Chapter 2

SPARQL Query Processing in the Cloud

Francesca Bugiotti

Università Roma Tre, Italy and Inria Saclay, France

Jesús Camacho-Rodríguez

Université Paris-Sud and Inria Saclay, France

François Goasdoué

Université Paris-Sud and Inria Saclay, France

Zoi Kaoudi

Inria Saclay and Université Paris-Sud, France

Ioana Manolescu

Inria Saclay and Université Paris-Sud, France

Stamatis Zampetakis

Inria Saclay and Université Paris-Sud, France

2.1 Introduction

Since its emergence, cloud computing has been massively adopted due to the scalability, fault-tolerance and elasticity features it offers. 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 recent works have focused on the performance and cost analysis of cloud platforms, and on the extension of the services that they provide. For instance, [9] focuses on extending public cloud services with basic database primitives, while extensions for the MapReduce paradigm [14] are proposed in [5] for efficient parallel processing of queries in cloud infrastructures.

At the same time, there is an abundance of RDF data published on the Web nowadays. DBpedia, BBC and Open Government Data are only a few examples of the constantly increasing Linked Open Data cloud (LOD)[1]. To exploit such large volumes of Linked 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 the data into the repository where further processing can be applied. Efficient systems have been devised in order to handle large volumes of RDF data in a centralized setting, with RDF-3X [22] being among the best-known. However, as the amount of data continues to grow, it is no longer feasible to store the entire linked data sets on a single machine and still be able to scale to multiple and varied user requests. Thus, such huge data volumes have raised the need for distributed storage architectures and query processing frameworks, such as the ones provided by P2P networks and discussed in Chapter **??** or federated databases discussed in Chapter **??**.

In this chapter we focus on recent proposals for distributed and parallel query processing techniques which are suited to cloud infrastructures. In particular, we present an architecture for storing RDF data within the Amazon cloud that provides efficient query performance, both in terms of time and monetary costs. We consider hosting RDF data in the cloud, and its efficient

[1] `lod-cloud.net`

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. At the core of our proposed architecture reside RDF indexing strategies that allow to direct queries to a (hopefully tight) superset of the RDF datasets which provide answers to a given query, thus reducing the total work entailed by query execution. This is crucial as, in a cloud environment, the total consumption of storage and computing resources translates into monetary costs.

This chapter is organized as follows. First, we provide a brief survey of the existing works which aim at storing and querying large volumes of RDF data in clouds, in Section 2.2. We introduce in Section 2.3 the different parts of the Amazon Web Services (AWS) cloud platform that we use in our work and our architecture. Then we focus on our specific indexing and query answering strategies in Section 2.4. Finally, we provide relevant implementation details in Section 2.5 and experiments that validate the interest and performance of our architecture in Section 2.6.

2.2 Classification of RDF Data Management Platforms in the Cloud

We first review the state-of-the-art works in cloud-based management of RDF data. The field is very active and numerous ideas and systems have appeared recently. We present them classified according to the way in which they implement three fundamental functionalities: data storage, query processing, and reasoning, which (going beyond cloud-based data storage and querying) is specific to the RDF context.

2.2.1 Cloud-based RDF storage

A first classification of existing platforms can be made according to their underlying data storage facilities. From this perspective, existing systems can be split into the following categories:

- systems which use existing "NoSQL" key-value stores [12] as back-ends for storing and indexing RDF data;
- systems relying on a distributed file system, such as HDFS, for warehousing RDF data;
- systems relying on other storage facilities, such as a set of independent single-site RDF stores, or data storage services supplied by the cloud providers.

Systems of the first category use the underlying key-value stores for both storage and indexing of RDF data. The most commonly used indexing scheme is one already adopted in centralized RDF stores, which either uses all six permutations of subject, predicate, object of the RDF triples or a subset of them. Because of the key-value pair data model of the key-value stores, usually one table is created for each one of these permutations. Representatives of this category include systems such as Rya [25] which uses Apache Accumulo [2], CumulusRDF [21] based on Apache Cassandra [3], Stratustore [30] which relies on Amazon's SimpleDB [1], and H_2RDF [23] or MAPSIN [28] built on top of HBase [4]. Depending on the specific capabilities of the underlying key-value store, different designs have been chosen for the key and values. In H_2RDF, the first two elements are used as the key, and the last one as the attribute value, while in Rya, all three elements are used as the key while the value remains empty. In our own AMADA platform [6], we use the first element as the key, the second as the attribute name and the last one as the attribute value, as we will detail further on in this chapter. Trinity.RDF [33], a recent system developed in Microsoft, also belongs to this category. Trinity.RDF is a graph engine for RDF data based on Trinity [29], a distributed in-memory key-value store. Although it takes advantage of the graph structure of RDF, essentially it also indexes RDF data based on three different permutations of subject, predicate, object. While key-value stores are ideal for matching individual triple patterns, join operations are not supported by the key-value stores and thus, the join evaluation should be implemented and performed out of the store. This may raise performance issues.

The second category comprises platforms, such as those described in [17, 26, 27], that use the Hadoop Distributed File System (HDFS) to store RDF data. In [27] RDF datasets are simply stored in HDFS as they are provided by the user, while in [17] a specific partitioning scheme is used which groups triples based on their predicate values and the type of their objects. These systems are built to make the most out of the parallel processing capacities provided by the underlying MapReduce paradigm. They are able to handle large data chunks (files), however they do not provide fine-grained access to this data.

Within the third category lies [16], where RDF data is partitioned based on a graph partitioning tool and each partition is stored in one machine within a centralized RDF store. While this approach works well for star-shaped queries, it needs a big amount of data replication for more complicated ones which makes it not scalable to very large datasets. In our work [6, 10], we use a mixed approach with data residing in Amazon's storage service (S3) and a full data index built in Amazon's key-value store. As we explain in the rest of the chapter, this approach highly depends on the data partitioning and is suitable for selective queries.

2.2.2 Cloud-based SPARQL query processing

A second relevant angle of analysis of cloud-based RDF platforms comes from their strategy for processing SPARQL queries. From this perspective, we identify two main classes:

- systems relying on the parallel programming paradigm of MapReduce [14];
- systems attempting to reduce or avoid altogether MapReduce steps. The reason is that while MapReduce achieves important parallel scalability, the start-up time of a MapReduce job is significant [13], an overhead which may be too high for interactive-style queries.

Systems such as [17, 27, 26] belong to the first class above where different MapReduce-based evaluation strategies are proposed. In [27] one MapReduce job is used for each join operation, while in [17, 26] the goal of the query evaluation is to heuristically reduce the number of MapReduce jobs by performing as many joins as possible in parallel. Using MapReduce for query processing is suitable for analytical-style queries but may cause a big overhead for interactive-style very selective queries.

In the second class we find systems relying on key-value stores which exploit the indices to efficiently find matches to the triple patterns of the query. Such systems typically gather the matches of the triple patterns in a single site and implement their own join operators [25, 30]. Works such as [6, 10, 16], which take advantage of existing RDF stores, also belong to this group. Trinity.RDF [33] is classified in this category as well; it uses a graph-oriented approach by exploring the distributed RDF graph in parallel. Finally, H_2RDF uses a hybrid approach depending on the selectivity of the query; for non-selective queries a MapReduce-based query plan is used, while for very selective queries data retrieved from the key-value store are joined locally.

2.2.3 RDF reasoning in the cloud

The first chapter has introduced the role of *inference* (or reasoning) and the importance of *implicit data* in an RDF data management context. From the perspective of their way to handle implicit data, cloud-based RDF data management platforms can be classified in three classes:

- pre-compute and materialize all implicit triples;
- compute the necessary implicit triples at query time;
- some hybrid approach among the two above, with some implicit data computed statically and some at query time.

