AMADA: Web Data Repositories in the Amazon Cloud

Andrés Aranda-Andújar[1] Francesca Bugiotti[1,3] Jesús Camacho-Rodríguez[1,2]
Dario Colazzo[1,2] François Goasdoué[1,2] Zoi Kaoudi[1,2] Ioana Manolescu[1,2]
[1]Inria Saclay–Île-de-France, France [2]Université Paris-Sud, France [3]Università Roma Tre, Italy
firstname.lastname@inria.fr

ABSTRACT

We present AMADA, a platform for storing Web data (in particular, XML documents and RDF graphs) based on the Amazon Web Services (AWS) cloud infrastructure. AMADA operates in a Software as a Service (SaaS) approach, allowing users to upload, index, store, and query large volumes of Web data. The demonstration shows (*i*) the step-by-step procedure for building and exploiting the warehouse (storing, indexing, querying) and (*ii*) the monitoring tools enabling one to control the expenses (monetary costs) charged by AWS for the operations involved while running AMADA.

Categories and Subject Descriptors

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

General Terms

Design, Experimentation, Performance, Economics

Keywords

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

1. INTRODUCTION

Increasing volumes of data are produced or exported into Web data formats. Among these, the W3C's XML standard for structured documents (and in particular Web pages) and RDF for Semantic Web data are the best known. XML allows encoding complex documents whose structure may be constrained by an XML Schema. RDF graphs consist of triples of the form (s, p, o) stating that the subject node s has the property edge p whose value is the object node o. XML takes the lion's share of Web content nowadays. At the same time, RDF is increasingly being used in numerous data sources such as in Linked Open Data.

To exploit large volumes of Web data, an interesting option is to warehouse it into a single access point repository. This typically involves some crawling or other means of identifying interesting data sources and loading these data sources into the repository where further processing can be applied. Huge data volumes have raised the need for distributed storage architectures and platforms typically deployed in a cloud environment, which provides scalable and elastic resource allocation. *In this work, we consider hosting large volumes of Web data in the cloud, and their efficient storage and querying through a (distributed, parallel) platform also running in the cloud.* Such an architecture belongs to the general Software as a Service (SaaS) setting where the whole stack from the hardware to the data management layer are hosted and rented from the cloud.

In this context, an important challenge is *how to efficiently identify the parts of the data which need to be consulted in order to answer a given query.* In a cloud SaaS setting, efficient access path selection not only speeds up query processing, but also helps to reduce the monetary costs charged for querying, by avoiding work on some cloud machines that do not end up producing query results.

AMADA is a scalable platform for Web data management within the cloud, with a particular focus on this cloud-based data access path selection challenge [3, 4]. AMADA stores Web data in the cloud, and establishes indexes at various granularity over the data. Following other works [2, 11], AMADA uses the Amazon Web Services (AWS) cloud (`http://aws.amazon.com`), among the most prominent ones today. An important AWS feature is its elasticity, i.e., the ability to smoothly allocate computing power, storage, or other services, as the application demands vary.

The contribution of AMADA is twofold. First, AMADA presents a novel architecture harnessing the various subsystems of the popular cloud platform AWS for higher-level, efficient management of complex data. Second, AMADA includes an index-based mechanism for access path selection within a cloud-based repository, reducing query processing time as well as the warehouse operating costs.

The remainder of this work is structured as follows. Section 2 describes AMADA architecture while Section 3 outlines our Web data indexing algorithms. We present the demonstrated features in Section 4. Finally, we briefly discuss other related works in Section 5 and conclude in Section 6.

2. ARCHITECTURE

The design of AMADA was guided by the following objectives. First, we aimed to leverage AWS resources by scaling up to large data volumes. Second, we aimed 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, i.e., the total work required for our operations. This is all the more important since in a cloud, total work translates into monetary costs. Fourth, the architecture should not be too closely tied to a specific XML or RDF query processor.

