.xml"	/$\underline{e}$painting/$\underline{e}$painter/$\underline{e}$name
$\underline{a}$id	"manet.xml"	/$\underline{e}$painting/$\underline{a}$id
$\underline{v}$1863-1	"manet.xml"	/$\underline{e}$painting/$\underline{a}$id/$\underline{v}$1863-1
$\underline{w}$Olympia	manet.xml"	/$\underline{e}$painting/$\underline{e}$name/$\underline{w}$Olympia

Limits An LUP index confined in a single SimpleDB domain can hold at most 256 (URI, path) pairs for a given value of the key function. Moreover, at most 10^9 different URI-path pairs can hold in the index.

Look-up The look-up strategy consists of finding, for each root-to-leaf path appearing in the query q, all documents having a data path that matches the query path. Here, a root-to-leaf query path is obtained simply by traversing the query tree and recording node keys and edge types. For instance, for the query q_3 in Figure 2, the paths are: /$\underline{e}$painting/$\underline{e}$name and //$\underline{e}$painting/$\underline{e}$painter//$\underline{e}$last. To find the URIs of all documents matching a given query path $a_1(/|//)a_2(/|//)\ldots(/|//)a_k$, we look up in the LUP index all paths associated to $key(a_k)$, and then filter them to only those matching the path.

C. Strategy LUI (Label-URI-ID)

Index To each key $k(n)$, strategy LUI associates the pair $(URI(d), id(n))$. For instance, from the documents "delacroix.xml" and "manet.xml", some extracted tuples are:

key	attribute name	attribute value
$\underline{e}$name	"delacroix.xml"	$[24, 50, 1]$
$\underline{e}$name	"delacroix.xml"	$[61, 120, 2]$
$\underline{e}$name	"manet.xml"	$[24, 45, 1]$
$\underline{e}$name	"manet.xml"	$[56, 112, 2]$
$\underline{a}$id	"manet.xml"	$[10, 21, 2]$
$\underline{v}$1863-1	"manet.xml"	$[14, 20, 3]$
$\underline{w}$Olympia	"manet.xml"	$[30, 37, 3]$

Limits If the index is confined to one domain, a given node name, attribute value or word cannot appear more than 256 times in the repository. At the same time, not more than 10^9 keys can be used in the repository.

Look-up Index look-up based on LUI starts by searching the index for all the query labels. For instance, for the query q_3 in Figure 2, the look-ups will be $\underline{e}$painting, $\underline{e}$description, $\underline{e}$year and $\underline{w}$1854. Then, the resulting (URI, id) collections are combined though a Holistic Twig Join [10]. One remaining difficulty is that the Holistic Twig Join needs inputs to be sorted, first, by URI, and then, by the *pre* component of the structural identifier. SimpleDB does not allow one to control the order in which data is stored, thus, after retrieving it from the index, a blocking sorting step is needed.

The LUI strategy has no false positives. It leads directly to the IDs of nodes matching the query.

D. Strategy $2LUPI$ (Label-URI-Path, Label-URI-ID)

Index This strategy uses a two level index: the first index is the same as for LUP, while the second index is the one used by LUI. Sample index tuples resulting from the documents "delacroix.xml" and "manet.xml" are shown in Figure 6.

Limits This index limits can be derived from those described in sections IV-B and IV-C. Each of the indexes should be stored in a different SimpleDB domain in order to avoid interferences between the keys of both indexes.

Look-up The look-up for $2LUPI$ combines the information glanced from both halves of the index. On one hand, a look-up is made with the "1st key" to retrieve the URIs of all documents matching some query paths. For instance, given the query q_4 in Figure 2, we extract the URIs of the documents matching //$\underline{e}$painting/$\underline{e}$description and //$\underline{e}$painting/$\underline{e}$year/$\underline{w}$1854. The URI sets are intersected, and we obtain a relation which we denote $R_1(URI)$. This is reminiscent of the LUP look-up.

