Web Data Indexing in the Cloud: Efficiency and Cost Reductions

Jesús Camacho-Rodríguez Dario Colazzo Ioana Manolescu
OAK team, Inria Saclay and LRI, Université Paris-Sud
firstname.lastname@inria.fr

ABSTRACT

An increasing part of the world's data is either shared through the Web or directly produced through and for Web platforms, in particular using structured formats like XML or JSON. Cloud platforms are interesting candidates to handle large data repositories, due to their elastic scaling properties. Popular commercial clouds provide a variety of subsystems and primitives for storing data in specific formats (files, key-value pairs etc.) as well as dedicated sub-systems for running and coordinating execution within the cloud.

We propose an architecture for warehousing large-scale Web data, in particular XML, in a commercial cloud platform, specifically, Amazon Web Services. Since cloud users support monetary costs directly connected to their consumption of cloud resources, we focus on indexing content in the cloud. We study the applicability of several indexing strategies, and show that they lead not only to reducing query evaluation time, but also, importantly, to reducing the monetary costs associated with the exploitation of the cloud-based warehouse. Our architecture can be easily adapted to similar cloud-based complex data warehousing settings, carrying over the benefits of access path selection in the cloud.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Concurrency, Distributed databases, Query processing*

General Terms

Design, Performance, Economics, Experimentation

Keywords

Cloud Computing, Web Data Management, Query Processing, Monetary Cost

1. INTRODUCTION

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 or more recently JSON. Data-rich Web sites such as product catalogs, social media sites, RSS and tweets, blogs or online publications exemplify this trend. By today, many organizations recognize the value of the trove of Web data and the need for scalable platforms to manage it.

Simultaneously, the recent development of 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.

Over the last few years, big IT companies such as Amazon, Google or Microsoft have started providing an increasing number of cloud services built on top of their infrastructure. Using these commercial cloud platforms, organizations and individuals can take advantage of a deployed infrastructure and build their applications on top of it. An important feature of such platforms is their elasticity, i.e., the ability to allocate more (or less) computing power, storage, or other services, as the application demands grow or shrink. Cloud services are rented out based on specific service-level agreements (SLAs) characterizing their performance, reliability etc.

Although the services offered by public clouds vary, they all provide some form of scalable, durable, highly-available store for files, or (equivalently) binary large objects. Cloud platforms also provide virtual machines (typically called *instances*) which are started and shut down as needed, and on which one can actually deploy code to be run. This gives a basic roadmap for warehousing large volumes of Web data in the cloud in a Software-as-a-Service (SaaS) mode: to store the data, load it in the cloud-based file store; to process a query, deploy some instances, have them read data from the file store, compute query results and return them.

Clearly, the performance (response time) incurred by this processing is of importance; however, so are the monetary costs associated to this scenario, that is, the costs to load, store, and process the data for query answering. The costs billed by the cloud provider, in turn, are related to the total effort (or total work), in other terms, the total consumption of all the cloud resources, entailed by storage and query processing. In particular, when the warehouse is large, if query evaluation involves all (or a large share of) the data, this leads to high costs for: (*i*) reading the data from the file store and (*ii*) process the query on the data.

In this work, we investigate the usage of content indexing, as a tool to both improve query performance, and reduce the total costs of exploiting a warehouse of Web data within the cloud. We focus on tree-structured data, and in particular XML, due to the large adoption of this and other tree-shaped formats such as JSON, and we considered the

particular example of the Amazon Web Services (AWS, in short) platform, among the most widely adopted, and also target of previous research works [6, 23]. Since there is a strong similarity among commercial cloud platforms, our results could easily carry on to another platform (we briefly discuss this in Section 3). The contribution of this work are:

- A generic architecture for large-scale warehousing of complex semistructured data (in particular, tree-shaped data) in the cloud. Our focus is on building and exploiting an index to simultaneously speed up processing and reduce cloud resource consumption (and thus, warehouse operating costs);
- A concrete implemented platform following this architecture, demonstrating its practical interest and validating the claimed benefits of our indexes through experiments on a 40 GB dataset. We show that indexing can reduce processing time by up to two orders of magnitude and costs by one order of magnitude; moreover, index creation costs amortize very quickly as more queries are run.

Our work is among few to focus on cloud-based indexing for complex data; a preliminary version appeared in [8], and an integrated system based on the work described here was demonstrated in [4]. Among other previous works, our work can be seen as a continuation of [6, 19], which also exploited commercial clouds for fine-granularity data management, but (*i*) for relational data, (*ii*) with a stronger focus on transactions, and (*ii*) prior to the very efficient key-value stores we used to build indexes reducing both costs and query evaluation times.

The remainder of this paper is organized as follows. Section 2 discusses related works. Section 3 describes our architecture. Section 4 presents our query language, while Section 5 focuses on cloud-based indexing strategies. Section 6 details our system implementation on top of AWS, while Section 7 introduces its associated monetary cost model. Section 8 provides our experimental evaluation results. We then conclude and outline future directions.

2. RELATED WORK

To the best of our knowledge, distributed XML indexing and query processing directly based on commercial cloud services had not been attempted elsewhere, with the exception of our already mentioned preliminary work [8]. That work presented index-based querying strategies together with a first implementation on top of AWS, and focused on techniques to overcome Amazon SimpleDB[1] limitations for managing indexes. This paper features several important novelties. First, it presents an implementation which relies on Amazon's new key-value store, DynamoDB, in order to ensure better performance in managing indexes, and, quite importantly, more predictable monetary cost estimation. We provide a performance comparison between both services in Section 8. Second, it presents a proper monetary cost model, which still remains valid in the contexts of several alternative commercial cloud services. Third, it introduces multiple optimizations in the indexing strategies. Finally, it reports about extensive experiments on a large dataset, with a particular focus on performances in terms of monetary cost.

[1] `http://aws.amazon.com/simpledb/`

Alternative approaches which may permit to attain 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 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 [5]. Differently from our framework, such systems have many functionalities beyond those for XML stores, and require non-negligible efforts for their administration, since they are characterized by complex architectures.

Recently, a Hadoop-based architecture for processing multiple twig-patterns on a very large XML document has been proposed [10]. This system adopts static document partitioning; the input document is statically partitioned into several blocs before query processing, and some path information is added to blocs to avoid loss of structural information. Also, the system is able to deal with a fragment of XPath 1.0, and assumes the query workload to be known in advance, so that a particular query-index can be built and initially sent to mappers to start XML matching. Differently, in our system we do not adopt document partitioning, the query workload is dynamic (indexes only depends on data) and, importantly, we propose a model for monetary cost estimation.

Another related approach is to aim at leveraging large-scale distributed infrastructures (e.g., clouds) by intra-query parallelism, as in [11], enabling parallelism in the processing of each query, by exploiting multiple machines. Differently, in our work, we consider the evaluation of one query as an atomic (inseparable) unit of processing, and focus on the horizontal scaling of the overall indexing and query processing pipeline distributed over the cloud.

The greater problem of distributed XML data management has been previously addressed from many other angles. For instance, issues related to XML data management in clusters were largely studied within the Xyleme project [1]. Further, XML data management in P2P systems has been a topic of extensive research from the early 2000's [18].

Finally, some recent works related to cloud services have put special emphasis on the economic side of the cloud. For instance, the cost of multiple architectures for transaction processing on top of different commercial cloud providers is studied in [19]. In this case, the authors focus on read and update workloads rather than XML processing, as in our study. On the other hand, some works have proposed models for determining the optimal *price* [14] or have studied cost amortization [15] for data-based cloud services. However, in these works, the monetary costs are studied from the perspective of the cloud providers, rather than from the user perspective, as in our work.

3. ARCHITECTURE

We now describe our proposed architecture for Web data warehousing. To build a scalable, inexpensive data store, we store documents as files within Amazon's Simple Storage Service (S3, in short). To host the index, we have different requirements: fine-grained access, and fast look-up. Thus, we rely on Amazon's DynamoDB efficient key-value store for storing and exploiting the index. Within AWS, instances can be deployed through the Elastic Compute Cloud service (EC2, in short). We deploy EC2 instances to (*i*) extract from the loaded data index entries, and send them to

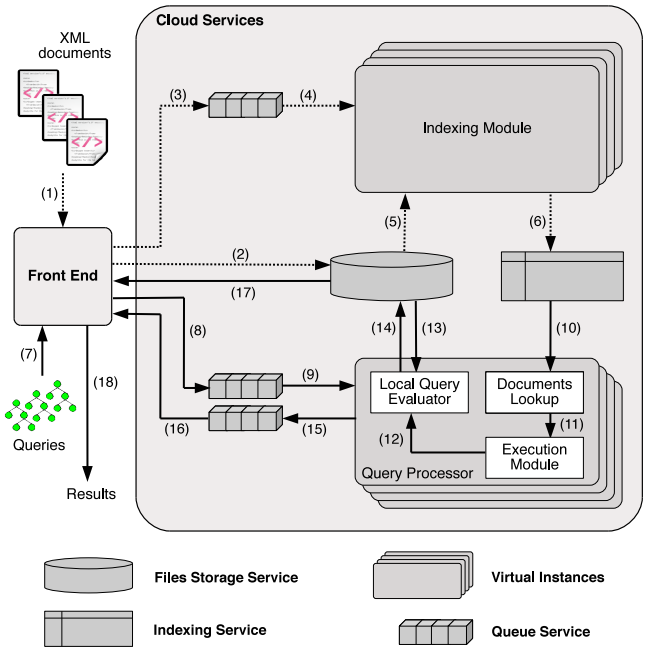


Figure 1: Architecture overview.