In [31] the RDF(S) inference rules are used for precomputing the whole RDFS entailment using MapReduce jobs. In this case, query processing can

take place using regular query processing techniques and can be very efficient. On the other hand, computing the whole RDFS entailment causes a significant storage overhead and on the presence of data or schema updates the whole RDFS entailment should be recomputed again. On the contrary, in [32] the RDFS entailment is computed on demand based on a given triple pattern. However, no general query evaluation algorithms are presented. The only work that injects some RDFS entailment within query processing is [17] using query reformulation. Certainly, such an approach does not impose any storage overhead and is very flexible for updates but does incur an overhead during query time depending on the complexity of the query.

The remaining systems do not consider reasoning at all, which implies that they assume all the implicit data has been made explicit (through inference) and stored before evaluating queries. Our own AMADA platform is also currently based on this assumption, and in the remainder of the chapter, we will not further consider cloud-based RDF reasoning. Given the complexity of distributed reasoning and the interest in large-scale Semantic Web data, we expect this to attract significant interest in the near future.

2.3 AMADA: RDF Data Repositories in the Amazon Cloud

We consider hosting RDF data in the cloud, and its 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. We envision an architecture where large amounts of RDF data reside in an elastic cloud-based store, and focus on the task of efficiently routing queries to only those datasets that are likely to have matches for the query.

Our proposal has been implemented using the Amazon Web Services (AWS) cloud platform [1], one of the most prominent commercial cloud platforms today, and our platform is called AMADA. AWS provides elastic scalable cloud-based services that organizations and individuals can use to develop their own applications.

In the following, Section 2.3.1 introduces the Amazon services used by AMADA, while Section 2.3.2 presents our proposed architecture built on top of it.

2.3.1 Amazon Web Services

In AMADA, we store RDF files in Amazon Simple Storage Service (S3) and use Amazon DynamoDB for storing the index. SPARQL queries are evaluated against the RDF files retrieved from S3, within the Amazon Elastic Compute Cloud (EC2) machines and the communication among these components is done through the Simple Queue Service (SQS).

In the following we describe the services used by our architecture. We also introduce the parameters used by AWS for calculating the pricing of each of its services; the actual figures are shown in Table 2.1 (the notations in the table are explained in the following subsections). More details about AWS pricing can be found in [1].

$ST^{\$}_{m,GB}$ = \$0.125	$IDX^{\$}_{m,GB}$ = \$1.13
$STput^{\$}$ = \$0.000011	$IDXput^{\$}$ = \$0.00000032
$STget^{\$}$ = \$0.0000011	$IDXget^{\$}$ = \$0.000000032
$VM^{\$}_{h,l}$ = \$0.38	$QS^{\$}$ = \$0.000001
$VM^{\$}_{h,xl}$ = \$0.76	$egress^{\$}_{GB}$ = \$0.12

TABLE 2.1: AWS Ireland costs as of February 2013.

2.3.1.1 Simple Storage Service

Amazon Simple Storage Service (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.

To retrieve an object from S3, the bucket containing it should be accessed, and within bucket the object can be retrieved by its name. S3 allows to access the metadata associated to an object without retrieving the complete entity. Storing objects in one or multiple S3 buckets has no impact on the storage performance.

Pricing. Each read file operation costs $STget^{\$}$, while each write operation costs $STput^{\$}$. Further, $ST^{\$}_{m,GB}$ is the cost charged for storing 1 GB of data in S3 for one month. AWS does not charge anything for data transferred to or within their cloud infrastructure. However, data transferred out of the cloud incurs a cost: $egress^{\$}_{GB}$ is the price charged for transferring 1 GB.

2.3.1.2 DynamoDB

Amazon DynamoDB[2] is a key-value based store that provides fast access to small objects, ensuring high availability and scalability for the data stored [15].

Figure 2.1 outlines the structure of a DynamoDB database. A DynamoDB

[2] `http://aws.amazon.com/dynamodb/`

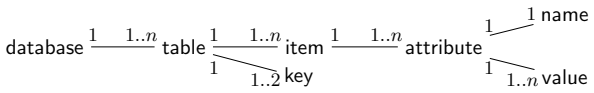

FIGURE 2.1: Structure of a DynamoDB database.

database is organized in *tables*. Each table is a collection of *items* identified by a primary composite key. Each item contains one or more *attributes*; in turn, an attribute has a *name* and a set of associated *values*[3].

An item in a table can be accessed by providing its composite key which consists of two attributes: the *hash key* and the *range key*. Internally, DynamoDB maintains an unordered hash index on the hash key and a sorted range index on the range key. Further, it partitions the items of a table across multiple servers according to a hash function defined on the hash key.

DynamoDB provides a very simple API to execute read and write operations. The methods that we use in our platform include[4]:

- `PutItem(T, Key(hk, [rk]), (a,v)+)` creates a new item in the table `T` containing a set of attributes `(a,v)+` and having a key composed by a hash key `hk` and range key `rk`, or replaces it if it already existed. Specifying the range key is optional.
- `BatchWriteItem(item+)` puts and/or deletes up to 25 `Items` in a single request, thus obtaining better performance.
- `GetItem(T, Key(hk, [rk]), (a)*)` returns the item having the key `Key(hk, [rk])` in table `T`. Once again, specifying the range key is optional. It is possible to retrieve only a subset of the attributes associated to an item by specifying their names `(a)*` in the request.

DynamoDB does not provide support for operations executed on data from different tables. Therefore, if combining data across tables is required, the results from the respective tables have to be combined at the application layer. The read and write throughputs for each table in DynamoDB are set independently by the developer. AWS ensures that operations over different tables run in parallel therefore the maximum performance (and a reduced monetary cost) can be obtained splitting data across multiple tables. In addition, DynamoDB does not follow a strict transactional model based on locks or timestamps. Instead, it implements an eventual consistency model[5] that privileges high availability and throughput at the expense of strong synchronization.

[3]An item can have any number of attributes, although there is a limit of 64 KB on the item size.

[4]For the sake of readability in the rest of the chapter we will refer to those operations simplifying the notation and the parameters, the full specification of those operations can be found in the DynamoDB documentation [1].

[5]`http://aws.amazon.com/dynamodb/faqs/#What_is_the_consistency_model_of_Amazon_DynamoDB`

Pricing. Each item read and write API request has a fixed price, $IDXget^{\$}$ and $IDXput^{\$}$ respectively. One can adjust the number of API requests that a table can process per second. Further, DynamoDB charges $IDX^{\$}_{m,GB}$ for storing 1 GB of data in the index store during one month.

2.3.1.3 Elastic Compute Cloud

Amazon Elastic Compute Cloud (EC2) provides virtual machines, called *instances*, which users can rent to run their applications on. A developer can store in AWS the image or static data containing the software that an instance should run once it is started. Then, it can launch instances e.g. *large*, *extra-large*, etc that have different hardware characteristics, such as CPU speed, RAM size, etc.

Pricing. The EC2 utilization cost depends on the kind of virtual machines used. In our system, we use large (l) and extra-large (xl) instances. Thus, $VM^{\$}_{h,l}$ is the price charged for using a large instance for one hour, while $VM^{\$}_{h,xl}$ is the price charged for using an extra-large instance for one hour.

2.3.1.4 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.

Pricing. $QS^{\$}$ is the price charged for any request to the queue service API, including send message, receive message, delete message, renew lease etc.

2.3.2 General architecture

We envision an architecture where large amounts of RDF data reside in an elastic cloud-based store, and 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, total work also translates in financial costs! 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.

