RDF Data Management in the Amazon Cloud

Francesca Bugiotti, François Goasdoué, Zoi Kaoudi, Ioana Manolescu

RDF Data Management in the Amazon Cloud

Francesca Bugiotti*
Università Roma Tré, Italy
franbugiotti@yahoo.it

François Goasdoué
Université Paris-Sud and Inria Saclay, France
fg@lri.fr

Zoi Kaoudi
Inria Saclay and Université Paris-Sud, France
zoi.kaoudi@inria.fr

Ioana Manolescu
Inria Saclay and Université Paris-Sud, France
ioana.manolescu@inria.fr

ABSTRACT

Cloud computing has been massively adopted recently in many applications for its elastic scaling and fault-tolerance. At the same time, given that the amount of available RDF data sources on the Web increases rapidly, there is a constant need for scalable RDF data management tools. In this paper we propose a novel architecture for the distributed management of RDF data, exploiting an existing commercial cloud infrastructure, namely Amazon Web Services (AWS). We study the problem of indexing RDF data stored within AWS, by using SimpleDB, a key-value store provided by AWS for small data items. The goal of the index is to efficiently identify the RDF datasets which may have answers for a given query, and route the query only to those. We devised and experimented with several indexing strategies; we discuss experimental results and avenues for future work.

1. INTRODUCTION

Cloud computing has been massively adopted recently in many applications for the scalability, fault-tolerance and elasticity features it provides. Cloud-based platforms free the application developer from the burden of administering the hardware and provide resilience to failures, as well as elastic scaling up and down of resources according to the demand. The recent development of such environments has a significant impact on the data management research community, in which the cloud provides a distributed, shared-nothing infrastructure for scalable data storage and processing. Many works have relied on cloud infrastructures focusing on different aspects such as implementing basic database primitives in cloud services [4] or algebraic extensions of the MapReduce paradigm [5] for efficient parallelized processing of queries [3].

*Part of the work has been performed while the author was visiting Inria Saclay.

DanaC 2012 March 30, 2012, Berlin, Germany

Within the wider data management field, significant effort has been invested in techniques for the efficient management of Web data. A constantly increasing number of sources expose and/or share their data represented in the W3C's Resource Description Framework (or RDF, in short) [9]. A well-known interesting RDF data source is DBPedia (`http://aws.amazon.com/datasets/2319`), while more data sources are catalogued in the Linked Open Data Web site (`http://linkeddata.org`). RDF has also been used in highly dynamic news management scenarios, such as the BBC's reporting on the World Football Cup [10]. Efficient systems have been devised in order to handle large RDF volumes in a centralized setting, with RDF-3X [15, 16] being among the best-known.

To exploit ever-increasing volumes of data in a cloud, works such as [6, 11, 14, 21], either focus on MapReduce or use cloud-based key-value stores to store RDF data. These works mostly target at designing parallel techniques for efficiently handling massive amounts of data.

In this work, we explore an alternative and possibly complementary approach. We envision an architecture where large amounts of RDF data is partitioned in small datasets and reside in an elastic cloud-based store. We focus on the task of efficiently routing queries to only those datasets that are likely to have matches for the query. Selective query routing reduces the total work associated to processing a query, and in a cloud environment where total work also translates in financial costs, leads to reduced expenses. To achieve this, whenever data is uploaded in the cloud store, we index it and store the index in an efficient (cloud-resident) store for small key-value pairs. Thus, we take advantage of: large-scale stores for the data itself; elastic computing capabilities to evaluate queries; and the fine-grained search capabilities of a fast key-value store, for efficient query routing.

Our implementation relies on the Amazon Web Services (AWS) cloud platform [1], one of the most prominent commercial cloud platforms today, which has been used in other data management research works [4, 18]. We store RDF files in Amazon Simple Storage Service (S3) and use Amazon SimpleDB for storing the index. Finally, RDF queries are evaluated against the RDF files retrieved from S3, within the Amazon Elastic Compute Cloud (EC2). While our results only hold within the Amazon platform, the architecture is quite generic and could be ported to other similar cloud-based environments.

This paper is organized as follows. Section 2 provides some background on the RDF data model and the Amazon Web Services platform. Section 3 presents our proposed system architecture while Section 4 describes in detail our RDF indexing techniques within the Amazon cloud. In Section 5 we present the results of our experimental evaluation. Finally, we discuss related work in Section 6 and conclude with Section 7 by presenting directions for future work.

2. PRELIMINARIES

In this section, we briefly introduce the basics of our supported data model and query language as well as give a description of the Amazon cloud.

2.1 RDF and SPARQL

In order to lay the background and fix the terminology of the paper, here we briefly summarize some features of the Resource Description Framework (RDF) data model [9, 12]. RDF is a data model and formalism recommended by W3C and designed for the exchange and reuse of structured data among web applications. It is based on the concept of *resource* which is everything that can be referred to through a Uniform Resource Identifier (URI). RDF data model is built on resources, properties (a.k.a. predicates) and values. Properties can be seen as relations linking resources and values. The values can be either URIs, constants from primitive types called literals (such as string or integers), or blank nodes. Blank nodes are identifiers for unknown values. We use an underscore-prefixed notation to refer to them, as in `_:bnodeID`.

Information about resources is encoded using RDF triples, also called statements. A statement is a *triple* of the form $(subject, property, object)$, abbreviated as (s, p, o). The subject of a triple identifies the resource that the statement is about, the property identifies an attribute describing the subject, while the object gives the value of the property.

More specifically, let U, L and B denote three (pairwise disjoint) sets of URIs, literals, and blank nodes, respectively. A (well-formed) *triple* is a tuple (s, p, o) from $(U \cup B) \times U \times (U \cup L \cup B)$, where s is the subject, p is the property and o is the object of the triple.

A set of triples comprises a *graph*, which can be also called a *dataset*. Indeed, a set of triples encodes a graph structure in which every triple (s, p, o) describes a directed edge labelled with p from the node labelled with s to the node labelled with o. A graph can be identified by a URI value. In RDF, graphs can be built from other graphs through a *merge* operation. This is particularly useful for traversing various data sources with queries through the integration of these data sources.

The *merge* of RDF graphs is as follows. If these graphs have no blank nodes in common, then merging them results in their union. Otherwise, the graphs do share blank nodes and merging them amounts to renaming the (shared) blank nodes within the graphs with fresh identifiers, so that we fall into the previous case.

SPARQL [17] is the W3C standard for querying RDF graphs. In this paper, we consider the Basic Graph Pattern (BGP) queries of SPARQL, i.e., its conjunctive fragment allowing to express the core Select-Project-Join database queries. The normative syntax of BGP queries is

`SELECT` $?v_1 \ldots ?v_m$ `FROM` uri_1 ... `FROM` uri_n `WHERE` $\{t_1, \ldots, t_o\}$

with $\{t_1, \ldots, t_o\}$ an RDF graph whose triples can also use variables, $?v_1 \ldots ?v_m$ a set of variables occurring in $\{t_1, \ldots, t_o\}$ that defines the output of the query, and $uri_1, \ldots, uri_n$ the URIs of the graphs whose merging must be queried. Here, the notion of triple is actually generalized to that of *triple pattern* (s, p, o) from $(U \cup B \cup V) \times (U \cup V) \times (U \cup L \cup B \cup V)$, where V is a set of variables. Observe that repeating a variable in a SPARQL query is the way of expressing joins.

In the following, when a query has no `FROM` clause, we assume that it must be evaluated against the merge of all the graphs whose URIs are known.

Let us now turn to the semantics of a BGP query. First, a mapping μ from $B \cup V$ to $U \cup B \cup L$ is defined as a partial function $\mu : B \cup V \rightarrow U \cup B \cup L$. If o is a triple pattern or a set of variables, $\mu(o)$ denotes the result of replacing the blank nodes and variables in o according to μ. The domain of μ, $dom(\mu)$, is the subset of V where μ is defined. Let $q =$ `SELECT` $?v_1 \ldots ?v_m$ `FROM` uri_1 ... `FROM` uri_n `WHERE` $\{t_1, \ldots, t_o\}$ be a BGP query and D the graph obtained by merging the datasets whose URIs are $uri_1, \ldots, uri_n$. The evaluation of q is: $eval(q) = \{\mu(?v_1 \ldots ?v_m) \mid dom(\mu) = varbl(q)$ *and* $\{\mu(t_1), \mu(t_2), ..., \mu(t_n)\} \subseteq D\}$, with $varbl(q)$ the set of variables and blank nodes occurring in q.