On the other hand, a look-up on the "2nd key" identifies the structural identifiers of the XML nodes whose labels appear in the query, together with the URIs on their documents. This reminds us of the LUI look-up. We denote these relations by $R_2^{a_1}$, $R_2^{a_2}$, ... $R_2^{a_k}$, assuming the query node labels and values are a_1, a_2, ..., a_k. Then:

- For each a_i, $1 \leq i \leq k$, we compute $S_2^{a_i} = R_2^{a_i} \ltimes_{URI} R_1(URI)$. In other words, we use $R_1(URI)$ to reduce the R_2 relations.
- We evaluate a holistic twig join over $S_2^{a_1}$, $S_2^{a_2}$, ..., $S_2^{a_k}$ to obtain tuples of IDs for all query matches in the indexed documents, together with the respective document URIs.

1st key	1st attribute name	1st attribute value
$\underline{e}$name	"delacroix.xml"	/$\underline{e}$painting/$\underline{e}$name
$\underline{e}$name	"delacroix.xml"	/$\underline{e}$painting/$\underline{e}$painter/$\underline{e}$name
$\underline{e}$name	"manet.xml"	/$\underline{e}$painting/$\underline{e}$name
$\underline{e}$name	"manet.xml"	/$\underline{e}$painting/$\underline{e}$painter/$\underline{e}$name
$\underline{a}$id	"manet.xml"	/$\underline{e}$painting/$\underline{a}$id
$\underline{v}$1863-1	"manet.xml"	/$\underline{e}$painting/$\underline{a}$id/$\underline{v}$1863-1
$\underline{w}$Olympia	"manet.xml"	/$\underline{e}$painting/$\underline{e}$name/$\underline{w}$Olympia

2nd key	2nd attribute name	2nd attribute value
$\underline{e}$name	"delacroix.xml"	$[24, 50, 1]$
$\underline{e}$name	"delacroix.xml"	$[61, 120, 2]$
$\underline{e}$name	"manet.xml"	$[24, 45, 1]$
$\underline{e}$name	"manet.xml"	$[56, 112, 2]$
$\underline{a}$id	"manet.xml"	$[10, 21, 2]$
$\underline{v}$1863-1	"manet.xml"	$[14, 20, 3]$
$\underline{w}$Olympia	"manet.xml"	$[30, 37, 3]$

Fig. 6. Sample tuples extracted by the $2LUPI$ strategy from the documents in Figure 4.

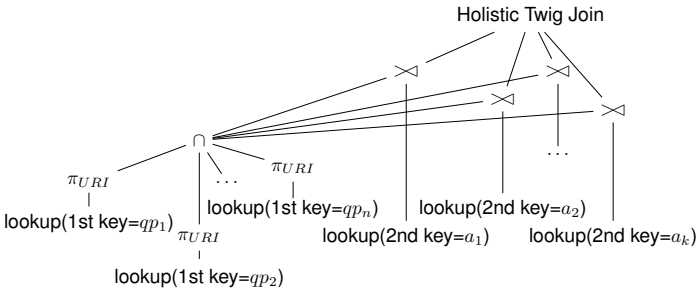


Fig. 7. Outline of the look-up on the $2LUPI$ strategy.

Figure 7 outlines this process; $a_1, a_2, \ldots, a_k$ are the labels extracted from the query, while $qp_1, qp_2, \ldots, qp_n$ are the root-to-leaf paths extracted from the query.

E. Queries with value joins

The processing of such queries is different, since one tree pattern only matches one XML document, whereas a query consisting of several tree patterns connected by a value join needs to be answered by combining tree pattern query results from different documents. Indeed, this is our evaluation strategy for such queries and any given indexing strategy I: evaluate first each tree pattern individually, exploiting the index; then, apply the value joins on the tree pattern results thus obtained.

V. Index distribution over multiple domains

To exploit the indexing strategies described in Section IV, we face two different scale-up problems due to global limits posed by AWS (Section III-A1). The first is associated to the domain limitations (e.g. 10 GB of data per domain), while the second is associated to the item limitations (e.g. 256 attribute name-value pairs per item). Our solution consists of distributing the index over several domains, and distributing each tuple produced by the indexing strategy into several items. This allows to overcome AWS limitations, and scale well while obtaining maximum performance from SimpleDB. To attain these goals we introduce and adopt a variant of *static* hash indexing. An index consists of a set of *directory* domains, each one pointing to a set of chained *data* domains.

