AMADA: Web Data Repositories in the Amazon Cloud

A. Aranda-Andújar[1,2] F. Bugiotti[1,3] J. Camacho-Rodríguez[1,2] Z. Kaoudi[1,2]
[1]Inria Saclay–Île-de-France, France [2]Université Paris-Sud, France [3]Universitá Roma Tré, Italy
firstname.lastname@inria.fr

Abstract

AMADA permet la gestion de données du Web dans l'architecture "cloud" d'Amazon, en particulier de documents XML et de graphes RDF. Cette plateforme est déployée comme un service permettant aux utilisateurs de charger, stocker et interroger efficacement des grands volumes de données, à l'aide d'index construits automatiquement. La démonstration illustre pas-à-pas (i) la construction et l'exploitation d'un entrepôt de données, ainsi que (ii) les outils de surveillance des coûts facturés par AWS en fonction des opérations exécutées par AMADA.

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 sometimes 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 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 [4, 5]. AMADA stores Web data in the cloud, and establishes indexes at various granularity over the data. Following other works [11, 9], 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 sub-systems 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 feature in Section 4. Finally we briefly discuss other related works and conclude in Sections 5 and 6, respectively.

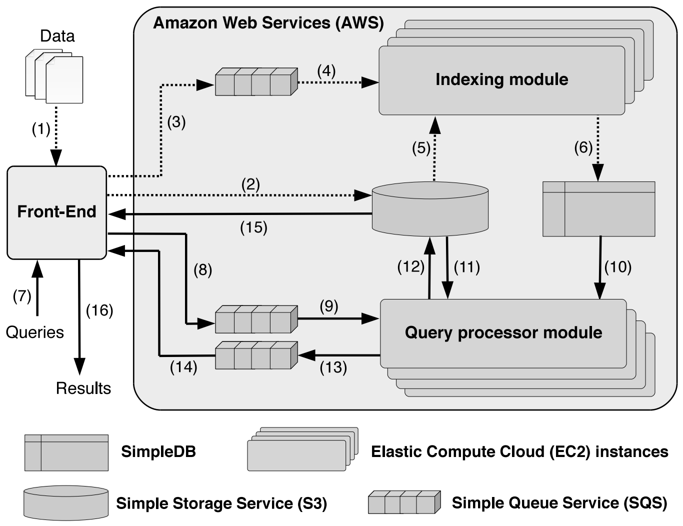


Figure 1: System architecture.

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.

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 the distributed file system S3, which is the AWS store for (very) large data. S3 assigns to each dataset a URI, based on which it can be retrieved. Furthermore, 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 actually processes queries runs on virtual machines within the Amazon Elastic Compute Cloud (EC2). Finally, in order to synchronize the distributed components of our application, 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*, so that it retrieves the query answers from S3 and returns them.

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 benefits from the fault-tolerance which AWS provides by periodically monitoring the queues. If an instance crashes while loading a document or answering the query, AWS notices that the instance has not released the SQS message which had caused the work to start. In this case, another EC2 instance will take over the job.

AMADA is implemented in Java 6 and it communicates with AWS through the Amazon Web Services

SDK for Java v1.3.6. For processing XML queries within EC2, it uses the query processor developed within our ViP2P project [10]. To process RDF queries, AMADA uses the RDF-3X system [13].

3 Web data indexing strategies

An important feature of AMADA is data indexing within SimpleDB. A description of SimpleDB's structure is helpful to understand the available options for the index and is as follows.

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

SimpleDB data is organized in *domains*. Each domain is a collection of *items* identified by their *key* (a key may be at most 1024 bytes). 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. Beyond the API, SimpleDB also provides a SQL-like higher-level language, e.g., one can use the query *select * from mydomain where Year* $>$ *'1955'* to retrieve the items matching the condition. However, *a query cannot span multiple domains*, thus joins or unions have to be coded outside SimpleDB.