Notice that evaluation *treats blank nodes in a query as non-distinguished variables*. That is, one could consider without loss of generality queries without blank nodes.

Notation. From now on, to avoid writing long URIs, we use *namespaces*. Namespaces allow associating a short convenient prefix to the first part of a lengthy URI. Following the namespace usage, a URI can be replaced by the prefix to which is appended the last part of the URI. For example, the URI `http://xmlns.com/foaf/0.1/name` can be written `foaf:name`, provided that `foaf` has been declared as the namespace for `http://xmlns.com/foaf/0.1/`.

2.1.1 Running Example

Throughout the paper we rely on a simple running example consisting of three datasets, representing: the articles published by Inria researchers, the books published by Inria researchers, and the Inria labs. The content of these datasets is depicted in Figure 1. Although for now we use the datasets as defined by the users, clearly, partitioning large sets of triples into datasets so as to optimize the selectivity of the index is an interesting avenue for future work.

The *articles* dataset describes the resource `inria:article1` whose author (`inria:hasAuthor`) is represented by the resource `inria:bar` whose name (`inria:hasName`) is "Bar" and whose nationality (`inria:hasNationality`) is "American". The namespace `inria` is used to abbreviate the URI prefix `http://inria.fr/`.

The *books* dataset describes the resource `inria:book1` whose author (`inria:hasAuthor`) is `inria:foo`, and for which there is an unknown (`_:uid1`) contact author (`inria:hasContactInfo`) whose role (`inria:hasRole`) is "Professor" and whose telephone number (`inria:hasTel`) is "+33 134879". The resource `inria:foo` has also a name (`inria:hasName`) which is "Foo", nationality (`inria:hasNationality`) "French" and telephone number (`inria:hasTel`) "+33 12345678". The resource `inria:book2` is also described whose author (`inria:hasAuthor`) is the unknown author `_:uid1`.

Finally, the *labs* dataset describes the resource `labInria:lab1` whose name (`labInria:hasName`) is "ResearchLab" and whose location (`labInria:hasLocation`) described by the

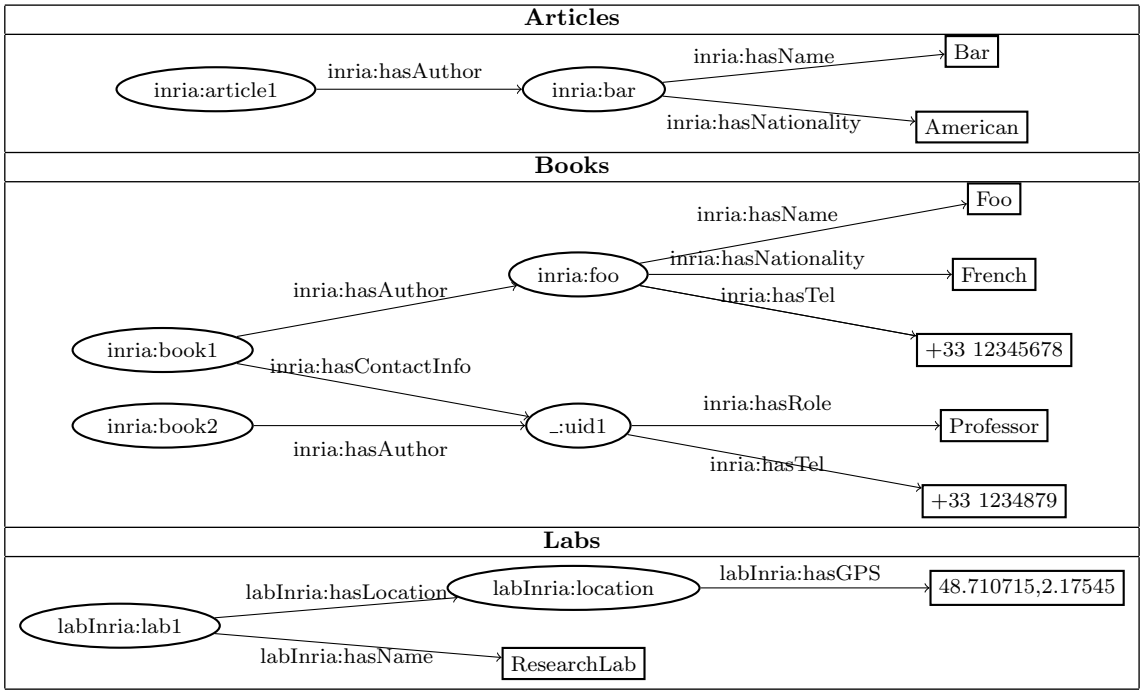

Figure 1: Graph representation of the example RDF data.

$$
\begin{array}{lcl}
db & = & dom+ \\
dom & = & (name, item+) \\
item & = & (key, attribute+) \\
attribute & = & (name, value)
\end{array}
$$

Figure 2: SimpleDB database layout.

resource `http://labs.inria.fr/lab1/location` has the GPS coordinates (`labInria:hasGPS`) "48.710715,2.17545". The namespace `labInria` is defined by the URI `http://labs.inria.fr/rdfExample/`.

2.2 Amazon cloud

Amazon Web Services (AWS) provides since 2006 a cloud-based services platform which organizations and individuals can take advantage of, in order to develop elastic scalable applications. For the purpose of this work, we mostly relied on the Amazon SimpleDB, a structured store for small atomic objects, which we describe in Section 2.2.1, and on Amazon's Simple Storage Service (S3), outlined in Section 2.2.2.

2.2.1 Amazon SimpleDB

SimpleDB is a non relational data store provided by Amazon which focuses on high availability (ensured through replication), flexibility and scalability. SimpleDB supports a set of APIs to query and store items in the database. A SimpleDB data store is organized in *domains*. Each domain is a collection of *items* identified by their name. Each item contains one or more *attributes*; an attribute has a *name* and a set of associated *values*. Figure 2 outlines the structure of a SimpleDB database.

In the sequel, we can thus summarize the layout of data within SimpleDB as a four-level hierarchy $D|I|A|V$, where D is the domain name, I is the item name, A and V are attribute name and attribute value, respectively.

SimpleDB API. The main operations of SimpleDB API are the following:

- `ListDomains()` retrieves all the domains associated to one AWS account.
- `CreateDomain(D)` and `DeleteDomain(D)` creates a new domain `D` or deletes an existing one, respectively.
- `PutAttributes(D, k, (a,v)+)` inserts or replaces attributes `(a,v)+` into an item with name `k` of a domain `D`. If the item specified does not exist, SimpleDB will create a new item. `BatchPutAttributes` performs up to 25 `PutAttributes` operations in a single API call, which allows for obtaining a better throughput performance.
- `GetAttributes(D, k)` returns the set of attributes associated with item `k` in domain `D`.
- `Select(expr)` operation queries a specified SimpleDB domain using query expressions similar to the standard SQL SELECT statements. We elaborate more about this API operation in the next section.

It is not possible to execute an API operation across different domains. Therefore, if required, the aggregation of results from API operations executed over different domains has to be done in the application layer. AWS ensures that operations over different domains run in parallel. Hence, it is beneficial to split the data in several domains in order to obtain maximum performance.

As most non-relational databases, SimpleDB does not follow a strict transactional model based on locks or timestamps. It only provides the simple model of conditional puts. It is possible to update fields on the basis of the values

of other fields. It allows for the implementation of elementary transactional models such as some entry level versions of optimistic concurrency control.

SimpleDB Select operation. The `Select` operation which can be used for querying SimpleDB is similar to the standard SQL select statements and has the following structure:

```
SELECT (* | count(*) | itemName() | (attr1, ... attrN))
FROM domain_name
[WHERE expression]
[sort_instructions]
[LIMIT limit]
```

The `WHERE` expression can be any of the following:

```
(<simple comparison>)
(<select expression> intersection <select expression> )
(NOT <select expression>)
(<select expression>)
(<select expression> or <select expression>)
(<select expression> and <select expression>)
```

Comparison operators (`=`, `!=`, >, ..., `LIKE`, `IN`, `IS NULL`, `IS NOT NULL`, etc.) are applied to a single attribute and are lexicographical in nature.