Figure 8 depicts an example. The directory consists of n primary domains, identified by numbers from 0 to $n-1$, such that $n = 2^d$ for some integer d, also called the *directory depth*; in the example we have $n = 4$. We use a hash function h which, given a key value k, returns a number between 0 and $n-1$, determining the directory domain for k. This directory domain holds pointers to all the data domains storing items associated to k.

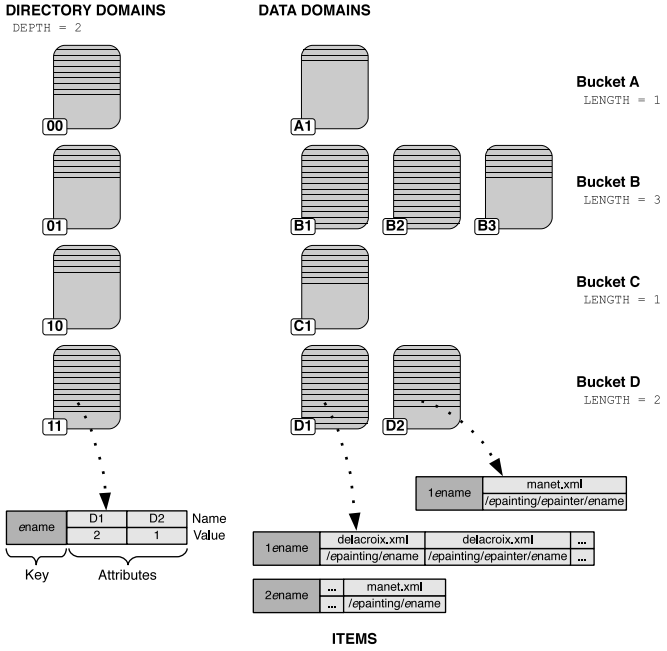


Fig. 8. Example of hash-based partitioning scheme ($n = 4$).

In the following, the set of data domains pointed to from a single directory domain is called a *bucket*, and its cardinality is called the *bucket length*. It is worth observing that in our specific AWS context, a bucket formed by multiple domains can be efficiently accessed during look-up operations as AWS guarantees parallel access to the these domains.

Each tuple $(k, (a, v)^+)$ extracted from a document by the index strategy I is stored in a set of n disjoint items. Items for the tuple $(k, (a, v)^+)$ are put in domains belonging to the bucket related to the domain $h(k)$. Several items are needed either because one item could not suffice for the entire tuple (see Section III-A1), or because the item containing the tuple can not be stored entirely in the current bucket domain D with free space. Each of these items contains a subset of attributes in $(k, (a, v)^+)$, and has as key the k value prefixed with an integer $i \in [1 + m \ldots n + m]$ (this prefix is needed to meet uniqueness of keys in a domain) where s is the number of existing items in D for the key value k.

If one domain can not include all the items for $(k, (a, v)^+)$, then these are distributed over multiple domains. To make look-up operations possible, the $h(k)$ domain will contain an item having key k, and attributes (dat,c) where: dat is the name of a bucket domain containing some of the items for a tuple $(k, (a, v)^+)$, and c is the number of such items in dat (we will see next how these data are used for look-up operations).

To illustrate, consider the example in Figure 8, containing items created by the following operations. A document "delacroix.xml" arrives to the store and the following two tuples are extracted from it:

$\underline{e}$name	"delacroix.xml"	/$\underline{e}$painting/$\underline{e}$name
$\underline{e}$name	"delacroix.xml"	/$\underline{e}$painting/$\underline{e}$painter/$\underline{e}$name

The result of $h(\underline{e}\text{name})$ maps the item to directory domain 11. As no item with key $\underline{e}$name exists in the directory, an item with key $1\underline{e}$name is created and enriched with the attribute pairs of the above tuples. This item is inserted into $D1$, which is the current bucket domain with available space. Finally, a pointer item ($\underline{e}$name,$(D1, 1)$) in the directory domain is created. When a second document "manet.xml" arrives to the store, the following tuples are extracted:

$\underline{e}$name	"manet.xml"	/$\underline{e}$painting/$\underline{e}$name
$\underline{e}$name	"manet.xml"	/$\underline{e}$painting/$\underline{e}$painter/$\underline{e}$name

The directory domain 11 contains an entry for $\underline{e}$name. This indicates that the bucket domain where new items have to be inserted is $D1$. This becomes full after adding the first attribute above to an item with key $2\underline{e}$name. Therefore, a new bucket domain $D2$ is created and enriched with an item with key $1\underline{e}$name containing the second attribute above; the bucket length is incremented accordingly. Finally, the attribute $(D2, 1)$ is added to the domain item with key $\underline{e}$name.

Look-up strategy To apply the look-up algorithms presented in Section IV, we need to obtain the $(a, v)^+$ pairs associated to a given key k. To this end, we proceed as follows. First, we retrieve the item with key k from the domain $h(k)$ in the directory. For each attribute (dat_i, c_i) contained in such item, we execute a set of *get* operations get($j \cdot k$, dat_i) with $j = 1 \ldots c_i$ to retrieve all the items for the k value in the data domain dat_i ($j \cdot k$ denotes the k value prefixed with j).

The look-up operation result (i.e. the set of $(a, v)^+$ pairs associated to k) is the union of the attribute pairs contained in the items returned by every get operation.

Consider again the example depicted in Figure 8. If we want to extract the pairs associated to item with key $\underline{e}$name, first we access the directory domain and obtain the item with such key. Then we retrieve the items referenced by the $\underline{e}$name item content, i.e. $1\underline{e}$name and $2\underline{e}$name from $D1$ and $1\underline{e}$name from $D2$. The expected result is the union of the attributes associated to each of them.

Directory limitations The proposed strategy provides a mean to scale the index by adding domains to buckets when needed. If a directory domain becomes full, then the whole index must be reorganized into a new index with a larger directory. However, we are quite confident that this is not likely to happen in practice because in a directory domain we have one item for each possible key value, and because the number of attributes in an item is upper bounded by the bucket length. Also, in our implementation, for each item in a directory domain, if needed we can cluster its attributes (dat_i, c_i) and compress them into a few attributes.

VI. Storage and prices

Uploading, indexing, hosting and querying data based on AWS services occupies some space and incurs some processing, leading to a price. For space reasons, we delegate details on the price calculations to our technical report [11]. The prices we paid as part of this study were those applying for the Ireland AWS facility, where all our experiments took place, in July-November 2011. The main figures are: $\$0.14$ charged per month to store 1 GB of data in S3; $\$0.275$ per month to store 1 GB of data in SimpleDB; $\$0.154$ per hour of SimpleDB instance utilization; $\$0.38$ per hour of a "large" EC2 instance, on which our query processor runs.

Importantly, SimpleDB imposes a constant storage overhead of 45 bytes for each distinct key, distinct attribute name, and distinct value. As we will show, this strongly penalizes some of our indexing strategies.

VII. Experiments

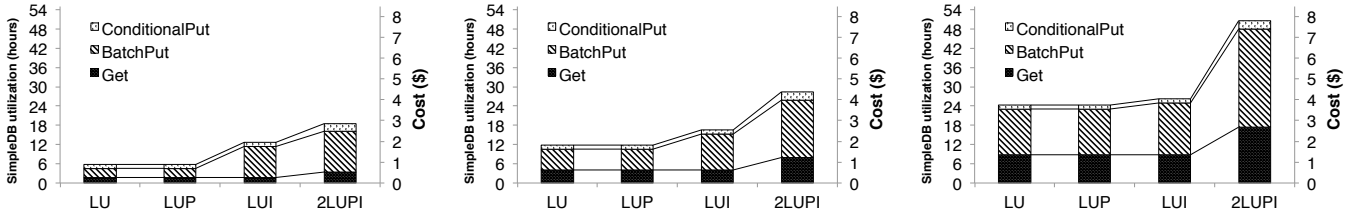
We implemented all algorithms previously described in Java, based on the Amazon Web Services SDK for Java v1.1.9 [12]. Our experiments ran in the AWS machines hosted in the EU (Ireland) region. We used the (centralized) Java-based XML query processor developed within our ViP2P project (http://vip2p.saclay.inria.fr), implementing an extension of the algorithm of [13] to our larger subset of XQuery. On our dialect, ViP2P's performance is close to (or better than) Saxon-B v9.1, thus we consider it a reasonable choice.