RDF datasets are stored in S3 and each dataset is treated as an uninterpreted BLOB object. As explained in Section 2.3.1.1, it is necessary to associate a key to every resource stored in S3 in order to be able to retrieve it. Thus, we assign to each dataset: (i) a URI consisting of the bucket name denoting the place where it is saved; (ii) and the name of the dataset. The combination of both (i), (ii) describes uniquely the dataset. In general we indicate

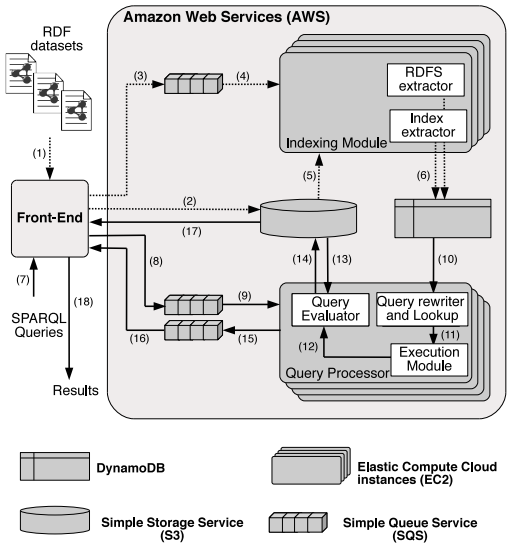

FIGURE 2.2: AMADA architecture based on AWS components.

as URI_{ds_j} the URI associated to the dataset j. The indexes use this URI for retrieving the correct datasets. Finally, we store our indexes in DynamoDB, as it provides fast retrieval for fine-granularity objects.

An overview of our system architecture is depicted in Figure 2.2. A user interaction with our system can be described as follows.

The user submits to the *front-end* component an RDF dataset (1) and the front-end module stores the file in S3 (2). The front-end then creates a message containing the reference to the dataset and inserts it to the *loader request queue* (3). Any 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, creates the index entries and inserts them in DynamoDB (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). Then, the query processor performs a lookup in the index stored in DynamoDB (10). Depending on the indexing strategy, the lookup will return data that can be used to answer the query directly (without scanning any data stored in S3) or data that can be used to find out which datasets contain information to answer the query. Any processing required on the data retrieved from DynamoDB is performed by the *execution module* (11). If a final results extraction step is required, 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). The results of the

query are written to S3 (14) and a message is created and inserted into the *query response queue* (15). Finally, the front-end receives this message (16), retrieves the results from S3 (17) and the query results are returned to the user and deleted from S3 (18).

Although we use Amazon Web Services, our architecture could be easily adapted to run on top of other cloud platforms that provide similar services. Examples of such platforms include Windows Azure[6] and Google Cloud[7].

2.4 RDF Storing and SPARQL Querying within AMADA

In the following, we introduce the different strategies we have devised to answer SPARQL queries efficiently within AMADA, both in terms of time and monetary costs.

First, Section 2.4.1 introduces the running example to demonstrate the indexing strategies used in AMADA. Section 2.4.2 proposes a strategy that loads all the RDF data in DynamoDB, making it possible to answer queries only by looking up in the index. Finally, Section 2.4.3 presents indexing strategies that allow to select the RDF datasets that should be retrieved from S3 to answer a given SPARQL query.

2.4.1 Data model and running example

In the following, we consider RDF graphs consisting of triples of the form `(s,p,o)`. We use the expression *RDF graphs* and *RDF datasets* interchangeably. We also use the notion of *RDF merge* (recall Definition 4 in Chapter 1) to integrate two or more RDF graphs without there being blank node conflicts.

Regarding the query language, we focus on a subset of SPARQL. We deal with `SELECT` or `ASK` queries and we consider basic graph pattern (BGP) queries, i.e., the conjunctive fragment of SPARQL allowing to express the core Select-Project-Join database queries. We also exclude queries with unbound triple patterns as such queries cannot benefit from any indexing strategy. Support of more complex SPARQL queries, such as OPTIONAL, UNION, etc., depends on the operations supported by the execution module of the query processor. We evaluate a BGP query B over the RDF merge G of several RDF graphs using the semantics presented in Chapter 1, i.e., the results of the query are equal to $[[B]]_G$.

Throughout this chapter we rely on a simple running example consisting of three linked RDF datasets and a SPARQL query. The data consists of (i) the *publications* dataset that contains information about publications, (ii) the *au-*

[6] `http://www.windowsazure.com`
[7] `https://cloud.google.com/`

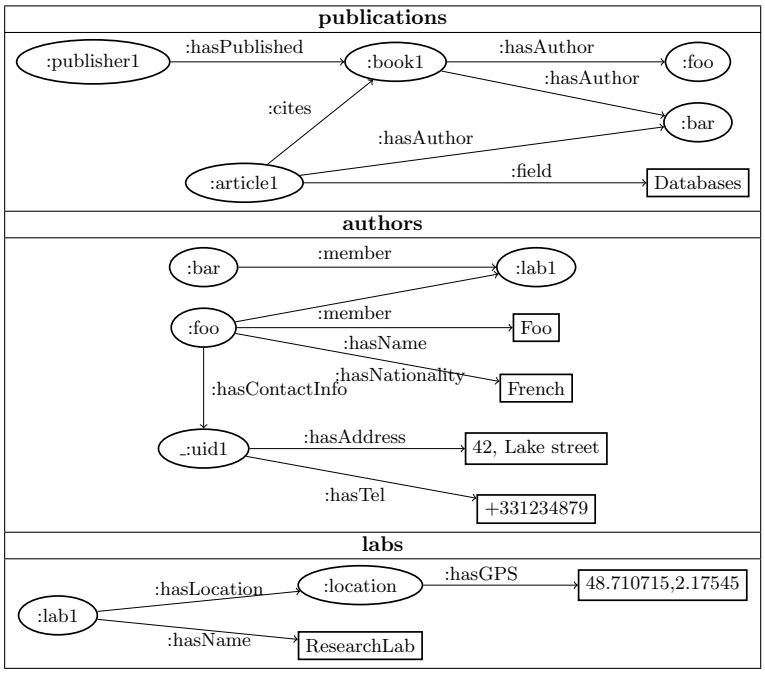

FIGURE 2.3: Graph representation of the example RDF data.

```
SELECT ?pub ?author
WHERE {
   ?pub :hasAuthor ?author .
   ?author :member :lab1 .
}
```

FIGURE 2.4: SPARQL example query Q1

thors dataset that contains information about the authors of the publications, and (iii) the *labs* dataset that contains information about the labs the authors are member of. The content of these datasets is depicted in Figure 2.3[8]. Note that in our work, we use the datasets as uploaded by the users. However, if very large data sets are uploaded, one could envision partitioning them into smaller ones in a divide-and-conquer fashion, to make query processing more efficient; this is an orthogonal extension that we do not discuss here. The SPARQL query used in our example is depicted in Figure 2.4. The query asks for the publications of the authors who are members of `:lab1`.

[8]For convenience, we omit the namespaces and denote by the prefix ':' that a node of an RDF graph is a URI.

RDF			
S	subject	$\underline{S}$	token string for subject
P	predicate	$\underline{P}$	token string for predicate
O	object	$\underline{O}$	token string for object
U	URI	$\underline{N}$	constant table name
T	RDF term		

Datasets	
$\mathcal{D}$	set of RDF datasets
ds	a dataset in $\mathcal{D}$
$\|ds\|$	#triples in ds
q	a SPARQL query
$\|q\|$	#triple patterns in q

DynamoDB	
N	table name
K	item key
A	attr. name
V	attr. value

TABLE 2.2: Notation.

Notation. In the following, we denote by $\mathcal{D}$ the RDF datasets that need to be indexed by our system. In addition for any dataset $ds \in \mathcal{D}$, we use $|ds|$ to represent the total number of triples of this dataset. In turn, we denote by q a SPARQL query, and $|q|$ is the number of triple patterns in q.

To simplify our presentation, we describe each indexing strategy with respect to the four levels of information that DynamoDB allows us to use, namely table name (N), item key (K), attribute name (A) and attribute value (V). Each indexing strategy can be represented by a concatenation of four |-separated symbols, specifying which information item is used in the table name, item key, attribute name and attribute value, respectively.