SimpleDB limitations. AWS imposes some size and cardinality limitations on SimpleDB. These limitations include:

- *Domains number*: the default settings of a AWS account allow for at most 250 domains. While it is possible to negotiate more, this has some overhead (one must discuss with a sale representative etc. - it is not as easy as reserving more resources through an online form).
- *Domain size*: the maximum size of a domain cannot exceed 10 GB and the 10^9 attributes.
- *Item name length*: the name of an item should not occupy more than 1024 bytes.
- *Number of (attribute, value) pairs in an item*: this cannot exceed 256. As a consequence, if an item has only one attribute, that attribute cannot have more than 256 associated values.
- *Length of an attribute name or value*: this cannot exceed 1024 bytes.

In addition, when we execute a `Select` query in a domain, there are also some limitations. Here we present only the ones related to our proposed architecture.

- The query cannot return more than 2500 items and the size of the result cannot exceed 1MB. If any of these conditions are not met, it is possible to retrieve the additional results by iterating the query execution using an identifier returned from the previous round.
- The maximum query execution time is 5 seconds, i.e., if a query takes longer than 5 seconds to be executed, it returns an error.

2.2.2 Amazon Simple Storage Service

Amazon S3 is a storage web service for raw data and hence, ideal for storing large objects or files. S3 stores the data in named buckets. Each object stored in a bucket has associated a unique name (key) within that bucket, metadata, an access control policy for AWS users and a version ID. The number of objects that can be stored within a bucket is unlimited.

If we want to retrieve an object from S3, we should access the bucket that contains it and request it by its name. S3 allows to access the metadata associated to an object without retrieving the complete entity. Unlike SimpleDB, there is no performance difference in S3 between storing objects in multiple buckets and storing them in just one.

S3 API. The S3 API includes the following basic operations:

- `ListBuckets()` returns the list of created buckets, `CreateBucket (B)` creates a new bucket `B` and `DeleteBucket(B)` deletes an existing bucket.
- `PutObject(buck, key, obj, meta)` stores an object `obj` with name `key` and metadata `meta` within bucket `buck`.
- `GetObject(buck, key)` retrieves object `key` from a bucket `buck`.
- `GetObjectMetadata(buck, key)` retrieves only the object's metadata without fetching the actual object.

2.2.3 Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud (EC2) is a virtual computing environment that allows the use of web service interfaces to launch virtual computer instances on which users' applications can be run. The virtual machine images are stored in the cloud and it is possible to configure them choosing hardware features such as RAM size, network access, etc. The utilization cost is calculated on the basis of the configuration of the machine, the running time of the application and the data transfer.

2.2.4 Amazon Simple Queue Service

Amazon Simple Queue Service (SQS) provides reliable and scalable queues that enable asynchronous message-based communication between the distributed components of an application. This service prevents an application from message loss and from requiring each component to be always available.

3. ARCHITECTURE

Our envisioned architecture relies on the following services provided by Amazon: S3 for permanent storage, SimpleDB for storing structured information about the data stored in S3, EC2 for running our modules and SQS for the communication between the different modules.

RDF datasets are stored in S3 and each dataset is treated as a uninterpreted BLOB object. As explained in Section 2.2, it is necessary to associate a key to every resource stored in S3 in order to be able to retrieve it. For this reason, we assign to each dataset as a key the URI of the dataset. In general we indicate as $URI(ds_j)$ the URI associated to the dataset j. On the other hand, dataset indexes are instead kept in SimpleDB. In this way, we allow for fast retrieval of the URIs of the RDF datasets.

An overview of our system architecture is depicted in Figure 3. A user interaction with our system can be described as follows.

The user submits to the *front-end* component RDF datasets (1) and the front-end module stores the file in S3 (2). Then, it creates a message containing the reference to the dataset and inserts it to the *loader request queue* (3). Any

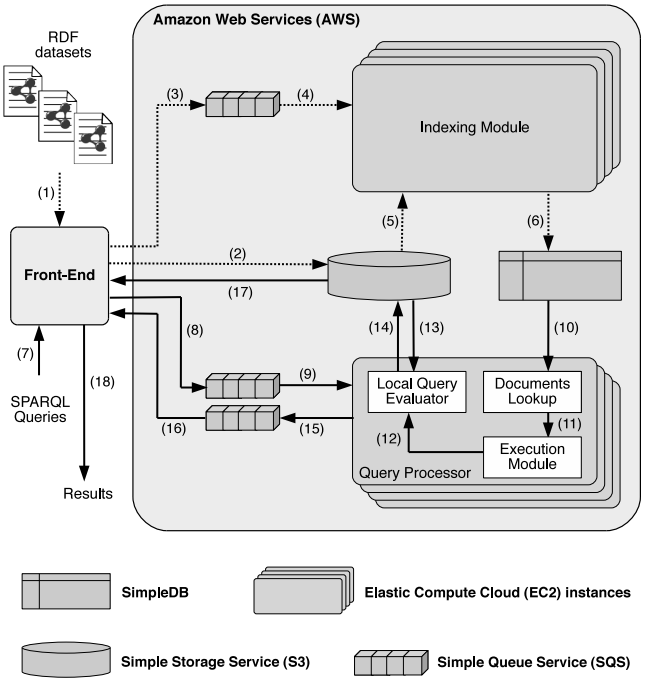

Figure 3: Proposed architecture.

EC2 instance running our *indexing module* receives such a message (4) and retrieves the dataset from S3 (5). The indexing module after transforming the dataset into a set of RDF triples, it creates the index data and inserts it in SimpleDB (6).

When a user submits a SPARQL query to the front-end (7), the front-end inserts the corresponding message into the *query request queue* (8). Any EC2 instance running our *query processor* receives such a message and parses the query (9). The query processor performs a lookup to the indexes in SimpleDB to find out the datasets that contain information to answer the query (10). Any processing required for merging or unioning the RDF datasets retrieved from SimpleDB is performed in the *execution module* (11). Then, the *local query evaluator* receives the final list of URIs pointing to the RDF datasets in S3 (12), retrieves them and evaluates the SPARQL query against these datasets (13). Then, it writes the results to S3 (14) and creates a message which is inserted into the *query response queue* (15). The front-end receives this message (16) and retrieves the results from S3 (17). Finally, the results are returned to the user (18).

4. INDEXING STRATEGIES IN SIMPLEDB

In this section we describe the RDF indexing strategies we have developed in SimpleDB that allow us to find out in a light way the RDF datasets that should be retrieved from S3 and then, queried, to form the answer to the query. All indexing strategies below return exactly the same set of datasets for a given query. Since we use SimpleDB to store the indexes they should conform with the data model of SimpleDB described in Section 2.

Notation. To simplify presentation, we will describe each indexing strategy in terms of the four levels of information that SimpleDB allows us to use, namely $(D|I|A|V)$ (see Section 2.2.1). To index RDF, we may use the values of subjects (S), properties (P) and objects (O) occurring in RDF triples, as well as the URIs (U) of the RDF datasets. Moreover, we will also use a set of three token strings, which we denote by $\underline{S}$, $\underline{P}$ and $\underline{O}$, and which we may insert in the index to specify whether some piece of data is to be treated as a subject, property, or object, respectively. In addition, we will use the symbol $\|$ to denote string concatenation. In cases where there is no confusion we may omit it (e.g., $\underline{SP}$ denotes the concatenation of the string values "subject" and "property"). Similarly, we will use a token string denoted by $\underline{D}$ to represent a constant domain name. As we will show, each indexing strategy can be represented by a concatenation of four |-separated symbols, specifying which information item is used in the domain name, item name, attribute name and attribute value, respectively.

4.1 Attribute-based strategy

In the following we describe our first indexing strategy of RDF datasets in SimpleDB and the query processing algorithm which utilizes the indices in order to retrieve datasets that are relevant to the SPARQL query. These datasets can be used then to form the answer to the query.