Dynamo DB's index and (*ii*) evaluate queries on those subsets of the database resulting from the query-driven index lookups. Finally, we rely on Amazon Simple Queue Service (SQS, in short) asynchronous queues to provide reliable communication between the different modules of the application. Figure 1 depicts this architecture.

User interaction with the system involves the following steps. When a document arrives (1), the front end of our application stores it in the file storage service (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 virtual machines running our indexing module (4). When a message is retrieved, the application loads the document referenced by the message from the file store (5) and creates the index data that is finally inserted into the index store (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 virtual instances running our query processor module (9). The index data is then extracted from the index store (10). Index storage services provide an API with rather simple functionalities typical of key-value stores, such as *get* and *put* requests. Thus, any other processing steps needed on the data retrieved from the index are performed by a standard XML querying engine (11), providing value- and structural joins, selections, projections etc.

After the document references have been extracted from the index, the local query evaluator receives this information (12) and the XML documents cited are retrieved from the file store (13). Our framework includes "standard" XML query evaluation, i.e. the capability of evaluating a given query q over a given document d. This is done by means of a single-site XML processor, which one can choose freely. Thus, the virtual instance runs the query processor over the documents and extracts the results for the query.

Finally, we write the results obtained in the file store (14) and we create a message with the reference to those results

Provider	File store	Key-value store	Computing	Queues
Amazon Web Services	Amazon Simple Storage Service	Amazon DynamoDB, Amazon SimpleDB	Amazon Elastic Cloud Compute	Amazon Simple Queue Service
Google Cloud	Google Cloud Storage	Google High Replication Datastore	Google Compute Engine	Google Task Queues
Windows Azure	Windows Azure BLOB Storage	Windows Azure Tables	Windows Azure Virtual Machines	Windows Azure Queues

Table 1: Component services from major commercial cloud platforms.

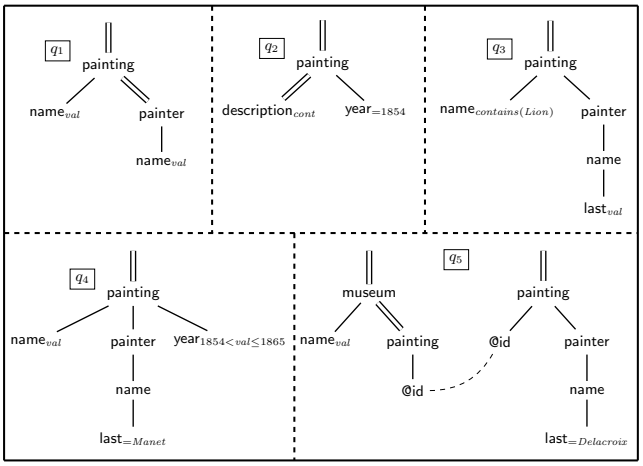


Figure 2: Sample queries.

and insert it into the *query response* queue (15). When the message is read by the front end (16), the results are retrieved from the file store (17) and returned (18).

Scalability, parallelism and fault-tolerance. The architecture described above exploits the elastic scaling of the cloud, for instance increasing and decreasing the number of virtual machines running each module. The synchronization through the message queues among modules provides inter-machine parallelism, whereas intra-machine parallelism is supported by multi-threading our code. We have also taken advantage of the primitives provided by message queues, in order to make our code resilient to a possible virtual instance crash: if an instance fails to renew its lease on the message which had caused a task to start, the message becomes available again and another virtual instance will take over the job.

Applicability to other cloud platforms. While we have instantiated the above architecture based on AWS, it can be easily ported to other well-known commercial clouds, since their services ranges are quite similar. Table 1 shows the services available in the Google and Microsoft cloud platforms, corresponding to the ones we use from AWS.

4. QUERY LANGUAGE

We consider queries are formulated in an expressive fragment of XQuery, amounting to value joins over tree patterns. For illustration, Figure 2 depicts some queries in a graphical notation which is easier to read. The translation to XQuery

syntax is pretty straightforward and we omit it; examples, as well as the formal pattern semantics, can be found in [21].

In a tree pattern, each node is labeled either with an XML element or attribute name. By convention, attribute names are prefixed with the @ sign. Parent-child relationships are represented by single lines while ancestor-descendant relationships are encoded by double lines.

Each node corresponding to an XML element may be annotated with zero or more among the labels *val* and *cont*, denoting, respectively, whether the string value of the element (obtained by concatenating all its text descendants) or the full XML subtree rooted at this node is needed. We support these different levels of information for the following reasons. The *val*ue of a node is used in the XQuery specification to determine whether two nodes are equal (e.g., whether the name of a painter is the same as the name of a book writer). The *cont*ent is the natural granularity of XML query results (full XML subtrees are returned by the evaluation of an XPath query). A tree pattern node corresponding to an XML attribute may be annotated with the label *val*, denoting that the string value of the attribute is returned. Further, any node may also be annotated with one among the following predicates on its *val*ue:

- An equality predicate of the form $= c$, where c is some constant, imposing that its *val*ue is equal to c.
- A containment predicate of the form $contains(c)$, imposing that its *val*ue contains the word c.
- A range predicate of the form $[a \leq val \leq b]$, where a and b are constants such that $a < b$, imposing that its *val*ue is inside that range.

Finally, a dashed line connecting two nodes joins two tree patterns, on the condition that the *val*ue of the respective nodes is the same on both sides.

For instance, in Figure 2, q_1 returns the pair (painting name, painter name) for each painting, q_2 returns the descriptions of paintings from *1854*, while q_3 returns the last name of painters having authored a painting whose name includes the word *Lion*. Query q_4 returns the name of the painting(s) by *Manet* created between *1854* and *1865*, and finally query q_5 returns the name of the museum(s) exposing paintings by *Delacroix*.

5. INDEXING STRATEGIES

Many indexing schemes have been devised in the literature, e.g., [13, 17, 22]. In this Section, we explain how we adapted a set of relatively simple XML indexing strategies, previously used in other distributed environments [2, 12] into our setting, where the index is built within a heterogeneous key-value store.

Notations. Before describing the different indexing strategies, we introduce some useful notations.

We denote by $URI(d)$ the Uniform Resource Identifier (or URI, in short) of an XML document 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). We rely on simple (*pre*, *post*, *depth*) identifiers used, e.g., in [3] and many follow-up works. Such IDs allow identifying if node n_1 is an ancestor of node n_2 by testing whether $n_1.pre < n_2.pre$ and $n_1.post < n_2.post$. If this is the case, moreover, n_1 is the parent of n_2 iff $n_1.depth+1 = n_2.depth$.

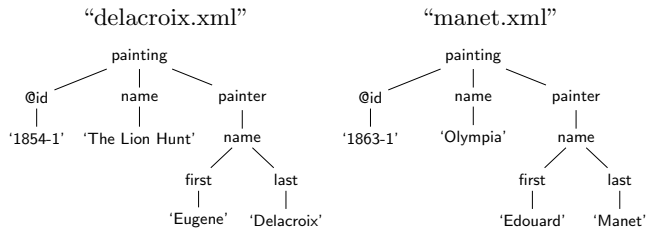


Figure 3: Sample XML documents.

For a given node $n \in d$, the function $key(n)$ computes a string key based on which n's information is indexed. Let $\underline{e}$, $\underline{a}$ and $\underline{w}$ be three constant string tokens, and $\|$ denote string concatenation. We define $key(n)$ as:

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

Observe that we extract *two* keys from an attribute node, one to reflect the attribute name and another which reflects also its value; these help speed up specific kinds of queries, as we will see. To simplify, we omit the $\|$ when this does not lead to confusion.

Indexing strategies. 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 keys k that must be added to the index store to reflect the new document d, as well as the attributes to be stored on the respective key. Each attribute contains a name a and a set of values v.