To index RDF, we use the values of subjects (S), predicates (P) and objects (O), and URIs (U) of the RDF datasets residing in S3. Moreover, we will also use RDF *terms* (T) to denote any among subject, predicate and object. Thus, a term t appears in a dataset D iff t appears as a subject, predicate or object in D. 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 use to differentiate data that needs to be treated as a subject, predicate, and object, respectively. 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 corresponding to "subject" and "predicate"). Similarly, we will use a token string denoted by $\underline{N}$ to represent a constant table name. Table 2.2 summarizes all the notation we use in the rest of the chapter.

Analytical cost model. For comparing the different strategies, we focus on the index size and query look-ups. Thus, for each strategy, we will present analytical models for calculating data storage size and query processing costs in the worst case scenario. This scenario will be described for each of the strategies.

In this chapter, we do not consider a complete cost model of AMADA that would include e.g. local processing, data transfer, etc. However, a full formalization of the monetary costs associated to our architecture can be found in [11].

$\underline{S}$ **table**	
item key	(attr. name, attr. value)
subject	(*predicate*, *object*)
$\underline{P}$ **table**	
item key	(attr. name, attr. value)
predicate	(*subject*, *object*)
$\underline{O}$ **table**	
item key	(attr. name, attr. value)
object	(*predicate*, *subject*)

(a) QAS indexing strategy

$\underline{S}$ **table**	
item key	(attr. name, attr. value)
:publisher1	(:hasPublished, :book1)
:article1	(:cites, :book1), (:hasAuthor, :bar) (:field, "Databases")
...	...
$\underline{P}$ **table**	
item key	(attr. name, attr. value)
:hasPublished	(:publisher1, :book1)
:hasAuthor	(:book1, :foo) (:book1, :bar) (:article1, :foo)
...	...
$\underline{O}$ **table**	
item key	(attr. name, attr. value)
:foo	(:hasAuthor, :book1)
:bar	(:hasAuthor, :book1) (:hasAuthor, :article1)
"Database"	(:field, :article1)
...	...

(b) Example of QAS index

TABLE 2.3: Query-answering indexing strategy

2.4.2 Answering queries from the index

The first strategy we describe relies exclusively on the index to answer queries. This is achieved by inserting the RDF data completely into the index, and answering queries based on the index without requiring accessing the dataset. We denote this strategy by QAS and we describe it in more details below.

Indexing. A DynamoDB table is allocated for each RDF triple attribute: one for the subjects, one for the predicates and one for the objects. We use the subject, predicate, object values of each triple in the datasets as the item keys in the respective DynamoDB table, and as attribute (name, value) pairs, the pairs: (predicate, object), (object, subject) and (subject, predicate) of the triple. Thus, each entry in the table completely encodes an RDF triple, and all database triples are encoded in the indexes: ($\underline{S}|S|P|O$), ($\underline{P}|P|O|S$) and ($\underline{O}|O|S|P$).

The organization of this index is illustrated in Table 2.3(a) while Table 2.3(b) shows the organization of the index for the triples of our example.

Querying. When querying data indexed according to QAS, one needs to perform index look-ups in order to extract from the index sets of triples that match the triple patterns of the query, and then process these triples through relational operators (selections, projections and joins) which AMADA provides in its execution module of the query processor (Figure 2.2). In our im-

plementation, we have used our in-house relational operators of ViP2P [20] but any relational query processor supporting these operators could be used.

For each triple pattern appearing in a given BGP query, a `GetItem` DynamoDB call is executed. If the triple pattern has only one bound value then the call is done to the respective table, i.e., if the bound value is a predicate, the call is performed to the predicate table. Otherwise, if two values of the triple pattern are bound, we choose the most selective of them. Thus, we get from the index the least amount of values that match the triple pattern. In our implementation we use the following heuristic: objects are more selective than subjects, which in turn are more selective than the predicates. Alternatively, one could use statistics on the RDF dataset to order the look-ups according to cardinality estimations on the number of triples returned by each index look-up, intermediary result sizes etc. Those statistics can be calculated for each dataset at indexing time and stored in DynamoDB as well.

For each triple pattern t_i, the resulting attribute name-value pairs retrieved from DynamoDB form a relation R_i with two columns: one holding the attribute names and another the attribute values. If the triple pattern has only one bound value, the values of these columns contain the bindings of the variables of t_i. Otherwise, if t_i contains two bound values, a selection operation is used to filter out the values that do not match the triple pattern. These relations are then joined and the result forms the answer to the query.

For instance, consider the SPARQL query of Figure 2.4. First, we define the following DynamoDB requests:

```
r1: GetItem(P, :hasAuthor)
r2: GetItem(O, :lab1)
```

Request `r1` returns attribute name-value pairs (s_1, o_1) which form a relation R_1, while `r2` returns attribute name-value pairs (s_2, p_2) which form another relation R_2. Then, a selection operation is performed which requires all values of the second column of R_2 to be equal to the predicate `:member` (i.e., $\sigma_{2=:member}(R_2)$). The remaining values of the first column of R_2 are the bindings to the variable of the second triple pattern. Finally, a join is performed between the second column of R_1 and the first column of R_2 and the results of the join form the answer to the SPARQL query $Q1$, i.e., $[[Q1]]_G = R_1 \bowtie_{2=1} \pi_1(\sigma_{2=:member}(R_2))$.

Analytical cost model. We now analyze the cost of the QAS indexing strategy as well as the number of required lookups while processing a SPARQL query.

We assume that the number of distinct subject, predicate and object 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). In this indexing strategy we create three entries to DynamoDB for each triple in our dataset $ds \in \mathcal{D}$. Therefore, the size of the index of this strategy is $\sum_{ds \in \mathcal{D}} 3 \times |ds|$.

To process queries, we perform one lookup for each triple pattern appearing in the SPARQL query q. Thus, the number of lookups to DynamoDB is $|q|$.

$\underline{N}$ table	
item key	(attr. name, attr. value)
v_1	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ), ...
v_2	(URI_{ds_2}, ϵ), ...
v_3	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ), ...

(a) RTS indexing.

$\underline{N}$ table	
item key	(attr. name, attr. value)
:publisher1	(*publications*, ϵ)
:book1	(*publications*, ϵ)
:article1	(*publications*, ϵ)
:bar	(*authors*, ϵ), (*publications*, ϵ)
:foo	(*authors*, ϵ)
:lab1	(*authors*, ϵ), (*labs*, ϵ)
:location	(*labs*, ϵ)
:hasAuthor	(*publications*, ϵ)
:hasPublished	(*publications*, ϵ)
:member	(*authors*, ϵ)
:hasName	(*authors*, ϵ), (*labs*, ϵ)
...	

(b) Sample RTS index entries.

TABLE 2.4: RTS indexing strategy.

2.4.3 Selective indexing strategies

In this section, we present three strategies for building RDF data sets indexes within DynamoDB: the term-based strategy, the attribute-based strategy and, the attribute-subset strategy. Each strategy exploits a different technique for indexing the RDF datasets and uses DynamoDB tables, items and attributes according to a different pattern. The system uses these indexes in order to identify among all the RDF data sets, those which may contribute to answer a given query then loads them from S3 into an EC2 instance and processes the query there. In this paragraph we illustrate the indexing techniques, the query strategies and the combining result methods then in Section 2.6 we analyze how the performances vary, in each index, according to the dataset and query characteristics.

2.4.3.1 Term-based strategy

This first indexing strategy, denoted RTS, relies on the RDF terms found within the datasets stored in S3. This strategy does not take into account whether a certain term is found as a subject, predicate or object inside a dataset.

Indexing. For each RDF term (URI or literal) appearing in a dataset, one DynamoDB item is created with the value of this term as key, the URI of the dataset that contains this RDF term as attribute name, and a null string (denoted ϵ) as attribute value. The name of the dataset is also the URI allowing to access the RDF graph stored in S3. That is, the index is of the form: $(\underline{N}|T|U|\epsilon)$.

Table 2.4(a) depicts the general layout for this indexing strategy, where v_i are the values of the RDF terms. Table 2.4(b) illustrates the index obtained for the running example.