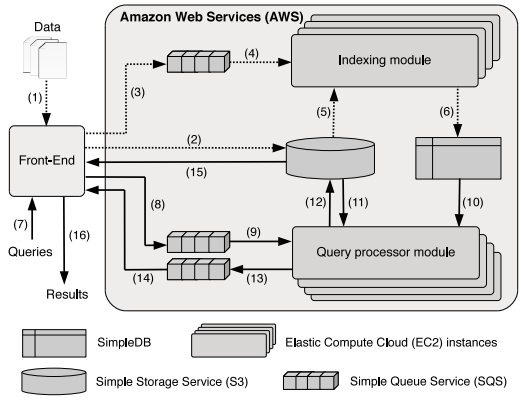

Figure 1: System architecture.

AMADA stores data and indexes in a distributed fashion within AWS, but the main ideas of our work can be translated to other platforms. *Data* is stored within Amazon Simple Storage Service (S3), which is the AWS store for large objects. S3 assigns to each dataset a URI, based on which it can be retrieved. We build *data indexes* within SimpleDB, a simple database system supporting SQL-style queries based on a key-value model. Observe that while SimpleDB generalizes relational databases by supporting heterogeneous tuple and multi-valued attributes, it is more restricted in that it only supports single-relation queries, i.e., no joins. We have implemented several indexing strategies for both XML and RDF, differing in the choice of index keys, and in their level of detail, i.e., whether they point to specific datasets, or to very fine-grained data items within the datasets. The code which processes queries runs on virtual machines within the Amazon Elastic Compute Cloud (EC2). Finally, in order to synchronize the distributed components of our system, we use Amazon Simple Queue Service (SQS), which provides asynchronous message-based communication.

Figure 1 gives an in-depth view of AMADA's architecture. A dataset submitted to the *front-end module* is stored as a file in S3, whose URI is sent to an indexing module running on an EC2 instance. This module retrieves the corresponding dataset from S3 and builds an index that is stored in SimpleDB. A query submitted to the *front-end module* is sent to a query processor module running on an EC2 instance. This module performs a look-up to the indexes in SimpleDB so as to find out the relevant datasets for answering the query, and evaluates the query against them. Results are written in a file stored in S3, whose URI is sent to the *front-end module* to retrieve the query answers.

Scalability, parallelism and fault-tolerance AMADA exploits the elastic scaling of AWS by increasing and decreasing the number of EC2 instances running each module. The synchronization through the SQS message queues among modules supports inter-machine parallelism, whereas intra-machine parallelism is supported by multi-threading our code. AMADA also provides fault-tolerance by using the queues. If an instance crashes while loading a document or answering the query, the message which had caused the work to start becomes available again in the queue, and thus another virtual instance will retrieve it and take over the job.

AMADA is implemented in Java and uses AWS SDK for Java. For processing XML queries within EC2, it uses the query processor developed within our ViP2P project [13], while for RDF queries, AMADA uses RDF-3X [12].

3. WEB DATA INDEXING STRATEGIES

An important feature of AMADA is data indexing within SimpleDB. The structure of SimpleDB is as follows.

$$\begin{array}{lcl} \mathit{database} & = & \mathit{domain}^+ \\ \mathit{domain} & = & (\mathit{name}, \mathit{item}^+) \\ \mathit{item} & = & (\mathit{key}, \mathit{attribute}^+) \\ \mathit{attribute} & = & (\mathit{name}, \mathit{value}) \end{array}$$

SimpleDB data is organized in *domains*. Each domain is a collection of *items* identified by their *key*. In turn, each item has one or more *attributes*; an attribute has a *name*, and one or several *values*. An attribute value may be empty (denoted by ϵ). Different items within a SimpleDB domain may have different attribute names.

The SimpleDB API provides a *get(d,k)* operation retrieving all items in the domain d having the key k, and a *put* operation to set values of attributes: *put(d,k,(a,v)+)* inserts the attributes *(a,v)+* into an item with key k in domain d. A *batchPut* variant inserts 25 items at a time and leads to a better performance. AWS ensures that queries to different SimpleDB domains run in parallel.