Table 2 depicts the proposed indexing strategies, which are explained in detail in the following.

5.1 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 3, LU produces among others these tuples:

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

Look-up. Given a query q expressed in the language described in Section 4, the look-up task consists of finding, by exploiting the index, and as precisely as possible, those parts of the document warehouse that may lead to query answers.

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 evaluated on those documents whose URIs are part of the intersection.

5.2 Strategy *LUP* (Label-URI-Path)

Index. For each node $n \in d$, LUP associates to $key(n)$ the attribute $(URI(d), \{inPath_1(n), \ldots, inPath_y(n)\})$. On

Name	Indexing function
LU	$I_{LU}(d) = \{(key(n),\ (URI(d), \epsilon)) \mid n \in d\}$
LUP	$I_{LUP}(d) = \{(key(n),\ (URI(d), \{inPath_1(n), inPath_2(n), \ldots, inPath_y(n)\})) \mid n \in d\}$
LUI	$I_{LUI}(d) = \{(key(n),\ (URI(d), id_1(n)\|id_2(n)\|\ldots\|id_z(n))) \mid n \in d\}$
$2LUPI$	$I_{2LUPI}(d) = \{[(key(n),\ (URI(d), \{inPath_1(n), inPath_2(n), \ldots, inPath_y(n)\})),$ $(key(n),\ (URI(d), id_1(n)\|id_2(n)\|\ldots\|id_z(n)))] \mid n \in d\}$

Table 2: Indexing strategies.

the "delacroix.xml" and "manet.xml" documents shown in Figure 3, extracted tuples include:

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

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_1 in Figure 2, the paths are: //$\underline{e}$painting/$\underline{e}$name and //$\underline{e}$painting//$\underline{e}$painter/$\underline{e}$name. To find the URIs of all documents matching a given query path

$$(/|//)a_1(/|//)a_2 \ldots (/|//)a_j$$

we look up in the LUP index all paths associated to $key(a_j)$, and then filter them to only those matching the path.

5.3 Strategy *LUI* (Label-URI-ID)

Index. The idea of this strategy is to concatenate the structural identifiers of a given node in a document, already *sorted* by their *pre* component, and store them into a single attribute value. We propose this implementation because structural XML joins which are used to identify the relevant documents need sorted inputs: thus, by keeping the identifiers ordered, we reduce the use of expensive sort operators after the look-up.

To each key $key(n)$, strategy LUI associates the pair

$$(URI(d), id_1(n)\|id_2(n)\|\ldots\|id_z(n))$$

such that $id_1(n) < id_2(n) < \ldots < id_z(n)$. For instance, from the documents "delacroix.xml" and "manet.xml", some extracted tuples are:

key	attribute name	attribute values
$\underline{e}$name	"delacroix.xml"	(3, 3, 2)(6, 8, 3)
	"manet.xml"	(3, 3, 2)(6, 8, 3)
$\underline{a}$id	"delacroix.xml"	(2, 1, 2)
	"manet.xml"	(2, 1, 2)
$\underline{a}$id 1863-1	"manet.xml"	(2, 1, 2)
$\underline{w}$Olympia	"manet.xml"	(4, 2, 3)

Look-up. Index look-up based on LUI starts by searching the index for all the query labels. For instance, for the query q_2 in Figure 2, the look-ups will be $\underline{e}$painting, $\underline{e}$description, $\underline{e}$year and $\underline{w}$1854. Then, the input to the Holistic Twig Join [7] must just be sorted by URI (recall that the structural identifiers for any given document are already sorted).

5.4 Strategy *2LUPI* (Label-URI-Path, Label-URI-ID)

Index. This strategy materializes two previously introduced indexes: LUP and LUI. Sample index tuples resulting from the documents in Figure 3 are shown in Figure 4.

Look-up. *2LUPI* exploits, first, LUP to obtain the set of documents containing matches for the query paths, and second, LUI to retrieve the IDs of the relevant nodes.

For instance, given the query q_2 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. A second look-up 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 $R_2^{a_1}$, $R_2^{a_2}$, ... $R_2^{a_j}$, assuming the query node labels and values are a_1, a_2, ..., a_j. Then:

- We compute $S_2^{a_i} = R_2^{a_i} \ltimes_{URI} R_1(URI)$ for each a_i, $1 \leq i \leq j$. In other words, we use $R_1(URI)$ to reduce the R_2 relations.
- We evaluate a holistic twig join over $S_2^{a_1}$, $S_2^{a_2}$, ..., $S_2^{a_j}$ to obtain URIs of the documents with query matches. The tuples for each input of the holistic join are obtained by sorting the attributes by their name and then breaking down their values into individual IDs.

Figure 5 outlines this process; $a_1, a_2, \ldots, a_j$ are the labels extracted from the query, while $qp_1, qp_2, \ldots, qp_m$ are the root-to-leaf paths extracted from the query.

It follows from the above explanation that *2LUPI* returns the same URIs as LUI. The reduction phase serves for prefiltering, to improve performance.

5.5 Range and value-joined queries

This type of queries which are supported by our language need special evaluation strategies.

Queries with range predicates. Range look-ups in key-value stores usually imply a full scan, which is very expensive. Thus, we adopt a two-step strategy. First, we perform the index look-up without taking into account the range predicate, in order to restrict the set of documents to be queried. Second, we evaluate the complete query over these documents, as usual.

Queries with value joins. Since one tree pattern only matches one XML document, 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.

1st key	1st attribute name	1st attribute values
ename	"delacroix.xml"	/epainting/ename, /epainting/epainter/ename
	"manet.xml"	/epainting/ename, /epainting/epainter/ename
aid	"delacroix.xml"	/epainting/aid
	"manet.xml"	/epainting/aid
aid 1863-1	"manet.xml"	/epainting/aid 1863-1
wOlympia	"manet.xml"	/epainting/ename/wOlympia

2nd key	2nd attribute name	2nd attribute values
ename	"delacroix.xml"	(3, 3, 2)(6, 8, 3)
	"manet.xml"	(3, 3, 2)(6, 8, 3)
aid	"delacroix.xml"	(2, 1, 2)
	"manet.xml"	(2, 1, 2)
aid 1863-1	"manet.xml"	(2, 1, 2)
wOlympia	"manet.xml"	(4, 2, 3)

Figure 4: Sample tuples extracted by the *2LUPI* strategy from the documents in Figure 3.

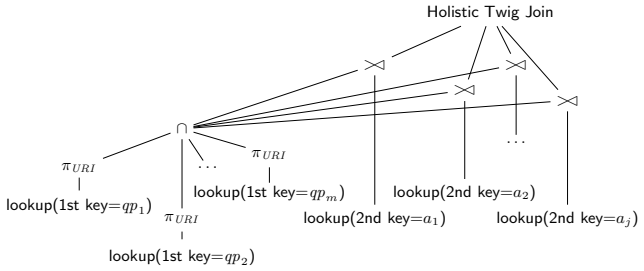


Figure 5: Outline of look-up using *2LUPI* strategy.

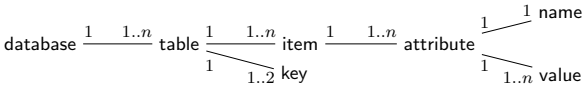


Figure 6: Structure of a DynamoDB database.

6. CONCRETE DEPLOYMENT

As outlined before, our architecture can be deployed on top of the main existing commercial cloud platforms (see Table 1). The concrete implementation we have used for our tests relies on Amazon Web Services. In this Section, we describe the AWS components employed in the implementation, and discuss their role in the whole architecture.

Amazon Simple Storage Service, or S3 in short, is a file storage service for raw data. S3 stores the data in buckets identified by their name. Each bucket consists of a set of objects, each having an associated unique name (within the bucket), metadata (both system-defined and user-defined), an access control policy for AWS users and a version ID. We opted for storing the whole data set in one bucket because (*i*) in our setting we did not use e.g. different access control policies for different users, and (*ii*) Amazon states that the number of buckets used for a given dataset does not affect S3 storage and retrieval performance.

Amazon DynamoDB is a NoSQL database service for storing and querying a collection of *tables*. Each table is a collection of *items* whose size can be at most 64KB. In turn, each item contains one or more *attributes*; an attribute has a name, and one or several values. Each table must contain a *primary key*, which can be either (*i*) a single-attribute hash key with a value of at most 2KB or (*ii*) a composite hash-range key that combines a 2KB hash key and a 1KB range key. Figure 6 shows the structure of a DynamoDB database. Different items within a table may have different sets of attributes.