Querying. For each RDF term of a BGP query a `GetItem` look-up in the RTS index retrieves the URIs of the datasets containing a triple with the given term. For each triple pattern, the results of all the `GetItem` look-ups must be intersected, to ensure that all the constants of a triple pattern will be found in the same dataset. The union of all URI sets thus obtained from the triple patterns of a SPARQL query provides the URIs of the data sets to retrieve from S3 and from the merge of which the query must be answered.

Using our running example, assume that we want to evaluate the SPARQL query of Figure 2.4. The corresponding DynamoDB queries required in order to retrieve the corresponding datasets are the following:

```
r1: GetItem(T, :hasAuthor)
r2: GetItem(T, :member)
r3: GetItem(T, :lab1)
```

The datasets retrieved from the DynamoDB request `r1` will be merged with those obtained by intersecting the results of `r2` and `r3`. The query will be then evaluated on the resulting (merged) graph to get the correct answers.

Analytical cost model. We assume that each RDF term appears only once in a dataset (worst case scenario) and thus, the number of RDF terms equals to three times the number of triples. For each RDF term in a dataset we create 1 entry in DynamoDB. Then, the number of items in the index for this strategy is $\sum_{ds \in \mathcal{D}} 3 \times |ds|$.

For query processing, the number of constants a query can have is at most $3 \times |q|$, i.e. a boolean query. Using this strategy, one lookup per constant in the query is performed to the index and thus, the number of lookups to DynamoDB is $3 \times |q|$.

2.4.3.2 Attribute-based strategy

The next indexing strategy, denoted ATT, uses each attribute present in an RDF triple and indexes it in a different table depending on whether it is subject, predicate or object.

Indexing. Let *element* denote any among the subject, predicate and object value of an RDF triple. For each triple of a dataset and for each element of the triple, one DynamoDB item is created. The key of the item is named after the element value. As DynamoDB attribute name, we use the URI of the dataset containing a triple with this value; as DynamoDB attribute value, we use ϵ. This index distinguishes between the appearances of an URI in the subject, predicate or object of a triple: one DynamoDB table is created for subject-based indexing, one for predicate- and one for value-based indexing. Using our notation we therefore have the following indexes: $(\underline{S}|S|U|\epsilon)$, $(\underline{P}|P|U|\epsilon)$, and $(\underline{O}|O|U|\epsilon)$. In this way, false positives can be avoided (e.g., datasets that contain a certain URI but not in the position that this URI appears in the query will not be retrieved).

$\underline{S}$ table	
item key	(attr. name, attr. value)
s_1	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ), ...
s_2	(URI_{ds_2}, ϵ), ...
$\underline{P}$ table	
item key	(attr. name, attr. value)
p_1	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ), ...
p_2	(URI_{ds_2}, ϵ), ...
$\underline{O}$ table	
item key	(attr. name, attr. value)
o_1	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ), ...
o_2	(URI_{ds_2}, ϵ), ...

(a) ATT indexing.

$\underline{S}$ table	
item key	(attr. name, attr. value)
:publisher1	(*publications*, ϵ)
:book1	(*publications*, ϵ)
:article1	(*publications*, ϵ)
:bar	(*authors*, ϵ)
:foo	(*authors*, ϵ)
_:uid1	(*authors*, ϵ)
:lab1	(*labs*, ϵ)
:location	(*labs*, ϵ)
$\underline{P}$ table	
item key	(attr. name, attr. value)
:hasAuthor	(*publications*, ϵ)
:hasPublished	(*publications*, ϵ)
:hasNationality	(*authors*, ϵ)
:hasName	(*authors*, ϵ), (*labs*, ϵ)
...	
$\underline{O}$ table	
item key	(attr. name, attr. value)
:book1	(*publications*, ϵ)
:bar	(*publications*, ϵ)
"Databases"	(*publications*, ϵ)
"French"	(*authors*, ϵ)
:location	(*labs*, ϵ)
...	

(b) Sample ATT index entries.

TABLE 2.5: ATT strategy.

Querying. For each RDF term (URI or literal) of a BGP query, a DynamoDB `GetItem` look-up is submitted to the $\underline{S}$, $\underline{P}$, or $\underline{O}$ table of the ATT index, depending on the position of the constant in the query. Each such look-up retrieves the URIs of the datasets which contain a triple with the given term in the respective position. For each triple pattern, the results of all the `GetItem` look-ups based on constants of that triple need to be intersected. This ensures that all the constants of a triple pattern will be located at the same dataset. The union of all URI sets thus obtained from the triple patterns of a SPARQL query provides the URIs of the data sets to retrieve from S3 and from the merge of which the query must be answered.

Using our running example assume that we want to evaluate the SPARQL query of Figure 2.4. The corresponding DynamoDB queries that are required in order to retrieve the corresponding datasets are the following:

```
r1: GetItem(S, :hasAuthor)
r2: GetItem(P, :member)
r3: GetItem(O, :lab1)
```

The dataset URIs retrieved from DynamoDB request `r1` will be merged with the datasets resulting from the intersection of those retrieved from the requests `r2` and `r3`. The query will be then evaluated on the resulting (merged) graph to get the correct answers.

Analytical cost model. We assume that the number of distinct subjects,

predicates and objects values appearing in a dataset is equal to the size of the dataset itself, and thus equal to the number of triples (worst case scenario). For each triple in a dataset we create three entries in DynamoDB. Thus, the size of the index for this strategy will be $\sum_{ds \in \mathcal{D}} 3 \times |ds|$.

Given a SPARQL query q, one lookup per constant in a request is performed to the appropriate table. Thus, the number of lookups to DynamoDB is $3 \times |q|$.

2.4.3.3 Attribute-subset strategy

The following strategy, denoted ATS, is also based on the RDF terms occurring in the datasets, but records more informations on how terms are combined within these triples.

Indexing. This strategy encodes each triple `(s, p, o)` by a set of seven patterns `s`, `p`, `o`, `sp`, `po`, `so` and `spo`, corresponding to all non-empty attribute subsets. For each of these seven patterns a new DynamoDB table is created. For each triple seven new items are created and inserted into the corresponding table. 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: $(\underline{S}|S|U|\epsilon)$, $(\underline{P}|P|U|\epsilon)$, $(\underline{O}|O|U|\epsilon)$, $(\underline{SP}|SP|U|\epsilon)$, $(\underline{PO}|PO|U|\epsilon)$, $(\underline{SO}|SO|U|\epsilon)$ and $(\underline{SPO}|SPO|U|\epsilon)$.

A general outline of this strategy is shown in Table 2.6(a) and the data from our running example leads to the index configuration outlined in Table 2.6(b).

Querying. For each triple pattern of a BGP query the corresponding `GetItem` call is sent to the appropriate table depending on the position of the bound values of the triple pattern. The item key is a concatenation of the bound values of the triple pattern. The URIs obtained through all the `GetItem` calls identify the datasets on which the query must be evaluated.

For example, for the SPARQL query of Figure 2.4 we need to perform the following DynamoDB API calls:

```
r1: GetItem(P, :hasAuthor)
r2: GetItem(PO, :member‖:lab1)
```

We then evaluate the SPARQL query on the RDF merge of the retrieved datasets.

Analytical cost model. For each triple in $\mathcal{D}$, we create at most seven entries in DynamoDB. Thus, the size of the index for this strategy is $\sum_{ds \in \mathcal{D}} 7 \times |ds|$.

To answer a query q, we perform one lookup for each triple pattern appearing in the SPARQL query. Thus, $|q|$ is the number of DynamoDb requests an attribute-subset strategy performs.