Indexing strategies Conceptually, given a data model $\mathcal{M}$, an indexing strategy $\mathcal{I}$ is a function extracting quadruplets of the form (*domain name*, *item key*, *attribute name*, *attribute value*) from an input dataset D. Indexing D according to $\mathcal{I}$, then, amounts to (i) computing the quadruplets in $\mathcal{I}(D)$ and (ii) adding these quadruplets to SimpleDB, using appropriate (batched, sometimes conditional) *put* operations. AMADA implements four indexing strategies for XML and four for RDF, detailed in [4] and [3], respectively. In the following, we present only some of them.

XML indexing The XML indexing strategy Label-URI (or LU, in short) computes quadruplets in which:

- the domain is assigned using a hashing-based mapping technique [4];
- for each element e whose name is n_e, the string $\underline{e}n_e$ is an item key, i.e., we concatenate a token $\underline{e}$ indicating that this is an element, to the element tag n_e. Similarly, for each attribute a with name n_a, $\underline{a}n_a$ is an item key, while another key $\underline{a}n_a v_a$ reflects its value v_a. We proceed similarly for every word w of a text node.
- each item thus obtained has an attribute whose name is the URI of D, and whose value is empty (ϵ).

Given an XML query, strategy LU leads to the set of documents featuring each element and attribute name, value, and keyword of the query; intersecting these sets leads to the URIs of documents featuring all of them. Some of the documents whose URIs are thus retrieved may not satisfy the query's structural constraints, e.g., the URI of a document of the form ⟨a⟩⟨b/⟩⟨c/⟩⟨/a⟩ will be retrieved for a query of the form /a//b/c, i.e., there are some false positives.

A different strategy is Label-URI-ID (or LUI, in short), similar to LU but using identifiers of the XML nodes instead of the ϵ attribute values of LU. LUI does not return false positives, but leads to a much larger index than LU, because it introduces e.g., 10 index entries for 10 elements labeled a in a document, whereas LU uses only one.

RDF indexing Let D be an RDF graph whose URI is U_D and (s_1, p_1, o_1) be a triple in D. Let $\underline{s}$, $\underline{p}$ and $\underline{o}$ be three distinct tokens representing subjects, properties and objects, respectively. The simplest RDF indexing strategy called ATT (for attribute-based) uses a single default domain which is split as the index grows. For (s_1, p_1, o_1), ATT builds 3 item keys: $\underline{s}s_1$, $\underline{p}p_1$, $\underline{o}o_1$. Each such item has a single

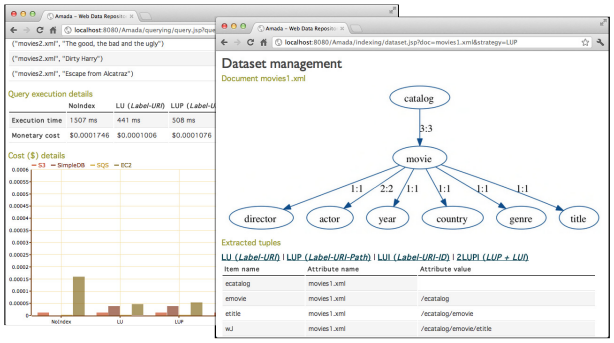

Figure 2: Demonstration interface.

attribute U_D, whose value is ϵ. Assume now that we want to evaluate the SPARQL query:

```
SELECT ?o WHERE {?s :hasAuthor "Foo" . ?s :hasTitle ?o .}
```

AMADA performs three *get* queries; one for each constant of the query: `q1` having key `hasAuthor`, `q2` having key `oFoo` and `q3` having key `hasTitle`. Then, it intersects the results of `q1` and `q2` to ensure that the `:hasAuthor` property and the `"Foo"` object value occur in the same dataset. The result is then *unioned* with the result of `q3` to obtain the datasets on which the SPARQL query will be evaluated. SPARQL semantics allows matches for this query to span over multiple datasets, i.e., query results are defined on a global "merged" graph.