A table can be queried through a *get(T,k)* operation, retrieving all items in the table T having a hash key value k. If the table uses a composite hash-range key, we may use it in a call *get(T,k,c)* which retrieves all items in table T with hash key k and range key satisfying the condition c. A *batchGet* variant permits to execute 100 *get* operations through a single API request. To create an item, we use the *put(T,(a,v^+)$^+$)* operation, which inserts the attribute(s) *(a,v^+)$^+$* into a newly created item in table T; in this case, *(a,v^+)$^+$* must include the attribute(s) defining the primary key in T. If an item already exists with the same primary key, the new item completely replaces the existing one. A *batchPut* variant inserts 25 items at a time.

Multiple tables cannot be queried by a single query. The combination of query results on different tables has to be done in the application layer.

In our system, Dynamo DB is used for storing and querying indexes. Previously presented indexing strategies are mapped to DynamoDB as follows. For every strategy but *2LUPI* the index is stored in a single table, while for *2LUPI* two different tables (one for each sub-index) are used. For any strategy, an entry is mapped into one or more items. Each item has a composite primary key, formed by a hash key and a range key. The first one corresponds to the key of the index entry, while the second one is a UUID [20] *global unique* identifier that can be created without a central authority, generated at indexing time. Attribute names and values of the entry are respectively stored into item attribute names and values.

Using UUIDs as range keys ensures that we can insert items in the index concurrently, from multiple virtual machines, as items with the same hash key always contain different range keys and thus cannot be overwritten. Also, using UUID instead of mapping each attribute name to a range key allows the system to reduce the number of items in the store for an index entry, and thus to improve performances at query time (we can recover all items for an index entry by means of a simple *get* operation).

Amazon Elastic Compute Cloud, or EC2 in short, 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.

AWS provides different types of instances, with different hardware specifications and prices, among which the users can choose: standard instances are well suited for most applications, high-memory instances for high throughput applications, high-CPU instances for compute-intensive applications. We have experimented with two types of standard instances, and we show in Section 8 the performance and cost differences between them.

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 over AWS. We rely on SQS heavily for circulating computing tasks between the various modules of our architecture, that is: from the front end to the virtual instances running our indexing module for loading

and indexing the data; from the front end again to the instances running the query processor for answering a query, then from the query processor back to the front end to indicate that the query has been answered and thus the results can be fetched.

7. APPLICATION COSTS

A relevant question in a cloud-hosted application context is, how much will this cost? Commercial cloud providers charge specific costs for the usage of their services. This section presents our model for estimating the monetary costs of uploading, indexing, hosting and querying Web data in a commercial cloud using our algorithms previously described. Section 7.1 presents the metrics used for calculating these costs, while Section 7.2 introduces the components of the cloud provider's pricing model relevant to our application. Finally, Section 7.3 proposes the cost model for index building, index storage and query answering.

7.1 Data set metrics

The following metrics capture *the impact of a given dataset, indexing strategy and query* on the charged costs.

Data-dependent metrics. Given a set of documents $\mathcal{D}$, $|\mathcal{D}|$ and $s(\mathcal{D})$ indicate the number of documents in $\mathcal{D}$, and the total size (in GB) of all the documents in $\mathcal{D}$, respectively.

Data- and index-determined metrics. For a given set of documents $\mathcal{D}$ and indexing strategy I, let $|op(\mathcal{D}, I)|$ be the number of *put* requests needed to store the index for $\mathcal{D}$ based on strategy I.

Let $t_{idx}(\mathcal{D}, I)$ be the time needed to build and store the index for $\mathcal{D}$ based on strategy I. The meaningful way to measure this in a cloud is: *the time elapsed between*

- *the moment when the first message (document loading request) entailed by loading $\mathcal{D}$ is retrieved from our application's queue, until*
- *the moment when the last such message was deleted from the queue.*

To compute the index size, we use:

- $sr(\mathcal{D}, I)$ is the raw size (in GB) of the data extracted from $\mathcal{D}$ according to I.
- Cloud key-value stores (in our case, DynamoDB) create their own auxiliary data structures, on top of the user data they store. We denote by $ovh(\mathcal{D}, I)$ the size (in GB) of the storage overhead incurred for the index data extracted from $\mathcal{D}$ according to I.

Thus, the size of the data stored in an indexing service is:

$$s(\mathcal{D}, I) = sr(\mathcal{D}, I) + ovh(\mathcal{D}, I)$$

Data-, index- and query-determined metrics. First, let $|r(q)|$ be the size (in GB) of the results for query q.

When using the indexing strategy I, for a query q and documents set $\mathcal{D}$, let $|op(q, \mathcal{D}, I)|$ be the number of *get* operations needed to look up documents that may contain answers to q, in an index for $\mathcal{D}$ built based on the strategy I. Similarly, let $\mathcal{D}_I^q$ be the subset of $\mathcal{D}$ resulting from look-up on the index built based on strategy I for query q.

$ST^{\$}_{m,GB}$ = \$0.125	$IDXst^{\$}_{m,GB}$ = \$1.14
$STput^{\$}$ = \$0.000011	$IDXput^{\$}$ = \$0.00000032
$STget^{\$}$ = \$0.0000011	$IDXget^{\$}$ = \$0.000000032
$VM^{\$}_{h,l}$ = \$0.34	$QS^{\$}$ = \$0.000001
$VM^{\$}_{h,xl}$ = \$0.68	$egress^{\$}_{GB}$ = \$0.19

Table 3: AWS Singapore costs as of October 2012.

Let $pt(q, \mathcal{D})$ be the time needed by the query processor to answer a query q over a dataset D without using any index, and $ptq(q, \mathcal{D}, I, \mathcal{D}_I^q)$ be the time to process q on an index built according to the strategy I (thus, on the reduced document set $\mathcal{D}_I^q$), respectively. We measure it as the time elapsed from the moment the message with the query was retrieved from the queue service to the moment the message was deleted from it.

7.2 Cloud services costs

We list here the costs spelled out in the the cloud service provider's pricing policy, which impact the costs charged by the provider for our Web data management application.

File storage costs. We consider the following three components for calculating costs associated to a file store.

- $ST^{\$}_{m,GB}$ is the cost charged for storing and keeping 1 GB of data in a file store for one month.
- $STput^{\$}$ is the price per document storage operation request.
- $STget^{\$}$ is the price per document retrieval operation request.

Indexing costs. We consider the following components for calculating the index store associated costs.

- $IDX^{\$}_{m,GB}$ is the cost charged for storing and keeping 1 GB of data in the index store for one month.
- $IDXput^{\$}$ is the cost of a *put* API request that inserts a row into the index store.
- $IDXget^{\$}$ is the cost of a *get* API request that retrieves a row from the index store.

Virtual instance costs. $VM^{\$}_h$ is the price charged for running for one hour a virtual machine. This price depends on the kind of virtual machine.

Queue service costs. $QS^{\$}$ is the price charged for each request to the queue service API, e.g., send message, receive message, delete message, renew lease etc.

Data transfer. The commercial cloud providers considered in this work do not charge anything for data transferred to or within their cloud infrastructure. However, data transferred out of the cloud incurs a cost: $egress^{\$}_{GB}$ is the price charged for transferring 1 GB.

Concrete AWS costs vary depending on the geographic region where AWS hosts the application. Our experiments took place in the Asia Pacific (Singapore) AWS facility, and the respective prices as of September-October 2012 are collected in Table 3. Note that virtual machine (instance) costs are provided for two kinds of instances, "large" ($VM^{\$}_{h,l}$) and "extra-large" ($VM^{\$}_{h,xl}$).

7.3 Web data management costs

We now show how to compute, based on the data-, index- and query-driven metrics (Section 7.1), together with the cloud service costs (Section 7.2), the costs incurred by our Web data storage architecture in a commercial cloud.

Indexing and storing the data. Given a set of documents $\mathcal{D}$, we calculate the cost of uploading it a file store as follows:

$$ud^{\$}(\mathcal{D}) = STput^{\$} \times |\mathcal{D}| + QS^{\$} \times |\mathcal{D}|$$

Thus, the cost of building the index for $\mathcal{D}$ by means of the indexing strategy I is:

$$ci^{\$}(\mathcal{D}, I) = ud^{\$}(\mathcal{D}) + IDXput^{\$} \times |op(\mathcal{D}, I)| + STget^{\$} \times |\mathcal{D}| + VM_h^{\$} \times t_{idx}(\mathcal{D}, I) + QS^{\$} \times 2 \times |\mathcal{D}|$$

Note that we need two queue service requests for each document: the first obtains the URI of the document that needs to be processed, while the second deletes the message from the queue when the document has been indexed.