To test the selectivity provided by our indexing, we needed an XML corpus with some heterogeneity. We used XMark [14] data, and took advantage of an XMark generator feature enabling the generation of a "fragmented" auction site document. For a document size and a fragmentation factor k, one obtains k XML documents, all of which have a root element named ⟨site⟩, and which, together, contain the data of an XMark document of the specified size. Thus, for instance, the first document contains the subtree rooted in /site/regions/africa, the second document contains the subtree rooted in /site/regions/asia, and so on until the k-th document, which contains data from the /site/closed_auctions subtree.

We generated three data sets for the values $k = 10$, $k = 40$ and $k = 160$. The larger k is, the more documents differ from each other, since each represents a smaller "slice" from the global document. Each data set has a global size of 20 MB.

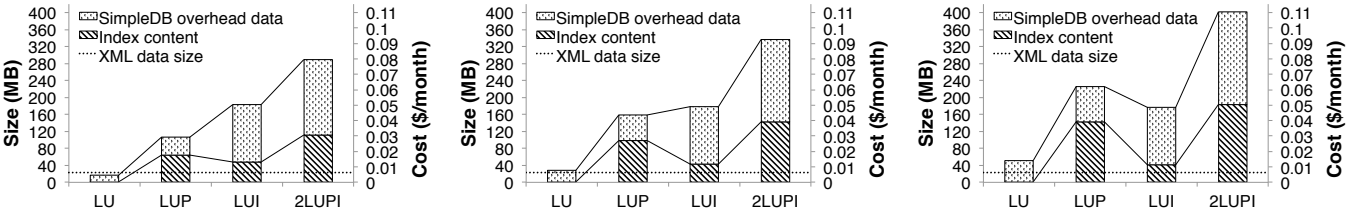
XML indexing We first study the cost associated to the creation and storage of the different indexes in SimpleDB. The indexing module ran on a single EC2 instance with 1.7 GB of memory and 1 virtual core with 1 EC2 Compute Unit. An EC2 Compute Unit is equivalent to the CPU capacity of a 1.0-1.2 GHz 2007 Xeon processor. For all the experiments, the depth of the directory (Section V) was fixed to 2.

The documents were initially stored in S3, from which they were gathered by the indexing module, which performs: *get* operations to check if an element already exists, *batchPut* operations to efficiently insert 25 items at a time into a SimpleDB domain, and *conditionalPut* operations to either



(a) 10 documents (b) 40 documents (c) 160 documents

Fig. 9. SimpleDB usage time, and cost in $ for indexing our XML data sets.



(a) 10 documents (b) 40 documents (c) 160 documents

Fig. 10. Index size and storage cost for our XML data sets.

insert a new item into a domain or update it (to associate a new attribute name and/or value) if it was already present.

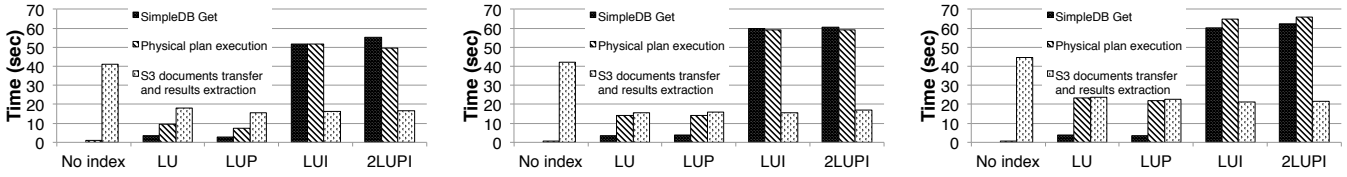
Figure 9 shows the machine utilization and cost associated to indexing the data sets according to each strategy. We notice, first, that the usage and costs increase as the data is fragmented into more (smaller) documents. This increase is due to the distinct URIs which lead to more attribute names within SimpleDB, and thus more calls to SimpleDB overall. The costs of LU and LUP (and therefore $2LUPI$) increase faster than for LUI. The reason is the *BatchPut* operation: while its cost for the LUI index is almost constant because the number of structural identifiers is practically the same among the documents sets, the rest of strategies benefits from the repetition of attributes that arises as the documents become larger. Finally, as expected, for each corpus of 10, 40 or 160 documents, the operation costs of fine-grained indexing strategies, such as LUI and $2LUPI$, are about twice as large as those associated to lower-granularity strategies.