Another RDF indexing strategy similar with ATT is ATS (for attribute subset) but it creates 7 item keys for each triple. Strategy ATS builds a larger index, in exchange for less *get* calls to identify the relevant graphs to a query.

Performance A detailed performance study of the XML side of AMADA is described in [14]. We compare four indexing strategies and show that indexing leads to cost savings that offset the index building and maintenance costs, in our experiments, after roughly 1000 queries [14].

4. DEMONSTRATION SCENARIO

The demonstration showcases loading and querying data in AMADA in real time within AWS. We use datasets about cultural artifacts, e.g., Open Data catalog of Bibliothèque Nationale de France (`http://data.bnf.fr/semanticweb`), as well as the general DBPedia corpus. AMADA displays in a Web-based interface (*i*) index entries to be added to SimpleDB, and (*ii*) our real-time rendition of the monetary costs entailed by index creation. Further, for a given query and indexing strategy, demo attendees will be shown (*i*) the calls to the SimpleDB API issued by the query processor to look up relevant datasets, (*ii*) the logical and physical plans for recombining (through intersection, join and/or union) the look-up results to identify the relevant datasets, and finally (*iii*) the AWS resource consumption associated to loading the respective datasets from S3 into EC2, processing the query there, and downloading the results from the cloud. AMADA includes an advisor tool, which based on the datasets and workload, provides index recommendations that depend on the relative importance given to the minimization of query response time and up-front monetary cost.

Figure 2 illustrates the interface; a more complete look-and-feel is available at `http://team.inria.fr/oak/amada/`.

5. RELATED WORK

An early work [2] has established the feasibility of building and exploiting B-tree indexes in S3 of AWS, while [11] focused on the problem of executing transactional workloads on cloud architectures, and AWS in particular. More recently, economic models for selecting indexes to materialize in a cloud were proposed in [9]. These works, however, considered regular, table-structured data, whereas we focus on irregular, tree- or graph-structured Web data.

Since the proposal of MapReduce [5] and the appearance of Hadoop, massively parallel data management using distributed infrastructures (a typical example of which are cloud-based) is a hot topic in industry and academia. For instance, XML query processing on top of Hadoop is studied in [10, 6]. Further, the increasing popularity of RDF has led to many recent works on parallelizing RDF processing using MapReduce [7, 8].

6. CONCLUSIONS AND PERSPECTIVES

AMADA exploits AWS components in order to achieve scalable storage and query processing for RDF and XML data. A main feature is content indexing, which is also the main focus of this demo proposal. Ongoing work on the platform includes (*i*) integrating content indexing within a generic, AWS-independent framework for massively parallel data management [1] and (*ii*) further investigating the impact of RDF graph partitioning on RDF query answering in a cloud context. Partitioning RDF graphs and efficiently parallelizing queries over triples entailed by the RDF semantics are open issues in this context.

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.

7. REFERENCES

[1] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: a programming model and execution framework for web-scale analytical processing. In *SoCC*, 2010.

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

[3] F. Bugiotti, F. Goasdoué, Z. Kaoudi, and I. Manolescu. RDF data management in the Amazon Cloud. In *DanaC Workshop (collocated with EDBT/ICDT)*, 2012.

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

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

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

[7] J. Huang, D. J. Abadi, and K. Ren. Scalable SPARQL querying of large RDF graphs. *PVLDB*, 4(11), 2011.

[8] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Trans. on Knowl. and Data Eng.*, 2011.

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

[10] S. Khatchadourian, M. P. Consens, and J. Siméon. Having a ChuQL at XML on the Cloud. In *A. Mendelzon Int'l. Workshop*, 2011.

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

[12] T. Neumann and G. Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *VLDBJ*, 19(1), 2010.

[13] ViP2P web site. `http://vip2p.saclay.inria.fr`.

[14] Technical report. `http://jesus.camachorodriguez.name/_media/xml-aws/tech.pdf`, 2012.