The cost for storing $\mathcal{D}$ in the file store and the index structure created for $\mathcal{D}$ according to I in the index store for one month is calculated as:

$$st_m^{\$}(\mathcal{D}, I) = ST_{m,GB}^{\$} \times s(\mathcal{D}) + IDX_{m,GB}^{\$} \times s(\mathcal{D}, I)$$

Querying. First, we estimate the cost incurred by the front-end for sending query q and retrieving its results as:

$$rq^{\$}(q) = STget^{\$} + egress_{GB}^{\$} \times |r(q)| + QS^{\$} \times 3$$

Three queue service requests are issued: the first one sends the query, the second one retrieves the reference to the query results, and the third one deletes the message retrieved by the second request.

The cost for answering a query q without using any index is calculated as follows:

$$cq^{\$}(q, \mathcal{D}) = rq^{\$}(q) + STget^{\$} \times |\mathcal{D}| + STput^{\$} + VM_h^{\$} \times pt(q, \mathcal{D}) + QS^{\$} \times 3$$

Note that, again, three queue service requests are issued: the first one retrieves the message containing the query, the second one sends the message with the reference to the results for the query, while the third removes the message with the query from the corresponding queue. The same holds for the formula below which calculates the cost of evaluating a query q over $\mathcal{D}$ indexed according to I:

$$cq^{\$}(q, \mathcal{D}, I, \mathcal{D}_I^q) = rq^{\$}(q) + IDXget^{\$} \times |op(q, \mathcal{D}, I)| + STget^{\$} \times |\mathcal{D}_I^q| + STput^{\$} + VM_h^{\$} \times ptq(q, \mathcal{D}, I, \mathcal{D}_I^q) + QS^{\$} \times 3$$

This finalizes our monetary cost model for Web data stores in commercial clouds, according to the architecture we described in Section 3. Some parameters of the cost model are well-known (those determined by the input data size and the provider's cost policy), while others are query- and strategy-dependent (e.g., how may documents match a query etc.) In Section 8 we measure actual charged costs, where the query- and strategy-dependent parameters are instantiated to concrete operations. These measures allow to highlight the cost savings brought by our indexing.

Indexing strategy	Average extraction time (hh:mm)	Average uploading time (hh:mm)	Total time (hh:mm)
LU	0:24	1:33	2:11
LUP	0:32	3:47	4:25
LUI	0:41	2:31	3:22
2LUPI	1:13	6:30	7:46

Table 4: Indexing times using 8 large (L) instances.

8. EXPERIMENTAL RESULTS

This section describes the experimental environment and results obtained. Section 8.1 describes the experimental setup, Section 8.2 reports our performance results, and finally, Section 8.3 presents our cost study.

8.1 Experimental setup

Our experiments ran on AWS servers from the Asia Pacific region in September-October 2012. We used the (centralized) Java-based XML query processor developed within our ViP2P project [16], implementing an extension of the algorithm of [9] to our larger subset of XQuery. On this dialect, our experiments have shown that ViP2P's performance is close to (or better than) Saxon-B v9.1[2].

We use two types of EC2 instances for running the indexing module and query processor:

- **Large (L)**, with 7.5 GB of RAM memory and 2 virtual cores with 2 EC2 Compute Units each.
- **Extra large (XL)**, with 15 GB of RAM memory and 4 virtual cores with 2 EC2 Compute Units each.

An EC2 Compute Unit is equivalent to the CPU capacity of a 1.0-1.2 GHz 2007 Xeon processor.

To test index selectivity, we needed an XML corpus with some heterogeneity. We generated XMark [24] documents (20000 documents in all, adding up to 40 GB), using the *split option* provided by the data generator[3]. We modified a fraction of the documents to alter their *path* structure (while preserving their labels), and modified another fraction to make them "more" heterogeneous than the original documents, by rendering more elements optional children of their parents, whereas they were compulsory in XMark.

8.2 Performance study

XML indexing. To test the performance of index creation, the documents were initially stored in S3, from which they were gathered in batches by multiple L instances running the indexing module. We batched the documents in order to minimize the number of calls needed to load the index into DynamoDB. Moreover, we used L instances because in our configuration, DynamoDB was the bottleneck while indexing. Thus, using more powerful XL instances could not have increased the throughput.

Table 4 shows the time spent extracting index entries on 8 L EC2 instances and uploading the index to DynamoDB using each proposed strategy. We show the average time spent by each EC2 machine to extract the entries, the average time spent by DynamoDB to load the index data, and the total observed time (elapsed between the beginning and the end of the overall indexing process). As expected, the more and

[2] `http://saxon.sourceforge.net/`

[3] `http://www.xml-benchmark.org/faq.txt`

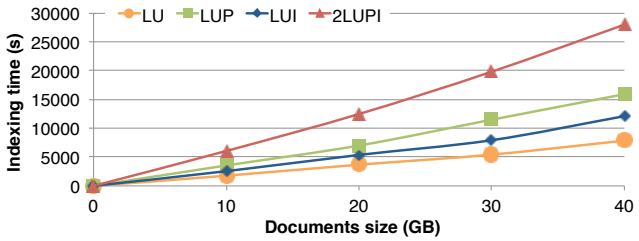


Figure 7: Indexing in 8 large (L) EC2 instances.

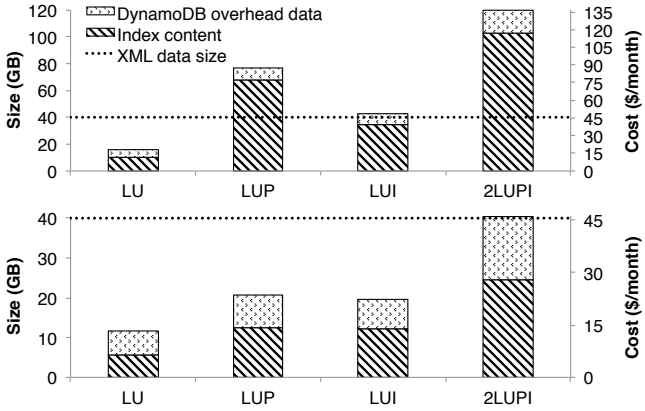


Figure 8: Index size and storage costs per month with full-text indexing (top) and without (bottom).

Query	# Doc. IDs from index				# Docs. w. results	Results size (KB)
	LU	*LUP*	*LUI*	*2LUPI*		
q_1	3	2	1	1	1	0.04
q_2	523	349	349	349	349	94000.00
q_3	144	66	33	33	33	52400.00
q_4	1089	1089	775	775	775	519.20
q_5	1115	740	370	370	370	7500.00
q_6	285	283	283	283	283	278.20
q_7	285	283	142	142	142	96.20
q_8	1400	1025	882	882	507	13800.00
q_9	1115	740	740	740	740	338800.00
q_{10}	1400	1025	512	512	116	9.10

Table 5: Query processing details (20000 documents).

the larger the entries a strategy produces, the longer indexing takes. Next, Figure 7 shows that indexing time scales well, linearly in the size of the data for each strategy.

A different perspective on the indexing is given in Figure 8 which shows the size of the index entries, compared with the original XML size. In addition to the full-text indexes size, the figure includes the size for each strategy if keywords are not stored. As expected, the index for the latter strategies is quite smaller than the full-text variant. *LUP* and *2LUPI* are the larger indexes, and in particular, if we index the keywords, the index is quite larger than the data. The *LUI* index is smaller than the *LUP* one, because IDs are more compact than paths; moreover, we exploit the fact that DynamoDB allows storing arbitrary binary objects, to store compressed (encoded) sets of IDs in a single DynamoDB value. Finally, the DynamoDB space overhead (Section 7) is noticeable, especially if keywords are not indexed, but in both variants grows slower than the index size.

XML query processing. We now study the query processing performance, using 10 queries from the XMark benchmark; they can be found in the Appendix. The queries have an average of ten nodes each; the last three queries feature value joins.

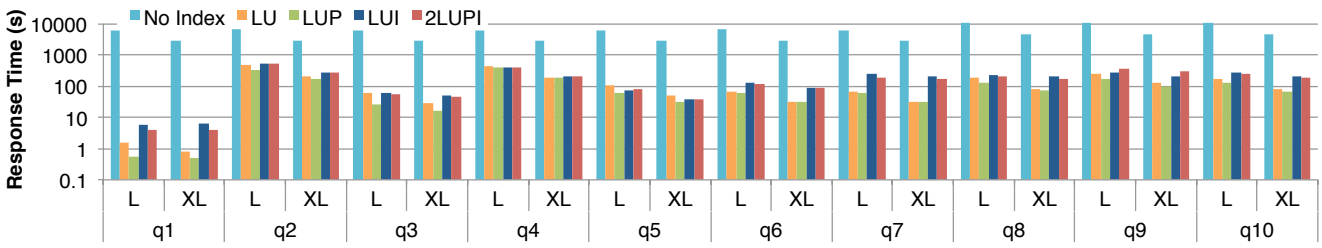
Table 5 shows, for each query and indexing strategy, the number of documents retrieved by index look-up, the number of documents which actually contain query results, and the result size for each query. (These are obviously independent of the platform, and we provide them only to help interpret our next results.) Table 5 shows that *LUI* and *2LUPI* are exact for queries q_1-q_7, which are tree pattern queries (the look-up in the index returns no false positive in these cases). The imprecision of *LU* and *LUP* varies across the queries, but it may reach 200% (twice as many false positives, as there are documents with results), even for tree pattern queries like q_5. For the last three queries, featuring value joins, even *LUI* and *LUPI* may bring false positives. For these queries, Table 5 sums the numbers of document IDs retrieved for each tree pattern.