Figure 10 shows the size and storage cost per month of each index for each data set, also compared with the original XML size. The Figure shows that indexes are quite larger than the data, in all cases except LU indexing on the 10 documents database. As expected, LU and LUP lead to more compact indexes. The SimpleDB space overhead (Section VI) plays a very visible role in this index space occupancy. Moreover, the index size difference increases with the fragmentation, because more documents means more URIs and thus more distinct SimpleDB entries. LUI index data is smaller than in the case of LUP, because IDs are more compact than paths. However, there are more nodes (thus, more node IDs) than paths, leading to more entries and significantly more space for LUI than for LUP. With $k = 10$, each path appears in "relatively few" documents; with $k = 160$, a given path, such as /site/regions/namerica/item may appears in tens of different documents, leading to more index content. Putting it all together, LUP occupies more space than LUI when there are few documents (some paths are frequent within a document), and the opposite holds with more documents.

XML query processing We used a subset of the XMark benchmark queries. Our workload consisted of 10 queries with an average of 10 nodes each (3 queries contained value joins). The query processor ran on EC2 instances with 7.5 GB of RAM and 2 virtual cores with 2 EC2 Compute Units.

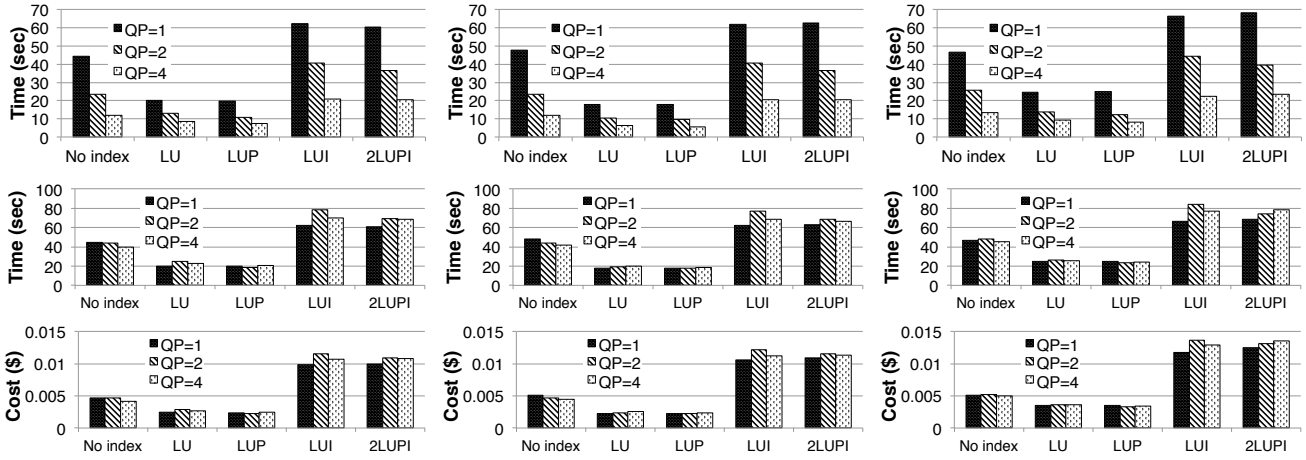
Figure 11 shows the workload response time for our four indexing strategies. For reference, we also show the time to process the query without using the index at all, that is: the query is evaluated against all documents. The Figure distinguishes the time to consult the index (SimpleDB *get*), the time to run the physical plan which identifies the relevant document URIs out of the data retrieved from the index, and finally the time to transfer the relevant documents from S3 to the EC2 instance and to extract results into S3 The Figure shows that only low-granularity indexes such as LU and LUP are competitive. The problem encountered by LUI and $2LUPI$ is that the structural XML joins which identify the relevant documents (recall Section IV-C) need *sorted* inputs, whereas SimpleDB index data is stored unordered. This requires sort operators, which are expensive and blocking. The important SimpleDB *get* times for LUI and $2LUPI$ is due to the fact that more index data is read and/or more calls are made.