Indexing. According to this strategy, for each dataset three indexes are created: one for the subjects, one for the properties and one for the objects. Each index resides in a different SimpleDB domain. Then, for each dataset, an item named after the dataset is inserted in the respective domain. The name of the dataset is also the URI that allows us to access the RDF graph stored in S3. Using our notation we therefore have the following indexes:

1. $(\underline{S}|U|\underline{S}|S)$, which enumerates subjects using attribute-value pairs where the attribute name is the word "subject" and the value is the subject itself.
2. $(\underline{P}|U|\underline{P}|P)$, which enumerates properties using attribute-value pairs where the attribute name is the word "property" and the value is the property itself.
3. $(\underline{O}|U|\underline{O}|O)$, which enumerates objects using attribute-value pairs where the attribute name is the word "object" and the value is the object itself.

Handling SimpleDB limitations. As already discussed, SimpleDB can manage up to 256 attribute-value elements for each item. This means that a given dataset, confined in a single item by this strategy, can have up to 256 distinct subjects, 256 distinct properties and 256 distinct objects. While for properties this might not be a problem in many cases, for subjects and objects the limit is quickly reached. We have two dimensions of growth at our disposal to cope with this:

1. We may assign "partition" URIs $URI|1$, $URI|2$, ..., $URI|k$ for a given real dataset URI URI. When e.g., the $\underline{S}$ domain overflows for the first time and a given dataset URI, we create the artifficial $URI|1$ and register the subsequent items as belonging to the (fictitious) dataset $URI|1$. (This also amounts to a virtual partitioning of the input dataset in several slices.) When $URI|1$ overflows, say, in the $\underline{S}$ or $\underline{O}$ index, we move to $URI|2$ and so on. To ensure complete index lookups, a secondary index tracks all the partition URIs associated to a given URI.

Table 1: Attribute-based indexing strategy

subject domain i	
item key	(attr. name, attr. value)
$URI_k(ds_1)$	($\underline{S}$, s'_{ds_1}), ($\underline{S}$, s''_{ds_1}), ...
$URI_k(ds_2)$	($\underline{S}$, s'_{ds_2}), ...
property domain j	
item key	(attr. name, attr. value)
$URI_k(ds_1)$	($\underline{P}$, p'_{ds_1}), ($\underline{P}$, p''_{ds_1}), ...
$URI_k(ds_2)$	($\underline{P}$, p'_{ds_2}), ($\underline{P}$, p''_{ds_2}), ...
object domain l	
item key	(attr. name, attr. value)
$URI_k(ds_1)$	($\underline{O}$, o'_{ds_1}), ($\underline{O}$, o''_{ds_1}), ...
$URI_k(ds_2)$	($\underline{O}$, o''_{ds_2}), ...

2. We may use more domains. Let B denote the maximum number of SimpleDB domains available to us ($B = 250$ for a regular AWS account). We partition these domains according to their usage: some domains, denoted $\underline{S}_1$, $\underline{S}_2$, ..., $\underline{S}_i$ will be used for the subjects, and similarly $\underline{P}_1$, $\underline{P}_2$, ..., $\underline{P}_j$, respectively, $\underline{O}_1$, $\underline{O}_2$, ..., $\underline{O}_l$ for the objects, with $i, j, l \geq 1$ and $3 \leq i + j + l \leq B$. For a given dataset, whenever we reach the 256 attribute-value limitation in an item in a domain $\underline{S}_i$ (or $\underline{P}_j$, or $\underline{O}_l$), we create a *new* item for this dataset in the "next" domain $\underline{S}_{i+1}$ (respectively, $\underline{P}_{j+1}$, or $\underline{O}_{l+1}$). The domain $\underline{S}_{i+1}$ is created the first time that the limits of the domain $\underline{S}_i$ are reached (and similarly for $\underline{O}$ and $\underline{P}$). Increasing the number of domains favors parallelism, since index entries for a given dataset, that are partitioned over several domains, can be simultaneously filled in, and consulted.

Our current implementation first, expands to several domains in order to maximize parallelism, and then, once the maximum number of domains has been taken, introduces partition URIs. We plan to further the analysis of the tradeoffs between the two techniques in our future work.

Table 1 outlines data organization in SimpleDB using this strategy. The data shown in Figure 1 leads to the index shown in Table 2. For this small example, we use only three domains and one item per dataset in each domain.

Querying. For each constant (URI or literal) of a SPARQL query a SimpleDB `SELECT` query is submitted to the $\underline{S}$ (or $\underline{P}$, or $\underline{O}$) domain(s), depending on the position of the constant in the query. Each such look-up retrieves the URIs of the dataset containing the respective subject (or object, or property) value. For each triple pattern, the results of all the `SELECT` queries based on constants of that triple need to be intersected. The intersection leads to a set of URIs obtained out of a given triple pattern. The union of all URI sets thus obtained from the triples in a SPARQL query is the set of datasets on which the query must be evaluated.

Using our running example assume that we want to evaluate the following SPARQL query:

Listing 1: Example SPARQL query

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

Table 2: Example of attribute-based index

subject domain	
item key	(attr. name, attr. value)
articles	($\underline{S}$, inria:article1), ($\underline{S}$, inria:bar)
books	($\underline{S}$, inria:book1), ($\underline{S}$, inria:foo), ($\underline{S}$, _:uid1), ($\underline{S}$, inria:book1)
labs	($\underline{S}$, labInria:lab1), ($\underline{S}$, labInria:location)
property domain	
item key	(attr. name, attr. value)
articles	($\underline{P}$, inria:hasAuthor), ($\underline{P}$, inria:hasName), ($\underline{P}$, inria:hasNationality)
books	($\underline{P}$, inria:hasAuthor), ($\underline{P}$, inria:hasContactInfo), ($\underline{P}$, inria:hasRole)($\underline{P}$, inria:hasTel), ($\underline{P}$, inria:hasNationality), ($\underline{P}$, inria:hasRole)
labs	($\underline{P}$, labInria:hasLocation), ($\underline{P}$, labInria:hasName), ($\underline{P}$, labInria:hasGPS)
object domain	
item key	(attr. name, attr. value)
articles	($\underline{O}$, inria:bar), ($\underline{O}$, "Bar"), ($\underline{O}$, "American")
books	($\underline{O}$, inria:foo), ($\underline{O}$, "Foo"), ($\underline{O}$, "+33 12345678"), ($\underline{O}$, "French"), ($\underline{O}$, "+33 1234879"), ($\underline{O}$, "Professor")
labs	($\underline{O}$, labInria:location), ($\underline{O}$, "ResearchLabs"), ($\underline{O}$, "48.710715,2.17545")

The corresponding SimpleDB queries that are required in order to retrieve the corresponding datasets is the following:

```
q1: SELECT itemName()
    FROM property_domain
    WHERE property = inria:hasAuthor;

q2: SELECT itemName()
    FROM object_domain
    WHERE object = "Foo";

q3: SELECT itemName()
    FROM property_domain
    WHERE property = inria:hasContactInfo;
```

The datasets retrieved from SimpleDB queries `q1` and `q2` will be intersected and the resulted datasets will be merged with the datasets retrieved from query `q3`. The query will be then evaluated on final set of the merged datasets.

Analytical cost model. In this section, we analyze the cost of the index strategy with respect to the size of the index as well as the number of required lookups while processing a SPARQL query. In both cases we present analytical costs for the worst case scenario.

Let n be the number of datasets stored in S3 and T the maximum size of a dataset in terms of the number of triples it consists. Hence, if T_{ds_i} is the size of dataset ds_i then $T = max(T_{ds_1}, T_{ds_2}, ..., T_{ds_n})$. We assume that the number of distinct subjects, properties and objects values appearing in a dataset is equal to the size of the dataset itself, and thus equals to the number of triples (worst case scenario). For each triple in a dataset we create three entries in SimpleDB. The size of the index for this strategy will be $3 \times n \times T$.

For the query processing, let q be the number of triple patterns of a BGP SPARQL query, then in the worst case scenario, the number of constants a query can have is at most $3 \times q$ (i.e., in case of a boolean query). Using this strategy, one lookup per constant in a query is performed to the appropriate domain type. For the case where the SimpleDB limit has not been reached and we thus have only one domain for the subjects, properties and objects, the number of lookups to SimpleDB is $3 \times q$.