$\underline{S}$ table	
item key	(attr. name, attr. value)
s_1	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ), ...
s_2	(URI_{ds_2}, ϵ), ...
$\underline{P}$ table	
item key	(attr. name, attr. value)
p_1	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ), ...
p_2	(URI_{ds_2}, ϵ), ...
$\underline{O}$ table	
item key	(attr. name, attr. value)
o_1	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ), ...
o_2	(URI_{ds_2}, ϵ), ...
$\underline{SP}$ table	
item key	(attr. name, attr. value)
$s_1 \| p_1$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),...
$s_1 \| p_2$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ), ...
$s_2 \| p_1$	(URI_{ds_2}, ϵ), ...
$\underline{PO}$ table	
item key	(attr. name, attr. value)
$p_1 \| o_1$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),...
$p_1 \| o_2$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ), ...
$p_2 \| o_1$	(URI_{ds_2}, ϵ), ...
$\underline{SO}$ table	
item key	(attr. name, attr. value)
$s_1 \| o_1$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),...
$s_1 \| o_2$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ), ...
$s_2 \| o_3$	(URI_{ds_2}, ϵ), ...
$\underline{SPO}$ table	
item key	(attr. name, attr. value)
$s_1 \| p_1 \| o_1$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ),...
$s_1 \| p_2 \| o_2$	(URI_{ds_1}, ϵ), (URI_{ds_2}, ϵ), ...
$s_2 \| p_1 \| o_3$	(URI_{ds_2}, ϵ), ...

(a) ATS indexing.

$\underline{S}$ table	
item key	(attr. name, value)
:publisher1	(publications, ϵ)
:book1	(publications, ϵ)
:article1	(publications, ϵ)
...	...
$\underline{P}$ table	
item key	(attr. name, value)
:hasPublished	(publications, ϵ)
:hasAuthor	(publications, ϵ)
:member	(authors, ϵ)
:hasName	(publications, ϵ), (labs, ϵ)
...	...
$\underline{O}$ table	
item key	(attr. name, value)
:book	(publications, ϵ)
:lab1	(authors, ϵ)
"ResearchLab"	(labs, ϵ)
...	...
$\underline{SP}$ table	
item key	(attr. name, value)
:publisher1‖:hasPublished	(publications, ϵ)
:bar‖:lab1	(authors, ϵ)
...	...
$\underline{PO}$ table	
item key	(attr. name, value)
:hasPublished‖:book1	(publications, ϵ)
:hasAuthor‖:bar	(publications,ϵ)
...	...
$\underline{SO}$ table	
item key	(attr. name, value)
inria:publisher‖inria:book1	(publications, ϵ)
inria:article1‖"Databases"	(publications, ϵ)
...	...
$\underline{SPO}$ table	
item key	(attr. name, value)
:article1‖:field‖"Databases"	(publications, ϵ)
...	...

(b) Sample ATS index entries.

TABLE 2.6: ATS strategy.

2.5 Implementation Details

The majority of RDF terms used are URIs which consist of long strings of text. Since working with long strings is expensive in general, mapping dictionaries have been used in many centralized RDF stores such as [22]. In these works, RDF terms are mapped to numerical values and then, triple storage and query evaluation is performed using these numerical values. The final answers of the query evaluation is decoded again to the original RDF terms.

We adopt a similar mapping dictionary for the QAS strategy. In particular, we use a hash function to map RDF terms to binary values. At query runtime, we need to decode the answers of the query. Thus, a *dictionary table* which

holds the reverse mapping is required, i.e., from the binary values to the original RDF terms. The dictionary table is stored in DynamoDB and contains the binary values in the item keys and their corresponding representation as the attribute values. After query evaluation has finished and our answer is in binary form, we perform the appropriate `GetItem` requests to the dictionary table to decode the results.

We also use a hash function to store the key items for the RTS, ATT and ATS indexes. For these strategies, only the encoding part is required since the actual answer to the queries is extracted from the documents stored in S3. In this way we avoid storing arbitrary long URIs as keys in the indexes and use smaller values (16 bytes), reducing the space occupied by the index.

Finally, note that using a hashing procedure enables us to encode RDF terms to binary values from different machines without any node coordination. This is because of the deterministic nature of hash functions which always generate the same hash value for the same given input. On the other hand, hashing can lead to collisions, i.e., two different inputs can be mapped to the same hash value. In the RTS, ATT and ATS strategies such a collision would only affect the number of datasets that need to be retrieved from S3 (false positives) and not the answers of the query. But even in the QAS strategy we can minimize the probability of a collision by choosing an appropriate hash function. For instance, for a 128-bit hash function, such as MD5, and a number of different elements 2.6×10^{10}, the probability of a collision is 10^{-18}! [8].

2.6 Experimental Evaluation

The proposed architecture and algorithms we presented in Section 2.4 have been fully implemented in our system AMADA[9] [6]. In this section we present an experimental evaluation of our strategies and techniques.

2.6.1 Experimental setup and datasets

Our experiments were run in the AWS Ireland region in February 2013. For the local SPARQL query evaluation needed by strategies RTS, ATT, and ATS we have used RDF-3Xv0.3.7[10] [22], a widely known RDF research database, to process incoming queries on the datasets identified by our index look-ups. Thus, RDF-3X was deployed on EC2 machines in order to process queries on a (hopefully tight) subset of the dataset, as identified by each respective strategy. For the QAS strategy, when the queries are processed directly on the DynamoDB data (thus, no data is loaded in an RDF database),

[9] `https://team.inria.fr/oak/amada/`
[10] `http://code.google.com/p/rdf3x/`

```
select ?x
where {
?x yago:hasWonPrize ?y .
}
```

FIGURE 2.5: Experiments query Q3 with low selectivity.

we relied on the physical relational algebraic select, project and join operators of our ViP2P project [20].

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

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

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

As a dataset we have used subsets of YAGO[11] and DBpedia[12] RDF dumps. The subsets we used consist of approximately 35 million triples in total (5 GB in NTRIPLE syntax).

We have hand-picked 9 queries over these two datasets with different characteristics. Queries are Figures 2.5, 2.6, 2.7 show in detail three of these queries; the other are similar. The number of triple patterns each query contains ranges from 1 to 5, which is a number used more often in real-life SPARQL queries [7]. The characteristics of the queries we use are shown in Table 2.7, where *struct* indicates the structure of each query (*simple* for single triple pattern queries, *star* for star-shaped join queries and *mix* for complex queries combining both star and path joins), $\#tp$ is the number of triple patterns, $\#c$ is the number of constant values each query contains, and $\#results$ is the number of triples each query returns. Furthermore we present the number of distinct datasets $\#d$ which will be used from each strategy to answer the query.

2.6.2 Indexing time and costs

In this section we study the performance of our four RDF indexing strategies. The RDF datasets are initially stored in S3, from which they are gathered in batches by 4 L instances running the indexing module. We batched the datasets in order to minimize the number of calls needed to load the index into

[11] http://www.mpi-inf.mpg.de/yago-naga/yago/
[12] http://dbpedia.org/

```
select ?x ?z ?w
where {
?x rdf:type yago:wordnet_scientist_110560637 .
?x yago:diedOnDate ?w .
?x yago:wasBornOnDate ?z .
}
```

FIGURE 2.6: Experiments query Q5 with medium selectivity.

```
SELECT ?name1 ?name2
WHERE {
   ?p1 yago:isMarriedTo ?p2 .
   ?p2 yago:hasGivenName ?name2 .
   ?p1 yago:hasGivenName ?name1.
   ?p2 yago:wasBornIn ?city .
   ?p1 yago:wasBornIn ?city .
}
```

FIGURE 2.7: Experiments query Q9 with high selectivity.

DynamoDB. Moreover, we used L instances because we found out that DynamoDB is the bottleneck while indexing. We should also note that we used a *total* throughput capacity in our DynamoDB tables of $10,000$ write units. This means that if a strategy required more than one table we divided the throughput among all tables.

We measure the indexing time and monetary costs of building the indexes in DynamoDB. For the strategies RTS, ATTand ATS we show results only with the dictionary on, as there is always a benefit from it. For the QAS strategy we show results both with (QAS_on) and without (QAS_off) the dictionary as the difference between the two leads to some interesting observations.