AWS ensures that queries to different SimpleDB domains run in parallel, thus in general, splitting an index across domains leads to better performance.

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 RDF and four for XML, detailed in [4] and [5], respectively. In the following, we present only some of them.

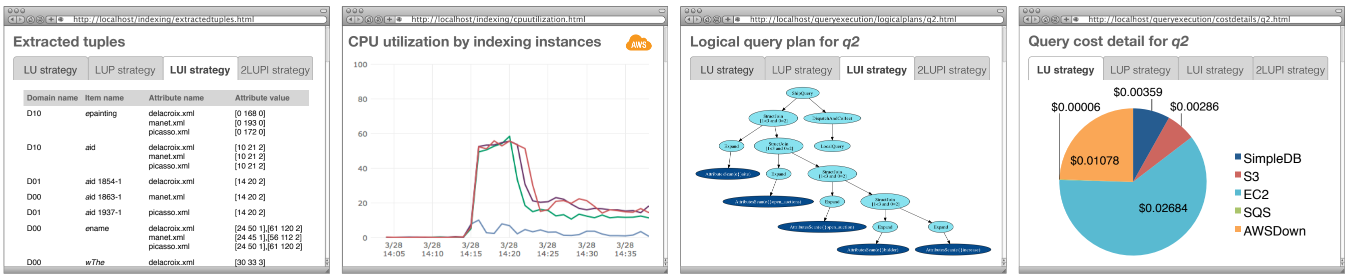
XML indexing The XML indexing strategy Label-URI (or LU, in short) computes quadruplets in which:

- all quadruplets use the same *default* domain, until it overflows, at which point we split it using a technique akin to extensible hashing [5];
- 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 whose name is n_a, $\underline{a}n_a$ is an item key, while $\underline{a}v_a$ is the key of another item reflecting the attribute value v_a. We proceed similarly for every word w occurring in 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 D 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 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 respectively objects. The simplest RDF indexing strategy called ATT (for attribute-based) uses a set of *subject domains* named $sd_1, sd_2, \ldots$; initially there is only sd_1 and as the index overgrows it, we split over multiple domains [4] as in the XML case. Similarly, ATT uses, *property domains* named $pd_1, pd_2, \ldots$ and *object domains* named $od_1, od_2, \ldots$ For (s_1, p_1, o_1),


Extracted tuples

LU strategy | LUP strategy | LUI strategy | 2LUPI strategy

Domain name	Item name	Attribute name	Attribute value
D10	*e*painting	delacroix.xml manet.xml picasso.xml	[0 168 0] [0 193 0] [0 172 0]
D10	*a*id	delacroix.xml manet.xml picasso.xml	[10 21 2] [10 21 2] [10 21 2]
D01	*a*id 1854-1	delacroix.xml	[14 20 2]
D00	*a*id 1863-1	manet.xml	[14 20 2]
D01	*a*id 1937-1	picasso.xml	[14 20 2]
D00	*e*name	delacroix.xml manet.xml picasso.xml	[24 50 1],[61 120 2] [24 45 1],[56 112 2] [24 50 1],[61 120 2]
D00	*w*The	delacroix.xml	[30 33 3]

Figure 2: Demonstration screenshots: tuples extracted using one of our indexing strategies (left); CPU utilization while indexing documents (center-left); sample logical plan (center-right); and sample query cost detail (right).

ATT builds: $(sd_i, U_D, \underline{s}, s_1)$, $(pd_j, U_D, \underline{p}, p_1)$ and $(od_k, U_D, \underline{o}, o_1)$, where: sd_i, pd_j and od_k are the domains where we currently insert resp. new subject, property, and object index entries. Based on an ATT index, the URIs of the graphs featuring the constant and URI values appearing in a query can be extracted from SimpleDB. For instance, assume that we evaluate the SPARQL query:

```
PREFIX inria: <http://inria.fr/> SELECT ?s
SELECT ?s WHERE { ?s inria:hasAuthor "Foo" .
                  ?s inria:hasContactInfo ?o . }
```