Table 3: Attribute-pair indexing strategy

subject-property domain i	
item key	(attr. name, attr. value)
$URI_k(ds_1)$	(s'_{ds_1}, p'_{ds_1}), (s'_{ds_1}, p''_{ds_1}), (s''_{ds_1}, p'_{ds_1}) ...
$URI_k(ds_2)$	(s'_{ds_2}, p'_{ds_2}), (s'_{ds_2}, p''_{ds_2}), (s''_{ds_2}, p'_{ds_2}) ...
property-object domain j	
item key	(attr. name, attr. value)
$URI_k(ds_1)$	(p'_{ds_1}, o'_{ds_1}), (p''_{ds_1}, o''_{ds_1}) , ...
$URI_k(ds_2)$	(p'_{ds_2}, o'_{ds_2}), ...
object-subject domain l	
item key	(attr. name, attr. value)
$URI_k(ds_1)$	(o'_{ds_1}, s'_{ds_1}), (o''_{ds_1}, s'_{ds_1}), ...
$URI_k(ds_2)$	(o'_{ds_2}, s'_{ds_2}), ...

In the case where we have reached the SimpleDB limitation and more than one domain for either of the subject, property, object domains are created, we need to perform one lookup to each domain separately. If d is the total number of allocated domains, the number of lookups to SimpleDB in the worst case scenario equals to $3 \times q \times d$.

4.2 Attribute-pair strategy

Indexing. This strategy uses three indexes, one for each pair of attributes in an RDF triple:

1. The first index is ($\underline{SP}|U|S|P$), which enumerates the relation `(s, p)`, asserting whenever a subject has a property.
2. The second index is ($\underline{PO}|U|P|O$), which enumerates the relation `(p, o)`, asserting whenever a property refers to an object.
3. Similarly, the third index is ($\underline{OS}|U|O|S$), which enumerates the relation `(o, s)` asserting whenever an object is connected to a subject.

Handling SimpleDB limitations. SimpleDB attribute-value limitation leads to constraining a single dataset to 256 distinct subject-property pairs, 256 distinct property-object pairs and 256 distinct object-subject pairs. To overcome this limitation, we use the same technique as we described in the previous indexing strategy. We increase the number of each type of domain up to the number we have available. At the point where this number is reached we start filling the already existing domains by adding new items for the same datasets. The general organization of this indexing strategy in SimpleDB is depicted in Table 3. Using our running example of Figure 1, we have the index organization as shown in Table 4.

Querying. For each SPARQL triple pattern having at least one constant, we evaluate one SimpleDB `SELECT` query on the corresponding domain(s) depending on the position of the constants in the triple. In case a triple pattern has one bound value we define a query where the corresponding attribute of the domain should not be null. In case a triple pattern has two bound values c_1 and c_2 we define a SimpleDB query whose `where` clause asks that c_1 equals with c_2. The query will be evaluated against the union of the datasets returned by the SimpleDB `SELECT` queries corresponding to each triple.

Using our running example and the example SPARQL query of Listing 1 we show the corresponding SimpleDB queries that are required in order to retrieve the corresponding datasets:

```
q1: SELECT itemName()
    FROM property_object_domain
    WHERE inria:hasAuthor = "Foo";

q2: SELECT itemName()
    FROM property_object_domain
    WHERE inria:hasContactInfo IS NOT NULL;
```

The datasets retrieved from queries `q1` and `q2` will be merged and the query will be then evaluated against the merged datasets.

Table 4: Example of attribute-pair index

subject-property domain	
item key	(attr. name, attr. value)
articles	(inria:article1, inria:hasAuthor), (inria:bar, inria:hasName), (inria:bar, inria:hasNationality)
books	(inria:book1, inria:hasAuthor), (inria:book, inria:hasContactInfo), (inria:foo, inria:hasName), (inria:foo, inria:hasNationality), (inria:foo, inria:hasTel), (_:uid1, inria:hasRole), (_:uid1, inria:hasTel) (inria:book2, inria:hasAuthor)
labs	(labInria:lab1, labInria:hasLocation), (labInria:lab1, labInria:hasName), (labInria:location, labInria:hasGPS)
property-object domain	
item key	(attr. name, attr. value)
articles	(inria:hasAuthor, inria:bar), (inria:hasName, "Bar"), (inria:hasNationality, "American")
books	(inria:hasAuthor, inria:foo), (inria:hasContactInfo, _:uid1), (inria:hasTel, "+33 12345678"), (inria:hasNationality, "French"), (inria:hasRole, "Professor"), (inria:hasAuthor, _:uid1)
labs	(labInria:hasLocation, labInria:location), (labInria:hasName, "ResearchLabs"), (labInria:hasGPS, "48.710715,2.17545")
object-subject domain	
item key	(attr. name, attr. value)
articles	(inria:bar, inria:article), ("Bar", inria:bar), ("American", inria:bar)
books	(inria:foo, inria:book), (_:uid1, inria:book), ("Foo", inria:foo), ("French", inria:foo), ("+33 12345678", inria:foo), ("Professor", _:uid1), ("+33 1234879", _:uid1), (_:uid1, inriafr:Foo)
labs	(labInria:location, labInria:lab1), ("ResearchLabs", labInria:lab1), ("48.710715,2.17545", labInria:location)

Analytical cost model. Similarly with the attribute-based index, for each triple of a certain dataset three entries are created in SimpleDB. In order to compute the size of this index, we assume that the number of distinct values of the subject-property pairs, property-object pairs and object-subject pairs appearing in a dataset equal to the the number of triples of the dataset (worst case scenario). Then, the size of the index for the attribute-pair strategy equals to $3 \times n \times T$, where n is the number of datasets and T the maximum number of triples of a dataset as introduced previously.

In the query processing procedure of this indexing strategy, one lookup is performed for each triple pattern of a SPARQL query. Certainly, this holds only in the case where the SimpleDB limit has not been reached. Then, the number of lookups to SimpleDB when using the attribute-pair strat-

Table 5: Attribute-subset indexing strategy

attribute-subset domain	
item key	(attr. name, attr. value)
$\underline{S}\|\|subject$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ), ...
$\underline{P}\|\|property$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),...
$\underline{O}\|\|object$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),...
$\underline{SP}\|\|subject\|\|property$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),....
$\underline{PO}\|\|property\|\|object$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),...
$\underline{SO}\|\|subject\|\|object$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),...
$\underline{SPO}\|\|subject\|\|property\|\|object$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),...

egy is equal to q, where q is the number of triple patterns of the SPARQL query.

In the case where we have reached the SimpleDB limitation and created more than one domain for either type of domain, we need to performe one lookup for each such domain. Then, the number of lookups to SimpleDB for the worst case scenario is $q \times d$, where q is the number of triple patterns of the SPARQL query and d is the total number of created domains in SimpleDB.

4.3 Attribute-subset strategy

Indexing. This strategy encodes each triple `(s, p, o)` by a set of seven patterns `(s)`, `(p)`, `(o)`, `(s, p)`, `(s, p, o)`, `(p, o)` and `(s, o)` corresponding to all non-empty attribute subsets. For each triple and each of these seven patterns, a new SimpleDB item is created and named after the pattern. As attribute name, we use the URI of the dataset containing this pattern; as attribute value, we use ϵ.

Using our notation, the indexes we create can be described as: $(\underline{D}|\underline{S}S|U|\epsilon)$, $(\underline{D}|\underline{P}P|U|\epsilon)$, $(\underline{D}|\underline{O}O|U|\epsilon)$, $(\underline{D}|\underline{SP}SP|U|\epsilon)$, $(\underline{D}|\underline{PO}PO|U|\epsilon)$, $(\underline{D}|\underline{SO}SO|U|\epsilon)$ and $(\underline{D}|\underline{SPO}SPO|U|\epsilon)$.

The general organization of this index is illustrated in Table 5. The data from our running example leads to the index configuration outlined in Table 6.

Handling SimpleDB limitations. In this indexing strategy the limits of SimpleDB are exceeded when we have more than 256 datasets stored in S3 and all these datasets have one triple element value or a combination of them in common. Although this situation is not so often in various application scenarios, we cope with this by creating an extra domain each time the limitation is reached. In addition, when more than 10^9 distinct values of all triple elements combinations appear in the datasets stored in S3, the limit of the number of items allowed in a domain is surpassed. In this case we follow the same technique of adding a new domain.