In Figure 2.8 we demonstrate for each strategy the time required to create the indexes, their size and their indexing cost. Note that to add the items into DynamoDB we used the `BatchWriteItem` operation which can insert up to 25 items at a time in a table. We observe from the blue bars of the left graph of Figure 2.8 that the ATS index is the most time-consuming, since for each triple it inserts seven items into DynamoDB. The same holds for the size of the index, as the ATS occupies about 11 GB. In contrast, the RTS index which inserts only one item for each RDF term is more time-efficient. An interesting observation is that the QAS_off indexing requires significantly less time than when the dictionary is used. This is because inserting items in the dictionary table for each batch becomes a bottleneck. Also, the size of the QAS index with the dictionary is only slightly smaller than when the dictionary is not used, i.e., 9 GB in QAS_on vs. 10.6 GB in QAS_off. This is because of the datasets used in the experiments where URIs are not repeated many times

Query	struct	#tp	#c	#results	#d by RTS	#d by ATT	#d by ATS
Q1	*simple*	1	2	1	2	2	1
Q2	*simple*	1	2	433	3	3	3
Q3	*simple*	1	1	72829	2	2	2
Q4	*star*	2	4	1	19	19	19
Q5	*star*	3	4	2895	26	25	25
Q6	*star*	3	3	50686	34	34	34
Q7	*star*	4	4	42785	39	39	39
Q8	*mix*	5	6	2	9	9	9
Q9	*mix*	5	5	12	5	5	5

TABLE 2.7: Query characteristics.

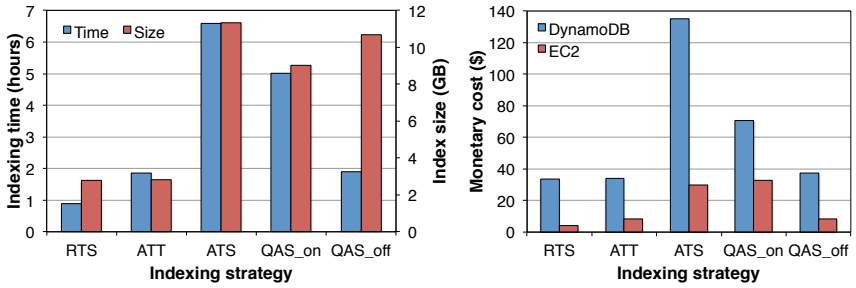

FIGURE 2.8: Indexing time and size (left) and cost (right).

across the datasets and thus, the storage space gain is not exemplary. Also note that the size of the index also affects the money spent for keeping the index. For example, the QAS_on index would cost about 10$ per month, while the QAS_off would cost an extra 2$ per month. On the other hand, the RTS or ATS indexes are more economical and would only cost acout 3$ per month.

In the right graph of Figure 2.8, we show the monetary cost of DynamoDB and the EC2 usage when creating the index. Again, the ATS index is the most expensive one both in DynamoDB and EC2. Moreover, we observe that the QAS_on is more expensive than QAS_off due to the increased number of items that we insert in the index when using the dictionary. The costs of S3 and SQS are constant for all strategies (0.0022$ and 0.0004$, respectively) and negligible compared to the costs of DynamoDB and EC2 usage. We thus omit them from the graph.

2.6.3 Querying time and costs

For this set of experiments, we use the data and indexes we have created in the previous experiment (see Section 2.6.2) and measure the query response times and monetary costs of our queries. We ran one query after the other sequentially using only one XL machine.

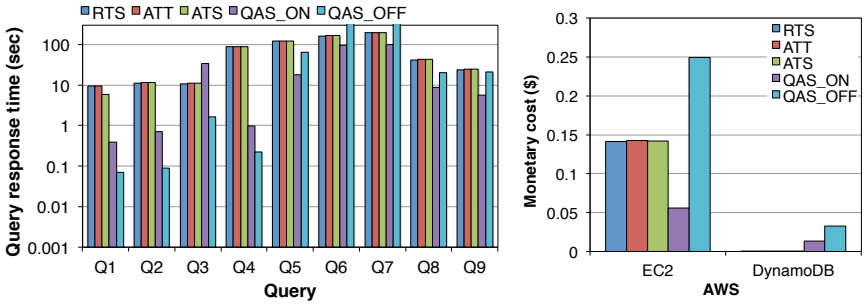

FIGURE 2.9: Querying response time (left) and cost (right).

Figure 2.9 presents the response times of each query in each strategy and the total monetary cost for the whole query workload in each strategy regarding EC2 and DynamoDB usage. We observe that for the datasets oriented strategies, i.e., RTS, ATS, and ATT, the queries accessing a small number of dataset (less than 10) are very efficient and are executed in less than 50 seconds. As the number of dataset increases (Q4-Q9) so does the response time for these strategies. This is expected since the retrieved dataset have to be loaded in RDF-3X in order to answer the query; as this number increases, RDF-3X loading time also goes up. Out of these three strategies we cannot pick a winner since all strategies retrieve almost the same dataset from DynamoDB. The only cases where we had a false positive, i.e., datasets not contributing to the query result, are RTS and ATT for query Q1 and RTS for query Q5 (see Table 2.7). We believe this may often be the case in practice when a triple pattern has two constants: intersecting the respective two sets of dataset URIs will not leave many false positives.

The strategies relying solely in DynamoDB to answer the queries (QAS_on and QAS_off) are better for highly selective queries than those relying on RDF-3X. Especially the one using the dictionary encoding is good even for not very selective queries like Q6 and Q7. On the other hand, answering queries with low selectivity without a dictionary through DynamoDB seems a bad idea due to the large number of items requested from DynamoDB and the large number of intermediate results that are loaded in memory. An interesting exception is Q3, for which the dictionary did not improve the performance. Note that the dictionary encoding invokes a big overhead for decoding the final results (transforming them from compact identifiers to their actual URI values), and especially if the number of returned results is large. If there are no joins in a query, as it is in the case of Q3, there is no profit from the dictionary encoding, and thus, decoding the large number of returned results is a big overhead.

In terms of monetary cost shown at right of Figure 2.9 we observe that the most expensive strategy regarding both EC2 and DynamoDB is QAS_off. For EC2, this can be easily explained by considering the query response times for this strategy and having in mind that queries Q6 and Q7 required more than 300 seconds to be evaluated, overwhelming the CPU for a large period

of time. Regarding DynamoDB, the strategy is also expensive since the size of the items that need to be retrieved is significantly larger than for other strategies, which return only dataset names or compact encodings in the case of QAS_on. As anticipated, strategies RTS, ATS and ATT have almost the same EC2 costs, explained by their similar query response times.

2.6.4 Scalability

In this section we measure the total time for a workload of 27 queries (a mix of the queries whose characteristics appeared in Table 2.7) as the number of EC2 machines increases (scale-out) for strategies RTS and QAS_on. ATT and ATS present similar behavior with RTS and thus they are omitted from this experiment. In addition QAS_on is always better than QAS_off so we chose to drop it from the graphs. The experiments were executed using XL machines varying their number from 1 to 8 and keeping the threads number (4) equal to the number of cores of each machine (allowing a concurrent execution of 4 queries per machine).

In Figure 2.10 we demonstrate how increasing the EC2 machines can affect the total response time for executing the whole query workload. The query response time follows a logarithmic equation where in the beginning and until reaching 4 EC2 instances the time is constantly dropping until reaching a threshold where we cannot run faster due to the fact that all queries are distributed among machines and run in parallel. For example for our workload of 27 queries, using 8 machines will result into running 3 queries on each machine and due to the number of threads all queries will run in parallel and the total time will be equal with the less efficient query. Both strategies scale well with QAS_on being slightly worse due the large number of concurrent requests in DynamoDB.

Scaling-out the machines for DynamoDB is not feasible in the Amazon cloud. In general, similar services from AWS are usually offered as black boxes and the user does not have control over them other than specifying some performance characteristics, such as the throughput in DynamoDB discussed in Section 2.3.1.2. Finally, we have also experimented with scaling the size of the data in [10] and observed that the time for building the index scales linearly with the number of triples in the datasets, as it is also evident from our analytical cost model, so we omit it from this experimental evaluation.