The corresponding SimpleDB look-up queries are:

```
q1: SELECT itemName() FROM pd1 WHERE property = inria:hasAuthor;
q2: SELECT itemName() FROM od1 WHERE object = "Foo";
q3: SELECT itemName() FROM pd1 WHERE property = inria:hasContactInfo;
```

AMADA intersects the results of `q1` and `q2` to ensure that the `inria:hasAuthor` property and the `"Foo"` 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. Indeed, 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 is ATS (for attribute subset). It uses a single default domain which is split as the index grows. From (s_1, p_1, o_1), ATS builds 7 item keys: $\underline{s}s_1$, $\underline{p}p_1$, $\underline{o}o_1$, $\underline{sp}s_1p_1$, $\underline{so}s_1o_1$, $\underline{po}p_1o_1$ and $\underline{spo}s_1p_1o_1$. Each such item has a single attribute whose name is U_D, and whose value is ϵ. Strategy ATS builds a larger index, in exchange for less *get* calls required to identify the relevant graphs to a query.

Efficiently building and exploiting the index Each indexing strategy cuts a specific trade-off between indexing time, indexing cost, query response time and query processing cost. To reach interesting trade-offs, we have implemented several optimizations, e.g., efficient encoding of sorted series of XML node IDs within single attribute values, or early-stop techniques in the joins and intersection physical operators combining the results of index look-ups. By using these techniques, indexing leads to cost savings that offset the index building and maintenance costs, in our experiments, after roughly 1.000 queries [14]. When the repository grows larger and/or more diverse, index selectivity is likely to pay off faster.

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, (ii) the batched *put* statements adding them, and (iii) our real-time rendition of the monetary costs entailed. 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 within S3, (ii) the logical and physical plans for recombining (through intersection, join and/or union) the look-up results to identify the relevant datasets, (iv) the real-time AWS resource consumption while the physical plan is run to find out datasets, and finally (v) the resource consumption within S3 and EC2 associated to loading the respective datasets from S3 into EC2, processing the query there, and downloading the results out of the cloud. Figure 2 illustrates the interface.

We will also show an index advisor tool, which (i) based on user-chosen XML or RDF datasets, hypothetical workload, and a set of preferences (relative weights given to query response time and monetary costs, types of AWS instances etc.) estimates the costs entailed by the workload on the datasets for each indexing strategy, and (ii) based on the log of queries run, aggregates the overall costs based on the used indexing strategy as well as the would-be costs, had other strategies been used.

5 Related work

An early work [3] 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 [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 [6] and the appearance of Hadoop, massively parallel data management is a hot topic in industry and academia. Many works have focused on improving the performance and reliability of Hadoop [12]. A separate approach is taken in works such as Hyracks [2] and Stratosphere [1], which provide more flexible, expressive parallel execution frameworks, focusing on complex nested objects [2]. The increasing popularity of RDF has led to many recent works on parallelizing RDF processing within MapReduce-style frameworks; recent proposals are [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 We are grateful to Dario Colazzo, François Goasdoué and Ioana Manolescu for their substantial guidance and support in this work. 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.

References

[1] 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.

[2] 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.

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

[4] F. Bugiotti, F. Goasdoué, Z. Kaoudi, and I. Manolescu. RDF Data Management in the Amazon Cloud. In *DanaC workshop (next to EDBT)*, 2012.

[5] J. Camacho-Rodríguez, D. Colazzo, and I. Manolescu. Building Large XML Stores in the Amazon Cloud. In *ICDE Workshops*, 2012.

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

[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] K. Karanasos, A. Katsifodimos, I. Manolescu, and S. Zoupanos. The ViP2P Platform: XML Views in P2P. *CoRR*, arXiv:1112.2610, 2011.

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

[12] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with MapReduce: a survey. *SIGMOD Record*, 40(4):11–20, 2011.

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

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