Querying. This index cannot be queried directly using SimpleDB `SELECT` statements, since one cannot use them to search and retrieve data according to an item key. For this reason, we exploit this index through the `GetAttributes(D, k)` SimpleDB API call, where D is the domain name and k is the item name. This call returns the set of attributes associated with that item.

For each triple pattern of a SPARQL query the corresponding `GetAttributes` call is generated, giving as item name a concatenation of the bound values of the triple pattern. The URIs obtained through all the `GetAttributes` calls resulting from each triple pattern are those of the datasets on which the query must be evaluated.

For example, for the SPARQL query of Listing 1 we need

Table 6: Example of attribute-subset index

attribute-subset domain	
item key	(attr. name, attr. value)
S‖inria:article1	(articles, ϵ)
S‖inria:bar	(articles, ϵ)
S‖inria:book1	(books, ϵ)
S‖inria:book2	(books, ϵ)
S‖inria:foo	(books, ϵ)
S‖inria:_:uid1	(books, ϵ)
S‖labinria:lab1	(labs, ϵ)
S‖labInria:location	(labs, ϵ)
P‖inria:hasAuthor	(articles, ϵ), (books, ϵ)
P‖inria:hasName	(articles, ϵ), (books, ϵ)
P‖inria:hasNationality	(articles, ϵ), (books, ϵ)
P‖inria:hasTel	(books, ϵ)
P‖inria:hasRole	(books, ϵ)
P‖inria:hasContactInfo	(books, ϵ)
P‖labInria:hasName	(labs, ϵ)
P‖labinria:hasLocation	(labs, ϵ)
P‖inria:hasLocation	(labs, ϵ)
P‖labInria:hasGPS	(labs, ϵ)
O‖inria:bar	(articles, ϵ)
O‖“Bar”	(articles, ϵ)
O‖“American”	(articles, ϵ)
...	...
O‖“48.710715,2.17545”	(labs, ϵ)
SP‖inria:article1‖inria:hasAuthor	(articles, ϵ)
SP‖inria:bar‖inria:hasName	(articles, ϵ)
SP‖inria:bar‖inria:hasNationality	(articles, ϵ)
...	...
SP‖labInria:lab1‖labInria:hasName	(labs, ϵ)
PO‖inria:hasName‖“Bar”	(articles, ϵ)
PO‖inria:hasNationality‖“American”	(articles, ϵ)
PO‖inria:hasAuthor‖inria:Bar	(articles,ϵ)
...	...
PO‖labInria:hasGPS‖“48.710715,2.17545”	(labs, ϵ)
SO‖inria:article1‖inria:Bar	(articles, ϵ)
SO‖inria:bar‖“Bar”	(articles, ϵ)
SO‖inria:bar‖“American”	(articles, ϵ)
...	...
SO‖labInria:location‖“48.710715,2.17545”	(labs, ϵ)
SPO‖inria:bar‖inria:hasName‖“Bar”	(articles, ϵ)
...	...

to perform the following SimpleDB API calls:

`GetAttributes(attribute-subset, PO‖inria:hasAuthor‖"Foo")`
`GetAttributes(attribute-subset, P‖inria:hasContactInfo)`

If more than one domains have been created due to the limitation of SimpleDB, then we execute the `GetAttributes` to every domain. As in the previous cases, we then evaluate the SPARQL query to the the merge of the retrieved datasets.

Analytical cost model. Since for each triple of a dataset we create seven entries in SimpleDB, the size of the index of this strategy, let it be I_3, is $7 \times n \times T$, where n is the number of datasets and T is the maximum number of triples in a dataset.

For the query processing, we perform one lookup for each triple pattern appearing in the SPARQL query in the case where the SimpleDB limit has not been reached. In this case the number of lookups to SimpleDB when using the attribute-subset strategy is equal to q.

In the case where we have reached the SimpleDB limitation, we create more than one domain. Let d be the to-

Table 7: Domain-per-dataset indexing strategy

URI_{ds_i}	
item key	(attr. name, attr. value)
subject	(P‖property, object)
property	(O‖object, subject)
object	(S‖subject, property)

Table 8: Example of domain-per-dataset index

articles	
item key	(attr. name, attr. value)
inria:article1	(P‖inria:hasAuthor, inria:bar)
inria:hasAuthor,	(O‖inria:bar, inria:article1)
inria:bar	(S‖inria:article1, inria:hasAuthor)
	(P‖inria:hasName, "Bar"),
	(P‖inria:hasNationality, "American")
inria:hasName	(O‖"Bar", inria:bar)
inria:hasNationality	(O‖"American", inria:bar)
"American"	(S‖inria:bar, inria:hasNationality)
"Bar"	(S‖inria:bar, inria:hasName)

tal number of domains in SimpleDB. Then, the number of lookups to SimpleDB equals to $q \times d$.

4.4 Domain-per-dataset strategy

Indexing. According to this strategy, a SimpleDB domain is allocated for and named after each dataset with URI URI_{ds_i}. We use the subject, property, object values of each triple in the dataset as the item names. Within our notations, for each dataset U we create the following indexes: $(U|S|\underline{P}P|O)$, $(U|P|\underline{O}O|S)$ and $(U|O|\underline{S}S|P)$.

The organization of this index is illustrated in Table 7 while Table 8 shows a the organization of the index for a specific dataset of our example.

Handling SimpleDB limitations. SimpleDB limitations leads to constraining a single dataset to 256 property-object value pairs for each distinct subject value, 256 object-subject value pairs for each property value and 256 subject-property value pairs for each distinct object value. This means that each subject, property, object value can appear 256 times inside a dataset. In addition, each dataset is constrained to having 10^9 distinct triple element values. In case any of the above situations occur for a dataset we add a new domain for this dataset.

Querying. For each triple pattern appearing in a given SPARQL query, a `SELECT` SimpleDB query is defined and submitted to each existing domain. The resulting URI sets (one URI set for each triple pattern) are unioned and the query will be evaluated on the union of all such sets.

For instance, for the SPARQL query of Listing 1, we define the following SimpleDB queries for each domain i:

```
q1: SELECT *
    FROM domain_i
    WHERE property||inria:hasAuthor = "Foo";

q2: SELECT *
    FROM domain_i
    WHERE property||inria:hasContactInfo IS NOT NULL;
```

The datasets retrieved from queries `q1` and `q2` then merged and the SPARQL query is evaluated against the merged result.

Analytical cost model. In this indexing strategy we create three entries to SimpleDB for each triple of a dataset. Therefore, the size of the index of this strategy is $3 \times n \times T$, where n is the total number of datasets stored in S3 and T is the maximum size of a dataset.

For the query processing part of this strategy, we perform one lookup to each domain for each triple pattern appearing in the SPARQL query (if the SimpleDB limit has not been reached). If q is the number of triple patterns of a SPARQL query, the number of lookups to SimpleDB when using the domain-per-dataset strategy is $q \times n$.

In the case where we have reached the SimpleDB limitation, we create more domains per dataset. If d is the total number of domains in SimpleDB, then the number of lookups to SimpleDB is $q \times d$.

5. EXPERIMENTS

We have fully implemented our RDF data management architecture using all indexing strategies previously described. In this section, we describe a preliminary set of experiments conducted by deploying our system in the Amazon Web Services (AWS) environment.

5.1 Implementation and set up

We have used Java 1.6 to implement all the modules described in Section 3. The EC2 instance where we run our indexing module and query processor was part of the Ireland AWS facility and consisted of a 64-bit machine with 7.5 GB of memory, 2 virtual core with 4 EC2 Compute Units. An EC2 Compute Unit is equivalent to the CPU capacity of a 1.0-1.2 GHz 2007 Xeon processor. For the local RDF query evaluation we used the query processor ARQ 2.8.8 with Jena 2.6.4.

We used synthetic RDF data generated by the SP^{2}Bench generator [20] which produces data based on the DBLP bibliography schema. We created 10 datasets consisted of 10.000 each and used a set of queries which are SPARQL BGP queries obtained by some simplification of the SP^{2}Bench queries. For instance, we limited the queries to their BGP part, or modified them so that they would all have non-empty results when evaluated directly on the data[1]. The queries we used had from 1 to 8 triple patterns.