The response times (perceived by the user) for each query, using each indexing strategy, and also without any index, is shown in Figure 9a; note the logarithmic y axis. We have evaluated the workload using L and then, separately, XL EC2 instances. We see that all indexes considerably speed up each query, by one or two orders of magnitude in most cases. Figure 9a also demonstrates that our strategies are able to take advantage of more powerful EC2 instances, that is, for every query, the XL running times are shorter than the times using an L instance. The strategy with the shortest evaluation time is *LUP*, which strikes a good balance between precision and efficiency; most of the time, *LU* is next, followed by *LUI* and *2LUPI* (recall again that the y axis is log-scale). The difference between the slowest and fastest strategy is a factor of 4 at most whereas the difference between the fastest index and no index is of 20 at least.

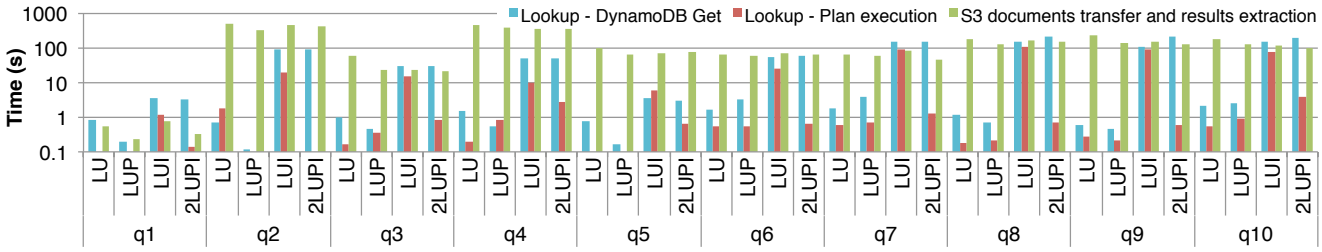
The charts in Figures 9b and 9c provide more insight. They split query processing time into: the time to consult the index (DynamoDB *get*), the time to run the physical plan identifying the relevant document URIs out of the data retrieved from the index, and the time to fetch the documents from S3 into EC2 and evaluate the queries there. *Importantly, since we make use of the multi-core capabilities of EC2 virtual machines, the times individually reported in Figures 9b and 9c were in fact measured in parallel.* In other words, the overall response time observed and reported in Figure 9a is systematically less than the sum of the detailed times reported in Figures 9b and 9c.

We see that low-granularity indexing strategies (*LU* and *LUP*) have systematically shorter index look-up and index post-processing times than the fine-granularity ones (*LUI* and *2LUPI*). The times to transfer the relevant documents to EC2 and evaluate queries there, is proportional to the number of documents retrieved from the index look-ups (these numbers are provided in Table 5). For a given query, the document transfer + query evaluation time differs between the strategies by the factor of up to 3, corresponding to the number of documents retrieved.

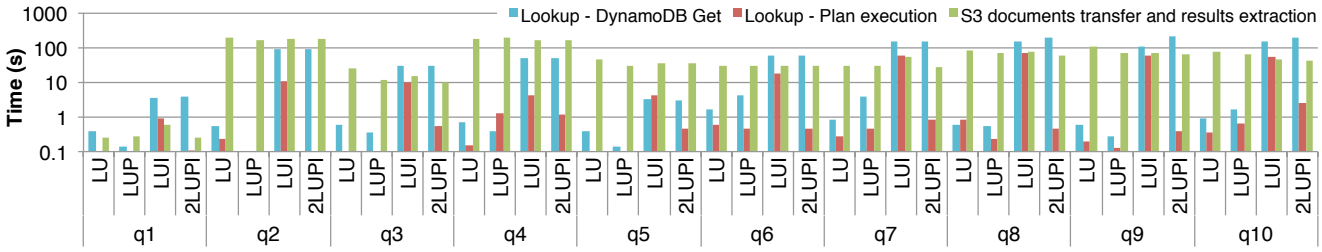
Impact of parallelism. Figure 10 shows how the query response time varies when running multiple EC2 query processing instances. To this purpose, we sent to the front-end all our workload queries, successively, 16 times: $q_1, q_2, \ldots, q_{10}$, $q_1, q_2, \ldots$ etc. We report the running times on a single EC2 instance (no parallelism) versus running times on eight EC2 instances in parallel. We can see that more instances significantly reduce the running time, more so for L instances than



(a) Response time using large (L) and extra large (XL) EC2 instances on the 40 GB database



(b) Detail large (L) EC2 instance on the 40 GB database



(c) Detail extra large (XL) EC2 instance on the 40 GB database

Figure 9: Response time (top) and details (middle and bottom) for each query and indexing strategy.

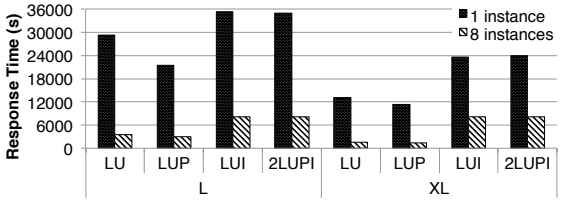


Figure 10: Impact of using multiple EC2 instances.

Indexing strategy	DynamoDB	EC2	S3 + SQS	Total
LU	$21.15	$5.47	$0.02	$26.64
LUP	$44.78	$11.95		$56.75
LUI	$33.47	$8.95		$42.44
2LUPI	$78.25	$21.17		$99.44

Table 6: Indexing costs for 40 GB using L instances.

for XL ones: this is because many strong instances sending indexing requests in parallel come close to saturating DynamoDB's capacity of absorbing them.

8.3 Amazon charged costs

We now study the costs charged by AWS for indexing the data, and for answering queries. We also consider the amortization of the index, i.e., when query cost savings brought by the index balance the cost of the index itself.

Indexing cost. Table 6 shows the monetary costs for indexing data according to each strategy. These costs are broken down across the specific AWS services. The most costly index to build is *2LUPI*, while the cheapest is *LU*. The combined price for S3 and SQS is constant across strategies, and is negligible compared to EC2 costs. In turn, the EC2 cost is dominated by the DynamoDB cost in all strategies.

Finally, Figure 8 shows the storage cost per month of each index, which is proportional to its size in DynamoDB.

Query processing cost. Figure 11 shows the cost of answering each query when using no index, and when using the different indexing strategies. Note that using indexes, the cost is practically independent of the machine type. This is because (*i*) the hourly cost for an XL machine is double than that of a L machine; but at the same time (*ii*) the four cores of an XL machine allow processing queries simultaneously on twice as many documents as the two cores of L machines, so that the *cost* differences pretty much cancel each other (whereas the *time* differences are noticeable). Figure 11 also shows that *indexing significantly reduces monetary costs* compared to the case where no index is used; the savings vary between 92% and 97%.

To better understand the monetary costs shown in Figure 11, we provide the details of evaluating the query workload on an XL instance in Figure 12, again decomposed across the services we use, to which we add *AWSDown*, the price charged by Amazon for transferring query results out of AWS. *AWSDown* cost is the same for all strategies, since the same results are obtained. S3 cost is proportional to the selectivity of the index strategy (recall Table 5). DynamoDB costs reflect the amount of data extracted for each strategy from the index, and finally, EC2 cost is proportional to the time necessary to answer the workload using each strategy, which means that the shorter it takes to answer a query, the

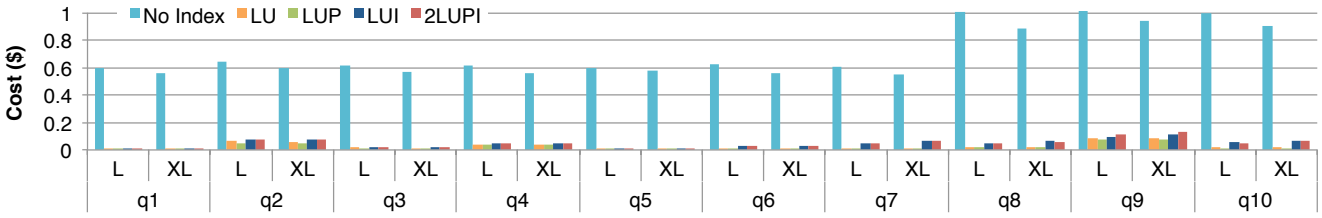


Figure 11: Query processing costs decomposition.