Figure 12 shows the response time (perceived by the user), the total work (summing up parallel processing times), and the monetary costs associated to our workload, four indexing strategies plus the "no index" option, and different degrees of parallelism in the query evaluation step. In the Figure, QP=1 denotes a single EC2 instance running the query processor (no parallelism), whereas QP=2 and QP=4 show the times using 2, respectively 4 EC2 instances in parallel. The graphs in the top row depict the perceived response time. As expected, parallelism reduces response time, since the transfers between S3 and EC2 and query evaluation on EC2 are parallelized. The total work graphs (middle row) show that parallelism brings



(a) 10 documents (b) 40 documents (c) 160 documents

Fig. 11. Decomposition of the query response time for our workload and indexing strategies.



(a) 10 documents (b) 40 documents (c) 160 documents

Fig. 12. Response time (top), total work, summing up parallel execution times (center) and query processing costs (bottom) for our workload.

a low overhead, in exchange for the response time reduction. Finally, the query processing costs in the third row track, as expected, the total work.

Conclusion of the experiments Our experiments demonstrate the feasibility of our AWS-based architecture, using SimpleDB to index large XML corpora, S3 to store them and EC2 to process queries. The simple indexing strategies, LUI and LUP are quite efficient, reducing query response time and total work by a factor of 2 on average on our experiments. The benefit increased with the number and heterogeneity of documents, which is encouraging from the perspective of building even larger and more heterogeneous stores. Finally, we split the index over multiple SimpleDB domains and parallelized queries up to a factor of 4, demonstrating that our architecture scales well horizontally within the AWS cloud.

VIII. Conclusion

We have presented techniques to manage large XML stores in the cloud by means of AWS resources. We have investigated and compared by means of experiments several indexing strategies designed to statically filtering out documents with no result for queries. Our initial experimental results show that simpler index strategies may be the best ones either in terms of time and monetary costs, and that parallelization capabilities made available by AWS imply effective improvements in terms of time for query evaluation. In the future, we plan to study query optimization based on algebraic query representation and rewriting techniques. A cost model will play a crucial role in this task, and studying the impact of the monetary dimension will be of particular interest.

Acknowledgments

This work has been partially supported by the KIC EIT ICT Labs activity "Clouds for Connected Cities" 2011 (TCDT-11880) and an AWS in Education research grant.

References

[1] M. Brantner, D. Florescu, D. A. Graf, D. Kossmann, and T. Kraska, "Building a database on S3," in *SIGMOD*, 2008.
[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, 2004.
[3] A. Alexandrov, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, and D. Warneke, "MapReduce and PACT - Comparing Data Parallel Programming Models," in *BTW*, 2011.
[4] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *ICDE*, 2011.
[5] S. Khatchadourian, M. P. Consens, and J. Siméon, "Having a ChuQL at XML on the cloud," in *A. Mendelzon Int'l. Workshop*, 2011.
[6] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *PVLDB*, vol. 3, no. 1, 2010.
[7] "Amazon Web Services," http://aws.amazon.com/.
[8] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius, "Adapting Microsoft SQL server for cloud computing," in *ICDE*, 2011.
[9] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," in *ICDE*, 2002.
[10] N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: optimal XML pattern matching." in *SIGMOD*, 2002.
[11] http://jesus.camachorodriguez.name/_media/xml-aws/tech.pdf, 2011.
[12] "AWS SDK for Java," http://aws.amazon.com/sdkforjava/.
[13] Y. Chen, S. B. Davidson, and Y. Zheng, "An efficient XPath query processor for XML streams," in *ICDE*, 2006.
[14] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "XMark: A benchmark for XML data management," in *VLDB*, 2002.