5.2 Indexing

In this section we study the performance of our four RDF indexing strategies, by measuring the performance of inserting index entries into SimpleDB. In this set of experiments we used 10 datasets of 10.000 triples each (i.e., 100.000 triples in total). We had 120 SimpleDB domains available to us for the experiments described here. Therefore, for the attribute-based and attribute-pair indexes the maximum number of domains that can be allocated for each domain type (i.e., S/P/O for attribute-based or SP/PO/OS for attribute-pair) was set to 30. After indexing all 10 datasets, the numbers of domains allocated were as follows:

- For the attribute-based strategy: 9 S domains, 1 P

[1]The full semantics of a SPARQL query on an RDF database should contain answers both from the explicit triples, and the implicit ones which are derived from the explicit triples using various RDF inference rules [17]. In the work described here, we have not yet considered cloud-based reasoning; this is part of our future work.

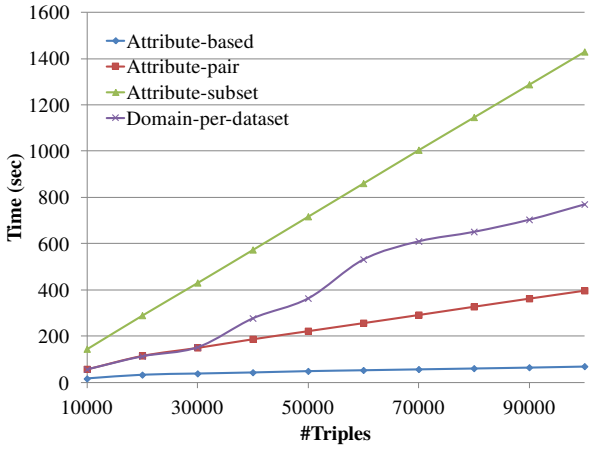

Figure 4: Indexing time.

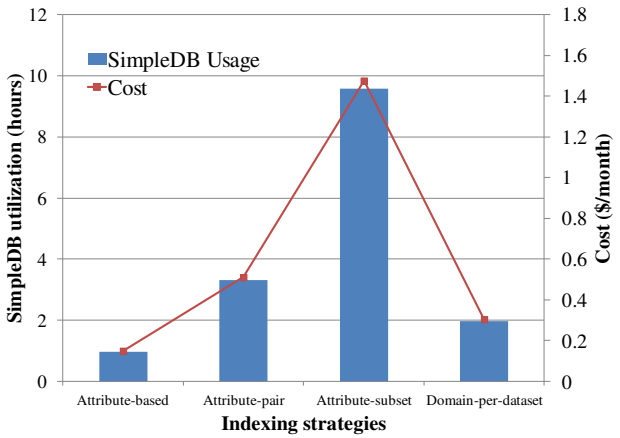

Figure 5: SimpleDB utilization and cost.

domain and 22 O domains

- For the attribute-pair strategy: 30 SP domains, 22 PO domains and 30 OS domains
- For the attribute-subset index: 1 domain;
- For the domain-per-dataset index: 10 domains, one for each dataset.

Because the SimpleDB limitations for the first two strategies are very restrictive even for small datasets, the domains were partitioned throughout our indexing experiments (starting from the smallest dataset of 10.000 triples).

For each indexing strategy we measure the time from the moment we start indexing the data, until the moment the index has been completely built in SimpleDB. Figure 4 shows the time required to build each index as the number of stored RDF triples increases. Note that we have used the `BatchPutAttributes` operation provided by SimpleDB which inserts to a single domain 25 items at a time, in a transactional fashion. We observe from the graph that the time required for the index construction grows linearly with the number of triples stored. As already shown by the analytical cost model of our indexing strategies, the attribute-subset index is the most time-consuming one since for each triple it inserts seven items into SimpleDB. On the other hand, the attribute-based index which defines only one attribute name for each item is more efficient. The attribute-pair strategy uses more domains and creates more unique attribute-value pairs that should be inserted in SimpleDB, and thus, is more expensive than the simple attribute-based approach. Finally, the domain-per-dataset index inserts each time data to a specific domain and does not scale well as the number of datasets stored in S3 increases. While it performs similar to the attribute-pair index up to 3 datasets, the time then increases rapidly for more datasets.

Figure 5 shows the total machine utilization of SimpleDB together with the cost for indexing all 10 datasets. Amazon charges 0.154 dollars per hour of utilization of a SimpleDB machine located in their Ireland facility. The attribute-based indexing strategy requires less machine utilization time and is thus more cost-efficient as well. On the other hand, the attribute-subset index is more expensive since it creates many more entries in SimpleDB than the rest of the indexes.

Table 9: Number of index look-up calls to SimpleDB, for each query and indexing strategy.

Query	#q	#c	Attr-based	Attr-pair	Attr-subset	Domain per dataset
Q1	1	1	1	22	1	10
Q2	2	3	2	44	2	20
Q3	2	2	2	44	2	20
Q4	2	3	24	44	2	20
Q5	3	4	25	66	3	30
Q6	4	5	26	88	4	40
Q7	8	9	30	176	8	80

Finally, while the attribute-pair and domain-per-dataset indexes create about the same attribute-value pairs to insert in SimpleDB, in the attribute-pair index we create many more domains and thus consume more of SimpleDB resources.

5.3 Querying

In this section, we present our preliminary results when evaluating various SPARQL BGP queries. The characteristics of the queries we used are shown in Table 9, where $\#q$ is the number of triple patterns and $\#c$ is the number of constant values each query contains. For this set of experiments, we have stored 10 RDF datasets in S3, each one consisting of 10.000 triples, and built the four indexes for this data. The number of domains created for each index is as described in Section 5.2.

In Table 9 we also depict the number of SimpleDB calls that were made for retrieving the appropriate URIs in each indexing strategy, i.e., `SELECT` queries for the attribute-based, attribute-pair and domain-per-dataset index, and `GetAttributes` calls for the attribute-subset index. This table verifies our analytical cost model which shows that the number of lookups depends on the number of triple patterns in each query, as well as on the number of created domains. For example, query Q7 which consists of 8 triple patterns requires the largest number of lookups compared to the rest of the queries in all indexing strategies. Moreover, since the attribute-subset index consists of only one domain, the number of calls performed to SimpleDB for any SPARQL query is smaller than the calls performed by the other indexing strategies. The attribute-pair index, which allocates

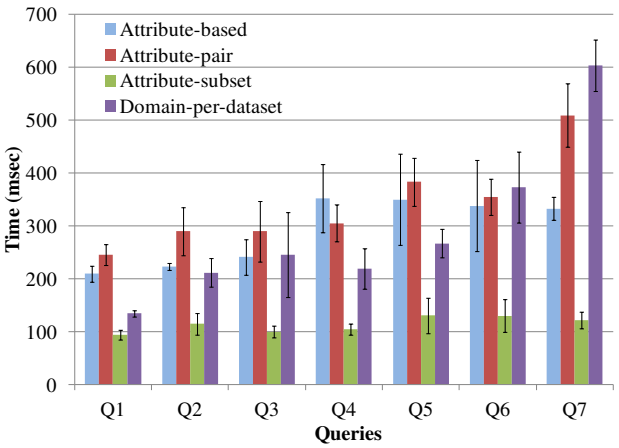

Figure 6: Index exploitation time.

the largest number of SimpleDB domains, sustains a great amount of SimpleDB calls for any kind of queries. Observe that within AWS, calls to different SimpleDB domains are evaluated in parallel, and this parallelization also benefits our work, however, we did not attempt to split and parallelize computations beyond that; we are interested to do so as part of our future work.

In this experiment, we have also measured the time required for retrieving the final set of URIs required to evaluate a SPARQL query (index exploitation time). We show the index exploitation time in Figure 6 for each indexing strategy and for various SPARQL queries. This time includes the time to build the appropriate `SELECT` queries or `GetAttributes` calls, the time required by SimpleDB to provide the results of these queries/calls, and the time for intersecting the URI sets thus obtained, whenever the look-up strategy requires such post-processing.