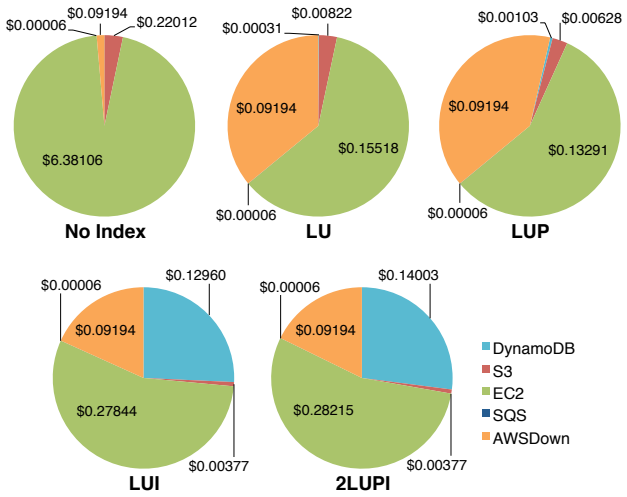


Figure 12: Workload evaluation cost details on an extra large (XL) instance.

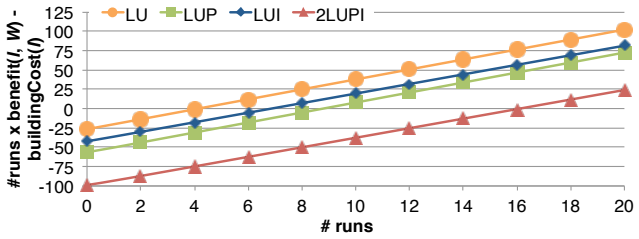


Figure 13: Index cost amortization for a single extra large (XL) EC2 instance.

lower it will be its cost. For every strategy, the cost of using EC2 clearly dominates, which is expected and desired, since this is the time actually spent processing the query.

Amortization of the index costs. We now study how indexing pays off when evaluating queries. For an indexing strategy I and workload W, we term *benefit* of I for W the difference between the monetary cost to answer W using no index, and the cost to answer W based on the index built according to I. At each run of W, we "save" this benefit, whereas we had to pay a certain cost to build I. (The index costs and benefits also depend on the data set, and increase with its size.) Figure 13 shows when the cumulated benefit (over several runs of the workload on a L instance) overweighs the index building cost (similarly, on a single L instance, recall Table 6). Figure 13 shows that any of our strategies allows recovering the index creation costs quite fast, i.e. just running the workload 4 times for *LU*, 8 times for *LUP* and *LUI*, and *16* times for *2LUPI* respectively. (The cost is recovered when the curves cross the $Y = 0$ axis.)

Indexing strategy	Indexing speed (ms/MB of XML data)		Indexing cost ($/MB of XML data)	
	[8]	This work	[8]	This work
LU	7491	196	0.019	0.00067
LUP	8335	398	0.057	0.00142
LUI	12447	302	0.021	0.00106
2LUPI	11265	699	0.070	0.00249

Monthly storage cost ($/GB of XML data)

Index, [8]	Index, this work	Data, [8] and this work
0.275	1.14	0.125

Table 7: Indexing comparison.

Indexing strategy	Query speed (ms/MB of XML data)		Query costs ($/MB of XML data)	
	[8]	This work	[8]	This work
LU	141	21	4.7×10^{-5}	0.6×10^{-5}
LUP	121	18	4.2×10^{-5}	0.6×10^{-5}
LUI	186	37	5.6×10^{-5}	1.3×10^{-5}
2LUPI	164	37	5.4×10^{-5}	1.3×10^{-5}

Table 8: Query processing comparison.

8.4 Comparison with previous works

The closest related works are [6, 19] and our previous work [8], which build database services on top of a commercial cloud, and in particular AWS.

The focus in [6] was on implementing transactions in the cloud with various consistency models; they present experiments on the TPC-W relational benchmark, using 10.000 products (thus, a database of 315 MB of data, *about 125 times smaller than ours*). The setting thus is quite different, but a rough comparison can still be done.

At that time, Amazon did not provide a key-value store, therefore the authors built B+ trees indexes within S3. Each *transaction* in [6] retrieves one customer record, searches for six products, and orders three of them. These are all *selective (point) queries (and updates)*, thus, they only compare with q_1 among our queries, the only one which can be assimilated to a point query (very selective path matched in few documents, recall Table 5). For a *transaction*, they report running times between 2.8 and 11.3 seconds, while individual queries/updates are said to last less than a second. Our q_1, running in 0.5 seconds (using one instance) on our 40 GB database of data, is thus quite competitive. Moreover, [6] reports transaction costs between $\$1.5 \times 10^{-4}$ and $\$2.9 \times 10^{-3}$, very close to our $\$1.2 \times 10^{-4}$ cost of q_1 using *LUP*.

Next, we compare with our previous work [8], evaluated on only 1 GB of XML data. From a performance perspective, the main difference is that [8] stores the index within AWS' previous key-value store, namely SimpleDB. Table 7 compares this work with [8] from the perspective of indexing, and Table 8 from that of querying; for a fair comparison, we report measures *per MB (or GB)* of data.

The tables show that the present work *speeds up index-*

ing by one to two orders of magnitude, all the while *indexing costs are reduced by two to three orders of magnitude*; *querying is faster (and query costs lower) by a factor of five* (roughly) wrt [8]. The reason is that DynamoDB allows storing arbitrary binary objects as values, a feature we exploited in order to efficiently encode our index data. Moreover, DynamoDB has a shorter response time and can handle more concurrent requests than SimpleDB.

The authors of [6] subsequently also ported their index to SimpleDB [19]. Still on TPC-W transactions on a 10.000 items database, processing with a SimpleDB index was moderately faster (by a factor of less than 2) than by using their previous S3-based one. As we have seen above, DynamoDB significantly outperforms SimpleDB when indexing.

8.5 Experiments conclusion

Our experiments demonstrate the feasibility and interest of our architecture based on a commercial cloud, using a distributed file system to store XML data, a key-value store to store the index, and the cloud's processing engines to index and process queries. All our indexing strategies have been shown to reduce query response time *and* monetary cost, by 2 orders of magnitude in our experiments; moreover, our architecture is capable of scaling up as more instances are added. The monetary costs of query processing are shown to be quite competitive, compared with previous similar works [6, 8], and we have shown that the overhead of building and maintaining the index is modest, and quickly offset by the cost savings due to the ability to narrow the query to only a subset of the documents. In our tests, the *LUP* indexing strategy allowed for the most efficient query processing, at the expense of an index size somehow larger than the data. Further compression of the paths in the *LUP* index could probably make it even more competitive.

In our experiments, query execution based on the LU and LUP strategies is always faster than using the LUI and 2LUPI strategies. We believe that cases for which LUI and 2LUPI strategies behave better are those in which query tree patterns are multi-branched, highly selective and evaluated over a document set where most of the documents *only* match linear paths of the query. Such cases can be statically detected by using data summaries and some statistical information. We postpone this study to future work.

9. CONCLUSION

We have presented an architecture for building scalable XML warehouses by means of commercial cloud resources, which can exploit parallelism to speed up index building and query processing. We have investigated and compared through experiments several indexing strategies and shown that they achieve query processing speed-up and monetary costs reductions of several orders of magnitude within AWS.

Our future works include the development of a *platform and index advisor tool*, which based on the expected dataset and workload, estimates an application's performance and cost and picks the best indexing strategy to use.

Acknowledgments

This work has been partially funded by the KIC EIT ICT Labs activities 11803, 11880 and RCLD 12115, as well as an AWS in Education research grant.

10. REFERENCES

[1] S. Abiteboul, S. Cluet, G. Ferran, and M.-C. Rousset. The Xyleme project. *Computer Networks*, 39(3), 2002.

[2] S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. XML processing in DHT networks. In *ICDE*, 2008.

[3] 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.

[4] A. Aranda-Andújar, F. Bugiotti, J. Camacho-Rodríguez, D. Colazzo, F. Goasdoué, Z. Kaoudi, and I. Manolescu. AMADA: Web Data Repositories in the Amazon Cloud (demo). In *CIKM*, 2012.

[5] 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.

[6] M. Brantner, D. Florescu, D. A. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD*, 2008.

[7] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.

[8] J. Camacho-Rodríguez, D. Colazzo, and I. Manolescu. Building Large XML Stores in the Amazon Cloud. In *DMC Workshop (collocated with ICDE)*, 2012.