2.6.5 Experiments conclusion

Summing up, our baseline strategy RTS is the best, providing a good trade-off between indexing time, index size, query efficiency and overall monetary cost both for building the indexes and answering queries as well. Targeting query efficiency, QAS_on is the best strategy being 50% more expensive than RTS. In addition the size of the index for QAS_on is five times bigger in comparison with RTS, making the strategy highly expensive for large periods

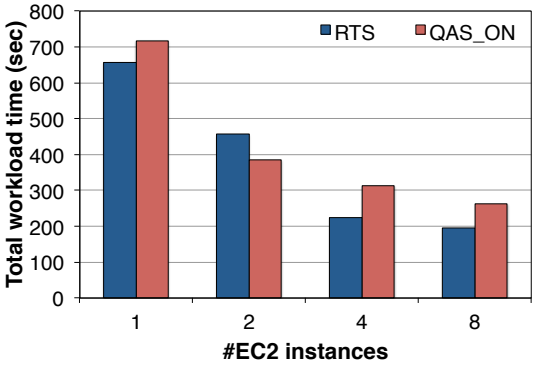

FIGURE 2.10: Total time for workload of #27 queries.

of usage. Among the discussed strategies ATS can be considered as one of the worst since it is the most costly in terms of money and index size, whereas from the efficiency perspective the indexing time is huge and the query response time does not defer significantly from the other strategies (RTS and ATT) relying on RDF-3X to answer the queries.

2.7 Summary and Perspectives

This chapter described an architecture for storing and querying RDF data using off-the-shelf cloud services, in particular the AMADA platform we have developed and demonstrated recently [6, 10]. The starting point of the present work is [10], however in this chapter we have presented a different set of strategies and accordingly new experiments, at a much larger scale than we had previously described in [10]. A brief classification of the state-of-the art in this area according to three main dimensions (data storage, query processing and reasoning) is included, while further detail can be found in our tutorial [19].

Within AMADA, we devised indexing techniques for identifying a tight superset of the RDF datasets which may contain results for a specific query, and we have proposed a technique for answering a SPARQL query from the index itself. We presented analytical cost models for each strategy and evaluated their indexing and querying performance and monetary costs.

A direction we have not considered in this work is the parallelization of the task of evaluating a single query on a large RDF dataset. This is obviously interesting, especially for non-selective queries, since the parallel processing capabilities of a cloud may lead to shorter response times. Algorithms and techniques proposed for other distributed architectures such as P2P-based or federated ones may also be applicable (see Chapters **??**).

At the same time, when considering RDF data, a first significant obsta-

cle consists of the difficulty of finding a way to partition the data, in order to enable different processors to work each partition in parallel. RDF graph partitioning algorithms for this setting are presented in works such as [16, 17] and is also discussed in Chapter **??**.

Finally, a full solution for a cloud-based large RDF store must include an intelligent pricing model, reflecting the usage of cloud resources. In this work we have outlined the monetary costs of the index, which are a first ingredient of a comprehensive pricing scheme. Working in this direction, the ultimate goal would be to formalize a cost model of different indexing and query answering strategies that expresses the trade-off between their efficiency/performance and associated monetary costs.

Bibliography

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

[2] Apache Accumulo. `http://accumulo.apache.org/`.

[3] Apache Cassandra. `http://cassandra.apache.org/`.

[4] Apache HBase. `http://hbase.apache.org/`.

[5] Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. MapReduce and PACT - Comparing Data Parallel Programming Models. In *BTW*, 2011.

[6] Andrés Aranda-Andújar, Francesca Bugiotti, Jesús Camacho-Rodríguez, Dario Colazzo, François Goasdoué, Zoi Kaoudi, and Ioana Manolescu. Amada: Web Data Repositories in the Amazon Cloud (demo). In *ACM CIKM*, 2012.

[7] M. Arias, J.D. Fernández, M.A. Martínez-Prieto, and P. de la Fuente. An Empirical Study of Real-World SPARQL Queries. In *USEWOD2011 Workshop (in conjunction with WWW)*, 2011.

[8] Mihir Bellare and Tadayoshi Kohno. Hash function balance and its impact on birthday attacks. In *Advances in Cryptology – EUROCRYPT '04, Lecture Notes in Computer Science*, pages 401–418. Springer-Verlag, 2004.

[9] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. Building a Database on S3. In *SIGMOD*, 2008.

[10] Francesca Bugiotti, François Goasdoué, Zoi Kaoudi, and Ioana Manolescu. RDF Data Management in the Amazon Cloud. In *DanaC Workshop (in conjunction with EDBT)*, 2012.

[11] Jesús Camacho-Rodríguez, Dario Colazzo, and Ioana Manolescu. Web Data Indexing in the Cloud: Efficiency and Cost Reductions. In *EDBT*, 2013.

[12] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, May 2011.

[13] Tyson Condie, Neil Conway, Peter Alvaro, and Joseph M. Hellerstein. Mapreduce online. In *NSDI*, 2010.

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

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *IN PROC. SOSP*, pages 205–220, 2007.

[16] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL Querying of Large RDF Graphs. *PVLDB*, 4(11):1123–1134, 2011.

[17] Mohammad Husain, James McGlothlin, Mohammad M. Masud, Latifur Khan, and Bhavani M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Trans. on Knowl. and Data Eng.*, 2011.

[18] Verena Kantere, Debabrata Dash, Georgios Gratsias, and Anastasia Ailamaki. Predicting Cost Amortization for Query Services. In *SIGMOD*, 2011.

[19] Zoi Kaoudi and Ioana Manolescu. Triples in the clouds (tutorial). In *ICDE*, 2013.

[20] Konstantinos Karanasos, Asterios Katsifodimos, Ioana Manolescu, and Spyros Zoupanos. The ViP2P Platform: XML Views in P2P. In *ICWE*, 2012.

[21] Gunter Ladwig and Andreas Harth. CumulusRDF: Linked Data Management on Nested Key-Value Stores. In *SSWS*, 2011.

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

[23] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. H2RDF: adaptive query processing on RDF data in the cloud. In *21st international conference on World Wide Web (demo)*, 2012.

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

[25] Roshan Punnoose, Adina Crainiceanu, and David Rapp. Rya: A Scalable RDF Triple Store for the Clouds. In *1st International Workshop on Cloud Intelligence (in conjunction with VLDB)*, 2012.

[26] Padmashree Ravindra, HyeongSik Kim, and Kemafor Anyanwu. An Intermediate Algebra for Optimizing RDF Graph Pattern Matching on MapReduce. In *ESWC*, pages 46–61, 2011.

[27] Kurt Rohloff and Richard E. Schantz. High-Performance, Massively Scalable Distributed Systems using the MapReduce Software Framework: the SHARD Triple-Store. In *Programming Support Innovations for Emerging Distributed Applications*, 2010.

[28] Alexander Schätzle, Martin Przyjaciel-Zablocki, Christopher Dorner, Thomas Hornung, and Georg Lausen. Cascading Map-Side Joins over HBase for Scalable Join Processing. In *SSWS+HPCSW*, 2012.

[29] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD*, 2013.

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

[31] Jacopo Urbani, Spyros Kotoulas, Eyal Oren, and Frank van Harmelen. Scalable Distributed Reasoning using MapReduce. In *Proceedings of the 8th International Semantic Web Conference (ISWC2009)*, October 2009.

[32] Jacopo Urbani, Frank van Harmelen, Stefan Schlobach, and Henri Bal. QueryPIE: Backward Reasoning for OWL Horst over Very Large Knowledge Bases. In *Proceedings of the 10th International Semantic Web Conference (ISWC 2011)*, Bonn, Germany, 2011.

[33] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. A Distributed Graph Engine for Web Scale RDF Data. In *PVLDB 2013*.