All measurements are averaged over 10 runs. Since previous studies established that the performance of Amazon EC2 performance may vary significantly over time [18, 7], we also depict the 95% confidence intervals. The attribute-subset indexing strategy outperforms the other strategies for all queries because it imposes the least amount of calls to SimpleDB. The attribute-based and attribute-pair indexes exhibit a similar performance with the former to perform slightly better because of the smaller amount of SimpleDB lookups. However for some queries (e.g., Q5 and Q6) the confidence intervals have a large overlap meaning that a sorting between the two strategies for such queries is not possible. Finally, although the domain-per-dataset indexing strategy gives better results than the aforementioned two indexes for most of the queries, it exhibits a very poor performance for queries which contain more than 5 triple patterns. This results from the large number of `SELECT` queries posed to the same domains, which after a certain number of requests become a bottleneck.

5.4 Experiments conclusion

Our preliminary results show the feasibility and efficiency of our architecture as well as the performance of our proposed indexing strategies. Comparing the indexing strategies highlights a trade-off between the cost of the creation of an index (both monetary and time cost) and the efficiency on the lookup process. A prominent example of this trade-off is the attribute-subset indexing strategy which is the most expensive index to build but gives the best performance while querying. Using bigger datasets and more heterogeneous data is our next step of experimentation.

6. RELATED WORK

Significant attention has been paid recently to RDF stores using cloud-based services. One system closely related to our work is Stratustore [21], an RDF store that uses Amazon's SimpleDB as an RDF store back-end in combination with Jena's API. Stratustore indexes all triples in SimpleDB using the subjects of the triples as items, the properties as attribute names and the objects as the values of the attributes. A drawback of this approach is that SPARQL queries having a variable in the property position cannot be answered. The authors propose to insert one more entry per triple having as attribute names the objects with values the properties but this leads to a great increase in storage. The evaluation of Stratustore is performed by running queries with up to 20 simultaneously Stratustore instances. Results show that performance is not competitive with other RDF stores such as Virtuoso. This is caused by the joins required for complex queries which have to be performed at the client side. However, as the number of Stratustore instances grows, the throughput of the system also increases.

The CumulusRDF [11] system uses Apache Cassandra, a nested key-value store, as a triple store back-end and proposes two different indexing strategies for storing RDF triples in Cassandra. The authors of [11] propose a hierarchical indexing scheme using supercolumns where all six combinations of subject, property, object are built-in indexes. In the second indexing scheme, called flat layout, simple columns are used where three main indexes are required together with a secondary index for several cases. CumulusRDF is evaluated in 8 machines using an instance of the DBPedia dataset and the queries used were only single triple pattern lookups. The authors conclude that their flat layout approach outperforms the hierarchical one. However, both Stratustore and CumulusRDF focus on providing full indexing capabilities in order to be able to answer SPARQL queries from indexes. Different from this approach, our main concern is to use the indexes for efficiently retrieving a smaller subset of datasets from which we are able to extract the answer to SPARQL queries using any in-memory RDF store.

Dydra [2] is an RDF store relying on the Amazon EC2 infrastructure which provides a SPARQL endpoint to query the data stored. Although Dydra addresses an RDF data management problem similar to our, there is not much information available revealing the details of their approach.

Various works using MapReduce and related technologies have appeared in the literature as well. These works focus on developing large-scale RDF stores using the MapReduce paradigm. [13] is one of the first works to introduce cloud computing in the area of Semantic Web. It gives some preliminary experimental results using Apache Hadoop, a very popular implementation of MapReduce and Pig, a tool that translates queries expressed in Pig Latin to MapReduce jobs. In [6] the authors use Hadoop and propose a specific storage scheme that partitions RDF files into smaller ones to be stored in HDFS, the file system of Hadoop. They also use summary statistics to determine the best plan to evaluate a SPARQL query. [14] considers the evaluation of SPARQL

basic graph pattern queries in a MapReduce framework. Specifically, the authors propose a multi-way join algorithm to process SPARQL queries efficiently, as well as two methods to select the best query plan for executing the joins. Experiments were conducted with Cloudera's Hadoop distribution on the Amazon EC2. Finally, [19] presents a method to map SPARQL queries to Pig Latin queries.

7. CONCLUSIONS AND PERSPECTIVES

We have presented a novel architecture for the distributed management of RDF data stored in cloud infrastructures. We designed indexing techniques for retrieving the appropriate RDF files related to a specific query. We chose Amazon Web Services as a platform and implemented all our indexing strategies. We presented an analytical cost model and an experimental evaluation of both the indexing and the querying process of our strategies.

We plan to consider several extensions of this work.

Since the results of our experimental evaluation are only preliminary, we plan to continue experimenting with bigger, real-world datasets is our ongoing work. Interestingly, Amazon has recently launched a new data service, namely DynamoDB, that lifts some of the limitations we encountered with SimpleDB. However, it brings its own limitations, and in particular specific restrictions on the way one is allowed to switch between different levels of service. Dynamo DB is in beta stage for now; we may consider it as a replacement for SimpleDB once its functioning is consolidated and better known.

A full solution for a cloud-based large RDF store must include an intelligent pricing model for the store as well as for the index, reflecting the usage of cloud resources. In this work we have outlined the monetary costs associated to the index, which are a first ingredient of a comprehensive pricing scheme. How indexes built once in a cloud amortize over several users has been discussed in [8].

A direction we have not considered in this work is the parallelization of the task of evaluating a query on a given RDF dataset. This is obviously interesting, since the parallel processing capabilities of a cloud may lead to very short response times. However, when considering RDF data, a first significant obstacle consists of the difficulty of finding a way to partition the data, in order to enable different processors to work each partition in parallel. This is complex not only because RDF consists of "inextricable" graphs, but also because conceptually, as explained in Section 2, queries should be evaluated over the merge of all the datasets residing in the store. This concurs to make the problem quite challenging.

Finally, RDF is a well-known data model that allows for extracting inferred information from the already existing data. An interesting and important task therefore, is to adapt our architecture and indexing strategies so that RDFS reasoning can be also supported.

8. REFERENCES

[1] Amazon Web Services. http://aws.amazon.com/.

[2] Dydra. `http://dydra.com/`.

[3] A. Alexandrov, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp, and D. Warneke. MapReduce and PACT - Comparing Data Parallel Programming Models. In *BTW*, 2011.

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

[5] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating Systems Design and Implementation*, 2004.

[6] M. F. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham. Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools. In *3rd International Conference on Cloud Computing*, 2010.

[7] A. Iosup, N. Yigitbasi, and D. Epema. On the Performance Variability of Production Cloud Services. In *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2011.

[8] V. Kantere, D. Dash, G. Gratsias, and A. Ailamaki. Predicting Cost Amortization for Query Services. In *SIGMOD*, 2011.

[9] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation, 2004.

[10] R. Krummenacher, K. Aberer, A. Kiryakov, and R. Kanagasabai. Workshop on Semantic Data Management: A Summary Report. *SIGMOD Record*, 39(3):24–26, 2010.

[11] G. Ladwig and A. Harth. CumulusRDF: Linked Data Management on Nested Key-Value Stores. In *SSWS*, 2011.

[12] F. Manola and E. Miller. RDF Primer. W3C Recommendation, February 2004.

[13] P. Mika and G. Tummarello. Web Semantics in the Clouds. *IEEE Intelligent Systems*, 23(5):82–87, 2008.

[14] J. Myung, J. Yeon, and S.-g. Lee. SPARQL Basic Graph Pattern Processing with Iterative MapReduce. In *Workshop on Massive Data Analytics on the Cloud*, 2010.

[15] T. Neumann and G. Weikum. Scalable Join Processing on Very Large RDF Graphs. In *SIGMOD*, 2009.

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

[17] E. Prud'hommeaux and A. Seaborn. SPARQL Query Language for RDF. W3C Recommendation, `http://www.w3.org/TR/rdf-sparql-query/`, 2008.

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

[19] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *SWIM*, 2011.

[20] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^{2}Bench: A SPARQL Performance Benchmark. In *ICDE*, 2009.

[21] R. Stein and V. Zacharias. RDF On Cloud Number Nine. In *4th Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic*, May 2010.