[9] Y. Chen, S. B. Davidson, and Y. Zheng. An Efficient XPath Query Processor for XML Streams. In *ICDE*, 2006.

[10] H. Choi, K.-H. Lee, S.-H. Kim, Y.-J. Lee, and B. Moon. HadoopXML: A Suite for Parallel Processing of Massive XML Data with Multiple Twig Pattern Queries (demo). In *CIKM*, 2012.

[11] L. Fegaras, C. Li, U. Gupta, and J. Philip. XML Query Optimization in Map-Reduce. In *WebDB*, 2011.

[12] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *VLDB*, 2003.

[13] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, 1997.

[14] V. Kantere, D. Dash, G. François, S. Kyriakopoulou, and A. Ailamaki. Optimal service pricing for a cloud cache. *IEEE Trans. Knowl. Data Eng.*, 23(9), 2011.

[15] V. Kantere, D. Dash, G. Gratsias, and A. Ailamaki. Predicting cost amortization for query services. In *SIGMOD*, 2011.

[16] K. Karanasos, A. Katsifodimos, I. Manolescu, and S. Zoupanos. The ViP2P Platform: XML Views in P2P. In *ICWE*, 2012.

[17] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD*, 2002.

[18] G. Koloniari and E. Pitoura. Peer-to-peer management of XML data: issues and research challenges. *SIGMOD Record*, 34(2), 2005.

[19] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *SIGMOD*, 2010.

[20] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. IETF RFC 4122, July 2005.

[21] I. Manolescu, K. Karanasos, V. Vassalos, and S. Zoupanos. Efficient XQuery rewriting using multiple views. In *ICDE*, 2011.

[22] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[23] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *PVLDB*, 3(1), 2010.

[24] 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.

APPENDIX

We provide here additional details on our experiments.

A. QUERY WORKLOAD DETAILS

Figure 14 depicts the workload used in the experimental section; it consists of 10 queries with an average of 10 nodes each. The queries are taken from the XMark benchmark. Three workload queries contained value joins, the other each correspond to a tree pattern. Furthermore, query q_1 is very selective (point query); query q_4 uses a full-text search predicate.

B. EXPERIMENTAL RESULTS WITHOUT KEYWORDS

Indexing strategy	Average extraction time (hh:mm)	Average uploading time (hh:mm)	Total time (hh:mm)
LU	0:05	1:32	1:44
LUP	0:06	2:08	2:20
LUI	0:09	2:13	2:28
2LUPI	0:15	4:20	4:46

Table 9: Indexing times details using 8 large (L) EC2 instances (40 GB database).

Query	# Doc. IDs from index				# Docs. w. results	Results size (KB)
	LU	*LUP*	*LUI*	*2LUPI*		
q_1	3	2	1	1	1	0.04
q_2	523	349	349	349	349	94000.00
q_3	144	66	33	33	33	52400.00
q_4	1089	1089	775	775	775	519.20
q_5	1115	740	370	370	370	7500.00
q_6	285	283	283	283	283	278.20
q_7	285	283	142	142	142	96.20
q_8	1400	1025	882	882	507	13800.00
q_9	1115	740	740	740	740	338800.00
q_{10}	1400	1025	512	512	116	9.10

Table 10: Queries response detail without using keywords (20.000 documents).

Indexing strategy	DynamoDB	EC2	S3 + SQS	Total
LU	19.29	4.56	0.02	23.86
LUP	26.32	6.26		32.60
LUI	25.38	6.61		32.00
2LUPI	51.70	12.93		64.64

Table 11: Cost detail [$] for indexing the 40 GB database using large (L) EC2 instance(s).

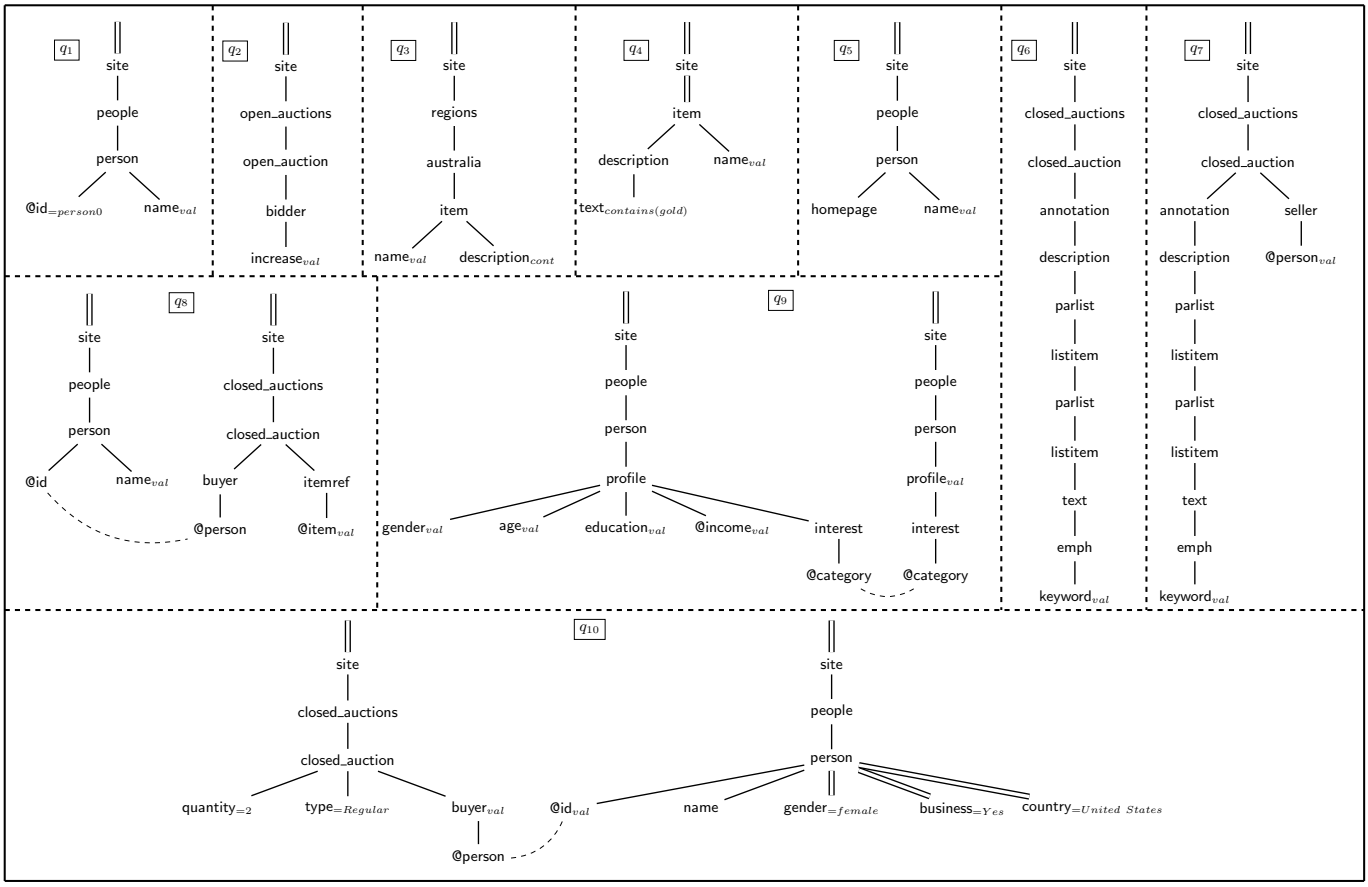


Figure 14: Query workload used in the experimental section.

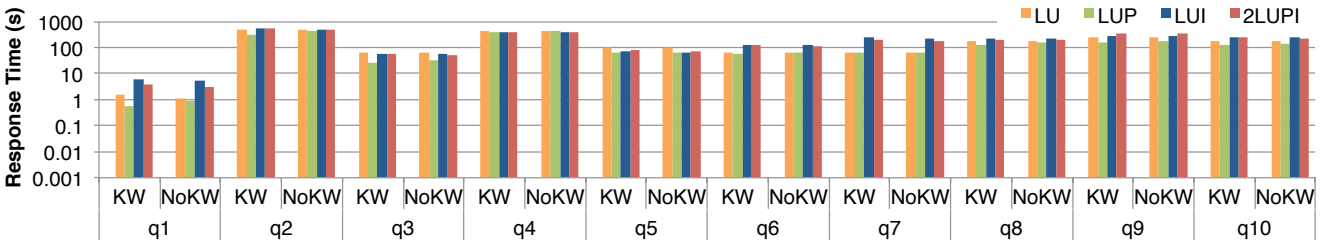


Figure 15: Response time using large (L) EC2 instances on the 40 GB database with and without using keywords (KW and NoKW, respectively).